5. Service

先前提到 Controller 應該只負責處理請求跟回應而已，所以這邊要將上章節做的東西抽離出來，把對資料的操作放在 Service 層中：

```
|--com.example.demospringboot
   |--DemospringbootApplication.java
|--com.example.demospringboot.configuration
   |--SwaggerConfig.java
|--com.example.demospringboot.controller
   |--TestController.java
   |--ProductController.java // 修改的檔案
|--com.example.demospringboot.model
   |--Product.java
|--com.example.demospringboot.service
   |--ProductService.java // 新增的檔案
|--com.example.demospringboot.service.impl
   |--ProductServiceImpl.java // 新增的檔案
```

```
public interface ProductService {
    public abstract void createProduct(Product product);

    public abstract void updateProduct(String id, Product product);

    public abstract void deleteProduct(String id);

    public abstract Collection<Product> getProducts();
}
```

```java
@Service
public class ProductServiceImpl implements ProductService {
    private static Map<String, Product> productRepo = new HashMap<>();
    static {
        Product honey = new Product();
        honey.setId("1");
        honey.setName("Honey");
        productRepo.put(honey.getId(), honey);

        Product almond = new Product();
        almond.setId("2");
        almond.setName("Almond");
        productRepo.put(almond.getId(), almond);
    }

    @Override
    public void createProduct(Product product) {
        productRepo.put(product.getId(), product);
    }

    @Override
    public void updateProduct(String id, Product product) {
        productRepo.remove(id);
        productRepo.put(product.getId(), product);
    }


    @Override
    public void deleteProduct(String id) {
        productRepo.remove(id);
    }

    @Override
    public Collection<Product> getProducts() {
        return productRepo.values();
    }
}
```

```java
@RestController
public class ProductController {
    @Autowired
    private ProductService productService;

    @RequestMapping(value = "/productsSev/{id}", method = RequestMethod.DELETE)
    public ResponseEntity<Object> deleteSev(@PathVariable("id") String id) {
        productService.deleteProduct(id);
        return new ResponseEntity<>("Product is deleted successsfully", HttpStatus.OK);
    }

    @RequestMapping(value = "/productsSev/{id}", method = RequestMethod.PUT)
    public ResponseEntity<Object> updateProductSev(@PathVariable("id") String id
            , @RequestBody Product product) {
        productService.updateProduct(id, product);
        return new ResponseEntity<>("Product is updated successsfully", HttpStatus.OK);
    }

    @RequestMapping(value = "/productsSev", method = RequestMethod.POST)
    public ResponseEntity<Object> createProductSev(@RequestBody Product product) {
        productService.createProduct(product);
        return new ResponseEntity<>("Product is created successfully", HttpStatus.CREATED);
    }

    @RequestMapping(value = "/productsSev", method = RequestMethod.GET)
    public ResponseEntity<Object> getProductSev() {
        return new ResponseEntity<>(productService.getProducts(), HttpStatus.OK);
    }
}
```

這邊目的只是抽離業務邏輯而已。來看 Service 的部分，我們建立一個拿來放 Service Interface 的 package，裡頭建了 `ProductService.java` ，只要定義好使用的方法即可。再來建立一個 Service Implement 的 package，在裡頭建立 `ProductServiceImpl.java` 來實作 `ProductService.java` ，注意上面有個 annotation `@Service` ，目的是將 class 標成一個 `@Component` ，讓 Spring Boot 啟動時可以用 `@ComponentScan` 掃描到，所以寫 `@Component` annotation 其實也可以。

下一步是在本來的 `ProductController.java` 注入 Service，讓我們可以直接操作 Service 內的方法，注入我們採用 `@Autowired` 這個 annotation，他可以把一開始放在 Spring Boot Bean Pool 裡面的 Bean 拿出來。

Spring Boot 對物件的預設，基本上都是 Singleton，不管 request 進來幾次，使用 `@Autowired` 拿到的物件都會是同一個，有興趣的人可以自行嘗試看看發送兩次 request，第一次 request 改變物件裡面的值，第二次 request 查詢物件裡面的值是否有所改動。

> 可以讓 Spring Boot 不要是 Singleton 嗎？ 可以
> 如果有兩個 class 實作 `ProductServiceImpl.java` 會有問題嗎？ 所以需要加上額外的annotation做指定

参考

https://www.baeldung.com/spring-bean-scopes
https://www.baeldung.com/spring-qualifier-annotation