Spring Boot & Spring DATA JPA

YuShangLee

Agenda

- Spring boot 是什麼? 有什麼特性?
- 建置第一個 Spring boot 專案
- SpringBootApplication 運作流程
- Spring boot Swagger
- Spring boot Bean Lifecycle
- Spring boot Bean Scope
- Spring boot IOC (Inversion of Control)
- Spring boot AOP (Aspect Oriented Programming)
- Spring boot RestController
- Spring boot restful Request、Response
- Spring Data JPA
- Spring Data JPA DB multiple datasource
- Spring Data JPA DB Entity
- Spring Data JPA Query methods
- Spring Data JPA JPQL
- Spring Data JPA Criteria Queries
- Spring Data JPA DB DML operations
- Spring Data JPA Batch Inserts
- Spring Data JPA Entity Lifecycle
- Spring Data JPA Data Locking
- Spring Data JPA Transaction Manager

Spring 是什麼

Spring Framework 是一個「輕量級」(Lightweight)的容器(Container) 的框架。

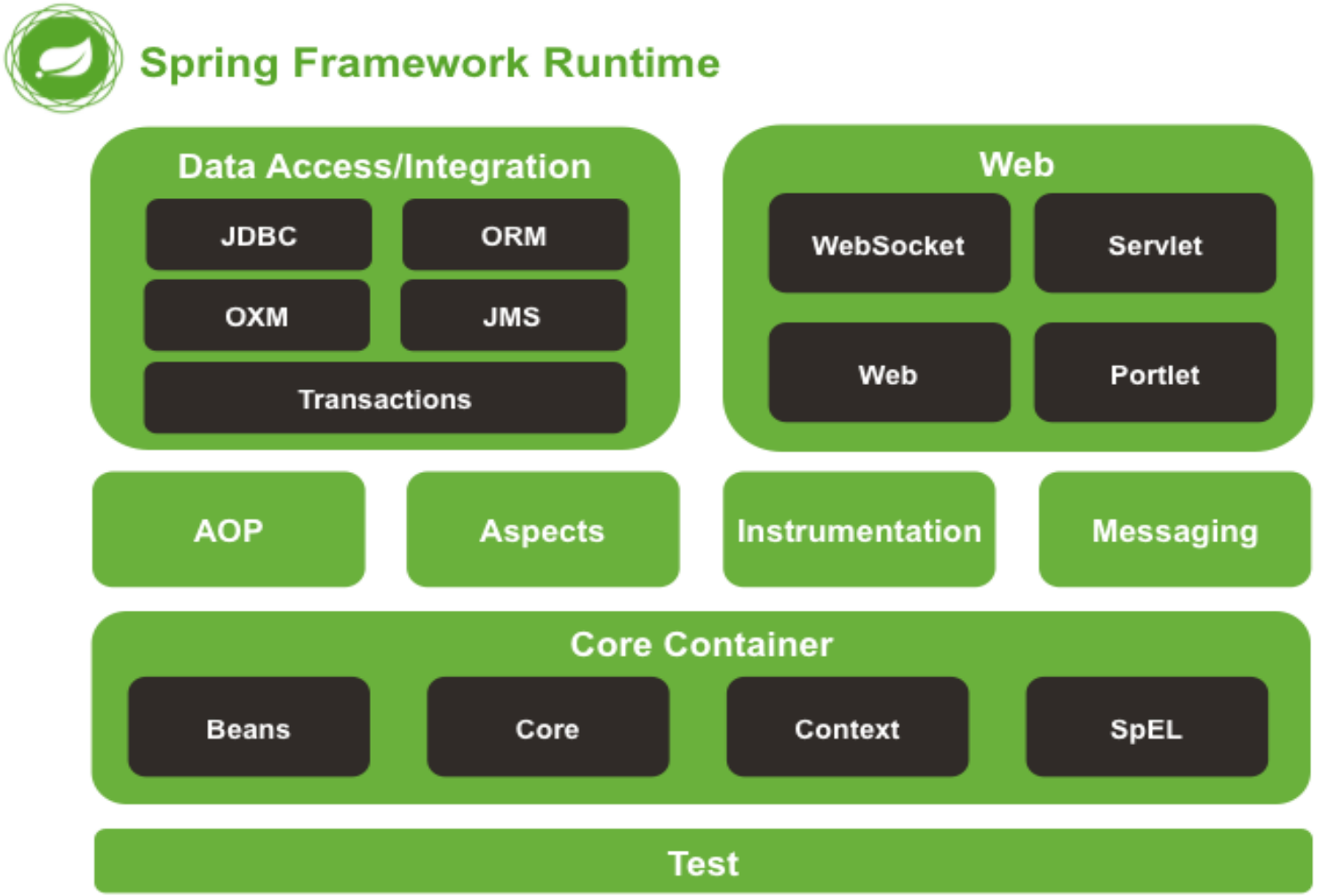
Java 的世界中，除了標準的Java SDK(一般稱之為J2SE)外，另外還有一個主要的分支，J2EE(Java 2 Enterprise Edition)，該分支的主要目的在於支援大型組織及企業的分散式運算。J2EE 主要的精神在於將企業邏輯與系統服務區分開來，而其採取的策略就是：在程式設計階段，不考慮系統服務(將系統服務都視為可以取用的資源，在程式中，利用JNDI 來取得這些資源)。但J2EE因其複雜性及必須先安裝在特定的環境中，然後利用部署機制來運作，這種類型的Container 稱之為「重量級容器」(Heavy Weight Container)。這種繁複的系統配置，也造成了開發人員的困擾。

反觀Spring Framework 的結構設計，所有的企業服務都利用POJO(Plain Old Java Object)來進行設計，在實作上，不包含任何除了標準Java 語言外其他的平台限制，而當其他的物件(無論是Web 的物件或是其他的Client 物件)要去使用該服務時，只要透過使用者定義的企業介面(Business Interface)來進行存取就可以了。

至於跟後端的企業資源溝通，也都可以透過標準的OR Mapping (Object- Relationship Mapping)機制就可以了，其中所使用到的，也都是POJO 的Java Bean。由於使用POJO 的關係，企業邏輯與平台(這裡指的是實際運作的平台)沒有必然的關係，你可以任意選擇要將你的程式部署在單純的Web Container(如Tomcat)，或是標準的J2EE Server(如Weblogic, Websphere 或是JBoss)，甚或是單獨運作的JVM 上。由於企業邏輯的程式具備了可攜性，基於Spring Framework 所實作出來的程式也具備了可攜性，這也是為什麼稱Spring Framework 是一個「輕量級」容器的緣故。

Spring 是什麼

Spring Framework 的應用程式模組則如下圖所示



Spring 的生態圈

Spring 發展到今天已經不僅僅是 Spring 框架本身的內容了， Spring 目前提供了非常多的基於 Spring 的項目，可以用來更加快捷地開發專案

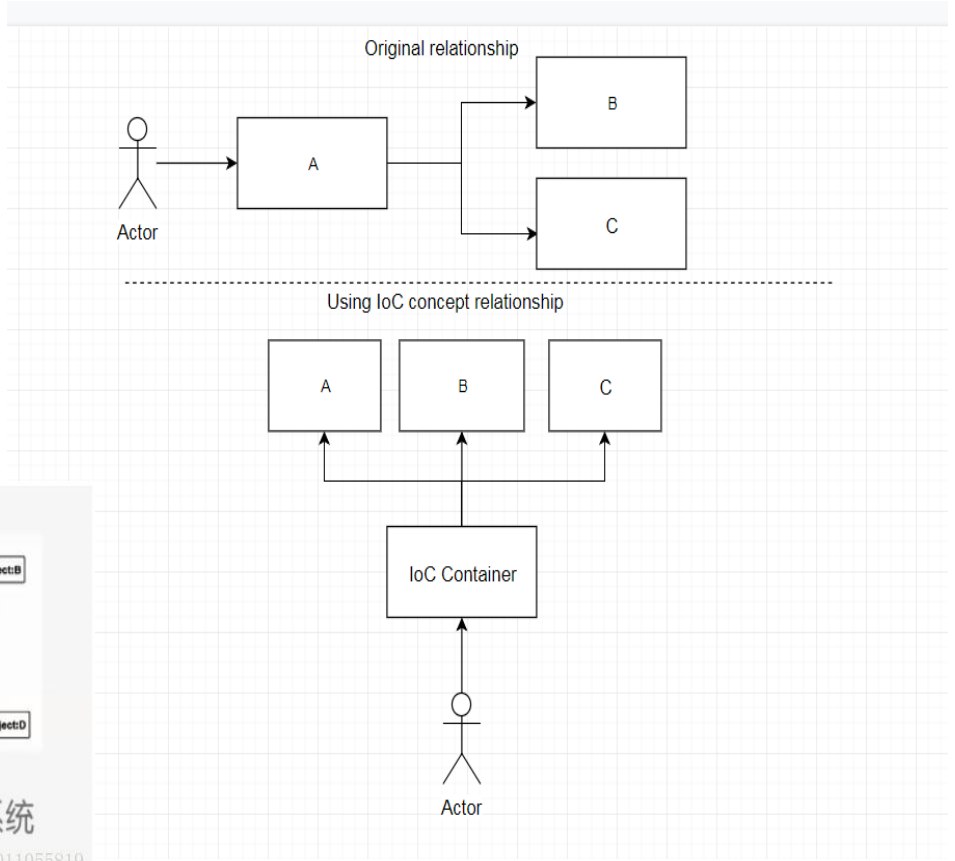
Spring 目前主要有以下專案

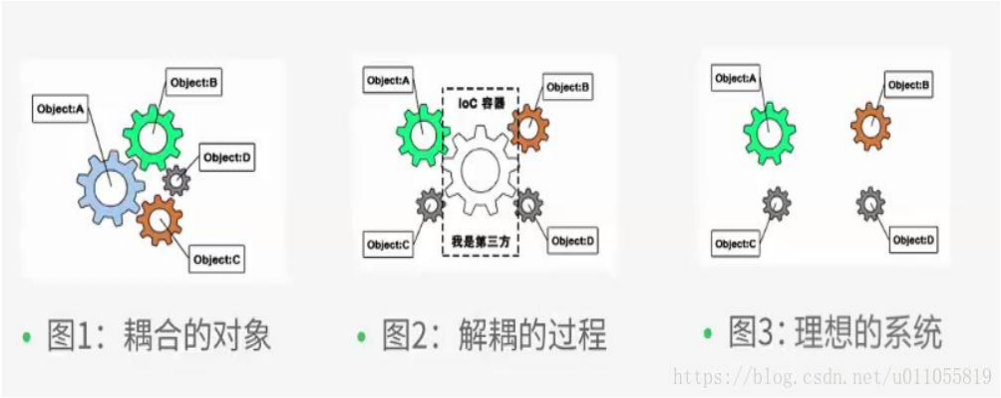
- Spring Boot ：使用默認 Java 配置來實現快速開發
- Spring XD ：用於簡化大資料應用的開發
- Spring Cloud ：為分散式系統開發提供工具集。
- Spring Data ：對主流的關係型數據庫和 NoSQL 資料 提供支援
- Spring Integration ：通過消息機制對企業集成模式提供支援。
- Spring Batch ：簡化及優化大資料的批次處理操作
- Spring Security ：通過認證和 權保護應用
- Spring REST ：簡化 Rest 服務開發
- Spring Social ：與社交網路 API （如 Facebook 新浪微博等）的集成
- Spring AMQP ：對 AMQP 的消息的支援
- Spring Mobile ：提供對手機設備進行檢測的功能，給不同的設備返回不同的支援
- Spring for Android ：提供在 Android 上消費 RESTful 的功能
- Spring Web Flow ：基於 Spring MVC 提供的 Web 應用開發。
- Spring Web Services ：提供基於協議有限的 SOAP/Web 服務
- Spring LDAP ：簡化使用 LDAP 開發
- Spring Session ：提供 API 及實現來管理使用者會話資訊

Spring 如何運作

- Spring 最成功的是其提出的理念，而不是技術本身，Spring Framework 依賴的2個核心理念，
  - ✓ 一個是採用所謂控制反轉(Inversion of Control; IoC)的實作技術，來實現其追求輕量化的結構目標。 IoC container是 Spring的核心，可以說 Spring 是一種基於 IoC container 開發的框架。
  - ✓ 另一個是AOP (Aspect Oriented Programming)。

控制反轉是一個設計思想 ，把對於某個物件的控制權移轉給第三方容器，簡單解釋
A物件程式內部需要使用B物件 A, B 物件中有依賴的成份
控制反轉是把原本 A 對 B 控制權移交給第三方容器
降低 A 對 B 物件的耦合性，讓雙方都倚賴第三方容器。

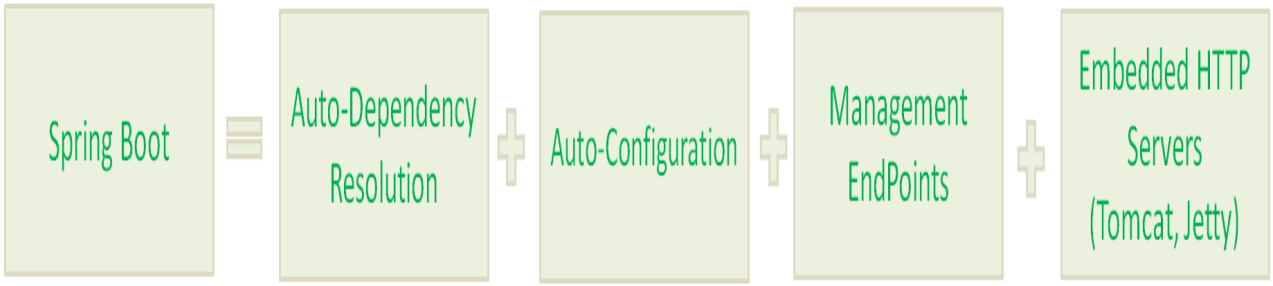




Spring Boot是什麼

Spring Boot 是由 Pivotal 團隊提供的全新框架，該框架使用了特定的方式來進行配置，從而使開發人員不再需要定義樣板化的配置。Spring Boot 預設配置了很多框架的使用方式，就像 Maven 整合了所有的 Jar 包，Spring Boot 整合了所有的框架。

Spring Boot 簡化了 Spring 應用開發，不需要配置就能運行 Spring 應用，Spring Boot簡化了管理 Spring 容器、協力廠商外掛程式 並提供很多預設系統級的服務。大部分Spring的應用開發，無論是簡單的系統還是構建複雜系統，都只需要少量配置和代碼就能完成。



What is Spring Boot?

How Does Spring Boot Work?

Java

Main method entry point

Spring Application

Spring context

Spring environment

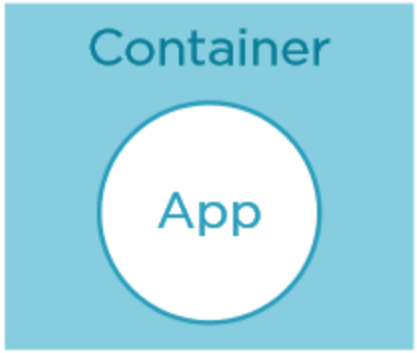
Initializers

Embedded Server

Default is Tomcat

Auto configured

Why Move to Containerless Deployments?



