# Microservices Code Compilation

## docker-compose.yml

version: '3.8'  
services:  
 postgres:  
 image: postgres  
 environment:  
 POSTGRES\_USER: postgres  
 POSTGRES\_PASSWORD: root  
 POSTGRES\_DB: postgres  
 ports:  
 - "5432:5432"  
 healthcheck:  
 test: ["CMD-SHELL", "pg\_isready -U postgres"]  
 interval: 10s  
 retries: 5  
 start\_period: 20s  
  
  
 zookeeper:  
 image: confluentinc/cp-zookeeper  
 container\_name: zookeeper  
 environment:  
 ZOOKEEPER\_CLIENT\_PORT: 2181  
 ports:  
 - "2181:2181"  
  
 kafka:  
 image: confluentinc/cp-kafka  
 container\_name: kafka  
 environment:  
 KAFKA\_BROKER\_ID: 1  
 KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:2181  
 KAFKA\_ADVERTISED\_LISTENERS: PLAINTEXT://kafka:9092  
 KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 1  
 depends\_on:  
 - zookeeper  
 ports:  
 - "9092:9092"  
  
 user-service:  
 build: ./user-service  
 container\_name: user-service  
 depends\_on:  
 - postgres  
 ports:  
 - "8081:8080"  
  
 product-service:  
 build: ./product-service  
 container\_name: product-service  
 depends\_on:  
 - postgres  
 ports:  
 - "8082:8080"  
   
 payment-service:  
 build: ./payment-service  
 container\_name: payment-service  
 depends\_on:  
 - order-service  
 - kafka  
 ports:  
 - "8083:8080"  
  
 order-service:  
 build: ./order-service  
 container\_name: order-service  
 depends\_on:  
 - postgres  
 - kafka  
 ports:  
 - "8084:8080"

## microservices.odt

## order-service-deployment.yaml

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: order-service  
spec:  
 replicas: 1  
 selector:  
 matchLabels:  
 app: order-service  
 template:  
 metadata:  
 labels:  
 app: order-service  
 spec:  
 containers:  
 - name: order-service  
 image: mydockerhub/order-service:latest  
 ports:  
 - containerPort: 8084  
 env:  
 - name: SPRING\_KAFKA\_BOOTSTRAP\_SERVERS  
 value: "kafka:9092"  
 - name: SPRING\_KAFKA\_PRODUCER\_ACKS  
 value: "all"  
 - name: SPRING\_DATASOURCE\_URL  
 value: "jdbc:postgresql://postgres-service:5432/order\_db"  
 - name: SPRING\_DATASOURCE\_USERNAME  
 value: "user"  
 - name: SPRING\_DATASOURCE\_PASSWORD  
 value: "password"  
 resources:  
 requests:  
 memory: "256Mi"  
 cpu: "250m"  
 limits:  
 memory: "512Mi"  
 cpu: "500m"  
 livenessProbe:  
 httpGet:  
 path: /actuator/health  
 port: 8084  
 initialDelaySeconds: 5  
 periodSeconds: 10  
 readinessProbe:  
 httpGet:  
 path: /actuator/health  
 port: 8084  
 initialDelaySeconds: 5  
 periodSeconds: 10

## order-service-service.yaml

apiVersion: v1  
kind: Service  
metadata:  
 name: order-service  
spec:  
 selector:  
 app: order-service  
 ports:  
 - protocol: TCP  
 port: 8084  
 targetPort: 8084  
 type: ClusterIP

## payment-service-deployment.yaml

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: payment-service  
spec:  
 replicas: 1  
 selector:  
 matchLabels:  
 app: payment-service  
 template:  
 metadata:  
 labels:  
 app: payment-service  
 spec:  
 containers:  
 - name: payment-service  
 image: mydockerhub/payment-service:latest  
 ports:  
 - containerPort: 8083  
 env:  
 - name: SPRING\_KAFKA\_BOOTSTRAP\_SERVERS  
 value: "kafka:9092"  
 - name: SPRING\_KAFKA\_CONSUMER\_GROUP\_ID  
 value: "payment-group"  
 resources:  
 requests:  
 memory: "256Mi"  
 cpu: "250m"  
 limits:  
 memory: "512Mi"  
 cpu: "500m"  
 livenessProbe:  
 httpGet:  
 path: /actuator/health  
 port: 8083  
 initialDelaySeconds: 5  
 periodSeconds: 10  
 readinessProbe:  
 httpGet:  
 path: /actuator/health  
 port: 8083  
 initialDelaySeconds: 5  
 periodSeconds: 10

