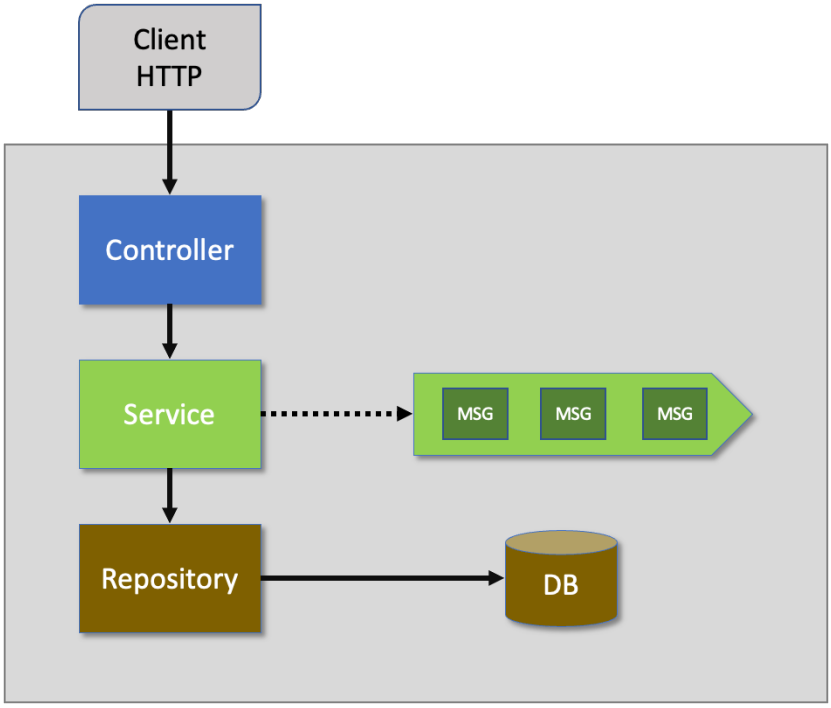
Hands On Exercises

Introduction to Spring Cloud

Lab 1 Create Product Service

We will use start.spring.io along with Spring Boot and Spring Cloud modules to quickly and easily create the Product Service model. All services will use a very similar architecture to implement the services. The following diagram illustrates the core pieces.



All services have a controller, service and a repository layer. The controller layer receives the HTTP request from service clients. The actual business logic is implemented in the service layer. Some services need to communicate with a message queue with other services (order and notification service). Each of the services stores its information in a database. The repository layer will be responsible for this and communicates with a DB. The actual database may differ amongst services.

1. Setup Lab Docker Container Environment

   1.1 Make sure that you have completed Lab 1 to setup the docker environment with docker-composite.yml. If you have not done so, ask instructor to provide you with a Zip file that you can use to get started.

   1.2 Verify that the lab containers are all up and running

      1.2.1 User `docker ps` to check

2. Create Maven project from https://start.spring.io

2.1 Setup

2.1.1 Project > Maven project

2.1.2 Language > Java

2.1.3 Spring Boot > 3.1.0 (or current version, just do not select a SNAPSHOT)

2.2 Project Meta

2.2.1 Group > com.example

2.2.2 Artifact > product-service

2.2.3 Name > product-service

2.2.4 Description > as you wish

2.2.5 Package name: com.example.productservice (make sure to remove the hyphen)

2.2.6 Packaging > Jar

2.2.7 Java > 17

2.3 Dependencies to add

2.3.1 Spring Web

2.3.2 Lombok

2.3.3 Spring Data MongoDb

2.4 [Generate] zip file

2.5 Create a working folder (parent folder) and unzip above zip file inside the parent folder.

3. Open project in IDE

3.1 Open Eclipse Enterprise or Spring Tools Suite 4

3.2 Import Existing Maven project as folder and point to the working folder above

4. Setup MongoDB model

4.1 In application.properties add:

```
spring.data.mongodb.uri=mongodb://admin:password@localhost:27017/
product-service?authSource=admin
```

