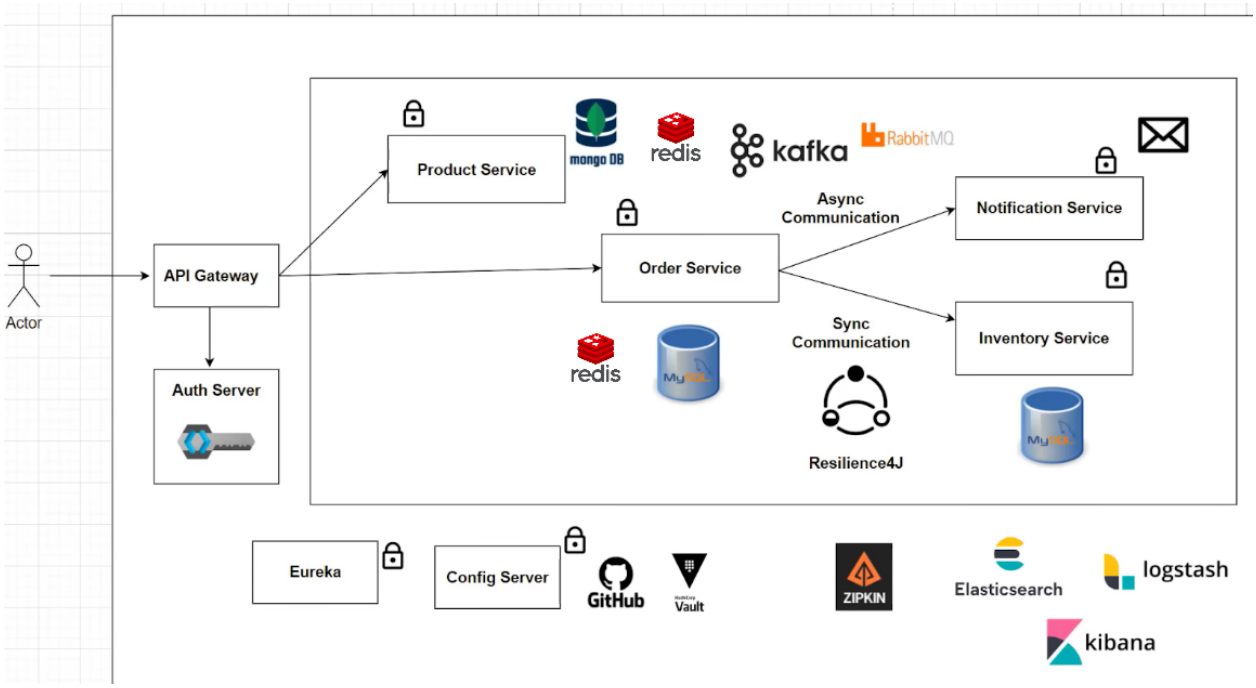
PROJECT: ONLINE SHOPPING APPLICATION

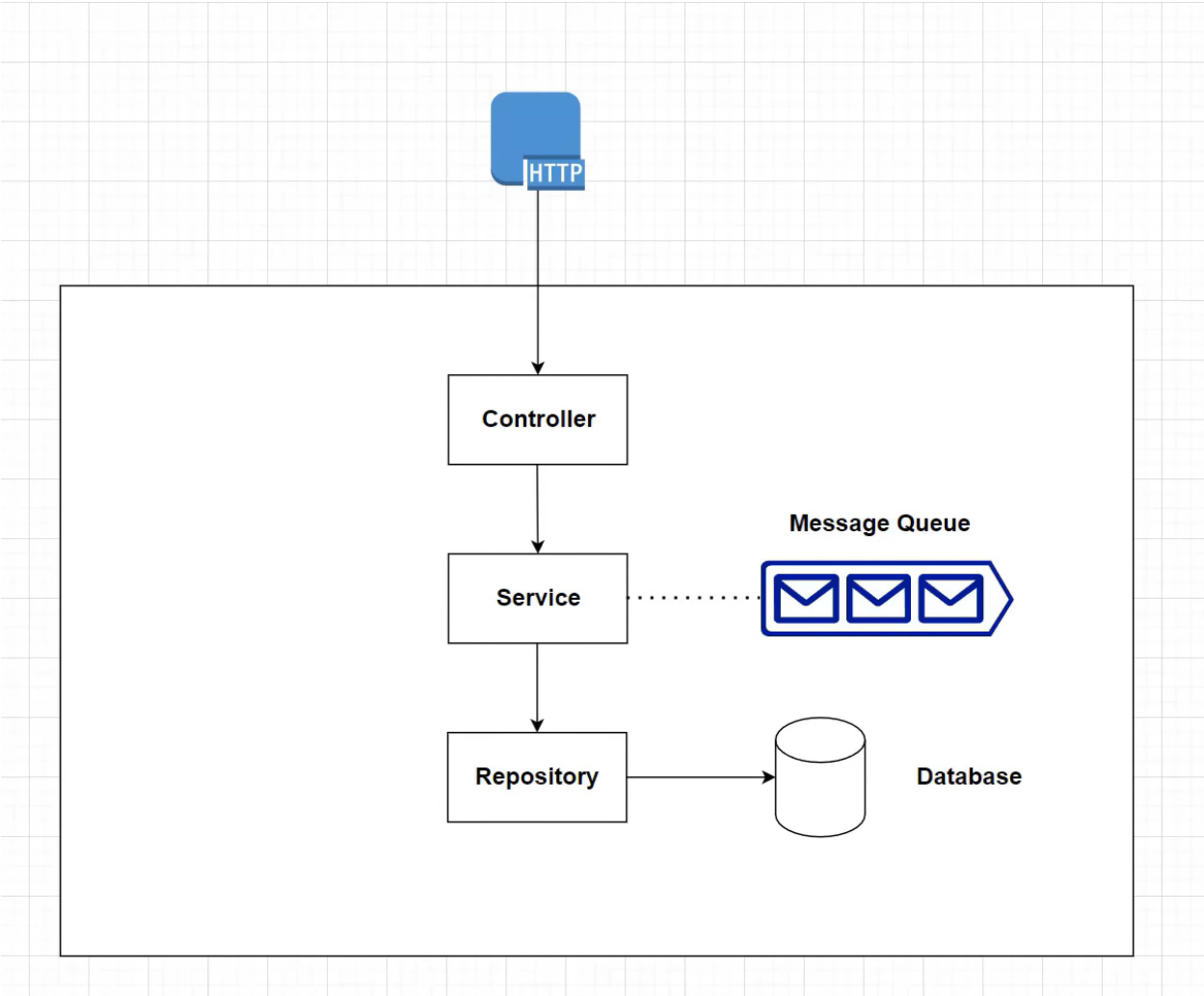
Neler işleyeceğiz?

