# **电商购物平台 - 软件系统设计 (SDD)**

## 

## **1. 设计概述**

本文档基于已设计的 《电商购物平台 - 软件需求规格说明书 (SRS)》 进行系统设计。设计目标是利用确定的技术栈（Spring Boot, MyBatis, MySQL, Redis, Vue3, Nginx 等）构建一个满足所有需求的、高性能、高可用、易维护的电商平台。

本文档将详细阐述系统架构、模块划分、数据库设计、核心流程、API 规范、缓存策略、安全考虑以及部署方案，并包含使用 PlantUML 绘制的关键设计图。

## **2. 架构设计 (Architecture Design)**

### **2.1 核心架构模式**

采用 **分层架构 (Layered Architecture)**，严格划分职责边界：

* **表现层 (Presentation Layer):** 基于 Spring MVC (@RestController)，负责定义和暴露 RESTful API 接口，处理 HTTP 请求/响应，进行输入参数校验（使用 Spring Validation），调用业务逻辑层。**不处理业务逻辑。**
* **业务逻辑层 (Business Logic Layer):** 基于 Spring Service (@Service)，封装核心业务规则、流程和状态管理。协调数据访问层和缓存层，处理事务（@Transactional）。
* **数据访问层 (Data Access Layer):** 基于 **MyBatis** 框架，负责与 **MySQL** 数据库进行交互。包含 Mapper 接口和对应的 XML 映射文件（或注解），执行 SQL 语句。
* **领域模型层 (Domain Model Layer):** 定义核心业务实体（POJO/Entity），以及数据传输对象 (DTO)，用于在各层之间传递数据。
* **缓存层 (Caching Layer):** 使用 **Redis** 作为分布式缓存，由业务逻辑层调用（通过 RedisTemplate 或 Spring Cache Abstraction），用于缓存热点数据，提升性能。

结合 **RESTful API + SPA (Single Page Application)** 架构：后端提供无状态的 RESTful API，前端（Vue3）负责用户界面的渲染和交互逻辑，通过 API 与后端通信。

**设计理由:** 分层架构清晰、易于维护和扩展。RESTful + SPA 是现代 Web 应用的流行模式，有利于前后端分离开发和部署。

### **2.2 技术栈**

* **后端:** Java (JDK 17+), Spring Boot 3.x, Spring MVC, Spring Security, Spring Validation, **MyBatis** 3.x (整合 Spring 使用 mybatis-spring-boot-starter)
* **数据库:** **MySQL** 8.x
* **前端:** **Vue3** (使用 **TypeScript**), Vite (构建工具), Axios (HTTP client)
* **Web 服务器/反向代理:** **Nginx**
* **构建工具:** **Maven** 3.x
* **缓存:** **Redis** 6.x+ (使用 spring-boot-starter-data-redis)
* **应用服务器:** Spring Boot 内嵌 **Tomcat** 10+
* **API 文档:** Swagger 3 / OpenAPI 3 (使用 springdoc-openapi-starter-webmvc-ui)
* **数据库版本管理:** **Flyway** / Liquibase (推荐)
* **日志:** SLF4j + Logback
* **监控:** Prometheus (Metrics), Grafana (Dashboard), (可选) SkyWalking/Zipkin (Tracing)

### **2.3 高层交互流程**

1. 用户浏览器加载由 Nginx 提供的 Vue3 应用静态文件 (HTML, JS, CSS)。
2. Vue3 应用渲染界面，需要数据时通过 Axios 发起 API 请求 (HTTPS) 到 Nginx。
3. Nginx 将 API 请求 (/api/…) 反向代理 (HTTP) 到后端 Spring Boot 应用集群 (运行在内嵌 Tomcat 上)。
4. Spring Security 拦截请求进行认证 (JWT 校验) 和授权。
5. 请求到达 @RestController，进行参数绑定和 DTO 校验。
6. Controller 调用相应的 @Service 方法。
7. Service 执行业务逻辑：
   * 可能先查询 Redis 缓存 (RedisTemplate 或 @Cacheable)。
   * 若缓存未命中或需写操作，则调用 MyBatis Mapper 接口。
   * Mapper 通过 XML/注解 映射执行 MySQL 数据库操作。
   * Service 处理事务（@Transactional）。
   * (可选) Service 更新或清除 Redis 缓存。
8. 结果逐层返回给 Controller。
9. Controller 组装成标准的 JSON 响应体。
10. (若发生异常) 全局异常处理器 (@RestControllerAdvice) 捕获异常，返回统一错误 JSON。
11. 响应通过 Nginx 返回给前端 Vue3 应用。
12. Vue3 应用更新界面展示数据。

## **3. 模块设计 (Module Design)**

系统核心后端模块划分如下（对应 Maven 模块或顶级包）：

* ecommerce-user-service: 用户模块 (注册、登录、认证、授权、信息管理、地址管理)。
* ecommerce-product-service: 商品模块 (分类、商品信息、库存接口、搜索接口)。
* ecommerce-cart-service: 购物车模块 (购物车状态管理 - 采用 Redis 存储)。
* ecommerce-order-service: 订单模块 (订单创建、查询、状态流转)。
* ecommerce-payment-service: 支付模块 (模拟支付逻辑、预留接口)。
* ecommerce-admin-api: 后台管理 API 模块 (提供给后台管理前端的接口)。
* ecommerce-common: 公共模块 (工具类、常量、枚举、全局异常处理、统一响应体、MyBatis/Redis 配置基类、DTO 基类)。
* ecommerce-domain: 领域模型模块 (包含所有业务实体 POJO/Entity 和 DTO)。

