**Deploying a Java Application**

**1. Introduction**

This project demonstrates the deployment of a multi-component Java application using Docker and Kubernetes. It is designed as a microservices-based architecture with three primary services: **Product Catalogue**, **Shopfront**, and **Stock Manager**. These services are built using Java and packaged as Docker containers, which are deployed and orchestrated using Kubernetes.

**2. Project Structure**

The project follows a well-organized structure to manage the application code, Docker configurations, and Kubernetes deployment files.

**3. Components Overview**

**3.1. Product Catalogue Service**

* **Function**: This service manages the product catalogue, providing APIs to retrieve product information.
* **Technology**: Java-based microservice using Spring Boot.
* **Key Files**:
  + ProductServiceApplication.java: The main class for bootstrapping the service.
  + ProductService.java: Handles business logic related to product management.
  + BasicHealthCheck.java: Implements basic health checks to ensure service availability.

**3.2. Shopfront Service**

* **Function**: Acts as the frontend for the application, interacting with the user and displaying product details.
* **Technology**: Java-based, Spring MVC framework.
* **Key Files**:
  + ShopfrontApplication.java: The entry point for the shopfront application.
  + HomeController.java: Manages user interaction and handles incoming requests.
  + ProductService.java: Connects with the Product Catalogue and Stock Manager services to fetch and display data.

**3.3. Stock Manager Service**

* **Function**: Manages stock information, ensuring that inventory levels are accurately reflected in the system.
* **Technology**: Java-based microservice using Spring Boot.
* **Key Files**:
  + StockManagerApplication.java: The main class for the Stock Manager service.
  + StockService.java: Contains the core logic for managing stock availability and data synchronization.
  + StockNotFoundException.java: Custom exception handling for stock-related errors.

**4. Docker Configuration**

Each service is containerized using Docker. The Dockerfile for each service defines the steps to build the service into a container image.

**4.1. Dockerfile for Product Catalogue Service**

****

**4.2. Dockerfile for Shopfront Service**

**A screen shot of a computer

Description automatically generated**

**4.3. Dockerfile for Stock Manager Service**

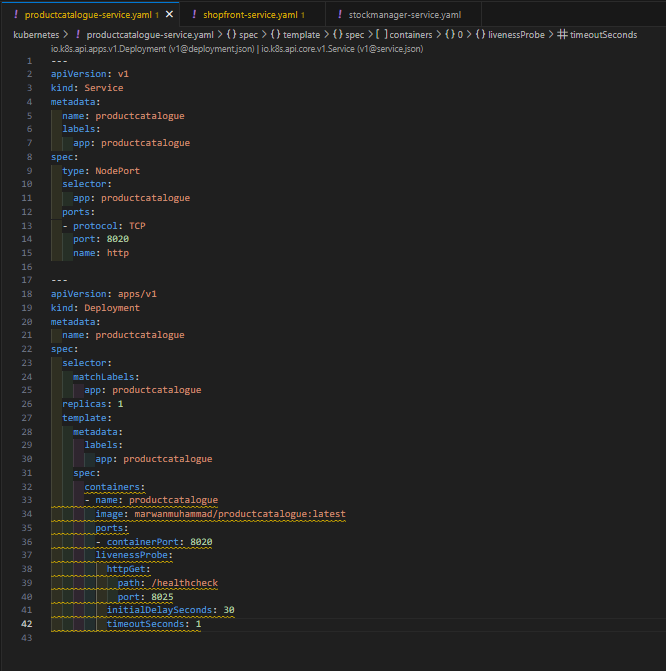
**A screen shot of a computer

Description automatically generated**

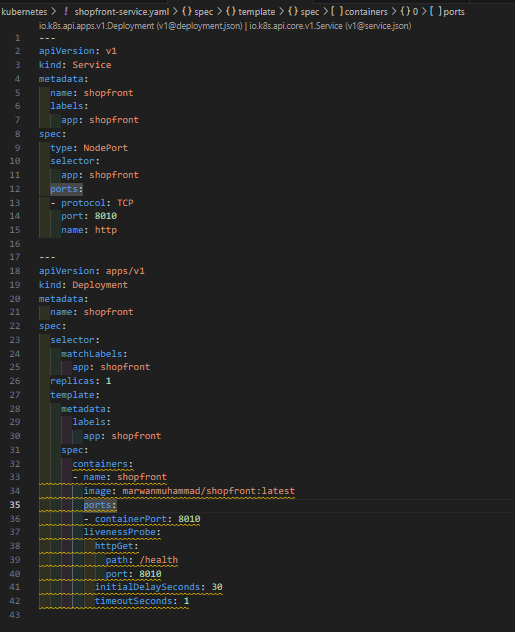
**5. Kubernetes Deployment**

Kubernetes is used to manage and orchestrate the deployment of these containerized services. Each service has a corresponding YAML file to define its deployment and service configuration.

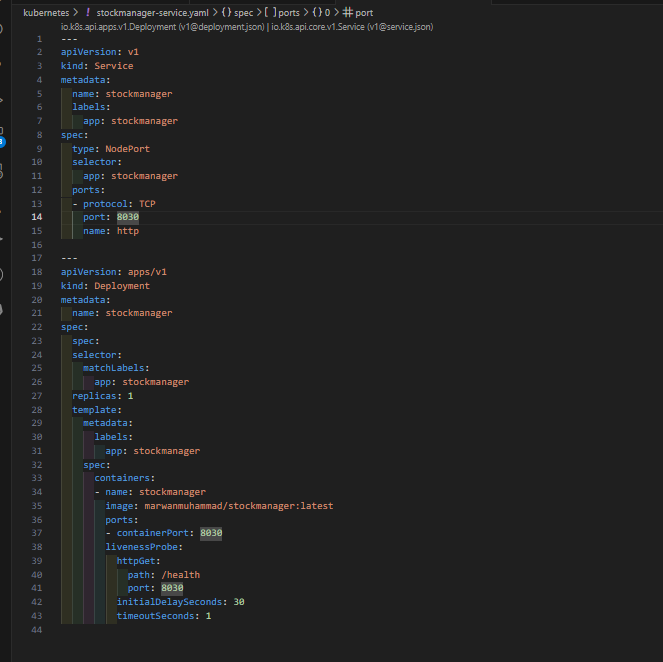
**5.1. Product Catalogue Kubernetes YAML**

****

**5.2. Shopfront Kubernetes YAML**

****

**5.3. Stock Manager Kubernetes YAML**

****

**6. CI/CD Integration**

In a real-world scenario, this project can be integrated with a CI/CD pipeline, such as Jenkins, to automate the process of building Docker images, pushing them to a container registry, and deploying the updated images to Kubernetes.

* **Build**: Compile the Java code and run tests using Maven.
* **Dockerize**: Build Docker images for each service.
* **Deploy**: Use Kubernetes YAML files to deploy the services to a Kubernetes cluster.

**7. Conclusion**

This project showcases a complete deployment pipeline for a microservices-based Java application using Docker and Kubernetes. By breaking the application into smaller, independently deployable services, the project achieves modularity, scalability, and flexibility, allowing for easier management and updates.

This approach ensures efficient orchestration, monitoring, and scaling of services in production environments.