- Service Discovery
- Centralized Configuration
- Distributed Tracing
- Event Driven Architecture
- Centralized Logging
- Circuit Breaker
- Secure Microservice using Keycloak

Geliştireceğimiz Servisler:

- **Product Service**: Create and View Products, acts as Product Catalog (MongoDB)
- **Order Service**: Can order Products (PostgreSQL)
- **Inventory Service**: Can check if a product is in stock or not(PostgreSQL).
- **Notification Service**: Can send notifications, after order is placed. (RabbitMQ)





Implementation:

Product Service için MongoDB veritabanı, editor olarak da MongoExpress kullanacağım.
İkisi için bir docker-compose.yml dosyası hazırladım.

docker-compose.yml
—----------

```yaml
version: "3.5"

services:
  mongo:
    image: mongo:latest
    container_name: mongo
    environment:
      MONGO_INITDB_ROOT_USERNAME: admin
      MONGO_INITDB_ROOT_PASSWORD: admin
    restart: unless-stopped
    ports:
      - "0.0.0.0:27017:27017"
    networks:
      - MONGO
    volumes:
      - type: volume
        source: MONGO_DATA
        target: /data/db
      - type: volume
        source: MONGO_CONFIG
        target: /data/configdb
  mongo-express:
    image: mongo-express:latest
    container_name: mongo-express
    environment:
      ME_CONFIG_MONGODB_ADMINUSERNAME: admin
      ME_CONFIG_MONGODB_ADMINPASSWORD: admin
      ME_CONFIG_MONGODB_SERVER: mongo
      ME_CONFIG_MONGODB_PORT: "27017"
    restart: unless-stopped
    ports:
      - "0.0.0.0:8081:8081"
    networks:
      - MONGO
    depends_on:
      - mongo

networks:
  MONGO:
    name: MONGO
```

```
volumes:
  MONGO_DATA:
    name: MONGO_DATA
  MONGO_CONFIG:
    name: MONGO_CONFIG
```

Dosyaya PostgreSQL ve PGADMIN eklenmiş hali:

****

```
version: "3.5"


services:
  mongo:
    image: mongo:latest
    container_name: mongo
    environment:
      MONGO_INITDB_ROOT_USERNAME: admin
      MONGO_INITDB_ROOT_PASSWORD: admin
    restart: unless-stopped
    ports:
      - "27017:27017"
    networks:
      - MONGO
    volumes:
      - type: volume
        source: MONGO_DATA
        target: /data/db
      - type: volume
        source: MONGO_CONFIG
        target: /data/configdb
  mongo-express:
    image: mongo-express:latest
    container_name: mongo-express
    environment:
      ME_CONFIG_MONGODB_ADMINUSERNAME: admin
      ME_CONFIG_MONGODB_ADMINPASSWORD: admin
      ME_CONFIG_BASICAUTH_USERNAME: express
      ME_CONFIG_BASICAUTH_PASSWORD: express
```

```
      ME_CONFIG_MONGODB_SERVER: mongo
      ME_CONFIG_MONGODB_PORT: "27017"
    restart: unless-stopped
    ports:
      - "8081:8081"
    networks:
      - MONGO
    depends_on:
      - mongo


  postgres:
    container_name: pgdb
    image: postgres
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      PGDATA: /data/postgres
    volumes:
      - postgres:/data/postgres
    ports:
      - "5432:5432"
    networks:
      - postgres
    restart: unless-stopped
  pgadmin:
    container_name: pgadmin
    image: dpage/pgadmin4
    environment:
      PGADMIN_DEFAULT_EMAIL: ${PGADMIN_DEFAULT_EMAIL:-pgadmin4@pgadmin.org}
      PGADMIN_DEFAULT_PASSWORD: ${PGADMIN_DEFAULT_PASSWORD:-admin}
      PGADMIN_CONFIG_SERVER_MODE: 'False'
    volumes:
      - pgadmin:/var/lib/pgadmin
    ports:
      - "5050:80"
    networks:
      - postgres
    restart: unless-stopped
networks:
  MONGO:
    name: MONGO
  postgres:
    driver: bridge
```

```
volumes:
  MONGO_DATA:
    name: MONGO_DATA
  MONGO_CONFIG:
    name: MONGO_CONFIG
  postgres:
  pgadmin:
```

****

Bu dosyayı,

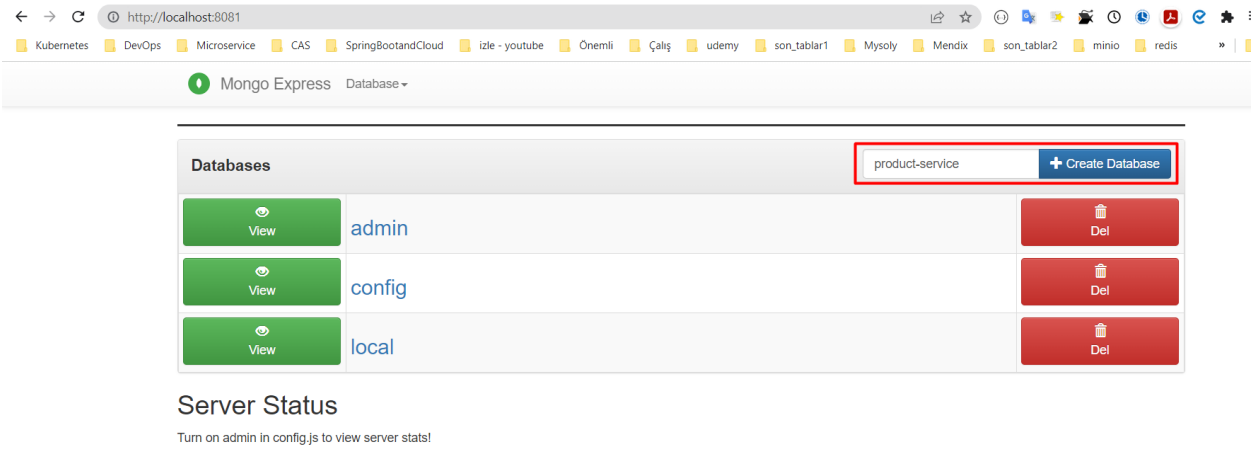
docker-compose up -d

komutuyla çalıştıralım.

http://localhost:8081

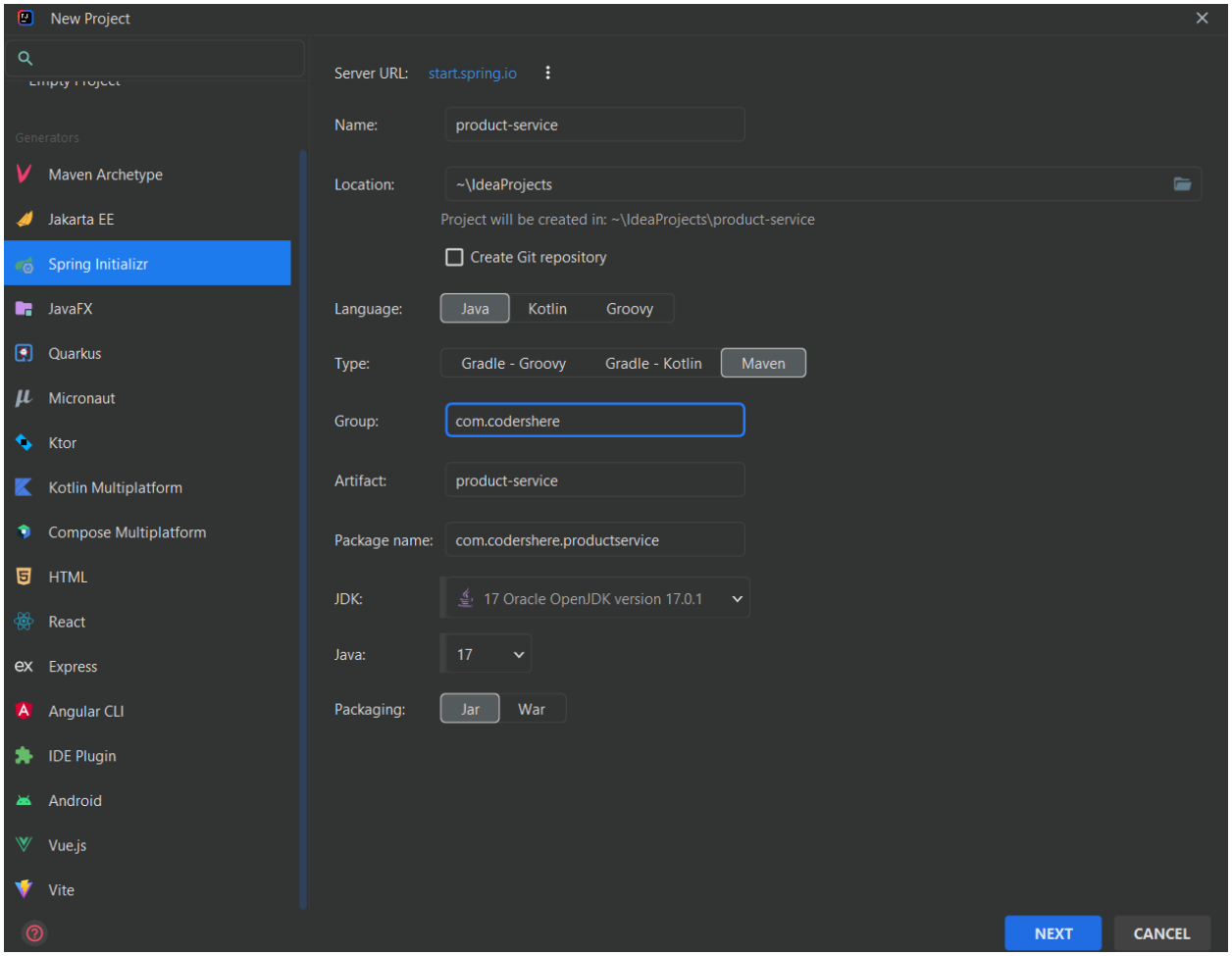
adresinden MongoExpress ekranına gidelim.