**(前端 Vue3 项目是独立的代码库)**

**设计理由:** 按业务领域划分模块，职责清晰，便于团队分工和独立部署（未来微服务演进的基础）。common 和 domain 模块提供共享能力和模型。

## **4. 详细设计 (Detailed Design)**

### **4.1 类图设计 (Class Diagram Description - MyBatis Focus)**

**(核心实体类 Domain POJOs - 位于 ecommerce-domain 模块)**

* **User:**
  + 属性: id (Long, PK), username (String, unique), passwordHash (String), email (String, unique), phoneNumber (String, unique, nullable), status (UserStatus enum: ACTIVE, INACTIVE), createdAt (LocalDateTime), updatedAt (LocalDateTime)
  + 关系: 1 User <–> \* Address (OneToMany); 1 User <–> \* Order (OneToMany); 1 User <–> 1 ShoppingCart (OneToOne - 逻辑关联)
* **Address:**
  + 属性: id (Long, PK), userId (Long, FK), receiverName (String), phoneNumber (String), province (String), city (String), district (String), detailedAddress (String), isDefault (Boolean), createdAt, updatedAt
  + 关系: \* Address <–> 1 User (ManyToOne)
* **Product:**
  + 属性: id (Long, PK), name (String), description (String/TEXT), categoryId (Long, FK), price (BigDecimal), stockQuantity (Integer), imageUrl (String), status (ProductStatus enum: ON\_SALE, OFF\_SALE, DELETED), createdAt, updatedAt
  + 关系: \* Product <–> 1 Category (ManyToOne); 1 Product <–> \* OrderItem (OneToMany - 关联订单项快照)
  + **核心方法 (Service 层体现):** decreaseStock(quantity), increaseStock(quantity)
* **Category:**
  + 属性: id (Long, PK), name (String), parentId (Long, nullable - 支持层级), sortOrder (Integer), createdAt, updatedAt
  + 关系: 1 Category <–> \* Product (OneToMany); 1 Category <–> \* Category (Self-referencing OneToMany/ManyToOne for hierarchy)
* **Order:**
  + 属性: id (Long, PK), orderNo (String, unique), userId (Long, FK), totalAmount (BigDecimal), payableAmount (BigDecimal), status (OrderStatus enum: PENDING\_PAYMENT, PAID, …), receiverName (String), receiverPhone (String), receiverAddress (String), paymentMethod (String), paidAt (LocalDateTime, nullable), shippedAt (LocalDateTime, nullable), completedAt (LocalDateTime, nullable), createdAt, updatedAt
  + 关系: 1 Order <–> 1 User (ManyToOne); 1 Order <–> \* OrderItem (OneToMany, Composition - Order ‘owns’ OrderItems)
  + **核心方法 (Service 层体现):** cancel(), markAsPaid(), ship()
* **OrderItem:**
  + 属性: id (Long, PK), orderId (Long, FK), productId (Long, FK - 指向当时的 Product), productName (String - 快照), productImageUrl (String - 快照), unitPrice (BigDecimal - 快照), quantity (Integer), totalPrice (BigDecimal)
  + 关系: \* OrderItem <–> 1 Order (ManyToOne); 1 OrderItem --> 1 Product (ManyToOne - 指向原始商品)
* **ShoppingCart (逻辑概念或 Redis 结构):**
  + (若数据库存储) 属性: id (Long, PK), userId (Long, Unique, FK)
  + (若 Redis Hash: cart:{userId}) Field: productId, Value: quantity
  + 关系: 1 ShoppingCart <–> \* CartItem (OneToMany / Hash entries)
* **CartItem (逻辑概念或 Redis 结构的一部分):**
  + (若数据库存储) 属性: id (Long, PK), cartId (Long, FK), productId (Long, FK), quantity (Integer), addedAt (LocalDateTime)
  + 关系: \* CartItem <–> 1 ShoppingCart (ManyToOne); 1 CartItem --> 1 Product (ManyToOne)

**(数据访问层接口 Mappers - 位于各业务模块的 mapper 包)**

* 使用 MyBatis 的 Mapper 接口，通常与实体类对应。
* **UserMapper (interface):**
  + 方法: @Insert(…) insert(User user), @Select(…) findById(Long id), @Select(…) findByUsername(String username), @Select(…) findByEmail(String email), @Update(…) update(User user)… (SQL 语句在 XML 或注解中定义)。
* **ProductMapper (interface):**
  + 方法: @SelectProvider(…) / @Select(…) listByCategory(Map<String, Object> params), @SelectProvider(…) / @Select(…) search(Map<String, Object> params), @Select(…) findById(Long id), @Options(useGeneratedKeys=true, keyProperty=“id”) @Insert(…) insert(Product product), @Update(…) updateStock(@Param(“id”) Long id, @Param(“quantity”) Integer quantity), @Update(“UPDATE t\_product SET stock\_quantity = stock\_quantity - #{quantity} WHERE id = #{id} AND stock\_quantity >= #{quantity}”) int decreaseStock(@Param(“id”) Long id, @Param(“quantity”) Integer quantity)…
* **OrderMapper (interface):**
  + 方法: @Options(useGeneratedKeys=true, keyProperty=“id”) @Insert(…) insert(Order order), @Select(…) findById(Long id), @SelectProvider(…) / @Select(…) listByUserId(Map<String, Object> params), @Select(…) findByOrderNo(String orderNo), @Update(…) updateStatus(Order order)…
* **OrderItemMapper (interface):**
  + 方法: @Insert(“<script>INSERT INTO t\_order\_item (…) VALUES <foreach collection=‘list’ item=‘item’ separator=‘,’>(#{item.orderId}, …)</foreach></script>”) insertBatch(List<OrderItem> items), @Select(…) findByOrderId(Long orderId)…
* **(类似地定义 CategoryMapper, AddressMapper 等)**

