Spring Boot

後端開發實戰教材

從零打造具備認證授權的書店 REST API，一路做到容器化部署

Spring Boot 3.5 | Java 17 | MySQL 8 | JWT | Docker | Postman

本教材以一個完整可執行的專案貫穿全書，你將學會：

- ✓ 設計並實作 GET / POST / PUT / PATCH / DELETE 五種 REST API
- ✓ 使用 Spring Security 與 JWT 完成認證（Authentication）與授權（Authorization）
- ✓ 透過 Hibernate / JPA 操作 MySQL 資料庫，理解 ORM 與連線設定
- ✓ 將專案打包成 Docker 映像檔，並以 Docker Compose 一鍵啟動整套服務
- ✓ 使用 Postman 匯入集合、管理變數、撰寫自動化測試腳本

版本 1.0　2026 年 8 月

隨附完整專案原始碼與 Postman 測試集合

前言

本教材的目標

這是一份「做中學」的 Spring Boot 後端開發教材。全書以一個完整、可直接執行的 **線上書店 REST API** 專案貫穿，學完後你將能夠：

- 使用 Spring Boot 建立分層架構的後端專案（Controller / Service / Repository）。
- 實作符合 REST 規範的 API，正確運用 `GET`、`POST`、`PUT`、`PATCH`、`DELETE` 五種 HTTP 方法與狀態碼。
- 透過 Hibernate / JPA 操作 MySQL 資料庫，理解 Entity 對映、衍生查詢與交易管理，並學會設定資料庫連線。
- 使用 Spring Security 搭配 JWT 實作註冊、登入、認證與角色授權。
- 將專案打包成 Docker 映像檔，並使用 Docker Compose 一鍵啟動「MySQL + API」整組服務。
- 使用 Postman 匯入測試集合、管理環境變數、撰寫自動化測試腳本並匯出分享。

適用對象與先備知識

本教材適合已具備 Java 基礎語法（類別、介面、集合、例外處理），想要系統性學習後端 API 開發的讀者。不需要有 Spring、SQL 或 Docker 的經驗——所有觀念都會從零開始說明。

開發環境與版本

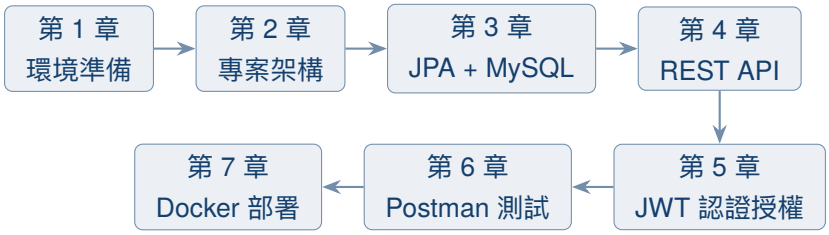
項目	版本	用途
Java (JDK)	17（或更新）	程式語言執行環境
Spring Boot	3.5.4	後端框架
Maven	3.9+	建置與相依套件管理
MySQL	8.x	關聯式資料庫
JJWT	0.12.6	JWT 產生與驗證函式庫
Docker	24+ / Desktop	容器化部署
Postman	最新版	API 測試工具

隨附檔案結構

教材資料夾的內容如下，建議搭配原始碼一邊閱讀一邊操作：

```
Springboot/
|-- bookstore-api/                # Spring Boot 專案（本書主角）
|   |-- src/main/java/...         # Java 原始碼（所有檔案皆有中文註解）
|   |-- src/main/resources/       # application.yml 設定檔
|   |-- pom.xml                   # Maven 設定
|   |-- Dockerfile                # Docker 映像檔建置腳本
|   `-- docker-compose.yml        # 一鍵啟動 MySQL + API
|-- postman/
|   |-- Bookstore-API.postman_collection.json    # Postman 測試集合
|   `-- Bookstore-Local.postman_environment.json # Postman 環境檔
|-- latex/                        # 本教材的 LaTeX 原始檔
`-- springboot-tutorial.pdf       # 本教材 PDF
```

學習路線圖



如何使用本教材

- **跟著做**：每一章都有可實際執行的指令與程式碼，強烈建議邊讀邊操作。
- **讀註解**：專案中所有程式碼都加上了詳細中文註解，書中列出的程式碼与專案檔案完全一致。
- **觀察輸出**：設定檔已開啟 `show-sql`，啟動後可在主控台看到 Hibernate 產生的每一條 SQL，眼見為憑。
- **動手實驗**：各章穿插「實驗」建議（例如故意送錯資料觀察 400 回應），是加深理解的最好方式。

目錄

前言 i

第 1 章　開發環境準備 1

1.1　工具總覽 . . . 1

1.2　安裝 JDK 17 . . . 1

1.3　安裝 Maven . . . 1

1.4　準備 MySQL 8 . . . 2

1.4.1　方式 A：macOS 使用 Homebrew . . . 2

1.4.2　方式 B：Windows 官方安裝程式 . . . 2

1.4.3　方式 C：使用 Docker（推薦） . . . 2

1.5　安裝 Docker Desktop . . . 3

1.6　安裝 Postman . . . 3

1.7　選擇 IDE . . . 3

1.8　環境檢查清單 . . . 4

第 2 章　建立專案與認識架構 5

2.1　Spring Boot 是什麼？ . . . 5

2.2　用 Spring Initializr 產生專案骨架 . . . 5

2.3　專案目錄結構 . . . 6

2.4　pom.xml 完整解說 . . . 7

2.5　程式進入點 . . . 11

2.6　三層式架構：一個請求的旅程 . . . 12

2.7　第一次啟動專案 . . . 13

第 3 章　Hibernate / JPA 與 MySQL 資料庫 14

3.1　ORM、JPA、Hibernate、Spring Data JPA 的關係 . . . 14

3.2 準備資料庫 . . . 14
3.3 資料庫連線設定：application.yml . . . 15
3.3.1 JDBC 連線字串解析 . . . 17
3.3.2 ddl-auto：Hibernate 如何管理資料表結構 . . . 17
3.4 Entity 對映：Book . . . 18
3.5 Entity 對映：User 與 Role . . . 22
3.6 Repository：宣告介面就完成資料存取層 . . . 26
3.6.1 衍生查詢：用方法名稱寫查詢 . . . 28
3.7 眼見為憑：觀察 Hibernate 的 SQL . . . 29
3.8 交易管理：@Transactional . . . 29

第 4 章 REST API 設計與實作 30

4.1 REST 核心概念 . . . 30
4.2 五種 HTTP 方法完整解析 . . . 30
4.2.1 PUT 與 PATCH 的差異（高頻考點） . . . 31
4.3 HTTP 狀態碼 . . . 31
4.4 本專案 API 總覽 . . . 32
4.5 DTO 與輸入驗證 . . . 32
4.6 Service：商業邏輯層 . . . 37
4.7 Controller：控制層 . . . 41
4.8 統一例外處理 . . . 45
4.9 用 curl 快速測試 . . . 51

第 5 章 JWT 認證與授權 52

5.1 認證與授權：先分清楚兩件事 . . . 52
5.2 為什麼用 Token 而不是 Session？ . . . 52
5.3 JWT 結構解剖 . . . 53
5.4 認證流程全景圖 . . . 54
5.5 實作 1：JwtService——產生與驗證 token . . . 54
5.6 實作 2：CustomUserDetailsService——載入使用者 . . . 57
5.7 實作 3：JwtAuthenticationFilter——每個請求的守門員 . . . 58
5.8 實作 4：SecurityConfig——規則總部 . . . 60

5.9 實作 5：註冊與登入 API 64
5.10 種子資料與測試帳號 70
5.11 完整流程測試（curl 版） 72

第 6 章 使用 Postman 測試 API **74**

6.1 為什麼用 Postman？ 74
6.2 匯入測試集合 74
6.3 Postman 變數機制 74
6.4 集合結構導覽 75
6.5 跟著做：完整測試流程 75
6.5.1 步驟 1：登入並自動儲存 token 75
6.5.2 步驟 2：走一遍 CRUD 76
6.5.3 步驟 3：錯誤情境 76
6.6 撰寫 Tests：常用 API 速查 77
6.7 Collection Runner：一鍵跑完整套測試 77
6.8 如何匯出自己的 Collection 77

第 7 章 Docker 容器化部署 **79**

7.1 為什麼需要 Docker？ 79
7.2 Dockerfile：逐行解說 79
7.3 建置映像檔：docker build 82
7.4 執行容器：docker run 82
7.5 Docker Compose：一鍵啟動整組服務 83
7.6 常用指令速查 85
7.7 完整驗證：從零到整組服務上線 85

附錄 A 常見問題排除（FAQ） **87**

A.1 啟動與資料庫連線 87
A.2 JWT 與權限 87
A.3 Maven 與 Docker 88

附錄 B 指令速查表 **89**

B.1 Maven 89

B.2 MySQL 89
B.3 curl 測試 API 89

附錄 C 延伸學習方向 **90**

第 1 章　開發環境準備

工欲善其事，必先利其器。本章逐一安裝並驗證開發所需的工具。如果某項工具已經安裝，直接執行對應的「驗證指令」確認版本即可。

1.1　工具總覽

工具	用途	驗證指令
JDK 17+	編譯與執行 Java 程式	`java -version`
Maven 3.9+	建置專案、管理相依套件	`mvn -v`
MySQL 8	關聯式資料庫	`mysql --version`
Docker Desktop	建置與執行容器	`docker --version`
Postman	測試 REST API	（圖形介面程式）
IDE	撰寫程式（IntelliJ IDEA / VS Code）	（圖形介面程式）

1.2　安裝 JDK 17

Spring Boot 3.x 最低需求為 Java 17。推薦安裝長期支援（LTS）版本：17 或 21。

macOS（使用 Homebrew）：

```
# 安裝 OpenJDK 17（Temurin 與 openjdk 擇一即可）
brew install openjdk@17

# 驗證安裝
java -version
# 預期輸出類似：
# openjdk version "17.0.12" 2024-07-16
```

Windows：至 `https://adoptium.net` 下載 Eclipse Temurin 17 的安裝檔，安裝時勾選「Set JAVA_HOME」與「Add to PATH」。

1.3　安裝 Maven

Maven 負責下載相依套件與建置專案。

```
# macOS
brew install maven

# 驗證安裝
mvn -v
# 預期輸出類似：
# Apache Maven 3.9.x
```

Maven Wrapper

用 Spring Initializr 建立的專案通常附有 `mvnw`（Maven Wrapper）腳本，執行 `./mvnw spring-boot:run` 會自動下載指定版本的 Maven，即使電腦沒安裝 Maven 也能建置專案，也確保團隊成員使用相同版本。

1.4 準備 MySQL 8

三種方式擇一即可。**最推薦方式 C（Docker）**——不污染本機環境，版本切換容易，砍掉重練也只要一行指令。

1.4.1 方式 A：macOS 使用 Homebrew

```
brew install mysql
brew services start mysql          # 啟動 MySQL 服務
mysql_secure_installation          # 設定 root 密碼（本教材假設為 root1234）
```

1.4.2 方式 B：Windows 官方安裝程式

至 https://dev.mysql.com/downloads/installer/ 下載 MySQL Installer，選擇「Server only」安裝，過程中設定 root 密碼。

1.4.3 方式 C：使用 Docker（推薦）

安裝好 Docker Desktop（見 1.5 節）後，一行指令啟動 MySQL：

```
# 啟動 MySQL 8.4 容器：
#   --name  容器名稱
#   -e      設定環境變數（root 密碼、預設建立的資料庫）
#   -p      連接埠對映「本機:容器」
#   -d      在背景執行
docker run --name bookstore-mysql \
  -e MYSQL_ROOT_PASSWORD=root1234 \
  -e MYSQL_DATABASE=bookstore_db \
  -p 3306:3306 \
  -d mysql:8.4

```

```
# 確認容器正在執行
docker ps

# （選用）進入 MySQL 命令列
docker exec -it bookstore-mysql mysql -uroot -proot1234
```

注意

本教材的預設連線設定為：主機 `localhost`、連接埠 `3306`、帳號 `root`、密碼 `root1234`、資料庫 `bookstore_db`。若你的環境不同，請記下你的設定，第 3 章會說明如何調整 `application.yml`。

1.5　安裝 Docker Desktop

至 https://www.docker.com/products/docker-desktop/ 下載並安裝 Docker Desktop（macOS / Windows 皆同）。安裝後啟動 Docker Desktop 應用程式，等待狀態列圖示顯示執行中，再驗證：

```
docker --version          # Docker version 27.x ...
docker run hello-world    # 會下載並執行測試容器，印出 Hello from Docker!
```

小提示

Docker 指令失敗顯示 `Cannot connect to the Docker daemon` 時，表示 Docker Desktop 沒有啟動——先打開 Docker Desktop 應用程式即可。

1.6　安裝 Postman

至 https://www.postman.com/downloads/ 下載安裝。Postman 是最流行的 API 測試工具，第 6 章會完整教學：匯入本教材附的測試集合、使用變數、撰寫自動化測試腳本。

1.7　選擇 IDE

- **IntelliJ IDEA**（推薦）：Community 免費版即可開發 Spring Boot。開啟專案時選擇 `bookstore-api` 資料夾（含 `pom.xml` 的那層），IDEA 會自動辨識為 Maven 專案並下載相依套件。
- **VS Code**：安裝兩個延伸模組——`Extension Pack for Java` 與 `Spring Boot Extension Pack`。

1.8 環境檢查清單

繼續之前，請確認以下指令都能正常執行

- `java -version` 顯示 17 或更新版本
- `mvn -v` 顯示 3.9 或更新版本
- MySQL 正在執行，且知道 root 密碼
- `docker run hello-world` 成功印出訊息
- Postman 可以開啟

第 2 章　建立專案與認識架構

本章認識 Spring Boot 專案的骨架：如何產生專案、目錄結構的意義、`pom.xml` 在管什麼，以及貫穿全書的「三層式架構」。

2.1　Spring Boot 是什麼？

Spring 是 Java 後端最主流的框架家族，但傳統 Spring 需要大量的 XML 與環境設定，光是「跑起來」就要耗掉新手一整天。**Spring Boot** 的使命就是消滅這些繁瑣設定：

- **自動設定（Auto-configuration）**：偵測 classpath 上的套件自動完成設定。加入 MySQL 驅動與 JPA，它就自動幫你設定好連線池與 Hibernate。
- **起步依賴（Starter）**：`spring-boot-starter-web` 一個座標，就帶入 Web 開發需要的整組相容套件。
- **內嵌伺服器**：Tomcat 直接包在程式裡，`java -jar` 即可啟動，不必另外安裝與部署到應用伺服器——這也是它適合容器化的原因。

約定優於配置（Convention over Configuration）

Spring Boot 為所有事情提供「合理的預設值」：連接埠預設 8080、設定檔預設讀 `application.yml`、JSON 轉換預設用 Jackson……你只需要設定「跟預設不同」的部分，因此設定檔可以非常精簡。

2.2　用 Spring Initializr 產生專案骨架

Spring 官方提供產生器網站 `https://start.spring.io`，本教材的專案就是用以下參數產生的（IntelliJ IDEA 的 New Project 也內建同樣功能）：

欄位	值
Project	Maven
Language	Java
Spring Boot	3.5.x
Group	`com.example`
Artifact	`bookstore-api`
Packaging	Jar
Java	17

Dependencies（相依套件）請加入以下五個：

套件	用途
Spring Web	開發 REST API（Spring MVC + 內嵌 Tomcat）
Spring Data JPA	ORM 資料存取（Hibernate + Repository）
Spring Security	認證與授權框架
Validation	Bean Validation 輸入驗證
MySQL Driver	MySQL JDBC 驅動程式

按下 Generate 會下載一個 zip，解壓縮後用 IDE 開啟即可。（JWT 函式庫 JJWT 不在 Initializr 清單中，需手動加入 `pom.xml`，見 2.4 節。）

小提示

本教材已附上完整專案 `bookstore-api/`，可以直接開啟使用，不必自己從 Initializr 重新產生。建議先直接使用附帶專案，學完全書後再練習從零開始建立一次。

2.3　專案目錄結構

```
bookstore-api/
|-- pom.xml                          # Maven 設定：相依套件與建置方式
|-- Dockerfile                       # Docker 映像檔建置腳本（第 7 章）
|-- docker-compose.yml               # 一鍵啟動 MySQL + API（第 7 章）
|-- .dockerignore                    # Docker 建置時要排除的檔案
`-- src/
    `-- main/
        |-- java/com/example/bookstore/
        |   |-- BookstoreApiApplication.java   # 程式進入點
        |   |-- config/              # 設定類別（Security 設定、種子資料）
        |   |-- controller/          # 控制層：接收 HTTP 請求
        |   |-- dto/                 # 資料傳輸物件（請求 / 回應格式）
        |   |-- entity/              # JPA Entity：對映資料表
        |   |-- exception/           # 自訂例外與全域例外處理
```

```
        |   |-- repository/        # 資料存取層：操作資料庫
        |   |-- security/          # JWT 與 Spring Security 元件
        |   `-- service/           # 商業邏輯層
        `-- resources/
            `-- application.yml    # 主設定檔（資料庫連線、JWT 參數）
```

套件位置的重要規則

`@SpringBootApplication` 所在的套件（`com.example.bookstore`）是「元件掃描」的起點：Spring 只會自動掃描**這個套件與其子套件**。若把類別放到外面（例如 `com.other`），Spring 會完全看不到它，出現「Bean 找不到」的錯誤時，第一件事就是檢查套件位置。

2.4　pom.xml 完整解說

`pom.xml` 是 Maven 的核心設定檔。先看完整內容（每一段都有註解），再整理幾個重要觀念。

程式碼 2.1: pom.xml——Maven 專案設定檔

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!--
  ============================================================================
  pom.xml : Maven 專案設定檔（Project Object Model）
  ============================================================================
  Maven 是 Java 生態圈最常用的「建置工具 + 相依套件管理工具」。
  這個檔案告訴 Maven 三件事：
    1. 專案的基本資訊（座標 GAV：groupId / artifactId / version）
    2. 專案需要哪些外部函式庫（<dependencies>）
    3. 如何編譯與打包專案（<build> 內的 plugins）

  常用指令：
    mvn spring-boot:run          啟動開發伺服器
    mvn clean package            清除舊檔並打包成可執行的 JAR
    mvn clean package -DskipTests  打包但跳過測試（加快速度）
  ============================================================================
-->
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/
    ↪ maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <!--
      繼承 Spring Boot 官方的 parent POM：
      - 統一管理數百個常用套件的「相容版本號」，避免版本衝突
      - 預先設定好編譯參數、資源處理、打包方式
      因此下方宣告 spring-boot-starter-* 時都「不需要」再寫版本號。
```

```
    -->
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.5.4</version>
        <relativePath/> <!-- 留空：一律從遠端倉庫下載，不找本機上層目錄 -->
    </parent>

    <!-- ===== 專案座標（GAV）：Maven 用來唯一識別這個專案 ===== -->
    <groupId>com.example</groupId>           <!-- 組織代號，慣例為反轉網域名稱 -->
    <artifactId>bookstore-api</artifactId>   <!-- 專案（產出物）名稱 -->
    <version>1.0.0</version>                 <!-- 專案版本 -->
    <name>bookstore-api</name>
    <description>Spring Boot 教學專案：線上書店後端 REST API（JWT + JPA + MySQL + Docker）</
    ↪ description>

    <properties>
        <!-- 指定 Java 版本（Spring Boot 3.x 最低需求為 Java 17） -->
        <java.version>17</java.version>
        <!-- JJWT：產生與驗證 JWT 的函式庫版本，集中在此方便日後升級 -->
        <jjwt.version>0.12.6</jjwt.version>
    </properties>