## payment-service-service.yaml

apiVersion: v1  
kind: Service  
metadata:  
 name: payment-service  
spec:  
 selector:  
 app: payment-service  
 ports:  
 - protocol: TCP  
 port: 8083  
 targetPort: 8083  
 type: ClusterIP

## product-service-deployment.yaml

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: product-service  
spec:  
 replicas: 1  
 selector:  
 matchLabels:  
 app: product-service  
 template:  
 metadata:  
 labels:  
 app: product-service  
 spec:  
 containers:  
 - name: product-service  
 image: mydockerhub/product-service:latest  
 ports:  
 - containerPort: 8082  
 env:  
 - name: SPRING\_DATASOURCE\_URL  
 value: "jdbc:postgresql://postgres-service:5432/product\_db"  
 - name: SPRING\_DATASOURCE\_USERNAME  
 value: "user"  
 - name: SPRING\_DATASOURCE\_PASSWORD  
 value: "password"  
 resources:  
 requests:  
 memory: "256Mi"  
 cpu: "250m"  
 limits:  
 memory: "512Mi"  
 cpu: "500m"  
 livenessProbe:  
 httpGet:  
 path: /actuator/health  
 port: 8082  
 initialDelaySeconds: 5  
 periodSeconds: 10  
 readinessProbe:  
 httpGet:  
 path: /actuator/health  
 port: 8082  
 initialDelaySeconds: 5  
 periodSeconds: 10

## product-service-service.yaml

apiVersion: v1  
kind: Service  
metadata:  
 name: product-service  
spec:  
 selector:  
 app: product-service  
 ports:  
 - protocol: TCP  
 port: 8082  
 targetPort: 8082  
 type: ClusterIP

## user-service-deployment.yaml

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: user-service  
spec:  
 replicas: 1  
 selector:  
 matchLabels:  
 app: user-service  
 template:  
 metadata:  
 labels:  
 app: user-service  
 spec:  
 containers:  
 - name: user-service  
 image: mydockerhub/user-service:latest  
 ports:  
 - containerPort: 8081

## user-service-service.yaml

apiVersion: v1  
kind: Service  
metadata:  
 name: user-service  
spec:  
 selector:  
 app: user-service  
 ports:  
 - protocol: TCP  
 port: 8081  
 targetPort: 8081  
 type: ClusterIP

## .gitattributes

/gradlew text eol=lf  
\*.bat text eol=crlf  
\*.jar binary

## Dockerfile

# Use OpenJDK 21 image  
FROM openjdk:21-jdk-slim  
  
# Set working directory  
WORKDIR /app  
  
# Copy built JAR from Gradle build  
COPY build/libs/\*.jar app.jar  
  
# Expose the application port (change based on service)  
EXPOSE 8084  
  
# Run the application  
ENTRYPOINT ["java", "-jar", "app.jar"]

## HELP.md

# Getting Started  
  
### Reference Documentation  
For further reference, please consider the following sections:  
  
\* [Official Gradle documentation](https://docs.gradle.org)  
\* [Spring Boot Gradle Plugin Reference Guide](https://docs.spring.io/spring-boot/3.4.3/gradle-plugin)  
\* [Create an OCI image](https://docs.spring.io/spring-boot/3.4.3/gradle-plugin/packaging-oci-image.html)  
\* [Spring Web](https://docs.spring.io/spring-boot/3.4.3/reference/web/servlet.html)  
\* [Spring Boot Actuator](https://docs.spring.io/spring-boot/3.4.3/reference/actuator/index.html)  
\* [Spring Data JPA](https://docs.spring.io/spring-boot/3.4.3/reference/data/sql.html#data.sql.jpa-and-spring-data)  
  
