Design Model Documentation

System Analysis and Design

Spring 2021

@Tongji

A Comprehensive Campus Community Platform for Students

Team Members:

1853572 Qiao Liang

1854062 Zhibo Xu

1854117 Zhenyu Li

1953803 Yixi Zheng

1954106 Wenchao Chen

同济大学
TONGJI UNIVERSITY

Table of Contents

1 Introduction

1.1 Main Goals

This project aims to provide a campus information exchange platform with the **three core functions** of on-campus second-hand trading, community information management, and evaluation of public elective courses for the majority of students of Tongji University. Among them, campus second-hand trading is the **core function** of this software platform.

1.2 Target Domains

Our project aims to serve the students of Tongji University, facilitates students' life, course selection and extracurricular activities.

1.3 Progress Made on System Design

In the previous report, we mainly completed system analysis including system architecture, analysis model and user interface perfection. In addition, a set of class diagrams are constructed and the relationship between the classes is shown. **At this stage**. We first completed the update of the **logical architecture** and provided the corresponding **development architecture** and **deployment architecture**. Secondly, we completed the design of the **API** based on the previous use case implementation, and provided two **design mechanisms**, the **use case implementation** and **three design patterns** based on the design mechanism, and finally a **prototype demo** of one of the core technologies was implemented.
The detailed progress is as follows:

1. By comparing the architecture styles horizontally, selecting the most suitable **micro-service architecture style** and **port-adapter style** for our subsystems ,we finally completed the logical architecture update. The overall micro-service architecture was adopted. At the same time, the deployment architecture was updated and the development architecture of this project was designed. The details will be introduced in the later part.
2. We have designed a specific interface list for each subsystem. In the API design part, we adopted Rest-style API suitable for micro-service architecture, and gave a list of **RESTful APIs** that need to be implemented. Taking Second-hand Sales System as an example, we provided interface specifications including request parameter types and return parameter types. And parameter list.
3. Provides detailed interface specifications for using our subsystem to call third-party APIs.
4. Adopting two design mechanisms named **data persistence** and **distribution** to improve the safety, reliability, fault tolerance and flexibility of the project, and provide **use case realization** based on these two design mechanisms.
5. Further discuss the design patterns, analyze the **strategy pattern**, the **proxy pattern** and the bridge pattern in detail, and summarize the key design decisions adopted in our analysis process.
6. Provides a prototype implementation of one of the technologies.
7. Summarize and reflect on the shortcomings of the project.

1.4 Description of the Platform and Framework

In the actual development of the **@tongji campus community website**, we adopted the **micro-service framework** as the overall framework for system development and design**.** Different from the **monolithic application framework**, the micro-service framework can make the system division of labor clearer and the boundaries clearer. At the same time, the persistence layers of the database are isolated from each other. Using the micro-service framework to build a **low-coupling** system can make the application

more scalable, more flexible and stable. In addition to using **Node.js** as the javascript runtime environment, we also use **NPM** as the package management tool, **Vue.js** as the front-end development framework, **webpack** as the module packaging tool. In order to prevent the database from becoming a performance bottleneck and better deal with the service dependency problem caused by the implementation of the microservice framework, we adopted the distributed database system **DDBS**. Some of the most frequently accessed databases introduce **MEMCache** and **Redis** Cache mechanisms, Use both **Redis exporter** and **MySQL exporter** as the indicator interface of the cache and database, and **Prometheus** as the indicator collector. The specific construction of the remaining micro-service framework will be elaborated in the micro-service architecture diagram of the architecture part.f

In terms of improving the **interface call problems** brought by the micro-service framework, we build the corresponding **API gateway** as a unified platform for providing service interfaces. For this, we plan to use **Eureka** (service discovery), **Hystrix** (circuit breaker), **Zuul** (intelligent routing) and **Ribbon** (client load balancing) services provided by **Spring Cloud Netflix**.

Since we use a message mechanism based on **queue**, we use **RocketMQ** as the system's distributed message middle-ware. RocketMQ has the characteristics of high performance, high reliability, distribution, etc., especially in line with the management needs of high-frequency message requests of consumer clusters in the second-hand transaction service of our system.

2 Architecture Refinement

2.1 Architecture Description

2.1.1 Logical Architecture

In the last assignment, we developed a logical architecture based on the **hierarchical architectural style**. In this assignment, we used the **micro-service architecture pattern** to accommodate the requirements of our project's many functionally independent partitions. Therefore, we first improve the logic architecture based on the single application pattern to the logic architecture based on the micro service pattern. The following is our **analysis** of the selection of architectural style.

Since the micro-services architecture is to some extent born of the port and adapter architecture style, we are also considering using the port-adapter architecture style. Horizontal comparison was made to draw micro-service architecture diagrams using layered architecture and port-adapter architecture respectively, as shown in the figure below:

