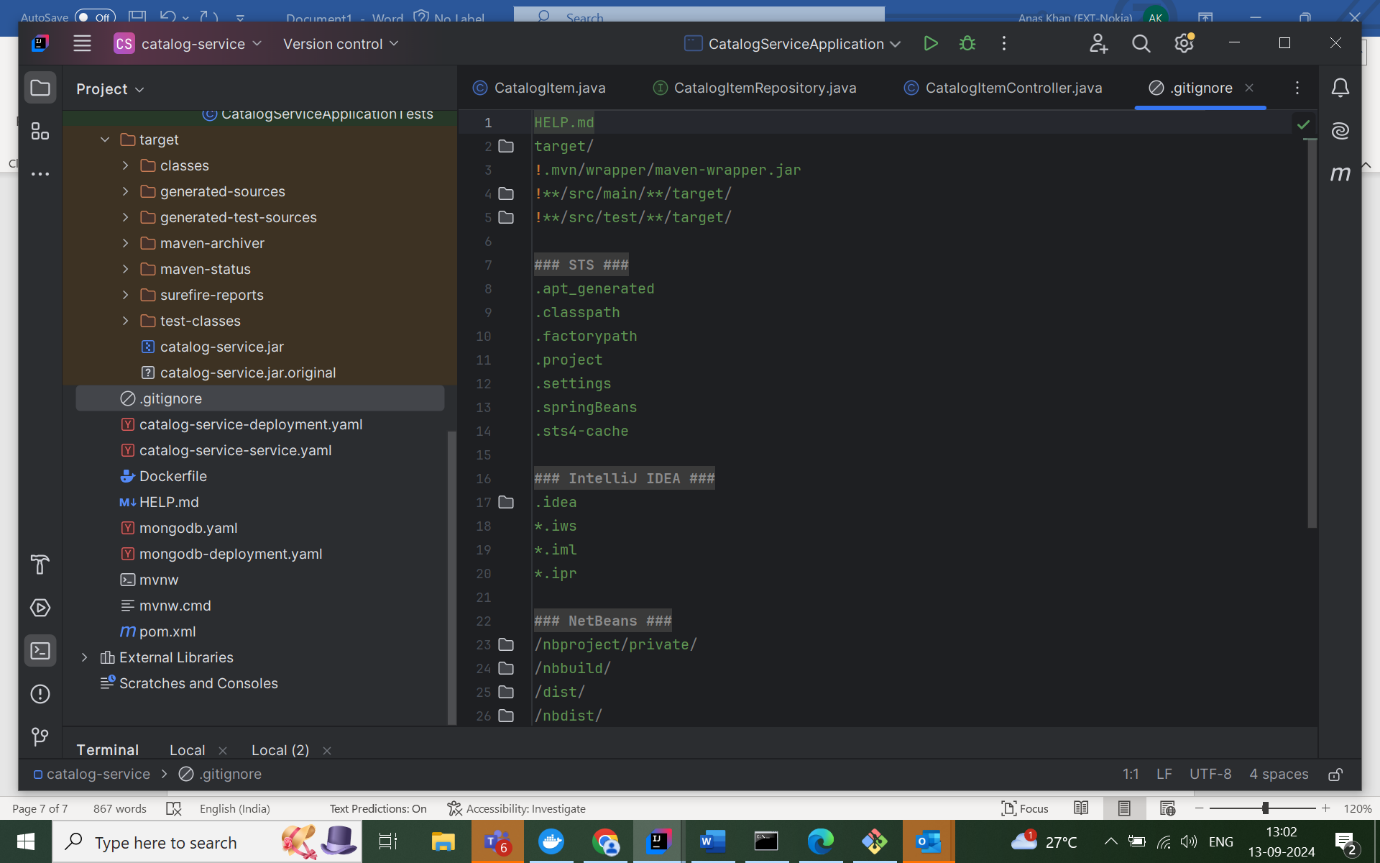
**Flow One R&D Engineer**

**Assessment**

A computer screen shot of a black screen

Description automatically generated



*  a new directory for project.

Initialize a Git repository

git init

* Created a .gitignore file to exclude files .

