🕮 逢甲大學資工系學士後專班之專案管理期末報告：

全端外語學習書電子商務系統之開發與架構實踐

Full-Stack E-commerce System for Foreign Language Books: Architecture and Implementation

■ 第一部分：報告者資訊與技術大綱

1. 報告者資訊

- **姓名：** 周彥廷
- **專業背景：** 全端開發與軟體工程實踐（專精於 Spring Boot 與 Vue 3 整合架構）
- **語言專長：** 日語能力試驗 JLPT N3、TOEIC 740（具備閱讀國際技術文件與開發雙語應用之專業能力）

2. 專案摘要 (Abstract)

本專案為一套獨立開發之全端電子商務平台，結合個人外語學習背景，實作具備 14 種語系精確檢索、ACID 事務性管理與無狀態認證架構之書籍銷售系統。專案不僅完成功能實作，更深研資料庫約束、API 安全性與生產環境部署優化，展現從系統架構設計至維運調優之全生命週期開發能力（SDLC）。

3. 開發核心理念與轉化目標 (Core Philosophy)

- **系統化軟體工程實踐**：致力於計算機科學領域的深層應用，透過實作大型軟體專案建立厚實的**資料結構**與**分散式系統架構**觀念。
- **軟體品質與嚴謹性**：核心設計不侷限於 UI 呈現，而是追求後端資料的一致性（Transactional Integrity）與認證體系的安全性（JWT/CORS），體現專業開發者對軟體品質的追求。

- **跨領域技術整合**：利用語言優勢直接對接國際技術社群，並將其應用於 14 種學習語系之複合檢索與 UGC 評論系統實作，解決特定垂直領域的檢索痛點。

4. 核心技術棧 (Technical Stack)

類別	採用的技術與工具
後端框架	Java 17, Spring Boot 3 (Spring MVC, Spring Data JPA)
安全驗證	Spring Security, JWT (Stateless Authentication)
資料庫系統	PostgreSQL (Supabase), JPA/Hibernate 關係模型設計
前端開發	Vue 3 (Composition API), Pinia (狀態管理), Vite
部署維運	Docker 容器化部署, Vercel Rewrite 規則, Vitest 測試環境

■ 第二部分：系統架構設計與資料持久化模型 (Architecture & Data Modeling)

1. 前後端分離架構 (Decoupled Architecture)

本系統採用現代化前後端分離架構，確保系統具備高擴展性與維護性：

- **前端層 (Frontend)**：基於 **Vue 3 (Composition API)** 與 **Vite** 構建，利用 **Pinia** 進行全域狀態管理（如用戶認證與購物車同步）。
- **後端層 (Backend)**：以 **Spring Boot 3** 為核心，採用 **RESTful API** 規範與前端通訊，並整合 **Spring Security** 實作安全性防護。
- **通訊協定**：全面導入 **JWT (JSON Web Token)** 實作無狀態認證，並透過 **Axios 攔截器** 自動處理 Token 注入與 401 異常響應。

2. 關聯式資料庫設計 (ERD & Persistence)

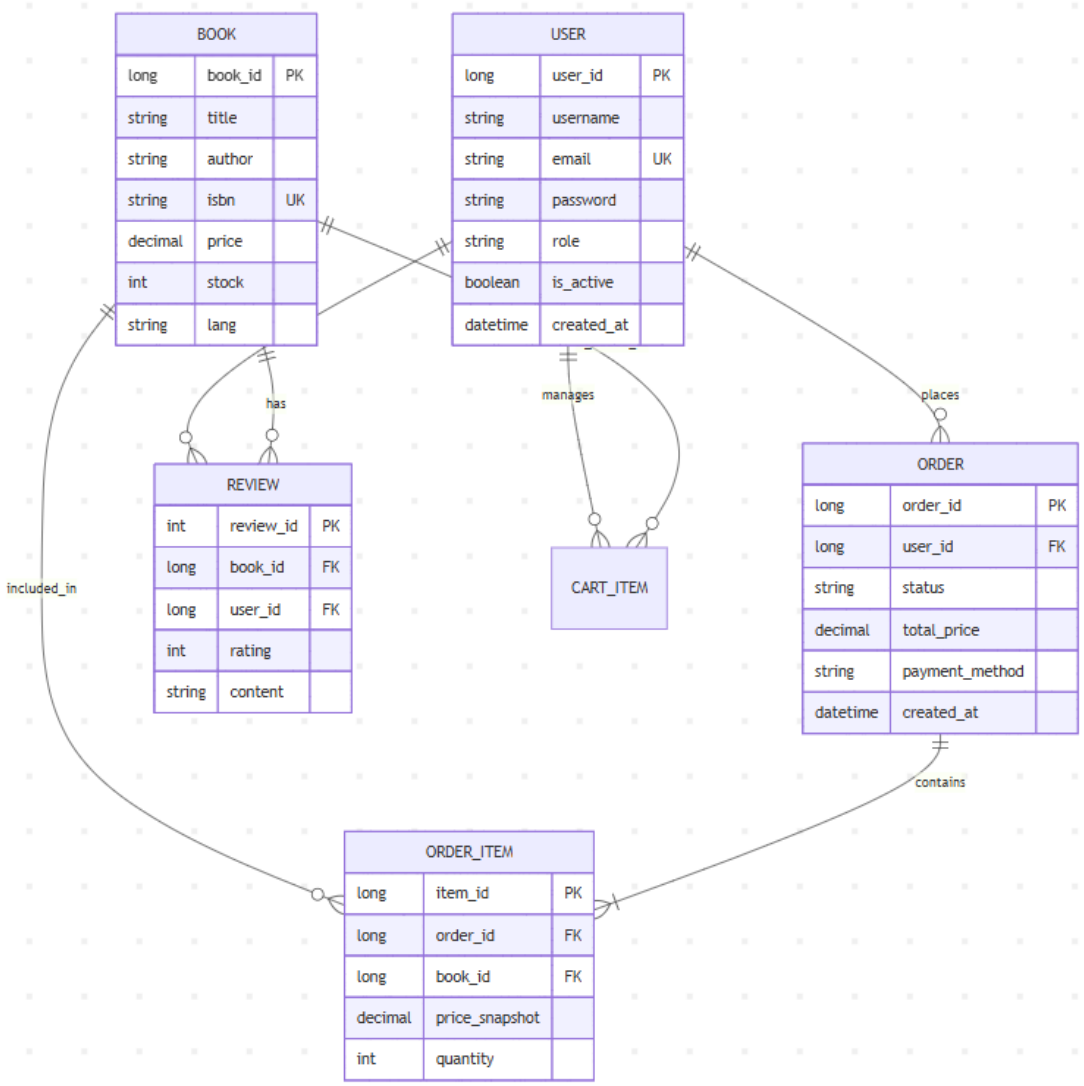
系統底層基於 **PostgreSQL**，透過 **JPA/Hibernate** 進行物件關係映射（ORM），精確定義業務邏輯中的實體鏈條：

- **核心實體關聯**：
  - **User - Order**：一對多關聯，維護用戶歷史消費紀錄。
  - **Order - OrderItem - Book**：透過組合模式管理訂單明細，並實作「結帳價格快照」機制，避免因書籍調價導致的歷史訂單數據異常。
  - **Book - Review - User**：構建用戶生成內容（UGC）生態，實作複雜的多對一關聯鏈。
- **資料完整性約束 (Integrity Constraints)**：
  - 利用 JPA 的 `@Enumerated(EnumType.STRING)` 確保 14 種語言分類的類型安全。
  - 在持久層實作 `Check Constraint`，確保庫存（Stock）與價格（Price）始終符合業務邏輯規範（≥ 0）。