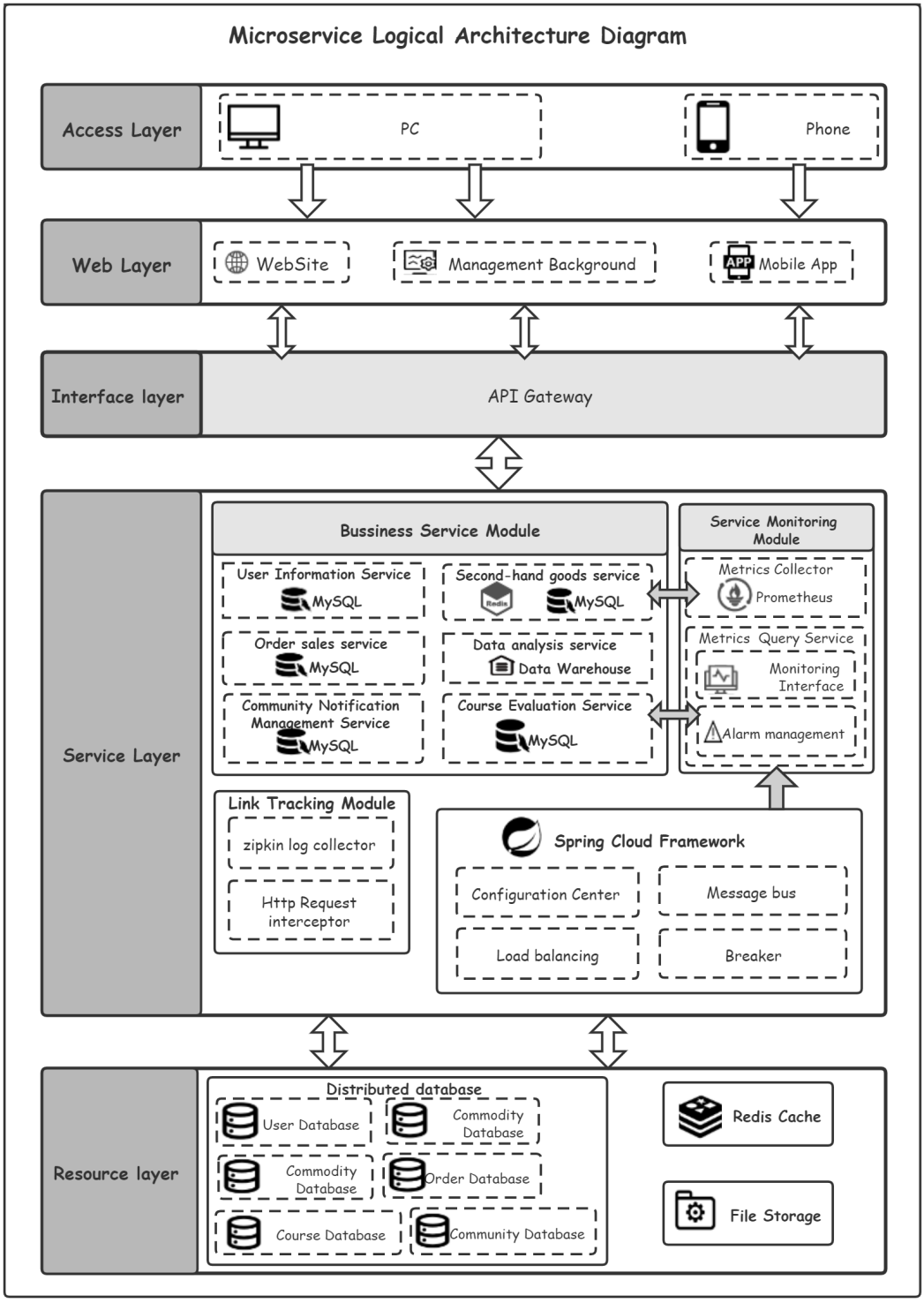

Figure 2.1 Micro-service architecture diagram based on the **Layer Style**

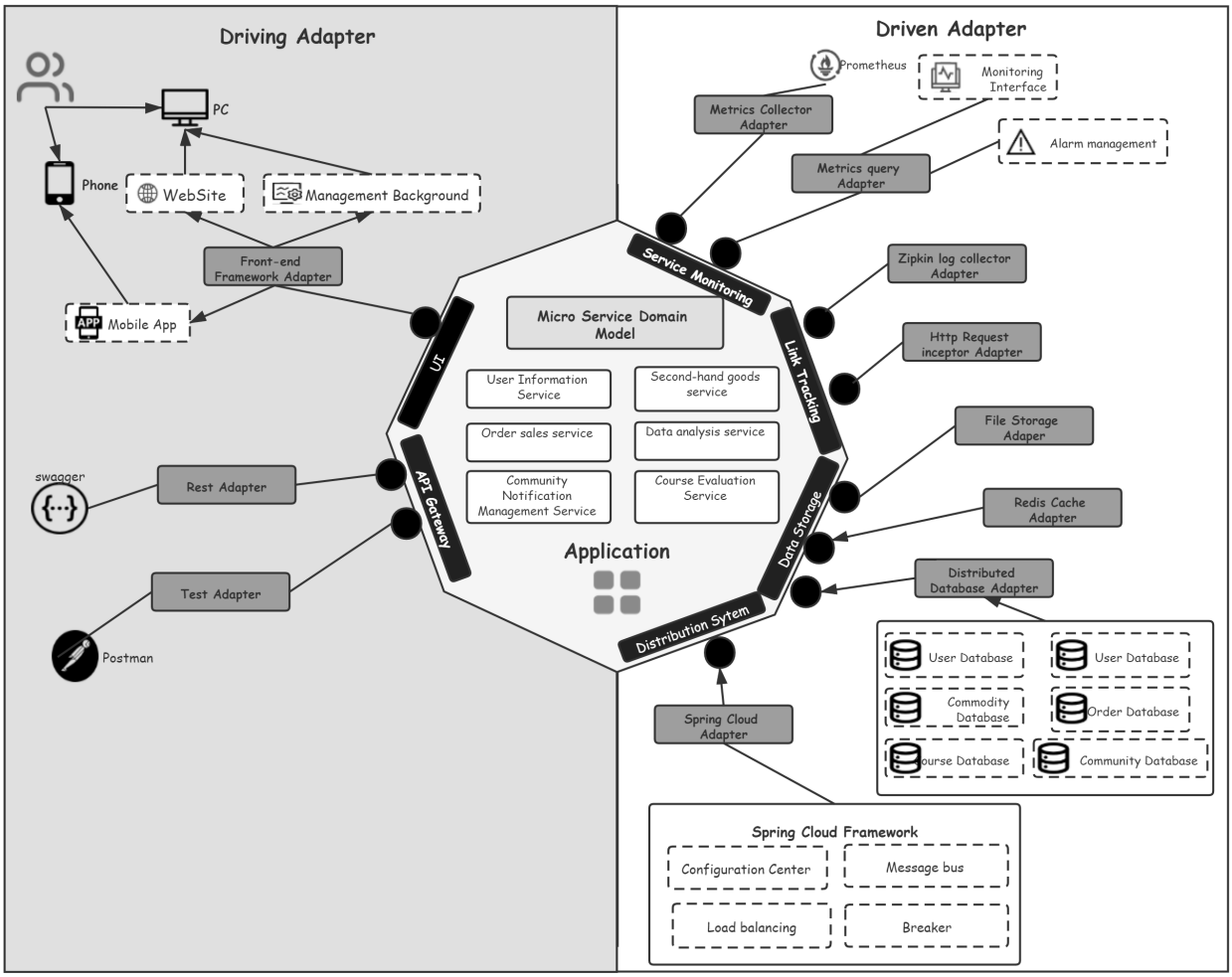

Figure 2.2 Micro-service architecture diagram based on **Port and Adapter** Style（Finally Chosed)

First, the layered architecture is still the most widely used architectural style today, but it has the following disadvantages:

1. The architecture is oversimplified, and some special functional requirements (such as sending email notifications) cannot be simply placed in a layer
2. Some businesses cannot be simply layered with other businesses

For our system, because of the need for users to obtain data directly from the data layer, this need cannot be represented by the layered architecture. In fact, any cross-layer business logic cannot be well represented by the layered architecture. The port and adapter architecture solves these problems by dividing our system into loosely coupled interchangeable components, such as the **program core, database, user interface, API interface**, and so on. A detailed analysis of the port and adapter architectural styles is described in the section on architectural styles later in this article.

To sum up, we chose **port and adapter** styles that are more suitable for the micro-service pattern as the architectural style of our logical architecture design. The specific design ideas of our micro-service architecture are as follows:

The logical architecture consists of a core program, six main ports, and 11 adapters, which are divided into two categories: Driving adapters and Driven adapters. The first is the core program. The core program is mainly used to implement the domain model including six micro-services, which are:

- User Information Service