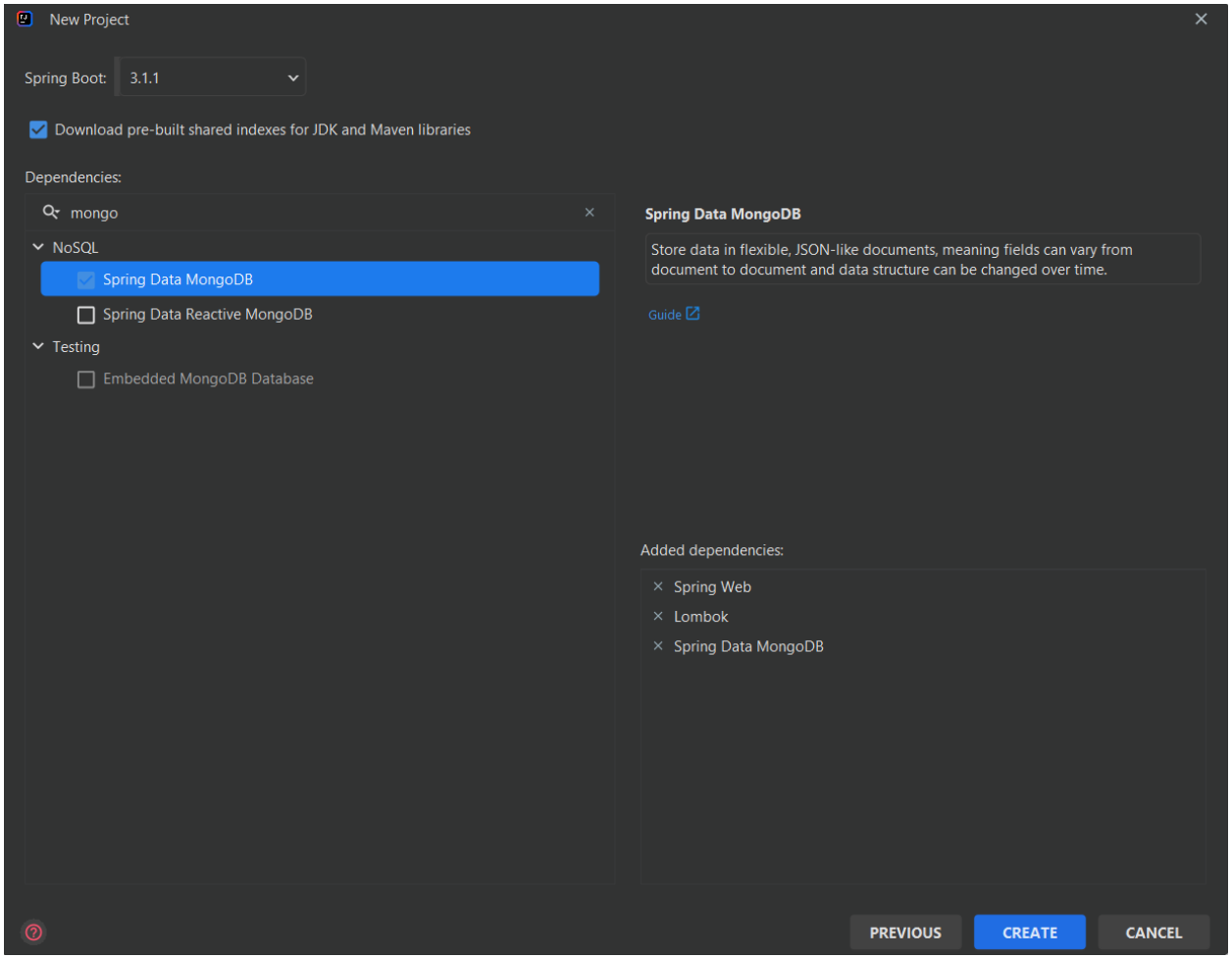
product-service isminde bir database oluşturalım.



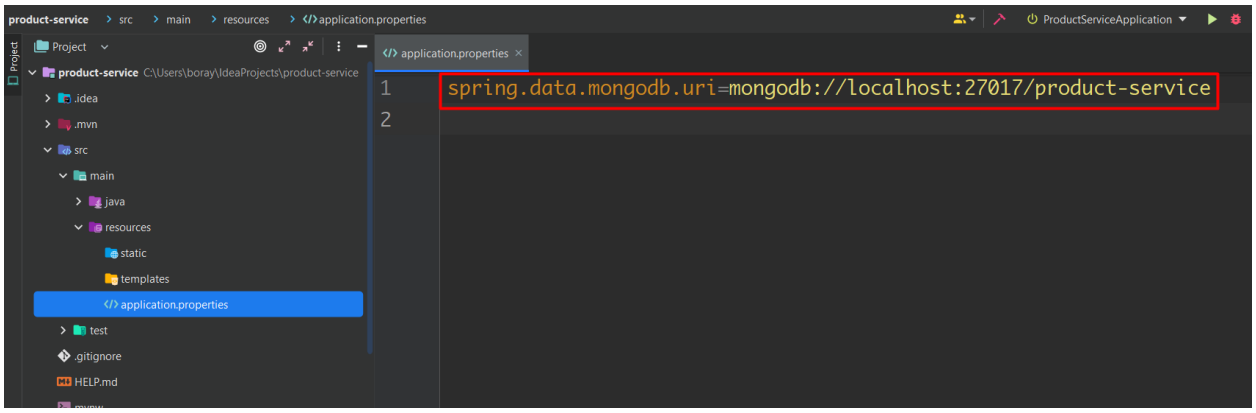
İlk mikroservisimizde bu veritabanını kullanacağız.

Şimdi Intellij Idea'yı açıp product-service'i implement edelim.