    <dependencies>

        <!-- ================ Web 層 ================ -->
        <!--
          spring-boot-starter-web：開發 REST API 的核心套件，包含
          - Spring MVC（@RestController、@GetMapping 等註解）
          - 內嵌 Tomcat 伺服器（不必另外安裝 Tomcat）
          - Jackson（Java 物件 <-> JSON 自動轉換）
        -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <!-- ================ 資料存取層 ================ -->
        <!--
          spring-boot-starter-data-jpa：包含
          - Hibernate（JPA 規範的實作，負責 ORM 物件關聯對映）
          - Spring Data JPA（只要宣告介面就自動產生 CRUD 實作）
          - HikariCP（高效能資料庫連線池）
        -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
```

```
76
77          <!--
78            MySQL JDBC 驅動程式：程式「執行期」才需要用它連線 MySQL，
79            編譯期不需要，所以 scope 設為 runtime。
80            版本號由 Spring Boot parent 統一管理，不必自己指定。
81          -->
82          <dependency>
83              <groupId>com.mysql</groupId>
84              <artifactId>mysql-connector-j</artifactId>
85              <scope>runtime</scope>
86          </dependency>
87
88          <!-- ================= 安全層 ================= -->
89          <!--
90            spring-boot-starter-security：Spring Security 框架，
91            提供認證（Authentication）與授權（Authorization）機制。
92            加入後所有 API 預設都會被保護，需在 SecurityConfig 中設定規則。
93          -->
94          <dependency>
95              <groupId>org.springframework.boot</groupId>
96              <artifactId>spring-boot-starter-security</artifactId>
97          </dependency>
98
99          <!--
100           JJWT（Java JWT）：業界常用的 JWT 函式庫，分成三個模組：
101           - jjwt-api     : 介面與類別定義（編譯期需要）
102           - jjwt-impl    : 實作細節（僅執行期需要）
103           - jjwt-jackson : 使用 Jackson 做 JSON 序列化（僅執行期需要）
104         -->
105         <dependency>
106             <groupId>io.jsonwebtoken</groupId>
107             <artifactId>jjwt-api</artifactId>
108             <version>${jjwt.version}</version>
109         </dependency>
110         <dependency>
111             <groupId>io.jsonwebtoken</groupId>
112             <artifactId>jjwt-impl</artifactId>
113             <version>${jjwt.version}</version>
114             <scope>runtime</scope>
115         </dependency>
116         <dependency>
117             <groupId>io.jsonwebtoken</groupId>
118             <artifactId>jjwt-jackson</artifactId>
119             <version>${jjwt.version}</version>
120             <scope>runtime</scope>
121         </dependency>
122
123         <!-- ================= 輸入驗證 ================= -->
```

```
        <!--
          spring-boot-starter-validation：Bean Validation（Hibernate Validator），
          提供 @NotBlank、@Size、@Email 等註解，
          搭配 Controller 參數上的 @Valid 自動驗證請求資料。
        -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-validation</artifactId>
        </dependency>

        <!-- ================= 測試 ================= -->
        <!-- 測試框架組合包：JUnit 5、Mockito、AssertJ、MockMvc 等 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <!-- Spring Security 的測試輔助工具（例如 @WithMockUser） -->
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <!--
              Spring Boot Maven Plugin：
              1. 提供 mvn spring-boot:run 指令直接啟動應用程式
              2. mvn package 時把所有相依套件一起打包成
                 「可獨立執行的 fat JAR」（java -jar 即可啟動，Docker 部署必備）
            -->
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

- **parent 繼承**：`spring-boot-starter-parent` 統一管理了數百個套件的相容版本，所以大多數 dependency 都不用（也不建議）自己寫版本號。
- **starter 起步依賴**：每個 `spring-boot-starter-*` 都是一組套件的集合，例如 `starter-web` 內含 Spring MVC、Jackson、內嵌 Tomcat。

- **scope 範圍**：`runtime` 表示編譯期不需要、執行期才需要（如 MySQL 驅動）；`test` 表示只有測試程式用得到。
- **spring-boot-maven-plugin**：讓 `mvn package` 打包出「可獨立執行的 fat JAR」，內含所有相依套件與內嵌 Tomcat。

2.5　程式進入點

程式碼 2.2: BookstoreApiApplication.java——應用程式進入點

```java
package com.example.bookstore;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

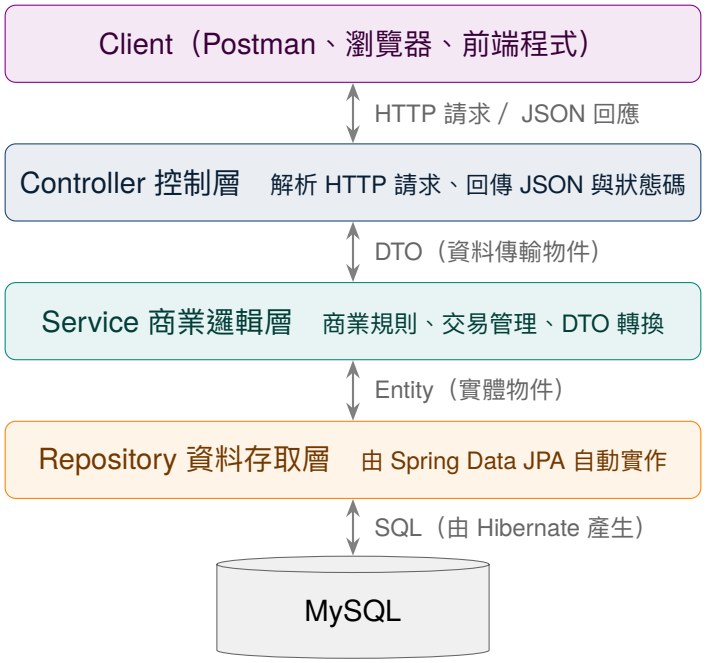
/**
 * ========================================================================
 * 應用程式進入點（Entry Point）
 * ========================================================================
 * {@code @SpringBootApplication} 是一個「三合一」的組合註解，等同於同時標上：
 *
 * 1. {@code @Configuration}
 *    ——— 表示此類別可以定義 Spring Bean（用 @Bean 方法）。
 *
 * 2. {@code @EnableAutoConfiguration}
 *    ——— 啟用「自動設定」：Spring Boot 會掃描 classpath 上有哪些套件，
 *        自動完成對應設定。例如：
 *        - 偵測到 spring-webmvc  → 自動設定內嵌 Tomcat 與 DispatcherServlet
 *        - 偵測到 mysql-connector-j + JPA → 自動設定資料庫連線池與 Hibernate
 *
 * 3. {@code @ComponentScan}
 *    ——— 自動掃描「本套件（com.example.bookstore）與其所有子套件」中
 *        標註 @Component / @Service / @Repository / @RestController
 *        的類別，建立成 Bean 並放入 Spring 容器，供依賴注入（DI）使用。
 *        （這就是為什麼其他類別都必須放在本套件底下的原因！）
 * ========================================================================
 */
@SpringBootApplication
public class BookstoreApiApplication {

    /**
     * main 方法：整個後端服務的啟動起點。
     *
     * SpringApplication.run() 依序會：
     *   1. 建立 Spring 容器（ApplicationContext）
     *   2. 掃描並實例化所有 Bean、完成依賴注入
     *   3. 啟動內嵌 Tomcat（預設監聽 8080 連接埠）開始接收 HTTP 請求
     */
```

```
39      public static void main(String[] args) {
40          SpringApplication.run(BookstoreApiApplication.class, args);
41      }
42  }
```

2.6　三層式架構：一個請求的旅程

本專案採用後端最經典的**三層式架構（Three-tier Architecture）**，每一層只做自己的事，彼此透過介面協作：



以「新增書籍」為例，一個請求的完整旅程：

1. Client 送出 `POST /api/books`，本文是書籍 JSON。
2. **Controller** 把 JSON 轉成 `BookRequest` DTO 並執行輸入驗證，然後呼叫 Service。
3. **Service** 檢查商業規則（ISBN 不可重複），把 DTO 轉成 `Book` Entity，交給 Repository；整個方法包在一個交易中。
4. **Repository**（Spring Data JPA 自動實作）呼叫 Hibernate，產生 `INSERT INTO books ...` 送往 MySQL。
5. 結果一路回傳：Entity → 轉成 `BookResponse` DTO → Controller 包裝成 `201 Created` 回應。

為什麼要分層？

關注點分離：HTTP 細節只存在 Controller、商業規則集中在 Service、SQL 細節只存在 Repository。好處是：改一層不動其他層、每一層可以獨立測試、程式碼各歸其位容易維護。

2.7　第一次啟動專案

確認 MySQL 已啟動（第 1 章），然後在 bookstore-api/ 目錄執行：

```
# 方式一：用 Maven 插件啟動（開發時最常用）
mvn spring-boot:run

# 方式二：先打包成 JAR 再執行（部署時的形式）
mvn clean package -DskipTests
java -jar target/bookstore-api-1.0.0.jar
```

啟動成功會看到類似輸出：

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/

 :: Spring Boot ::                (v3.5.4)

... Tomcat started on port 8080 (http) with context path '/'
... 已建立測試帳號 -> admin / Admin@123（管理員）、user / User@123（一般使用者）
... 已建立 3 本範例書籍
... Started BookstoreApiApplication in 3.2 seconds
```

開啟瀏覽器連到 http://localhost:8080/api/books，看到書籍列表的 JSON 就代表整條「API → 資料庫」的路都通了！

啟動失敗最常見的原因

Communications link failure：連不上 MySQL——確認 MySQL 有啟動、連接埠 3306 正確、帳號密碼與 application.yml 一致（第 3 章詳述）。Port 8080 was already in use：8080 被別的程式佔用——關掉佔用的程式，或在 application.yml 修改 server.port。

第 3 章　Hibernate / JPA 與 MySQL 資料庫

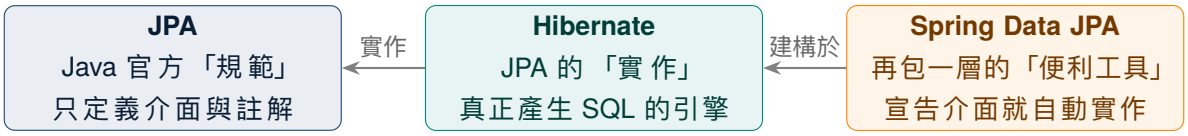
本章打通「Java 物件」與「資料庫資料」之間的橋樑：理解 ORM 的概念、設定資料庫連線、定義 Entity 與 Repository，並學會觀察 Hibernate 在背後產生的 SQL。

3.1　ORM、JPA、Hibernate、Spring Data JPA 的關係

寫過 JDBC 的人都知道痛苦所在：手寫 SQL 字串、手動把 `ResultSet` 一欄一欄搬進物件、自己管理連線與交易。**ORM（Object-Relational Mapping，物件關聯對映）**就是把這些苦工自動化：

Java 世界	資料庫世界
類別 `Book`	資料表 `books`
物件（一個 `Book` 實例）	資料列（一筆記錄）
欄位 `title`	欄位 `title VARCHAR(200)`
呼叫 `findAll()`	執行 `SELECT * FROM books`
呼叫 `save(book)`	執行 `INSERT` 或 `UPDATE`

三個常被混用的名詞，其實是三個層次：



重點觀念

我們寫程式時面對的是 **Spring Data JPA**（宣告 Repository 介面）與 **JPA 註解**（`@Entity`、`@Column`...）；執行期真正做事的是 **Hibernate**。這就是為什麼設定檔中會看到 `hibernate` 開頭的設定項。

3.2　準備資料庫

如果採用第 1 章「方式 C」的 Docker 指令，容器啟動時就自動建立了 `bookstore_db` 資料庫，本節可以跳過。

若是自行安裝的 MySQL，可以手動建立資料庫：

```sql
-- 登入 MySQL 後執行：
-- utf8mb4 是 MySQL 的完整 UTF-8 實作（支援表情符號等 4-byte 字元）
CREATE DATABASE bookstore_db
  CHARACTER SET utf8mb4
  COLLATE utf8mb4_unicode_ci;

-- 確認建立成功
SHOW DATABASES;
```

小提示

其實本專案的連線字串加了 `createDatabaseIfNotExist=true`，資料庫不存在時會自動建立，手動建立主要是為了讓你熟悉 SQL 操作。

3.3 資料庫連線設定：application.yml

Spring Boot 的主設定檔位於 `src/main/resources/application.yml`。先看完整內容，重點在 `datasource`（連線資訊）與 `jpa`（Hibernate 行為）兩段：

程式碼 3.1: application.yml——主設定檔

```yaml
# ================================================================================
# application.yml : Spring Boot 主設定檔
# ================================================================================
# YAML 格式：用「縮排」表示層級（一律用空格，不可用 Tab！）
#
# ${環境變數:預設值} 語法：
#   優先讀取「環境變數」，讀不到才用冒號後面的「預設值」。
#   好處：同一份設定檔，本機開發用預設值，Docker / 正式環境用環境變數覆寫，
#   完全不需要修改程式或設定檔。
# ================================================================================

spring:
  application:
    name: bookstore-api          # 應用程式名稱（顯示在日誌與監控工具中）

  # ================================================================================
  # 資料庫連線設定（DataSource）
  # ================================================================================
  datasource:
    # JDBC 連線字串，格式：jdbc:mysql://主機:連接埠/資料庫名稱?參數
    # 各參數說明：
    #   createDatabaseIfNotExist=true : 資料庫不存在時自動建立（教學方便用）
    #   useSSL=false                  : 本機開發不使用 SSL 加密連線
    #   allowPublicKeyRetrieval=true  : 允許取得伺服器公鑰（MySQL 8 新驗證機制需要）
    #   serverTimezone=Asia/Taipei    : 指定時區，避免 JVM 與 MySQL 時區不一致
```

```
    #   characterEncoding=UTF-8       : 連線編碼，確保中文不變成亂碼
    url: jdbc:mysql://${DB_HOST:localhost}:${DB_PORT:3306}/${DB_NAME:bookstore_db}?
    ↪ createDatabaseIfNotExist=true&useSSL=false&allowPublicKeyRetrieval=true&serverTimezone=
    ↪ Asia/Taipei&characterEncoding=UTF-8
    username: ${DB_USERNAME:root}          # 資料庫帳號（預設 root）
    password: ${DB_PASSWORD:root1234}      # 資料庫密碼（正式環境務必用環境變數！）
    # 驅動程式類別通常不用寫，Spring Boot 會根據 url 自動推斷
    # driver-class-name: com.mysql.cj.jdbc.Driver

  # ================================================================================
  # JPA / Hibernate 設定
  # ================================================================================
  jpa:
    hibernate:
      # ddl-auto：Hibernate 如何處理「資料表結構」
      #   none        完全不動資料表
      #   validate    只驗證 Entity 與資料表是否一致，不一致就啟動失敗
      #   update      比對差異並自動「新增」缺少的資料表 / 欄位（不會刪除）★教學採用
      #   create      每次啟動都刪掉重建（資料會消失！）
      #   create-drop 啟動時重建、關閉時刪除（適合測試）
      # 正式環境建議：validate + Flyway/Liquibase 做資料庫版本控管
      ddl-auto: update
    # 在主控台印出 Hibernate 產生的 SQL（學習利器：眼見為憑！）
    show-sql: true
    properties:
      hibernate:
        format_sql: true         # 將 SQL 排版美化，方便閱讀
    # 關閉 Open-Session-In-View：
    # 預設 true 會讓資料庫連線佔用到「整個 HTTP 請求結束」，
    # 容易造成連線池耗盡；後端 API 專案建議關閉（這也是官方警告的由來）
    open-in-view: false

server:
  port: 8080                     # 內嵌 Tomcat 監聽的連接埠（預設就是 8080）

# ================================================================================
# 自訂設定：JWT 相關（程式中以 @Value("${app.jwt.secret}") 讀取）
# ================================================================================
app:
  jwt:
    # 簽章密鑰：HS256 演算法要求至少 32 字元（256 bits）
    # ★ 正式環境絕對不要把密鑰寫在設定檔！請用環境變數 JWT_SECRET 覆寫
    secret: ${JWT_SECRET:bookstore-teaching-secret-key-change-me-0123456789ABCDEF}
    # token 有效毫秒數：3600000 ms = 1 小時
    expiration-ms: ${JWT_EXPIRATION_MS:3600000}

# ================================================================================
# 日誌等級設定
```

```yaml
# ================================================================================
logging:
  level:
    root: INFO
    # 想觀察 Spring Security 的過濾器運作細節時，可改成 DEBUG
    org.springframework.security: INFO
    # 顯示 SQL 綁定參數的實際值（搭配 show-sql 學習非常有用；正式環境請關閉）
    org.hibernate.orm.jdbc.bind: TRACE
```

3.3.1 JDBC 連線字串解析

組成 / 參數	說明
`jdbc:mysql://`	協定：使用 MySQL 的 JDBC 驅動
`localhost:3306`	資料庫主機與連接埠
`/bookstore_db`	要使用的資料庫名稱
`createDatabaseIfNotExist`	資料庫不存在時自動建立（教學方便用）
`useSSL=false`	本機開發不使用 SSL 加密
`allowPublicKeyRetrieval`	允許取得伺服器公鑰（MySQL 8 驗證機制需要）
`serverTimezone=Asia/Taipei`	明確指定時區，避免時間欄位差 8 小時
`characterEncoding=UTF-8`	連線編碼，避免中文變亂碼

「環境變數優先」的設定寫法

`${DB_HOST:localhost}` 的意思是：優先讀取環境變數 `DB_HOST`，讀不到才用預設值 `localhost`。好處：本機開發直接用預設值；到了 Docker 或正式環境，只要設定環境變數就能改變連線目標，**完全不用改程式與設定檔**。第 7 章的 docker-compose 就是靠這招把 `DB_HOST` 改成 `mysql`。

3.3.2 ddl-auto：Hibernate 如何管理資料表結構

選項	行為
`none`	完全不動資料表結構
`validate`	只驗證 Entity 與資料表是否一致，不一致就啟動失敗
`update`	比對差異，自動「新增」缺少的表與欄位（不會刪除舊欄位）——**本教材採用**
`create`	每次啟動都先刪表再重建（**資料會全部消失**）
`create-drop`	啟動時重建、程式關閉時刪除（適合自動化測試）

正式環境的作法

`update` 很適合教學與快速原型，但正式環境不應讓程式隨意改資料表。業界標準做法是 `ddl-auto: validate` 搭配 Flyway 或 Liquibase 這類「資料庫遷移（migration）工具」，讓每一次結構變更都有版本、可追蹤、可重播。

3.4 Entity 對映：Book

Entity 是 ORM 的核心——一個標註了 JPA 註解的 Java 類別，描述「物件如何對映到資料表」。先看本專案的主角 `Book`：

程式碼 3.2: entity/Book.java——書籍 Entity

```java
package com.example.bookstore.entity;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.PrePersist;
import jakarta.persistence.PreUpdate;
import jakarta.persistence.Table;
import jakarta.persistence.UniqueConstraint;

import java.math.BigDecimal;
import java.time.LocalDateTime;

/**
 * 書籍 Entity——對應資料庫中的 books 資料表。
 * 這是本教材 CRUD（Create / Read / Update / Delete）操作的主角。
 *
 * 對應關係示意：
 *   Java 類別 Book        <-->  資料表 books
 *   Java 欄位 title       <-->  資料表欄位 title (VARCHAR(200))
 *   一個 Book 物件         <-->  books 資料表的一列資料
 */
@Entity
@Table(name = "books", uniqueConstraints = {
        // ISBN 是書籍的國際標準編號，同一本書不可重複建檔
        @UniqueConstraint(name = "uk_books_isbn", columnNames = "isbn")
})
public class Book {