**(XML 映射文件 - 位于 resources/mapper/ 目录下)**

* 与 Mapper 接口对应的 XML 文件（如 UserMapper.xml, ProductMapper.xml）。
* 包含 <mapper namespace=“…”>, <resultMap>, <sql>, <select>, <insert>, <update>, <delete> 等标签，定义 SQL 语句、结果映射、动态 SQL (<if>, <foreach> 等)。

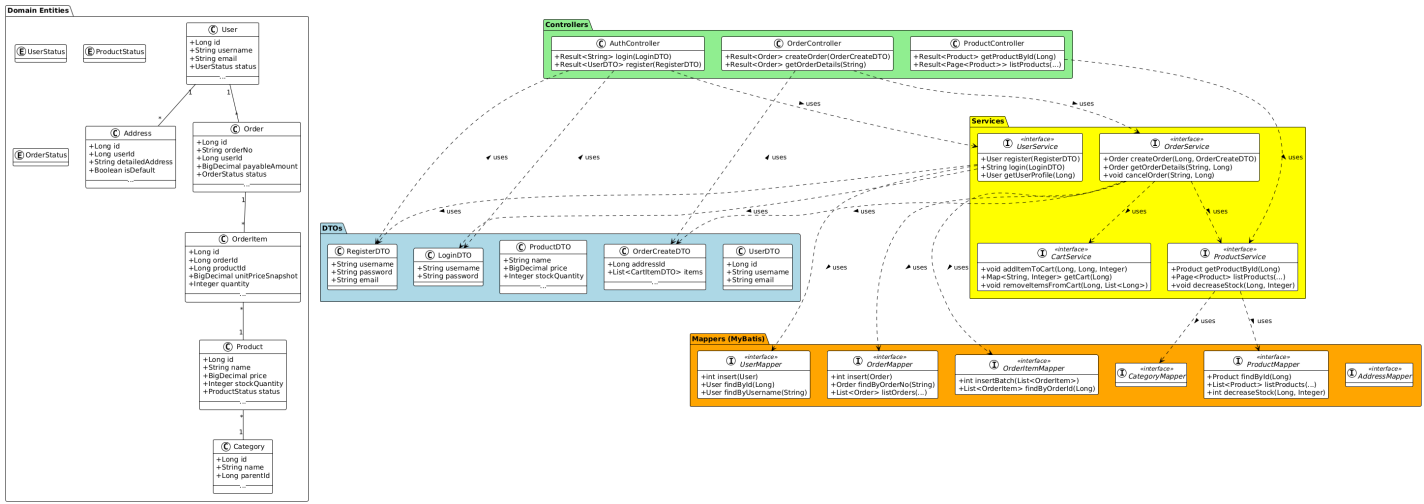
**(业务逻辑层接口与实现 Services - 位于各业务模块的 service 包)**

* 接口 (UserService, ProductService 等) 定义业务方法。
* 实现类 (UserServiceImpl, ProductServiceImpl 等) 使用 @Service 注解，注入对应的 Mapper 接口和可能的其他 Service。
* 实现类负责业务逻辑、事务管理 (@Transactional)、调用 Mapper 执行数据库操作、与 Redis 缓存交互。
* **示例: ProductServiceImpl**  
  @Service  
  public class ProductServiceImpl implements ProductService {  
  @Autowired private ProductMapper productMapper;  
  @Autowired private RedisTemplate<String, Object> redisTemplate;  
  private static final String CACHE\_KEY\_PREFIX = “product:”;
* @Override   
  public Product getProductById(Long productId) {   
   String cacheKey = CACHE\_KEY\_PREFIX + productId;   
   // 1. 尝试从 Redis 获取   
   Product product = (Product) redisTemplate.opsForValue().get(cacheKey);   
   if (product != null) {   
   return product; // 缓存命中   
   }   
   // 2. 缓存未命中，查询数据库   
   product = productMapper.findById(productId);   
   if (product != null) {   
   // 3. 查到数据，写入 Redis (设置 10-30 分钟随机过期时间防雪崩)   
   long expiration = 600 + new Random().nextInt(1200); // 10-30 minutes   
   redisTemplate.opsForValue().set(cacheKey, product, expiration, TimeUnit.SECONDS);   
   } else {   
   // 可选：缓存空对象防穿透，设置较短过期时间   
   // redisTemplate.opsForValue().set(cacheKey, null, 5, TimeUnit.MINUTES);   
   }   
   return product;   
  }  
    
  @Override   
  @Transactional // 确保数据库操作原子性   
  public void decreaseStock(Long productId, Integer quantity) throws InsufficientStockException {   
   // 1. 调用 Mapper 尝试扣减库存 (包含 WHERE stock >= quantity)   
   int affectedRows = productMapper.decreaseStock(productId, quantity);   
   if (affectedRows == 0) {   
   // 2. 影响行数为 0，表示库存不足或商品不存在   
   throw new InsufficientStockException("商品 " + productId + " 库存不足");   
   }   
   // 3. 成功扣减库存后，清除缓存 (Cache Aside Pattern)   
   redisTemplate.delete(CACHE\_KEY\_PREFIX + productId);   
  }   
  // ... 其他方法
* }

**(表现层控制器 Controllers - 位于各业务模块的 controller 包)**

* 使用 @RestController, @RequestMapping, @GetMapping, @PostMapping, @PutMapping, @DeleteMapping, @PathVariable, @RequestBody, @RequestParam 等注解。
* 使用 @Validated 注解配合 DTO 中的 JSR-303/JSR-380 (Bean Validation) 注解进行参数校验。
* 严格面向 API: 返回统一的 JSON 响应体 (使用 common 模块定义的 Result<T> 类)。使用 DTO (Data Transfer Object) 进行参数传递和结果返回，避免直接暴露领域实体。
* 集成 OpenAPI: 使用 @Operation, @Parameter, @ApiResponse 等 springdoc-openapi 注解来自动生成 API 文档。