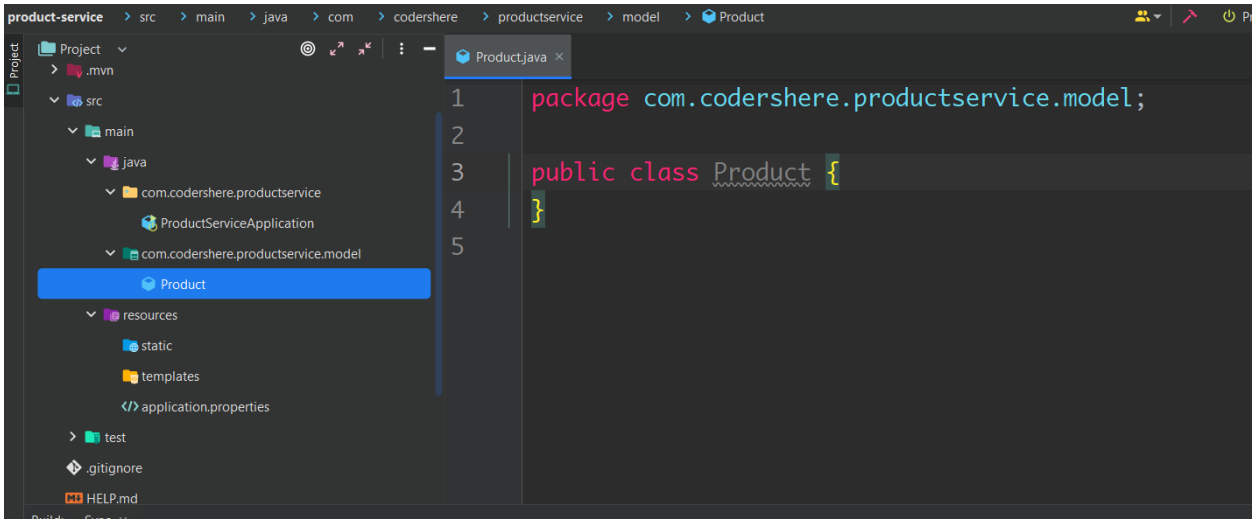

New Project
Generators
Maven Archetype
Jakarta EE
Spring Initializr
JavaFX
Quarkus
Micronaut
Ktor
Kotlin Multiplatform
Compose Multiplatform
HTML
React
Express
Angular CLI
IDE Plugin
Android
Vue.js
Vite
Server URL: start.spring.io
Name: product-service
Location: ~\IdeaProjects
Project will be created in: ~\IdeaProjects\product-service
Create Git repository
Language: Java Kotlin Groovy
Type: Gradle - Groovy Gradle - Kotlin Maven
Group: com.codershere
Artifact: product-service
Package name: com.codershere.productservice
JDK: 17 Oracle OpenJDK version 17.0.1
Java: 17
Packaging: Jar War
NEXT
CANCEL



İlk olarak application.properties dosyasına mongodb bağlantımızı yazalım.



model diye bir package oluşturup, içinde Product class'ını oluşturalım.



class'ı bir MongodB documentine çevirelim.

Lombok annotationlarını ekleyelim.

```java
Product.java
import lombok.NoArgsConstructor;
import org.springframework.data.mongodb.core.mapping.Document

@Document(value = "product")
@AllArgsConstructor
@NoArgsConstructor
@Builder
@Data
public class Product {

}
```

Classın değişkenlerini ekleyelim.

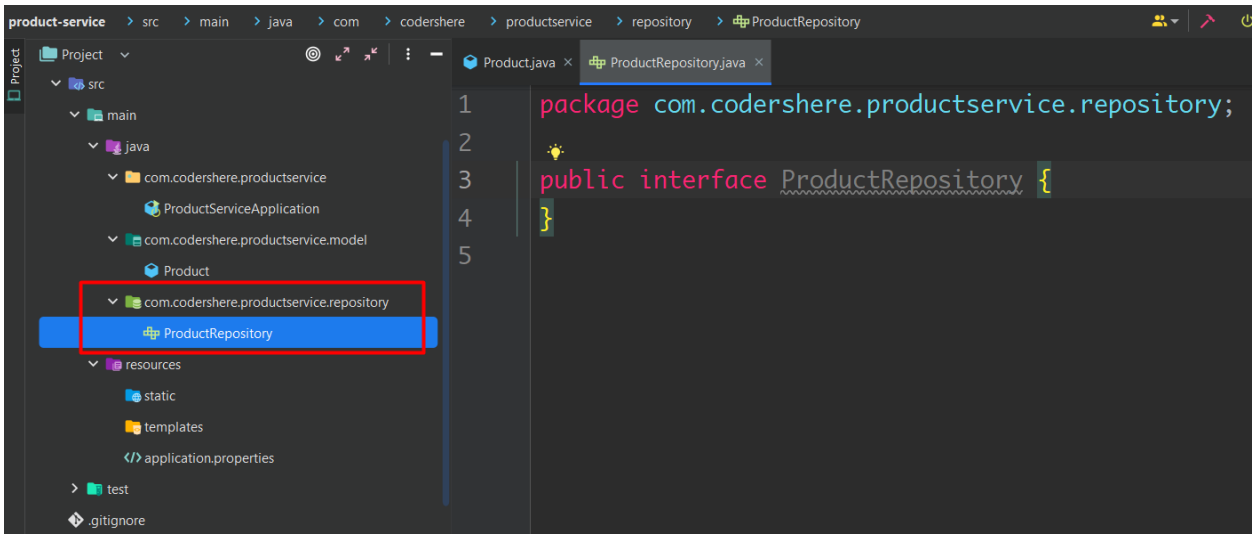
```java
Product.java
@AllArgsConstructor
@NoArgsConstructor
@Builder
@Data
public class Product {

    private String id;
    private String name;
    private String description;
    private BigDecimal price;

}
```

Ve id'yi primary key yapalım.

```java
@AllArgsConstructor
@NoArgsConstructor
@Builder
@Data
public class Product {

    @Id
    private String id;
    private String name;
    private String description;
    private BigDecimal price;

}
```

Repository’i oluşturalım.

Bunun için repository adında bir package oluşturup, içine ProductRepository adında bir interface oluşturalım.



interface'i MongoRepository'den extend edelim.

```java
package com.codershere.productservice.repository;

import com.codershere.productservice.model.Product;
import org.springframework.data.mongodb.repository.MongoRepository;

public interface ProductRepository extends MongoRepository<Product, String> {
}
```

controller package'ı oluşturalım ve içine ProductController class'ı koyalım.

```java
package com.codershere.productservice.controller;

public class ProductController {
}
```

RestController annotation'ı ve RequestMapping ekleyelim.

```java
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api/product")
public class ProductController {
}
```

create product metoduyla başlayalım.

```java
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api/product")
public class ProductController {

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public void createProduct()
    {

    }

}
```

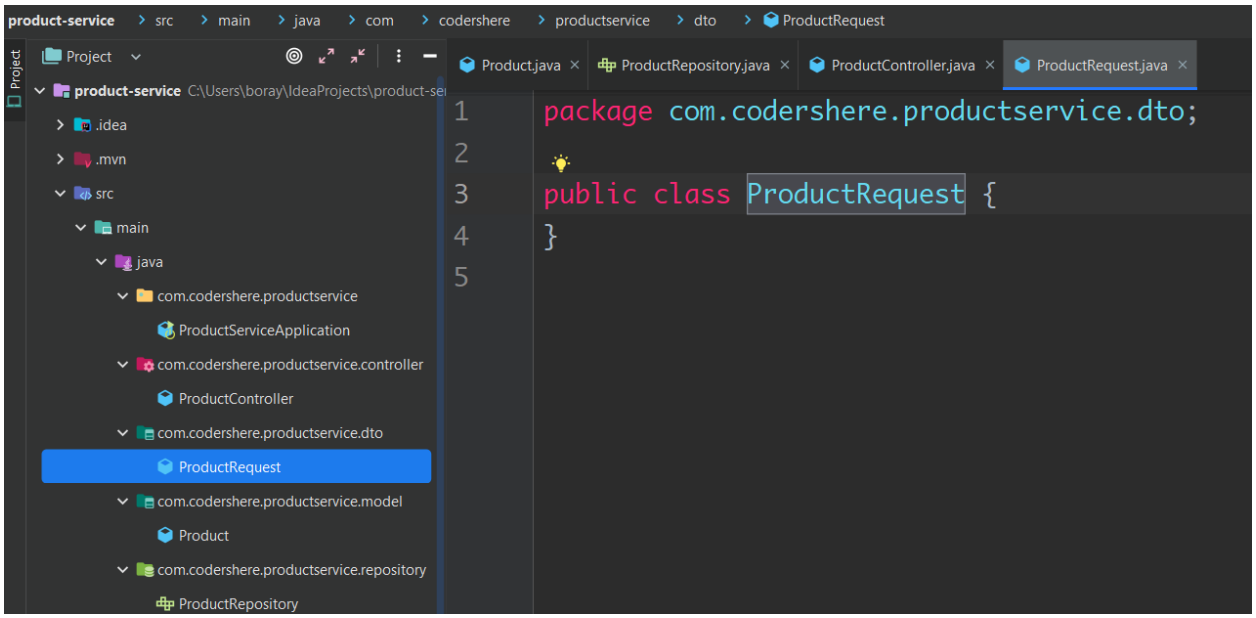
RequestBody'i Entity tipinde değil, DTO tipinde ekleyelim.