3. 數據存取優化 (Data Access Optimization)

- **解決 N+1 查詢問題**：在處理訂單詳情與評論列表時，顯式使用 **JPQL** `LEFT JOIN FETCH` 技術，強制在一條 SQL 語句中加載關聯實體，大幅降低資料庫 I/O 頻率。
- **懶加載策略 (Lazy Loading)**：針對非即時顯示的關聯數據（如書籍的詳細描述或用戶頭像）使用 `FetchType.LAZY`，優化 JVM 內存使用效率。

4. ERD 圖 (Entity Relationship Diagram)



■ 第三部份：核心業務邏輯與事務傳播機制 (Business Logic & Transaction)

1. 健壯的狀態變更邏輯與錯誤處理

本系統之訂單狀態更新邏輯 `updateOrderStatusAndGetDetail` 實作了多重防護機制：

- **輸入清洗與標準化**：透過 `.trim().toUpperCase()` 處理前端傳入之字串，並利用 `try-catch` 捕獲非法狀態枚舉值，避免系統因非預期輸入而崩潰。
- **業務流程閉環**：精確判斷狀態轉換條件（如從非 CANCELLED 轉為 CANCELLED），確保庫存回補邏輯 `restoreStock` 僅在必要時觸發。

2. 事務傳播機制 (Transaction Propagation) 的實踐

在訂單取消流程中，庫存回補邏輯 `restoreStock` 被設計為 `private` 方法，並納入父方法的事務生命週期：

- **技術抉擇**：由於 `restoreStock` 在標註了 `@Transactional` 的 `updateOrderStatusAndGetDetail` 內部被調用，系統利用了 Spring 的 **REQUIRED 事務傳播特性**。
- **原子性保證**：若後續執行 `orderRepository.save(order)` 失敗，或拋出任何 RuntimeException，先前執行的庫存回補操作將同步回滾（Rollback），確保數據一致性。

3. 核心邏輯程式碼 Snippet

```
243     @Transactional  1 usage   Chris Chou
244 @   public OrderDetailDTO updateOrderStatusAndGetDetail(Long orderId, String newStatus) {
245         Order order = orderRepository.findById(orderId)
246                 .orElseThrow(() -> new OrderNotFoundException("訂單 ID: " + orderId + " 未找到"));
247
248         // 🌟 修正點 1: 加上 .toUpperCase() 並處理空格，防止前端小寫造成的 400 錯誤
249         OrderStatus nextStatus;
250         try {
251             nextStatus = OrderStatus.valueOf(newStatus.trim().toUpperCase());
252         } catch (IllegalArgumentException e) {
253             throw new RuntimeException("不支援的訂單狀態: " + newStatus);
254         }
255
256         // 🌟 修正點 2: 庫存回補邏輯
257         if (nextStatus == OrderStatus.CANCELLED && order.getStatus() != OrderStatus.CANCELLED) {
258             restoreStock(order);
259         }
260
261         // 🌟 修正點 3: 調整取消限制（如果你希望管理員擁有最高權限強行取消，請移除或註解掉這段）
262      if (nextStatus == OrderStatus.CANCELLED && order.getStatus() == OrderStatus.PAID) {
263         // 如果是期末專案為了方便演示，建議把這個限制拿掉，或者讓管理員可以取消
264     }
265
266
267         order.setStatus(nextStatus);
268         orderRepository.save(order);
269
270         return mapToDetailDTO(order);
271     }
```

■ 第四部份：API 安全性設計與無狀態認證架構 (Security & Access Control)

本系統採用 **無狀態 (Stateless)** 認證機制，確保伺服器資源在大量併發下具備橫向擴展能力，並嚴格落實最小權限原則。

1. JWT 認證過濾器鏈 (JWT Authentication Filter Chain)

系統透過自定義的 JwtAuthenticationFilter 在每個 Request 進入業務邏輯前進行身分驗證：

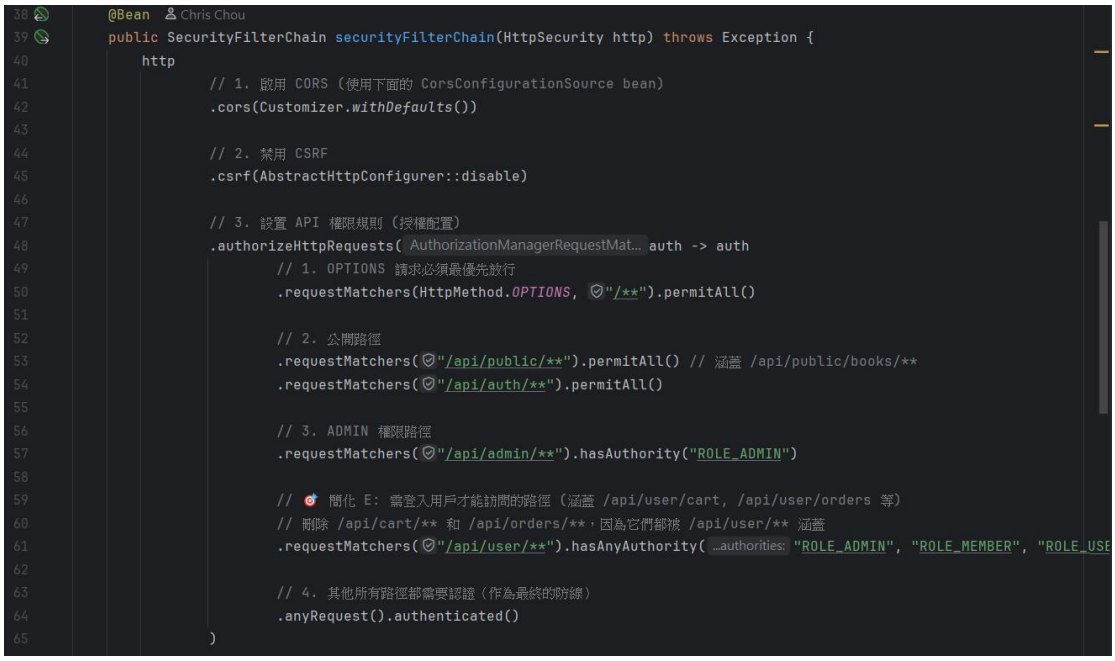
- **攔截與解析**：從 Authorization Header 中提取 Bearer Token，並利用 JwtService 進行 Base64 解碼與簽章驗證。
- **安全性驗證**：檢查 Token 是否過期（Expiration）以及 Subject (Email) 是否與資料庫用戶匹配。
- **上下文注入**：驗證成功後，將用戶權限（Role）注入至 SecurityContextHolder，供後續 @PreAuthorize 或路由攔截器使用。

2. 基於路徑的權限隔離設計 (Path-based Access Control)

在 SecurityConfig 中，根據資源敏感度將 API 分為三個層級，從底層杜絕非法訪問：

- **公開路徑 (/api/public/**)**：放行書籍瀏覽、搜尋與評論讀取，不需認證即可存取，提升 SEO 與首頁載入效率。
- **會員路徑 (/api/user/**)**：僅限持有有效 JWT 且角色為 USER 或 ADMIN 者訪問，保護購物車、訂單與個人資料。
- **管理端路徑 (/api/admin/**)**：嚴格限制角色必須具備 ROLE_ADMIN，確保書籍上架、訂單管理等高權限操作之安全性。