### Guides  
The following guides illustrate how to use some features concretely:  
  
\* [Building a RESTful Web Service](https://spring.io/guides/gs/rest-service/)  
\* [Serving Web Content with Spring MVC](https://spring.io/guides/gs/serving-web-content/)  
\* [Building REST services with Spring](https://spring.io/guides/tutorials/rest/)  
\* [Building a RESTful Web Service with Spring Boot Actuator](https://spring.io/guides/gs/actuator-service/)  
\* [Accessing Data with JPA](https://spring.io/guides/gs/accessing-data-jpa/)  
  
### Additional Links  
These additional references should also help you:  
  
\* [Gradle Build Scans – insights for your project's build](https://scans.gradle.com#gradle)

## OrderApplication.java

package com.example.order;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication  
public class OrderApplication {  
 public static void main(String[] args) {  
 SpringApplication.run(OrderApplication.class, args);  
 }  
}

## OrderController.java

package com.example.order.controllers;  
  
import com.example.order.entities.Order;  
import com.example.order.kafka.OrderProducer;  
import com.example.order.repositories.OrderRepository;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.http.ResponseEntity;  
import org.springframework.web.bind.annotation.\*;  
  
import java.util.List;  
  
@RestController  
@RequestMapping("/orders")  
public class OrderController {  
  
 @Autowired  
 private OrderRepository orderRepository;  
  
 @Autowired  
 private OrderProducer orderProducer;  
  
 @PostMapping("/")  
 public ResponseEntity<String> placeOrder(@RequestBody Order order) {  
 // Save order to database  
 Order savedOrder = orderRepository.save(order);  
   
 // Send order event to Kafka  
 orderProducer.sendOrderEvent(savedOrder);  
   
 return ResponseEntity.ok("Order placed successfully! Order ID: " + savedOrder.getId());  
 }  
  
 @GetMapping("/")  
 public List<Order> getAllOrders() {  
 return orderRepository.findAll();  
 }  
}

## Order.java

package com.example.order.entities;  
  
import jakarta.persistence.\*;  
import lombok.Data;  
import lombok.Getter;  
import lombok.Setter;  
  
@Entity  
@Table(name = "orders")  
@Data  
public class Order {  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 private Long id;  
 private Long productId;  
 private int quantity;  
}

## OrderProducer.java

package com.example.order.kafka;  
  
import com.example.order.entities.Order;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.kafka.core.KafkaTemplate;  
import org.springframework.stereotype.Component;  
  
@Component  
public class OrderProducer {  
  
 @Autowired  
 private KafkaTemplate<String, Order> kafkaTemplate;  
  
 public void sendOrderEvent(Order order) {  
 kafkaTemplate.send("order-topic", order);  
 }  
}

## OrderRepository.java

package com.example.order.repositories;  
  
import com.example.order.entities.Order;  
import org.springframework.data.jpa.repository.JpaRepository;  
import org.springframework.stereotype.Repository;  
  
import java.util.List;  
  
@Repository  
public interface OrderRepository extends JpaRepository<Order, Long> {  
 // Custom query to find orders by productId (if needed)  
 List<Order> findByProductId(Long productId);  
}

## OrderService.java

package com.example.order.services;  
  
import com.example.order.entities.Order;  
import com.example.order.repositories.OrderRepository;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
import java.util.List;  
  
@Service  
public class OrderService {  
  
 @Autowired  
 private OrderRepository orderRepository;  
  
 public Order placeOrder(Order order) {  
 return orderRepository.save(order);  
 }  
  
 public List<Order> getAllOrders() {  
 return orderRepository.findAll();  
 }  
}

## application.yml

spring:  
 datasource:  
 url: jdbc:postgresql://postgres:5432/postgres # ✅ Change from localhost to postgres  
 username: postgres  
 password: root  
 driver-class-name: org.postgresql.Driver  
 jpa:  
 database-platform: org.hibernate.dialect.PostgreSQLDialect # ✅ Add this  
 hibernate:  
 ddl-auto: update  
 show-sql: true