**Directory Structure:**

telecommunication-system/

├── catalog-service/

├── order-management-service/

├── inventory-management-service/

├── ui/

├── k8s/

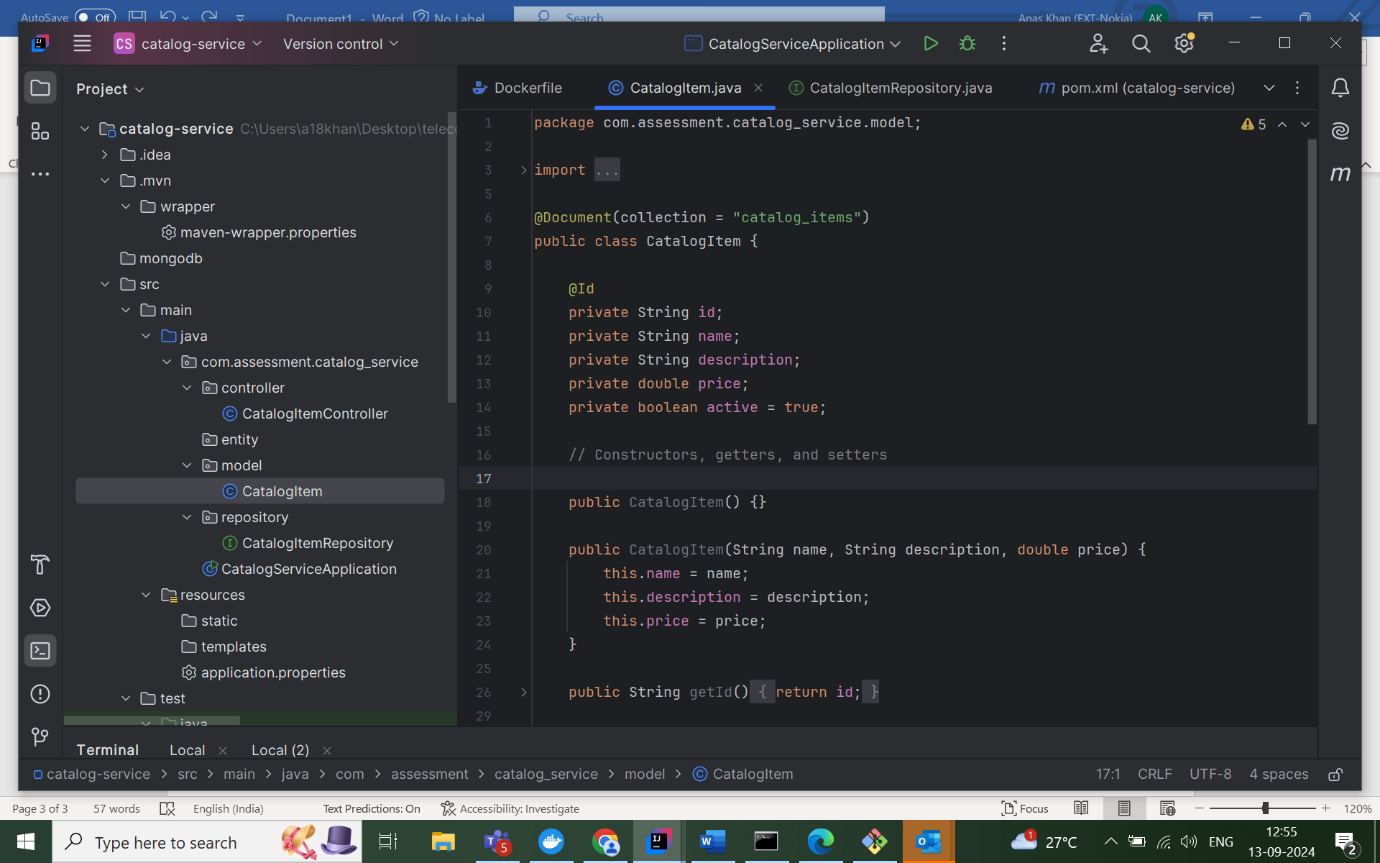
└── .gitignore

**Commit Initial Structure**

Add files to Git and commit:

git add .

git commit -m "Initial project structure"

**Class: CatalogItem**

**Description:**

The CatalogItem class represents an item in the catalog for the catalog service. It is annotated with @Document to indicate that it maps to a MongoDB document, specifically the catalog\_items collection. This class contains fields for storing information about the catalog item, such as:

* **id**: A unique identifier for the item, annotated with @Id, which maps it to the MongoDB document ID.
* **name**: The name of the catalog item.
* **description**: A brief description of the item.
* **price**: The price of the item, represented as a double.
* **active**: A boolean flag indicating whether the item is active or not. By default, the value is set to true.