Ama ProductRequest(id si yok) ve ProductResponse(id si var) tipinde iki ayrı DTO kullanacağız.

create metodunda ProductRequest kullanacağız.

```
@RestController
@RequestMapping("/api/product")
public class ProductController {

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public void createProduct(@RequestBody ProductRequest productRequest)
    {

    }

}
```

class'ı dto package'ında oluşturalım.



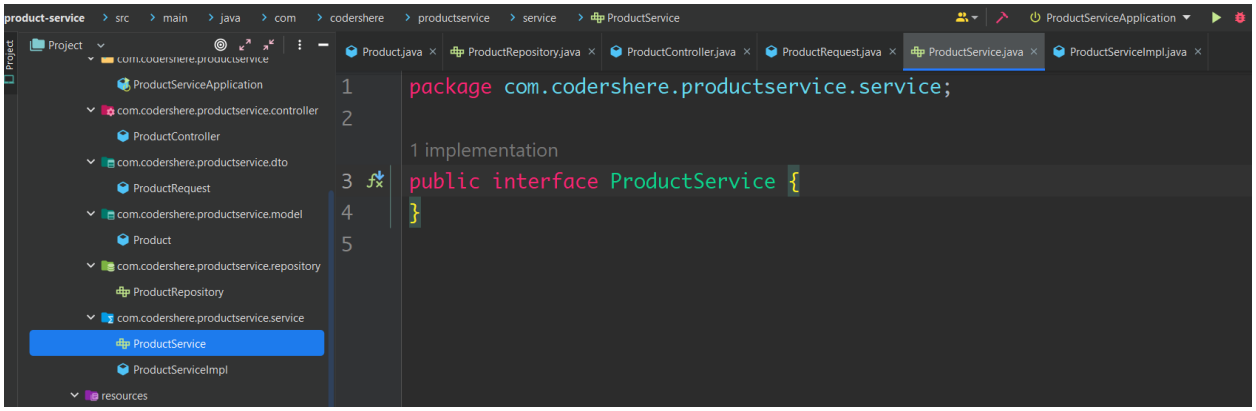
Lombok annotationlarını ekleyelim.

```
@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class ProductRequest {
}
```

Product class'ından id hariç diğer alanları kopyalayalım.

```
@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class ProductRequest {
    private String name;
    private String description;
    private BigDecimal price;
}
```

create işlemini yapmak üzere bir service package oluşturalım. İçine de service interface ve serviceImpl class'ı oluşturalım.



```java
package com.codershere.productservice.service;

import org.springframework.stereotype.Service;

@Service
public class ProductServiceImpl implements ProductService{
}
```

Daha önce öğrendiğimiz gibi controller’dan service metodunu çağırarak ilerleyelim.

```java
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/product")
public class ProductController {

    @Autowired
    private ProductService productService;

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public void createProduct(@RequestBody ProductRequest productRequest)
    {
        productService.createProduct(productRequest);
    }

}
```

Tamamladığımızda interface ve class şöyle olacak.

```java
1 implementation
public interface ProductService {
    1 implementation
    void createProduct(ProductRequest productRequest);
}
```

```java
@Service
public class ProductServiceImpl implements ProductService{
    @Override
    public void createProduct(ProductRequest productRequest) {

    }
}
```

Önce metodun içinde ProductRequest objesini, Product objesine çevirelim. (mapper da kullanabilirdik, lombok'un builder pattreniyle yapmayı tercih ettim.)

```java
@Service
public class ProductServiceImpl implements ProductService{
    @Override
    public void createProduct(ProductRequest productRequest) {
        Product product = Product.builder()
                .name(productRequest.getName())
                .description(productRequest.getDescription())
                .price(productRequest.getPrice())
                .build();

    }
}
```

ProductRepository kullanarak veritabanına kaydedelim.

```java
public class ProductServiceImpl implements ProductService{

    @Autowired
    private ProductRepository productRepository;

    @Override
    public void createProduct(ProductRequest productRequest) {
        Product product = Product.builder()
                .name(productRequest.getName())
                .description(productRequest.getDescription())
                .price(productRequest.getPrice())
                .build();

        productRepository.save(product);


    }
}
```

Slf4j ile loglayalım.

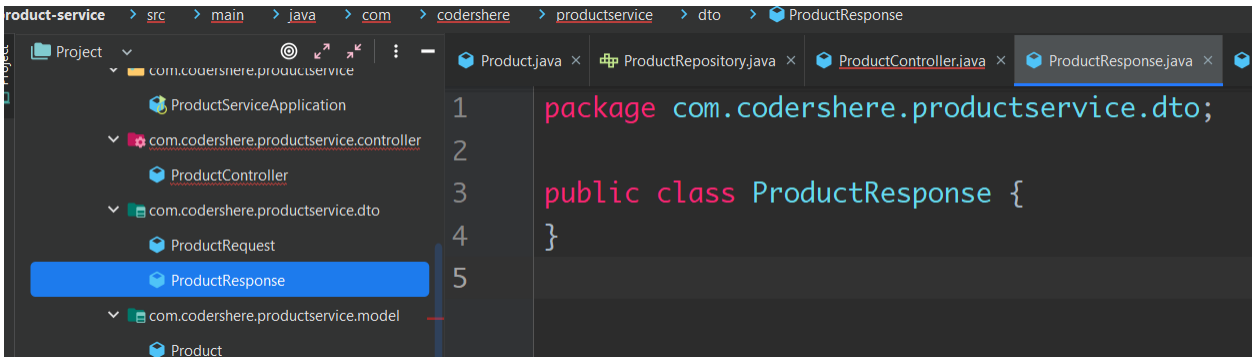
```java
@Service
@Slf4j
public class ProductServiceImpl implements ProductService{

    @Autowired
    private ProductRepository productRepository;

    @Override
    public void createProduct(ProductRequest productRequest) {
        Product product = Product.builder()
                .name(productRequest.getName())
                .description(productRequest.getDescription())
                .price(productRequest.getPrice())
                .build();

        productRepository.save(product);
        log.info("Product {} is saved", product.getId());
```

createProduct metodunu tamamlamış olduk.

Controller’da getAllProducts() metodunu yazalım.

```java
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public void createProduct(@RequestBody ProductRequest productRequest)
{
    productService.createProduct(productRequest);
}

@GetMapping
@ResponseStatus(HttpStatus.OK)
public List<ProductResponse> getAllProducts()
{

}
```

ProductResponse classını dto package da oluşturalım.



ProductRequest den annotationlarını, Product sınıfından da değişkenlerini kopyalayalım. (ProductRequest DTO'sundan farklı olarak bunun id değişkeni var.)

```
ProductServiceApplication
com.codershere.productservice.controller
ProductController
com.codershere.productservice.dto
ProductRequest
ProductResponse
com.codershere.productservice.model
Product
com.codershere.productservice.repository
ProductRepository
com.codershere.productservice.service
ProductService
ProductServiceImpl
resources
static
templates
application.properties
test
.gitignore
HELP.md
mvnw
mvnw.cmd
pom.xml
```

```java
package com.codershere.productservice.dto;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.math.BigDecimal;

@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class ProductResponse {
    private String id;
    private String name;
    private String description;
    private BigDecimal price;
}
```

ProductController’a dönüp metodun içini dolduralım.

```java
@GetMapping
@ResponseStatus(HttpStatus.OK)
public List<ProductResponse> getAllProducts()
{
    return productService.getAllProducts();
}
```

getAllProducts metodunu service classında oluşturalım.

```
1 implementation   1 related problem
public interface ProductService {
    1 implementation
    void createProduct(ProductRequest productRequest);

    List<ProductResponse> getAllProducts();
}
```

```
com.codershere.productservice.model
    Product
com.codershere.productservice.repository
    ProductRepository
com.codershere.productservice.service
    ProductService
    ProductServiceImpl
resources
    static
    templates
    application.properties
test
```

```
29          log.info("Product {} is saved", product.getId());
30
31
32      }
33
34      @Override
35      public List<ProductResponse> getAllProducts() {
36          return null;
37      }
38  }
```

Buraya dikkat, daha önce repository.findAll() deyip döndürüyorduk, bu sefer ProductResponse yani DTO döndürmemiz gerekecek.