Container Deployments

- Pre-setup and configuration
- Need to use files like web.xml to tell container how to work
- Environment configuration is external to your application

Application Deployments

- Runs anywhere Java is setup (think cloud deployments)
- Container is embedded and the app directs how the container works
- Environment configuration is internal to your application


SPRING FRAMEWORK


SPRING BOOT

Spring Boot 特性

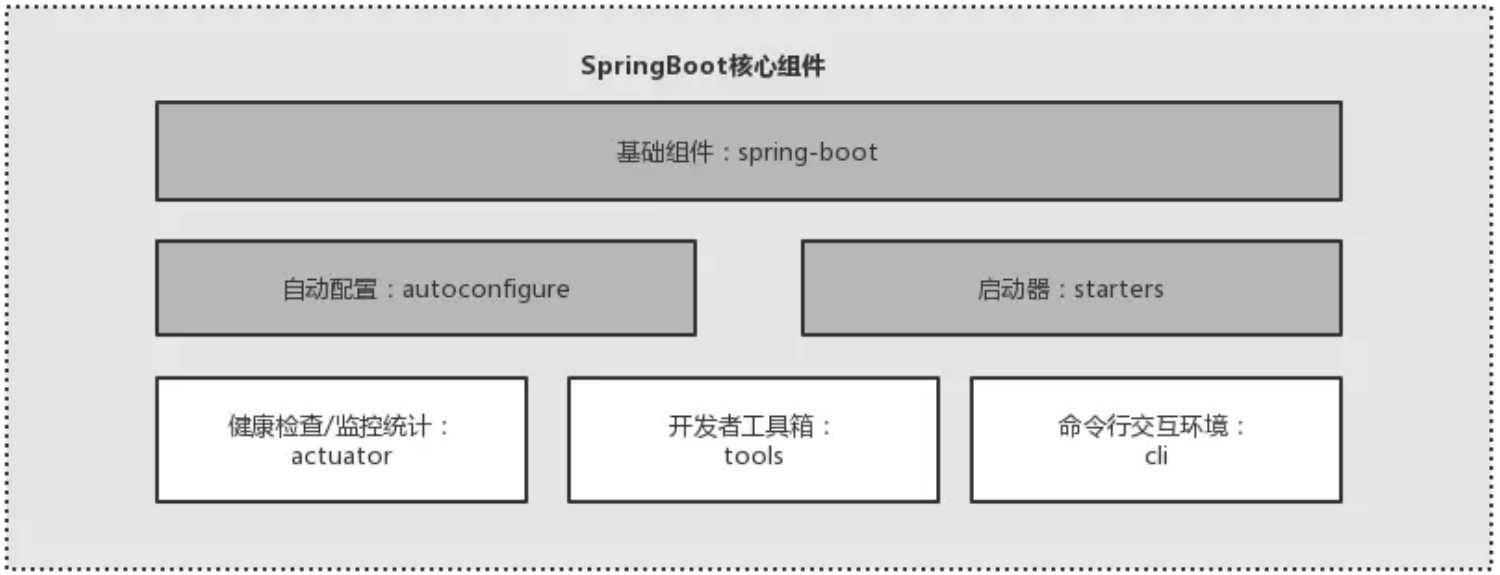
- Spring Boot 其設計目的是用來簡化新 Spring 應用的初始搭建以及開發過程。上手 Spring 開發更快、更廣泛。
- Spring Boot 的核心思想就是約定大於配置 (convention over configuration)，多數 Spring Boot 應用只需要很少的 Spring 配置，可以使用預設方式實現快速開發。
- 使用注解，使編碼變得簡單。
- 使用自動配置，使配置變得簡單，快速構建專案，快速集成新技術的能力。
- 使部署變得簡單： 內嵌 Tomcat Jetty Web 容器。
- 採用 Spring Boot 可以 大大的簡化開發模式，所有你想集成的常用框架，它都有對應的組件支援。
- 自帶應用監控

Spring Boot 的核心 → 約定優於配置

什麼是約定優於配置呢？
約定優於配置（Convention Over Configuration），也稱作按約定程式設計，是一種軟體設計範式，旨在減少軟體發展人員需做決定的數量、獲得簡單的好處，而又不失靈活性。

本質是說，開發人員僅需調整應用中不符約定的部分。例如如果模型中有個名爲Sale的類，那麼數據庫中對應的表就會默認命名爲sales。只有在偏離這一約定時，例如將該表命名爲"products_sold"，才需寫有關這個名字的配置。如果您所用工具的約定與你的期待相符，便可省去配置；反之你可以配置來達到你所期待的方式。

Spring Boot 體系將約定優於配置的思想展現得淋漓盡致，小到設定檔、中介軟體的預設配置，大到內置容器、生態中的各種 Starters 無不遵循此設計規則，Spring Boot 約定優於配置的思想讓 Spring Boot 項目非常容易上手，讓程式設計變得更簡單。



Spring Boot 的核心 → Starter

Spring Boot Starter ，開箱即用的依賴配置

Spring Boot Starters 基於約定優於配置的理念來設計，Spring Boot Starter 中有兩個核心組件：自動配置代碼和提供自動配置模組及其它有用的依賴。也就意味著當我們項目中引入某個 Starter，即擁有了此軟體的預設使用能力，除非我們需要特定的配置，一般情況下我僅需要少量的配置或者不配置即可使用元件對應的功能。

Spring Boot 由眾多 Starter 組成

- Spring-boot-starter ，核心 Starter，包括自動配置支援，日誌和 YAML
- spring-boot-starter-actuator，Actuator 是 SpringBoot 提供的監控功能
- Spring-boot-starter-cache ，用於使用 Spring 框架的緩存支持
- Spring-boot-data-jpa ，用於使用 Hibernate 實現 Spring Data JPA
- Spring-boot-starter-security，對 Spring Security 的支持
- Spring-boot-starter-test ，用於測試 Spring Boot 應用，支援常用測試類庫，包括 JUnit、Hamcrest 和 Mockito
- Spring-boot-starter-web ，用於使用 Spring MVC 構建 Web 應用，包括 RESTful。Tomcat 是默認的內嵌容器

建置第一個 Spring boot 專案

1. 創建 Maven 工程，構建專案結構

2. 配置 pom.xml，引用各種 starter 啟動器簡化配置

3. 配置 YML 設定檔運行參數

4. 運行與測試

5. 打包與獨立運行 (Linux 環境指令如下)

   mvn clean package

   nohup java –jar spring-boot-restful.jar > /dev/null 2>&1 &

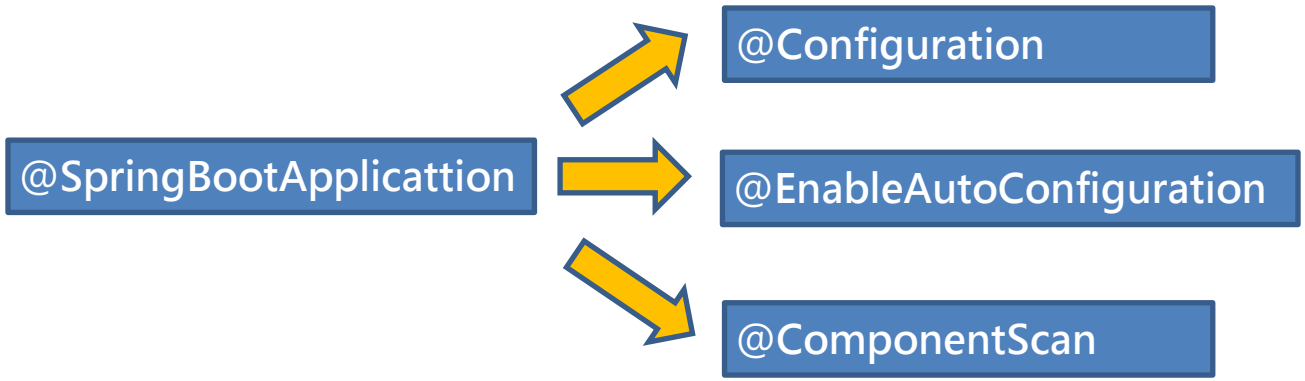
Spring Boot 運作流程

- Spring Boot 啟動核心 annotation @SpringBootApplication

@SpringBootApplication annotation 是 Spring Boot 的核心註解，用註解標注入口類別是應用的啟動類別，透過在啟動類別中的 main 方法中通過 SpringApplication.run(App.class , args) 來啟動 Spring Boot 應用程式。

- SpringBootApplication 註解主要組合了以下註解。

1. @Configuration ，這是 Spring Boot 專案的配置註解。
2. @EnableConfiguration ，啟動自動配置，該註解會讓 Spring Boot 根據當前專案所依賴的 jar 包自動配置專案相關配置
3. @ComponentScan ，掃描配置，Spring Boot 會掃描SpringBootApplication 的同級目錄以及它的子目錄， 所以建議將@SpringBootApplication 修飾的入口類放置在主目錄下，這樣做可以保證 Spring Boot 掃描到專案所有的程式。



Spring Boot 運作流程

- context-path：Context path of the application.
- session.timeout：Session timeout. If a duration suffix is not specified, seconds will be used
- accesslog：tomcat server access log

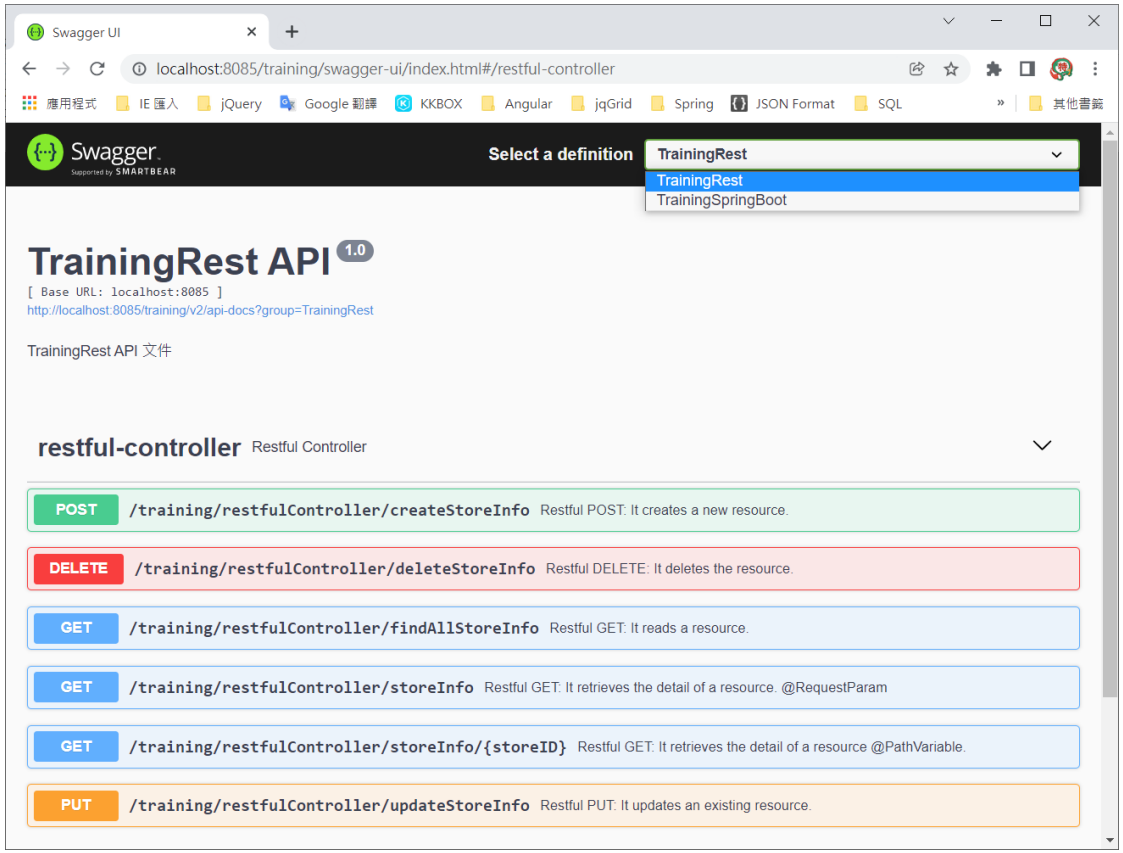
application.yml

```yaml
server:
  port: 8085
  servlet:
    context-path: /training
    session:
      timeout: 1m
  tomcat:
    threads:
      max: 200
    uri-encoding: UTF-8
    basedir: /data/logs/training/access_log
    accesslog:
      enabled: true
      pattern: '{"Local IP":"%A","Date time":"%{yyyy-MM-dd HH:mm:ss.SSS}t","Request method":"%m","URL"
      suffix: .log

```

Spring boot Swagger

- 建立視覺化可操作 API Swagger 頁面、api-docs文件自動建立
- 設置各不同 Package Controller 於 Swagger 上可以下拉選單區分功能
- Swagger URL

http://localhost:8085/training/swagger-ui/index.html



Spring boot Bean Lifecycle

- Spring boot Bean 生命週期 (從創建到銷毀)

1. Bean Constructor
   - BeanPostProcessor postProcessBeforeInitialization
2. Bean @PostConstruct
3. InitializingBean afterPropertiesSet method
4. @Bean initMethod
   - BeanPostProcessor postProcessAfterInitialization
5. Bean @PreDestroy
6. DisposableBean destroy method
7. @Bean destroyMethod

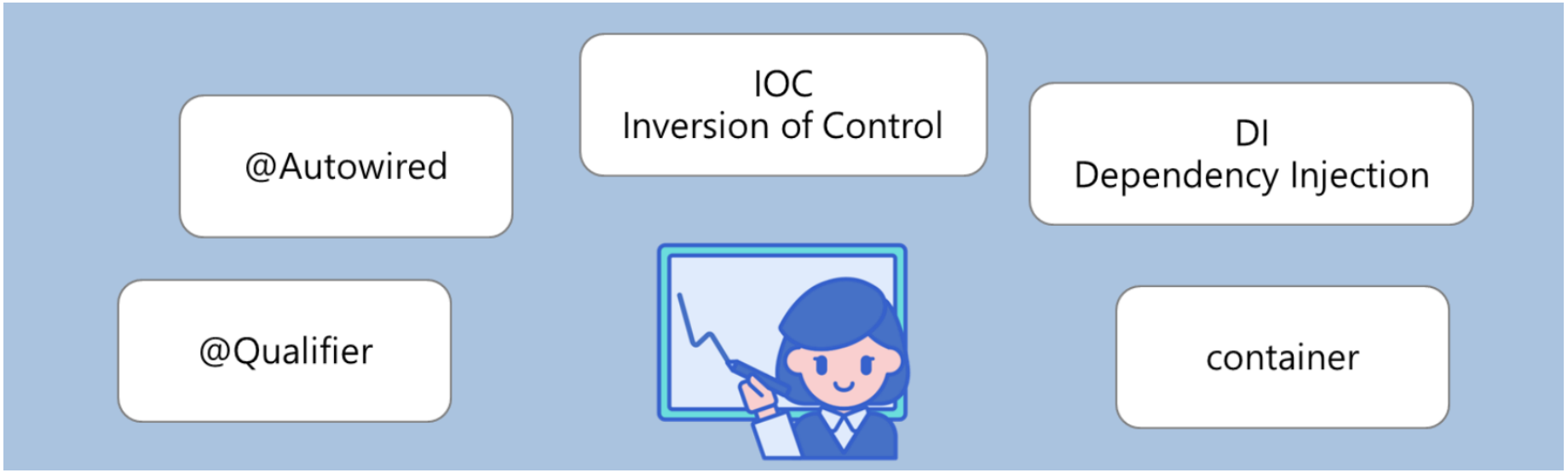
- Spring boot Bean Scop (Bean存取範圍)

1. singleton：每次建立都是相同的單一物件實例
2. prototype：每次建立都是不同的新的物件實例
3. request：無法跨 Http Request 之間存取
4. session：可跨 Http Request之間存取
5. application：全域的 ServletContext 不受到 Session timeout 影響

Spring boot IOC (Inversion of Control)

- 控制反轉（inversion of control，IOC）是一種程式設計的概念，而依賴注入 DI (Dependency Injection) 則是實現這種概念的方式。

  實務應用上常見的使用方式是，在Service Layer 也就是商業邏輯元件設計出共同要操作使用的 Interface 並且設計抽象方法再撰寫其 implements 各實作類別，並在實作類別上標注**@Service** 交由 Spring container 建立實體後，在經由 **@Autowired** 的方式注入不同的實作（而非使用 new 物件的方式）且型別宣告為介面統一操作，再透過 **@Qualifier** 的方式明確指定實作類別名稱，以達成依賴關係轉移的動態結果!



Spring boot IOC (Inversion of Control)

- 以下三種為 Spring boot 常見的不同角色服務元件標注

@Service

專案內部所自行選寫的 Business logic Layer 商業邏輯層

@Component

AOP Aspect 元件、呼叫外部第三方系統 API 元件

@Repository

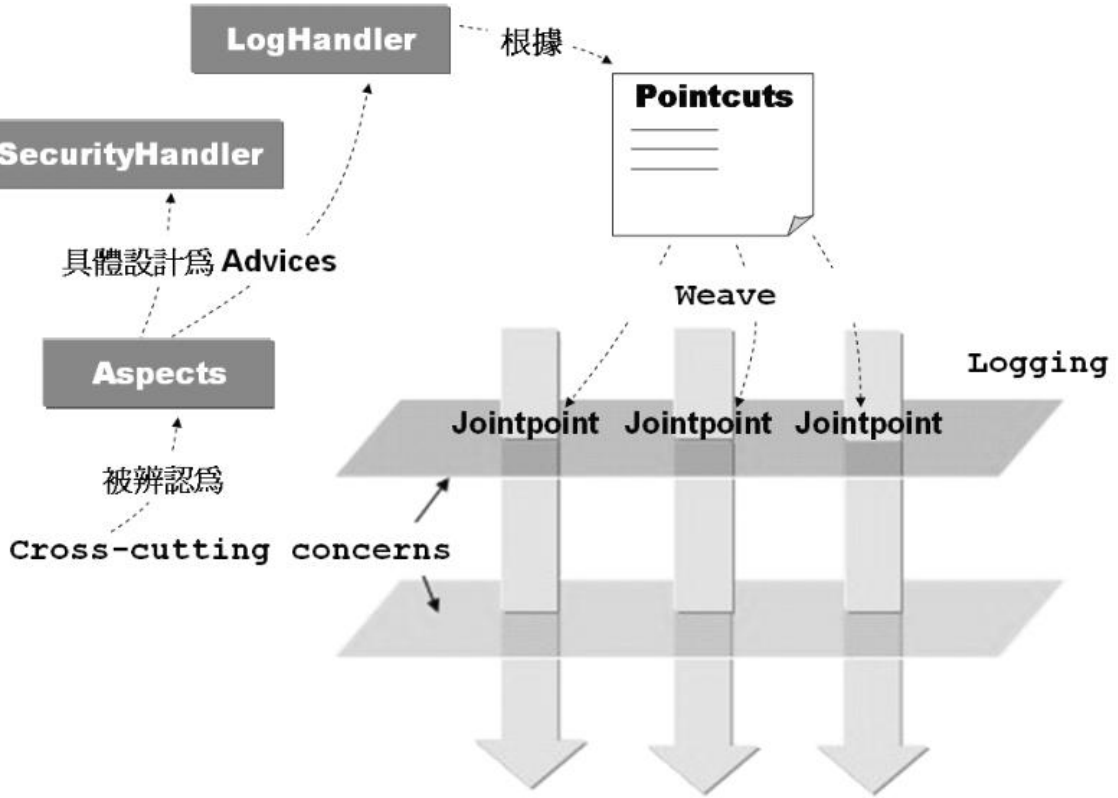
與資料庫交易 Persistence Layer 持久層 DAO (data access object) 元件