**Constructors:**

* **Default constructor**: Provides a no-argument constructor for creating empty instances.
* **Parameterized constructor**: Takes name, description, and price as arguments to initialize an item with these values.

**Getters and Setters:**

Standard getters and setters are provided to allow access and modification of the fields. These include:

* getId, setId: For accessing and setting the ID.
* getName, setName: For accessing and setting the name.
* getDescription, setDescription: For accessing and setting the description.
* getPrice, setPrice: For accessing and setting the price.
* isActive, setActive: For checking and modifying whether the item is active.

**Usage:**

This class can be used to model catalog items in an e-commerce or service-based application where items have attributes like name, description, price, and an active status. The items will be persisted in a MongoDB database.

A screenshot of a computer

Description automatically generated

**Interface: CatalogItem Repository**

**Description:**

The CatalogItemRepository interface extends the MongoRepository interface provided by Spring Data MongoDB. It is responsible for performing CRUD operations and custom queries on the CatalogItem documents in the MongoDB database.

**Key Features:**

* **MongoRepository Inheritance**: The repository extends MongoRepository<CatalogItem, String>, where CatalogItem is the entity type, and String is the type of its primary key (id). This gives it access to various predefined methods, such as:
  + save(CatalogItem item): Saves or updates a CatalogItem document.
  + findById(String id): Finds a CatalogItem by its unique id.
  + findAll(): Retrieves all CatalogItem records.
  + deleteById(String id): Deletes a CatalogItem by its id.
* **Custom Query Method**:
  + List<CatalogItem> findByNameContaining(String name): This is a derived query method that finds and returns all catalog items whose name contains the specified string. Spring Data MongoDB will automatically generate the query for this method based on its name.

**Usage:**

This repository is used to interact with the CatalogItem collection in the MongoDB database. By extending MongoRepository, it allows for easy integration with MongoDB and eliminates the need to write boilerplate CRUD operations manually. The custom method findByNameContaining can be used for searching catalog items based on partial matches to their name, which is useful for implementing search functionality in the application.