## OrderApplicationTests.java

package com.example.order;  
  
import org.junit.jupiter.api.Test;  
import org.springframework.boot.test.context.SpringBootTest;  
import org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration;  
  
@SpringBootTest(classes = OrderApplication.class)  
class OrderApplicationTests {  
  
 @Test  
 void contextLoads() {  
 }  
}

## application-test.yml

spring:  
 datasource:  
 url: jdbc:h2:mem:testdb # ✅ Use in-memory DB for tests  
 driver-class-name: org.h2.Driver  
 username: sa  
 password:  
 jpa:  
 database-platform: org.hibernate.dialect.H2Dialect  
 hibernate:  
 ddl-auto: update

## .gitattributes

/gradlew text eol=lf  
\*.bat text eol=crlf  
\*.jar binary

## Dockerfile

# Use OpenJDK 21 image  
FROM openjdk:21-jdk-slim  
  
# Set working directory  
WORKDIR /app  
  
# Copy built JAR from Gradle build  
COPY build/libs/\*.jar app.jar  
  
# Expose the application port (change based on service)  
EXPOSE 8083  
  
# Run the application  
ENTRYPOINT ["java", "-jar", "app.jar"]

## HELP.md

# Getting Started  
  
### Reference Documentation  
For further reference, please consider the following sections:  
  
\* [Official Gradle documentation](https://docs.gradle.org)  
\* [Spring Boot Gradle Plugin Reference Guide](https://docs.spring.io/spring-boot/3.4.3/gradle-plugin)  
\* [Create an OCI image](https://docs.spring.io/spring-boot/3.4.3/gradle-plugin/packaging-oci-image.html)  
\* [Spring Web](https://docs.spring.io/spring-boot/3.4.3/reference/web/servlet.html)  
\* [Spring Boot Actuator](https://docs.spring.io/spring-boot/3.4.3/reference/actuator/index.html)  
\* [Spring Data JPA](https://docs.spring.io/spring-boot/3.4.3/reference/data/sql.html#data.sql.jpa-and-spring-data)  
  
### Guides  
The following guides illustrate how to use some features concretely:  
  
\* [Building a RESTful Web Service](https://spring.io/guides/gs/rest-service/)  
\* [Serving Web Content with Spring MVC](https://spring.io/guides/gs/serving-web-content/)  
\* [Building REST services with Spring](https://spring.io/guides/tutorials/rest/)  
\* [Building a RESTful Web Service with Spring Boot Actuator](https://spring.io/guides/gs/actuator-service/)  
\* [Accessing Data with JPA](https://spring.io/guides/gs/accessing-data-jpa/)  
  
### Additional Links  
These additional references should also help you:  
  
\* [Gradle Build Scans – insights for your project's build](https://scans.gradle.com#gradle)

## PaymentApplication.java

package com.example.payment;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication  
public class PaymentApplication {  
 public static void main(String[] args) {  
 SpringApplication.run(PaymentApplication.class, args);  
 }  
}

## Order.java

package com.example.payment.dto;  
  
import lombok.Data;  
import lombok.Getter;  
import lombok.Setter;  
  
@Data // ✅ Generates getters, setters, and `toString()`  
@Getter  
@Setter  
public class Order {  
 private Long id;  
 private Long productId;  
 private int quantity;  
}

## PaymentConsumer.java

package com.example.payment.kafka;  
  
import com.example.payment.dto.Order; // ✅ Import the new Order DTO  
import com.example.payment.services.PaymentService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.kafka.annotation.KafkaListener;  
import org.springframework.stereotype.Component;  
  
@Component  
public class PaymentConsumer {  
  
 @Autowired  
 private PaymentService paymentService;  
  
 @KafkaListener(topics = "order-topic", groupId = "payment-group")  
 public void processOrder(Order order) { // ✅ Use the new Order DTO  
 System.out.println("Received Order: " + order);  
 paymentService.processPayment(order.getId());  
 }  
}

## PaymentService.java

package com.example.payment.services;  
  
import org.springframework.stereotype.Service;  
  