3. 關鍵安全性配置 Snippet



```
@Bean  Chris Chou
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
            // 1. 啟用 CORS (使用下面的 CorsConfigurationSource bean)
            .cors(Customizer.withDefaults())

            // 2. 禁用 CSRF
            .csrf(AbstractHttpConfigurer::disable)

            // 3. 設置 API 權限規則 (授權配置)
            .authorizeHttpRequests( AuthorizationManagerRequestMat... auth -> auth
                    // 1. OPTIONS 請求必須最優先放行
                    .requestMatchers(HttpMethod.OPTIONS, "/**").permitAll()

                    // 2. 公開路徑
                    .requestMatchers("/api/public/**").permitAll() // 涵蓋 /api/public/books/**
                    .requestMatchers("/api/auth/**").permitAll()

                    // 3. ADMIN 權限路徑
                    .requestMatchers("/api/admin/**").hasAuthority("ROLE_ADMIN")

                    // 🎯 簡化 E: 需登入用戶才能訪問的路徑 (涵蓋 /api/user/cart, /api/user/orders 等)
                    // 刪除 /api/cart/** 和 /api/orders/**，因為它們都被 /api/user/** 涵蓋
                    .requestMatchers("/api/user/**").hasAnyAuthority( ...authorities: "ROLE_ADMIN", "ROLE_MEMBER", "ROLE_USE

                    // 4. 其他所有路徑都需要認證 (作為最終的防線)
                    .anyRequest().authenticated()
            )
```

```
                    // 3. ADMIN 權限路徑
                    .requestMatchers("/api/admin/**").hasAuthority("ROLE_ADMIN")

                    // 🎯 簡化 E: 需登入用戶才能訪問的路徑 (涵蓋 /api/user/cart, /api/user/orders 等)
                    // 刪除 /api/cart/** 和 /api/orders/**，因為它們都被 /api/user/** 涵蓋
                    .requestMatchers("/api/user/**").hasAnyAuthority( ...authorities: "ROLE_ADMIN", "ROLE_MEMBER", "ROLE_USER")

                    // 4. 其他所有路徑都需要認證 (作為最終的防線)
                    .anyRequest().authenticated()
            )

            // 4. 設置會話管理為無狀態 (JWT)
            .sessionManagement( SessionManagementConfigurer<HttpSecurity> session -> session
                    .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            )

            // 5. 設置身份驗證提供者
            .authenticationProvider(authenticationProvider)

            // 6. 添加 JWT 過濾器 (在 UsernamePasswordAuthenticationFilter 之前執行)
            .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class);

    return http.build();
}
```

■ 第五部份：資料傳輸層 (DTO) 封裝與系統解耦設計 (Data Transfer Object Pattern)

為了提升系統安全性、降低網路傳輸負載並解決 JPA 循環引用問題，本系統在 Controller 與 Service 層之間全面導入了 **DTO (Data Transfer Object)** 模式。

1. 資訊隱私與安全性封裝 (Security & Privacy)

直接暴露資料庫實體（Entity）會導致內部結構（如密碼 Hash、版本號）外流。透過 DTO 模式，本系統達成精確的欄位過濾：

- **敏感資料屏蔽**：在 UserResponse 中僅回傳用戶名與權限，嚴格禁止回傳密碼相關欄位。
- **入參合法性檢查**：在 RegisterRequest 中排除 role 與 isActive 標記，強制由後端業務邏輯決定初始權限，杜絕前端惡意竄改。

2. 解決關聯序列化之循環引用 (Circular Reference Handling)

在處理 Order 與 OrderItem 的雙向關聯時，直接序列化會導致堆疊溢位 (StackOverflow)：

- **嵌套 DTO 設計**：定義 OrderDetailDTO 嵌套 OrderItemDTO 列表，將複雜的雙向關聯簡化為單向的樹狀結構。
- **業務快照紀錄**：OrderItemDTO 中記錄了結帳當下的價格快照 (price)，而非實時關聯書籍現價，這確保了交易紀錄的歷史準確性。

3. 關鍵實作程式碼 (Mapping Logic)

本系統使用 Java 8 Stream API 實作實體與 DTO 之間的精確映射：

```java
/**
 * 將 Order 實體轉換為 OrderDetailDTO
 */
private OrderDetailDTO mapToDetailDTO(Order order) {  3 usages  Chris Chou
    List<OrderItemDTO> itemDTOs = order.getItems().stream()  Stream<OrderItem>
            .map(this::mapOrderItemToDTO)  Stream<OrderItemDTO>
            .collect(Collectors.toList());

    return OrderDetailDTO.builder()
            .orderId(order.getOrderId())
            .userId(order.getUserId())
            .status(order.getStatus().name())
            .totalPrice(order.getTotalPrice())
            // 🌟 核心檢查點：這裡必須確保有拿到資料
            .paymentMethod(order.getPaymentMethod() != null ? order.getPaymentMethod().name() : "CASH_ON_DELIVERY")
            .createdAt(order.getCreatedAt())
            .recipientName(order.getRecipientName())
            .recipientPhone(order.getRecipientPhone())
            .shippingAddress(order.getShippingAddress())
            .items(itemDTOs)
            .build();
}
```

■ 第六部份：前端組件化開發與動態視圖設計 (Frontend Architecture & UI)

本系統前端基於 **Vue 3 Composition API** 構建，強調「組件化」與「狀態驅動」，確保在高複雜度的電商業務下仍能維持優異的用戶體驗與裝置適應性。

1. 基於 Vue 3 響應式原理的動態 UI (Reactive UI)

系統利用 Vue 的 ref、computed 與 watch 實作資料與視圖的即時同步：

- **狀態驅動交互**：例如在 BooksView.vue 中，利用 computed 屬性動態過濾書籍列表，並結合 debounce 函數優化搜尋引擎的效能，減少無謂的 API 調用。
- **條件式渲染**：透過 v-if 與 v-for 根據 API 回傳狀態切換「載入中」、「無結果」或「書籍清單」，提升前端的魯棒性。

2. 系統響應式佈局策略 (Responsive Strategy)

雖然樣式表（CSS）負責最終呈現，但本系統在架構層級即導入了適應性設計思維：

- **語義化佈局 (Semantic HTML)**：使用 <header>, <main>, <section> 等 HTML5 標籤構建結構，這不僅有利於 SEO，更是 CSS Flexbox 與 Grid 佈局實作響應式的基礎。
- **後端數據支援**：後端傳回的 DTO（如 imageUrl）支援前端根據裝置解析度動態加載圖片，配合 CSS 的 max-width: 100% 實作跨螢幕適配。

3. 關鍵交互邏輯程式碼 (UX Logic)

```
// 🎯 判斷書籍是否已在購物車（用於按鈕狀態控制）
const isBookInCart = (bookId) => {  Show usages  Chris Chou
  return cartStore.items.some(item => item.book.bookId === bookId);
};

const fetchBooks = async () => {  Show usages  Chris Chou
  isLoading.value = true;
  error.value = null;
  try {
    let response;
    if (searchKeyword.value) {
      response = await BookService.searchBooks(searchKeyword.value);
    } else if (selectedLang.value !== 'all') {
      response = await BookService.getBooksByLang(selectedLang.value);
    } else {
      response = await BookService.searchBooks( keyword: '');
    }
    books.value = response.data;
  } catch (err) {
    error.value = '載入書籍列表失敗。' + (err.response?.data || '');
    console.error(err);
  } finally {
    isLoading.value = false;
  }
};
```

```
const debounceSearch = () => {  Show usages  Chris Chou
  clearTimeout(searchTimeout);
  searchTimeout = setTimeout( handler: () => {
    fetchBooks();
  },  timeout: 300);
};

const viewDetail = (bookId) => {  Show usages  Chris Chou
  router.push(`/book/${bookId}`);
};
```