**UML 类图:**

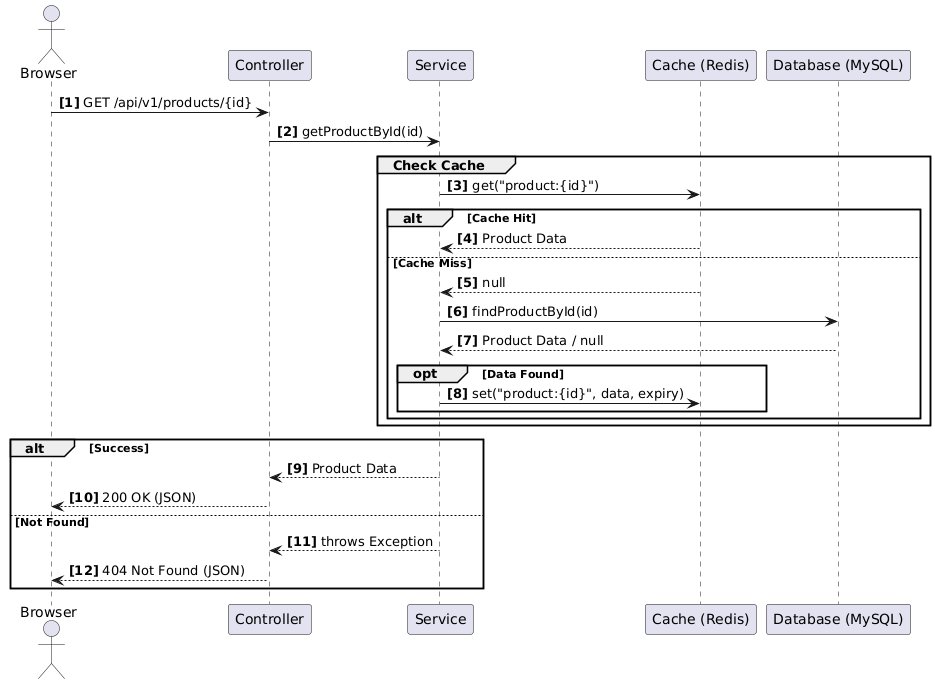
[](plantuml_diagrams/diagram_01_类图.png)

### **4.2 顺序图设计 (Sequence Diagram Description - 含 Redis)**

#### **场景1: 获取商品详情 (UC-12)**

* **参与者 (Lifelines):** User (Browser) -> Nginx -> ProductController -> ProductService -> RedisTemplate -> Redis Server -> ProductMapper -> MySQL Database
* **流程描述:**
  1. Browser 发送 GET /api/v1/products/{id} 请求到 Nginx。
  2. Nginx 代理请求到 ProductController。
  3. ProductController 调用 ProductService.getProductById(id)。
  4. ProductService 构建缓存 Key (product:{id})。
  5. ProductService 调用 RedisTemplate.opsForValue().get(key)。
  6. RedisTemplate 发送 GET key 命令到 Redis Server。
  7. **[alt: 缓存命中]**
     + Redis Server 返回缓存的 Product 对象数据。
     + RedisTemplate 返回反序列化的对象给 ProductService。
     + ProductService 直接返回 Product 对象给 ProductController。
  8. **[else: 缓存未命中]**
     + Redis Server 返回 nil。
     + RedisTemplate 返回 null 给 ProductService。
     + ProductService 调用 ProductMapper.findById(id)。
     + ProductMapper 执行 SELECT SQL 到 MySQL Database。
     + MySQL Database 返回查询结果。
     + ProductMapper 映射结果为 Product 对象并返回给 ProductService。
     + **[opt: 数据库查到数据]**
       - ProductService 调用 RedisTemplate.opsForValue().set(key, product, expirationTime) (带随机过期时间)。
       - RedisTemplate 发送 SET key value EX timeout 命令到 Redis Server。
       - Redis Server 存储数据并返回成功。
       - ProductService 返回 Product 对象给 ProductController。
     + **[else: 数据库未查到数据]**
       - ProductService 抛出 ResourceNotFoundException。
       - (全局异常处理器处理该异常，返回 404 JSON 响应)
  9. (若成功) ProductController 将 Product 对象包装成标准 JSON 响应体 (Result.success(product)) 返回。

**UML 顺序图 (获取商品详情):**

[](plantuml_diagrams/diagram_02_顺序图%20(获取商品详情).png)