Önce getAllProducts() metodunu şu hale çevirelim.

```
@Override
public List<ProductResponse> getAllProducts() {
    List<Product> products = productRepository.findAll();

    return products.stream().map(product -> mapToProductResponse(product)).toList();
}
```

Context action'la metodu oluşturalım.

```java
@Override
public List<ProductResponse> getAllProducts() {
    List<Product> products = productRepository.findAll();

    return products.stream().map(product -> mapToProductResponse(product)).toList();
}

private ProductResponse mapToProductResponse(Product product) {

}
```

Bu şekilde de metodu tamamlayalım.

```java
@Override
public List<ProductResponse> getAllProducts() {
    List<Product> products = productRepository.findAll();

    return products.stream().map(product -> mapToProductResponse(product)).toList();
}

private ProductResponse mapToProductResponse(Product product) {
    return ProductResponse.builder()
            .id(product.getId())
            .name(product.getName())
            .description(product.getDescription())
            .price(product.getPrice())
            .build();
}
```

Dikkat ederseniz stream map'deki metodu farklı renklendirmiş. Çünkü öneride bulunmak istiyor, üzerine gidelim.

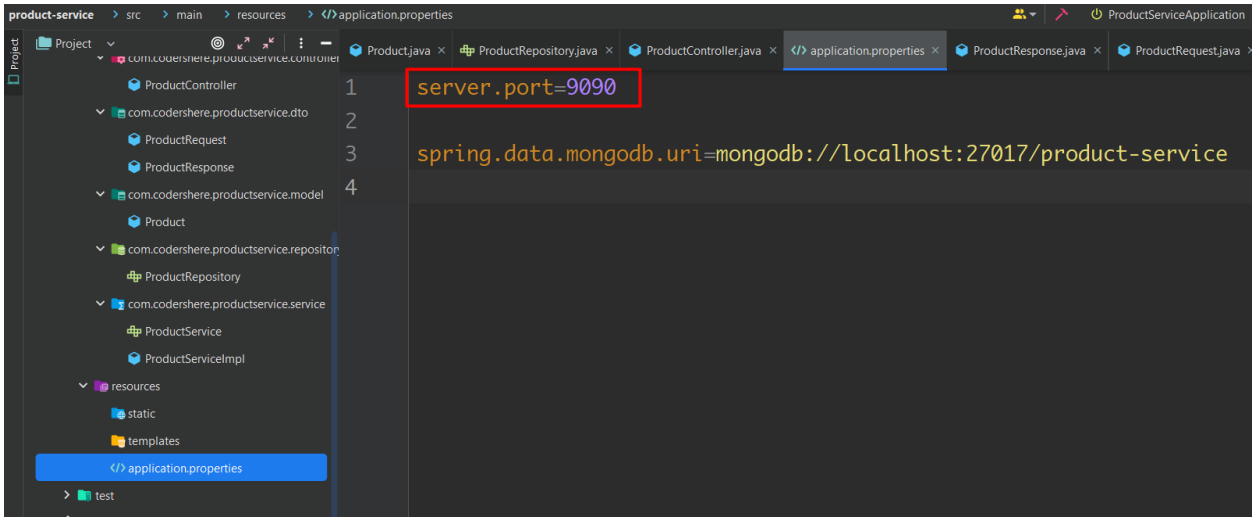
Uyarıdaki replace et önerisini kabul ettiğimde şuna çevirdi.
Ben böyle yazmazdım ama kabul ediyorum 🙂

```java
@Override
public List<ProductResponse> getAllProducts() {
    List<Product> products = productRepository.findAll();

    return products.stream().map(this::mapToProductResponse).toList();
}

private ProductResponse mapToProductResponse(Product product) {
    return ProductResponse.builder()
            .id(product.getId())
            .name(product.getName())
            .description(product.getDescription())
            .price(product.getPrice())
            .build();
}
```

getAllProducts metodunu da tamamladık.

Artık uygulamaya port verip başlatalım.

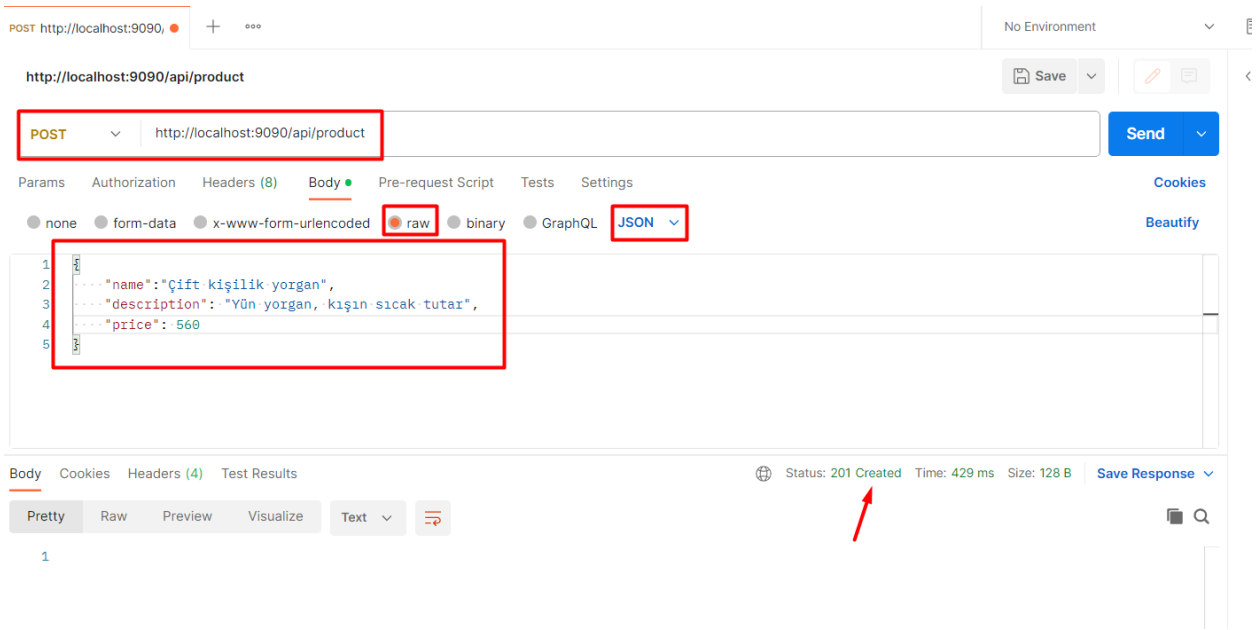
9090 portuyla başlatacağım.