A screenshot of a computer

Description automatically generated

**Class: CatalogItem Controller**

**Description:**

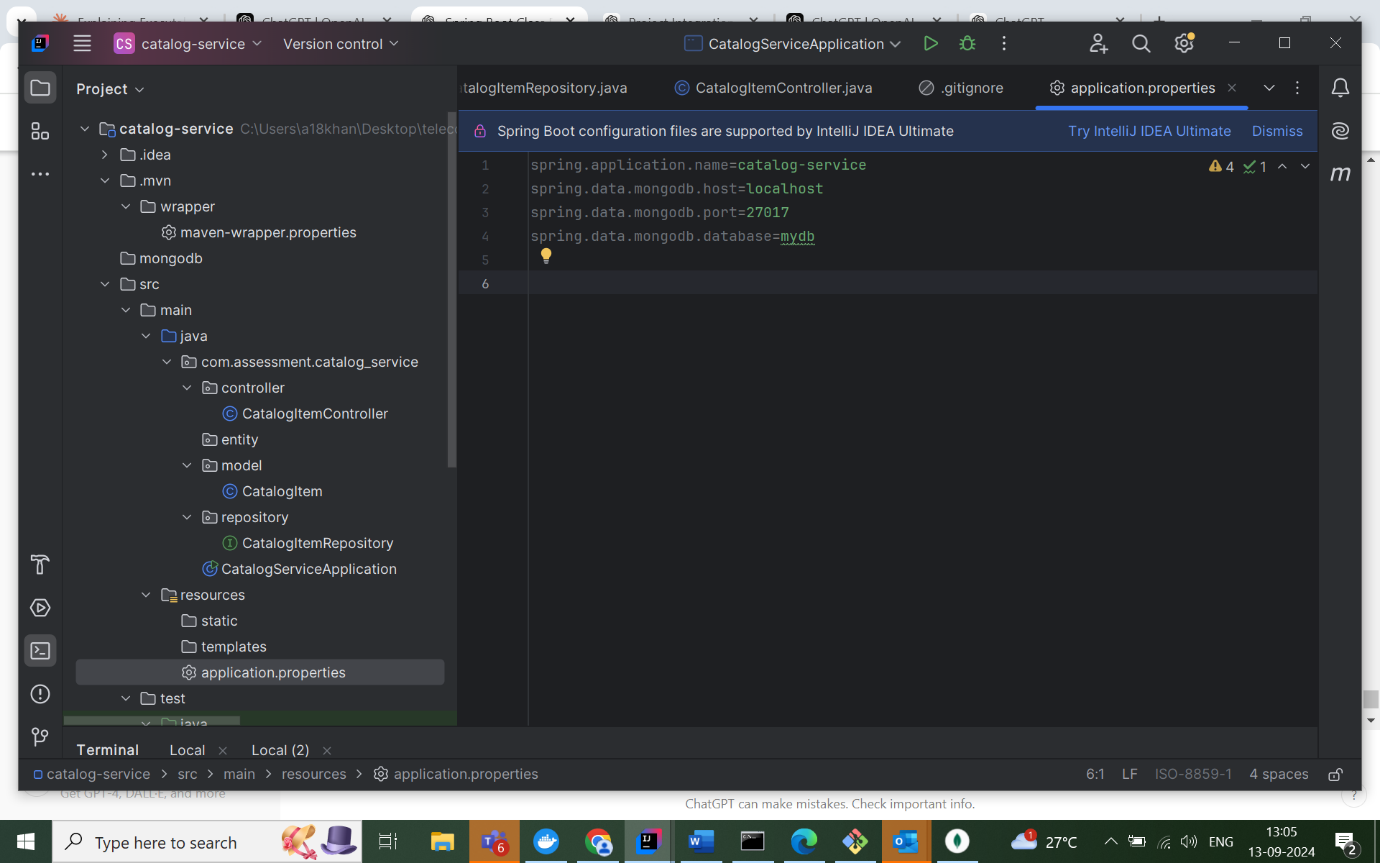
The CatalogItemController is a REST controller in the catalog service responsible for handling HTTP requests related to catalog items. It is annotated with @RestController and @RequestMapping("/api/catalog"), meaning it exposes API endpoints under the /api/catalog URL path. The controller interacts with the CatalogItemRepository to perform CRUD operations and some additional functionalities.

**Key Features:**

1. **Autowired Repository**:
   * The CatalogItemRepository is injected into the controller using the @Autowired annotation. This allows the controller to interact with the MongoDB database to perform operations on CatalogItem documents.
2. **Endpoints**:
   * **GET /api/catalog/items**:
     + Retrieves all catalog items from the database.
     + Returns a list of CatalogItem objects.
   * **GET /api/catalog/items/{id}**:
     + Retrieves a catalog item by its id.
     + Returns 200 OK with the CatalogItem if found, or 404 Not Found if the item doesn't exist.
   * **POST /api/catalog/items**:
     + Creates a new catalog item.
     + Expects a JSON object in the request body representing the new item.
     + Saves and returns the newly created CatalogItem.
   * **PUT /api/catalog/items/{id}**:
     + Updates an existing catalog item by its id.
     + The updated details are provided in the request body.
     + If the item exists, it updates the fields (name, description, price) and returns the updated item.
     + If the item does not exist, it returns a 404 Not Found.
   * **DELETE /api/catalog/items/{id}**:
     + Deletes a catalog item by its id.
     + Returns 204 No Content on successful deletion.
   * **DELETE /api/catalog/items1/{id}** (Soft Delete):
     + Soft deletes an item by setting its active flag to false instead of permanently deleting it.
     + This is useful when you want to keep track of inactive items without removing them from the database.
   * **GET /api/catalog/items/search**:
     + Provides a search functionality for catalog items.
     + Supports search by name (partial match) or by price range (minPrice to maxPrice).
     + Returns a list of matching catalog items based on the provided search criteria.