■ 第七部份：全域狀態管理與業務邏輯流轉 (Global State & Business Flow)

本系統採用 **Pinia** 作為核心狀態管理工具，透過單一數據源（Single Source of Truth）的原則，解決了複雜組件樹中的 Prop Drilling 與數據不同步問題。

1. 核心狀態模組化設計 (Modular Store Design)

系統將狀態拆分為兩大自治模組，實現關注點分離：

- **Auth Store (認證模組)**：管理用戶 JWT Token、權限等級（Role）與登入狀態。透過持久化儲存機制，確保頁面重整後仍能維持認證狀態，並支援 API 攔截器進行身分校驗。
- **Cart Store (購物車模組)**：集中處理商品的新增、修改數量與刪除操作。當用戶在 BookDetailView 新增商品時，導航列的購物車圖示能即時透過響應式連結同步更新數據。

2. 業務流程中的原子狀態更新

在電商核心流程「結帳」中，Store 負責協調不同 Service 之間的數據流轉：

- **庫存動態校驗**：在 updateCartItem 動作中，結合 API 請求進行即時庫存驗證，防止用戶在前端將超過庫存數量的商品加入購物車。
- **結帳後狀態清理**：當 OrderService.checkout 成功回應後，由 Store 觸發 clearCart 動作，確保前端介面與資料庫端同步重設，維持業務閉環的完整性。

3. 關鍵 Store 邏輯 Snippet (以購物車為例)

```
69    async updateCartItem(bookId, quantity) : Promise<void> {
70        // 1. 檢查商品是否已在購物車中（根據 bookId）
71        const existingItem = this.items.find(item => item.book.bookId === bookId);
72
73        // 2. 🎯 實踐親戚的建議：
74        // 如果商品已存在，且這次呼叫是來自「加入購物車」按鈕（通常 quantity 會傳 1 或目前的增量）
75        // 我們可以判斷：如果商品已存在，就彈出提醒並中止操作。
76        if (existingItem) {
77            alert(`🛒 購物車已有此商品！\n欲修改數量請至購物車頁面進行調整。`);
78            return; // 攔截操作，不發送 API 請求
79        }
80
81        if (quantity <= 0) {
82            if (existingItem) {
83                await this.deleteCartItem(existingItem.cartItemId);
84            }
85            return;
86        }
```

```
81        if (quantity <= 0) {
82            if (existingItem) {
83                await this.deleteCartItem(existingItem.cartItemId);
84            }
85            return;
86        }
87
88        try {
89            const payload : {bookId: any, quantity: any} = { bookId, quantity };
90            const response : Object = await CartService.addOrUpdateCartItem(payload);
91            const updatedItem = response.data;
92
93            const index : number = this.items.findIndex(item => item.cartItemId === updat
94
95            if (index !== -1) {
96                this.items[index] = updatedItem;
97            } else {
98                this.items.push(updatedItem);
99            }
100       } catch (error) {
101           console.error('更新購物車失敗:', error.response?.data?.message || error.message
102           throw error;
103       }
104   },
```

■ 第八部份：持久層架構設計與數據庫檢索優化 (Repository & Query Optimization)

本系統持久層基於 **Spring Data JPA**，透過物件關係映射（ORM）簡化數據操作。針對電商系統常見的高密度查詢需求，本專案實作了多項針對性的性能優化與安全防護。

1. 解決 N+1 查詢效能瓶頸 (Join Fetch 實踐)

在處理「訂單詳情」等跨多表關聯的業務時，傳統 JPA 的 Lazy Loading 會導致系統發送過多 SQL 請求（N+1 問題）。本系統透過顯式定義 **JOIN FETCH** 進行優化：

- **單一語義查詢**：利用 @Query 語法強制在單次資料庫連接中加載 OrderItem 與關聯的 Book 實體。
- **效能提升**：將原本 O(N) 的查詢複雜度降至 O(1)，大幅減少資料庫連接池的壓力，縮短 API 響應時間。

2. 安全性前置防範：橫向越權攻擊之防杜

在 Repository 層級即實作嚴格的「權限歸屬檢查」，確保資料存取的物理安全性：

- **複合條件校驗**：例如在 existsByCartItemIdAndUserUserId 方法中，強制同時驗證資源 ID 與當前認證用戶 ID。
- **防禦性檢索**：這能從底層杜絕惡意用戶透過列舉（Enumeration）攻擊嘗試獲取非本人訂單或購物車資料的風險。

3. 關鍵持久層實作 (Query Snippets)

```
21      @Query("SELECT o FROM Order o " +  1 usage  Chris Chou
22              "LEFT JOIN FETCH o.items oi " +
23              "LEFT JOIN FETCH oi.book b " +
24              "WHERE o.orderId = :orderId AND o.user.userId = :userId")
25      Optional<Order> findByIdAndUserIdWithDetails(
26              @Param("orderId") Long orderId,
27              @Param("userId") Long userId
28      );
```

■ 第九部份：前端 Service 封裝與請求攔截機制 (API Services & Interceptors)

為確保前端代碼的可讀性與可維護性，本系統將所有的網路請求邏輯從 UI 組件中抽離，構建了一套基於 **Axios** 的多層級 API 服務架構。

1. 單一職責原則的 Service 模組化 (Modular Services)

針對不同的業務領域，系統劃分了獨立的 Service 模組，每個模組僅負責與後端對應的 Controller 進行通訊：

- **AuthService**：專門處理登入、註冊與憑證發送。
- **AdminBook/OrderService**：封裝後端管理介面的複雜操作，確保前台用戶視圖不受後台邏輯干擾。
- **優點**：當後端 API 路徑變動時，僅需修改單一 Service 檔案即可完成全域同步，大幅提升系統的**可重構性 (Refactorability)**。

2. 全域請求攔截與 Token 自動化注入

透過配置 `apiClient` 實作自動化的安全防護與異常處理：

- **請求攔截器 (Request Interceptor)**：在發送請求前自動從 `localStorage` 或 Pinia Store 讀取 JWT，並注入至 `Authorization` Header。
- **響應攔截器 (Response Interceptor)**：監控後端回傳的狀態碼。若捕獲 **401 Unauthorized**，則判定為憑證失效，自動執行清除狀態並重導向至登入頁，實現優雅的錯誤處理機制。

3. 關鍵 API 封裝實作 Snippet

```
checkout(payload) : Promise<Object> {
    if (!payload.paymentMethod || !payload.recipientName || !payload.shippingAddress) {
        return Promise.reject(new Error("結帳請求缺少必要的收件資訊。"));
    }
    return apiClient.post( url: `/user/orders/checkout`, payload);
},
```

■ 第十部份：管理端權限隔離與架構職責拆解 (Admin Authorization & Architecture Roles)

此部分詳述系統如何針對管理權限進行物理隔離，並透過 Controller 與 Service 的職責分離，確保系統具備高度的安全性與可維護性。

1. 管理端控制器的獨立化與權限攔截 (Controller Layer)

為落實安全防護，本系統將管理功能從一般用戶邏輯中抽離，獨立出 AdminUserController。其核心職責在於 HTTP 協議層級的通訊與異常轉換：

- **權限硬隔離**：所有 API 路徑皆以 /api/admin/ 為前綴，並透過 Spring Security 強制要求 ROLE_ADMIN 權限。
- **通訊與回應封裝**：Controller 僅負責解析路徑參數並調用下游服務。當業務邏輯拋出異常時，Controller 負責捕捉並封裝為 ResponseEntity.badRequest()，確保前端能獲得標準化的錯誤提示。