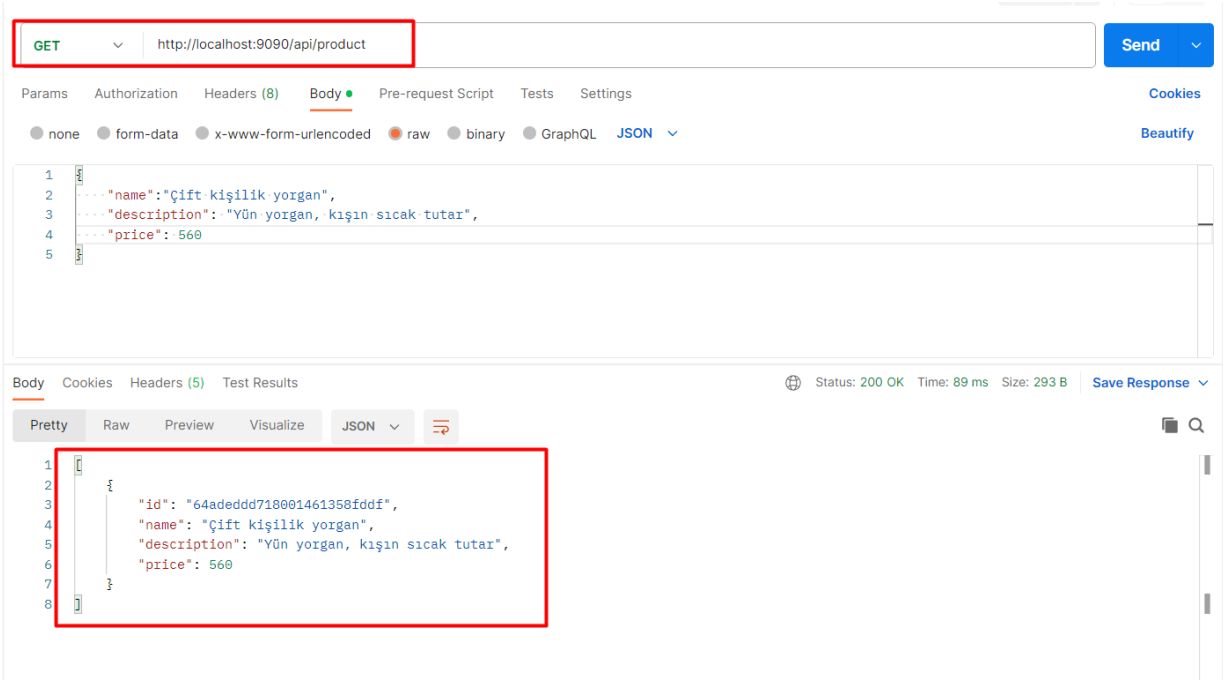
Başlatalım.

```
nsole    Actuator
.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 761 ms
.mongodb.driver.client                : MongoClient with metadata {"driver": {"name": "mongo-java-driver|sy
.mongodb.driver.cluster               : Monitor thread successfully connected to server with description Se
.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 9090 (http) with context path ''
.p.ProductServiceApplication          : Started ProductServiceApplication in 1.687 seconds (process running
```

Postman de create metodunu deneyelim.

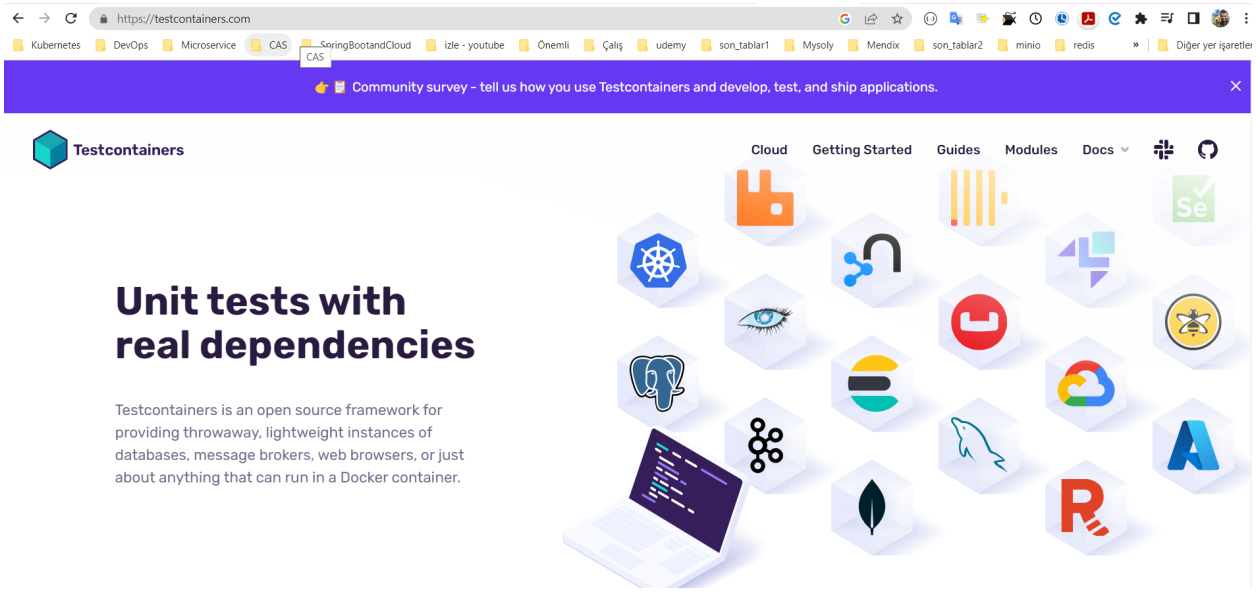


getAllProducts() ile de kontrol edelim.

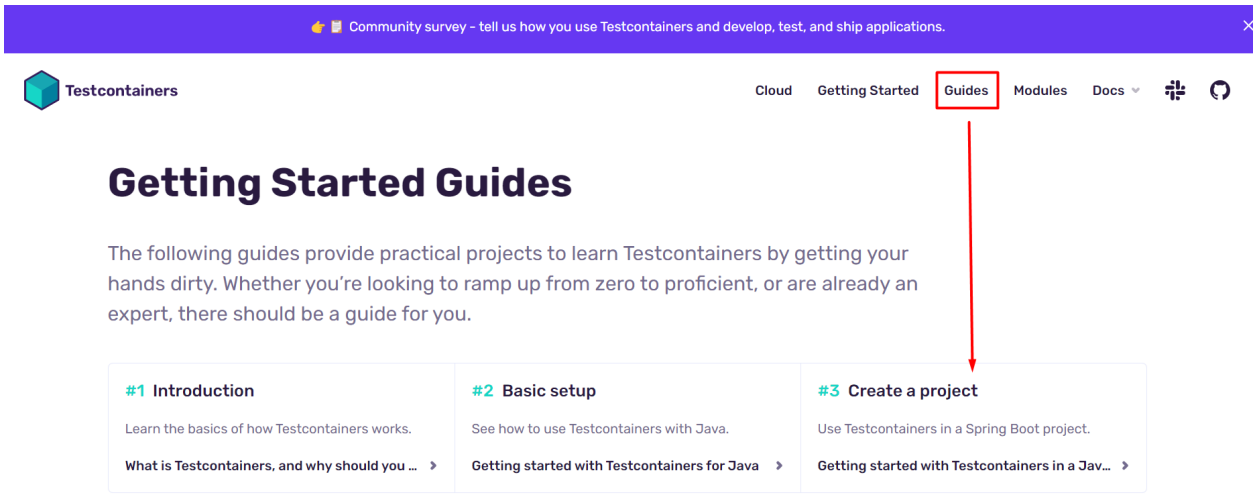


Integration Test

Mikroservislere integration test yazmak için TestContainers kullanacağız.



Guides -> Create a Spring Boot project bölümüne gidelim.



Öncelikle dependency management bölümünü pom.xml e dependencylerin altına kopyalayalım.

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.testcontainers</groupId>
            <artifactId>testcontainers-bom</artifactId>
            <version>${testcontainers.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

```
Product.java × | ProductRepository.java × | ProductController.java × | application.properties × | pom.xml (product-service) × | ProductServiceApplication.java

52
53
54    </dependencies>
55
56    <dependencyManagement>
57        <dependencies>
58            <dependency>
59                <groupId>org.testcontainers</groupId>
60                <artifactId>testcontainers-bom</artifactId>
61                <version>${testcontainers.version}</version>
62                <type>pom</type>
63                <scope>import</scope>
64            </dependency>
65        </dependencies>
66    </dependencyManagement>
```

versiyonu kopyalayalım.

```
<properties>
    <java.version>17</java.version>
    <testcontainers.version>1.18.0</testcontainers.version>
</properties>
```

```
<groupId>com.codershere</groupId>
<artifactId>product-service</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>product-service</name>
<description>product-service</description>
<properties>
    <java.version>17</java.version>
    <testcontainers.version>1.18.3</testcontainers.version>
</properties>
<dependencies>
```

MongoDB dependency sini kopyalayalım.