Spring boot AOP (Aspect Oriented Programming)

- AOP (Aspect Oriented Programming) 面向切面編程

將多個業務流程元件中共同要做的事 (EX：進入方法前後紀錄時間 Log 以計算方法執行時所需要的時間)，若這些程式散落各處且直接寫在各服務元件之中，很顯然並不是一個好的做法，將會造成重複的程式碼產生和若是想要移除或加入就必須要修改多隻程式。

我們將這些共用元件收集起來並設計撰寫為**@Aspect** (切面) 再透過**@Pointcut** (切入點) 來決定哪些要被加入切面執行的方法 Joinpoint (加入點)，達到被切面執行的目標程式可以抽離獨立出來這些切面實作，並且可以動態的決定是否要執行。



Spring boot AOP (Aspect Oriented Programming)

- Spring boot 提供了5種 Advice

@Before Advice

方法執行前，可修改方法的輸入參數

@After Advice

方法執行後

@AfterReturning Advice

方法執行後，可修改方法的回傳值

@Around Advice

方法執行前/後，可修改方法的回傳值，可決定目標的方法是否執行

@AfterThrowing Advice

方法發生例外錯誤時

Spring boot AOP (Aspect Oriented Programming)

- @RestControllerAdvice 統一處理應用程式中全部 Controller Request、Response、與所拋出的 Exception例外錯誤

- Extends **RequestBodyAdviceAdapter**
  overrides **afterBodyRead** method

- Implements **ResponseBodyAdvice**<Object>
  overrides **beforeBodyWrite** method

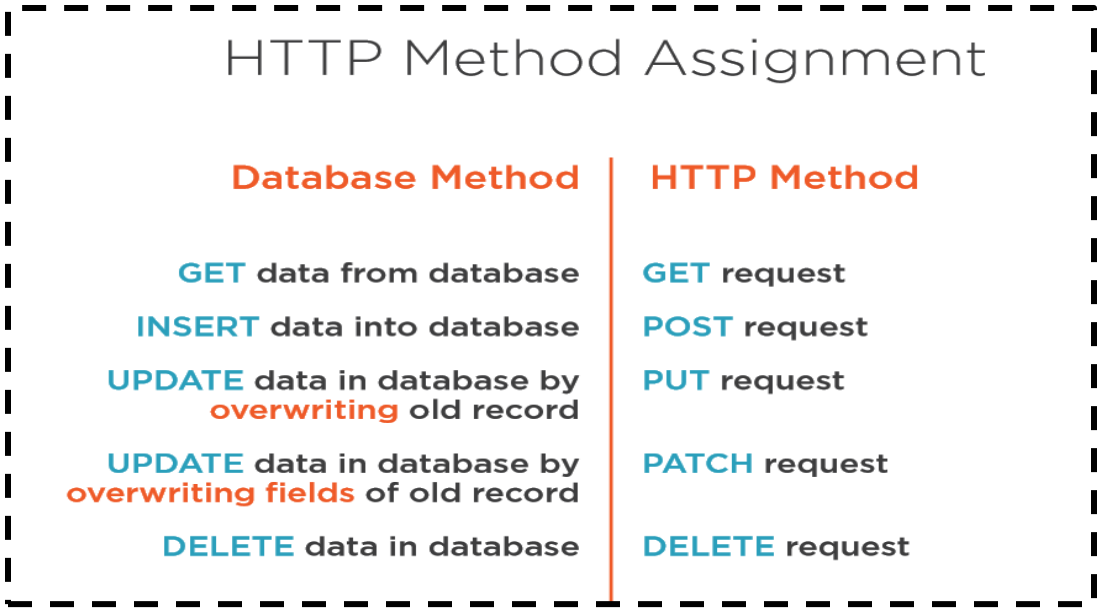
- @ExceptionHandler ({BusinessException.class})

Spring boot RestController

- REST (Representational State Transfer) 表現層狀態轉移
  Stateless (無狀態)： **Client 端自行保存狀態**，在請求 Server 的時候，一併附上給 Server 端，**Server 端無保存** Client 端的狀態資訊。

- REST uses various representations to represent a resource like text, JSON, XML. **JSON** (JavaScript Object Notation) is the most popular one

- Use HTTP methods explicitly (i.e. **POST, GET, PUT, PATCH, DELETE**)

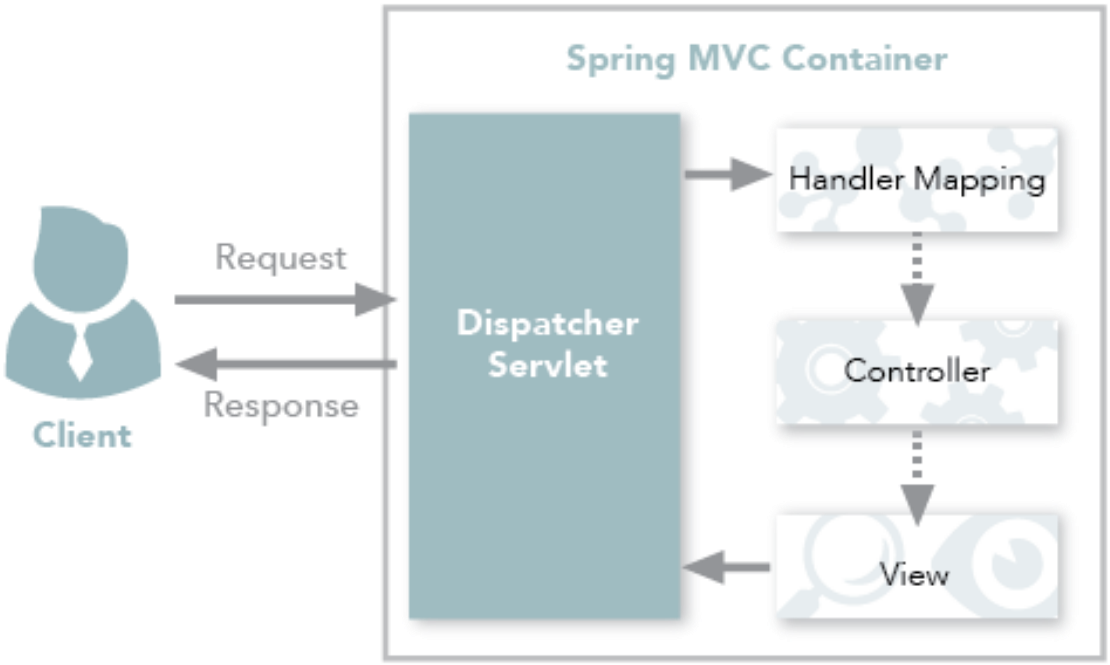


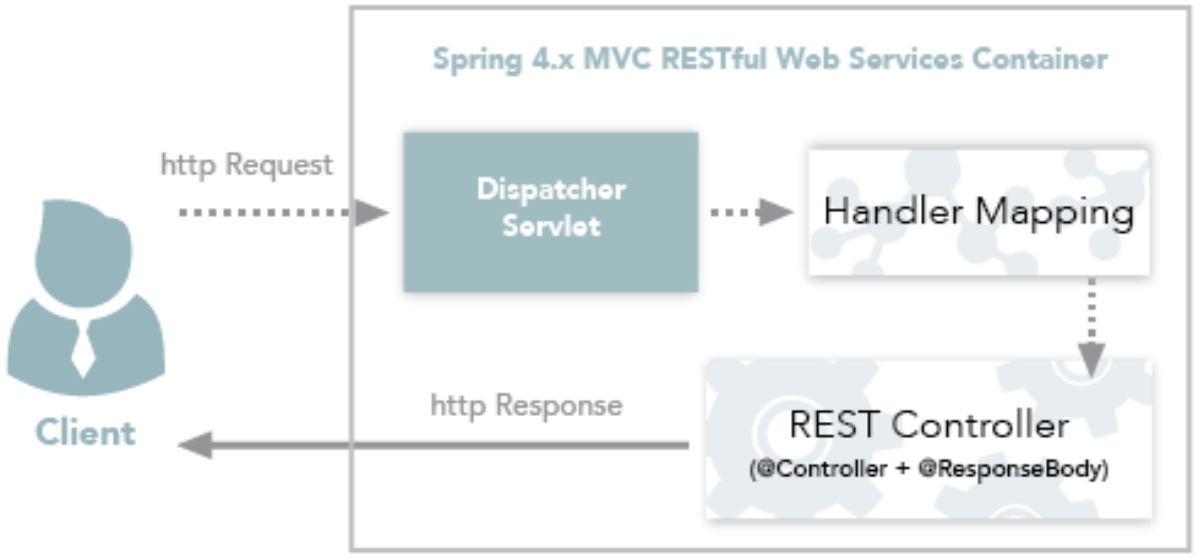
Spring boot RestController

Status Codes

HTTP status codes returned in the response header:

- **200 OK** The resource was read, updated, or deleted.
- **201 Created** The resource was created.
- **400 Bad Request** The data sent in the request was bad.
- **403 Not Authorized** The Principal named in the request was not authorized to perform this action.
- **404 Not Found** The resource does not exist.
- **409 Conflict** A duplicate resource could not be created.
- **500 Internal Server Error** A service error occurred.


Spring MVC Container
Handler Mapping
Request
Dispatcher Servlet
Controller
Response
Client
View


Spring 4.x MVC RESTful Web Services Container
http Request
Dispatcher Servlet
Handler Mapping
Client
http Response
REST Controller
(@Controller + @ResponseBody)

Spring boot RestController

- Microservices 微服務

✓ 專注於單一責任與功能的小型功能區塊 (Small Building Blocks) 為基礎

✓ 模組化的方式組合出複雜的大型應用程式，各功能區塊使用與語言無關 (Language-Independent/Language agnostic) 的API集相互通訊

✓ 微服務是一種以業務功能為主的服務設計概念，每一個服務都具有自主運行的業務功能，對外開放不受語言限制的 API (最常用的是 HTTP)

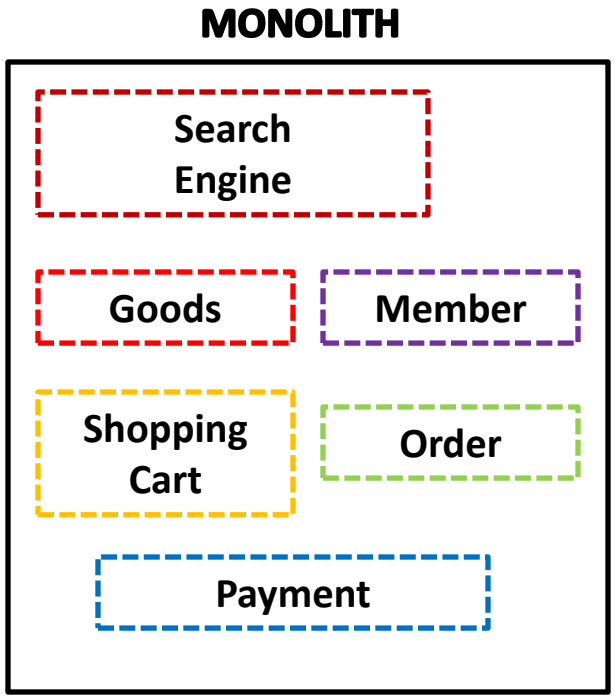
✓ 拆分每個微服務就可單獨部署及發佈上線，不受其它別的微服務所影響

✓ 可依照每個微服務所需的系統負載請求量的不同，給予相對的主機硬體支援

Spring boot RestController

- 單體式應用程式架構

單體式應用表示一個應用程式內包含了所有需要的業務功能，每一個業務功能是不可分割的。若新增或異動其中一個功能就需要整個應用程式下線打包再重新部署上線 (牽一髮而動全身)，往往應用程式中最吃耗費資源、需要運算資源的僅有某個業務部份 (例如跑分析報表或是數學演算法分析) ，但因為單體式應用無法分割該部份，因此無形中會有大量的資源浪費的現象，大機器且昂貴不易擴充。

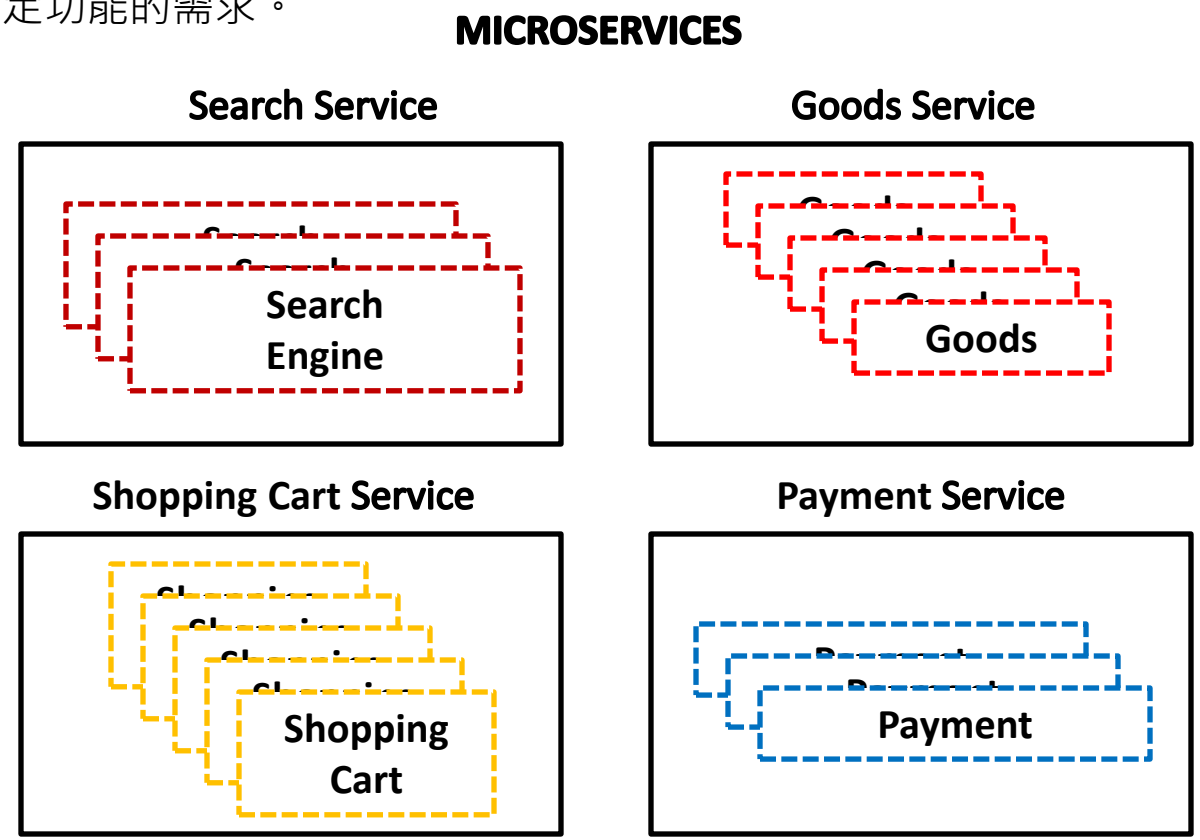


32 core
64GB memory
512GB ROM

Spring boot RestController

- Microservices 微服務架構

應用程式會**拆分**並且建立為**獨立元件**，並會以服務的形式執行個別應用程式程序 (業務功能或流程設計先行分割)。這些服務使用輕量型 API (各個業務功能都獨立實作成一個能自主執行的個體服務)，透過定義良好的界面進行通訊。服務係針對商業功能所建立，且每項服務皆可執行單一功能。因為每項服務皆獨立運作，因此可以**個別更新**、**部署**和**擴展**，以滿足應用程式特定功能的需求。



8 core
16GB memory
128GB ROM

Spring boot RestController

- Spring boot RestController 實作

@RestController
標注在 Controller 類別上，等同於 @Controller + @ResponseBody

@RequestMapping
標注在 Controller 類別上，決定在此 Controller 內所有 API 共同的 prefixes URL

@ApiOperation
標注在 Controller method 方法上，用以描述此 restful api 功能說明

@GetMapping
標注在 Controller method 方法上，對應 HTTP GET 請求用於資料查詢

@PostMapping
標注在 Controller method 方法上，對應 HTTP POST 請求用於資料新增

Spring boot RestController

@PutMapping

標注在 Controller method 方法上，對應 HTTP PUT 請求用於資料更新

@PatchMapping

標注在 Controller method 方法上，對應 HTTP PATCH 請求用於欄位資料部份更新

@DeleteMapping

標注在 Controller method 方法上，對應 HTTP DELETE 請求用於資料刪除

@PathVariable

標注在 Controller method 方法參數上，可將URL網址列值動態帶入方法參數裡

@RequestParam

標注在 Controller method 方法參數上，取得 HTTP 請求 Parameter 帶入方法參數

Spring boot restful Request、Response

@RequestBody

標注在 Controller method 方法參數上，支援前端參數值傳入 JSON 格式
Spring boot 自動轉換參數為 Java POJO 物件

@Validated

標注在 Controller method 方法參數上，用以驗証所傳入的物件參數欄位驗証
經常與 javax.validation.constraints.* 內的 annotation 類別使用搭配驗証功能