@Service  
public class PaymentService {  
  
 public void processPayment(Long orderId) {  
 System.out.println("Processing payment for Order ID: " + orderId);  
 }  
}

## application.yml

server:  
 port: 8083  
  
spring:  
 kafka:  
 bootstrap-servers: kafka:9092  
 consumer:  
 group-id: payment-group

## PaymentApplicationTests.java

package com.example.payment;  
  
import org.junit.jupiter.api.Test;  
import org.springframework.boot.test.context.SpringBootTest;  
import org.springframework.boot.test.util.TestPropertyValues;  
import org.springframework.context.ApplicationContextInitializer;  
import org.springframework.context.ConfigurableApplicationContext;  
import org.springframework.test.context.ContextConfiguration;  
import org.testcontainers.containers.KafkaContainer;  
import org.testcontainers.utility.DockerImageName;  
  
@SpringBootTest  
@ContextConfiguration(initializers = PaymentApplicationTests.Initializer.class)  
class PaymentApplicationTests {  
  
 static KafkaContainer kafkaContainer = new KafkaContainer(DockerImageName.parse("confluentinc/cp-kafka:7.4.1"));  
  
 static class Initializer implements ApplicationContextInitializer<ConfigurableApplicationContext> {  
 @Override  
 public void initialize(ConfigurableApplicationContext context) {  
 kafkaContainer.start();  
 TestPropertyValues.of(  
 "spring.kafka.bootstrap-servers=" + kafkaContainer.getBootstrapServers()  
 ).applyTo(context.getEnvironment());  
 }  
 }  
  
 @Test  
 void contextLoads() {  
 }  
}

## .gitattributes

/gradlew text eol=lf  
\*.bat text eol=crlf  
\*.jar binary

## Dockerfile

# Use OpenJDK 21 image  
FROM openjdk:21-jdk-slim  
  
# Set working directory  
WORKDIR /app  
  
# Copy built JAR from Gradle build  
COPY build/libs/\*.jar app.jar  
  
# Expose the application port (change based on service)  
EXPOSE 8082  
  
# Run the application  
ENTRYPOINT ["java", "-jar", "app.jar"]

## HELP.md

# Getting Started  
  
### Reference Documentation  
For further reference, please consider the following sections:  
  
\* [Official Gradle documentation](https://docs.gradle.org)  
\* [Spring Boot Gradle Plugin Reference Guide](https://docs.spring.io/spring-boot/3.4.3/gradle-plugin)  
\* [Create an OCI image](https://docs.spring.io/spring-boot/3.4.3/gradle-plugin/packaging-oci-image.html)  
\* [Spring Web](https://docs.spring.io/spring-boot/3.4.3/reference/web/servlet.html)  
\* [Spring Boot Actuator](https://docs.spring.io/spring-boot/3.4.3/reference/actuator/index.html)  
\* [Spring Data JPA](https://docs.spring.io/spring-boot/3.4.3/reference/data/sql.html#data.sql.jpa-and-spring-data)  
  
### Guides  
The following guides illustrate how to use some features concretely:  
  
\* [Building a RESTful Web Service](https://spring.io/guides/gs/rest-service/)  
\* [Serving Web Content with Spring MVC](https://spring.io/guides/gs/serving-web-content/)  
\* [Building REST services with Spring](https://spring.io/guides/tutorials/rest/)  
\* [Building a RESTful Web Service with Spring Boot Actuator](https://spring.io/guides/gs/actuator-service/)  
\* [Accessing Data with JPA](https://spring.io/guides/gs/accessing-data-jpa/)  
  
### Additional Links  
These additional references should also help you:  
  
\* [Gradle Build Scans – insights for your project's build](https://scans.gradle.com#gradle)

## ProductApplication.java

package com.example.product;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication  
public class ProductApplication {  
 public static void main(String[] args) {  
 SpringApplication.run(ProductApplication.class, args);  
 }  
}

## ProductController.java

package com.example.product.controllers;  
  
import com.example.product.entities.Product;  
import com.example.product.repositories.ProductRepository;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.\*;  
  
import java.util.List;  
  
@RestController  
@RequestMapping("/products")  
public class ProductController {  
  