    /** 主鍵：由 MySQL AUTO_INCREMENT 自動產生 */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
35      private Long id;
36
37      /** 書名：NOT NULL、最長 200 字元（對應 VARCHAR(200) NOT NULL） */
38      @Column(nullable = false, length = 200)
39      private String title;
40
41      /** 作者 */
42      @Column(nullable = false, length = 100)
43      private String author;
44
45      /** ISBN 國際標準書號（例如 9786263336070） */
46      @Column(nullable = false, length = 20)
47      private String isbn;
48
49      /**
50       * 價格：金額請務必使用 BigDecimal！
51       * float / double 有二進位浮點數誤差（0.1 + 0.2 != 0.3），不適合金錢計算。
52       * precision = 10, scale = 2 → 對應 MySQL 的 DECIMAL(10,2)，
53       * 表示總共 10 位數，其中小數佔 2 位。
54       */
55      @Column(nullable = false, precision = 10, scale = 2)
56      private BigDecimal price;
57
58      /** 庫存數量 */
59      @Column(nullable = false)
60      private Integer stock;
61
62      /**
63       * 書籍簡介：允許 NULL。
64       * columnDefinition = "TEXT" → 直接指定 MySQL 欄位型別為 TEXT，
65       * 可儲存比 VARCHAR 更長的文字內容。
66       */
67      @Column(columnDefinition = "TEXT")
68      private String description;
69
70      /** 建立時間：INSERT 前由 @PrePersist 自動填入，之後不再變動 */
71      @Column(name = "created_at", nullable = false, updatable = false)
72      private LocalDateTime createdAt;
73
74      /** 最後更新時間：每次 UPDATE 前由 @PreUpdate 自動更新 */
75      @Column(name = "updated_at")
76      private LocalDateTime updatedAt;
77
78      /** JPA 規定的無參數建構子（Hibernate 建立物件時使用） */
79      public Book() {
80      }
81
82      /** 方便在程式中建立新書籍的建構子 */
```

```
    public Book(String title, String author, String isbn,
                BigDecimal price, Integer stock, String description) {
        this.title = title;
        this.author = author;
        this.isbn = isbn;
        this.price = price;
        this.stock = stock;
        this.description = description;
    }

    /** INSERT 之前自動執行：填入建立時間與初始更新時間 */
    @PrePersist
    protected void onCreate() {
        this.createdAt = LocalDateTime.now();
        this.updatedAt = this.createdAt;
    }

    /** UPDATE 之前自動執行：刷新最後更新時間 */
    @PreUpdate
    protected void onUpdate() {
        this.updatedAt = LocalDateTime.now();
    }

    // ========================= Getter / Setter =========================

    public Long getId() {
        return id;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    public String getIsbn() {
        return isbn;
    }
```

```
131
132     public void setIsbn(String isbn) {
133         this.isbn = isbn;
134     }
135
136     public BigDecimal getPrice() {
137         return price;
138     }
139
140     public void setPrice(BigDecimal price) {
141         this.price = price;
142     }
143
144     public Integer getStock() {
145         return stock;
146     }
147
148     public void setStock(Integer stock) {
149         this.stock = stock;
150     }
151
152     public String getDescription() {
153         return description;
154     }
155
156     public void setDescription(String description) {
157         this.description = description;
158     }
159
160     public LocalDateTime getCreatedAt() {
161         return createdAt;
162     }
163
164     public LocalDateTime getUpdatedAt() {
165         return updatedAt;
166     }
167 }
```

常用 JPA 註解整理：

註解	用途
`@Entity`	宣告為 JPA Entity，交由 Hibernate 管理
`@Table`	指定資料表名稱與約束（不寫則用類別名稱）
`@Id`	標示主鍵欄位
`@GeneratedValue`	主鍵產生策略；`IDENTITY` = 資料庫自動流水號
`@Column`	欄位屬性：名稱、長度、NOT NULL、精度等
`@Enumerated`	enum 的儲存方式；一律建議 `EnumType.STRING`
`@PrePersist`	INSERT 前的回呼（如自動填建立時間）
`@PreUpdate`	UPDATE 前的回呼（如自動更新修改時間）

啟動程式後，Hibernate（`ddl-auto: update`）會自動產生對應的資料表：

程式碼 3.3: Hibernate 依 Book Entity 自動產生的 DDL

```
create table books (
    id bigint not null auto_increment,
    title varchar(200) not null,
    author varchar(100) not null,
    isbn varchar(20) not null,
    price decimal(10,2) not null,
    stock integer not null,
    description TEXT,
    created_at datetime(6) not null,
    updated_at datetime(6),
    primary key (id),
    constraint uk_books_isbn unique (isbn)
) engine=InnoDB;
```

3.5 Entity 對映：User 與 Role

使用者資料同樣用 Entity 描述。注意兩個教學重點：資料表刻意取名 `users`（`user` 是 MySQL 保留字），以及角色 enum 用 `EnumType.STRING` 儲存。

程式碼 3.4: entity/Role.java——角色列舉

```
package com.example.bookstore.entity;

/**
 * 使用者角色（Role）列舉，用於「授權」（Authorization）判斷。
 *
 * 【Spring Security 慣例】
 * 角色名稱必須以 "ROLE_" 開頭。
 * 之後在授權規則中寫 hasRole("ADMIN") 時，
 * Spring Security 會自動加上前綴，比對 "ROLE_ADMIN"。
 *
```

```
 * 本專案的權限設計：
 *   - 未登入訪客      ：只能「查詢」書籍（GET）
 *   - ROLE_USER       ：可查詢、新增、修改書籍（GET / POST / PUT / PATCH）
 *   - ROLE_ADMIN      ：擁有全部權限，包含「刪除」書籍（DELETE）
 */
public enum Role {

    /** 一般使用者：可讀取、新增、修改書籍 */
    ROLE_USER,

    /** 管理員：額外擁有刪除書籍的權限 */
    ROLE_ADMIN
}
```

程式碼 3.5: entity/User.java——使用者 Entity

```
package com.example.bookstore.entity;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.EnumType;
import jakarta.persistence.Enumerated;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.PrePersist;
import jakarta.persistence.Table;
import jakarta.persistence.UniqueConstraint;

import java.time.LocalDateTime;

/**
 * 使用者 Entity（實體類別）——對應資料庫中的 users 資料表。
 *
 * 【JPA / Hibernate 核心觀念：ORM 物件關聯對映】
 * Entity 是「Java 物件」與「資料庫資料列」之間的橋樑。
 * Hibernate 會根據這個類別上的註解：
 *   1. 自動建立 / 更新資料表結構（依 application.yml 的 ddl-auto 設定）
 *   2. 查詢時，把結果集的每一列轉成一個 User 物件
 *   3. 儲存時，把 User 物件的內容轉成 INSERT / UPDATE 語句
 */
@Entity                     // 宣告這是一個 JPA Entity，交由 Hibernate 管理
@Table(
        name = "users",     // 指定資料表名稱。若不指定則預設用類別名稱 user，
                            // 但 user 是 MySQL 的保留字，容易出錯，因此改名為 users
        uniqueConstraints = {
                // 資料庫層級的「唯一性約束」：確保帳號與 Email 不重複
                @UniqueConstraint(name = "uk_users_username", columnNames = "username"),
```

```
                @UniqueConstraint(name = "uk_users_email",    columnNames = "email")
        }
)
public class User {

    /** 主鍵（Primary Key） */
    @Id
    // IDENTITY 策略：主鍵交給 MySQL 的 AUTO_INCREMENT 自動產生流水號
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    /** 登入帳號：NOT NULL、最長 50 字元 */
    @Column(nullable = false, length = 50)
    private String username;

    /**
     * 密碼：儲存的是 BCrypt「雜湊值」，絕對不能存明碼！
     * BCrypt 雜湊字串長度約 60 字元，欄位保留 100 字元較保險。
     */
    @Column(nullable = false, length = 100)
    private String password;

    /** 電子信箱 */
    @Column(nullable = false, length = 100)
    private String email;

    /**
     * 角色：使用列舉（enum）型別。
     *
     * EnumType.STRING → 資料庫實際儲存 "ROLE_USER" 這樣的「文字」。
     * 若用預設的 EnumType.ORDINAL 會儲存 0、1 這種「順序數字」，
     * 一旦日後調整 enum 的宣告順序，舊資料的意義就全部錯亂，
     * 因此實務上幾乎一律使用 STRING。
     */
    @Enumerated(EnumType.STRING)
    @Column(nullable = false, length = 20)
    private Role role;

    /**
     * 建立時間。
     * updatable = false → 之後執行 UPDATE 時，Hibernate 不會更動這個欄位。
     */
    @Column(name = "created_at", nullable = false, updatable = false)
    private LocalDateTime createdAt;

    /**
     * 【JPA 規定】Entity 一定要有「無參數建構子」。
     * Hibernate 從資料庫讀取資料時，會先用它建立空物件，再逐一填入欄位值。
```

```
     */
    public User() {
    }

    /** 方便在程式中建立新使用者的建構子 */
    public User(String username, String password, String email, Role role) {
        this.username = username;
        this.password = password;
        this.email = email;
        this.role = role;
    }

    /**
     * 【生命週期回呼 Lifecycle Callback】
     * @PrePersist：在 Hibernate 執行 INSERT「之前」自動被呼叫，
     * 這裡用來自動填入建立時間，不需要每次手動設定。
     */
    @PrePersist
    protected void onCreate() {
        this.createdAt = LocalDateTime.now();
    }

    // ========================= Getter / Setter =========================
    // Hibernate 與 Jackson（JSON 轉換）都是透過 getter/setter 存取欄位。
    // 實務上許多團隊會用 Lombok 的 @Getter / @Setter 省去這些樣板程式碼；
    // 教學專案為了完整呈現原貌，選擇手動撰寫。

    public Long getId() {
        return id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getEmail() {
```

```
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public Role getRole() {
        return role;
    }

    public void setRole(Role role) {
        this.role = role;
    }

    public LocalDateTime getCreatedAt() {
        return createdAt;
    }
}
```

3.6 Repository：宣告介面就完成資料存取層

Spring Data JPA 最神奇的地方：**只要宣告介面、繼承 `JpaRepository`**，啟動時就會自動產生實作，完全不用寫 SQL 或實作類別。

程式碼 3.6: repository/BookRepository.java

```
package com.example.bookstore.repository;

import com.example.bookstore.entity.Book;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

import java.util.List;

/**
 * 書籍 Repository（資料存取層 / DAO）。
 *
 * 【Spring Data JPA 的魔法】
 * 只要「繼承 JpaRepository 介面」，完全不用寫實作類別，
 * Spring 會在啟動時自動產生實作，並內建以下常用方法：
 *
 *   findAll()              查詢全部          → SELECT * FROM books
 *   findAll(Pageable)      分頁查詢          → SELECT ... LIMIT ?, ?
 *   findById(id)           依主鍵查詢        → SELECT ... WHERE id = ?
 *   save(entity)           新增或更新        → INSERT 或 UPDATE（有 id 就更新）
```

```
 *   deleteById(id)        依主鍵刪除          → DELETE ... WHERE id = ?
 *   count()               計算筆數            → SELECT COUNT(*)
 *   existsById(id)        是否存在            → SELECT COUNT(*) ... WHERE id = ?
 *
 * 泛型參數：JpaRepository<Entity 類別, 主鍵型別>
 */
public interface BookRepository extends JpaRepository<Book, Long> {

    /**
     * 【衍生查詢 Derived Query】
     * Spring Data JPA 會解析「方法名稱」自動產生查詢：
     * existsByIsbn → SELECT COUNT(*) > 0 FROM books WHERE isbn = ?
     * 用於新增書籍時檢查 ISBN 是否已存在。
     */
    boolean existsByIsbn(String isbn);

    /**
     * 檢查「其他書籍」是否已使用某 ISBN。
     * 更新書籍時要排除自己，否則自己的 ISBN 也會被判定為重複。
     * 產生的查詢條件：WHERE isbn = ? AND id <> ?
     */
    boolean existsByIsbnAndIdNot(String isbn, Long id);

    /**
     * 【JPQL 自訂查詢】
     * 當方法名稱不足以表達複雜條件時，可用 @Query 撰寫 JPQL。
     *
     * 注意：JPQL 查詢的對象是「Entity 與其屬性」（Book、b.title），
     * 而不是資料表與欄位名稱！Hibernate 會自動把它轉成對應的 SQL。
     *
     * CONCAT('%', :keyword, '%') 實現「模糊查詢」：
     * 只要書名或作者包含關鍵字就會被查出來。
     */
    @Query("""
            SELECT b FROM Book b
            WHERE b.title  LIKE CONCAT('%', :keyword, '%')
               OR b.author LIKE CONCAT('%', :keyword, '%')
            """)
    List<Book> searchByKeyword(@Param("keyword") String keyword);
}
```

程式碼 3.7: repository/UserRepository.java

```
package com.example.bookstore.repository;

import com.example.bookstore.entity.User;
import org.springframework.data.jpa.repository.JpaRepository;

```

```java
import java.util.Optional;

/**
 * 使用者 Repository（資料存取層）。
 * 繼承 JpaRepository 後即可免費獲得 save / findById / findAll 等方法，
 * 下方再依需求宣告「衍生查詢」。
 */
public interface UserRepository extends JpaRepository<User, Long> {

    /**
     * 依帳號查詢使用者。
     * 產生 SQL：SELECT * FROM users WHERE username = ?
     *
     * 回傳 Optional<User>：
     * 明確表達「可能查不到」，呼叫端必須處理查無資料的情況
     * （例如 orElseThrow），可避免 NullPointerException。
     */
    Optional<User> findByUsername(String username);

    /** 註冊時檢查帳號是否已被使用 */
    boolean existsByUsername(String username);

    /** 註冊時檢查 Email 是否已被使用 */
    boolean existsByEmail(String email);
}
```

3.6.1 衍生查詢：用方法名稱寫查詢

Spring Data JPA 會解析方法名稱、自動產生查詢，常用關鍵字：

關鍵字	方法名稱範例	產生的查詢條件
findBy	findByUsername(name)	WHERE username = ?
existsBy	existsByIsbn(isbn)	SELECT COUNT(*)>0 ...
Containing	findByTitleContaining(kw)	WHERE title LIKE %kw%
IgnoreCase	findByAuthorIgnoreCase(a)	不分大小寫比對
And / Or	findByTitleAndAuthor(t,a)	WHERE title=? AND author=?
LessThan	findByPriceLessThan(p)	WHERE price < ?
OrderBy	findByOrderByPriceDesc()	ORDER BY price DESC

方法名稱寫不出來的複雜查詢，就用 `@Query` 自己寫 JPQL（如 `BookRepository.searchByKeyword()`）。JPQL 長得像 SQL，但操作對象是 **Entity 與屬性**（`Book`、`b.title`），不是資料表與欄位，由 Hibernate 負責翻譯成真正的 SQL。

3.7 眼見為憑：觀察 Hibernate 的 SQL

設定檔開啟了 `show-sql` 與參數綁定日誌，呼叫 API 時主控台會即時印出 Hibernate 產生的 SQL，例如查詢單本書籍：

```
Hibernate:
    select
        b1_0.id, b1_0.author, b1_0.created_at, b1_0.description,
        b1_0.isbn, b1_0.price, b1_0.stock, b1_0.title, b1_0.updated_at
    from
        books b1_0
    where
        b1_0.id=?
-- 綁定參數日誌：binding parameter [1] as [BIGINT] - [1]
```

學習建議

接下來測試每一個 API 時，都回頭看一眼主控台的 SQL——「我呼叫的方法」與「實際執行的 SQL」對照久了，ORM 就再也不是黑魔法。

3.8 交易管理：@Transactional

交易（Transaction）＝全有或全無

一個交易內的多個資料庫操作，要嘛**全部成功**（commit），要嘛**全部取消**（rollback），不會停在「做一半」的狀態。經典例子：轉帳 = 「A 扣款 + B 入帳」必須綁在同一個交易，否則扣了款卻入帳失敗，錢就憑空消失了。

在 Spring 中只要在 Service 方法標上 `@Transactional`：

- 方法開始時開啟交易，正常結束時 commit。
- 方法拋出 `RuntimeException` 時自動 rollback。
- 查詢類方法標 `@Transactional(readOnly = true)`，告訴 Hibernate 與資料庫「這是唯讀操作」，可以做效能最佳化。

實際用法會在下一章的 `BookService` 中看到。

第 4 章　REST API 設計與實作

本章是全書的核心之一：完整實作書籍的 CRUD API，正確運用 `GET`、`POST`、`PUT`、`PATCH`、`DELETE` 五種 HTTP 方法，並建立統一的錯誤處理機制。

4.1　REST 核心概念

REST（REpresentational State Transfer）是一種 API 設計風格，核心思想：

- **一切都是資源（Resource）**：書籍、使用者……每種資源用一個 URI 表示。
- **用 HTTP 方法表達動作**：URI 只寫「名詞」，動作交給 HTTP 方法。不要設計 `/getBooks` 或 `/deleteBook` 這種把動詞放進 URI 的風格。
- **無狀態（Stateless）**：伺服器不記住客戶端的狀態，每個請求都攜帶完整資訊（第 5 章的 JWT 正是為此而生）。

URI 設計慣例：

慣例	範例
用「複數名詞」表示資源集合	`/api/books`（而非 `/api/book`）
用路徑區隔單一資源	`/api/books/1`
查詢、分頁、排序用 Query String	`/api/books?page=0&size=10`
不要把動詞放進 URI	`DELETE /api/books/1`（而非 `/api/deleteBook?id=1`）

4.2　五種 HTTP 方法完整解析

方法	語意	本專案範例	成功狀態碼	冪等	安全
GET	讀取資源	查詢書籍列表 / 單筆	200	✓	✓
POST	建立資源	新增一本書	201	×	×
PUT	完整替換資源	更新書籍全部欄位	200	✓	×
PATCH	部分修改資源	只調整書籍價格	200	×[1]	×
DELETE	刪除資源	刪除一本書	204	✓	×

[1]PATCH 規範上「不保證」冪等；若實作成「設定欄位值」（如本專案），實際行為是冪等的。

冪等（Idempotent）與安全（Safe）

安全 = 不改變伺服器上的資料（只有讀取類的 GET 是安全的）。

冪等 = 同一請求執行 1 次和執行 N 次，資源的最終狀態相同：

- `PUT /books/1`（整包覆寫）做十次，結果都一樣 → 冪等。
- `POST /books`（每次建立新資料）做十次會有十筆 → 非冪等。
- `DELETE /books/1` 做十次，資源狀態都是「不存在」→ 冪等（第二次起雖回 404，資源狀態並沒有改變）。

冪等性很重要：網路逾時重送請求時，冪等的操作可以放心重試。

4.2.1 PUT 與 PATCH 的差異（高頻考點）

PUT——完整替換	PATCH——部分更新
必須提供資源的「所有」欄位	只提供「想修改」的欄位
沒帶到的欄位視同要被清空 / 重設	沒帶到的欄位維持原值
本專案用 `BookRequest`（欄位全部必填）	本專案用 `BookPatchRequest`（欄位全部可省略）

例如只想把價格改成 499：