ResponseEntity<T>

Spring boot 自動會將所要回傳的 POJO 物件轉為 JSON格式
因應不同情境回傳相對應的 HTTP status 代碼

Spring Data JPA

- JPA是 Java Persistence API的簡寫，是官方提出的一種ORM規範！
- ORM (Object Relational Mapping) Java Model 物件資料模型與資料庫之間的對應關係!
- Spring Data JPA是Spring基於Hibernate開發的一個JPA框架。可以極大的簡化JPA的寫法，可以在幾乎不用寫具體代碼的情況下，實現對資料的訪問和操作。除了「CRUD」外，還包括如分頁、排序等一些常用的功能。
- JPA規範，都在包路徑：javax.persistence.* 下，像一些常用的如：@Entity、@Id及@Table都在此路徑下。

Package	Description
javax.persistence	Core API
javax.persistence.criteria	Criteria API
javax.persistence.metamodel	Metamodel API
javax.persistence.spi	SPI for JPA providers

Spring Data JPA

- spring. datasource：為資料庫連線資訊 spring boot 內建設置
- spring.h2：表示使用內建h2資料庫(無須安裝資料庫)
  mem 表示使用記憶體儲存資料庫
- spring.datasource.hikari.maximum-pool-size ：設置資料庫連線池數量
- spring.jpa.hibernate.ddl-auto：資料庫自動化建立工程，none (不動作)、
  create (建立schema 並且清空資料)、 update (更新schema 已存在的不更新)

application.yml

```
27
28spring:
29  profiles:
30    active: local
31  h2:
32    console:
33      enabled: true
34      path: /h2
35  datasource:
36    url: jdbc:h2:mem:test_db
37    username: root
38    password: root
39    driverClassName: org.h2.Driver
40    hikari:
41      maximum-pool-size: 2
42      connection-timeout: 30000
43  jpa:
44    show-sql: true
45    hibernate:
46      # 不透過@Entity自動化工程建立資料表,經由schema.sql建立資料表與data.sql新增資料
47      ddl-auto: none
48    properties:
49      hibernate:
50        dialect: org.hibernate.dialect.H2Dialect
51        format_sql: true
52        show-sql: true
53
```

Spring Data JPA

- 使用spring boot 內建 spring.Datasource YML 設置，無須另外撰寫 DB @Configuration、@EnableTransactionManagement、@EnableJpaRepositories... 等繁鎖設置

- Spring data jpa @Entity 會自動對應資料庫中的資料表、資料欄位

```
19 @Entity
20 @Table(name = "STORE_INFORMATION")
21 public class StoreInfo {
22
23     @Id
24     @Column(name = "STORE_ID")
25     private long storeID;
26
27     @Column(name = "STORE_NAME")
28     private String storeName;
29
30     @Column(name = "SALES")
31     private long sales;
32
33     @Column(name = "STORE_DATE")
34     private LocalDateTime storeDate;
35
36     @Column(name = "GEOGRAPHY_ID")
37     private Long geographyID;
38
39     @Column(name = "REGION_NAME")
40     private String regionName;
41
42 }
```

Spring Data JPA

- 搭配 @RestController @GetMapping 接收查詢請求

```
14 @RestController
15 @RequestMapping("/restfulJpaController")
16 public class RestfulJpaController {
17
18     @Autowired
19     private StoreInfoJpaService storeInfoJpaService;
20
21     @ApiOperation(value = "Test Hikari Connection Pool")
22     @GetMapping(value = "/testHikariConnectionPool")
23     public ResponseEntity<List<StoreInfo>> testHikariConnectionPool(@RequestParam("userID") String userID) {
24         List<StoreInfo> storeInfos = storeInfoJpaService.findAllStoreInfo(userID);
25
26         return ResponseEntity.ok(storeInfos);
27     }
28
29 }
```

Spring Data JPA

- 透過Spring boot IOC @Autowired 的方式 Dependency Injection 注入 @Repository DAO 物件實體

- 透過繼承 JpaRepository<T, ID> 就可操作相關 API 操作資料庫相當方便!

```
11 @Service
12 public class StoreInfoJpaService {
13
14     private static Logger logger = LoggerFactory.getLogger(StoreInfoJpaService.class);
15
16     @Autowired
17     private StoreInfoDao storeInfoDao;
18
19     public List<StoreInfo> findAllStoreInfo(String userID){
20         List<StoreInfo> storeInfos = storeInfoDao.findAll();
21         storeInfos.forEach(s -> {
22             try { Thread.sleep(2000); } catch (InterruptedException e) { }
23             logger.info("userID("+ userID +"):" + s.toString());
24         });
25
26         return storeInfos;
27     }
28 }
```

```
 7 @Repository
 8 public interface StoreInfoDao extends JpaRepository<StoreInfo, Long>{
 9
10
11 }
```

Spring Data JPA DB multiple datasource

- 在專案實務上常有需要連線多個數據資料庫來源！於Spring boot上就必須另外設置YML及撰寫設置 Configuration java

- 設置多個資料庫數據源 (Oracle、MySQL) 連線資訊、連線驅動 Driver、連線池

PS：37行黃燈泡表示非 Spring boot yml 內建設置

```
36
37 springboot:
38   datasource:
39     oracle:
40       jdbc-url: jdbc:oracle:thin:@localhost:1521:xe
41       username: LOCAL
42       password: root
43       driverClassName: oracle.jdbc.driver.OracleDriver
44       maximumPoolSize: 2
45       connectionTimeout: 30000
46     mysql:
47       jdbc-url: jdbc:mysql://localhost:3306/local_db
48       username: root
49       password: root
50       driverClassName: com.mysql.cj.jdbc.Driver
51       maximumPoolSize: 2
52       connectionTimeout: 30000
53
```

Spring Data JPA DB multiple datasource

- 設置 @Configuration class 建立 DataSource @Bean ，並依據 YML 設定檔資料設置 @ConfigurationProperties 前綴 prefix；@Primary 為設定主要使用的資料庫

```
10 @Configuration
11 public class DB_DataSourceConfig {
12
13    @Primary
14    @Bean(name = "oracle")
15    @ConfigurationProperties(prefix = "springboot.datasource.oracle")
16    public DataSource oracleDataSource() {
17       return DataSourceBuilder.create().build();
18    }
19
20    @Bean(name = "mysql")
21    @ConfigurationProperties(prefix = "springboot.datasource.mysql")
22    public DataSource postgreDataSourceBackend() {
23       return DataSourceBuilder.create().build();
24    }
25
26 }
```

Spring Data JPA DB multiple datasource

- @EnableTransactionManagement 資料庫交易管理
- @EnableJpaRepositories
  1. EntityManagerFactory 實體管理工廠，可用來自行控制 Transaction 交易管理，但須建立一個獨立的 EntityManager
  2. TransactionManager 可在 Service method上標注使用 @Transactional
  3. basePackages 設置 @Repository DAO class 的 Package 路徑
- 透過 @Qualifier 指定要注入的 DataSource Bean 名稱

```
21 @Configuration
22 @EnableTransactionManagement
23 @EnableJpaRepositories(
24     entityManagerFactoryRef = "oracleEntityManagerFactory",
25     transactionManagerRef = "oracleTransactionManager",
26     basePackages = { "com.training.jpa.oracle.dao" }
27 )
28 public class OracleDataSourceConfig {
29
30    @Autowired
31    @Qualifier("oracle")
32    private DataSource dataSource;
33
```

Spring Data JPA DB multiple datasource

- 後續可在DAO 元件中注入 EntityManager 以建立並取得 CriteriaBuilder 進行進行進階動態查詢的 CriteriaQuery
- entityManagerFactory packages 設置 **@Entity** class data model 的路徑

```
34    @Autowired
35    private HibernateProperties hibernateProperties;
36
37    @Autowired
38    private JpaProperties jpaProperties;
39
40    @Primary
41    @Bean(name = "oracleEntityManager")
42    public EntityManager entityManager(EntityManagerFactoryBuilder builder) {
43       return entityManagerFactory(builder).getObject().createEntityManager();
44    }
45
46    @Primary
47    @Bean(name = "oracleEntityManagerFactory")
48    public LocalContainerEntityManagerFactoryBean entityManagerFactory(EntityManagerFactoryBuilder builder) {
49       return builder.dataSource(dataSource)
50                .properties(getVendorProperties())
51                .packages("com.training.jpa.oracle.entity").build();
52    }
53
54    private Map<String, Object> getVendorProperties() {
55       return hibernateProperties.determineHibernateProperties(jpaProperties.getProperties(), new HibernateSettings());
56    }
57
58    @Primary
59    @Bean(name = "oracleTransactionManager")
60    public PlatformTransactionManager transactionManager(EntityManagerFactoryBuilder builder) {
61       return new JpaTransactionManager(entityManagerFactory(builder).getObject());
62    }
```

Spring Data JPA DB multiple datasource

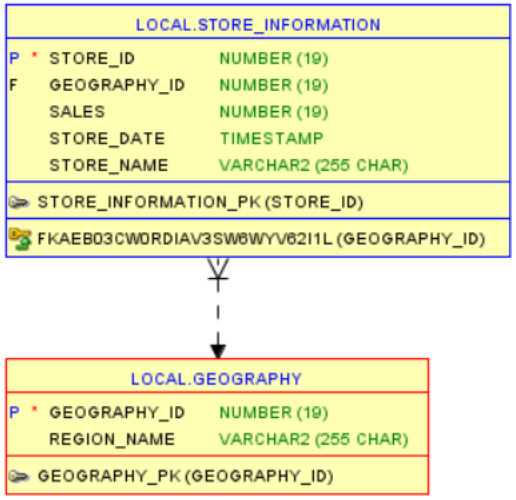
```java
package com.training.jpa.config;

import java.util.Map;

@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(
    entityManagerFactoryRef = "mySqlEntityManagerFactory",
    transactionManagerRef = "mySqlTransactionManager",
    basePackages = { "com.training.jpa.mysql.dao" }
)
public class MySqlDataSourceConfig {

   @Autowired
   @Qualifier("mysql")
   private DataSource dataSource;

   @Autowired
   private HibernateProperties hibernateProperties;

   @Autowired
   private JpaProperties jpaProperties;


   @Bean(name = "mySqlEntityManager")
   public EntityManager entityManager(EntityManagerFactoryBuilder builder) {
      return entityManagerFactory(builder).getObject().createEntityManager();
   }


   @Bean(name = "mySqlEntityManagerFactory")
   public LocalContainerEntityManagerFactoryBean entityManagerFactory(EntityManagerFactoryBuilder builder) {
      return builder.dataSource(dataSource)
              .properties(getVendorProperties())
              .packages("com.training.jpa.mysql.entity").build();
   }

   private Map<String, Object> getVendorProperties() {
      return hibernateProperties.determineHibernateProperties(jpaProperties.getProperties(), new HibernateSettings());
   }


   @Bean(name = "mySqlTransactionManager")
   public PlatformTransactionManager transactionManager(EntityManagerFactoryBuilder builder) {
      return new JpaTransactionManager(entityManagerFactory(builder).getObject());
   }

}
```

Spring Data JPA DB Entity (單一資料表)

- **@Entity**：標注類別為一個資料庫實體
- **@Table** ：設置實際對應的資料表名稱及來源 schema
- **@EmbeddedId**：Composite ID 複合式主鍵，一般主鍵欄位則設為 **@Id**
- **@Column** ：對應實際資料表欄位名稱

```
19 @Entity
20 @Table(name = "STORE_INFORMATION_2", schema="LOCAL")
21 public class StoreInfo2 {
22
23     @EmbeddedId
24     private StoreInfoPrimaryKey storeInfoPrimaryKey;
25
26     @Column(name = "SALES")
27     private long sales;
28
29     @Column(name = "STORE_DATE")
30     private LocalDateTime storeDate;
31
32     @Column(name = "GEOGRAPHY_ID")
33     private Long geographyID;
34
35     @SuperBuilder
36     @NoArgsConstructor
37     @ToString
38     @Data
39     @Embeddable
40     public static class StoreInfoPrimaryKey implements Serializable{
41
42         @Column(name = "STORE_ID")
43         private long storeID;
44
45         @Column(name = "STORE_NAME")
46         private String storeName;
47
48     }
49
50 }
```

Spring Data JPA DB Entity (單向/雙向 一對多)



- **@OneToMany (雙向一對多)**

物件雙方各自都擁有對方的實體參照，雙方皆意識到對方的存在

1. **mappedBy**：設置連結為**雙向關係**所對映的物件欄位名稱
   (除非關係是單向的，否則此參數必須)
2. **Cascade**：子項資料異動方式
   CascadeType.**PERSIST** 在儲存時一併"**儲存**"被參考的物件
   CascadeType.**MERGE** 在合併修改時一併"**合併**"修改被參考的物件
   CascadeType.**REMOVE** 在移除時一併"**移除**"被參考的物件 (刪除父項時連同子項一併刪除)
   CascadeType.**REFRESH** 在更新時一併"**更新**"被參考的物件
   CascadeType.**DETACH** 物件脫離 persistence context 管理
   CascadeType.**ALL** 無論**儲存、合併、更新、移除**，一併對被參考物件作出對應動作
3. **orphanRemoval**：針對未被參照到的子項資料刪除 (搭配 CascadeType.ALL)
4. **fetch**：載入方式 FetchType.**LAZY** 多方預設**延遲**載入查詢

```
34    @JsonIgnore
35    @OneToMany(
36        mappedBy = "geography",
37        cascade = {CascadeType.ALL}, orphanRemoval = true,
38        fetch = FetchType.LAZY
39    )
40    @OrderBy(value = "storeID")
41    private List<StoreInfo> storeInfos;
```

Spring Data JPA DB Entity (單向/雙向 一對多)

- **@OneToMany (單向一對多)**

物件雙方關係僅由單方維持擁有，僅由 Geography 單方控制

所以再儲存多方 StoreInfo 時，JPA INSERT 好資料後還須下 UPDATE 更新外來鍵 GEOGRAPHY_ID 較無效率!

```
60      @OneToMany(
61          cascade = {CascadeType.ALL}, orphanRemoval = true,
62          fetch = FetchType.LAZY
63      )
64      @JoinColumn(name="GEOGRAPHY_ID")
65      private List<StoreInfo> storeInfos;
```

```
Hibernate:
    insert
    into
        local.store_information
        (geography_id, sales, store_date, store_name, store_id)
    values
        (?, ?, ?, ?, ?)
Hibernate:
    update
        local.store_information
    set
        geography_id=?
    where
        store_id=?
```

Spring Data JPA DB Entity (單向/雙向 一對多)

- @ManyToOne (多對一)

1. fetch：載入方式 FetchType.**EAGER** 一方預設**即時**載入查詢

- @JoinColumn

1. name：設置外來鍵欄位名稱
2. insertable = false, updatable = false：

當使用JPA配置實體時，如果有兩個欄位（一般**屬性**欄位、多對一**物件**欄位）對應到的是資料表的相同欄位就必須設此參數，因為資料是透過一般屬性欄位給值新增的，就不必透過麻煩的要塞入完整的**物件**欄位，因此建議此做法較為便利!

```
52
53      @Column(name = "GEOGRAPHY_ID")
54      private Long geographyID;
55
56      @ManyToOne(fetch = FetchType.EAGER)
57      @JoinColumn(name = "GEOGRAPHY_ID", insertable = false, updatable = false)
58      private Geography geography;
59
```

Spring Data JPA DB Entity (一對一)

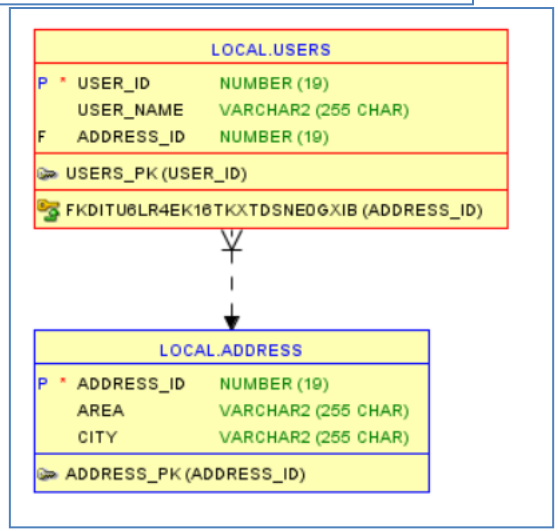
- @OneToOne (一對一)

透過 @JoinColumn 設置資料表外來鍵欄位名稱

透過 mappyBy 設置關聯所對應的一對一物件欄位資料

```java
@Entity
@Table(name = "USERS", schema="LOCAL")
public class User {

    private Long id;

    private String userName;

    // The address_id column in users is the foreign key to address.
    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "ADDRESS_ID", referencedColumnName = "ADDRESS_ID")
    private Address address;

}
```

```java
@Entity
@Table(name = "ADDRESS", schema="LOCAL")
public class Address {

    private Long id;

    private String city;

    private String area;

    @JsonIgnore
    @OneToOne(mappedBy = "address")
    private User user;

}
```