4.2 Create the Product model definition for mongodb

4.2.1 Create com.example.productservice.model package

4.2.2 Create Java class 'Product' in above package

4.3 Setup Product class with annotations

4.3.1 Define class as a mongodb document, add the @Document annotation with value="product" as parameter

4.3.2 Add Lomboc annotations to generate the constructors, builder and setters/getters

```
@AllArgsConstructor
@NoArgsConstructor
@Builder
@Data
```

4.4 Add private class fields

4.4.1 String id

(a) Annotate this field with @Id to indicate that this is a unique identifier field for product (Very similar to primary key field)

4.4.2 String name

4.4.3 String description

4.4.4 BigDecimal price

4.5 Your code so far should look similar to below:

```
package com.example.productservice.model;

import java.math.BigDecimal;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

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

@Document(value = "product")
@AllArgsConstructor
@NoArgsConstructor
@Builder
@Data
1 reference
public class Product {
    @Id
    private String id;
    2 references
    private String name;
    private String description;
    private BigDecimal price;

}
```

5. Create Spring Data repository interface

5.1 Create new package com.example.productservice.repository

5.2 Create new Java interface and name it 'ProductRepository'

5.3 The new interface extends the MongoRepository

5.4 The new interface works with our model Product and the Id is a String class.

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

6. Create the Controller layer – createProduct()

6.1 Create new package com.example.productservice.controller

6.2 Create new Java class and name it 'ProductController'

6.3 Annotate the class to indicate that this is a REST controller with the request mapping at "/api/product"

6.4 Create public void createProduct() method

6.4.1 This method is a POST method so annotate with @PostMapping

6.4.2 We want the POST to return the status 'Created' on success. Use @ResponseStatus annotation with HttpStatus.CREATED as the returned status

6.4.3 The method will process a request body with class ProductRequest.

```
public void createProduct(@RequestBody ProductRequest productRequest)
```

6.4.4 ProductRequest is a class that we will have to create. In Eclipse, we can have the IDE create the class for us. Simply hover the mouse over ProductRequest or CTRL(CMD)-1 it and select the option to create the class. We will create the class in package com.example.productservice.dto (data transfer object)

6.5 Define class ProductRequest

6.5.1 Use the Lomboc annotations as before to create the builder object, setters and getters and constructors

6.5.2 Inside the class, we will have 3 fields similar to the Product class.

(a) String name

(b) String description

(c) BigDecimal price

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

import java.math.BigDecimal;

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

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

}
```

We have created a POST mapping method that accepts a request body of class ProductRequest. We have defined the ProductRequest class. We still have to define the actual business logic for the POST method createProduct(). However, we should define this in the service layer. We do not want to define this is the controller layer.

7. Create the Service layer – createProduct()

   7.1 Create new package com.example.productservice.service

   7.2 Create new Java class and name it 'ProductService'

   7.3 Since this is a service layer for the business logic, annotate with @Service

   7.4 Create the createProduct() method which receives a ProductRequest class as its parameter

   7.5 Create a new Product class that we defined in the model package and use its builder() method to fill in the name, description and price from the ProductRequest object.

```
Product product = Product.builder()
        .name(productRequest.getName())
        .description(productRequest.getDescription())
        .price(productRequest.getPrice())
        .build();
```

   7.6 Now that we have built a new Product, it needs to be saved in the Product repository. We will need to inject a ProductRepository object and use it's save() method to accomplish this. Start by defining a new ProductRepository.

```
private final ProductRepository productRepository;
```

You will notice that Eclipse is complaining that productRepository has not been initialized. We can fix the problem by constructing a ProductService with the code as shown below. Try adding the following code. You should see that the problem is resolved.

```
public ProductService(ProductRepository productRepository) {
     this.productRepository = productRepository;
}
```

While we could actually create this constructor, there is an easier solution. The @RequiredArgsConstructor annotation from Lomboc creates a constructor for an uninitialized final field. Remove the above constructor code and add the @RequiredArgsConstructor annotation to the ProductService class.

It is possible to use @Autowired as well. Using @Autowired is called field injection, while using @RequiredArgsConstructor is called constructor injection.

7.7 To complete the createProduct() method, we can now save the newly created Product product using the ProductRepository's save() method.