```
// PATCH /api/books/1 ——— 只帶要改的欄位，其他欄位不動
{ "price": 499.00 }

// 若用 PUT，就必須把 title、author、isbn、stock... 全部帶上，
// 漏帶任何一個必填欄位都會收到 400 驗證錯誤
```

4.3 HTTP 狀態碼

狀態碼	名稱	本專案使用時機
200	OK	查詢 / 更新成功
201	Created	新增成功（附 `Location` Header 指向新資源）
204	No Content	刪除成功（沒有回應本文）
400	Bad Request	輸入驗證失敗、JSON 格式錯誤
401	Unauthorized	未登入、token 無效或過期（認證失敗）
403	Forbidden	已登入但角色權限不足（授權失敗）
404	Not Found	查無資源（例如書籍 id 不存在）
409	Conflict	資料衝突（ISBN / 帳號 / Email 重複）
500	Internal Server Error	伺服器未預期錯誤

4.4 本專案 API 總覽

方法	路徑	說明	權限
POST	`/api/auth/register`	註冊新使用者	公開
POST	`/api/auth/login`	登入並取得 JWT	公開
GET	`/api/books`	分頁查詢所有書籍	公開
GET	`/api/books/{id}`	查詢單一書籍	公開
GET	`/api/books/search`	關鍵字搜尋	公開
POST	`/api/books`	新增書籍	USER / ADMIN
PUT	`/api/books/{id}`	完整更新書籍	USER / ADMIN
PATCH	`/api/books/{id}`	部分更新書籍	USER / ADMIN
DELETE	`/api/books/{id}`	刪除書籍	僅 ADMIN

4.5 DTO 與輸入驗證

為什麼不直接用 Entity 接收請求？

- **安全**：若直接用 Entity 接收 JSON，客戶端可以偷塞 `id`、`createdAt` 等不該由外部控制的欄位（Mass Assignment 漏洞）。
- **解耦**：API 欄位與資料表結構可以各自演進，互不牽制。
- **驗證**：驗證規則集中寫在 DTO，Entity 保持乾淨。

程式碼 4.1: dto/BookRequest.java——新增與完整更新的請求 DTO

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

import jakarta.validation.constraints.DecimalMin;
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.NotNull;
import jakarta.validation.constraints.Pattern;
import jakarta.validation.constraints.Size;

import java.math.BigDecimal;

/**
 * 「新增（POST）/ 完整更新（PUT）書籍」的請求 DTO（Data Transfer Object）。
 *
 * 【為什麼不直接用 Book Entity 接收請求？】
 *  1. 安全：避免客戶端偷塞不該修改的欄位（如 id、createdAt），
 *     這種攻擊稱為 Mass Assignment（大量賦值）漏洞。
 *  2. 解耦：API 的欄位格式與資料庫結構可以各自獨立演進。
 *  3. 驗證：輸入驗證規則集中寫在 DTO 上，Entity 保持乾淨。
 *
 * 【Java Record】
```

```
 * record 是 Java 16+ 的「不可變資料類別」，
 * 自動產生建構子、存取方法（title()）、equals / hashCode / toString，
 * 非常適合拿來當 DTO，省去大量樣板程式碼。
 *
 * 【Bean Validation 驗證註解】
 * 需搭配 Controller 參數上的 @Valid 才會生效。
 * 驗證失敗時會拋出 MethodArgumentNotValidException，
 * 由 GlobalExceptionHandler 統一轉成 400 Bad Request 回應。
 */
public record BookRequest(

        @NotBlank(message = "書名不可為空白")                    // 不可為 null、不可為空字串或全空白
        @Size(max = 200, message = "書名長度不可超過 200 字")
        String title,

        @NotBlank(message = "作者不可為空白")
        @Size(max = 100, message = "作者長度不可超過 100 字")
        String author,

        @NotBlank(message = "ISBN 不可為空白")
        @Pattern(regexp = "^[0-9\\-]{10,17}$",
                 message = "ISBN 格式不正確（10~17 位的數字與連字號）")
        String isbn,

        @NotNull(message = "價格不可為空")                       // BigDecimal 用 @NotNull（@NotBlank 只能用在字串）
        @DecimalMin(value = "0.0", message = "價格不可為負數")
        BigDecimal price,

        @NotNull(message = "庫存不可為空")
        @Min(value = 0, message = "庫存不可為負數")
        Integer stock,

        @Size(max = 2000, message = "簡介長度不可超過 2000 字")  // 沒加 @NotBlank → 此欄位可以不填
        String description
) {
}
```

程式碼 4.2: dto/BookPatchRequest.java——部分更新的請求 DTO（全欄位可省略）

```
package com.example.bookstore.dto;

import jakarta.validation.constraints.DecimalMin;
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.Pattern;
import jakarta.validation.constraints.Size;
```

```

import java.math.BigDecimal;

/**
 * 「部分更新（PATCH）書籍」的請求 DTO。
 *
 * 【PATCH 與 PUT 的關鍵差異】
 *   PUT   = 完整替換：所有欄位都必填，沒帶到的欄位視同要清空 → 用 BookRequest
 *   PATCH = 部分更新：只更新「有帶值」的欄位，其餘維持原狀   → 用本類別
 *
 * 因此本 DTO 的所有欄位都「允許為 null」（沒有 @NotBlank / @NotNull）。
 * Service 層會逐一判斷：欄位不是 null 才更新對應的資料。
 *
 * 【小知識】@Size、@Pattern、@DecimalMin 這些註解遇到 null 值會「直接跳過」
 * （Bean Validation 規範：null 交給 @NotNull 把關），
 * 所以它們只會驗證「有傳入」的欄位，正好符合 PATCH 的語意。
 */
public record BookPatchRequest(

        @Size(max = 200, message = "書名長度不可超過 200 字")
        String title,

        @Size(max = 100, message = "作者長度不可超過 100 字")
        String author,

        @Pattern(regexp = "^[0-9\\-]{10,17}$",
                 message = "ISBN 格式不正確（10~17 位的數字與連字號）")
        String isbn,

        @DecimalMin(value = "0.0", message = "價格不可為負數")
        BigDecimal price,

        @Min(value = 0, message = "庫存不可為負數")
        Integer stock,

        @Size(max = 2000, message = "簡介長度不可超過 2000 字")
        String description
) {
}
```

程式碼 4.3: dto/BookResponse.java——回應 DTO

```
package com.example.bookstore.dto;

import com.example.bookstore.entity.Book;

import java.math.BigDecimal;
import java.time.LocalDateTime;
```

```
7
8  /**
9   * 「書籍」的回應 DTO——回傳給客戶端的 JSON 格式。
10  *
11  * 使用回應 DTO 的好處：
12  *   1. 精準控制要對外公開的欄位（例如未來 Entity 加了內部欄位也不會外洩）。
13  *   2. 與資料庫結構解耦，調整資料表不影響 API 契約。
14  *
15  * Jackson 會把這個 record 自動序列化成 JSON，例如：
16  * {
17  *   "id": 1,
18  *   "title": "Clean Code",
19  *   "author": "Robert C. Martin",
20  *   "isbn": "9780132350884",
21  *   "price": 1200.00,
22  *   "stock": 10,
23  *   "description": "...",
24  *   "createdAt": "2026-08-14T10:30:00",
25  *   "updatedAt": "2026-08-14T10:30:00"
26  * }
27  */
28 public record BookResponse(
29         Long id,
30         String title,
31         String author,
32         String isbn,
33         BigDecimal price,
34         Integer stock,
35         String description,
36         LocalDateTime createdAt,
37         LocalDateTime updatedAt
38 ) {
39
40     /**
41      * 【靜態工廠方法】把 Entity 轉換成回應 DTO。
42      * 集中轉換邏輯，讓 Service 層程式碼保持簡潔：
43      * BookResponse.from(book) 或 method reference：BookResponse::from
44      */
45     public static BookResponse from(Book book) {
46         return new BookResponse(
47                 book.getId(),
48                 book.getTitle(),
49                 book.getAuthor(),
50                 book.getIsbn(),
51                 book.getPrice(),
52                 book.getStock(),
53                 book.getDescription(),
54                 book.getCreatedAt(),
```

```
                book.getUpdatedAt()
        );
    }
}
```

程式碼 4.4: dto/PageResponse.java——分頁回應 DTO

```
package com.example.bookstore.dto;

import org.springframework.data.domain.Page;

import java.util.List;

/**
 * 分頁查詢的統一回應格式（泛型 DTO）。
 *
 * Spring Data 的 Page 物件包含大量內部欄位，直接序列化成 JSON
 * 會輸出很多用不到的資訊、且格式不受我們控制，
 * 因此自訂一個乾淨、穩定的分頁回應格式。
 *
 * 回應範例：
 * {
 *   "content": [ {...}, {...} ],   // 本頁的資料列表
 *   "page": 0,                     // 目前頁碼（從 0 開始）
 *   "size": 10,                    // 每頁筆數
 *   "totalElements": 25,           // 總資料筆數
 *   "totalPages": 3,               // 總頁數
 *   "last": false                  // 是否為最後一頁
 * }
 *
 * @param <T> 資料內容的型別（本專案為 BookResponse）
 */
public record PageResponse<T>(
        List<T> content,
        int page,
        int size,
        long totalElements,
        int totalPages,
        boolean last
) {

    /** 把 Spring Data 的 Page 物件轉成我們自訂的格式 */
    public static <T> PageResponse<T> from(Page<T> page) {
        return new PageResponse<>(
                page.getContent(),
                page.getNumber(),
                page.getSize(),
                page.getTotalElements(),
```

```
                page.getTotalPages(),
                page.isLast()
        );
    }
}
```

常用驗證註解：

註解	規則
`@NotNull`	不可為 null（任何型別）
`@NotBlank`	字串不可為 null / 空字串 / 全空白
`@Size`	字串長度或集合大小範圍
`@Min` / `@Max`	數值上下限
`@DecimalMin`	小數下限（適用 BigDecimal）
`@Email`	Email 格式
`@Pattern`	自訂正規表達式

小提示

驗證註解要生效，Controller 的參數前必須加 `@Valid`。另外注意：`@Size`、`@Pattern` 等註解遇到 null 會「跳過不驗」，null 的把關是 `@NotNull` / `@NotBlank` 的責任——這個特性正好被 `BookPatchRequest` 用來實現「只驗證有傳的欄位」。

4.6 Service：商業邏輯層

程式碼 4.5: service/BookService.java——書籍商業邏輯

```
package com.example.bookstore.service;

import com.example.bookstore.dto.BookPatchRequest;
import com.example.bookstore.dto.BookRequest;
import com.example.bookstore.dto.BookResponse;
import com.example.bookstore.dto.PageResponse;
import com.example.bookstore.entity.Book;
import com.example.bookstore.exception.DuplicateResourceException;
import com.example.bookstore.exception.ResourceNotFoundException;
import com.example.bookstore.repository.BookRepository;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

```

```java
/**
 * 書籍 Service（商業邏輯層）。
 *
 * 【三層式架構中的角色】
 *   Controller（接 HTTP）→ Service（商業邏輯）→ Repository（存取資料庫）
 *
 * Service 層的職責：
 *   1. 商業規則（例如：ISBN 不可重複）
 *   2. 交易管理（@Transactional）
 *   3. Entity 與 DTO 之間的轉換
 *
 * 【@Transactional 交易管理】
 * 標註後，方法內的所有資料庫操作會包在「同一個交易」裡：
 * 全部成功才 commit；中途拋出例外則全部 rollback，確保資料一致性。
 * readOnly = true 用於純查詢方法，可讓資料庫與 Hibernate 做效能最佳化。
 */
@Service
public class BookService {

    private final BookRepository bookRepository;

    /**
     * 【建構子注入 Constructor Injection】
     * Spring 建立 BookService 時，會自動把 BookRepository 的實例傳進來
     * （這就是「依賴注入 DI」）。相較於 @Autowired 欄位注入，
     * 建構子注入的依賴關係更明確、欄位可宣告為 final、也更容易撰寫單元測試。
     */
    public BookService(BookRepository bookRepository) {
        this.bookRepository = bookRepository;
    }

    /** 分頁查詢所有書籍（給 GET /api/books 使用） */
    @Transactional(readOnly = true)
    public PageResponse<BookResponse> findAll(Pageable pageable) {
        // Page.map()：把分頁裡的每個 Book Entity 轉成 BookResponse DTO
        Page<BookResponse> page = bookRepository.findAll(pageable)
                .map(BookResponse::from);
        return PageResponse.from(page);
    }

    /** 依 id 查詢單一書籍（給 GET /api/books/{id} 使用） */
    @Transactional(readOnly = true)
    public BookResponse findById(Long id) {
        Book book = getBookOrThrow(id);
        return BookResponse.from(book);
    }

    /** 關鍵字搜尋（給 GET /api/books/search?keyword=xxx 使用） */
```

```
66      @Transactional(readOnly = true)
67      public List<BookResponse> search(String keyword) {
68          return bookRepository.searchByKeyword(keyword).stream()
69                  .map(BookResponse::from)
70                  .toList();
71      }
72
73      /** 新增書籍（給 POST /api/books 使用） */
74      @Transactional
75      public BookResponse create(BookRequest request) {
76          // 商業規則：ISBN 不可重複
77          if (bookRepository.existsByIsbn(request.isbn())) {
78              throw new DuplicateResourceException("ISBN 已存在：" + request.isbn());
79          }
80
81          Book book = new Book(
82                  request.title(), request.author(), request.isbn(),
83                  request.price(), request.stock(), request.description());
84
85          // save() 回傳「儲存後」的 Entity（此時 id 已由資料庫產生）
86          Book saved = bookRepository.save(book);
87          return BookResponse.from(saved);
88      }
89
90      /**
91       * 完整更新書籍（給 PUT /api/books/{id} 使用）。
92       * PUT 的語意是「整個資源替換」：所有欄位一律以請求內容覆寫。
93       */
94      @Transactional
95      public BookResponse update(Long id, BookRequest request) {
96          Book book = getBookOrThrow(id);
97
98          // 檢查新 ISBN 是否與「其他書籍」衝突（排除自己）
99          if (bookRepository.existsByIsbnAndIdNot(request.isbn(), id)) {
100             throw new DuplicateResourceException("ISBN 已被其他書籍使用：" + request.isbn());
101         }
102
103         // 覆寫全部欄位
104         book.setTitle(request.title());
105         book.setAuthor(request.author());
106         book.setIsbn(request.isbn());
107         book.setPrice(request.price());
108         book.setStock(request.stock());
109         book.setDescription(request.description());
110
111         //  【JPA 小知識：Dirty Checking 髒值檢查】
112         // book 是交易中的「受管理（Managed）」Entity，
113         // 交易提交時 Hibernate 會自動偵測欄位變更並送出 UPDATE。
```

```
        //
        // 這裡用 saveAndFlush()（而非 save()）立刻執行 UPDATE：
        // @PreUpdate 回呼是在「flush」階段才觸發的，
        // 若等交易提交才 flush，回應中的 updatedAt 會是舊值。
        Book saved = bookRepository.saveAndFlush(book);
        return BookResponse.from(saved);
    }

    /**
     * 部分更新書籍（給 PATCH /api/books/{id} 使用）。
     * PATCH 的語意是「只修改有提供的欄位」：null 代表「不要動這個欄位」。
     */
    @Transactional
    public BookResponse patch(Long id, BookPatchRequest request) {
        Book book = getBookOrThrow(id);

        // 逐一判斷：有帶值（非 null）的欄位才更新
        if (request.title() != null) {
            book.setTitle(request.title());
        }
        if (request.author() != null) {
            book.setAuthor(request.author());
        }
        if (request.isbn() != null) {
            if (bookRepository.existsByIsbnAndIdNot(request.isbn(), id)) {
                throw new DuplicateResourceException("ISBN 已被其他書籍使用：" + request.isbn());
            }
            book.setIsbn(request.isbn());
        }
        if (request.price() != null) {
            book.setPrice(request.price());
        }
        if (request.stock() != null) {
            book.setStock(request.stock());
        }
        if (request.description() != null) {
            book.setDescription(request.description());
        }

        // 同 update()：立刻 flush，讓 @PreUpdate 更新 updatedAt 後再組回應
        Book saved = bookRepository.saveAndFlush(book);
        return BookResponse.from(saved);
    }

    /** 刪除書籍（給 DELETE /api/books/{id} 使用） */
    @Transactional
    public void delete(Long id) {
        // 先確認資料存在，不存在就回 404（而不是默默地什麼都沒刪）
```

```
        Book book = getBookOrThrow(id);
        bookRepository.delete(book);
    }

    /** 共用小工具：依 id 取書，查無資料就拋出 404 例外 */
    private Book getBookOrThrow(Long id) {
        return bookRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("書籍", "id", id));
    }
}
```

幾個設計重點：

- **建構子注入**：依賴宣告為 `final`，由建構子傳入。比 `@Autowired` 欄位注入更明確、更好測試，是官方推薦作法。
- **查無資料就拋例外**：`getBookOrThrow()` 統一處理查無資料的情況，直接拋出 `ResourceNotFoundExceptio` 由全域例外處理器轉成 404。
- **PATCH 的實作**：逐欄位判斷「不是 null 才更新」，完美對應 PATCH 的部分更新語意。
- **商業規則**：ISBN 唯一性檢查放在 Service（而不是 Controller），因為這是「商業邏輯」，無論請求從哪裡來都必須遵守。