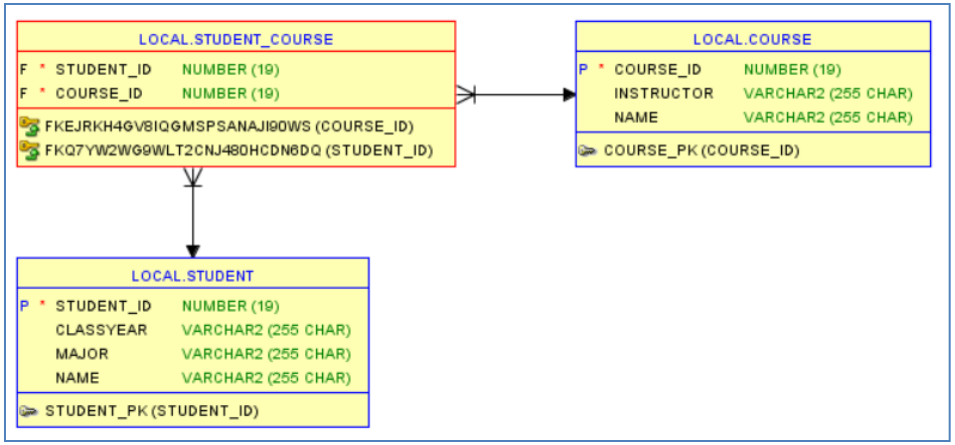
Spring Data JPA DB Entity (多對多)

- @ManyToMany (多對多)

透過 @JoinTable 設置多對多關係之間的中間對應資料表名稱，設置 joinColumns、inverseJoinColumns 兩個外來鍵欄位名稱

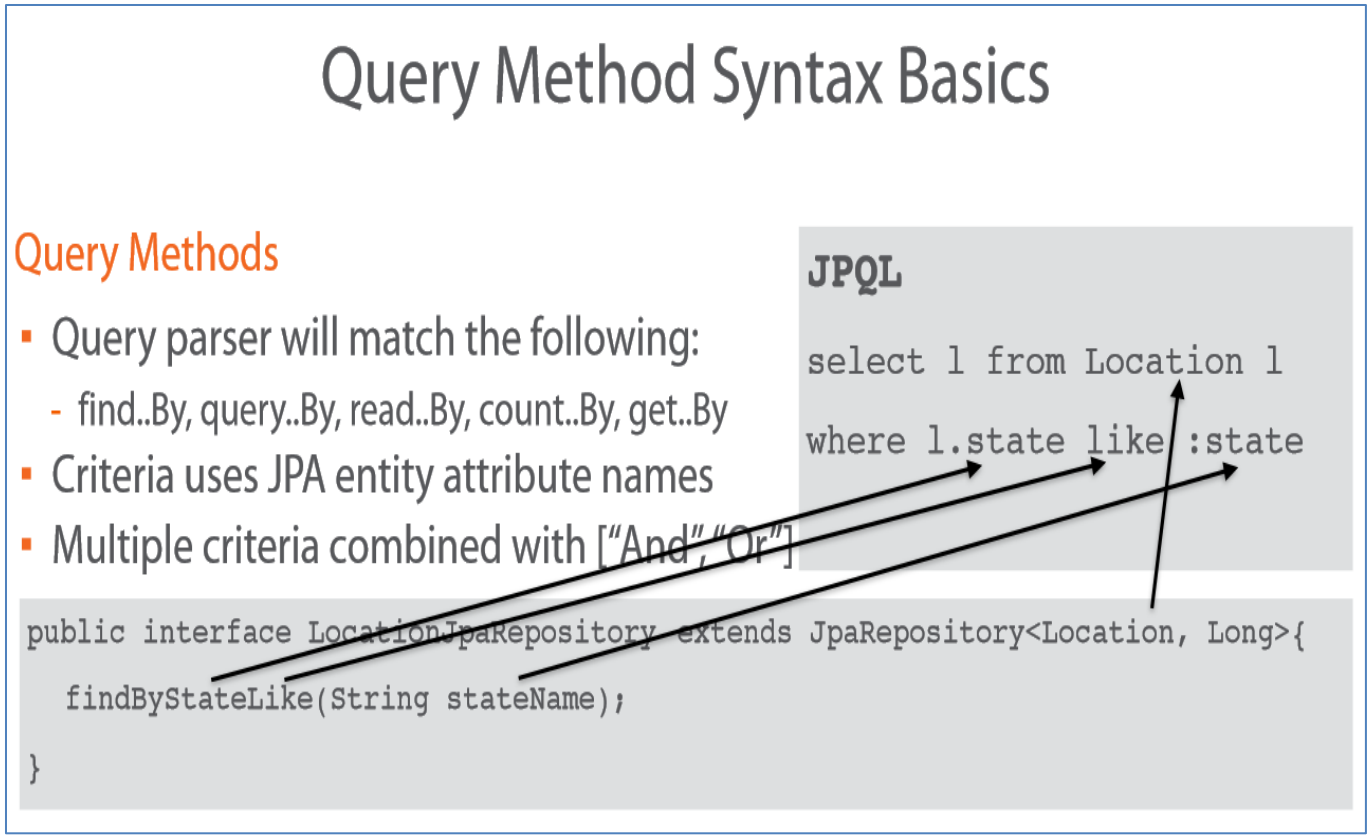
```java
@Entity
@Table(name = "STUDENT")
public class Student {

    private Long studentID;

    private String name;

    private String major;

    private String classYear;

    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable(
        name = "STUDENT_COURSE",
        joinColumns = @JoinColumn(name = "STUDENT_ID"),
        inverseJoinColumns = @JoinColumn(name = "COURSE_ID")
    )
    private List<Course> courses;

}
```

```java
@Entity
@Table(name = "COURSE")
public class Course {

    private Long courseID;

    private String name;

    private String instructor;

    @JsonIgnore
    @ManyToMany(mappedBy = "courses")
    private List<Student> students;

}
```



Spring Data JPA Query methods

- Spring Data JPA 支援 DAO interface 元件在繼承JpaRepository<T, ID>介面後，可撰寫籍由**語法關鍵字**及**實體屬性欄位**所組合的查詢**抽象方法**，背後將會自動轉化為對應SQL語法進行查詢動作



Spring Data JPA Query methods

- 針對關鍵字語法查詢所回傳相對應的回傳資料型別

Query Method Return Types

```
public interface LocationJpaRepository extends JpaRepository<Location, Long>{
  Location findFirstByState(String stateName);
  List<Location> findByStateLike(String stateName);
  Long countByStateLike(String stateName);
}
```

Keyword: And and Or

Uses	*Combines multiple criteria query filters together using a conditional And or Or*
Keyword Example	*findByStateAndCountry("CA", "USA");* *findByStateOrState("CA", "AZ");*
JPQL Example	*... where a.state = ?1 and a.country = ?2* *... where a.state = ?1 or a.state = ?2*

Keyword: Equals, Is and Not

Uses	*The default '=' when comparing the criteria with the filter value. Use Not when wanting to compare not equals*
Keyword Example	*findByState("CA");* *findByStateIs("CA");* *findByStateEquals("CA");* *findByStateNot("CA");*
JPQL Example	*... where a.state = ?1* *... where a.state = ?1* *... where a.state = ?1* *... where a.state <> ?1*

Keyword: Like and NotLike

Uses	*Useful when trying to match, or not match, a portion of the criteria filter value*
Keyword Example	*findByStateLike("Cali%");* *findByStateNotLike("Al%");*
JPQL Example	*... where a.state like ?1* *... where a.state not like ?1*

Keyword: StartingWith, EndingWith and Containing

Uses	*Similar to the "Like" keyword except the % is automatically added to the filter value*
Keyword Example	*findByStateStartingWith("Al"); //Al%* *findByStateEndingWith("ia"); //%ia* *findByStateContaining("in"); //%in%*
JPQL Example	*... where a.state like ?1* *... where a.state like ?1* *... where a.state like ?1*

Keyword: LessThan(Equal) and GreaterThan(Equal)

Uses	When you need to perform a <, <=, >, or >= comparison with number data types
Keyword Example	findByPriceLessThan(20); findByPriceLessThanEqual(20); findByPriceGreaterThan(20); findByPriceGreaterThanEqual(20);
JPQL Example	... where a.price < ?1 ... where a.price <= ?1 ... where a.price > ?1 ... where a.price >= ?1

```
findByPriceGreaterThanAndPriceLessThan(10, 20);
```

Keyword: Before, After and Between

Uses	*When you need to perform a less than, greater than or range comparison with date/time data types*
Keyword Example	*findByFoundedDate**Before**(dateObj);* *findByFoundedDate**After**(dateObj);* *findByFoundedDate**Between**(startDate, endDate);*
JPQL Example	*... where a.foundedDate < ?1* *... where a.foundedDate > ?1* *... where a.foundedDate **between** ?1 **and** ?2*

Keyword: True and False

Uses	***Useful when comparing boolean values with true or false.***
Keyword Example	***findByActive**True*(); ***findByActive**False*();
JPQL Example	***… where a.active = true*** ***… where a.active = false***

Keyword: IsNull, IsNotNull and NotNull

Uses	***Used to check whether a criteria value is null or not null***
Keyword Example	***findByStateIsNull();*** ***findByStateIsNotNull();*** ***findByStateNotNull();***
JPQL Example	***… where a.state is null*** ***… where a.state not null*** ***… where a.state not null***

Keyword: In and NotIn

Uses	*When you need to test if a column value is part of a collection or set of values or not*
Keyword Example	*findByStateIn(Collection<String> states);* *findByStateNotIn(Collection<String> states);*
JPQL Example	*… where a.state in ?1* *… where a.state not in ?1*

Keyword: IgnoreCase

Uses	*When you need to perform a case insensitive comparison*
Keyword Example	*findByStateIgnoreCase("ca");* *findByStateStartingWithIgnoreCase("c");*
JPQL Example	*... where UPPER(a.state) = UPPER(?1)* *... where UPPER(a.state) like UPPER(?1%)*

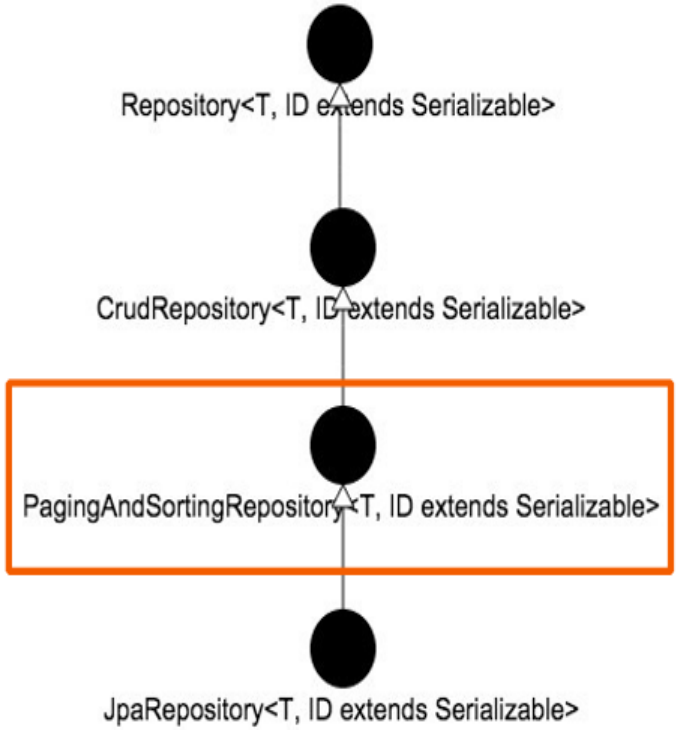
Keyword: OrderBy

Uses	Used to setup an order by clause on your query
Keyword Example	findByStateOrderByCountryAsc(); findByStateOrderByCountryDesc();
JPQL Example	… where a.state order by a.country asc … where a.state order by a.country desc

Keyword: First, Top and Distinct

Uses	***Used to limit the results returned by the query***
Keyword Example	***findFirstByStateLike("Al");*** ***findTop5ByStateLike("A");*** ***findDistinctManufacturerByStateLike("A");***
JPQL Example	***… where a.state like ?1 limit 1*** ***… where a.state like ?1 limit 5*** ***select distinct … where a.state like ?1***

Paging and Sorting

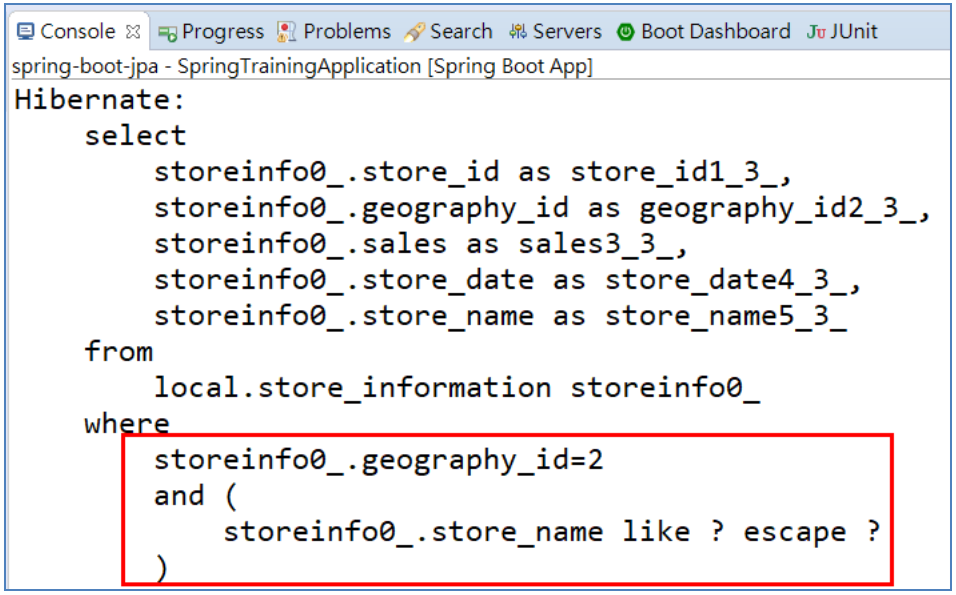



```
Iterable<T> findAll(Sort sort);
Page<T> findAll(Pageable pageable);
```

Spring Data JPA Query methods

- Query methods 以撰寫方法名稱的方式來達成查詢效果固然方便，但很顯然的不能達到動態查詢語法的功能，因此可透過 JpaRepository **findAll** 方法搭配動態條件組合成的 **ExampleMatcher** 物件來達成!

```
172         StoreInfo storeInfo = StoreInfo.builder().storeName(storeInfoVo.getStoreName())
173                     .geographyID(storeInfoVo.getGeographyID()).build();
174         // 使用 Example 實現動態條件查詢
175         ExampleMatcher matcher = ExampleMatcher.matching()
176                 .withMatcher("storeName", match -> match.contains()) // 模糊查詢，即%STORE_NAME%
177                 .withMatcher("geographyID", match -> match.endsWith()); // 精準查詢
178         Example<StoreInfo> example = Example.of(storeInfo, matcher);
179         List<StoreInfo> storeInfos  = storeInfoOracleDao.findAll(example);
```



Spring Data JPA JPQL

- JPQL (Java Persistence Query Language)

  JPQL 代表Java持久化查詢語言。它被用來創建針對實體的查詢存儲在關聯式資料庫中語法類似SQL，它是完全物件導向方式的查詢語句，具備多型、關聯等特性，以物件的思維查詢資料是以持久化物件 (Entity) 而非資料表 (Table)，與以往 HQL (Hibernate Query Language) 觀念相同

- @Query Enhanced JPQL Syntax

```
18      // By default, the query definition uses JPQL
19      @Query("SELECT s FROM StoreInfo s WHERE s.storeID IS NOT NULL")
20      List<StoreInfo> queryAllStoreInfo();
21
22      // Enhanced JPQL Syntax
23      @Query("SELECT s FROM StoreInfo s WHERE s.geographyID = ?1")
24      List<StoreInfo> queryByGeographyID(long geographyID);
```

```
Hibernate:
    select
        storeinfo0_.store_id as store_id1_3_,
        storeinfo0_.geography_id as geography_id2_3_,
        storeinfo0_.sales as sales3_3_,
        storeinfo0_.store_date as store_date4_3_,
        storeinfo0_.store_name as store_name5_3_
    from
        local.store_information storeinfo0_
    where
        storeinfo0_.store_id is not null
```

```
Hibernate:
    select
        storeinfo0_.store_id as store_id1_3_,
        storeinfo0_.geography_id as geography_id2_3_,
        storeinfo0_.sales as sales3_3_,
        storeinfo0_.store_date as store_date4_3_,
        storeinfo0_.store_name as store_name5_3_
    from
        local.store_information storeinfo0_
    where
        storeinfo0_.geography_id=?
```

Spring Data JPA JPQL

- @Query Named Parameters

```
26      // Named Parameters
27      @Query("SELECT s FROM StoreInfo s WHERE s.geographyID = :geographyID")
28      List<StoreInfo> queryByParamGeographyID(@Param("geographyID") long geographyID);
```

```
Hibernate:
    select
        storeinfo0_.store_id as store_id1_3_,
        storeinfo0_.geography_id as geography_id2_3_,
        storeinfo0_.sales as sales3_3_,
        storeinfo0_.store_date as store_date4_3_,
        storeinfo0_.store_name as store_name5_3_
    from
        local.store_information storeinfo0_
    where
        storeinfo0_.geography_id=?
```

- @Query Native Queries (使用原始一般的SQL查詢)

```
30      // Native Queries
31      @Query(value = "SELECT * FROM STORE_INFORMATION S WHERE S.GEOGRAPHY_ID = ?1", nativeQuery = true)
32      List<StoreInfo> queryNativeByGeographyID(long geographyID);
```