JDBC support
R2DBC support
Cassandra Module
CockroachDB Module
Couchbase Module
Clickhouse Module
CrateDB Module
DB2 Module
Dynalite Module
InfluxDB Module
MariaDB Module
MongoDB Module
MS SQL Server Module
MySQL Module

Add the following dependency to your `pom.xml` / `build.gradle` file:

Gradle **Maven**

```
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>mongodb</artifactId>
    <version>1.18.3</version>
    <scope>test</scope>
</dependency>
```

Hint

Adding this Testcontainers library JAR will not automatically add a database driver JAR to your project. You should ensure that your project also has a suitable database driver as a dependency

Copyright

```
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>org.testcontainers</groupId>
        <artifactId>mongodb</artifactId>
        <scope>test</scope>
    </dependency>

    <dependency>
```

JUnit dependency ekleyeceğiz.

Testcontainers for Java
Home
Quickstart
Features
Modules
Test framework integration
JUnit 4
Jupiter / JUnit 5
Spock
Manual container lifecycle control
External Integrations
Examples
System Requirements
Getting help
Contributing

Join the community

Jupiter / JUnit 5

While Testcontainers is tightly coupled with the JUnit 4.x rule API, this module provides an API that is based on the JUnit Jupiter extension model.

The extension supports two modes:

- containers that are restarted for every test method
- containers that are shared between all methods of a test class

Note that Jupiter/JUnit 5 integration is packaged as a separate library JAR; see below for details.

Extension

Jupiter integration is provided by means of the `@Testcontainers` annotation.

The extension finds all fields that are annotated with `@Container` and calls their container lifecycle methods (methods on the `Startable` interface). Containers declared as static fields will be shared between test methods. They will be started only once before any test method is executed and stopped after the last test method has executed. Containers declared as instance

Table of contents
Extension
Examples
Restarted containers
Shared containers
Singleton containers
Limitations
Adding Testcontainers JUnit 5 support to your project dependencies

```
    <dependency>
        <groupId>org.testcontainers</groupId>
        <artifactId>junit-jupiter</artifactId>
        <scope>test</scope>
    </dependency>


</dependencies>
```

dependency bölümü bitti.

Artık test bölümündeki Test classını açalım.

```
product-service > src > test > java > com > codershere > productservice > ProductServiceApplicationTests
Project
com.codershere.productservice.dto
    ProductRequest
    ProductResponse
com.codershere.productservice.model
    Product
com.codershere.productservice.repository
    ProductRepository
com.codershere.productservice.service
    ProductService
    ProductServiceImpl
resources
    static
    templates
    application.properties
test
    java
        com.codershere.productservice
            ProductServiceApplicationTests
target
ProductServiceApplicationTests.java
```

```java
1   package com.codershere.productservice;
2
3   import ...
5
6   @SpringBootTest
7   class ProductServiceApplicationTests {
8
9       @Test
10      void contextLoads() {
11      }
12
13  }
14
```

İlk olarak TestContainers annotationı ekleyelim.

```java
3   import ...
6
7   @SpringBootTest
8   @Testcontainers
9   class ProductServiceApplicationTests {
10
11      @Test
12      void contextLoads() {
13      }
14
15  }
16
```

MongoDBContainer oluşturalım ve bir property set edelim.

```
@SpringBootTest
@Testcontainers
class ProductServiceApplicationTests {

    @Container
    static MongoDBContainer mongoDBContainer =
            new MongoDBContainer( dockerImageName: "mongo:4.4.2");

    static void setProperties(DynamicPropertyRegistry dpr)
    {
        dpr.add( name: "spring.data.mongodb.uri",mongoDBContainer::getReplicaSetUrl);
    }

    @Test
```

Bu ayarlamaları bitirdikten sonra, createProduct metodu için entegrasyon testimizi yazalım.

Alttaki test metodunun(contextLoads) adını değiştirelim.

```
    static void setProperties(DynamicPropertyRegistry dpr)
    {
        dpr.add( name: "spring.data.mongodb.uri",mongoDBContainer::getReplicaSetU
    }

    @Test
    void shouldCreateProduct() {

    }
```

MockMvc objesi inject edelim.

```
import ...

@SpringBootTest
@Testcontainers
@AutoConfigureMockMvc
class ProductServiceApplicationTests {

    @Autowired
    private MockMvc mockMvc;

    @Container
    static MongoDBContainer mongoDBContainer =
            new MongoDBContainer( dockerImageName: "mongo:4.4.2");
```

Önce MockMvc perform metodunu yazalım.

```
@Test
void shouldCreateProduct() {

    mockMvc.perform(MockMvcRequestBuilders.post( urlTemplate: "/api/product")
            .contentType(MediaType.APPLICATION_JSON)
            .content()

    )


}
```

content metodunun içine ProductRequest objesi gitmesi lazım, yukarıda oluşturalım.

```java
ProductRequest productRequest = getProductRequest();

mockMvc.perform(MockMvcRequestBuilders.post( urlTemplate: "/api/product")
        .contentType(MediaType.APPLICATION_JSON)
        .content()

);

}

private ProductRequest getProductRequest() {
    return ProductRequest.builder()
            .name("Makarna")
            .description("500g fiyonk")
            .price(BigDecimal.valueOf(4.5))
            .build();
}
```

content sadece String kabul ettiği için ProductRequest objesini string'e çevirmemiz gerekir.

```java
@AutoConfigureMockMvc
class ProductServiceApplicationTests {

    @Autowired
    private MockMvc mockMvc;

    @Autowired
    private ObjectMapper objectMapper;

    @Container
    static MongoDBContainer mongoDBContainer =
            new MongoDBContainer( dockerImageName: "mongo:4.4.2");
```

```
@Test
void shouldCreateProduct() throws Exception {

    ProductRequest productRequest = getProductRequest();
    String productRequestStr = objectMapper.writeValueAsString(productRequest);

    mockMvc.perform(MockMvcRequestBuilders.post( urlTemplate: "/api/product")
            .contentType(MediaType.APPLICATION_JSON)
            .content(productRequestStr))
            .andExpect(status().isCreated());

}
```

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

Bir de gelen kayıtların size'ını kullanarak testi genişletelim.

```
@Autowired
private ObjectMapper objectMapper;

@Autowired
private ProductRepository productRepository;

@Container
static MongoDBContainer mongoDBContainer =
```

```
mockMvc.perform(MockMvcRequestBuilders.post( urlTemplate: "/api/product")
        .contentType(MediaType.APPLICATION_JSON)
        .content(productRequestStr))
        .andExpect(status().isCreated());

Assertions.assertEquals( expected: 1, productRepository.findAll().size());
```

Testi tekrar çalıştıralım.

Planlarımda yoktu ama kayıtlar sürekli biriktiği için şunu eklemek zorunda kaldım.

```
@Autowired
private ProductRepository productRepository;

@AfterEach
void cleanUp() {
    this.productRepository.deleteAll();
}

@Container
```

Her testten sonra test containerındaki product kayıtlarını siliyor.

Bu aşamada testlerimiz başarılı.

```
Run: ProductServiceApplication × ProductServiceApplicationTests ×
Tests passed: 1 of 1 test – 1 sec 621 ms
ProductServiceApplicationTests (co 1 sec 621 ms
shouldCreateProduct() 1 sec 621 ms
pported for boot loader classes because bootstrap classpath has been appended
n] c.c.p.service.ProductServiceImpl        : Product 64ae32336487676a11f3e324 is saved
```

Order Microservice