4.7 Controller：控制層

程式碼 4.6: controller/BookController.java——五種 HTTP 方法的完整示範

```
package com.example.bookstore.controller;

import com.example.bookstore.dto.BookPatchRequest;
import com.example.bookstore.dto.BookRequest;
import com.example.bookstore.dto.BookResponse;
import com.example.bookstore.dto.PageResponse;
import com.example.bookstore.service.BookService;
import jakarta.validation.Valid;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;
import org.springframework.http.ResponseEntity;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PatchMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
```

```java
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;

import java.net.URI;
import java.util.List;

/**
 * 書籍 Controller：完整示範 REST API 的五種 HTTP 方法。
 *
 * ┌────────┬──────────────────────┬──────────────┬──────────┐
 * │ 方法   │ 路徑                 │ 用途         │ 權限     │
 * ├────────┼──────────────────────┼──────────────┼──────────┤
 * │ GET    │ /api/books           │ 分頁查詢全部 │ 公開     │
 * │ GET    │ /api/books/{id}      │ 查詢單筆     │ 公開     │
 * │ GET    │ /api/books/search    │ 關鍵字搜尋   │ 公開     │
 * │ POST   │ /api/books           │ 新增         │ USER 以上 │
 * │ PUT    │ /api/books/{id}      │ 完整更新     │ USER 以上 │
 * │ PATCH  │ /api/books/{id}      │ 部分更新     │ USER 以上 │
 * │ DELETE │ /api/books/{id}      │ 刪除         │ ADMIN    │
 * └────────┴──────────────────────┴──────────────┴──────────┘
 *
 * 【Controller 層的職責】只做三件事：
 *   1. 接收並解析 HTTP 請求（路徑、參數、本文）
 *   2. 呼叫 Service 執行商業邏輯
 *   3. 包裝 HTTP 回應（狀態碼、Header、JSON 本文）
 * 商業邏輯一律放在 Service，Controller 保持輕薄。
 */
@RestController
@RequestMapping("/api/books")
public class BookController {

    private final BookService bookService;

    public BookController(BookService bookService) {
        this.bookService = bookService;
    }

    /**
     * 【GET】分頁查詢所有書籍。
     *
     *   GET /api/books?page=0&size=10&sortBy=price&direction=desc
     *
     * @RequestParam：從「查詢字串（Query String）」取值，
     * defaultValue 讓所有參數都可省略：GET /api/books 也能直接用。
     *
     * GET 的特性：安全（不改變資料）且冪等（重複呼叫結果相同）。
     */
```

```
70      @GetMapping
71      public ResponseEntity<PageResponse<BookResponse>> getAllBooks(
72              @RequestParam(defaultValue = "0") int page,          // 頁碼（從 0 開始）
73              @RequestParam(defaultValue = "10") int size,         // 每頁筆數
74              @RequestParam(defaultValue = "id") String sortBy,    // 排序欄位
75              @RequestParam(defaultValue = "asc") String direction // asc 升冪 / desc 降冪
76      ) {
77          // 組合排序與分頁條件，Spring Data 會自動轉成 ORDER BY ... LIMIT ...
78          Sort sort = "desc".equalsIgnoreCase(direction)
79                  ? Sort.by(sortBy).descending()
80                  : Sort.by(sortBy).ascending();
81          Pageable pageable = PageRequest.of(page, size, sort);
82
83          return ResponseEntity.ok(bookService.findAll(pageable));   // 200 OK
84      }
85
86      /**
87       * 【GET】依 id 查詢單一書籍。
88       *
89       *   GET /api/books/1
90       *
91       * @PathVariable：從「路徑」中取值（/api/books/1 → id = 1）。
92       * 查無資料時 Service 拋出 ResourceNotFoundException → 404。
93       */
94      @GetMapping("/{id}")
95      public ResponseEntity<BookResponse> getBookById(@PathVariable Long id) {
96          return ResponseEntity.ok(bookService.findById(id));        // 200 OK
97      }
98
99      /**
100      * 【GET】關鍵字搜尋（書名或作者含關鍵字）。
101      *
102      *   GET /api/books/search?keyword=Spring
103      */
104     @GetMapping("/search")
105     public ResponseEntity<List<BookResponse>> searchBooks(@RequestParam String keyword) {
106         return ResponseEntity.ok(bookService.search(keyword));     // 200 OK
107     }
108
109     /**
110      * 【POST】新增書籍。
111      *
112      *   POST /api/books
113      *   Header: Authorization: Bearer <token>
114      *   Body:   { "title": "...", "author": "...", "isbn": "...", ... }
115      *
116      * POST 的特性：非冪等——重複呼叫會（嘗試）建立多筆資料。
117      *
```

```
     * REST 慣例：建立成功回 201 Created，
     * 並在 Location Header 告知「新資源的網址」。
     */
    @PostMapping
    public ResponseEntity<BookResponse> createBook(@Valid @RequestBody BookRequest request) {
        BookResponse created = bookService.create(request);

        // 組出新資源的 URI：目前請求路徑 + /{id} → /api/books/42
        URI location = ServletUriComponentsBuilder
                .fromCurrentRequest()
                .path("/{id}")
                .buildAndExpand(created.id())
                .toUri();

        return ResponseEntity.created(location).body(created);     // 201 Created
    }

    /**
     * 【PUT】完整更新書籍——「整個資源替換」。
     *
     *   PUT /api/books/1
     *   Body: 必須提供「所有」欄位（少一個就會 400 驗證失敗）
     *
     * PUT 的特性：冪等——用同樣的內容呼叫一次和呼叫十次，結果相同。
     */
    @PutMapping("/{id}")
    public ResponseEntity<BookResponse> updateBook(
            @PathVariable Long id,
            @Valid @RequestBody BookRequest request) {
        return ResponseEntity.ok(bookService.update(id, request)); // 200 OK
    }

    /**
     * 【PATCH】部分更新書籍——「只改有提供的欄位」。
     *
     *   PATCH /api/books/1
     *   Body: { "price": 450 }   ← 只想調價，其他欄位完全不用帶
     *
     * 與 PUT 的差別：PUT 要求完整資料、PATCH 允許只給部分欄位。
     */
    @PatchMapping("/{id}")
    public ResponseEntity<BookResponse> patchBook(
            @PathVariable Long id,
            @Valid @RequestBody BookPatchRequest request) {
        return ResponseEntity.ok(bookService.patch(id, request));  // 200 OK
    }

    /**
```

```
     * 【DELETE】刪除書籍——僅限管理員。
     *
     *   DELETE /api/books/1
     *   Header: Authorization: Bearer <admin 的 token>
     *
     * @PreAuthorize：方法層級授權（Method Security）。
     * SecurityConfig 的 URL 規則已限制 DELETE 需要 ADMIN，
     * 這裡再標一次是「雙重防線」，也順便示範第二種授權寫法。
     *
     * DELETE 的特性：冪等——刪除同一筆資料多次，資源狀態都是「不存在」。
     * REST 慣例：刪除成功回 204 No Content（沒有回應本文）。
     */
    @DeleteMapping("/{id}")
    @PreAuthorize("hasRole('ADMIN')")
    public ResponseEntity<Void> deleteBook(@PathVariable Long id) {
        bookService.delete(id);
        return ResponseEntity.noContent().build();                // 204 No Content
    }
}
```

三種「取得請求資料」的註解：

註解	資料來源	範例
`@PathVariable`	URI 路徑的一部分	`/api/books/{id}` 中的 `id`
`@RequestParam`	Query String	`?page=0&size=10`
`@RequestBody`	請求本文（JSON）	POST / PUT / PATCH 的書籍資料

ResponseEntity 是什麼？

`ResponseEntity<T>` 讓你完整控制 HTTP 回應的三要素：**狀態碼**（`ok()` = 200、`created()` = 201、`noContent()` = 204）、**Header**（如 `Location`）、**本文**（自動序列化成 JSON 的物件）。

4.8 統一例外處理

沒有統一處理時，錯誤會以 Spring 預設格式（甚至 HTML）回傳，格式混亂。我們的做法：自訂例外 + 統一錯誤格式 + 全域攔截。

程式碼 4.7: exception/ResourceNotFoundException.java

```
package com.example.bookstore.exception;

/**
 * 「查無資料」的自訂例外，最終會被轉成 HTTP 404 Not Found。
 *
```

```
 * 【設計理念】
 * Service 層發現資料不存在時，直接拋出這個例外即可，
 * 不需要層層回傳 null 或 boolean。
 * GlobalExceptionHandler 會統一攔截並轉成 404 JSON 回應，
 * 讓 Controller / Service 的主要邏輯保持乾淨。
 *
 * 繼承 RuntimeException（非受檢例外）：
 * 呼叫端不必強制 try-catch，交給全域例外處理器即可。
 */
public class ResourceNotFoundException extends RuntimeException {

    /**
     * @param resourceName 資源名稱（例如「書籍」）
     * @param fieldName    查詢欄位（例如「id」）
     * @param fieldValue   查詢值（例如 99）
     * 組出訊息：「找不到書籍：id = '99'」
     */
    public ResourceNotFoundException(String resourceName, String fieldName, Object fieldValue) {
        super("找不到%s：%s = '%s'".formatted(resourceName, fieldName, fieldValue));
    }
}
```

程式碼 4.8: exception/DuplicateResourceException.java

```
package com.example.bookstore.exception;

/**
 * 「資料重複」的自訂例外，最終會被轉成 HTTP 409 Conflict。
 *
 * 使用時機：
 *   - 註冊時帳號或 Email 已被使用
 *   - 新增 / 修改書籍時 ISBN 已存在
 */
public class DuplicateResourceException extends RuntimeException {

    public DuplicateResourceException(String message) {
        super(message);
    }
}
```

程式碼 4.9: dto/ApiErrorResponse.java——統一錯誤回應格式

```
package com.example.bookstore.dto;

import com.fasterxml.jackson.annotation.JsonInclude;
import org.springframework.http.HttpStatus;

import java.time.LocalDateTime;
```

```java
import java.util.Map;

/**
 * 統一的「錯誤回應」格式。
 *
 * 【為什麼需要統一格式？】
 * 讓所有錯誤（404、400、401、403、409、500...）都長得一樣，
 * 前端只要寫一次錯誤處理邏輯，開發與除錯都更輕鬆。
 *
 * 回應範例（驗證失敗 400）：
 * {
 *   "timestamp": "2026-08-14T10:30:00",
 *   "status": 400,
 *   "error": "Bad Request",
 *   "message": "輸入資料驗證失敗",
 *   "path": "/api/books",
 *   "fieldErrors": { "title": "書名不可為空白", "price": "價格不可為負數" }
 * }
 *
 * @JsonInclude(NON_NULL)：值為 null 的欄位不輸出到 JSON
 *  （例如一般錯誤沒有 fieldErrors，就不會出現這個欄位）。
 */
@JsonInclude(JsonInclude.Include.NON_NULL)
public record ApiErrorResponse(
        LocalDateTime timestamp,          // 錯誤發生時間
        int status,                       // HTTP 狀態碼數字（如 404）
        String error,                     // 狀態碼名稱（如 "Not Found"）
        String message,                   // 給人看的錯誤說明
        String path,                      // 發生錯誤的 API 路徑
        Map<String, String> fieldErrors   // 欄位驗證錯誤明細（僅 400 使用）
) {

    /** 建立一般錯誤回應（不含欄位錯誤明細） */
    public static ApiErrorResponse of(HttpStatus status, String message, String path) {
        return new ApiErrorResponse(
                LocalDateTime.now(), status.value(), status.getReasonPhrase(),
                message, path, null);
    }

    /** 建立含「欄位錯誤明細」的回應（驗證失敗 400 專用） */
    public static ApiErrorResponse of(HttpStatus status, String message, String path,
                                      Map<String, String> fieldErrors) {
        return new ApiErrorResponse(
                LocalDateTime.now(), status.value(), status.getReasonPhrase(),
                message, path, fieldErrors);
    }
}
```

程式碼 4.10: exception/GlobalExceptionHandler.java——全域例外處理器

```java
package com.example.bookstore.exception;

import com.example.bookstore.dto.ApiErrorResponse;
import jakarta.servlet.http.HttpServletRequest;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.http.converter.HttpMessageNotReadableException;
import org.springframework.security.access.AccessDeniedException;
import org.springframework.security.authentication.BadCredentialsException;
import org.springframework.validation.FieldError;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;

import java.util.HashMap;
import java.util.Map;

/**
 * 全域例外處理器（Global Exception Handler）。
 *
 * 【運作原理】
 * @RestControllerAdvice = @ControllerAdvice + @ResponseBody
 * 它像一張「安全網」，攔截所有 Controller / Service 拋出的例外，
 * 依例外類型轉換成對應的 HTTP 狀態碼與統一 JSON 格式。
 *
 * 【好處】
 *   1. Controller / Service 不必到處寫 try-catch，邏輯乾淨。
 *   2. 所有錯誤回應格式一致（ApiErrorResponse），前端好處理。
 *   3. 錯誤與狀態碼的對應關係集中在一個檔案，一目了然。
 */
@RestControllerAdvice
public class GlobalExceptionHandler {

    private static final Logger log = LoggerFactory.getLogger(GlobalExceptionHandler.class);

    /**
     * 404 Not Found：查無資料。
     * 例如 GET /api/books/999 但資料庫沒有 id=999 的書。
     */
    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ApiErrorResponse> handleNotFound(
            ResourceNotFoundException ex, HttpServletRequest request) {
        return build(HttpStatus.NOT_FOUND, ex.getMessage(), request);
    }

```

```
48      /**
49       * 409 Conflict：資料重複（違反唯一性）。
50       * 例如註冊了已存在的帳號、新增了重複的 ISBN。
51       */
52      @ExceptionHandler(DuplicateResourceException.class)
53      public ResponseEntity<ApiErrorResponse> handleDuplicate(
54              DuplicateResourceException ex, HttpServletRequest request) {
55          return build(HttpStatus.CONFLICT, ex.getMessage(), request);
56      }
57
58      /**
59       * 400 Bad Request：請求本文的「欄位驗證」失敗。
60       * 當 @Valid 驗證不通過時，Spring 會拋出這個例外，
61       * 我們把每個欄位的錯誤訊息整理成 map 回傳，方便前端逐欄顯示。
62       */
63      @ExceptionHandler(MethodArgumentNotValidException.class)
64      public ResponseEntity<ApiErrorResponse> handleValidation(
65              MethodArgumentNotValidException ex, HttpServletRequest request) {
66
67          // 收集所有欄位錯誤：{ 欄位名稱: 錯誤訊息 }
68          Map<String, String> fieldErrors = new HashMap<>();
69          for (FieldError fieldError : ex.getBindingResult().getFieldErrors()) {
70              fieldErrors.put(fieldError.getField(), fieldError.getDefaultMessage());
71          }
72
73          ApiErrorResponse body = ApiErrorResponse.of(
74                  HttpStatus.BAD_REQUEST, "輸入資料驗證失敗",
75                  request.getRequestURI(), fieldErrors);
76          return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(body);
77      }
78
79      /**
80       * 400 Bad Request：請求本文不是合法的 JSON。
81       * 例如漏了逗號、引號，或 Content-Type 對但內容是空的。
82       */
83      @ExceptionHandler(HttpMessageNotReadableException.class)
84      public ResponseEntity<ApiErrorResponse> handleUnreadable(
85              HttpMessageNotReadableException ex, HttpServletRequest request) {
86          return build(HttpStatus.BAD_REQUEST, "請求本文格式錯誤，請確認是合法的 JSON", request);
87      }
88
89      /**
90       * 401 Unauthorized：登入失敗（帳號或密碼錯誤）。
91       * AuthenticationManager 驗證失敗時拋出 BadCredentialsException。
92       * 資安小知識：訊息刻意不說「是帳號錯還是密碼錯」，避免被駭客列舉帳號。
93       */
94      @ExceptionHandler(BadCredentialsException.class)
95      public ResponseEntity<ApiErrorResponse> handleBadCredentials(
```

```
            BadCredentialsException ex, HttpServletRequest request) {
        return build(HttpStatus.UNAUTHORIZED, "帳號或密碼錯誤", request);
    }

    /**
     * 403 Forbidden：已登入，但「權限不足」。
     * 例如 ROLE_USER 嘗試呼叫只有 ROLE_ADMIN 能用的刪除 API，
     * @PreAuthorize 檢查失敗時會拋出 AccessDeniedException。
     */
    @ExceptionHandler(AccessDeniedException.class)
    public ResponseEntity<ApiErrorResponse> handleAccessDenied(
            AccessDeniedException ex, HttpServletRequest request) {
        return build(HttpStatus.FORBIDDEN, "權限不足，無法執行此操作", request);
    }

    /**
     * 500 Internal Server Error：兜底處理。
     * 任何沒被上面攔到的例外都會到這裡，避免把堆疊追蹤（stack trace）
     * 等內部細節外洩給客戶端；完整錯誤只記錄在伺服器日誌。
     */
    @ExceptionHandler(Exception.class)
    public ResponseEntity<ApiErrorResponse> handleGeneral(
            Exception ex, HttpServletRequest request) {
        log.error("未預期的錯誤", ex);   // 完整堆疊寫入伺服器日誌，方便除錯
        return build(HttpStatus.INTERNAL_SERVER_ERROR,
                "伺服器發生未預期的錯誤，請稍後再試", request);
    }

    /** 共用的回應建構方法，避免重複程式碼 */
    private ResponseEntity<ApiErrorResponse> build(
            HttpStatus status, String message, HttpServletRequest request) {
        ApiErrorResponse body = ApiErrorResponse.of(status, message, request.getRequestURI());
        return ResponseEntity.status(status).body(body);
    }
}
```

例外與狀態碼的對應關係：

例外	觸發情境	狀態碼
`ResourceNotFoundException`	查無資料	404
`DuplicateResourceException`	ISBN / 帳號重複	409
`MethodArgumentNotValidException`	`@Valid` 驗證失敗	400
`HttpMessageNotReadableException`	JSON 格式錯誤	400
`BadCredentialsException`	帳密錯誤	401
`AccessDeniedException`	角色權限不足	403
其他 `Exception`	未預期錯誤	500

驗證失敗（400）的實際回應範例——`fieldErrors` 逐欄說明錯在哪：

```
{
  "timestamp": "2026-08-14T10:30:00",
  "status": 400,
  "error": "Bad Request",
  "message": "輸入資料驗證失敗",
  "path": "/api/books",
  "fieldErrors": {
    "title": "書名不可為空白",
    "price": "價格不可為負數"
  }
}
```

4.9 用 curl 快速測試

啟動專案後，先測試不需要登入的公開查詢 API：

```
# 查詢所有書籍（分頁）
curl "http://localhost:8080/api/books?page=0&size=10"

# 查詢單一書籍
curl http://localhost:8080/api/books/1

# 關鍵字搜尋
curl "http://localhost:8080/api/books/search?keyword=Java"

# 查詢不存在的書籍 -> 觀察 404 統一錯誤格式
curl -i http://localhost:8080/api/books/99999
# -i 參數會一併顯示回應的狀態列與 Header
```

寫入類的 API（POST / PUT / PATCH / DELETE）需要 JWT，下一章完成認證機制後就能完整測試。

第 5 章　JWT 認證與授權

到目前為止，任何人都能呼叫寫入類的 API。本章加上安全機制：使用 Spring Security 搭配 JWT，實作註冊、登入、認證（Authentication）與角色授權（Authorization）。

5.1　認證與授權：先分清楚兩件事

認證 Authentication	授權 Authorization
回答「**你是誰？**」	回答「**你能做什麼？**」
驗證帳號密碼、驗證 token	檢查角色 / 權限是否足夠
失敗回 **401 Unauthorized**	失敗回 **403 Forbidden**
範例：沒帶 token 就想新增書籍	範例：一般使用者想刪除書籍

401 vs 403（面試高頻題）

401：伺服器「不知道你是誰」——沒帶 token、token 過期或無效。
403：伺服器「知道你是誰，但你不夠格」——已登入，但角色權限不足以執行這個操作。

本專案的權限設計：

身分	可執行的操作
訪客（未登入）	查詢書籍（所有 GET）
`ROLE_USER`	查詢 + 新增、修改書籍（POST / PUT / PATCH）
`ROLE_ADMIN`	全部操作，包含刪除書籍（DELETE）

5.2　為什麼用 Token 而不是 Session？

Session（有狀態）	JWT（無狀態）
登入狀態存在「伺服器」記憶體	登入狀態存在「token 本身」
伺服器要記住每個使用者	伺服器完全不必記，驗簽章即可
多台伺服器需共享 Session（黏著或 Redis）	任何一台伺服器都能獨立驗證
依賴 Cookie，跨網域、App 較麻煩	放在 Header，網頁 / App / 第三方通用

REST 講究無狀態，JWT 正好讓「每個請求自帶完整的身分證明」，這也是它成為 API 認證主流的原因。

5.3 JWT 結構解剖

JWT（JSON Web Token）是三段 Base64Url 編碼的字串，用「.」串接：



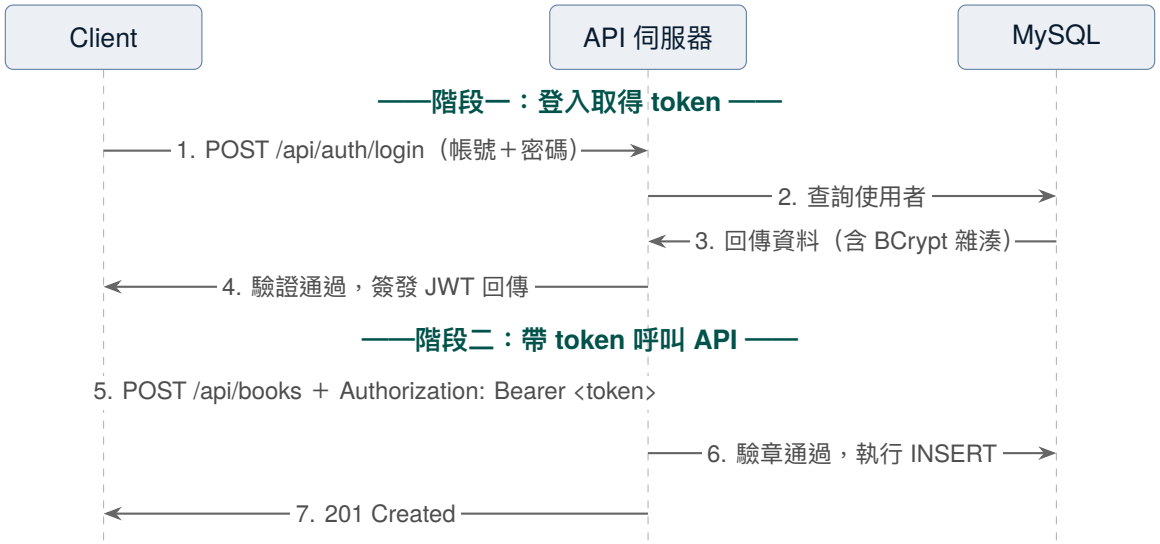
Payload 中的常見聲明（Claims）：

聲明	全名	意義
`sub`	Subject	主體——這個 token 屬於誰（本專案放帳號）
`iat`	Issued At	簽發時間
`exp`	Expiration Time	過期時間（過期的 token 驗證會失敗）
`role`	（自訂聲明）	本專案自訂：使用者角色