- Second-hand Goods Service
- Order Sales Service
- Data Analysis Service
- Community Notification Service
- Course Evaluation Service

The second is our port section. The first is the **UI port**, which is similar to the presentation layer of a hierarchical architecture and is responsible for interacting with users and users. For this port, we take the front-end framework as the adapter, such as VUE full technology stack, etc., which is directly responsible for the interaction with each display end, such as Web end, APP end, etc. The PC and mobile phone are the primary devices that ultimately connect with the user. The second is the **API gateway port**, which is really the only channel for front-end and back-end data interaction in the hierarchical architecture. For this port, there are two main adapters, starting with the REST adapter. The REST API is the way our system uses to make interface calls, and will be the only design style for our design API to complete a series of specifications for our interface calls. The REST adapter is primarily intended to speed up the development of our API gateway, using **PostMan** as the API test platform.

For the Driven Adapter, the first Port is the **Service Monitoring Port**, which connects three adapters, the Metric Collector Adapter and the Metrics Query Adapter. The second is the **Link Tracking Adapter**, which provides Link Tracking management for our project. Then it is the distributed system port, which mainly provides the distributed management module, specifically using the Spring Cloud Framework. Finally, the **data port** provides three adapters, which are responsible for the storage of distributed database, the implementation of file storage and caching mechanism respectively, to improve the speed of data access.