2. 服務層的事務管理與狀態演進 (Service Layer)

真正的業務核心與數據變更邏輯封裝於 UserService 中，並透過 @Transactional 註解確保操作的原子性：

- **數據庫事務保護**：將 @Transactional 標註於 Service 層，確保當 userRepository.save() 出現任何資料持久化錯誤時，系統能完整回滾（Rollback），防止數據產生中間狀態。
- **架構優勢**：這種設計使得業務邏輯與通訊協議解耦。未來若需更換 Controller 或導入定時任務，仍能複用具備事務保護的 Service 邏輯。

3. 核心程式碼職責對照 (Code Comparison)

【Controller 層：負責通訊與異常捕捉】

```
34      @PatchMapping("/{userId}/toggle-active")  Chris Chou
35      public ResponseEntity<?> toggleUserActive(@PathVariable Integer userId) {
36          try {
37              userService.toggleUserActive(userId);
38              return ResponseEntity.ok(Map.of( k1: "message",  v1: "用戶狀態已成功更新"));
39          } catch (RuntimeException e) {
40              return ResponseEntity.badRequest().body(e.getMessage());
41          }
42      }
```

【Service 層：負責事務與邏輯執行】

```
167     @Transactional  1 usage  Chris Chou
168     public void toggleUserActive(Integer userId) {
169         User user = userRepo.findById(Long.valueOf(userId))
170                 .orElseThrow(() -> new RuntimeException("用戶 ID " + userId + " 不存在"));
171
172         // 直接反轉狀態
173         user.setIsActive(!user.getIsActive());
174         userRepo.save(user);
175     }
```

■ 第十一部分：系統可靠性診斷與單元測試實踐 (Reliability & Testing)

此部分展示系統在開發過程中的品質管理機制，包含核心邏輯的偵錯（Debug）流程以及自動化測試環境的建置，體現軟體工程中的品質保證（QA）精神。

1. 系統診斷與權限鏈偵錯 (Diagnostics & Debugging)

為了解決 Spring Security 在複雜路徑下的權限判定問題，本專案實作了動態診斷機制：

- **認證上下文追蹤**：在 AdminOrderController 中實作權限監控邏輯，透過提取 SecurityContextHolder 中的 Authorities，解決了 JWT 角色注入不一致的潛在 Bug。
- **異常過濾與捕獲**：利用 JwtAuthenticationFilter 監控請求生命週期，確保任何無效 Token 都能被精確攔截並回傳 401 狀態碼，而非導致後端崩潰。

2. 前端自動化測試環境 (Testing Environment)

本專案導入 **Vitest** 作為測試引擎，模擬真實瀏覽器行為以驗證前端邏輯：

- **環境隔離設計**：配置 jsdom 環境以執行組件掛載測試，確保 UI 邏輯與 DOM 操作的正確性。
- **持續整合基礎**：透過定義 include 規則掃描 *.spec.js 測試檔案，為未來導入 CI/CD 自動化部署流程奠定測試基礎。

3. 關鍵測試配置 Snippet

第十二部分：全端系統功能驗證與演示 (System Demo)

為完整呈現本系統之交互邏輯與技術實作，本人錄製了詳盡的演示影片。影片涵蓋了從前端用戶體驗到後端管理營運的完整閉環，驗證了系統在不同裝置（手機端與桌機端）下的穩定性。

演示影片技術導覽 (Video Timeline)

【前台模組：行動端用戶交互與搜尋引擎】

- 0:00 - 0:33 | **多維度檢索功能**：展示基於「作者名」與「國際書號 (ISBN)」的精確/模糊查詢邏輯。
- 0:33 - 0:46 | **語系篩選器**：展示 14 種語系下拉式選單的即時過濾效果，體現了高效的資料檢索設計。
- 0:47 - 1:03 | **社交評論機制**：展示書籍評論功能，驗證了多對一 (Many-to-One) 資料關聯之呈現。
- 1:16 - 2:32 | **購物車狀態維護與邏輯保護**：
    - 實作「重複購買提醒」機制，防止使用者誤重複加入相同商品。
    - 展示動態修改購物車數量，驗證前端狀態管理（Pinia）與後端計算之連動。
- 2:33 - 3:12 | **支付模擬與交易軌跡**：模擬金融卡支付流程，並即時生成過往交易紀錄，驗證了資料庫的事務一致性（ACID）。

【後台模組：桌面端營運管理與權限控制】

- 3:15 - 3:28 | **管理員認證入口**：演示管理端專屬登入流程，體現了基於角色的存取控制 (RBAC)。
- 3:29 - 4:21 | **書籍元數據維護**：展示管理員對書籍資訊、狀態之即時編輯功能。
- 4:30 - 6:13 | **核心訂單調度系統**：展示完整的訂單生命週期管理。透過 DTO 模式優化海量訂單的呈現，並實作即時狀態更新功能，展現後台管理系統之營運可靠性。

🔗 演示影片存取資訊

- **YouTube 連結**：https://youtu.be/Hswm_nmELEQ?si=TDhiG-o2449yLaRN
- **影片配樂說明**：採用富有節奏感的熱血音樂，象徵本人對解決技術難題與開發全端系統的熱情。