```
productRepository.save(product);
```

7.8 Add some log information that the product has been saved at the INFO level. We will use the @Slf4j from Lombok along with the log.info("log message") for this.

7.8.1 Add @Slf4j to the ProductService class

7.8.2 Inside the createProduct() method, after the product has been saved, add

```
log.info("Product {}has been saved", product.getId())
```

7.9 You code should look similar to below:

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

import org.springframework.stereotype.Service;

import com.example.productservice.dto.ProductRequest;
import com.example.productservice.model.Product;
import com.example.productservice.repository.ProductRepository;

import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;

@Service
@RequiredArgsConstructor
1 reference
@Slf4j
public class ProductService {

    1 reference
    private final ProductRepository productRepository;

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

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

}
```

We are now ready to use the createProduct() method we just created in the Service module from the Controller module. Return to ProductController.java and inject ProductService. We can then use the productService.createProduct() method.

7.10 In ProductController.java, inject ProductService using the constructor injection method

7.11 In the createProdect() method, use the injected ProductService.createProduct() method to actually create a mongodb record of a new product. The parameter to the createProduct() is obtained from the ProductRequest request body.

```
productService.createProduct(productRequest);
```

We have now finalized creating an API endpoint (/api/product) to add a product to the product database. You code should look similar to below:

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

import java.util.List;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

import com.example.productservice.dto.ProductRequest;
import com.example.productservice.dto.ProductResponse;
import com.example.productservice.service.ProductService;

import lombok.RequiredArgsConstructor;

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

    private final ProductService productService;

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

8. Add ProductResponse Endpoint

8.1 Create ProductResponse DTO (data transport object)

8.1.1 The ProductReponse dto will have the same structure as our Product model.

You may wonder why we are creating another ProductResponse dto, when we have the ProductRequest dto as well as the actual Product model itself. Generally, we never want to expose our database model to the outside world. Each interaction with the outside world should be handled exclusively by dto handlers. We also want to make sure that we create a separate dto for each type of interaction. This allows us to change/modify our internal model in response to business change requirements, without having to modify the transport mechanisms exposed to the outside world.

8.2 Create getAllProducts() GET controller

8.2.1 Use @GetMapping

8.2.2 The @ResponseStatus should return HttpStatus.OK

8.2.3 Create getAllProducts() that returns a List of ProductResponse

8.2.4 The method should call getAllproducts() from ProductService class. We have already created productService previously, in order to service the createProduct.

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

8.2.1 Create getAllProducts() method to incorporate the business logic in the ProductService module. Currently, Eclipse will be complaining that productService.getAllProducts() does not exist. If you hover your mouse or CTRL-1, you will be given the option to create the method. Go ahead and use this to quickly get started.

8.2.2 From the getAllProducts(), make sure the method returns a List<ProductResponse>

```
public List<ProductResponse> getAllProducts()
```

8.2.3 Use productRepository's findall() method to get a list of all the products. Save it to variable, List<Product> products

We now have to map the List<Product> to List<ProductResponse> in order to pass back to the GET getAllProducts() controller. We will use Java's functional programming capabilities to map Product to ProductResponse. To use this capability, we first take products and stream() it. Then, we can use map(`mapToProductResponse`) to apply `mapToProductResponse` to every element in prouducts.

8.2.4 Add the following code:

```
products.stream().map(p -> mapToProductResponse(p));
```

8.2.5 We have not defined mapToProductResponse() function yet. We can use the builder method to build it. From Eclipse, hover on the mapping function and ask to create a local method.