#### **场景2: 提交订单 (UC-20 - 涉及缓存清除)**

* **参与者:** User (Browser) -> Nginx -> OrderController -> OrderService -> AddressMapper -> ProductService -> ProductMapper -> OrderMapper -> OrderItemMapper -> RedisTemplate (用于 Product Cache) -> CartService -> RedisTemplate (用于 Cart Cache) -> MySQL Database -> Redis Server
* **流程描述 (增加缓存相关步骤):**
  1. Browser 发送 POST /api/v1/orders 请求 (含 DTO) 到 Nginx。
  2. Nginx 代理请求到 OrderController。
  3. OrderController 获取 userId 及请求体验证后的 OrderCreateDTO。
  4. OrderController 调用 OrderService.createOrder(userId, dto)。
  5. **[Transaction Starts]** OrderService 开始数据库事务。
  6. OrderService 调用 AddressMapper.findByIdAndUserId(addressId, userId) 验证地址。
  7. OrderService 遍历 dto.items：
     + **[loop: for each item]**
       - OrderService 调用 ProductService.decreaseStock(productId, quantity)。(此方法内部包含数据库扣减库存、失败时抛异常、成功时清除产品缓存)
       - ProductService.decreaseStock 内部:
         * 调用 ProductMapper.decreaseStock(productId, quantity) (包含 WHERE stock >= quantity)。
         * 若影响行数为 0，抛 InsufficientStockException。
         * 若成功，调用 RedisTemplate.delete(productKey) 清除该商品缓存。
  8. OrderService 计算总金额。
  9. OrderService 创建 Order 实体。
  10. OrderService 调用 OrderMapper.insert(order) 获取 orderId。
  11. OrderService 遍历 dto.items 创建 OrderItem 列表。
  12. OrderService 调用 OrderItemMapper.insertBatch(orderItems)。
  13. OrderService 调用 CartService.removeItemsFromCart(userId, orderedProductIds)。
  14. CartService.removeItemsFromCart 内部: (假设购物车存于 Redis Hash cart:{userId})
      + 调用 RedisTemplate.opsForHash().delete(cartKey, itemHashKeys…) 删除对应条目。
  15. **[Transaction Commits]** 若所有步骤成功，提交事务。
  16. OrderService 返回成功创建的订单信息 (如 orderId, orderNo) 给 OrderController。
  17. OrderController 返回 201 Created JSON 响应给 User (Browser)。
  18. **[Exception Handling]** 若任何步骤失败 (包括 InsufficientStockException)，事务自动回滚，全局异常处理器捕获异常并返回相应错误 JSON。

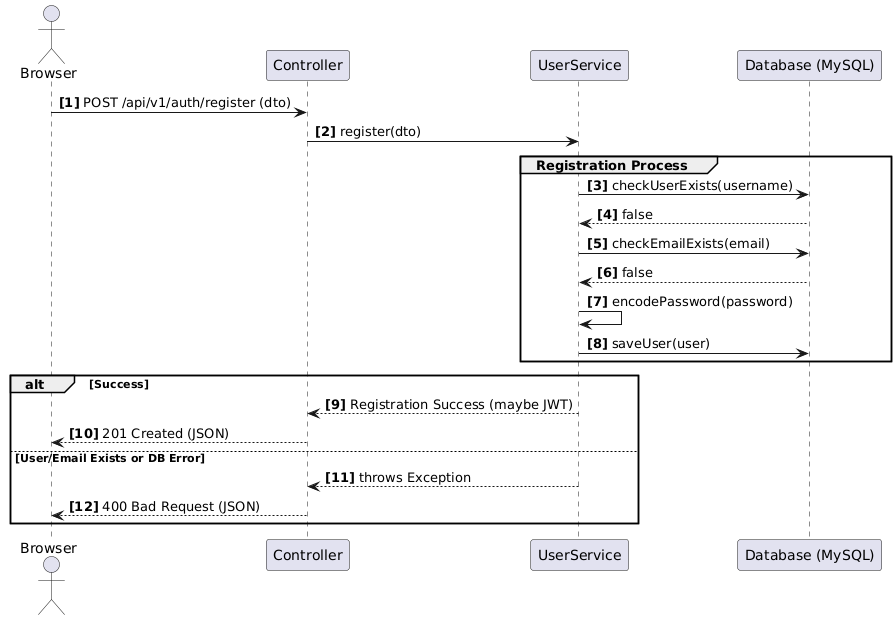
**UML 顺序图 (提交订单):**

#### **[D:/ProjectFile/Shopping-Gemini/plantuml_diagrams/diagram_03_ 顺序图 (提交订单).pngdiagram_03_ 顺序图 (提交订单)](plantuml_diagrams/diagram_03_%20顺序图%20(提交订单).png)**

#### **场景3: 用户注册 (UC-01)**

* **参与者:** User (Browser) -> Nginx -> AuthController -> UserService -> PasswordEncoder -> UserMapper -> MySQL Database
* **流程描述:**
  1. Browser 发送 POST /api/v1/auth/register 请求 (含 RegisterDTO) 到 Nginx。
  2. Nginx 代理请求到 AuthController。
  3. AuthController 调用 UserService.register(dto)。
  4. UserService 调用 UserMapper.findByUsername() 和 findByEmail() 检查是否存在。
  5. (假设不存在) UserService 调用 PasswordEncoder.encode(rawPassword) 获取哈希密码。
  6. UserService 创建 User 实体对象。
  7. UserService 调用 UserMapper.insert(user)。
  8. UserMapper 生成 INSERT SQL 语句并发送给 MySQL Database。
  9. MySQL Database 执行插入，返回成功。
  10. UserMapper 返回生成的 userId (如果配置了)。
  11. UserService 处理自动登录逻辑/准备成功响应。
  12. UserService 返回成功响应给 AuthController。
  13. AuthController 返回 201 Created JSON 响应给 Nginx。
  14. Nginx 返回响应给 User (Browser)。