**Usage:**

This controller exposes RESTful endpoints for managing catalog items in the catalog service. It handles basic CRUD operations, as well as soft deletion and item search by name or price. This makes it an essential part of the service for interacting with catalog data, particularly in web or API-based applications.



**application.properties Configuration for catalog-service**

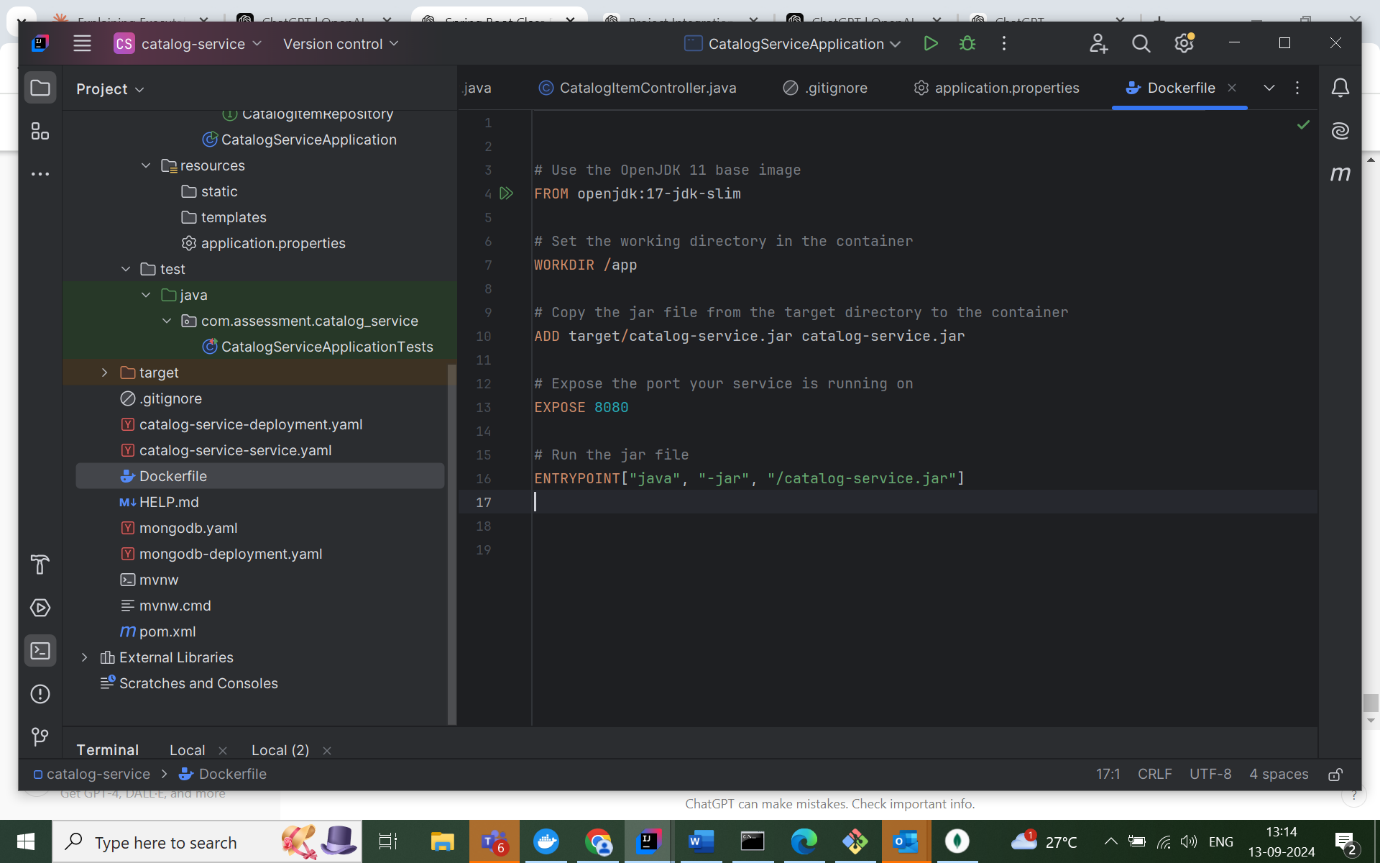
**Description:**

This is the Spring Boot **application.properties** file, which configures the catalog service application, particularly its connection to a MongoDB database. Below is an explanation of each property:

1. **spring.application.name=catalog-service**:
   * Sets the name of the Spring Boot application to catalog-service. This name will be used in logs and as part of service discovery in distributed systems like microservices.
2. **spring.data.mongodb.host=localhost**:
   * Specifies that the MongoDB instance is hosted on the local machine (localhost). If the MongoDB database is running on the same machine as the application, this is the correct setting.
3. **spring.data.mongodb.port=27017**:
   * Defines the port on which the MongoDB instance is running. By default, MongoDB runs on port **27017**.
4. **spring.data.mongodb.database=mydb**:
   * Specifies the name of the MongoDB database the Spring Boot application will connect to. In this case, the database is named **mydb**.

A screenshot of a computer

Description automatically generated



A computer screen shot of a computer

Description automatically generated

**Dockerfile**

1. **FROM openjdk:17-jdk-slim**
   * **Description**: This line specifies the base image to be used for the Docker container. In this case, it uses the **OpenJDK 17 slim version**, which is a minimalistic version of the JDK that is smaller in size and suitable for running Java applications in a lightweight environment.
2. **WORKDIR /app**
   * **Description**: This sets the working directory within the container to /app. All subsequent commands like ADD and ENTRYPOINT will be executed relative to this directory.
3. **ADD target/catalog-service.jar catalog-service.jar**
   * **Description**: This command copies the catalog-service.jar file from the target directory of the local machine into the container's /app directory. The ADD command is commonly used to copy files from the local filesystem into the container.
4. **EXPOSE 8080**
   * **Description**: This exposes port **8080** on the container, which allows network access to the Spring Boot application running inside the container. It tells Docker that the application will be accessible on this port.
5. **ENTRYPOINT["java", "-jar", "/catalog-service.jar"]**
   * **Description**: This line specifies the command to run the Spring Boot application inside the container. The ENTRYPOINT command ensures that when the container starts, it will run java -jar /catalog-service.jar to execute the Spring Boot application. This should be corrected to:

**Use Case:**

This Dockerfile is designed to package and run a Spring Boot application as a Docker container. The application runs on port 8080, and by using OpenJDK 17, it benefits from the most recent LTS (Long-Term Support) version of the Java platform. This setup is ideal for deploying a Spring Boot microservice in various environments like Kubernetes or Docker Swarm.

**Commands to deploy service on Kubernetes** 1. `docker images`

This command lists all Docker images available on your local system. It shows the repository, tag, image ID, creation time, and size of each image.

`minikube version`

This command displays the version of Minikube installed on your system.

`minikube status`

This command shows the current status of your Minikube cluster, including the state of various components like the host, kubelet, apiserver, and kubeconfig.

`minikube start --driver=docker`

This command starts a Minikube cluster using Docker as the driver. It sets up a local Kubernetes environment.

`minikube docker-env`

This command outputs the necessary environment variables to configure your shell to use Minikube's Docker daemon.

`docker build -t catalog-service:1.0 .`

This command builds a Docker image from the Dockerfile in the current directory. It tags the image as "catalog-service" with version "1.0".

`kubectl create deployment catalog-service-deployment --image=catalog-service:1.0 --port=8080`

This command creates a Kubernetes deployment named "catalog-service-deployment" using the "catalog-service:1.0" image and exposes port 8080.

. `kubectl get deployment

This command lists all the deployments in the current Kubernetes namespace, showing their status, desired replicas, current replicas, and age.

kubectl describe deployment catalog-service-deployment

This command provides detailed information about the "catalog-service-deployment" deployment, including its configuration, status, and events.

kubectl create deployment catalog-service-deployment --image=catalog-service:1.0 --port=8080

This command attempts to create a new deployment, but fails because a deployment with the same name already exists.

kubectl get pods

This command lists all pods in the current namespace, showing their status, restarts, and age.

docker build -t catalog-service:1.0 .