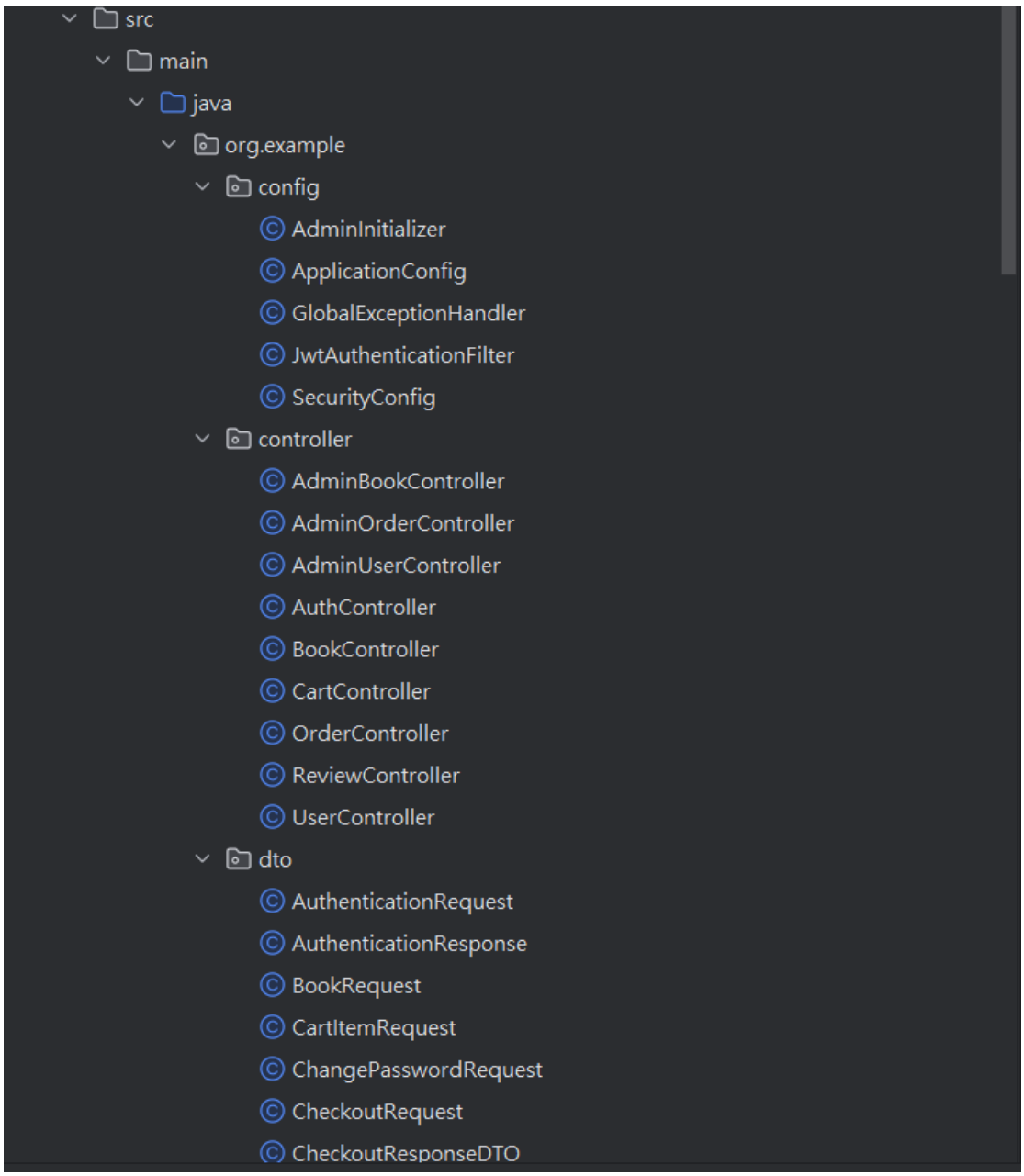
GitHub Repository：

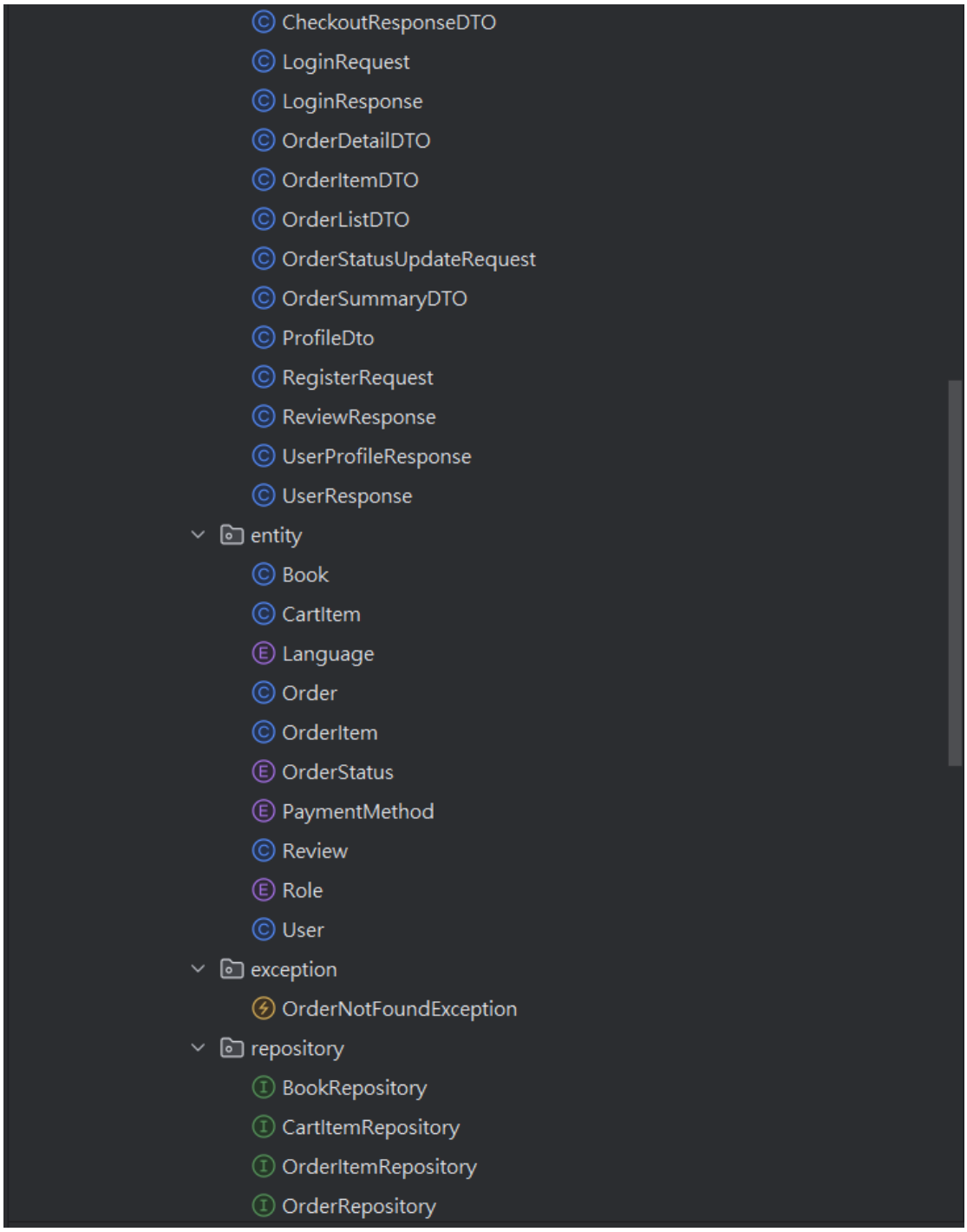
https://github.com/ChrisChou19990904/Foreign_Languages_Book_Back_End.git