 @Autowired  
 private ProductRepository productRepository;  
  
 @GetMapping("/")  
 public List<Product> getAllProducts() {  
 return productRepository.findAll();  
 }  
  
 @PostMapping("/")  
 public Product addProduct(@RequestBody Product product) {  
 return productRepository.save(product);  
 }  
}

## Product.java

package com.example.product.entities;  
  
import jakarta.persistence.\*;  
import lombok.Data;  
import lombok.Getter;  
import lombok.Setter;  
  
@Entity  
@Table(name = "products")  
@Data  
public class Product {  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 private Long id;  
 private String name;  
 private double price;  
 private int stock;  
}

## ProductRepository.java

package com.example.product.repositories;  
  
import com.example.product.entities.Product;  
import org.springframework.data.jpa.repository.JpaRepository;  
import org.springframework.stereotype.Repository;  
  
@Repository  
public interface ProductRepository extends JpaRepository<Product, Long> {  
}

## ProductService.java

package com.example.product.services;  
  
import com.example.product.entities.Product;  
import com.example.product.repositories.ProductRepository;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
import java.util.List;  
  
@Service  
public class ProductService {  
  
 @Autowired  
 private ProductRepository productRepository;  
  
 public List<Product> getAllProducts() {  
 return productRepository.findAll();  
 }  
  
 public Product addProduct(Product product) {  
 return productRepository.save(product);  
 }  
}

## application.yml

server:  
 port: 8082  
  
spring:  
 datasource:  
 url: jdbc:postgresql://postgres:5432/postgres # ✅ Ensure this is correct!  
 username: postgres  
 password: root  
 driver-class-name: org.postgresql.Driver  
 jpa:  
 database-platform: org.hibernate.dialect.PostgreSQLDialect  
 hibernate:  
 ddl-auto: update # ✅ This ensures tables are created automatically  
 show-sql: true

## ProductApplicationTests.java

package com.example.product;  
  
import org.junit.jupiter.api.Test;  
import org.springframework.boot.test.context.SpringBootTest;  
import org.springframework.test.context.ActiveProfiles;  
  
@SpringBootTest  
@ActiveProfiles("test") // ✅ Use "application-test.yml" during tests  
class ProductApplicationTests {  
  
 @Test  
 void contextLoads() {  
 }  
}

## application-test.yml

spring:  
 datasource:  
 url: jdbc:h2:mem:testdb  
 driver-class-name: org.h2.Driver  
 username: sa  
 password:  
 jpa:  
 database-platform: org.hibernate.dialect.H2Dialect  
 hibernate:  
 ddl-auto: update  
 properties:  
 hibernate:  
 dialect: org.hibernate.dialect.H2Dialect

## .gitattributes

/gradlew text eol=lf  
\*.bat text eol=crlf  
\*.jar binary

## Dockerfile

# Use OpenJDK 21 image  
FROM openjdk:21-jdk-slim  
  
# Set working directory  
WORKDIR /app  
  
# Copy built JAR from Gradle build  
COPY build/libs/\*.jar app.jar  
  
# Expose the application port (change based on service)  
EXPOSE 8081  
  
# Run the application  
ENTRYPOINT ["java", "-jar", "app.jar"]

## HELP.md

# Getting Started  
  
### Reference Documentation  
For further reference, please consider the following sections:  
  
\* [Official Gradle documentation](https://docs.gradle.org)  
\* [Spring Boot Gradle Plugin Reference Guide](https://docs.spring.io/spring-boot/3.4.3/gradle-plugin)  
\* [Create an OCI image](https://docs.spring.io/spring-boot/3.4.3/gradle-plugin/packaging-oci-image.html)  
\* [Spring Web](https://docs.spring.io/spring-boot/3.4.3/reference/web/servlet.html)  
\* [Spring Boot Actuator](https://docs.spring.io/spring-boot/3.4.3/reference/actuator/index.html)  
\* [Spring Data JPA](https://docs.spring.io/spring-boot/3.4.3/reference/data/sql.html#data.sql.jpa-and-spring-data)  
  