Spring Data JPA JPQL

- @NamedQuery (對應Entity實體類別上的標注查詢)

```
25 // JPQL
26 @NamedQuery(
27     name = "StoreInfo.queryNamedQueryByGeographyID",
28     query = "SELECT s FROM StoreInfo s WHERE s.geographyID = ?1"
29 )
30 @Entity
31 @Table(name = "STORE_INFORMATION", schema="LOCAL")
32 public class StoreInfo {
```

```
15 @Repository
16 public interface StoreInfoQueryAnnotationDao extends JpaRepository<StoreInfo, Long>{
17
18     // @NamedQuery JPQL
19     List<StoreInfo> queryNamedQueryByGeographyID(long geographyID);
20
```

Spring Data JPA JPQL

- @NamedNativeQuery (對應Entity實體類別上的標注**原始SQL**查詢)

```
30 // SQL
31 @NamedNativeQuery(
32     name = "StoreInfo.queryNamedNativeQueryByGeographyID",
33     query = "SELECT * FROM STORE_INFORMATION S WHERE S.GEOGRAPHY_ID = ?1",
34     resultClass = StoreInfo.class
35 )
36 @Entity
37 @Table(name = "STORE_INFORMATION", schema="LOCAL")
38 public class StoreInfo {
```

```
15 @Repository
16 public interface StoreInfoQueryAnnotationDao extends JpaRepository<StoreInfo, Long>{
17
18     // @NamedNativeQuery SQL
19     List<StoreInfo> queryNamedNativeQueryByGeographyID(long geographyID);
```

Spring Data JPA JPQL

- Customizing the Result of JPA Queries with Aggregation Functions

1. Customizing the Result **with Class Constructors**

   對於聚合函數查詢後的資料結果，可透過自訂義資料物件建構式的方法自動綁定資料結果

```
40      // Customizing the Result with Class Constructors
41      @Query(
42          "SELECT new com.training.jpa.model.StoreGroupSum ( " +
43          "    storeName, SUM(sales) " +
44          ") " +
45          "FROM StoreInfo " +
46          "GROUP BY storeName " +
47          "ORDER BY SUM(sales) DESC"
48      )
49      List<StoreGroupSum> queryStoreGroupSumSales();
```

```
 7 @AllArgsConstructor
 8 @Data
 9 @ToString
10 public class StoreGroupSum {
11
12     private String storeName;
13
14     private long sumSales;
15 }
16
```

Spring Data JPA JPQL

2. Customizing the Result **with Spring Data Projection** (資料反射欄位對應)
   建立 interface 訂義所回傳的資料物件，並透過 getXXX 方法對應所查詢的欄位別名
   PS：JPQL 必須搭配別名才有作用!

```
52      // Customizing the Result with Spring Data Projection(資料反射欄位對應)
53      // JPQL必須搭配別名才有作用!
54      @Query(
55          "SELECT G.geographyID AS geographyID, G.regionName AS regionName, " +
56          "S.storeID AS storeID, S.storeName AS storeName, S.sales AS sales, S.storeDate AS storeDate " +
57          "FROM Geography G INNER JOIN StoreInfo S " +
58  //      "FROM Geography G LEFT JOIN StoreInfo S " +
59  //      "FROM Geography G RIGHT JOIN StoreInfo S " +
60  //      "FROM Geography G FULL JOIN StoreInfo S " +
61          "ON G.geographyID = S.geographyID"
62      )
63      List<GeographyStore> queryGeographyJoinStore();
64
```

```
5  public interface GeographyStore {
6
7      Long getGeographyID();
8
9      String getRegionName();
10
11     Long getStoreID();
12
13     String getStoreName();
14
15     Long getSales();
16
17     LocalDateTime getStoreDate();
18
19 }
```

Spring Data JPA JPQL

- @Modifying Modifiable Queries
  方法所回傳的整數代表 DML (Data Manipulation Language) 執行所異動的資料筆數

```
66      // Modifiable Queries Insert
67      // 新增只能使用 nativeQuery SQL不支援 JPQL
68      @Transactional
69      @Modifying
70      @Query(
71          value = "INSERT INTO STORE_INFORMATION (STORE_ID, STORE_NAME, SALES, STORE_DATE, GEOGRAPHY_ID) " +
72                  "VALUES (?1, ?2, ?3, ?4, ?5)",
73          nativeQuery = true
74      )
75      int insertStoreInfo(long storeID, String storeName, long sales, LocalDateTime storeDate, long geographyID);
76
77      @Transactional
78      @Modifying
79      @Query(
80          value = "UPDATE StoreInfo SET sales = ?1 WHERE geographyID = ?2"
81      )
82      int updateStoreInfo(long sales, long geographyID);
83
84      @Transactional
85      @Modifying
86      @Query(
87          value = "DELETE FROM StoreInfo WHERE geographyID = ?1"
88      )
89      int deleteStoreInfo(long geographyID);
90
```

Spring Data JPA Criteria Queries 建立

- 對資料庫進行操作時一般是使用 SQL (Structured Query Language)，在使用 Spring JPA 時即使不了解 SQL 語法的使用與撰寫，也可以使用它所提供的 API javax.persistence.criteria 對 SQL 進行封裝，它是從 Java 物件導向的觀點來「動態」組合各種查詢條件，最後由 Spring JPA 自動產生 SQL 語句

- 注入 EntityManager 透過 getCriteriaBuilder 取得 CriteriaBuilder，再進行後續查詢條件的建立及組合，再透過 CriteriaBuilder 的 createQuery 方法建立 CriteriaQuery 以最後將所有的查詢物件及條件進行連結串接

```
27 @Repository
28 public class CriteriaQueryDao {
29
30     @PersistenceContext(name = "oracleEntityManager")
31     private EntityManager entityManager;
32
33     public List<StoreInfo> findStoreByGeographyAndSalesAndStoreDate(StoreInfoVo storeInfoVo) {
34         CriteriaBuilder cb = entityManager.getCriteriaBuilder();
35         CriteriaQuery<StoreInfo> cq = cb.createQuery(StoreInfo.class);
36         Root<StoreInfo> storeInfo = cq.from(StoreInfo.class);
37
```

Spring Data JPA Criteria Queries : AND、OR、OrderBy

- 透過 **CriteriaBuilder** 建立多個查詢條件 **Predicate**，在將全部的查詢條件組合為最終的 **Predicate**，最後由 **CriteriaQuery** 來串連所有的查詢物件

```
33    public List<StoreInfo> findStoreByGeographyAndSalesAndStoreDate(StoreInfoVo storeInfoVo) {
34        CriteriaBuilder cb = entityManager.getCriteriaBuilder();
35        CriteriaQuery<StoreInfo> cq = cb.createQuery(StoreInfo.class);
36        Root<StoreInfo> storeInfo = cq.from(StoreInfo.class);
37
38        // 商店地區編號 AND
39        Predicate geographyID = cb.equal(storeInfo.get("geographyID"), storeInfoVo.getGeographyID());
40        // 商店營業額 AND
41        Predicate sales = cb.greaterThanOrEqualTo(storeInfo.get("sales"), storeInfoVo.getSales());
42        // 商店營業日期 OR
43        Predicate storeDate = cb.greaterThan(storeInfo.get("storeDate"), storeInfoVo.getStoreDate());
44        // 組合查尋條件
45        Predicate restriction = cb.or(cb.and(geographyID, sales), storeDate);
46
47        // 排序 ORDER BY
48        Order order = cb.desc(storeInfo.get("sales"));
49        // 放入全部查詢條件
50        // PS:select(storeInfo)可省略
51        cq.select(storeInfo).where(restriction).orderBy(order);
52
53        // 執行查詢
54        TypedQuery<StoreInfo> query = entityManager.createQuery(cq);
55        /*
56            SELECT * FROM STORE_INFORMATION
57            WHERE GEOGRAPHY_ID = 2
58            AND SALES >= 1500
59            OR STORE_DATE > '2018-04-01 00:00:00'
60            ORDER BY SALES DESC;
61        */
62        return query.getResultList();
63    }
```

Spring Data JPA Criteria Queries : GROUP BY、HAVING

- 操作「群組」、「聚合函數」、「群組條件」等查詢功能

```
65    public List<StoreGroupSum> queryStoreGroupByAndHaving(long sales) {
66        CriteriaBuilder cb = entityManager.getCriteriaBuilder();
67        CriteriaQuery<StoreGroupSum> cq = cb.createQuery(StoreGroupSum.class);
68        Root<StoreInfo> storeInfo = cq.from(StoreInfo.class);
69
70        // 各別商店加總營業額
71        Expression<Long> sumSales = cb.sum(storeInfo.get("sales"));
72
73        // SELECT STORE_NAME, SUM(SALES) SUM_SALES
74        cq.multiselect(storeInfo.get("storeName"), sumSales)
75          // GROUP BY STORE_NAME
76          .groupBy(storeInfo.get("storeName"))
77          // HAVING SUM(SALES) > 1000
78          .having(cb.greaterThan(sumSales, sales))
79          // ORDER BY SUM(SALES) DESC
80          .orderBy(cb.desc(sumSales));
81
82        // 執行查詢
83        TypedQuery<StoreGroupSum> query = entityManager.createQuery(cq);
84
85        return query.getResultList();
86    }
```

Spring Data JPA Criteria Queries : INNER JOIN、OUTER JOIN

- 操作「INNER JOIN」、「LEFT OUTER JOIN」、「RIGHT OUTER JOIN」

```
87
88      public List<GeographyJoinStore> queryGeographyJoinStore(JoinType joinType) {
89          CriteriaBuilder cb = entityManager.getCriteriaBuilder();
90          CriteriaQuery<GeographyJoinStore> cq = cb.createQuery(GeographyJoinStore.class);
91
92          Root<Geography> geography = cq.from(Geography.class);
93          // Spring JPA 不支援 RIGHT JOIN
94          Join<Geography, StoreInfo> join = geography.join("storeInfos", joinType);
95          cq.multiselect(
96              geography.get("geographyID"), geography.get("regionName"),
97              join.get("storeID"), join.get("storeName"),
98              join.get("sales"), join.get("storeDate")
99          );
100
101         // 執行查詢
102         TypedQuery<GeographyJoinStore> query = entityManager.createQuery(cq);
103
104         return query.getResultList();
105     }
106
```

Spring Data JPA Criteria Queries : Sub Query

- 操作子查詢

```
106
107     public List<StoreInfo> queryStoreSubQuery() {
108         CriteriaBuilder cb = entityManager.getCriteriaBuilder();
109         CriteriaQuery<StoreInfo> cq = cb.createQuery(StoreInfo.class);
110
111         // 內查詢
112         Subquery<Long> subquery = cq.subquery(Long.class);
113         Root<StoreInfo> subStoreInfo = subquery.from(StoreInfo.class);
114         subquery.select(cb.max(subStoreInfo.get("sales")));
115
116         // 外查詢
117         Root<StoreInfo> storeInfo = cq.from(StoreInfo.class);
118         cq.where(
119             cb.equal(
120                 storeInfo.get("sales"), subquery.getSelection()
121             )
122         );
123
124         /*
125             -- 外查詢
126             SELECT * FROM STORE_INFORMATION
127             WHERE SALES = (
128                 -- 內查詢
129                 SELECT MAX(SALES) FROM STORE_INFORMATION
130             );
131          */
132         // 執行查詢
133         TypedQuery<StoreInfo> query = entityManager.createQuery(cq);
134
135         return query.getResultList();
136     }
137
```

Spring Data JPA EentityManager CreateQuery

- 透過 EntityManager 可建立 JPQL、SQL NativeQuery

```
140     // JPQL
141     TypedQuery<StoreInfo> query = entityManager.createQuery("SELECT s FROM StoreInfo s WHERE s.storeID IS NOT NULL", StoreInfo.class);
142     query.getResultList().stream().forEach(System.out::println);
143
144     // SQL NativeQuery
145     Query query2  = entityManager.createNativeQuery("SELECT * FROM STORE_INFORMATION S WHERE S.GEOGRAPHY_ID = ?1", StoreInfo.class);
146     query2.setParameter(1, 2);
147     Stream<StoreInfo> storeInfoStream = query2.getResultStream().map(s -> (StoreInfo)s);
148     List<StoreInfo> storeInfos = storeInfoStream.collect(Collectors.toList());
149     storeInfos.stream().forEach(System.out::println);
```

- SQL NativeQuery ResultSetMapping
  自訂義物件欄位對應查詢欄位資料取得查詢結果
  須建立欄位相對應的 **@SqlResultSetMapping** @Entity 實體物件

```
154         StringBuilder sql = new StringBuilder();
155         sql.append("SELECT STORE_NAME, SUM(SALES) SUM_SALES ");
156         sql.append("FROM STORE_INFORMATION ");
157         sql.append("GROUP BY STORE_NAME ");
158         sql.append("HAVING SUM(SALES) > ?1 ");
159         sql.append("ORDER BY SUM(SALES) DESC ");
160         // com.training.jpa.oracle.entity.StoreGroupSumMapping
161         Query query3  = entityManager.createNativeQuery(sql.toString(), "StoreGroupSumMapping");
162         query3.setParameter(1, 1000);
163         Stream<StoreGroupSumMapping> storeGroupSumStream = query3.getResultStream().map(s -> (StoreGroupSumMapping)s);
164         List<StoreGroupSumMapping> storeGroupSums = storeGroupSumStream.collect(Collectors.toList());
```

Spring Data JPA EentityManager CreateQuery

- 設置 @SqlResultSetMapping SQL查詢欄位與 Java 物件欄位名稱對應

```
10 @SqlResultSetMapping(
11     name="StoreGroupSumMapping",
12     entities={
13         @EntityResult(
14             entityClass = com.training.jpa.oracle.entity.StoreGroupSumMapping.class,
15             fields = {
16                 @FieldResult(name="storeName", column="STORE_NAME"),
17                 @FieldResult(name="sumSales",  column="SUM_SALES")
18             }
19         )
20     }
21 )
22 @Data
23 @Entity
24 public class StoreGroupSumMapping {
25
26     @Id
27     private String storeName;
28
29     private long sumSales;
30
31 }
32
```

Spring Data JPA DB DML operations

- Spring Data JPA 使用 JPQL或者透過繼承 JpaRepository<T, ID> API，來進行資料庫 INSERT、 UPDATE、DELETE 等資料 DML 操作

```
 7 @Repository
 8 public interface StoreInfoDao extends JpaRepository<StoreInfo, Long>{
 9
10
11 }
```

- 資料新增可使用 save 方法進行操作，透過主鍵欄位資料來判斷，當不存在則資料新增 (INSERT)、存在則資料更新 (UPDATE)

```
53    @ApiOperation(value = "Restful POST: It creates a new resource.")
54    @PostMapping(value = "/createStoreInfo")
55    public ResponseEntity<StoreInfo> createStoreInfo(@Validated @RequestBody StoreInfoVo storeInfoVo) {
56        /* Input:範例
57            {
58              "storeID": 10,
59              "storeName": "AppleStore",
60              "storeDate": "2021-11-10T05:42:15",
61              "sales": 5500,
62              "geographyVo": {
63                "geographyID": 3,
64                "regionName": "North"
65              }
66            }
67         */
68        StoreInfo storeInfo = storeInfoService.createStoreInfo(storeInfoVo);
69
70        return ResponseEntity.ok(storeInfo);
71    }
```

Spring Data JPA DB DML operations

```
14
15      @Autowired
16      private StoreInfoDao storeInfoDao;
17
18      public StoreInfo createStoreInfo(StoreInfoVo storeInfoVo) {
19          StoreInfo storeInfo = StoreInfo.builder()
20                  .storeID(storeInfoVo.getStoreID())
21                  .storeName(storeInfoVo.getStoreName())
22                  .storeDate(storeInfoVo.getStoreDate())
23                  .sales(storeInfoVo.getSales())
24                  .geographyID(storeInfoVo.getGeographyVo().getGeographyID())
25                  .regionName(storeInfoVo.getGeographyVo().getRegionName())
26                  .build();
27
28          return storeInfoDao.save(storeInfo);
29      }
30
```

```
Hibernate:
    select
        storeinfo0_.store_id as store_id1_0_0_,
        storeinfo0_.geography_id as geograph2_0_0_,
        storeinfo0_.region_name as region_n3_0_0_,
        storeinfo0_.sales as sales4_0_0_,
        storeinfo0_.store_date as store_da5_0_0_,
        storeinfo0_.store_name as store_na6_0_0_
    from
        store_information storeinfo0_
    where
        storeinfo0_.store_id=?
Hibernate:
    insert
    into
        store_information
        (geography_id, region_name, sales, store_date, store_name, store_id)
    values
        (?, ?, ?, ?, ?, ?)
```