Payload 是「編碼」不是「加密」！

Base64 人人都能解開——把 token 貼到 `https://jwt.io` 就能看到完整內容。因此 Payload **絕對不能放密碼、卡號等敏感資料**。JWT 的安全性來自**簽章**：內容被改動一個字元，簽章驗證就會失敗，而沒有伺服器密鑰的人偽造不出合法簽章。

5.4 認證流程全景圖



階段二的第 6 步之前，請求會先通過我們的 `JwtAuthenticationFilter`（驗證簽章與有效期），再通過授權規則（檢查角色）——接下來就逐一實作這些元件。

5.5 實作 1：JwtService——產生與驗證 token

程式碼 5.1: security/JwtService.java

```
package com.example.bookstore.security;

import com.example.bookstore.entity.Role;
import io.jsonwebtoken.Claims;
import io.jsonwebtoken.JwtException;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.security.Keys;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Service;

import javax.crypto.SecretKey;
import java.nio.charset.StandardCharsets;
import java.util.Date;

/**
 * JWT 服務：負責「產生」與「驗證」JSON Web Token。
 *
 * 【JWT 的結構】三段 Base64Url 字串，用「.」串接：
 *
 *   Header.Payload.Signature
 *   eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ1c2VyIn0.dBjftJeZ4CVP...
 *
```

```
 *   1. Header    : 宣告演算法，例如 {"alg":"HS256"}
 *   2. Payload   : 聲明（Claims），例如使用者帳號、角色、過期時間
 *   3. Signature : 用「伺服器私密金鑰」對前兩段計算出的簽章
 *
 * 【重要觀念】
 *   - Payload 只是 Base64 編碼，「任何人都能解讀」→ 千萬別放密碼等敏感資料！
 *   - 簽章能保證 token「沒有被竄改」：只要內容被改一個字元，
 *     簽章驗證就會失敗。而沒有金鑰的人無法偽造出合法簽章。
 */
@Service
public class JwtService {

    /**
     * 簽章用的密鑰字串，從 application.yml 的 app.jwt.secret 讀入。
     * HS256 演算法要求至少 256 bits（32 bytes），太短會直接拋出例外。
     */
    @Value("${app.jwt.secret}")
    private String secret;

    /** token 有效毫秒數，從 application.yml 的 app.jwt.expiration-ms 讀入 */
    @Value("${app.jwt.expiration-ms}")
    private long expirationMs;

    /** 把密鑰字串轉成 HMAC-SHA 演算法使用的 SecretKey 物件 */
    private SecretKey getSigningKey() {
        return Keys.hmacShaKeyFor(secret.getBytes(StandardCharsets.UTF_8));
    }

    /**
     * 產生 JWT（登入成功後呼叫）。
     *
     * @param username 使用者帳號 → 放入標準聲明 sub（Subject，主體）
     * @param role     使用者角色 → 放入自訂聲明 "role"
     */
    public String generateToken(String username, Role role) {
        Date now = new Date();
        Date expiry = new Date(now.getTime() + expirationMs);

        return Jwts.builder()
                .subject(username)          // 標準聲明 sub：這個 token 屬於誰
                .claim("role", role.name()) // 自訂聲明：使用者角色
                .issuedAt(now)              // 標準聲明 iat：簽發時間
                .expiration(expiry)         // 標準聲明 exp：過期時間
                .signWith(getSigningKey())  // 用密鑰簽章（自動採用 HS256）
                .compact();                 // 組合成最終的字串
    }

    /**
```

```
     * 從 token 中解析出使用者帳號（sub 聲明）。
     * 若 token 無效（簽章錯誤、已過期、格式錯誤），parseClaims 會拋出例外。
     */
    public String extractUsername(String token) {
        return parseClaims(token).getSubject();
    }

    /**
     * 驗證 token 是否有效：
     *   1. 簽章正確（沒被竄改、確實是本伺服器簽發）
     *   2. 尚未過期
     *   3. token 中的帳號與資料庫載入的使用者一致
     */
    public boolean isTokenValid(String token, UserDetails userDetails) {
        try {
            Claims claims = parseClaims(token);
            boolean notExpired = claims.getExpiration().after(new Date());
            boolean sameUser = claims.getSubject().equals(userDetails.getUsername());
            return notExpired && sameUser;
        } catch (JwtException | IllegalArgumentException e) {
            // 簽章不符、過期、格式錯誤……一律視為無效
            return false;
        }
    }

    /**
     * 解析並驗證 token，回傳 Payload（Claims）。
     * verifyWith(key)：解析時同步驗證簽章，驗證失敗會拋出 JwtException。
     */
    private Claims parseClaims(String token) {
        return Jwts.parser()
                .verifyWith(getSigningKey())  // 設定驗章金鑰
                .build()
                .parseSignedClaims(token)     // 解析 + 驗章 + 檢查過期
                .getPayload();
    }

    /** 提供 token 有效毫秒數（登入回應中告知客戶端） */
    public long getExpirationMs() {
        return expirationMs;
    }
}
```

密鑰管理

密鑰放在 `application.yml` 的 `app.jwt.secret`，並支援用環境變數 `JWT_SECRET` 覆寫。HS256 要求密鑰至少 32 字元；正式環境務必改用隨機產生的長密鑰，且只透過環境變數或密鑰管理服務提供，絕不進版本控制。

5.6 實作 2：CustomUserDetailsService——載入使用者

Spring Security 不關心使用者存在哪裡，它只認 `UserDetailsService` 介面。我們實作它，從 MySQL 載入使用者：

程式碼 5.2: security/CustomUserDetailsService.java

```
package com.example.bookstore.security;

import com.example.bookstore.entity.User;
import com.example.bookstore.repository.UserRepository;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import java.util.List;

/**
 * 自訂的 UserDetailsService——告訴 Spring Security「怎麼載入使用者」。
 *
 * 【呼叫時機】
 *   1. 登入時：AuthenticationManager 用它取得使用者資料後比對密碼。
 *   2. 每次帶 JWT 的請求：JwtAuthenticationFilter 用它載入使用者權限。
 *
 * Spring Security 不關心你的使用者存在哪裡（MySQL？LDAP？記憶體？），
 * 它只認 UserDetailsService 這個介面——我們實作它，從 MySQL 載入。
 */
@Service
public class CustomUserDetailsService implements UserDetailsService {

    private final UserRepository userRepository;

    /** 建構子注入（Constructor Injection）：Spring 官方推薦的注入方式 */
    public CustomUserDetailsService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    /**
     * 依帳號載入使用者，並轉換成 Spring Security 認識的 UserDetails 格式。
```

```
     */
    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

        // 1. 從資料庫查詢使用者，查不到就拋出 Spring Security 的標準例外
        User user = userRepository.findByUsername(username)
                .orElseThrow(() -> new UsernameNotFoundException("找不到使用者：" + username));

        // 2. 把我們的 User Entity 轉成 Spring Security 內建的 UserDetails 實作
        //    - username：帳號
        //    - password：BCrypt 雜湊值（登入時由 Spring 比對）
        //    - authorities：權限清單（這裡放入角色，如 ROLE_ADMIN）
        return new org.springframework.security.core.userdetails.User(
                user.getUsername(),
                user.getPassword(),
                List.of(new SimpleGrantedAuthority(user.getRole().name()))
        );
    }
}
```

5.7 實作 3：JwtAuthenticationFilter——每個請求的守門員

程式碼 5.3: security/JwtAuthenticationFilter.java

```
package com.example.bookstore.security;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.web.authentication.WebAuthenticationDetailsSource;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

import java.io.IOException;

/**
 * JWT 認證過濾器——整個 JWT 機制的「守門員」。
 *
 * 【運作位置】
 * 它被插在 Spring Security 過濾器鏈（Filter Chain）中，
 * 「每一個」HTTP 請求進入 Controller 之前都會先經過這裡。
 *
```

```
 * 【工作流程】
 *   請求 → 取出 Header 中的 token → 驗證 → 合法就把使用者放入
 *   SecurityContext（本次請求的「已登入名單」）→ 放行給後續授權檢查
 *
 * 繼承 OncePerRequestFilter：保證同一請求只執行一次過濾邏輯。
 */
@Component
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    /** Authorization Header 的標準前綴，例如：Authorization: Bearer eyJhb... */
    private static final String BEARER_PREFIX = "Bearer ";

    private final JwtService jwtService;
    private final UserDetailsService userDetailsService;

    public JwtAuthenticationFilter(JwtService jwtService,
                                   UserDetailsService userDetailsService) {
        this.jwtService = jwtService;
        this.userDetailsService = userDetailsService;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain)
            throws ServletException, IOException {

        // ---------- 步驟 1：從 Header 取出 Authorization ----------
        String authHeader = request.getHeader("Authorization");

        // 沒帶 token 或格式不對 → 直接放行。
        // 注意：「放行」不代表「允許存取」！後面的授權規則會判斷
        // 這個「匿名請求」能不能存取目標 API（不能就回 401）。
        if (authHeader == null || !authHeader.startsWith(BEARER_PREFIX)) {
            filterChain.doFilter(request, response);
            return;
        }

        // ---------- 步驟 2：去掉 "Bearer " 前綴，取得純 token ----------
        String token = authHeader.substring(BEARER_PREFIX.length());

        String username;
        try {
            // 解析 token 取得帳號；若 token 無效（簽章錯誤、過期、格式壞掉）
            // 會拋出例外
            username = jwtService.extractUsername(token);
        } catch (Exception e) {
            // token 無效 → 當作匿名請求放行，受保護的 API 之後自然會回 401
```

```
            filterChain.doFilter(request, response);
            return;
        }

        // ---------- 步驟 3：載入使用者並二次驗證 token ----------
        // 條件「SecurityContext 尚未有認證資訊」可避免重複處理
        if (username != null && SecurityContextHolder.getContext().getAuthentication() == null)
    ↪ {

            // 從資料庫載入使用者（含最新的角色權限）
            UserDetails userDetails = userDetailsService.loadUserByUsername(username);

            if (jwtService.isTokenValid(token, userDetails)) {

                // ---------- 步驟 4：建立「已認證」物件並放入 SecurityContext ----------
                // 三個參數：使用者資料、憑證（JWT 模式不需要，給 null）、權限清單
                UsernamePasswordAuthenticationToken authToken =
                        new UsernamePasswordAuthenticationToken(
                                userDetails, null, userDetails.getAuthorities());

                // 附加請求細節（來源 IP、Session ID 等），供稽核使用
                authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request))
    ↪ ;

                // 關鍵一步！放入 SecurityContext 後，Spring Security
                // 就視這個請求為「已登入」，後續授權規則據此判斷角色權限
                SecurityContextHolder.getContext().setAuthentication(authToken);
            }
        }

        // ---------- 步驟 5：放行，讓請求繼續往下走 ----------
        filterChain.doFilter(request, response);
    }
}
```

「放行」不等於「允許存取」

過濾器對「沒帶 token」的請求也是放行的——它只負責「有合法 token 就標記為已登入」。真正決定能不能存取的是下一關的**授權規則**：匿名請求存取受保護的 API 時，會在那裡被擋下並回 401。這樣設計讓公開 API（如查詢書籍）與受保護 API 能共用同一條過濾器鏈。

5.8 實作 4：SecurityConfig——規則總部

程式碼 5.4: config/SecurityConfig.java

```
package com.example.bookstore.config;

```

```
import com.example.bookstore.dto.ApiErrorResponse;
import com.example.bookstore.security.JwtAuthenticationFilter;
import com.fasterxml.jackson.databind.ObjectMapper;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpMethod;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.config.annotation.authentication.configuration.
    ↪ AuthenticationConfiguration;
import org.springframework.security.config.annotation.method.configuration.EnableMethodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.AuthenticationEntryPoint;
import org.springframework.security.web.access.AccessDeniedHandler;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

/**
 * Spring Security 主設定檔——認證與授權規則都集中在這裡。
 *
 * 【兩個核心名詞】
 *   認證 Authentication：「你是誰？」    → 驗帳號密碼、驗 JWT
 *   授權 Authorization ：「你能做什麼？」→ 依角色決定能存取哪些 API
 *
 * 【401 與 403 的差別】（超常考！）
 *   401 Unauthorized：沒通過「認證」——沒帶 token 或 token 無效
 *   403 Forbidden   ：通過認證但沒通過「授權」——登入了，但角色權限不足
 */
@Configuration
@EnableWebSecurity        // 啟用 Spring Security 的 Web 安全功能
@EnableMethodSecurity     // 啟用方法層級授權（@PreAuthorize 等註解）
public class SecurityConfig {

    private final JwtAuthenticationFilter jwtAuthenticationFilter;
    private final ObjectMapper objectMapper;   // Jackson 物件，用來手動輸出 JSON

    public SecurityConfig(JwtAuthenticationFilter jwtAuthenticationFilter,
                          ObjectMapper objectMapper) {
        this.jwtAuthenticationFilter = jwtAuthenticationFilter;
        this.objectMapper = objectMapper;
    }

```

```
    /**
     * 安全過濾器鏈（Security Filter Chain）——所有安全規則的核心。
     * 每個 HTTP 請求都會依序通過這條「過濾器鏈」再到達 Controller。
     */
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
                // 1. 關閉 CSRF 防護。
                //    CSRF 攻擊利用的是「瀏覽器自動帶 Cookie」的特性；
                //    本專案是無狀態 REST API，用 Header 帶 JWT、不用 Cookie，
                //    因此不受 CSRF 影響，可以安全關閉。
                .csrf(csrf -> csrf.disable())

                // 2. Session 管理設為 STATELESS（無狀態）。
                //    伺服器完全不建立、不使用 HttpSession，
                //    每個請求都只靠 JWT 自我證明身分——這是 Token 認證的精髓。
                .sessionManagement(session ->
                        session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))

                // 3. URL 授權規則（由上而下逐條比對，先中先贏）
                .authorizeHttpRequests(auth -> auth
                        // 註冊與登入：所有人（含未登入）都可存取
                        .requestMatchers("/api/auth/**").permitAll()
                        // 查詢書籍（GET）：開放訪客瀏覽
                        .requestMatchers(HttpMethod.GET, "/api/books/**").permitAll()
                        // 刪除書籍：僅限 ADMIN 角色
                        //  （hasRole("ADMIN") 會自動比對 "ROLE_ADMIN"）
                        .requestMatchers(HttpMethod.DELETE, "/api/books/**").hasRole("ADMIN")
                        // 其餘書籍操作（POST / PUT / PATCH）：USER 或 ADMIN 皆可
                        .requestMatchers("/api/books/**").hasAnyRole("USER", "ADMIN")
                        // 沒列到的任何請求：一律要求登入
                        .anyRequest().authenticated())

                // 4. 自訂「安全相關錯誤」的回應格式（預設是 HTML 錯誤頁，
                //    對 REST API 不友善，改成統一的 JSON）
                .exceptionHandling(ex -> ex
                        .authenticationEntryPoint(unauthorizedEntryPoint())  // 401
                        .accessDeniedHandler(forbiddenHandler()))            // 403

                // 5. 把我們的 JWT 過濾器插在「帳密認證過濾器」之前，
                //    讓每個請求先經過 JWT 驗證
                .addFilterBefore(jwtAuthenticationFilter,
                        UsernamePasswordAuthenticationFilter.class);

        return http.build();
    }

    /**
```

```
98       * 401 Unauthorized 的 JSON 回應。
99       * 觸發時機：匿名（或 token 無效的）請求存取需要登入的 API。
100      */
101     private AuthenticationEntryPoint unauthorizedEntryPoint() {
102         return (request, response, authException) -> writeErrorJson(
103                 response, HttpStatus.UNAUTHORIZED,
104                 "尚未登入或 Token 無效，請先登入取得有效的 Token",
105                 request.getRequestURI());
106     }
107
108     /**
109      * 403 Forbidden 的 JSON 回應。
110      * 觸發時機：已登入，但角色權限不足（例如 USER 想刪除書籍）。
111      */
112     private AccessDeniedHandler forbiddenHandler() {
113         return (request, response, accessDeniedException) -> writeErrorJson(
114                 response, HttpStatus.FORBIDDEN,
115                 "權限不足，無法執行此操作",
116                 request.getRequestURI());
117     }
118
119     /** 手動把 ApiErrorResponse 寫成 JSON 回應（過濾器層無法用 @RestControllerAdvice） */
120     private void writeErrorJson(HttpServletResponse response, HttpStatus status,
121                                 String message, String path) throws java.io.IOException {
122         response.setStatus(status.value());
123         response.setContentType(MediaType.APPLICATION_JSON_VALUE);
124         response.setCharacterEncoding("UTF-8");
125         ApiErrorResponse body = ApiErrorResponse.of(status, message, path);
126         response.getWriter().write(objectMapper.writeValueAsString(body));
127     }
128
129     /**
130      * 密碼編碼器：BCrypt。
131      * - 單向雜湊：無法從雜湊值反推出原始密碼
132      * - 自動加鹽（salt）：同一組密碼每次產生的雜湊值都不同
133      * - 計算成本可調：暴力破解的代價極高
134      * 註冊時用它加密；登入時 Spring Security 用它比對。
135      */
136     @Bean
137     public PasswordEncoder passwordEncoder() {
138         return new BCryptPasswordEncoder();
139     }
140
141     /**
142      * 認證管理器：登入 API 會呼叫它來驗證帳號密碼。
143      * 它背後會使用我們的 CustomUserDetailsService（載入使用者）
144      * 與上面的 PasswordEncoder（比對密碼）。
145      */
```

```
    @Bean
    public AuthenticationManager authenticationManager(
            AuthenticationConfiguration configuration) throws Exception {
        return configuration.getAuthenticationManager();
    }
}
```

授權規則逐條解讀（由上而下比對，先中先贏）：

規則	效果
`/api/auth/**` → `permitAll()`	註冊、登入完全公開
`GET /api/books/**` → `permitAll()`	查詢書籍開放訪客
`DELETE /api/books/**` → `hasRole("ADMIN")`	刪除僅限管理員
`/api/books/**` → `hasAnyRole("USER","ADMIN")`	其餘寫入需登入
`anyRequest()` → `authenticated()`	沒列到的一律要登入

兩種授權寫法

除了上表的「URL 規則」，也可以用「方法層級」授權——在 Controller 方法上標 `@PreAuthorize("hasRole('ADMIN')")`（需在設定類別啟用 `@EnableMethodSecurity`）。本專案的刪除 API 兩種都示範了：URL 規則當第一道防線，方法註解當第二道防線，也讓權限需求在程式碼上一目了然。

BCrypt：密碼的正確存法

資料庫永遠只存 BCrypt **雜湊值**，不存明碼：

- 單向：無法從雜湊值反推原始密碼。
- 自動加鹽：同一組密碼每次加密結果都不同，彩虹表攻擊無效。
- 慢速可調：計算成本高，暴力破解代價極大。

登入驗證是把「輸入密碼 + 雜湊值」交給 BCrypt 比對，而不是解密。

5.9 實作 5：註冊與登入 API

先看四個輕量的 DTO：

程式碼 5.5: dto/RegisterRequest.java

```
package com.example.bookstore.dto;

import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;
```

```java
import jakarta.validation.constraints.Pattern;
import jakarta.validation.constraints.Size;

/**
 * 「註冊新使用者」的請求 DTO。
 * 對應 POST /api/auth/register 的請求本文 (Request Body) 。
 */
public record RegisterRequest(

        @NotBlank(message = "帳號不可為空白")
        @Size(min = 3, max = 20, message = "帳號長度須為 3~20 字元")
        @Pattern(regexp = "^[a-zA-Z0-9_]+$",
                 message = "帳號只能包含英文字母、數字與底線")
        String username,

        @NotBlank(message = "密碼不可為空白")
        @Size(min = 8, max = 64, message = "密碼長度須為 8~64 字元")
        String password,

        @NotBlank(message = "Email 不可為空白")
        @Email(message = "Email 格式不正確")
        String email
) {
}
```