### Guides  
The following guides illustrate how to use some features concretely:  
  
\* [Building a RESTful Web Service](https://spring.io/guides/gs/rest-service/)  
\* [Serving Web Content with Spring MVC](https://spring.io/guides/gs/serving-web-content/)  
\* [Building REST services with Spring](https://spring.io/guides/tutorials/rest/)  
\* [Building a RESTful Web Service with Spring Boot Actuator](https://spring.io/guides/gs/actuator-service/)  
\* [Accessing Data with JPA](https://spring.io/guides/gs/accessing-data-jpa/)  
  
### Additional Links  
These additional references should also help you:  
  
\* [Gradle Build Scans – insights for your project's build](https://scans.gradle.com#gradle)

## UserApplication.java

package com.example.user;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.context.annotation.ComponentScan;  
  
@SpringBootApplication  
@ComponentScan(basePackages = "com.example.user") // ✅ Ensure Spring scans the right package  
public class UserApplication {  
 public static void main(String[] args) {  
 SpringApplication.run(UserApplication.class, args);  
 }  
}

## AuthController.java

package com.example.user.controllers;  
  
import com.example.user.security.JwtUtil;  
import org.springframework.security.authentication.AuthenticationManager;  
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;  
import org.springframework.security.core.Authentication;  
import org.springframework.web.bind.annotation.\*;  
import com.example.user.dto.AuthRequest;  
  
@RestController  
@RequestMapping("/auth")  
public class AuthController {  
  
 private final AuthenticationManager authManager;  
 private final JwtUtil jwtUtil;  
  
 public AuthController(AuthenticationManager authManager, JwtUtil jwtUtil) {  
 this.authManager = authManager;  
 this.jwtUtil = jwtUtil;  
 }  
  
 @PostMapping("/login")  
 public String login(@RequestBody AuthRequest request) {  
 Authentication auth = authManager.authenticate(  
 new UsernamePasswordAuthenticationToken(request.getUsername(), request.getPassword())  
 );  
 return jwtUtil.generateToken(request.getUsername());  
 }  
}

## AuthRequest.java

package com.example.user.dto;  
  
import lombok.Getter;  
import lombok.Setter;  
  
@Getter  
@Setter  
public class AuthRequest {  
 private String username;  
 private String password;  
}

## User.java

package com.example.user.entities;  
  
import jakarta.persistence.\*;  
import lombok.\*;  
  
@Entity  
@Table(name = "users")  
@Getter  
@Setter  
@NoArgsConstructor // ✅ Required for JPA to work  
@AllArgsConstructor // ✅ Generates a constructor with all fields  
public class User {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 private Long id;  
  
 private String username;  
 private String password;  
  
 // ✅ Custom constructor to allow creating new users  
 public User(String username, String password) {  
 this.username = username;  
 this.password = password;  
 }  
}

## UserRepository.java

package com.example.user.repositories;  
  
import org.springframework.data.jpa.repository.JpaRepository;  
import org.springframework.stereotype.Repository;  
import com.example.user.entities.User;  
import java.util.Optional;  
  
@Repository // ✅ Required for Spring Boot to detect this repository  
public interface UserRepository extends JpaRepository<User, Long> {  
  
 // Find user by username  
 Optional<User> findByUsername(String username);  
  
 // Check if user exists by username  
 boolean existsByUsername(String username);  
}

## JwtUtil.java

package com.example.user.security;  
  
import io.jsonwebtoken.Claims;  
import io.jsonwebtoken.Jwts;  
import io.jsonwebtoken.io.Decoders;  
import io.jsonwebtoken.security.Keys;  
import org.springframework.stereotype.Component;  
import java.security.Key;  
import java.util.Date;  
  