This command builds a new Docker image from the Dockerfile in the current directory, tagging it as "catalog-service:1.0".

kubectl get deployment

This command lists all deployments in the current namespace, showing their status and number of replicas.

docker stop $(docker ps -q)

This command attempts to stop all running Docker containers, but fails due to syntax error.

docker ps -q | ForEach-Object { docker stop $\_ }

This PowerShell command attempts to stop all running Docker containers, but fails because it's run in CMD.

for /F "tokens=\*" %i IN ('docker ps -q') DO docker stop %i

This command stops all running Docker containers.

for /F "tokens=\*" %i IN ('docker ps -a -q') DO docker rm %i

This command removes all Docker containers (including stopped ones).

docker imageskubectl get pods This shows the running pods. You have one pod springboot-k8-deployment-69c586d794-ks2cf in a running state.

kubectl logs springboot-k8-deployment-69c586d794-ks2cf

This displays the logs of the Spring Boot application running in the pod. The logs show that the application started successfully on port 8080.

kubectl get deployment

This shows the deployments.

We have one deployment named springboot-k8-deployment with 1 replica running.

kubectl expose deployment springboot-k8-deployment --type=NodePort

This exposes the deployment as a service of type NodePort, making it accessible from outside the cluster.

kubectl get service This lists the services.

You can see your springboot-k8-deployment service exposed on port 8080, mapped to a node port 30665.

minikube service springboot-k8-deployment --url

This command provides a URL to access your service. It returned http://127.0.0.1:62474, which is a proxied URL that Minikube sets up to access your service from your local machine.

The final message indicates that because you're using a Docker driver on Windows, you need to keep the terminal open for the service to be accessible.

To access Spring Boot application, we can use the URL provided by the last command (http://127.0.0.1:62474). This URL will route traffic to your application running inside the Kubernetes cluster.

A screenshot of a computer screen

Description automatically generatedA screenshot of a computer

Description automatically generated

Practical evidence for catalog, Business Service Tool (BST), and order management 

* Flow One Installation Process -:

* Install Virtual Box 6.0.14r133895

A black screen with a black background

Description automatically generated

* Install Vagrant 2.2.6

A black rectangular object with white border

Description automatically generated

* Install FO23 Zip Folder and extract in C Drive
* Connect with temper gateway with Nokia credential
* Connect with Nokia VPN
* Launch Command Prompt at the extracted file's location and verify if it's installed.
* We will open Command Prompt at the project directory location and run the command   set\_dns\_proxy.bat

A black background with white text

Description automatically generated

* Run Vagrant up command and wait for all tasks to complete. The script will install Catalog, Instantlink, and FlowOneAPI without any additional post-installation steps like HTTPS setup."

A screen shot of a computer program

Description automatically generated

* Now we will do following process

•  **Provisioning**: This refers to setting up or preparing the virtual environment (using Vagrant in this case).

•  **vagrant ssh**: After provisioning, we will use this command to enter the virtual machine (VM). It establishes a secure connection (SSH) to the VM.

•  **Password: vagrant**: Once you run the command, you'll need to enter the password (which is "vagrant" by default).

•  **sudo su catalog**: After logging in, you're switching users to the "catalog" user. "sudo" gives you administrative privileges, and "su catalog" switches the current user to "catalog" (who has the required permissions for the next steps)

A black screen with white text

Description automatically generated 

* Now we will start the service By ctl\_control start .   
     
  **ctl\_control**: This is likely a control script or command used to manage a specific service (e.g., starting, stopping, restarting).
* **start**: This argument tells the script to start the service or process it controls.

So, when you run ctl\_control start , you're initiating the service or application that this control script manages.

A screen shot of a computer

Description automatically generated

* If this is part of your FlowOneAPI or another specific setup, it would be used to start a component or service related to that environment.

* **Stopping Services**:
* Use the command ctl\_control stop to halt the service or process managed by the ctl\_control script.
* Additionally, execute ctl\_control stop to stop the specific catalog service or component.