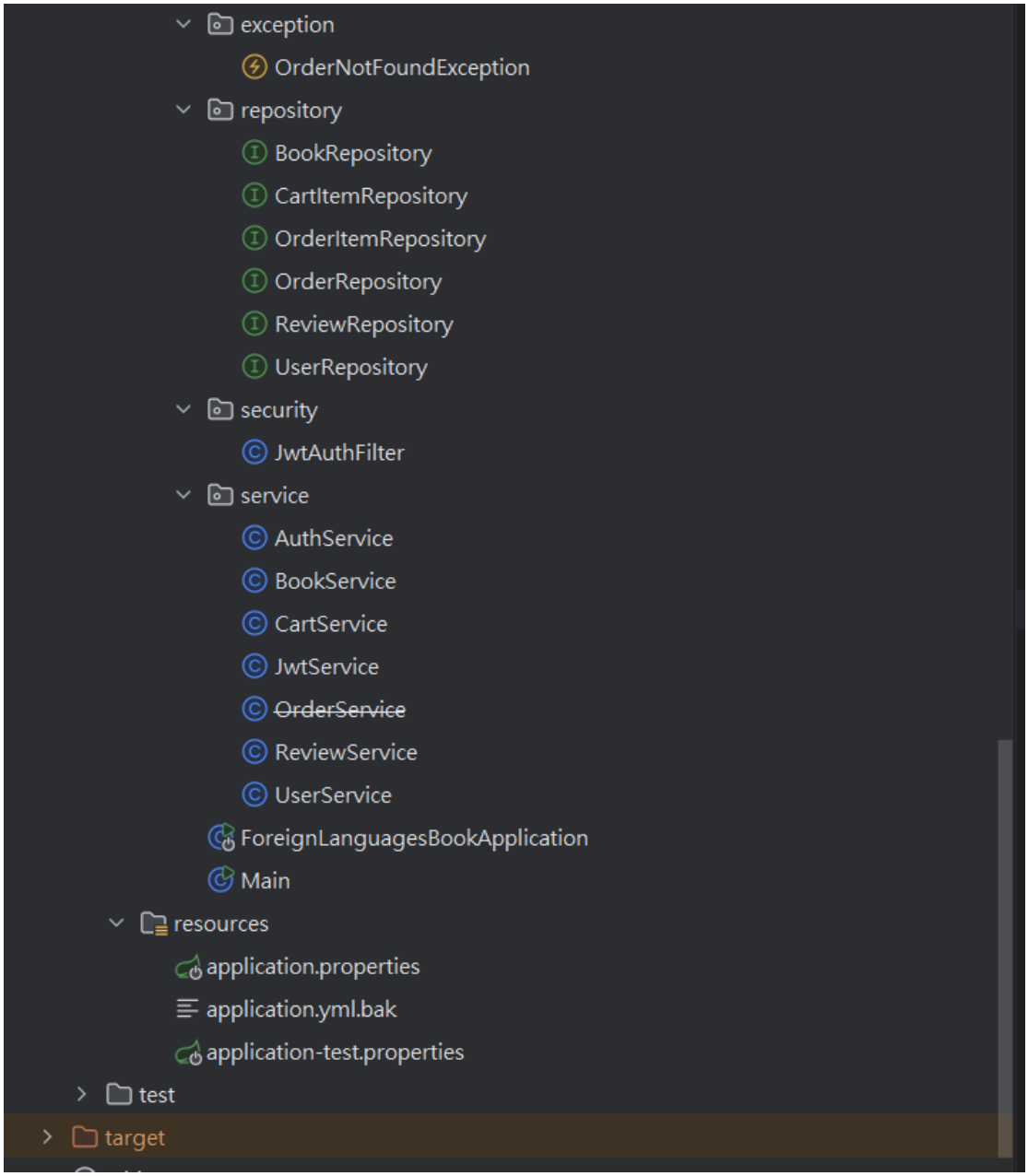
https://github.com/ChrisChou19990904/foreign_languages_book.git