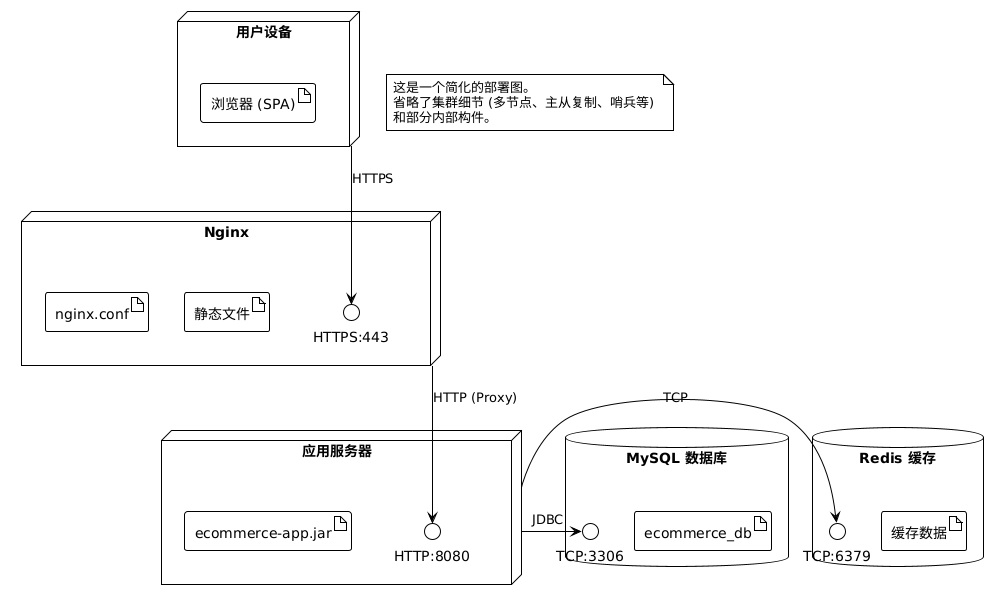
**UML 顺序图 (用户注册):**

[](plantuml_diagrams/diagram_04_顺序图%20(用户注册).png)

### **4.3 部署图设计 (Deployment Diagram Description - 最终版)**

* **节点 (Nodes - 物理/虚拟服务器):**
  + **用户设备 (User Device):** 运行 Web 浏览器 (Vue3 SPA)。
  + **Nginx 服务器 (Nginx Server):** 运行 Nginx 服务。职责：HTTPS 终止、静态文件服务、API 反向代理、负载均衡。
  + **应用服务器集群 (App Server Cluster):** 运行 Java 运行时 (JRE 17+), Spring Boot 应用 (内嵌 Tomcat)。至少两个节点实现高可用。
  + **MySQL 数据库服务器 (MySQL Server):** 运行 MySQL 服务。建议主从复制 (Master-Slave) 配置。
  + **Redis 服务器 (Redis Server):** 运行 Redis 服务。建议哨兵模式 (Sentinel) 或集群模式 (Cluster) 实现高可用。
* **部署构件 (Artifacts):**
  + Nginx: nginx.conf, Vue3 构建后的静态文件 (index.html, assets/)。
  + App Server: ecommerce-app.jar (Spring Boot Fat JAR)。
  + MySQL Server: ecommerce\_db 数据库实例。
  + Redis Server: Redis 实例。
* **通信路径:**
  + User Device <–HTTPS (443)–> Nginx Server
  + Nginx Server <–HTTP (80/8080)–> App Server Cluster (负载均衡)
  + App Server Cluster <–JDBC (TCP 3306)–> MySQL Server (主库写，从库读)
  + App Server Cluster <–TCP (6379)–> Redis Server

**UML 部署图:**



### **4.4 对象图设计 (Object Diagram Description)**

* **目的:** 展示系统在特定时间点上一组对象实例及其关系的静态快照，用于理解复杂交互或调试。
* **示例场景:** 用户查看待支付订单详情
* **对象实例 (格式: objectName:ClassName {attribute=value, …}):**
  + u101:User { id=101, username=‘john\_doe’ }
  + order567:Order { id=567, orderNo=‘20250414112400123’, status=PENDING\_PAYMENT, payableAmount=159.80, userId=101 }
  + item1:OrderItem { id=1001, quantity=1, unitPrice=99.90, productName=‘蓝牙耳机 V5.0’, orderId=567, productId=78 }
  + item2:OrderItem { id=1002, quantity=2, unitPrice=29.95, productName=‘USB-C 充电线 1m’, orderId=567, productId=90 }
  + p78:Product { id=78, name=‘蓝牙耳机 V5.0’ }
  + p90:Product { id=90, name=‘USB-C 充电线 1m’ }
* **链接 (Links - 表示对象间的关系实例):**
  + u101 -— (下单用户) -—> order567
  + order567 -— (包含订单项) -—> item1
  + order567 -— (包含订单项) -—> item2
  + item1 -— (对应商品快照) -—> p78
  + item2 -— (对应商品快照) -—> p90
* **(PlantUML 没有专门的对象图语法，通常用类图标注实例或使用其他工具绘制)**

## **5. API 设计 (API Design)**

### **5.1 规范**

* 严格遵循 **RESTful** 设计原则：
  + 使用 **名词复数** 表示资源 (e.g., /users, /products, /orders)。
  + 使用 **HTTP 方法** 表达操作 (GET-查询, POST-创建, PUT-更新, DELETE-删除)。
  + 使用 **HTTP 状态码** 表示结果 (200 OK, 201 Created, 204 No Content, 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 500 Internal Server Error)。
  + 无状态：每个请求包含所有必要信息，服务端不保存客户端状态。
* 使用 **OpenAPI 3** 规范进行定义。

### **5.2 文档**

* 使用 springdoc-openapi 库，结合 Controller 和 DTO 中的注解 (@Operation, @Schema, @Parameter, @ApiResponse 等)，自动生成交互式 API 文档。
* 文档访问地址：/swagger-ui.html 或 /v3/api-docs。
* API 文档作为前后端开发的**契约 (Contract)**。

### **5.3 认证与授权**

* 所有需要登录才能访问的 API 均通过 **Spring Security + JWT** 进行保护。
* 客户端在请求的 Authorization 头中携带 Bearer <token>。
* 登录接口 (/api/v1/auth/login) 验证用户名密码，成功后返回 JWT。
* 使用 Spring Security 的方法级安全注解 (@PreAuthorize) 或 SecurityFilterChain 配置进行细粒度的权限控制。

### **5.4 数据格式**

* 请求体和响应体统一使用 application/json;charset=UTF-8 格式。
* 日期时间格式统一使用 **ISO 8601** 标准字符串（例如：2025-04-14T15:10:00.123+08:00 或 2025-04-14T07:10:00.123Z）。