A computer screen shot of a program

Description automatically generated

* **Switching Users**:
* Utilize sudo su  ilink to switch to the ilink user. The sudo command grants administrative privileges, while su - ilink ensures that you fully switch to the ilink user's environment.
* **Accessing Order Management**:
* Once logged in as the ilink user, proceed to open or access the Order Management system.

This process ensures that the necessary services are stopped and that you have the correct user privileges to manage or interact with the Order Management system.

A screenshot of a computer

Description automatically generated

* User credentials ->
* Username / Administrator    
  Password / guiadmin

A screenshot of a computer

Description automatically generated

**Configuration Process Overview**

* Start by navigating to the **Configuration** section, which allows you to modify system settings.
* When changes are made, the system understands that they will take effect immediately. However, just making the updates does not finalize the changes.
* To fully apply the changes, go to the **Maintenance** page and select **Apply Configuration Changes**.
* This action will refresh the system and ensure that the modifications are applied without causing major disruptions to the request time or ongoing operations.
* After the configuration changes are applied and refreshed, InstantLink will reflect these updates within its processes.
* If any issues arise from the new configurations, it is possible to roll back to previous settings by importing a backup or previously saved configuration.

Now, we will create a logic in BST and test it using the test tool.

ShapeShapeA screenshot of a computer

Description automatically generated

Now, we will click on the 'New Logic' button, and a jar file will be downloaded, where we will create our first logic.

A computer screen shot of a diagram

Description automatically generated

"We create our logic with two parts:

1. **Instance Input Parameters**: Here, we define the name and value of the input parameters.
2. **Condition Parameters**: In this section, we define operand 1, the operator, and operand 2."

A screenshot of a computer

Description automatically generated

Now we will test that logic through the test tool

A screenshot of a computer

Description automatically generated

We will do following things ->

* First we create the new folder
* Then we create the test name
* And add the request

A screenshot of a computer

Description automatically generated

* here we define the parameter and save that request

A screenshot of a computer

Description automatically generated

We add the request to a specific test.

Next, we go to the test tool and click the play button to send the request and test the process.

We can now monitor the request through the monitoring tool."

A screenshot of a computer

Description automatically generated

We can monitor our request by filtering with the date, case ID, user ID, request ID, order number, or request group. After hitting the search button, the request will appear at the bottom. By clicking on the request ID, we can view the details of the request, such as which logic is being processed.

In the BST, the logic name will show as 'FL,' indicating that the request is processing through this workflow logic. The request is being processed by the Service Module Engine, which is running on our local VM server, fo23fp2.vagrant.local.

A screenshot of a computer

Description automatically generated

Below the request details, we can find the associated task. Every request has a task that needs to be completed, and here we can view the details of that task   
   
A screenshot of a computer

Description automatically generated

So same as request details we can click on task id and find more details about task    
   
   
A screenshot of a computer

Description automatically generated 

That’s how we create our logic, define the workflow within it, and test the logic using the test tool. We can monitor the request and task details through the monitoring tab, etc 

**Class: CatalogItem**

**Description:**

The CatalogItem class represents an item in the catalog for the catalog service. It is annotated with @Document to indicate that it maps to a MongoDB document, specifically the catalog\_items collection. This class contains fields for storing information about the catalog item, such as:

* **id**: A unique identifier for the item, annotated with @Id, which maps it to the MongoDB document ID.
* **name**: The name of the catalog item.
* **description**: A brief description of the item.
* **price**: The price of the item, represented as a double.
* **active**: A boolean flag indicating whether the item is active or not. By default, the value is set to true.

**Constructors:**

* **Default constructor**: Provides a no-argument constructor for creating empty instances.
* **Parameterized constructor**: Takes name, description, and price as arguments to initialize an item with these values.

**Getters and Setters:**

Standard getters and setters are provided to allow access and modification of the fields. These include:

* getId, setId: For accessing and setting the ID.
* getName, setName: For accessing and setting the name.
* getDescription, setDescription: For accessing and setting the description.
* getPrice, setPrice: For accessing and setting the price.
* isActive, setActive: For checking and modifying whether the item is active.

**Usage:**

This class can be used to model catalog items in an e-commerce or service-based application where items have attributes like name, description, price, and an active status. The items will be persisted in a MongoDB database.