2.1.2 Deployment Architecture

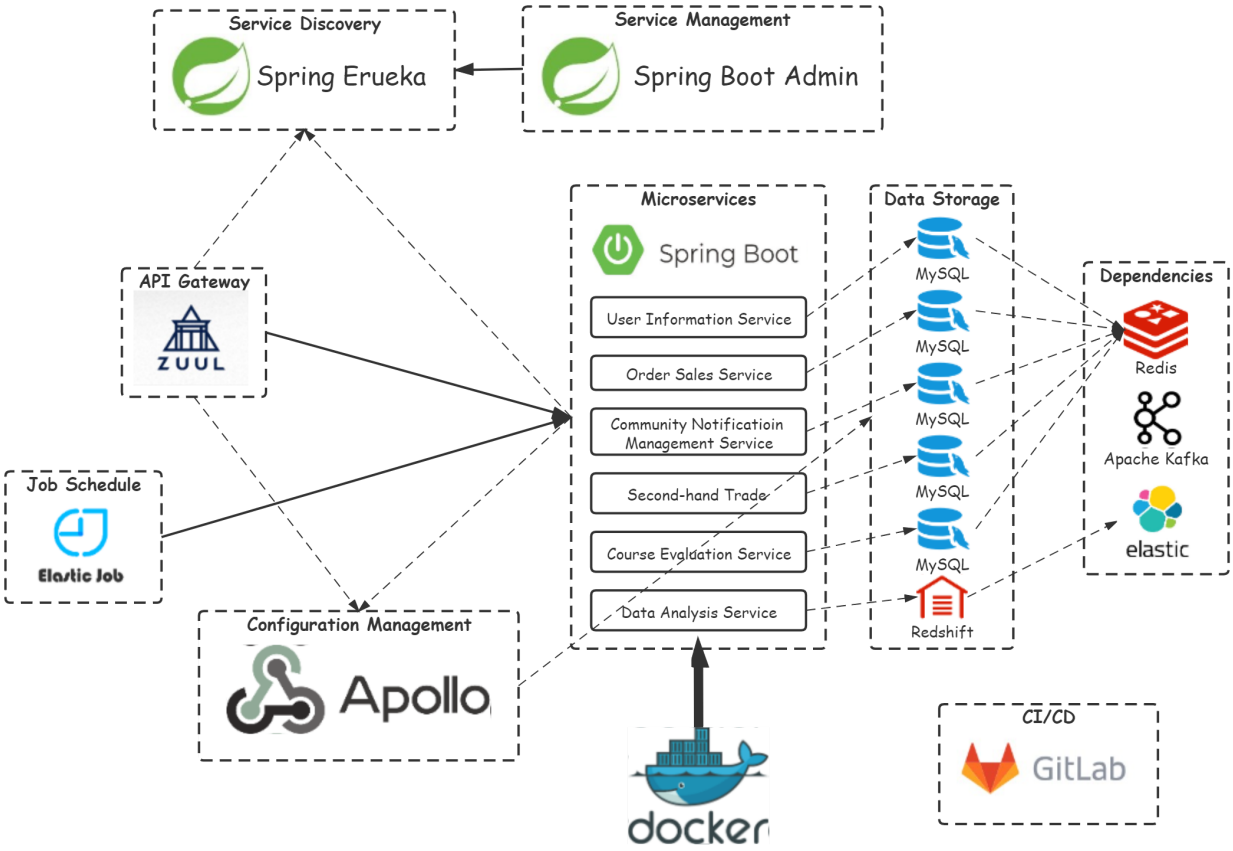

Figure 2.3 Deployment architecture diagram based on micro-service mode

The deployment architecture of our project uses micro-services style. Related terms are explained as follows:

Micro-services

Six micro-services all use **Spring Boot framework**, and are all registered in **Spring Eureka**, which is a service discovery component. **Spring Boot Admin** is used to manage our micro-services.

Configuration

We use Apollo to manage all of our configuration files of micro-services and the configures about the API Gateway, which is using ZUUL instead of Spring Cloud Gateway. Apollo is a distributed configuration center developed by Ctrip's framework department. It can centrally manage the configuration of different environments and different clusters of applications. After the configuration is modified, it can be pushed to the application side in real time, and it has standardized permissions, process governance and other features. It is suitable for microservice configuration management scenarios.

CI&CD

Continuous Integration and Continuous Delivery.

GitLab CI/CD is a tool built into GitLab for software development through the continuous methodologies:

- Continuous Integration (CI)
- Continuous Delivery (CD)
- Continuous Deployment (CD)

Continuous Integration works by pushing small code chunks to your application's code base hosted in a Git repository, and to every push, run a pipeline of scripts to build, test, and validate the code changes before merging them into the main branch.

Continuous Delivery and Deployment consist of a step further CI, deploying your application to production at every push to the default branch of the repository.

Data Storage and Dependencies

Except Data Analysis Service, the data base of all micro-services is **MySQL** because we only need Lightweight database. Considering the huge data to analyze user's behavior to make business decisions, we apply **Redshift** to implement data analysis service. All DBs need **Redis** to use distributed cache. Data Analysis Service need **ElasticSearch** to search informations. **Apache Kafka** is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications, it provides us a message system of excellent performance.

Docker

Docker is an open source application container engine that allows developers to package their applications and dependent packages into a portable image, and then publish to any popular Linux or Windows machine, which can also be virtualized. Containers use the sandbox mechanism completely, and there will be no interfaces between them.

2.1.3 Development Architecture

The development architecture of our system is shown in the figure below:

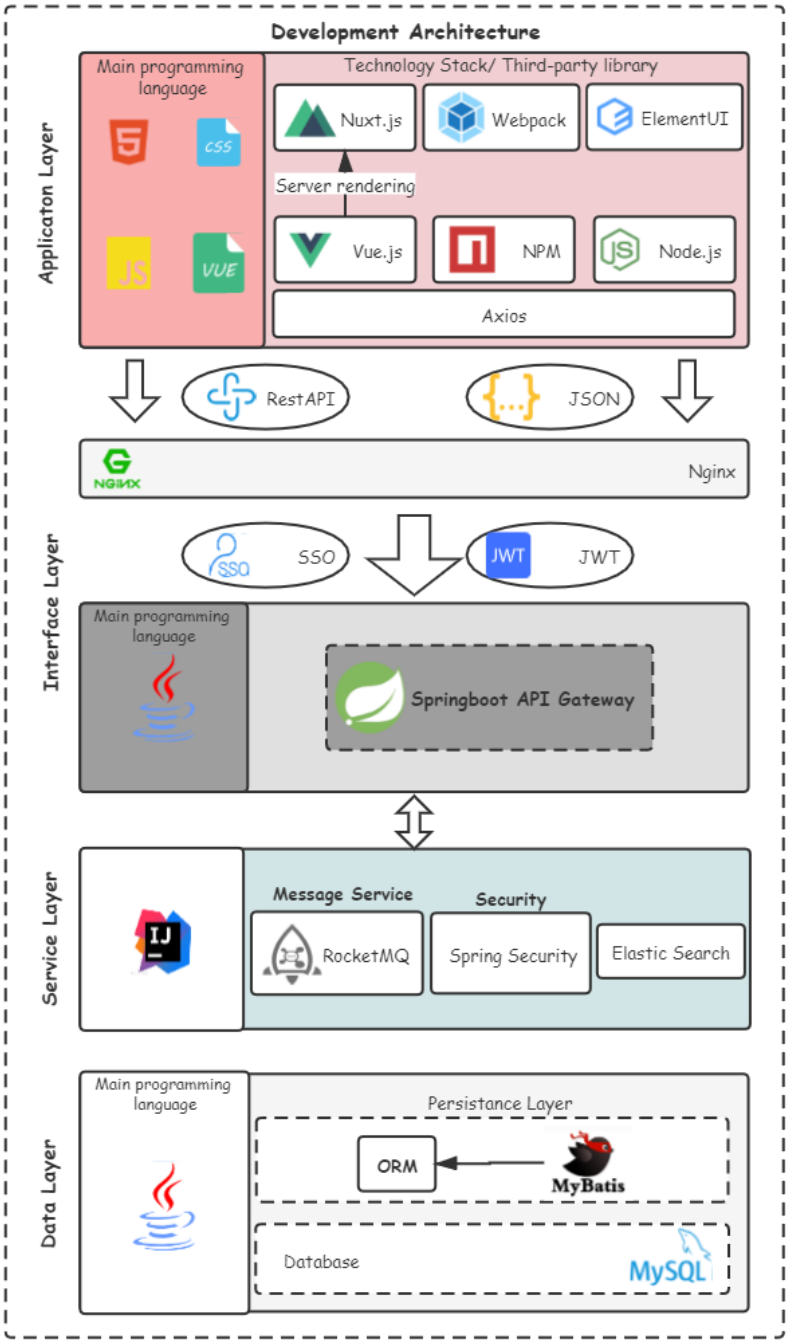

Figure 2.4 Development Architecture Diagram Based on Hierarchical Architecture style

Since the technology stacks between different layers in the project development process are relatively **concentrated and unified**, a layered architecture is adopted to show our development architecture. The **technology stack** of the development architecture is mainly divided into four layers:

- Application Layer
- Interface Layer
- Service Layer
- Data Layer

The first is the **front-end** part. The front-end part mainly has these core technologies:

1. **Nuxt.js**: **Nuxt.js** is an easy-to-use VUE framework. It is a framework for our web based on the VUE front-end framework to significantly speed up development efficiency. For the web side of our campus community system, Nuxt.js can be used to achieve server-side rendering, which can quickly and easily implement SSR, and can generate static websites according to the VUE program.
2. **axios**: a Promise-based HTTP client, is used in our system with the VUE front-end framework to process asynchronous communication with the back-end and better interact with the server.
3. **Node.js** is a javascript runtime environment based on Chrome V8 engine, which is used to conveniently build web applications with fast response speed and easy expansion.
4. **Vue.js** is a mature front-end development framework. Using VUE.js instead of native js code can achieve extensible data binding, while simplifying the API, using component management to build front-end pages, which can well adapt to the needs of our system's web side.

For the **interface layer technology stack**, we use **Nginx**, a reverse proxy web server. Since the web side of our community software will be in high concurrency for a long time, using Nginx can achieve **reverse proxy** and **load balancing**. Secondly, when designing the system, we considered the login method of unified identity authentication login, so we adopted the popular single sign-on method **SSO**. Corresponding to SSO, we adopt the standard of **JSON web token** (JWT) , to realize login authorization and secure information exchange.

In addition, we use **Springboot** to deploy a rest-style API framework. The reason why Springboot is used instead of other Spring frameworks is because it simplifies the initial setup and development process of spring applications and is more lightweight.

As for the technology stack of the service layer, due to the many technologies used in the technology stack of the service layer, due to space limitations, here are two core technologies that our system intends to adopt. The first is the **message service part**. Since our system often faces a large number of message requests, it is necessary to adopt an appropriate MQ mechanism to achieve timely response to messages. For this, we plan to use the recently popular **RocketMQ** core component developed by Alibaba. Using RocketMQ can provide the following two advantages for our system:

1. **Peak-shaving and valley-filling**, that is, to reduce the problems of system crash and missing information caused by excessive transient pressure (such as the period when students choose courses).
2. **System decoupling**. Our system adopts a microservice architecture. A very important requirement for adopting microservices is the decoupling between services. Using RocketMQ can reduce the avalanche phenomenon caused by a function crash.

There are many popular MQs, such as Kafka, RabitMQ, etc. Compared with these two components, **RocketMQ** supports transactional messages and meets the requirements of our system.

The second is the **security issue**. Since our system is mainly for students in Tongji, we need to ensure the security of every user who logs in to our system, so we adopted **Spring security**. Spring security can implement functions including authentication, authorization, and attack protection.

For the **technology stack** of the data layer, the first is the database management system **DBMS**. The current mainstream DBMSs include oracle and MySQL. Since our system is oriented to schools and the data volume is not too large, using an enterprise-level DBMS such as oracle may seem heavy. Therefore, we choose the lightweight **MySQL** as the DBMS.

The second important technology we need to achieve is **data persistence**. If we use SQL statements every time we access the data, it will inevitably be more troublesome, so we use the object-relational mapping **ORM** technology here. ORM technology encapsulates the relational database into objects to make it automatically persisted in the relational database, which makes the realization of persistence more convenient. **MyBatis** is a high-performance object-relational persistent storage and query service. Using this object-relational framework can easily realize the transformation of **RDBMS** items and Java objects.

2.2 Subsystem and Interface List

Since this system adopts the micro-service model, we divide the system into six core micro-services. The list of each microservice and its interface is shown in the following table:

2.2.1 Interface Function of Each Micro-Service

Personal Information Service

Interface Class:

- **IPersonInformationManagement**

  checkPersonType(string ID,string password) :bool

  resetUserPassword(string ID,string password): void

sendVerifyCode(string ID, string verifyCode): void

checkLoginInformation(string ID，string password): bool

checkReigisterInformation(string ID，string password, string eMail，string studentID):bool

createUserAccount():Account

deleteUserAccount(Account user): bool

- **ICollectionListManagement**

  addToCollectionList(string WebLink，CollectionList collections):bool

  deleteFromCollectionList(string WebLink):bool

  createCollectionList(string listName):CollectionList

  deleteCollectionList(CollectionList collections):bool

- **IHistoryManagent**

  addToHistoryList(string WebLink):bool

  deleteHistoryList(string WebLink):bool

Audit Management Service

Interface class:

- **IAuditManagement**

  addToAuditRecords(AuditRecord audit):bool

  sendNoticeToUser(string ID，Notice notice):void

  auditCommodityInformation(Commodity commodity):void

  auditAssociationInformation(Association association):void

  auditCourseInformation(Course cs):void

- **IPermissionManagement**

  restrictUserPermission(string ID):void

  restoreUserPermission(string ID):void

Second-hand Goods Service

Interface Class:

- **ICommodityManagement**

addCommodity(Commodity NewCommodity):JSON

deleteCommodity(string):void

modifyCommodityInfo(string CommodityID,Commodity NewCommodity):bool

getCommodity(string CommodityID):JSON

searchProductInfo(String CommodityInfo)： Commodity[]

filterCommodityByPrice(int Price):Commodity[]

filterCommodityByPlace(string CommodityPlace):Commodity[]

displayCommodityList(Commodity[]):JSON

getCommodityDetails(string CommodiyID):JSON

- **IFavoriteManagement**

  createFavorite(Favorite NewFavorite):bool

  deleteFavorite(string FavoriteID):bool

  getFavoriteCommodity(string FavoriteID):Commodity[]

  addCommodityToList(string CommodityID,string CommodityID):bool

Order Sales Service

Interface Class:

- **IOrderManagement**

  createOrder(Order NewOrder):Order

  getOrderInfo(string OrderID):JSON

  terminateOrder(string Order):JSON

  modifyOrderstatus():bool

- **IPaymentService**

  payByWechat(float Price)(External API)

  payByAlipay(float Price)(External API)

Community Notification Managment Service

Interface Class

- **ICommunityInformationManagement**

  modifyAssociationInfo(string AID):AssociationID

getAssociationInfo(string Keyword):Association

checkPersonType(string ID):PersonType

deleteAssociationInfo(string AID):AssociationID

Course Evaluation Service

Interface Class:

- **ICourseManagement**

  getPermission(string ID):bool

  getCourse():Course

  updateCourseFromAcademicAffairsSystem():Courses

- **ICourseEvaluationManagement**

  getCourseEvaluation(string Keyword):CourseEvaluation

  getStudentCourseSelectionInfo():SelectionInfo

  deleteCourseEvaluation(string EID):EvaluationID

  modifyCourseEvaluation(string EID):EvaluationID

2.2.2 REST API URL List

In the specific style design of the API, we adopt the **REST API style**. Therefore, we also give a list of REST style API requests for all micro-services. Since Restful API is a resource-based API style, the first layer of our API URL is api, and the second layer is the name of each resource. The manipulation of a specific data is represented by different levels of resource organization:

Personal Information Service

API NUM	API URL	DESCRIPTION
1	**GET** /api/user/{userid}	Obtain the user's personal information. Check whether the system already has the user's information when the user logs in.
2	**DELETE** /api/user/{userid}	Delete user from system.

API NUM	API URL	DESCRIPTION
3	**PUT** /api/user/{userid}/password	Update user's password. It is generally used to reset the password when the user forgets it.
4	**POST** /api/user/{userid}	Create a new user account (used when the user registers an account)
5	**GET** /api/administrator/{adid}	Obtain administrator's personal information.
6	**POST** /api/user/{userid}/collections	Create user's personal collection list.
7	**DELETE** /api/user/{userid}/collections	Delete user's personal collection list.
8	**PUT** /api/user/{userid}/collections	Update user's personal collection list.
9	**DELETE** /api/user/{userid}/history	Delete user's history.
10	**PUT** /api/user/{userid}/history	Update user's history.

Audit Management Service

API NUM	API URL	DESCRIPTION
1	**PUT** /api/user/userID/permission	Update user's permission. Users' permissions are generally managed by administrators.
2	**POST** /api/audit	Create audit record.
3	**PUT** /api/audits	Create list of audit records.

Second-hand Goods Service

API NUM	API URL	DESCRIPTION
1	**GET** /api/commodities/results	This API filters the corresponding second-hand products based on keywords and returns the product data .

API NUM	API URL	DESCRIPTION
2	**POST** /api/commodities	This API implementation creates the data entered on the page as a new second-hand product.
3	**PUT** /api/commodities/{CommodityID}	This API is implemented to update all the information of a specific id product.
4	**DELETE** /api/commodities/{CommodityID}	This API implements to delete all the information of a specific id product.
5	**POST** api/user/{UserID}/collections/{CommodityID}	This API can add specific product information to a specific user's favorites.

Order Sales Service

API NUM	API URL	DESCRIPTION
1	**POST** /api/orders	This API uploads the order information filled in by the user to the back-end database.
2	**DELETE** /api/orders/{orderid}	This API can delete all data of the order with a specific id.
3	**PUT** /api/orderes/{orderid}/status	This API can update the transaction status of a specific order to a specified transaction status, such as pending payment status, pending delivery status, refund status, etc.