@Component // ✅ Register JwtUtil as a Spring Bean  
public class JwtUtil {  
 private static final String SECRET\_KEY = "my-secret-key-for-jwt-my-secret-key-for-jwt"; // 256-bit key required  
  
 private static Key getSigningKey() {  
 byte[] keyBytes = Decoders.BASE64.decode(SECRET\_KEY);  
 return Keys.hmacShaKeyFor(keyBytes);  
 }  
  
 public static Claims extractClaims(String token) {  
 return Jwts.parserBuilder()  
 .setSigningKey(getSigningKey())  
 .build() // ✅ Required before `parseClaimsJws`  
 .parseClaimsJws(token)  
 .getBody();  
 }  
  
 public static String generateToken(String username) {  
 return Jwts.builder()  
 .setSubject(username)  
 .setIssuedAt(new Date(System.currentTimeMillis()))  
 .setExpiration(new Date(System.currentTimeMillis() + 1000 \* 60 \* 60 \* 10)) // 10 hours expiry  
 .signWith(getSigningKey())  
 .compact();  
 }  
}

## SecurityConfig.java

package com.example.user.security;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.security.authentication.AuthenticationManager;  
import org.springframework.security.config.annotation.authentication.configuration.AuthenticationConfiguration;  
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;  
  
@Configuration  
@EnableWebSecurity // ✅ Enables Spring Security  
public class SecurityConfig {  
  
 @Bean  
 public AuthenticationManager authenticationManager(AuthenticationConfiguration authenticationConfiguration) throws Exception {  
 return authenticationConfiguration.getAuthenticationManager();  
 }  
}

## UserService.java

package com.example.user.services;  
  
import com.example.user.entities.User;  
import com.example.user.repositories.UserRepository;  
import org.springframework.stereotype.Service;  
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;  
import org.springframework.security.crypto.password.PasswordEncoder;  
  
import java.util.Optional;  
  
@Service // ✅ Required for Spring Boot to detect this service  
public class UserService {  
  
 private final UserRepository userRepository;  
 private final PasswordEncoder passwordEncoder;  
  
 // ✅ Constructor Injection (Preferred over `@Autowired` field injection)  
 public UserService(UserRepository userRepository) {  
 this.userRepository = userRepository;  
 this.passwordEncoder = new BCryptPasswordEncoder(); // Secure password hashing  
 }  
  
 // 🔹 Register a new user  
 public User registerUser(String username, String rawPassword) {  
 if (userRepository.existsByUsername(username)) {  
 throw new IllegalArgumentException("Username already exists!");  
 }  
  
 String hashedPassword = passwordEncoder.encode(rawPassword);  
 User user = new User(username, hashedPassword);  
 return userRepository.save(user);  
 }  
  
 // 🔹 Find user by username  
 public Optional<User> findByUsername(String username) {  
 return userRepository.findByUsername(username);  
 }  
  
 // 🔹 Validate user credentials  
 public boolean authenticate(String username, String rawPassword) {  
 Optional<User> userOptional = userRepository.findByUsername(username);  
 return userOptional.map(user -> passwordEncoder.matches(rawPassword, user.getPassword()))  
 .orElse(false);  
 }  
}

## application.yml

spring:  
 datasource:  
 url: jdbc:postgresql://postgres:5432/postgres # ✅ Change from localhost to postgres  
 username: postgres  
 password: root  
 driver-class-name: org.postgresql.Driver  
 jpa:  
 database-platform: org.hibernate.dialect.PostgreSQLDialect # ✅ Add this  
 hibernate:  
 ddl-auto: update  
 show-sql: true

## UserApplicationTests.java

package com.example.user;  
  
import org.junit.jupiter.api.Test;  
import org.springframework.boot.test.context.SpringBootTest;  
import org.springframework.test.context.ActiveProfiles;  
import org.springframework.boot.autoconfigure.security.servlet.SecurityAutoConfiguration;   
  
@SpringBootTest(classes = UserApplication.class) // ✅ Explicitly load UserApplication  
@ActiveProfiles("test") // ✅ Use test configurations  
class UserApplicationTests {  
  
 @Test  
 void contextLoads() {  
 }  
}

## application-test.yml

spring:  
 datasource:  
 url: jdbc:h2:mem:testdb # ✅ Use in-memory DB for tests  
 driver-class-name: org.h2.Driver  
 username: sa  
 password:  
 jpa:  
 database-platform: org.hibernate.dialect.H2Dialect  
 hibernate:  
 ddl-auto: update