### **5.5 版本管理**

* API 通过 **URL 路径** 进行版本控制，当前版本为 /api/v1/…。

### **5.6 统一响应体**

* 所有 API 返回统一结构的 JSON 对象 (定义在 ecommerce-common 模块)：  
  public class Result<T> {  
  private int code; // 业务状态码  
  private String message; // 描述信息  
  private T data; // 业务数据
* // 构造函数, 静态成功/失败方法等...   
  public static <T> Result<T> success(T data) { ... }   
  public static Result<Void> fail(int code, String message) { ... }   
  public static Result<Void> fail(ErrorCode errorCode) { ... }
* }
* **业务状态码 (code):**
  + 0: 表示成功。
  + 非零: 表示特定业务错误（定义全局错误码枚举 ErrorCode，如 4001-用户名已存在, 4002-库存不足, 4010-未认证, 4030-无权限, 5000-服务器内部错误）。
* **全局异常处理:** 使用 @RestControllerAdvice 统一捕获业务异常 (BusinessException) 和其他运行时异常，转换为 Result 对象返回。

### **5.7 分页**

* 对于列表查询 API (如获取商品列表、订单列表)，采用标准的分页参数：
  + 请求参数: page (页码, 从 1 开始), pageSize (每页数量)。
  + 响应数据 (data 字段): 包含列表数据 list 和总记录数 total 的分页对象 (e.g., PageResult<T>)。

## **6. 缓存策略 (Caching Strategy - Redis)**

### **6.1 目标**

通过缓存常用且非实时性要求极高的数据，减少数据库访问压力，提高系统响应速度。

### **6.2 应用场景与策略**

* **商品数据:**
  + **商品详情 (Product):**
    - Key: product:{productId}
    - Value: Product 对象 (JSON 序列化)
    - 策略: **Cache Aside Pattern** (读：查缓存 -> 查库 -> 写缓存；写：更新库 -> **删除缓存**)。
    - 过期时间: 10-30 分钟 (随机值防雪崩)。
    - **理由:** 商品详情读多写少，适合缓存。Cache Aside 实现简单，删除缓存能较好保证一致性。
  + **商品分类 (List):**
    - Key: category:tree 或 category:all
    - Value: Category 列表/树结构 (JSON 序列化)
    - 策略: 全量缓存。后台更新分类后**主动清除**缓存。
    - 过期时间: 较长（如 1 小时）或不过期，依赖手动清除。
    - **理由:** 分类信息变动频率低，访问频繁，适合全量缓存。
* **购物车 (User Cart):**
  + 存储结构: **Redis Hash**
  + Key: cart:{userId}
  + Field: productId
  + Value: quantity (Integer)
  + 策略: 用户加减购物车、清空购物车等操作**直接读写 Redis Hash**。
  + 过期时间: 7 天 (自动清理不活跃购物车)。
  + **理由:** 购物车操作频繁，状态易变，Redis Hash 结构适合存储 K-V 对，且操作原子性好。
* **用户认证/会话:**
  + **JWT 黑名单:** (如果需要强制下线或修改密码后立即使旧 Token 失效)
    - Key: jwt:blacklist:{jti} 或 jwt:blacklist:user:{userId} (存储需要失效的 token 或 user 维度的失效时间戳)
    - Value: 任意值或失效时间戳
    - 策略: 用户登出或修改密码时，将旧的、未过期的 JWT 的 jti 或用户 ID 关联信息加入 Redis Set/String，并设置与 JWT 剩余有效期相同的过期时间。API 网关或 Spring Security 过滤器在验证 JWT 时，先检查其是否在黑名单中。
  + **(替代方案) Session 存储:** 若选择 Session 方案而非 JWT，可使用 spring-session-data-redis 将 Session 数据存储在 Redis 中，实现分布式 Session 共享。
* **热点配置/字典数据:**
  + Key: config:{key} 或 dict:{type}
  + Value: 配置值或字典列表 (JSON 序列化)
  + 策略: 启动时加载或首次访问时加载，后台修改配置后主动更新缓存。

### **6.3 实现**

* **主要方式:** 使用 Spring Boot 提供的 RedisTemplate<String, Object> 进行显式编程，对缓存的 Key、Value 格式、序列化方式（推荐 Jackson2JsonRedisSerializer）、过期时间有完全控制。
* **辅助方式:** 对不需复杂控制的只读缓存场景，可使用 Spring Cache Abstraction (@Cacheable, @CachePut, @CacheEvict 等)，需配置 RedisCacheManager 并指定序列化器。

### **6.4 注意事项**

* **缓存穿透:** 查询不存在的数据导致每次都打到数据库。
  + **解决方案:** 缓存空对象（设置较短过期时间）；布隆过滤器 (Bloom Filter) 提前拦截。
* **缓存雪崩:** 大量缓存同时过期，导致请求全部涌向数据库。
  + **解决方案:** 设置不同的随机过期时间；使用 Redis 集群/高可用方案；热点数据考虑逻辑过期。
* **缓存击穿:** 热点 Key 过期，大量并发请求访问该 Key，导致请求全部打到数据库。
  + **解决方案:** 使用互斥锁（如 Redis SETNX 或 Redisson 分布式锁）只允许一个请求去加载数据并写缓存，其他请求等待；热点数据设置逻辑过期（不设置物理过期时间，后台异步更新）。
* **数据一致性:**
  + **Cache Aside Pattern (先更新DB，再删除Cache):** 是常用策略，能保证最终一致性。存在短暂不一致窗口。
  + **强一致性要求场景:** 可能需要更复杂的方案（如读写锁、Canal+MQ 异步更新缓存等，本项目初期不考虑）。