程式碼 5.6: dto/LoginRequest.java

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

import jakarta.validation.constraints.NotBlank;

/**
 * 「登入」的請求 DTO。
 * 對應 POST /api/auth/login 的請求本文 (Request Body) 。
 */
public record LoginRequest(

        @NotBlank(message = "帳號不可為空白")
        String username,

        @NotBlank(message = "密碼不可為空白")
        String password
) {
}
```

程式碼 5.7: dto/AuthResponse.java

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

```

/**
 * 「登入成功」的回應 DTO——把 JWT 交給客戶端。
 *
 * 客戶端收到後應妥善保存 token，
 * 之後每次呼叫受保護的 API 時，都要在 HTTP Header 加上：
 *
 *   Authorization: Bearer <token>
 *
 * 回應範例：
 * {
 *   "token": "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ1c2VyIi...",
 *   "tokenType": "Bearer",
 *   "expiresIn": 3600,
 *   "username": "user",
 *   "role": "ROLE_USER"
 * }
 */
public record AuthResponse(
        String token,      // JWT 字串本體
        String tokenType,  // 固定為 "Bearer"（HTTP 認證方案名稱）
        long expiresIn,    // 有效秒數（方便客戶端判斷何時該重新登入）
        String username,   // 登入者帳號
        String role        // 登入者角色（方便前端顯示對應介面）
) {
}
```

程式碼 5.8: dto/UserResponse.java——注意：絕不回傳密碼

```
package com.example.bookstore.dto;

import com.example.bookstore.entity.User;

/**
 * 「使用者資料」的回應 DTO。
 *
 * 注意：這裡「刻意」不包含 password 欄位！
 * 即使存的是 BCrypt 雜湊值，也絕對不應該回傳給客戶端。
 * 這正是「用 DTO 而不是直接回傳 Entity」的典型好處。
 */
public record UserResponse(
        Long id,
        String username,
        String email,
        String role
) {

    /** 靜態工廠方法：Entity → DTO */
```

```
    public static UserResponse from(User user) {
        return new UserResponse(
                user.getId(),
                user.getUsername(),
                user.getEmail(),
                user.getRole().name()
        );
    }
}
```

再看商業邏輯與 Controller：

程式碼 5.9: service/AuthService.java——註冊與登入邏輯

```
package com.example.bookstore.service;

import com.example.bookstore.dto.AuthResponse;
import com.example.bookstore.dto.LoginRequest;
import com.example.bookstore.dto.RegisterRequest;
import com.example.bookstore.dto.UserResponse;
import com.example.bookstore.entity.Role;
import com.example.bookstore.entity.User;
import com.example.bookstore.exception.DuplicateResourceException;
import com.example.bookstore.repository.UserRepository;
import com.example.bookstore.security.JwtService;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

/**
 * 認證 Service：處理「註冊」與「登入」的商業邏輯。
 */
@Service
public class AuthService {

    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;
    private final AuthenticationManager authenticationManager;
    private final JwtService jwtService;

    public AuthService(UserRepository userRepository,
                       PasswordEncoder passwordEncoder,
                       AuthenticationManager authenticationManager,
                       JwtService jwtService) {
        this.userRepository = userRepository;
        this.passwordEncoder = passwordEncoder;
        this.authenticationManager = authenticationManager;
```

```
        this.jwtService = jwtService;
    }

    /**
     * 註冊新使用者。
     * 流程：檢查重複 → 密碼加密 → 存入資料庫 → 回傳使用者資料（不含密碼）
     */
    @Transactional
    public UserResponse register(RegisterRequest request) {

        // 1. 檢查帳號與 Email 是否已被使用（重複則回 409 Conflict）
        if (userRepository.existsByUsername(request.username())) {
            throw new DuplicateResourceException("帳號已被使用：" + request.username());
        }
        if (userRepository.existsByEmail(request.email())) {
            throw new DuplicateResourceException("Email 已被使用：" + request.email());
        }

        // 2. 建立使用者：密碼務必先用 BCrypt 加密再儲存！
        //    passwordEncoder.encode() 每次都會產生不同的雜湊值（自動加鹽）
        User user = new User(
                request.username(),
                passwordEncoder.encode(request.password()),
                request.email(),
                Role.ROLE_USER   // 新註冊者一律是一般使用者（安全預設）
        );

        // 3. 存入資料庫並回傳（UserResponse 不包含密碼欄位）
        User saved = userRepository.save(user);
        return UserResponse.from(saved);
    }

    /**
     * 登入。
     * 流程：驗證帳號密碼 → 產生 JWT → 回傳 token 給客戶端
     */
    public AuthResponse login(LoginRequest request) {

        // 1. 把帳號密碼交給 AuthenticationManager 驗證。
        //    它會透過 CustomUserDetailsService 載入使用者、
        //    再用 BCrypt 比對密碼；驗證失敗會拋出 BadCredentialsException
        //     （由 GlobalExceptionHandler 統一轉成 401 回應）。
        authenticationManager.authenticate(
                new UsernamePasswordAuthenticationToken(
                        request.username(), request.password()));

        // 2. 驗證通過 → 從資料庫取得完整使用者資料
        User user = userRepository.findByUsername(request.username())
```

```
                .orElseThrow();   // 前面已驗證成功，理論上必定存在

        // 3. 產生 JWT 並包裝成回應
        String token = jwtService.generateToken(user.getUsername(), user.getRole());
        return new AuthResponse(
                token,
                "Bearer",
                jwtService.getExpirationMs() / 1000,   // 毫秒轉成秒
                user.getUsername(),
                user.getRole().name());
    }
}
```

程式碼 5.10: controller/AuthController.java

```
package com.example.bookstore.controller;

import com.example.bookstore.dto.AuthResponse;
import com.example.bookstore.dto.LoginRequest;
import com.example.bookstore.dto.RegisterRequest;
import com.example.bookstore.dto.UserResponse;
import com.example.bookstore.service.AuthService;
import jakarta.validation.Valid;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
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.RestController;

/**
 * 認證 Controller：提供「註冊」與「登入」兩個公開 API。
 *  (SecurityConfig 中已設定 /api/auth/** 為 permitAll，不需要 token)
 *
 * @RestController = @Controller + @ResponseBody：
 *   方法的回傳值會自動被 Jackson 序列化成 JSON 寫入回應本文。
 * @RequestMapping("/api/auth")：
 *   此類別所有 API 的共同路徑前綴。
 */
@RestController
@RequestMapping("/api/auth")
public class AuthController {

    private final AuthService authService;

    public AuthController(AuthService authService) {
        this.authService = authService;
    }
```

```
34
35      /**
36       * 註冊新使用者。
37       *
38       *   POST /api/auth/register
39       *   Body: { "username": "student01", "password": "Passw0rd!", "email": "s01@example.com" }
40       *
41       * @Valid       ：觸發 RegisterRequest 上的驗證註解，失敗回 400
42       * @RequestBody ：把請求本文的 JSON 轉成 RegisterRequest 物件
43       * @return 201 Created + 新使用者資料（不含密碼）
44       */
45      @PostMapping("/register")
46      public ResponseEntity<UserResponse> register(@Valid @RequestBody RegisterRequest request) {
47          UserResponse created = authService.register(request);
48          // 建立資源成功，回 201 Created（而不是籠統的 200）
49          return ResponseEntity.status(HttpStatus.CREATED).body(created);
50      }
51
52      /**
53       * 登入並取得 JWT。
54       *
55       *   POST /api/auth/login
56       *   Body: { "username": "user", "password": "User@123" }
57       *
58       * @return 200 OK + token（客戶端之後要放在 Authorization: Bearer <token>）
59       *         帳密錯誤則回 401（由 GlobalExceptionHandler 處理）
60       */
61      @PostMapping("/login")
62      public ResponseEntity<AuthResponse> login(@Valid @RequestBody LoginRequest request) {
63          return ResponseEntity.ok(authService.login(request));
64      }
65  }
```

5.10 種子資料與測試帳號

為了讓大家一啟動就能測試，專案在啟動時自動建立測試帳號與範例書籍：

程式碼 5.11: config/DataInitializer.java——教學用種子資料

```
1  package com.example.bookstore.config;
2
3  import com.example.bookstore.entity.Book;
4  import com.example.bookstore.entity.Role;
5  import com.example.bookstore.entity.User;
6  import com.example.bookstore.repository.BookRepository;
7  import com.example.bookstore.repository.UserRepository;
8  import org.slf4j.Logger;
```

```
import org.slf4j.LoggerFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Component;

import java.math.BigDecimal;

/**
 * 初始資料建立器——「教學專用」的種子資料（Seed Data）。
 *
 * 實作 CommandLineRunner 介面的 Bean，
 * 會在 Spring Boot「啟動完成後」自動執行一次 run() 方法。
 *
 * 這裡預先建立：
 *   1. 兩個測試帳號（admin 管理員、user 一般使用者）
 *   2. 三本範例書籍
 * 讓大家一啟動就能直接用 Postman 測試，不必先手動建資料。
 *
 * 注意：正式環境不會這樣做！正式專案通常使用
 * Flyway / Liquibase 這類「資料庫版本控管工具」管理初始資料。
 */
@Component
public class DataInitializer implements CommandLineRunner {

    private static final Logger log = LoggerFactory.getLogger(DataInitializer.class);

    private final UserRepository userRepository;
    private final BookRepository bookRepository;
    private final PasswordEncoder passwordEncoder;

    public DataInitializer(UserRepository userRepository,
                           BookRepository bookRepository,
                           PasswordEncoder passwordEncoder) {
        this.userRepository = userRepository;
        this.bookRepository = bookRepository;
        this.passwordEncoder = passwordEncoder;
    }

    @Override
    public void run(String... args) {

        // ---------- 建立測試帳號（資料表是空的才建立，避免重複） ----------
        if (userRepository.count() == 0) {

            // 管理員：擁有全部權限（含 DELETE）
            userRepository.save(new User(
                    "admin",
                    passwordEncoder.encode("Admin@123"),   // 密碼一律加密後才儲存
```

```
                    "admin@example.com",
                    Role.ROLE_ADMIN));

            // 一般使用者：可查詢、新增、修改（不能刪除）
            userRepository.save(new User(
                    "user",
                    passwordEncoder.encode("User@123"),
                    "user@example.com",
                    Role.ROLE_USER));

            log.info("已建立測試帳號 -> admin / Admin@123（管理員）、user / User@123（一般使用
    ↪ 者）");
        }

        // ---------- 建立範例書籍 ----------
        if (bookRepository.count() == 0) {

            bookRepository.save(new Book(
                    "Clean Code", "Robert C. Martin", "9780132350884",
                    new BigDecimal("1200.00"), 10,
                    "無瑕的程式碼：敏捷軟體開發技巧守則。"));

            bookRepository.save(new Book(
                    "Effective Java", "Joshua Bloch", "9780134685991",
                    new BigDecimal("1500.00"), 5,
                    "Java 程式設計的最佳實務指南，共 90 個條目。"));

            bookRepository.save(new Book(
                    "Spring 實戰", "Craig Walls", "9787115582866",
                    new BigDecimal("880.00"), 20,
                    "Spring in Action：涵蓋 Spring Boot 的實戰經典。"));

            log.info("已建立 {} 本範例書籍", bookRepository.count());
        }
    }
}
```

帳號	密碼	角色
admin	Admin@123	ROLE_ADMIN（可刪除書籍）
user	User@123	ROLE_USER

5.11 完整流程測試（curl 版）

```
# 1. 登入，取得 token（回應中的 token 欄位）
curl -X POST http://localhost:8080/api/auth/login \
  -H "Content-Type: application/json" \
```

```
  -d '{"username": "user", "password": "User@123"}'

# 2. 把 token 存進 shell 變數（手動複製貼上也可以）
TOKEN="<把上一步回應中的 token 貼到這裡>"

# 3. 帶著 token 新增書籍 -> 201 Created
curl -i -X POST http://localhost:8080/api/books \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $TOKEN" \
  -d '{"title": "測試書", "author": "我", "isbn": "9990000000001",
       "price": 350, "stock": 5, "description": "curl 測試"}'

# 4. 不帶 token 新增 -> 401 Unauthorized（認證失敗）
curl -i -X POST http://localhost:8080/api/books \
  -H "Content-Type: application/json" -d '{}'

# 5. 用一般使用者的 token 刪除 -> 403 Forbidden（授權失敗）
curl -i -X DELETE http://localhost:8080/api/books/1 \
  -H "Authorization: Bearer $TOKEN"
```

指令列測試能用，但效率不高——下一章改用 Postman，讓 token 自動儲存、自動帶入，還能寫自動化驗證腳本。

第 6 章　使用 Postman 測試 API

本章使用教材附帶的 Postman 測試集合，學會匯入集合、管理變數、閱讀與撰寫自動化測試腳本（Tests），並用 Collection Runner 一鍵跑完整套測試。

6.1　為什麼用 Postman？

curl 適合快速測試，但有明顯限制：token 要手動複製、請求無法儲存重用、回應是純文字不好讀。Postman 解決了這些問題：

- **請求集合（Collection）**：把所有 API 請求分類儲存，可匯出分享給團隊。
- **變數**：`{{baseUrl}}`、`{{token}}` 一處修改處處生效。
- **Scripts**：請求前後可執行 JavaScript——自動存 token、自動驗證回應。
- **Runner**：整個集合依序自動執行，變成一鍵回歸測試。

6.2　匯入測試集合

教材附帶兩個檔案（位於 `postman/` 資料夾）：

檔案	內容
`Bookstore-API.postman_collection.json`	測試集合：15 個請求 + 自動化腳本
`Bookstore-Local.postman_environment.json`	環境檔：本機開發的 `baseUrl`

匯入步驟：

1. 開啟 Postman，點左上角 **Import** 按鈕。
2. 把兩個 JSON 檔一起拖進視窗（或點 files 選取）。
3. 完成後左側 Collections 出現「Bookstore API 教學」，右上角環境下拉選單可選「Bookstore Local（本機開發）」。

6.3　Postman 變數機制

變數用雙大括號引用：`{{baseUrl}}/api/books`。依作用範圍由大到小：

範圍	適用情境
Global	跨所有 collection 共用（少用）
Environment	隨環境切換的值：本機 / 測試機 / 正式機的 `baseUrl`
Collection	該集合內共用：本教材的 `token`、`bookId`
Local	單次執行暫存

小提示

同名變數「範圍小的優先」：Environment 的 `baseUrl` 會蓋過 Collection 的 `baseUrl`。本教材集合內建 `baseUrl` 預設值，所以就算不選環境也能直接使用；之後若要打測試機，只要新增一個環境改掉 `baseUrl` 即可，集合完全不用動。

6.4 集合結構導覽

資料夾	內容
1. 認證 Auth	註冊、登入（user 與 admin）；登入後自動儲存 token
2. 書籍 Books CRUD	GET / POST / PUT / PATCH / DELETE 完整流程
3. 錯誤情境測試	刻意觸發 400 / 401 / 403 / 404 / 409

集合層級已設定 **Bearer Token 認證**，值為 `{{token}}`——資料夾內的請求預設「繼承」這個設定（Inherit auth from parent），所以登入一次之後，所有需要認證的請求都會自動帶上 `Authorization: Bearer <token>` Header。

6.5 跟著做：完整測試流程

6.5.1 步驟 1：登入並自動儲存 token

確認專案已啟動，執行「1-2 登入（一般使用者 user）」。切到請求的 **Scripts** → **Post-response**（舊版稱 Tests）分頁，看看這段每次收到回應後自動執行的腳本：

程式碼 6.1: 登入請求的 Tests 腳本

```
pm.test('登入成功：狀態碼應為 200 OK', function () {
    pm.response.to.have.status(200);
});

// 解析回應 JSON
const json = pm.response.json();

```

```
pm.test('回應應包含 token', function () {
    pm.expect(json.token).to.be.a('string').and.to.have.length.above(0);
});

// 【重點】把 token 存入 collection 變數。
// 之後所有請求的 Authorization 都設定為 Bearer {{token}}，
// 因此登入一次，整個 collection 就能直接使用！
pm.collectionVariables.set('token', json.token);
```

送出後觀察下方 **Test Results** 分頁：兩個測試都顯示綠色 PASS。接著執行「1-3 登入（管理員 admin）」，它會把管理員的 token 存入 `{{adminToken}}`（刪除書籍時使用）。

6.5.2 步驟 2：走一遍 CRUD

依序執行「2. 書籍 Books CRUD」資料夾內的請求，重點觀察：

請求	觀察重點
2-1 GET 分頁查詢	回應的分頁欄位；改 `sortBy=price` 看排序變化
2-4 POST 新增	狀態碼 201、`Location` Header；新書 id 自動存入 `{{bookId}}`
2-5 PUT 完整更新	body 需帶「全部」欄位；試著刪掉一個欄位觀察 400
2-6 PATCH 部分更新	body 只帶 `price`；回應中其他欄位不變
2-7 DELETE 刪除	使用 `{{adminToken}}`；成功回 204 無本文

隨機 ISBN 的小技巧

「2-4 POST 新增書籍」設有 Pre-request 腳本：每次送出前產生隨機 ISBN 存入 `{{randomIsbn}}`，讓這個請求可以重複執行而不會因 ISBN 重複而收到 409。打開該請求的 **Scripts** → **Pre-request** 分頁即可看到寫法。

6.5.3 步驟 3：錯誤情境

「3. 錯誤情境測試」資料夾刻意觸發每一種錯誤，並用 Tests 腳本驗證狀態碼與錯誤格式：

情境	狀態碼	對應機制
3-1 未帶 token 新增書籍	401	SecurityConfig 的 EntryPoint
3-2 一般使用者嘗試刪除	403	角色授權規則
3-3 查詢不存在的書籍	404	ResourceNotFoundException
3-4 空白書名 + 負數價格	400	Bean Validation
3-5 重複的 ISBN	409	Service 商業規則

6.6 撰寫 Tests：常用 API 速查

Postman 的腳本使用 JavaScript，搭配內建的 `pm` 物件：

語法	用途
`pm.test(名稱, fn)`	定義一個測試（顯示於 Test Results）
`pm.response.to.have.status(200)`	驗證狀態碼
`pm.response.json()`	把回應本文解析成 JSON 物件
`pm.response.to.have.header('X')`	驗證 Header 存在
`pm.expect(值).to.eql(預期)`	驗證值相等（Chai 斷言庫）
`pm.collectionVariables.set(k, v)`	寫入集合變數
`pm.environment.set(k, v)`	寫入環境變數
`console.log(...)`	印到 Console（View → Show Postman Console）

範例——驗證回應時間與欄位型別：