Spring Data JPA DB DML operations

- 透過 **findById( )** 方法載入的物件為PersistenceContext 的管理，所以對其物件資料內容更新異動後，在 Transaction 交易結束後資料將會自動被更新

```
73    @ApiOperation(value = "Restful PUT: It updates an existing resource.")
74    @PutMapping(value = "/updateStoreInfo")
75    public ResponseEntity<StoreInfo> updateStoreInfo(@Validated @RequestBody StoreInfoVo storeInfoVo) {
76
77        StoreInfo storeInfo = storeInfoService.updateStoreInfo(storeInfoVo);
78
79        return ResponseEntity.ok(storeInfo);
80    }
```

```
41    @Transactional
42    public StoreInfo updateStoreInfo(StoreInfoVo storeInfoVo) {
43        // 搭配使用@Transactional 查詢載入的物件受到 PersistenceContext 的管理
44        Optional<StoreInfo> optStoreInfo = storeInfoDao.findById(storeInfoVo.getStoreID());
45        StoreInfo storeInfo = null;
46        if(optStoreInfo.isPresent()) {
47            storeInfo = optStoreInfo.get();
48            storeInfo.setStoreName(storeInfoVo.getStoreName());
49            storeInfo.setStoreDate(storeInfoVo.getStoreDate());
50            storeInfo.setSales(storeInfoVo.getSales());
51            storeInfo.setGeographyID(storeInfoVo.getGeographyVo().getGeographyID());
52            storeInfo.setRegionName(storeInfoVo.getGeographyVo().getRegionName());
53        }
54
55        return storeInfo;
56    }
```

```
Hibernate:
    update
        store_information
    set
        geography_id=?,
        region_name=?,
        sales=?,
        store_date=?,
        store_name=?
    where
        store_id=?
```

Spring Data JPA DB DML operations

- 透過 **deleteById( )** 刪除單一筆資料

```
93      @ApiOperation(value = "Restful DELETE: It deletes the resource.")
94      @DeleteMapping(value = "/deleteStoreInfo")
95      public ResponseEntity<Long> deleteStoreInfo(@RequestParam("storeID") long storeID) {
96          storeInfoService.deleteStoreInfo(storeID);
97
98          return ResponseEntity.ok(storeID);
99      }
```

```
11
12  @Service
13  public class StoreInfoService {
14
15      @Autowired
16      private StoreInfoDao storeInfoDao;
17
18      public void deleteStoreInfo(long storeID) {
19          storeInfoDao.deleteById(storeID);
20      }
21
22  }
23
```

```
Hibernate:
    select
        storeinfo0_.store_id as store_id1_0_0_,
        storeinfo0_.geography_id as geograph2_0_0_,
        storeinfo0_.region_name as region_n3_0_0_,
        storeinfo0_.sales as sales4_0_0_,
        storeinfo0_.store_date as store_da5_0_0_,
        storeinfo0_.store_name as store_na6_0_0_
    from
        store_information storeinfo0_
    where
        storeinfo0_.store_id=?
Hibernate:
    delete
    from
        store_information
    where
        store_id=?
```

Spring Data JPA Batch Inserts

- 我們可以通過將多筆資料進行「批次處理」新增、修改、刪除來提高對資料庫更新的效能

- application.yml

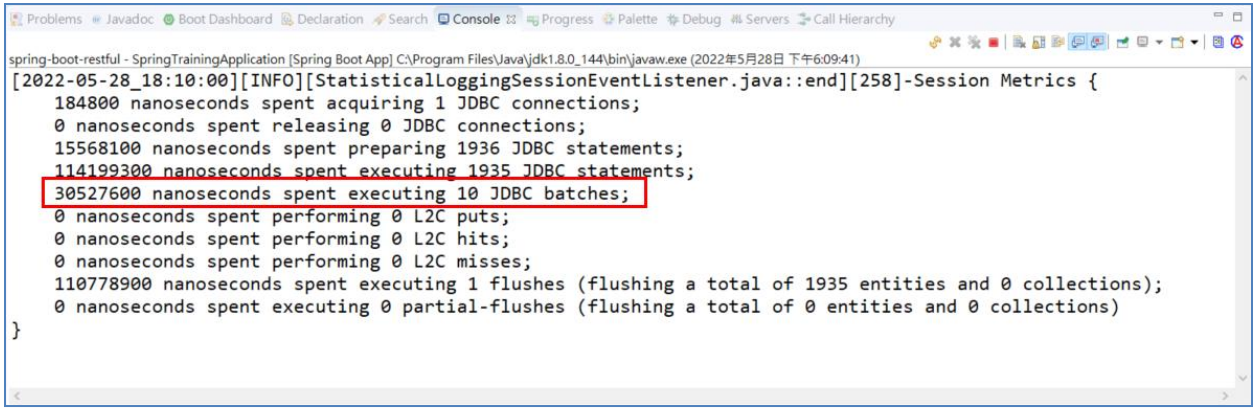
```yaml
spring:
  profiles:
    active: local
  h2:
    console:
      enabled: true
      path: /h2
  datasource:
    url: jdbc:h2:mem:test_db
    username: root
    password: root
    driverClassName: org.h2.Driver
    hikari:
      maximum-pool-size: 2
      connection-timeout: 30000
  jpa:
    hibernate:
      # 不透過@Entity自動化工程建立資料表,經由schema.sql建立資料表
      ddl-auto: none
    properties:
      hibernate:
        dialect: org.hibernate.dialect.H2Dialect
        format_sql: true
        show-sql: true
        # 查看 JDBC statements、JDBC batches 執行時間統計訊息
        generate_statistics: true
        # The order_inserts property tells Hibernate to
        order_inserts: true
        # Can apply the same approach to deletes and up
        order_updates: true
        jdbc:
          batch_size: 200
```

Spring Data JPA Batch Inserts

- 透過 saveAll、deleteAll 達到批次新增、更新、刪除處理資料庫

```java
@Service
public class UserService {

    @Autowired
    private UserDao userDao;

    public List<User> batchInsertUsers(int endChar) {
        List<User> users = new ArrayList<>();
        for(int i = 65 ; i < endChar ; i++) {
            User user = User.builder().userName((char) i).build();
            users.add(user);
        }

        // 透過 saveAll、deleteAll 達到批次新增、更新、刪除處理資料庫
        return userDao.saveAll(users);
    }

}

```

- 查看 JDBC statements、JDBC batches 執行時間統計訊息



Spring Data JPA Entity Lifecycle

- JPA中的 Entity 物件依儲存時期的不同狀態分為四種：

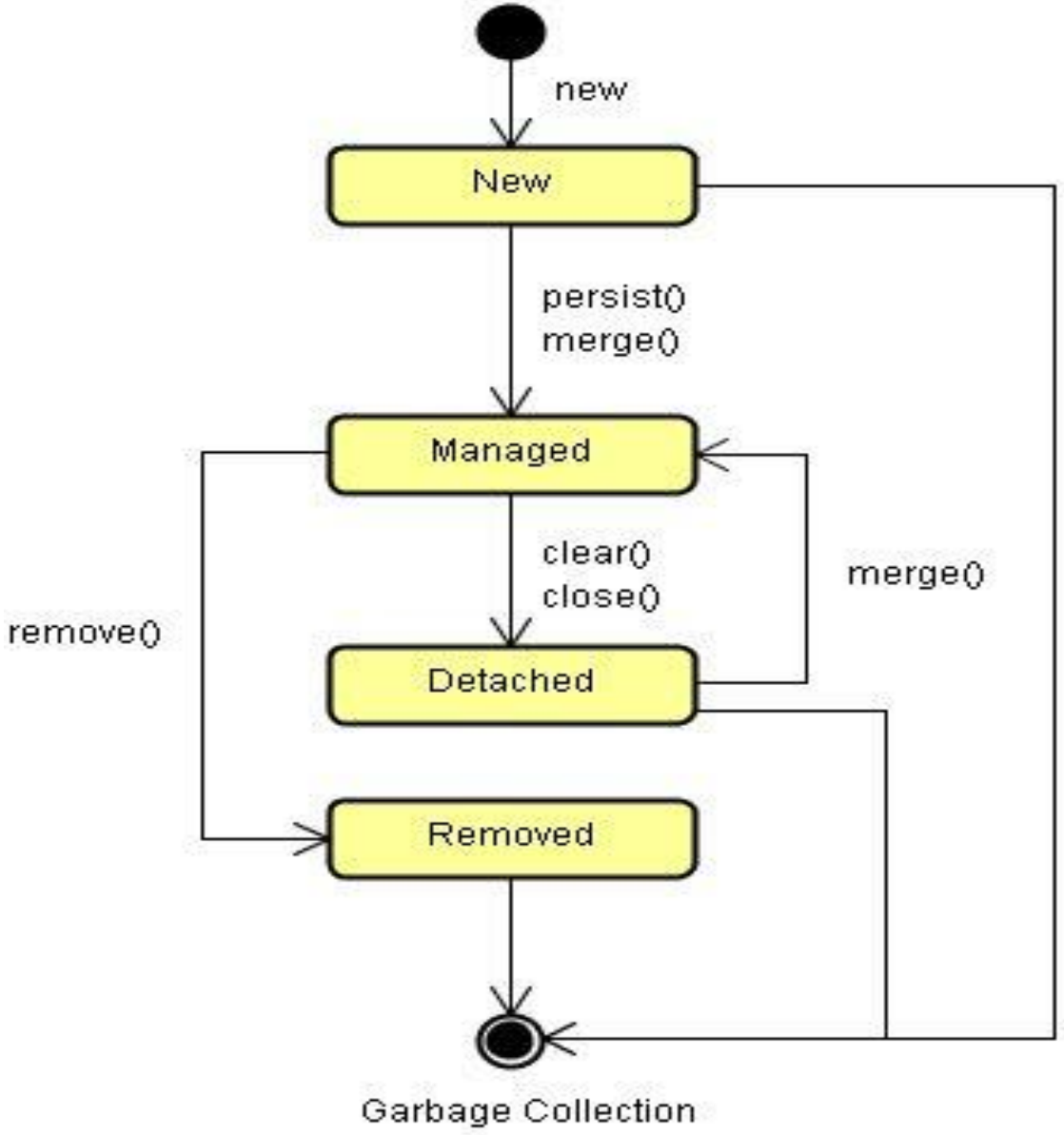
  1. **Transient**：新建立 **new Entity** 物件還沒有與資料庫發生任何的關係

  2. **Managed**： Transient 狀態的物件使用 EntityManager 的 **persist( )** 或 **merge( )** 方法加以儲存、合併，或是使用 **find( )** 從資料庫載入資料，在 EntityManager 實例關閉（close）或交易確認之後，資料庫中對應的資料也會跟著更新。
  (Managed狀態的Entity是在 **PersistenceContext** 的管理之中)

  3. **Detached**：脫離 EntityManager 實例 的管理，例如在使 find( ) 方法查詢到資料並封裝為物件之後，將 **EntityManager Closing** 實例關閉，則物件由 Managed 狀態變為 Detached 狀態，對 Detached 狀態的物件之任何屬性變動，都不會對資料庫中的資料造成任何的作用

  4. **Removed**：EntityManager 實例的 **remove( )** 方法刪除資料，Managed 狀態的物件由於失去了對應的資料，則它會成為 Removed 狀態

Spring Data JPA Entity Lifecycle



Spring Data JPA Data Locking

- 當資料庫有兩筆交易同時存取和更新同一筆資料時，就可能會發生更新資料遺失 lost update 的問題

1. **Optimistic Locking** (樂觀鎖定)

將資料表中的每一筆資料加入一個 **version 欄位記錄資料版號**，在讀取資料時連同版本號一同讀取，並在更新資料時比對差異讀取時的版本號與資料庫中的版本號，如果相等資料庫中的版本號則予以更新並且遞增版本號，如果小於資料庫中的版本號就丟出例外錯誤 (**ObjectOptimisticLockingFailureException**)

```
15 @Entity
16 @Table(name = "LOCKING", schema="LOCAL")
17 public class Locking {
18
19     @Id
20     @Column(name = "COLUMN_ID")
21     private long columnID;
22
23     @Column(name = "COLUMN_DATA")
24     private String columnData;
25
26     @Version
27     @Column(name = "VERSION")
28     private int version;
29
30 }
31
```

```
Hibernate:
    select
        locking0_.column_id as column_id1_2_0_,
        locking0_.column_data as column_data2_2_0_,
        locking0_.version as version3_2_0_
    from
        local.locking locking0_
    where
        locking0_.column_id=?
Hibernate:
    update
        local.locking
    set
        column_data=?,
        version=?
    where
        column_id=?
        and version=?
```

Spring Data JPA Data Locking

2. **Pessimistic Locking** (悲觀鎖定)

本次交易於**整個資料庫交易期間**進行資料鎖定，以防止同時間其它交易讀取或寫入直到本次交易關閉結束 (不建議此鎖定方式對系統效能影響大)

```
10
11 @Service
12 public class DataLockingService {
13
14     @Autowired
15     private LockingDao lockingDao;
16
17     @Transactional
18     public List<Locking> findByColumnIDIn(Collection<Long> columnIDs) {
19
20         List<Locking> lockings = lockingDao.findByColumnIDIn(columnIDs);
21         for(Locking locking : lockings) {
22             System.out.println(locking);
23         }
24
25         return lockings;
26     }
27
28 }
29
```

```
Hibernate:
    select
        locking0_.column_id as column_id1_2_,
        locking0_.column_data as column_data2_2_,
        locking0_.version as version3_2_
    from
        local.locking locking0_
    where
        locking0_.column_id in (
            ? , ? , ?
        ) for update
```

```
 9
10 public interface LockingDao extends JpaRepository<Locking, Long> {
11
12     @Lock(LockModeType.PESSIMISTIC_WRITE)
13     List<Locking> findByColumnIDIn(Collection<Long> columnIDs);
14
15 }
16
```

Spring Data JPA Transaction Manager

- 在 Server Layer 的方法上透過注釋 **@Transactional** 就可以進行交易事務的管理，統一在進入方法開始時取得建立資料庫連線，並且在最後方法結束時關閉資料庫連線

- Transactional **rollbackFor**

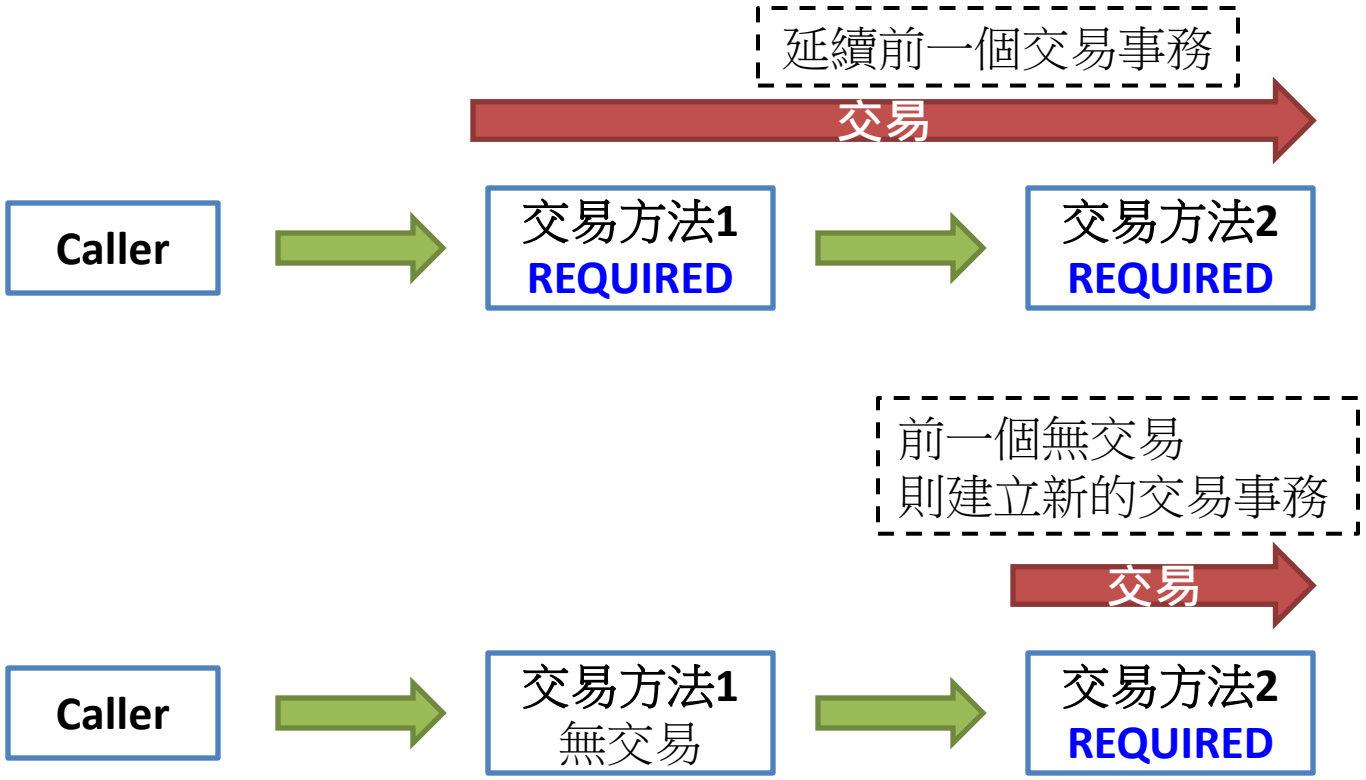
  當執行 service 方法時若發生例外錯誤，交易事務管理將會自動進行交易資料倒回 (rollback) 的動作；Transactional 預設 rollback RuntimeException (unCheck Exception)，可再透過 rollbackFor 屬性指定要rollback的例外錯誤 ex：Exception (Check Exception)

```
17⊖    @Transactional
18 //  @Transactional(rollbackFor = Exception.class)
19     public StoreInfo transactionalRollbackFor(StoreInfoVo storeInfoVo) throws Exception {
20         StoreInfo storeInfo = StoreInfo.builder()
21                 .storeID(storeInfoVo.getStoreID())
22                 .storeName(storeInfoVo.getStoreName())
23                 .storeDate(storeInfoVo.getStoreDate())
24                 .sales(storeInfoVo.getSales())
25                 .geographyID(storeInfoVo.getGeographyID()).build();
26
27         storeInfoOracleDao.save(storeInfo);
28
29         // throw RuntimeException (unCheck Exception)
30         // Transactional 預設 rollback RuntimeException
31         Integer.parseInt("ABC");
32
33         // throw Exception(Check Exception)
34         // Transactional 不會 rollback Exception 須設置 rollbackFor
35         // FileReader fr = new FileReader("C:/User.txt");
36         // fr.close();
37
38         return storeInfo;
39     }
```