8.2.6 mapToProductResponse(p) should be similar to code below:

```
    private ProductResponse mapToProductResponse(Product product)
{
        return ProductResponse.builder()
                .id(product.getId())
                .name(product.getName())
                .description(product.getDescription())
```

```
                .price(product.getPrice())
                .build();
    }
```

8.2.7 Finally, we want to collect each of the build ProductResponse into a List and return it. Your final code should look similar to:

```
public List<ProductResponse> getAllProducts() {
    // Use ProductRepository interface to get all products
    List<Product> products = productRepository.findAll();
    return products.stream().map(p -> mapToProductResponse(p)).toList();
}
```

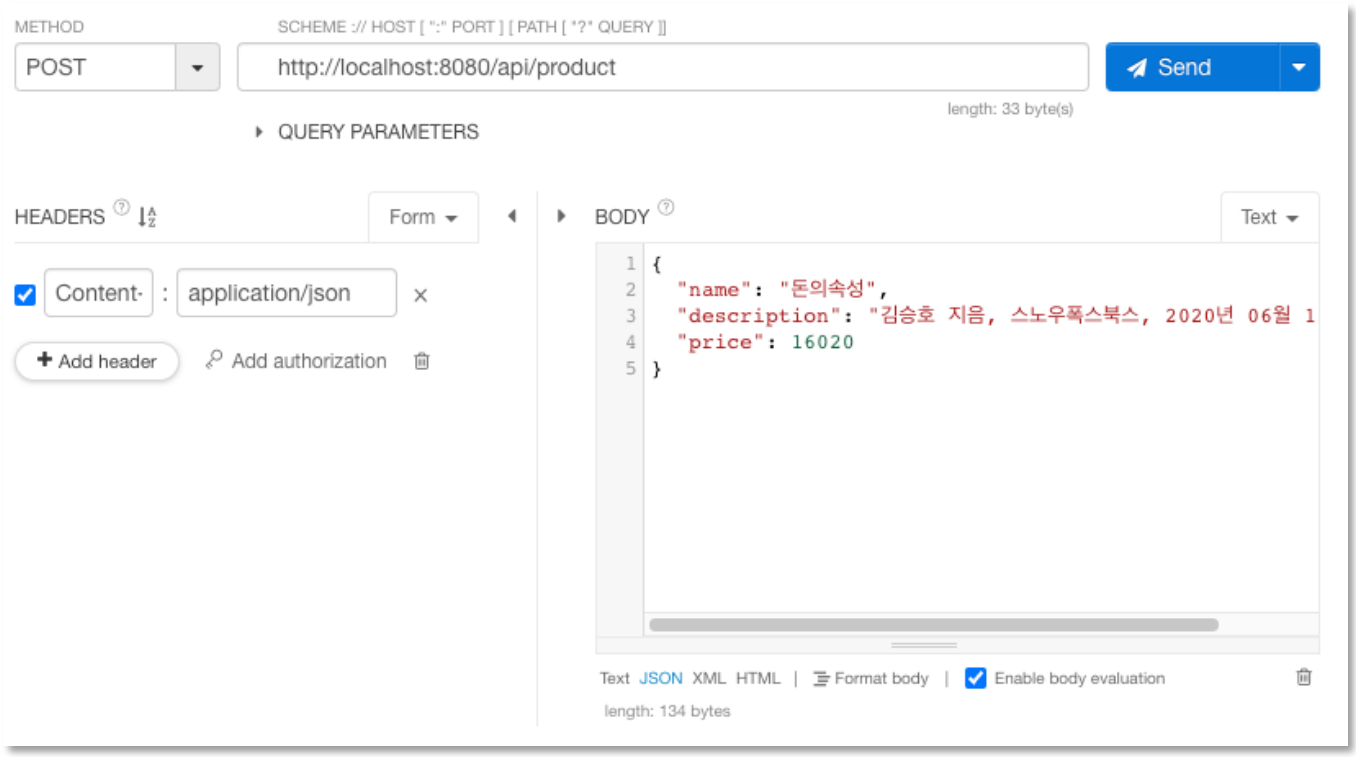
9. Test product-service module

9.1 Using Talend or Postman, POST the following raw JSON body

9.1.1 On Talend (Postman), navigate to http://localhost:8080/api/product

9.1.2 Use the POST method and add the following in the body

```
{
  "name": "돈의속성",
  "description": "김승호 지음, 스노우폭스북스, 2020 년 06 월 15 일 출간",
  "price": 16020
}
```



9.2 View the newly added record by issuing a GET command

9.2.1 On your browser, navigate to http://localhost:8080/api/product