```
pm.test('回應時間應小於 500ms', function () {
    pm.expect(pm.response.responseTime).to.be.below(500);
});

pm.test('price 應為數字', function () {
    const json = pm.response.json();
    pm.expect(Number(json.price)).to.be.a('number');
});
```

6.7 Collection Runner：一鍵跑完整套測試

1. 在集合名稱上按右鍵 → **Run collection**。
2. 保持預設順序（順序就是設計好的測試流程：先登入再 CRUD 再錯誤情境）。
3. 按 **Run**，Postman 會依序送出所有請求並統計測試結果。
4. 全部綠色 PASS = 整個後端的行為都符合預期——這就是最簡單的回歸測試！

之後每次改完程式，跑一次 Runner 就知道有沒有改壞東西。

6.8 如何匯出自己的 Collection

教材附帶的集合檔就是這樣匯出的，日後你建立自己的集合也可以分享給團隊：

1. 在集合名稱上按右鍵 → **Export**。
2. 選擇格式 **Collection v2.1（recommended）** → Export。

3. 存成 JSON 檔，即可放入版本控制或傳給同事匯入。

注意

匯出前記得檢查集合變數中有沒有殘留的真實 token 或密碼——變數的「Current value」不會被匯出，但「Initial value」會！機密資料請只放在 Current value，或匯出前清空。

第 7 章　Docker 容器化部署

最後一哩路：把專案打包成 Docker 映像檔，讓它在任何機器上都能一致地執行；再用 Docker Compose 把「MySQL + API」包成一鍵啟動的整組服務。

7.1　為什麼需要 Docker？

「在我電腦上明明可以跑！」——傳統部署的萬年名句。問題出在環境差異：JDK 版本、系統函式庫、環境變數、資料庫版本……Docker 的解法是把**應用程式與它需要的整個執行環境**打包在一起：

名詞	意義
映像檔 Image	唯讀的打包成品：程式 + 執行環境 + 設定（像「安裝光碟」）
容器 Container	映像檔執行起來的實體（像「開機後的電腦」），可同時跑多個
Registry	存放映像檔的倉庫（Docker Hub 就是官方公共倉庫）
Dockerfile	描述「如何建置映像檔」的腳本

重點觀念

映像檔在你電腦上能跑，在同事電腦、測試機、雲端上就都能跑——因為執行環境已經「凍結」在映像檔裡了。這也是為什麼 Spring Boot（內嵌 Tomcat、單一 JAR）與 Docker 是天作之合。

7.2　Dockerfile：逐行解說

專案根目錄的 `Dockerfile` 採用**多階段建置（Multi-stage Build）**：

程式碼 7.1: Dockerfile——多階段建置

```
# ========================================================================
# Dockerfile : 把 Spring Boot 專案打包成 Docker 映像檔 (Image)
# ========================================================================
# 這裡採用「多階段建置 (Multi-stage Build) 」 :
```

```
#
#   第 1 階段（build）   ：用「含 Maven + JDK 的大映像檔」編譯打包出 JAR
#   第 2 階段（runtime）：只用「輕量的 JRE 映像檔」執行 JAR
#
# 好處：最終映像檔「不包含」原始碼、Maven、JDK 編譯器，
#       體積從 800MB+ 縮小到約 300MB，攻擊面也更小。
#
# 建置指令：docker build -t bookstore-api:1.0.0 .
# 執行指令：docker run -p 8080:8080 bookstore-api:1.0.0
# ============================================================================

# ---------------- 第 1 階段：建置（Build Stage） ----------------
# 基底映像檔：官方 Maven 3.9 + Eclipse Temurin JDK 17
# 「AS build」幫這個階段取名字，第 2 階段要用名字來複製檔案
FROM maven:3.9-eclipse-temurin-17 AS build

# 設定容器內的工作目錄（之後的指令都在這個目錄下執行）
WORKDIR /app

# 【Layer Cache 快取技巧】先只複製 pom.xml 並下載相依套件。
# Docker 會把每個步驟的結果做成「層（Layer）」並快取：
# 只要 pom.xml 沒變動，重新建置時就會直接沿用快取，
# 跳過最耗時的「下載相依套件」步驟（改程式碼不會重新下載套件）。
COPY pom.xml .
RUN mvn dependency:go-offline -B
# -B（batch mode）：非互動模式，輸出比較乾淨，適合自動化環境

# 再複製原始碼並打包成 JAR（跳過測試以加快建置）
COPY src ./src
RUN mvn package -DskipTests -B
# 完成後 JAR 位於：/app/target/bookstore-api-1.0.0.jar

# ---------------- 第 2 階段：執行（Runtime Stage） ----------------
# 基底映像檔改用「只含 JRE」的輕量版本（不含編譯器）
FROM eclipse-temurin:17-jre-jammy

WORKDIR /app

# 【資安最佳實務】建立系統帳號來執行程式，不要用 root 執行！
# 萬一程式被攻破，攻擊者拿到的也只是低權限帳號
RUN useradd --system --create-home spring
USER spring

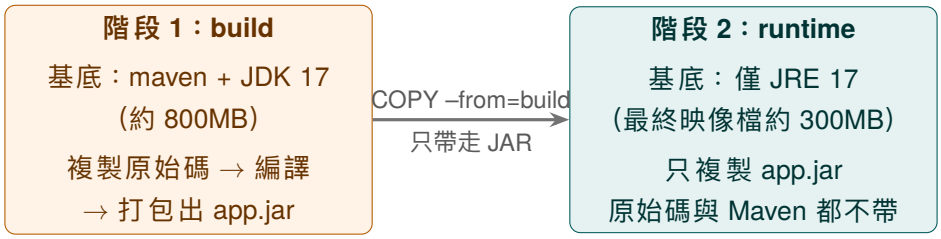
# 從第 1 階段（build）複製「打包好的 JAR」進來——
# 這就是多階段建置的精髓：只帶走成品，不帶走工具與原始碼
COPY --from=build /app/target/*.jar app.jar

# 聲明容器對外服務的連接埠（文件性質；實際對外仍需 docker run -p 參數）
```

```
EXPOSE 8080

# 容器啟動時執行的指令：java -jar app.jar
# 使用「exec 形式」（JSON 陣列）讓 Java 程序成為 PID 1，
# 才能正確接收 docker stop 的關閉訊號（graceful shutdown）
ENTRYPOINT ["java", "-jar", "app.jar"]
```

多階段建置的流程與效益：



Layer 快取：為什麼先 COPY pom.xml 再 COPY src？

Dockerfile 的每個指令會產生一「層」快取，只要該步驟的輸入沒變就直接重用。相依套件很少變、原始碼天天變——所以先只複製 `pom.xml` 下載套件（可快取），再複製 `src` 編譯。這樣日常改程式重建映像檔時，不用重新下載幾百 MB 的相依套件，建置時間從幾分鐘縮到幾十秒。

`.dockerignore` 排除不需要送進建置環境的檔案（類似 `.gitignore`）：

程式碼 7.2: .dockerignore

```
# ============================================================================
# .dockerignore : 排除不需要送進 Docker 建置環境的檔案
# ============================================================================
# docker build 時會把整個資料夾（建置上下文 build context）傳給 Docker，
# 排除不必要的檔案可以：
#   1. 加快建置速度（少傳輸大量檔案）
#   2. 避免快取失效（target/ 內容常變動，會害 COPY 步驟的快取失效）
#   3. 避免機密檔案（.env 等）意外被打包進映像檔
# ============================================================================

# Maven 建置輸出（容器內會重新編譯，不需要本機的舊產物）
target/

# IDE 設定檔
.idea/
*.iml
.vscode/
.settings/
.classpath
.project

```

```
# 版本控制
.git/
.gitignore

# 日誌與暫存
logs/
*.log

# 環境變數檔（可能含機密）
.env
```

7.3 建置映像檔：docker build

```
# 在 bookstore-api/ 目錄下執行（注意最後的「.」＝建置上下文為目前目錄）
#   -t 名稱:標籤   幫映像檔取名字與版本標籤
docker build -t bookstore-api:1.0.0 .

# 查看建好的映像檔
docker images
# REPOSITORY      TAG      IMAGE ID       CREATED         SIZE
# bookstore-api   1.0.0    3f5a1b2c4d6e   10 seconds ago  ~300MB
```

第一次建置需要下載基底映像檔與 Maven 套件，會花數分鐘；之後拜 Layer 快取所賜會快非常多。

7.4 執行容器：docker run

```
# 執行容器：
#   -d          在背景執行（detached）
#   -p 8080:8080 連接埠對映「本機:容器」
#   -e KEY=VAL  設定環境變數（覆寫 application.yml 的預設值）
#   --name      幫容器取名字，方便後續操作
docker run -d --name bookstore-api \
  -p 8080:8080 \
  -e DB_HOST=host.docker.internal \
  -e DB_PASSWORD=root1234 \
  bookstore-api:1.0.0

# 追蹤日誌（-f 持續輸出，Ctrl+C 離開）
docker logs -f bookstore-api
```

容器裡的 localhost 不是你的電腦！

容器有自己獨立的網路空間——在容器裡連 `localhost:3306`，連到的是「容器自己」，不是你電腦上的 MySQL。所以上面把 `DB_HOST` 設為 `host.docker.internal`（Docker Desktop 提供的特殊主機名稱，指向宿主機）。下一節的 Docker Compose 會用更優雅的方式解決這個問題。

7.5 Docker Compose：一鍵啟動整組服務

手動管理「先啟動 MySQL、等它就緒、再啟動 API、還要設定網路」太繁瑣。Docker Compose 把整組服務宣告在一個 YAML 檔裡：

程式碼 7.3: docker-compose.yml——MySQL + API 一鍵啟動

```yaml
# ============================================================================
# docker-compose.yml : 一鍵啟動「MySQL + Spring Boot API」整組服務
# ============================================================================
# Docker Compose 用一個 YAML 檔描述多個容器（服務）的組合，
# 常用指令：
#   docker compose up -d --build   建置並在背景啟動所有服務
#   docker compose logs -f app     持續查看 app 服務的日誌
#   docker compose ps              查看服務狀態
#   docker compose down            停止並移除所有容器（資料卷保留）
#   docker compose down -v         連同資料卷一起刪除（資料庫歸零）
#
# 【重要觀念】Compose 會自動建立一個「虛擬網路」，
# 容器之間可以直接用「服務名稱」當主機名稱互相連線——
# 所以 app 連 MySQL 的位址是 mysql:3306，不是 localhost:3306！
# （在容器裡，localhost 指的是「容器自己」）
# ============================================================================

services:

  # ---------------- 服務 1：MySQL 資料庫 ----------------
  mysql:
    image: mysql:8.4                    # 使用官方 MySQL 8.4（LTS 版本）
    container_name: bookstore-mysql
    restart: unless-stopped             # 異常關閉時自動重啟
    environment:
      # MySQL 官方映像檔提供的初始化環境變數：
      MYSQL_ROOT_PASSWORD: root1234     # root 密碼（教學用；正式環境請改用 secrets）
      MYSQL_DATABASE: bookstore_db      # 首次啟動時自動建立這個資料庫
      TZ: Asia/Taipei                   # 容器時區
    ports:
      # 連接埠對映「主機:容器」：
      # 讓本機的 MySQL Workbench / DBeaver 也能連 localhost:3306 觀察資料
      - "3306:3306"
```

```
    volumes:
      # 資料卷：把資料庫檔案存在 Docker 管理的空間，
      # 容器刪除重建後「資料仍然保留」
      - mysql-data:/var/lib/mysql
    healthcheck:
      # 健康檢查：每 5 秒 ping 一次 MySQL，確認它「真的可以服務」
      #  （容器啟動 ≠ MySQL 初始化完成，兩者可能差數十秒！）
      test: ["CMD", "mysqladmin", "ping", "-h", "127.0.0.1", "-uroot", "-proot1234"]
      interval: 5s        # 每 5 秒檢查一次
      timeout: 3s         # 單次檢查逾時時間
      retries: 10         # 連續失敗 10 次才視為 unhealthy
      start_period: 30s   # 給 MySQL 30 秒的初始化寬限期

  # ---------------- 服務 2：Spring Boot API ----------------
  app:
    build: .                              # 用本目錄的 Dockerfile 建置映像檔
    image: bookstore-api:1.0.0            # 建置出的映像檔名稱與標籤
    container_name: bookstore-api
    restart: unless-stopped
    depends_on:
      mysql:
        # 不只等 mysql 容器「啟動」，而是等它通過健康檢查「真正可用」
        # 才啟動 app，避免 app 啟動時連不上資料庫而崩潰
        condition: service_healthy
    environment:
      # 覆寫 application.yml 中的 ${DB_HOST:localhost} 等預設值。
      # ★ 關鍵：DB_HOST 是服務名稱「mysql」，不是 localhost！
      DB_HOST: mysql
      DB_PORT: "3306"
      DB_NAME: bookstore_db
      DB_USERNAME: root
      DB_PASSWORD: root1234
      # 正式環境的 JWT 密鑰請務必改成自己產生的隨機字串
      JWT_SECRET: docker-compose-secret-key-change-me-0123456789ABCDEF
      JWT_EXPIRATION_MS: "3600000"
    ports:
      - "8080:8080"                       # 本機 8080 → 容器 8080

# 具名資料卷的宣告（由 Docker 統一管理實際儲存位置）
volumes:
  mysql-data:
```

三個關鍵設計：

- **服務名稱＝主機名稱**：Compose 自動建立虛擬網路，app 直接用 `mysql:3306` 連資料庫（所以環境變數設 `DB_HOST: mysql`）。
- **healthcheck + depends_on**：MySQL 容器「啟動」不代表「初始化完成」（可能差幾十秒）。

健康檢查確認 MySQL 真的能服務後，才啟動 app，避免連線失敗。

- **具名資料卷 mysql-data**：資料庫檔案存在 Docker 管理的卷中，容器刪掉重建，資料依然保留。

啟動與驗證：

```
# 建置映像檔並在背景啟動所有服務
docker compose up -d --build

# 查看服務狀態（等 mysql 顯示 healthy、app 顯示 running）
docker compose ps

# 追蹤 API 日誌，看到 "Started BookstoreApiApplication" 就緒
docker compose logs -f app

# 測試 API（此時整組服務都跑在 Docker 裡！）
curl http://localhost:8080/api/books

# 全部停止並移除容器（資料卷保留，資料不會消失）
docker compose down

# 連資料卷一起刪除（資料庫歸零，重新開始）
docker compose down -v
```

7.6 常用指令速查

指令	用途
`docker ps（-a）`	列出執行中（含已停止）的容器
`docker images`	列出本機映像檔
`docker logs -f < 名稱 >`	追蹤容器日誌
`docker exec -it < 名稱 > bash`	進入容器內的終端機
`docker stop / start < 名稱 >`	停止 / 啟動容器
`docker rm < 名稱 >`	刪除容器
`docker rmi < 映像檔 >`	刪除映像檔
`docker compose up -d --build`	建置並啟動整組服務
`docker compose down`	停止並移除整組服務
`docker system prune`	清理沒用到的容器 / 映像檔 / 快取

7.7 完整驗證：從零到整組服務上線

最終驗收——確認任何一台裝了 Docker 的電腦都能跑起整套系統：

1. `docker compose up -d --build`，等待建置完成。

2. `docker compose ps` 確認 `mysql (healthy)` 與 `app` 都在執行。
3. 開啟 Postman，跑一次第 6 章的 Collection Runner——全部 PASS。
4. `docker compose down` 收工；下次 `up` 資料依然都在。

恭喜！

你已經完成了一個具備認證授權、連接真實資料庫、可容器化部署、附完整測試集合的後端專案——這正是業界後端工程師日常工作的縮影。附錄提供常見問題排除與延伸學習方向，祝學習順利！

附錄 A　常見問題排除（FAQ）

A.1　啟動與資料庫連線

Q1：啟動時出現 Communications link failure

原因：連不上 MySQL。

排查順序：

1. MySQL 有沒有在跑？（`docker ps` 或 `brew services list`）
2. 連接埠對不對？預設 3306，若有更改需同步調整連線字串。
3. 帳號密碼與 `application.yml` 是否一致？
4. 在 Docker 容器內跑 API 時，`DB_HOST` 不能是 `localhost`——請改用 `host.docker.internal`，或 Compose 的服務名稱（見第 7 章）。

Q2：Port 8080 was already in use

原因：8080 被其他程式佔用。

解法：找出並關閉佔用程式（macOS：`lsof -i :8080`），或修改 `application.yml` 的 `server.port`。

Q3：Access denied for user 'root'@'localhost'

原因：資料庫密碼錯誤。

解法：確認密碼；用環境變數覆寫：`DB_PASSWORD= 你的密碼 mvn spring-boot:run`。

Q4：中文存進資料庫變成問號或亂碼

解法：確認連線字串含 `characterEncoding=UTF-8`，且資料庫使用 `utf8mb4` 字元集（第 3 章的建庫指令）。

A.2　JWT 與權限

Q5：所有請求都回 401

排查順序：

1. Header 格式必須是 `Authorization: Bearer <token>`——注意 `Bearer` 後面有一個空格。
2. token 是否過期？（預設 1 小時，重新登入即可）

3. 伺服器重啟後若換了 `JWT_SECRET`，舊 token 全部失效。
4. Postman 中確認變數 `{{token}}` 有值（滑鼠移到變數上可預覽）。

Q6：登入成功，但呼叫 DELETE 回 403

原因：403 表示「已認證但權限不足」——你用的是 `ROLE_USER` 的 token。
解法：改用 admin 帳號登入取得的 token（Postman 中是 `{{adminToken}}`）。

Q7：The specified key byte array is 248 bits which is not secure enough

原因：JWT 密鑰太短，HS256 要求至少 256 bits（32 字元）。
解法：把 `app.jwt.secret` 改成 32 字元以上的字串。

A.3 Maven 與 Docker

Q8：mvn 指令找不到（command not found）

解法：安裝 Maven（第 1 章），或使用專案附的 Maven Wrapper：`./mvnw spring-boot:run`。

Q9：Cannot connect to the Docker daemon

原因：Docker Desktop 沒有啟動。
解法：開啟 Docker Desktop 應用程式，等狀態變成執行中再重試。

Q10：docker compose up 後 app 一直重啟

排查：`docker compose logs app` 看錯誤訊息。最常見是 MySQL 還沒就緒——本專案已用 healthcheck 處理；若自行修改過 compose 檔，確認 `depends_on` 的 `condition: service_healthy` 還在。

附錄 B　指令速查表

B.1　Maven

指令	用途
`mvn spring-boot:run`	啟動開發伺服器
`mvn clean package`	清除並打包成 JAR
`mvn clean package -DskipTests`	打包但跳過測試
`mvn test`	執行測試
`mvn dependency:tree`	查看相依套件樹（排查版本衝突）

B.2　MySQL

```
-- 進入 MySQL（Docker 版）：
--   docker exec -it bookstore-mysql mysql -uroot -proot1234

SHOW DATABASES;              -- 列出所有資料庫
USE bookstore_db;            -- 切換資料庫
SHOW TABLES;                 -- 列出資料表
DESCRIBE books;              -- 查看資料表結構
SELECT * FROM books;         -- 查詢資料
SELECT * FROM users;         -- 查看使用者（密碼是 BCrypt 雜湊值）
```

B.3　curl 測試 API

```
# GET（-i 顯示狀態列與 Header）
curl -i "http://localhost:8080/api/books?page=0&size=5"

# POST（-X 指定方法、-H 加 Header、-d 附上 JSON 本文）
curl -X POST http://localhost:8080/api/auth/login \
  -H "Content-Type: application/json" \
  -d '{"username": "admin", "password": "Admin@123"}'

# 帶 JWT 的請求
curl -X DELETE http://localhost:8080/api/books/1 \
  -H "Authorization: Bearer $TOKEN"
```

附錄 C　延伸學習方向

完成本教材後，建議依序探索以下主題，逐步補齊業界實務的完整拼圖：

- **API 文件自動化**：springdoc-openapi——加一個相依套件，自動產生 Swagger UI 互動式 API 文件。
- **資料庫版本控管**：Flyway 或 Liquibase——用版本化的 SQL 腳本管理資料表結構（取代 `ddl-auto: update`）。
- **自動化測試**：JUnit 5 + MockMvc（Controller 測試）、Testcontainers（在測試中啟動真實 MySQL 容器）。
- **Refresh Token**：Access Token 短效期 + Refresh Token 換發機制，兼顧安全性與使用者體驗。
- **關聯對映進階**：`@OneToMany` / `@ManyToMany`、LAZY 載入與 N+1 查詢問題。
- **快取與效能**：Spring Cache + Redis。
- **CI/CD**：GitHub Actions——push 後自動測試、建置映像檔、部署。
- **雲端部署**：把 Docker 映像檔部署到 AWS / GCP / Azure 或 Zeabur、Railway 等平台。

學習愉快，程式路上一帆風順！