■ 第十三部分：專案路徑截圖

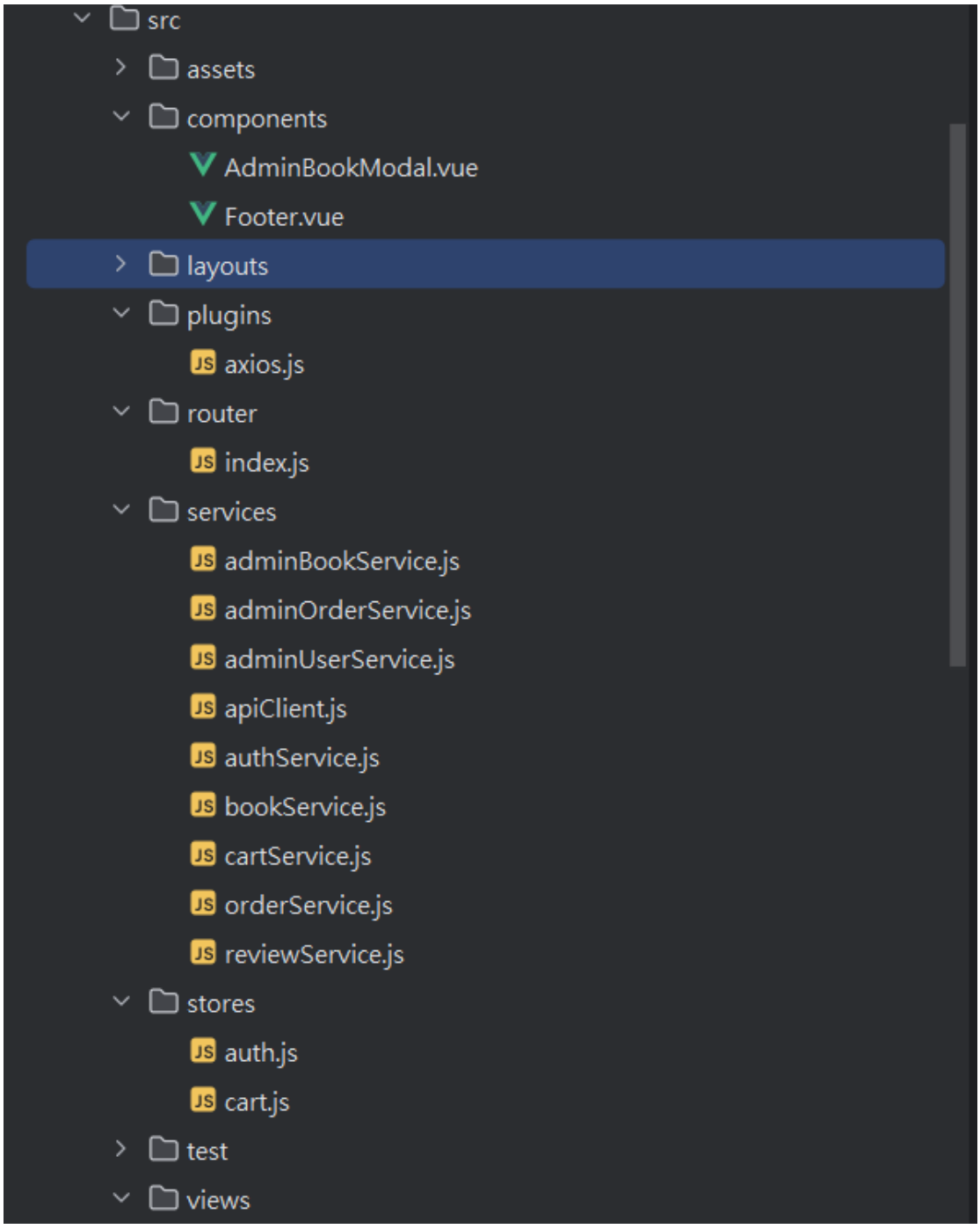
後端的部分：

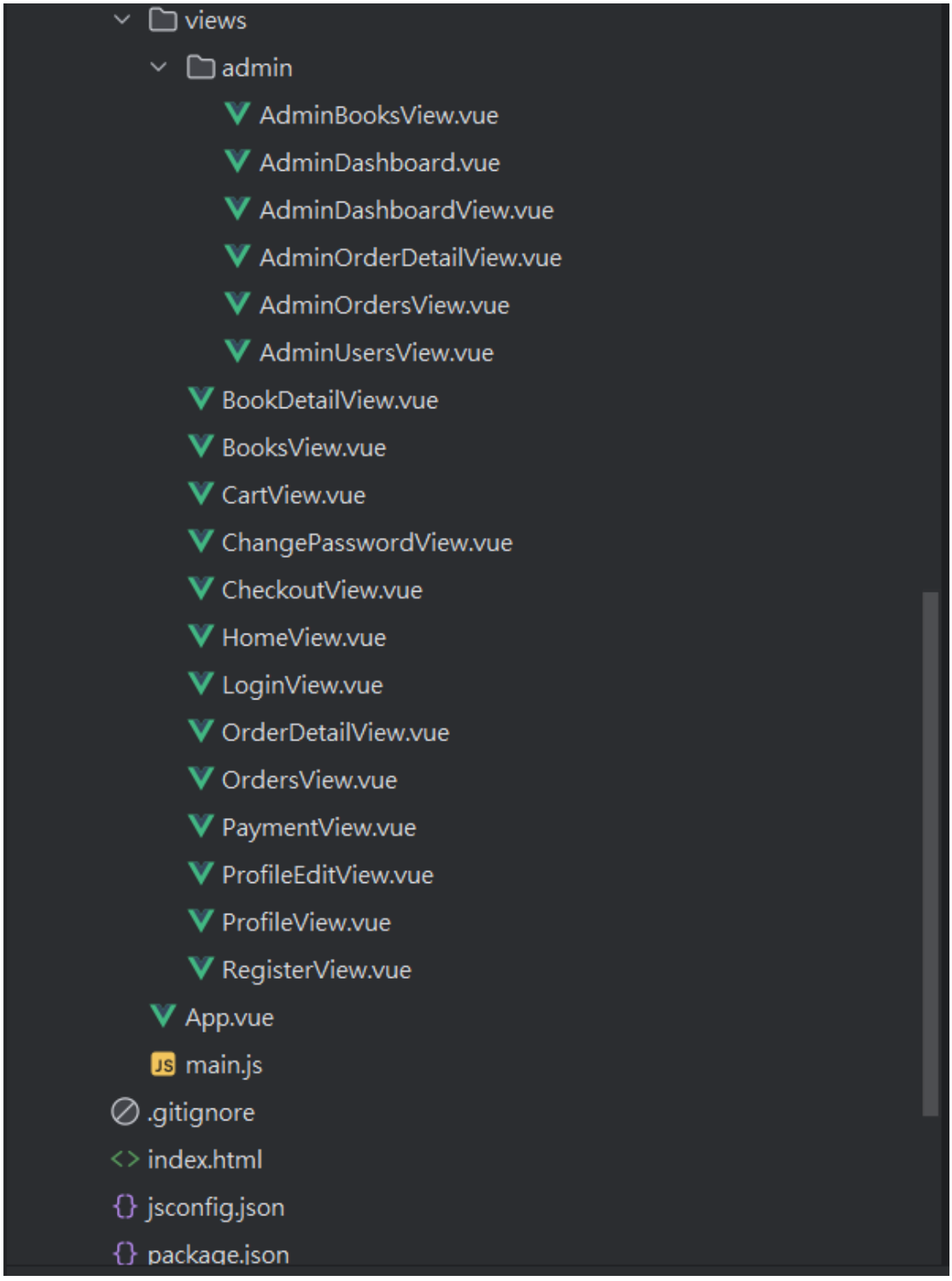

src
main
java
org.example
config
AdminInitializer
ApplicationConfig
GlobalExceptionHandler
JwtAuthenticationFilter
SecurityConfig
controller
AdminBookController
AdminOrderController
AdminUserController
AuthController
BookController
CartController
OrderController
ReviewController
UserController
dto
AuthenticationRequest
AuthenticationResponse
BookRequest
CartItemRequest
ChangePasswordRequest
CheckoutRequest
CheckoutResponseDTO


CheckoutResponseDTO
LoginRequest
LoginResponse
OrderDetailDTO
OrderItemDTO
OrderListDTO
OrderStatusUpdateRequest
OrderSummaryDTO
ProfileDto
RegisterRequest
ReviewResponse
UserProfileResponse
UserResponse
entity
Book
CartItem
Language
Order
OrderItem
OrderStatus
PaymentMethod
Review
Role
User
exception
OrderNotFoundException
repository
BookRepository
CartItemRepository
OrderItemRepository
OrderRepository


exception
OrderNotFoundException
repository
BookRepository
CartItemRepository
OrderItemRepository
OrderRepository
ReviewRepository
UserRepository
security
JwtAuthFilter
service
AuthService
BookService
CartService
JwtService
OrderService
ReviewService
UserService
ForeignLanguagesBookApplication
Main
resources
application.properties
application.yml.bak
application-test.properties
test
target

前端的部分：


src
assets
components
AdminBookModal.vue
Footer.vue
layouts
plugins
axios.js
router
index.js
services
adminBookService.js
adminOrderService.js
adminUserService.js
apiClient.js
authService.js
bookService.js
cartService.js
orderService.js
reviewService.js
stores
auth.js
cart.js
test
views


views
admin
AdminBooksView.vue
AdminDashboard.vue
AdminDashboardView.vue
AdminOrderDetailView.vue
AdminOrdersView.vue
AdminUsersView.vue
BookDetailView.vue
BooksView.vue
CartView.vue
ChangePasswordView.vue
CheckoutView.vue
HomeView.vue
LoginView.vue
OrderDetailView.vue
OrdersView.vue
PaymentView.vue
ProfileEditView.vue
ProfileView.vue
RegisterView.vue
App.vue
main.js
.gitignore
index.html
jsconfig.json
package.json

■ 第十四部分：部署實務、專案總結與未來展望 (Deployment & Conclusion)

最後一部分總結專案成果，並說明如何透過雲端技術優化部署流程，展現具備研究價值的技術延展性。

1. 生產環境部署優化 (Deployment & DevOps)

本專案成功部署於生產環境，並針對 Web 伺服器特性進行優化：

- **Docker 容器化**：透過 Docker 封裝 Java 執行環境與應用程式，徹底解決跨環境（Development vs Production）的依賴衝突問題。

```java
@Bean  Chris Chou
public CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration configuration = new CorsConfiguration();

    // 🚨 修正 1: 允許多個來源（本地測試 + Vercel 雲端）
    configuration.setAllowedOrigins(Arrays.asList(
            "http://localhost:5173", // Vite 預設
            "http://localhost:5174", // 你的本地開發埠
            "https://foreign-languages-book.vercel.app", // ✨ 填入你剛才產出的 Vercel 網址
            "https://foreign-languages-book-git-master-chrischou19990904s-projects.vercel.app" // 建議也加上這個預覽網址
    ));

    // 🚨 修正 2: 允許的方法
    configuration.setAllowedMethods(Arrays.asList("GET", "POST", "PUT", "DELETE", "OPTIONS", "PATCH"));

    // 🚨 修正 3: 允許的 Header
    configuration.setAllowedHeaders(Arrays.asList("Authorization", "Content-Type", "Accept", "Origin", "X-Requested-With"));

    // 🚨 修正 4: 很重要！你原本的 code 最後一行又把 AllowCredentials 設為 false，會蓋掉前面的設定
    // 如果前端 Axios 有設定 withCredentials: true，這裡就必須是 true
    configuration.setAllowCredentials(true);

    UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration( pattern: "/**", configuration);
    return source;
}
```

- **SPA 路由修復**：針對 Vercel 部署環境，撰寫 `vercel.json` 進行 URL Rewrite，解決單頁應用（Single Page Application）在重新整理時產生的 404 路由失效問題。

ce.js | cart.js | cartService.js | orderService.js | vite.config.js | vercel.json

```json
{
  "rewrites": [
    {
      "source": "/(.*)",
      "destination": "/index.html"
    }
  ]
}
```

2. 專案總結 (Final Summary)

本專案獨立完成了具備 14 種語系檢索能力的全端電商平台，實作了包含 JWT 安全體系、ACID 事務性管理與 DTO 解耦設計。這不僅是一個功能完整的作品，更是對**軟體架構、資安防護與效能優化**的深度實踐，證明本人具備資工研究所要求的工程思維與實作硬實力。

3. 未來研究方向 (Future Work)

進入海大資工系後，本專案預計朝以下方向演進：

- **分散式快取優化**：導入 Redis 實作 JWT 黑名單機制，解決 Token 撤銷的即時性問題。
- **搜尋引擎升級**：規劃導入 Elasticsearch 取代傳統資料庫模糊搜尋，提升大數據量下的檢索效能與精確度。