## **7. 关键设计考虑 (Key Design Considerations)**

### **7.1 安全性 (Security)**

* 已在架构设计（Nginx HTTPS, 内网部署）、API 设计（JWT, 权限控制）、缓存策略（Session/JWT 黑名单）、非功能性需求（密码策略, 输入验证, 依赖扫描, 日志审计）等章节详细阐述。
* **核心原则:** 纵深防御，默认不信任原则。

### **7.2 事务管理 (Transactions)**

* 使用 Spring 的 @Transactional 注解，由 DataSourceTransactionManager 管理 MyBatis 的数据库事务。
* 确保 Service 层方法的事务边界清晰，覆盖所有需要原子性操作的业务逻辑（如下单流程）。
* 注意事务传播行为 (propagation) 和隔离级别 (isolation) 的设置。

### **7.3 异常处理 (Exception Handling)**

* **全局异常处理:** 使用 @RestControllerAdvice + @ExceptionHandler 捕获全局异常。
* **自定义业务异常:** 定义继承自 RuntimeException 的业务异常类（如 InsufficientStockException, ResourceNotFoundException, UserAlreadyExistsException），携带错误码和信息。
* **统一错误响应:** 全局处理器将捕获的异常（包括业务异常、参数校验异常 MethodArgumentNotValidException 等）转换为统一的 Result 对象返回给前端。
* 错误码映射示例:

| 异常类 | HTTP Status | Result Code | Result Message (示例) |
| --- | --- | --- | --- |
| ResourceNotFoundException | 404 | 4040 | “资源未找到” |
| InsufficientStockException | 400 | 4002 | “商品库存不足” |
| UserAlreadyExistsException | 400 | 4001 | “用户名或邮箱已存在” |
| MethodArgumentNotValidException | 400 | 4000 | (从异常中提取具体校验错误) |
| AccessDeniedException | 403 | 4030 | “无权访问” |
| AuthenticationException | 401 | 4010 | “未认证或认证失败” |
| Exception (通用) | 500 | 5000 | “服务器内部错误” |

### **7.4 日志记录 (Logging)**

* **框架:** SLF4j (API) + Logback (实现)。
* **配置:** 使用 logback-spring.xml 进行配置，实现异步日志记录，区分不同环境的日志级别和输出目的地。
* **内容:** 日志输出包含时间戳、线程名、**TraceID** (通过 MDC 实现，用于链路追踪)、日志级别、类名方法名、行号、具体消息。
* **关键点记录:** 请求入口和出口 (含耗时)、Service 方法调用、关键业务步骤、数据库交互、缓存操作、异常堆栈。
* **日志级别:** 合理使用 DEBUG, INFO, WARN, ERROR。生产环境通常设置为 INFO 或 WARN。

### **7.5 配置管理 (Configuration Management)**

* **外部化配置:** 使用 Spring Boot 的 application.properties 或 application.yml。
* **多环境配置:** 利用 Spring Profiles (application-dev.yml, application-prod.yml) 管理不同环境的配置。
* **敏感信息管理:**
  + **推荐:** 使用配置中心（如 Nacos, Apollo）或操作系统环境变量、K8s Secrets/ConfigMaps。
  + **次选:** 对配置文件中的敏感信息（如数据库密码）进行加密（如使用 Jasypt）。**避免明文硬编码。**
* **动态刷新:** 若使用配置中心，可实现配置的动态刷新。

### **7.6 数据库 Schema 管理 (Database Schema Management)**

* **目的:** 系统化地管理数据库表结构、索引、初始化数据的变更和版本控制。
* **推荐工具:** **Flyway** 或 **Liquibase**。
* **集成:** 将工具集成到 Spring Boot 应用中，在应用启动时自动检查并执行必要的数据库迁移脚本。
* **脚本管理:** 在 src/main/resources/db/migration (Flyway) 或类似目录下按版本号顺序管理 SQL 迁移脚本。
* **流程:** 开发人员编写 SQL 脚本 -> 提交到版本控制 -> 应用部署时自动执行。

### **7.7 监控与可观测性 (Monitoring & Observability)**

* **目标:** 实时了解系统运行状态，快速定位和解决问题，保障 NFR 中定义的可用性和性能目标。
* **度量 (Metrics):**
  + **来源:** Spring Boot Actuator (/actuator/prometheus), Micrometer, JVM Metrics, DB Pool Metrics, Redis Metrics。
  + **采集:** 使用 **Prometheus** Server 拉取指标数据。
  + **关键指标:** QPS, API 接口 P95/P99 响应时间, 错误率, JVM 堆内存/GC 情况, 线程池状态, 数据库连接池使用率, Redis 命中率/内存使用。
* **日志 (Logging):** (见 7.4 日志记录) - 集中式日志收集（如 ELK Stack / EFK Stack）便于查询分析。
* **追踪 (Tracing):**
  + **(可选，推荐用于复杂系统)** 使用分布式追踪系统（如 **SkyWalking**, Zipkin, Jaeger）记录请求在系统内部（跨服务、跨组件）的调用链和耗时。
  + **实现:** 应用集成追踪 Agent 或 SDK (如 skywalking-java-agent)。
* **可视化与告警 (Visualization & Alerting):**
  + 使用 **Grafana** 创建 Dashboard 展示 Prometheus 收集的指标和日志数据。
  + 配置 **Alertmanager** (Prometheus 生态) 或 Grafana Alerting，根据预设规则（如错误率超阈值、响应时间过长、资源耗尽）触发告警通知（邮件、钉钉、微信等）。

## **附录：分析模型图 (UML)**

**(以下为 SRS 中描述的分析模型图的 UML ，包含在设计文档中作为参考和最终确认)**

### **状态图 (State Diagram - 订单状态)**

### **diagram_06**

### **活动图 (Activity Diagram - 简化下单流程)**

### **diagram_07**

### **用例图 (Use Case Diagram)**