Spring Data JPA Transaction Manager

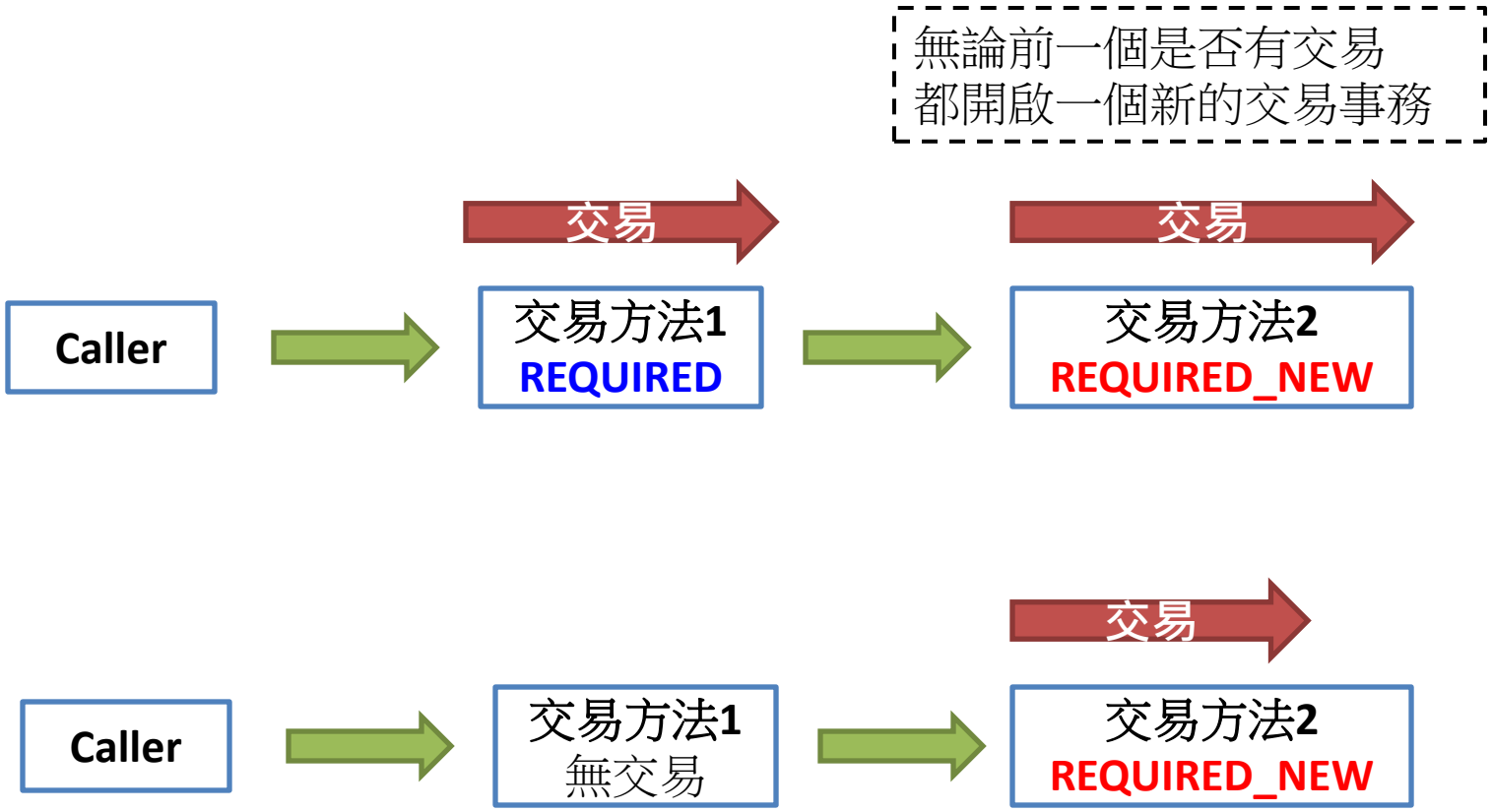
- Transactional **propagation** (交易傳播行為)

(1) **REQUIRED** (預設)：支援現在的交易，如果沒有的話就建立一個新的交易



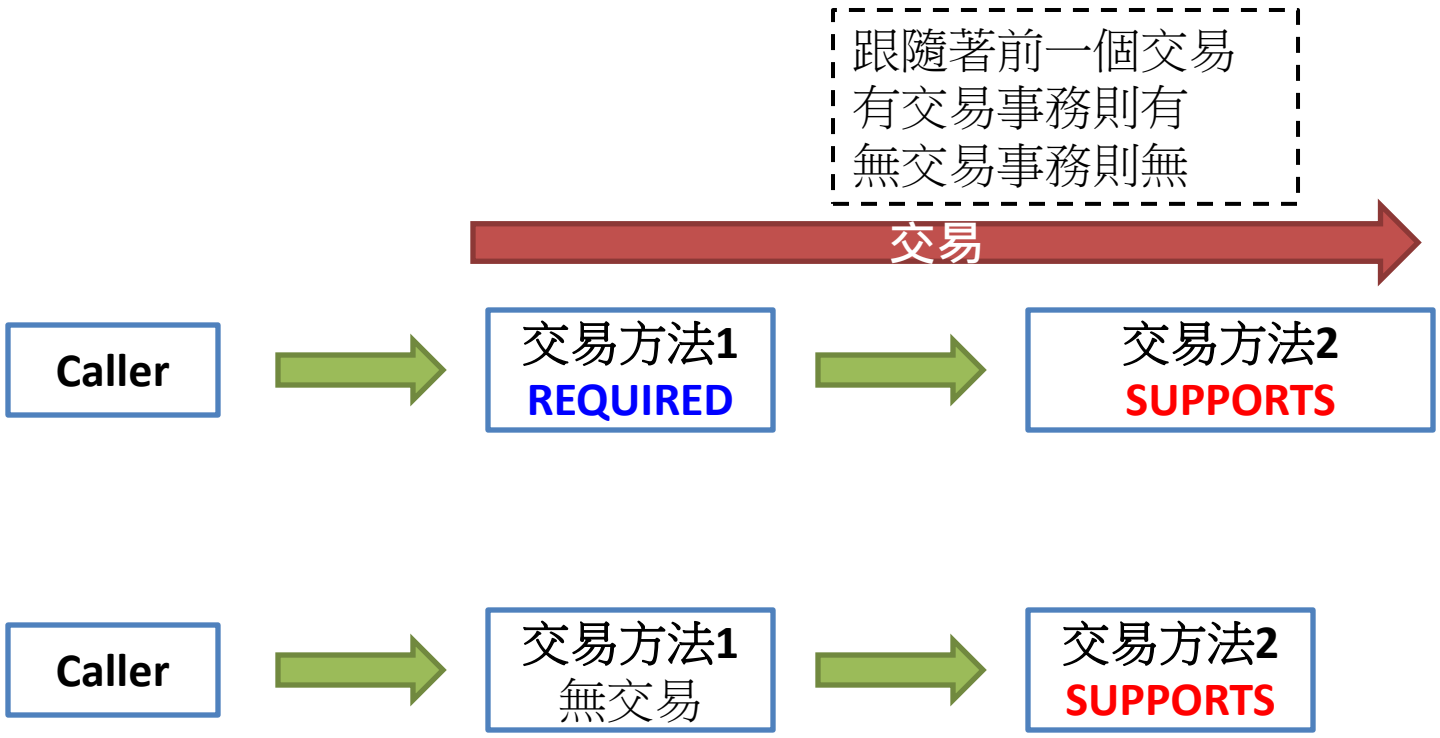
Spring Data JPA Transaction Manager

(2) REQUIRES_NEW：建立一個新的交易，如果現存一個交易的話就先暫停，並啟始一個新的交易來執行



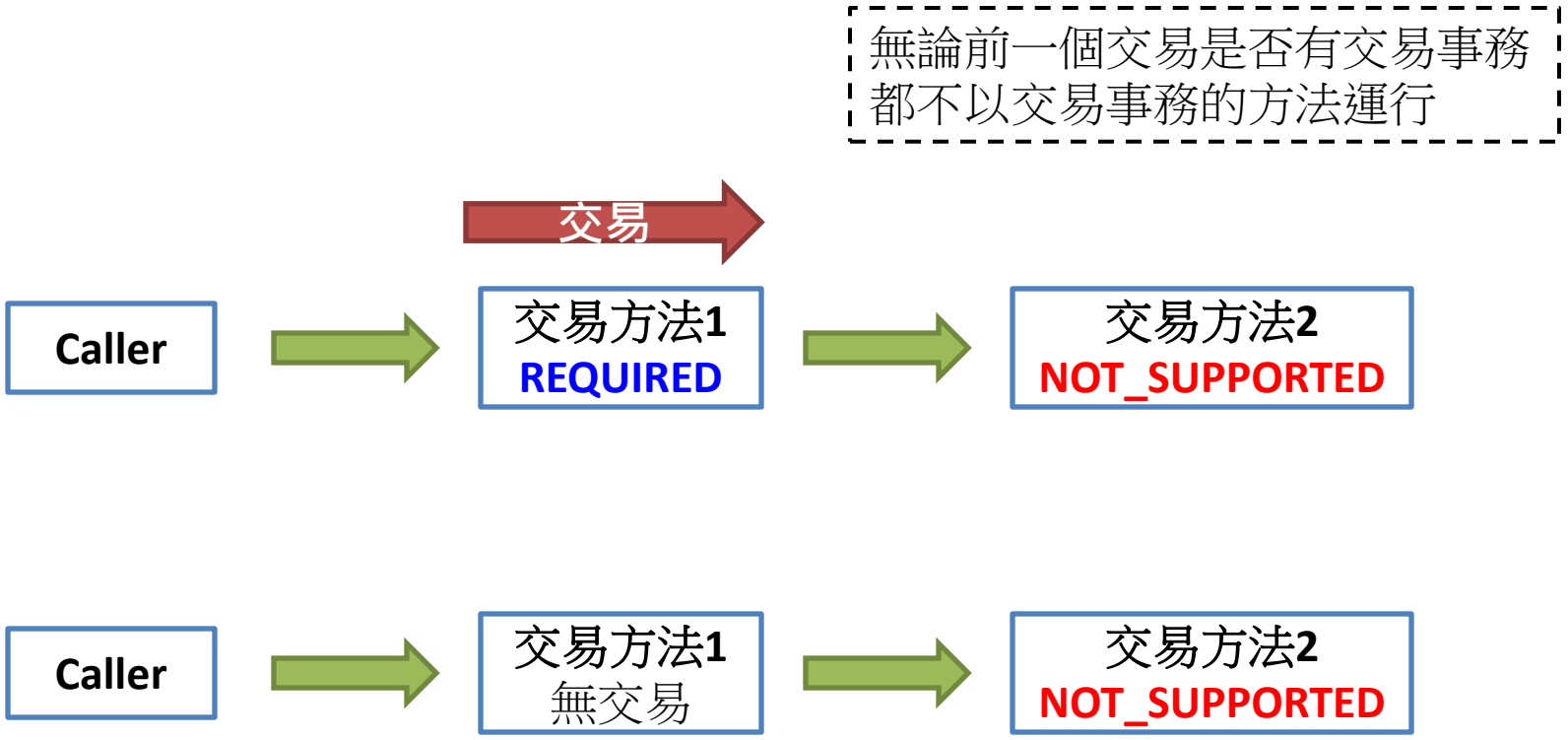
Spring Data JPA Transaction Manager

(3) SUPPORTS：支援現在的交易，如果沒有的話就以非交易的方式執行



Spring Data JPA Transaction Manager

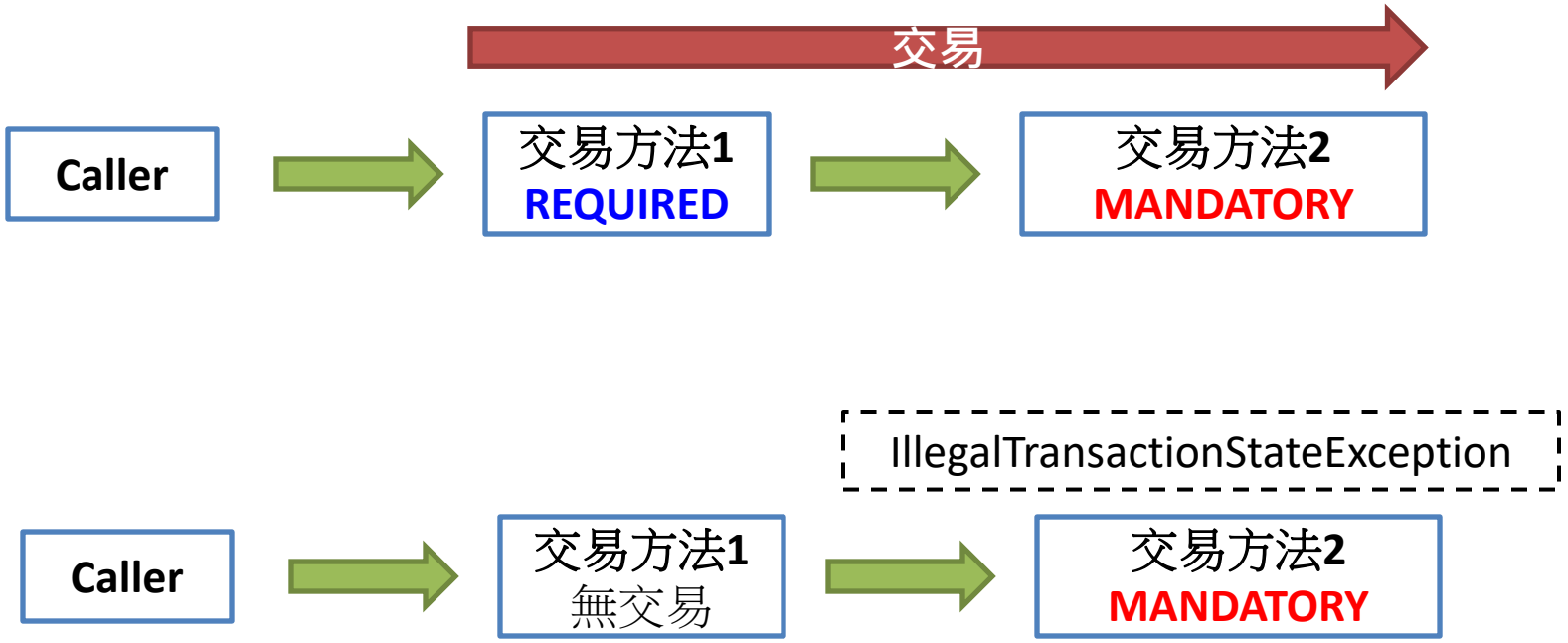
(4) NOT_SUPPORTED：指出不應在交易中進行，如果有的話就暫停現存的交易



Spring Data JPA Transaction Manager

(5) MANDATORY：方法必須在一個現存的交易中進行，否則丟出例外錯誤

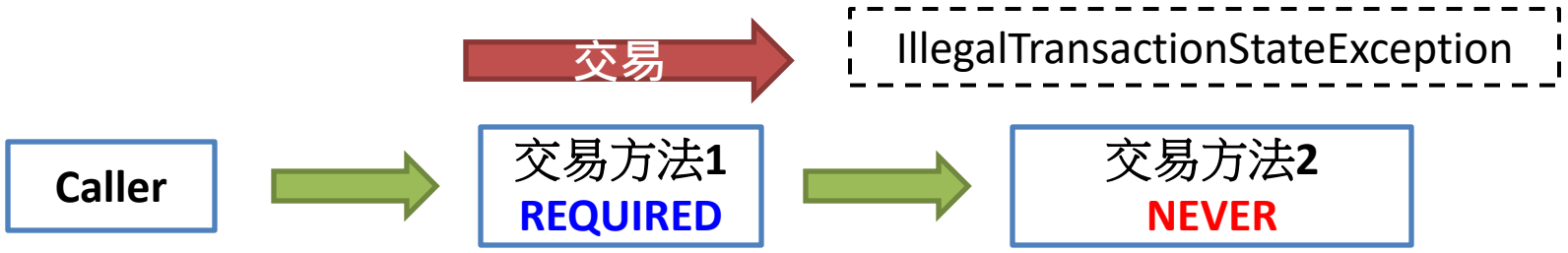
No existing transaction found for transaction marked with propagation 'mandatory'



Spring Data JPA Transaction Manager

(6) NEVER：指出不應在交易事務中進行否則丟出例外錯誤

Existing transaction found for transaction marked with propagation 'never'



Spring Data JPA Transaction Manager

(7) NESTED：在一個巢狀的交易中進行 (支援交易儲存點 savepoint)，如果不是的話則同 REQUIRED