Community Notification Management Service

API NUM	API URL	DESCRIPITION
1	**GET** /api/associations/result	the function of this interface is to accept the conditions of filtering activities and return the qualified association sequence

API NUM	API URL	DESCRIPITION
2	**PUT** /api/associations/{AssID}	the function of this interface is to update all information of the certain association (with id=AssID}).
3	**POST** /api/association	the function of this interface is to create a new association

Course Evaluation Service

API NUM	API URL	DESCRIPITION
1	**GET** /api/courses/result	the function of this interface is to accept the conditions of filtering activities and return the qualified courses sequence
2	**PUT** /api/courses/{course_id}	the function of this interface is to update all information of the certain course_id courses .
3	**PUT** /api/courses/{course_id}/Evaluations/{E_id}	the function of this interface is to update the evaluation of a course
4	**DELETE** /api/courses/{course_id}/Evaluations/{E_id}	the function of this interface is to delete the evaluation of a course
5	**POST** /api/courses/{course_id}/Evaluation	the function of this interface is to create a new evaluation

2.3 External Interface Specification Description

2.3.1 Overall Design

The platform uses Alipay and WeChat Pay to complete order payment and management, so the order management subsystem and the interface of Alipay and WeChat payment are designed to facilitate the transaction system to call, as shown in the figure below:

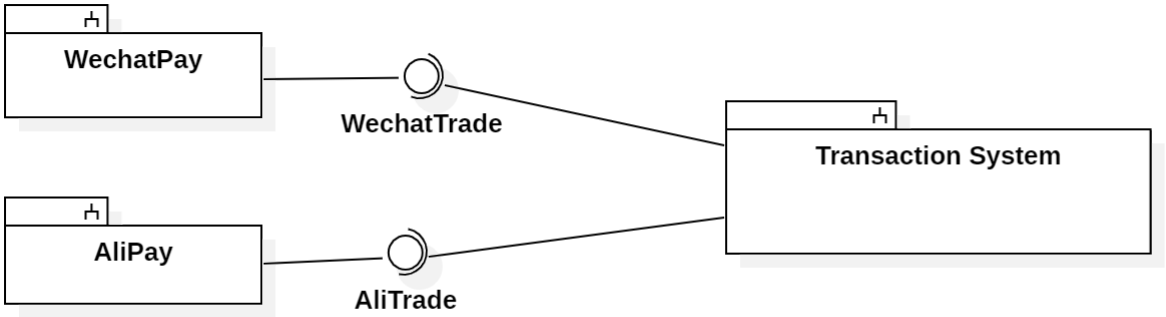

Figure 2.5 Interface Design Diagram

Since WeChat Pay is similar to Alipay, only the interface design of Alipay is introduced in detail. After consulting the API documentation of Alipay's open platform, the public request parameters and public responses required when calling are as follows:

公共请求参数

参数	类型	是否必填	最大长度	描述	示例值
app_id	String	是	32	支付宝分配给开发者的应用ID	2014072300007148
method	String	是	128	接口名称	alipay.trade.orderinfo.sync
format	String	否	40	仅支持JSON	JSON
charset	String	是	10	请求使用的编码格式，如utf-8,gbk,gb2312等	utf-8
sign_type	String	是	10	商户生成签名字符串所使用的签名算法类型，目前支持RSA2和RSA，推荐使用RSA2	RSA2
sign	String	是	344	商户请求参数的签名串，详见签名	详见示例
timestamp	String	是	19	发送请求的时间，格式"yyyy-MM-dd HH:mm:ss"	2014-07-24 03:07:50
version	String	是	3	调用的接口版本，固定为：1.0	1.0
app_auth_token	String	否	40	详见应用授权概述	
biz_content	String	是		请求参数的集合，最大长度不限，除公共参数外所有请求参数都必须放在这个参数中传递，具体参照各产品快速接入文档	

Figure 2.6 Schematic Diagram of API Request Parameters of Ali Payment System

公共响应参数

参数	类型	是否必填	最大长度	描述	示例值
code	String	是	-	网关返回码,详见文档	40004
msg	String	是	-	网关返回码描述,详见文档	Business Failed
sub_code	String	否	-	业务返回码，参见具体的API接口文档	ACQ.TRADE_HAS_SUCCESS
sub_msg	String	否	-	业务返回码描述，参见具体的API接口文档	交易已被支付
sign	String	是	-	签名,详见文档	DZXh8eeTuAHoYE3w1J+POiPhfDxOYBfUNn1IkeT/V7P4zJdyojWEa6IZs6Hz0yDW5Cp/viufUb5I0/V5WENS3OYR8zRedqo6D+fUTdLHdc+EFyCkiQhBxIzgngPdPdfp1PIS7BdhhzrsZHbRqb7o4k3Dxc+AAnFauu4V6Zdwczo=

Figure 2.7 Schematic Diagram of API Response Parameters of Ali Payment System

According to the needs of the system, the transaction management subsystem will call the five interfaces of AliPay. Corresponding methods are designed in the interface classes. As shown in the figure below, they are used to unify the input and output and facilitate the call of the transaction management subsystem.

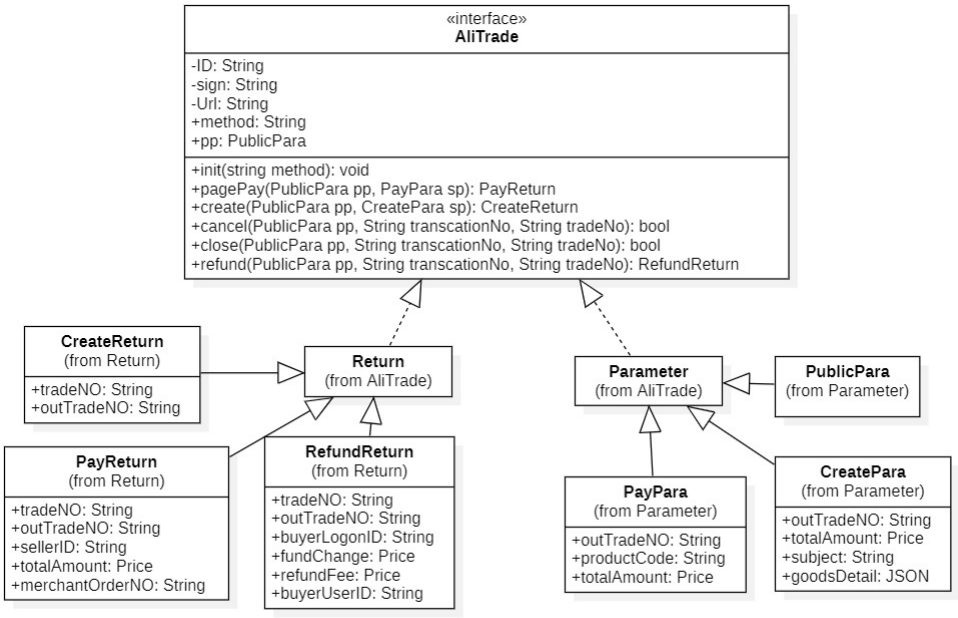

Figure 2.7 Class diagram of the Part Interacting With External Systems

2.3.2 Correspondence table

The following two tables show the correspondence between the methods and parameters in the interface and the AliPay API:

Table 1: Method and API

METHOD	CORRESPONDING INTERFACE
pagePay()	alipay.trade.page.pay
create()	alipay.trade.create
refund()	alipay.trade.refund
cancel()	alipay.trade.cancel
close()	alipay.trade.close

Table 2: Parameters and Corresponding Name

PARAMETERS	CORRESPONDING NAME	DESCRIPTIONS
tradeNO	trade_no	AliPay transaction number.
productCode	product_code	The product code signed by the merchant and AliPay.
totalAmount	total_amount	The total amount of orders.
outTradeNO	out_trade_no	Merchant order number. Customized by the merchant.
subject	subject	The title of the order. Note: Special characters cannot be used.
goodsDetail	goods_detail	The product list information contained in the order, in json format.
sellerID	seller_id	The Alipay unique user number corresponding to the receiving account.
merchantOrderNO	merchant_order_no	The original order number. The maximum length is limited to 32 digits.
buyerLogonID	buyer_logon_id	User's login id.
fundChange	fund_change	Wether there is any change in funds for this refund.
refundFee	refund_fee	Refers to the accumulated amount of the transaction that has been successfully refunded.

PARAMETERS	CORRESPONDING NAME	DESCRIPTIONS
buyerUserID	buyer_user_id	Buyer's user id on Alipay.

2.4 Subsystem Interface Specification Description

In this section, take **Second-hand Trading Subsystem** as an example to elaborate on the interface specifications of each REST API. As described in the architecture diagram, we used the **Swagger** tool in the API design phase to complete the writing of the interface specification **YAML** code. The startup page of Swagger is shown in the figure below:

Figure 2.5 Screen Shot of Swagger Local Connection

The interface part of the subsystem mainly involves using one interface class to manipulate two data classes, and its class diagram is shown in the figure:

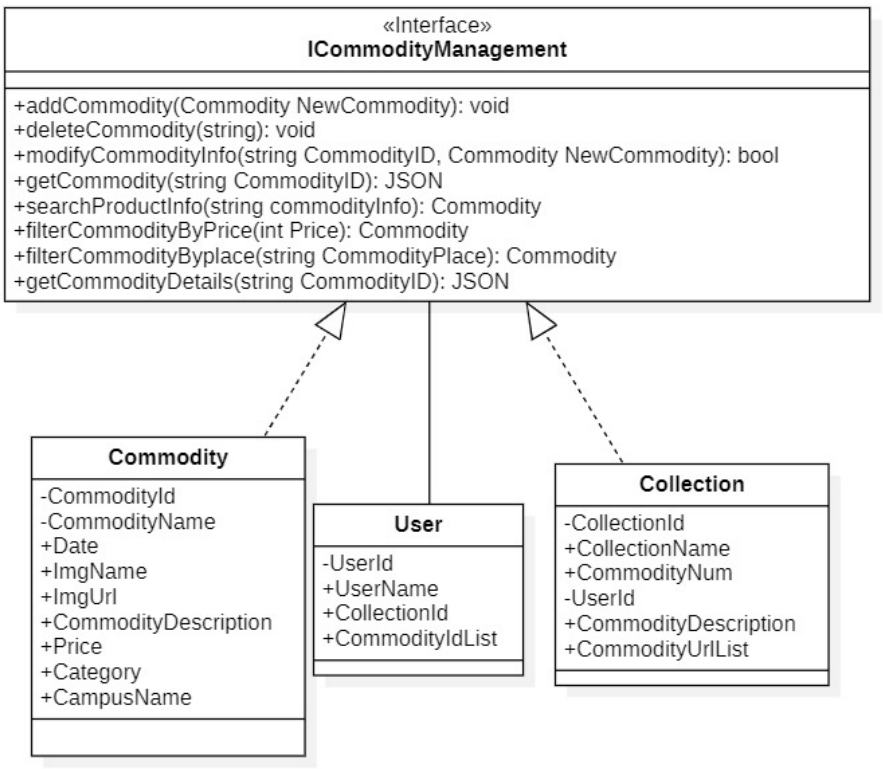

Figure 2.6 The Class Digram of Second-hand Trading Subsystem

2.4.1 Conditional query corresponding product list

Function Introduction:

This API can return corresponding search results that meet the query conditions according to the input query conditions.

Request URL:

GET /api/commodities/search/select/{q}

Public request parameters:

NAME	TYPE	NECESSITY	DESCRIPTION
q	string	YES	Query text, support elasticsearch query string syntax.
limit	integer	NO	The returned record size, the default is 100, and the maximum is 1000.
fields	array	NO	Comma-separated field array, query field array.

Public corresponding parameters:

NAME	TYPE	NECESSITY	DESCRIPTION
commodity_results	array of object	YES	The returned array of second-hand commodity objects that meet the query conditions.
result_num	integer		The total number of products that meet the query conditions.
error_code	interger		The requested error code, 1 means upload is successful, 405 means invalid input.

Request example:

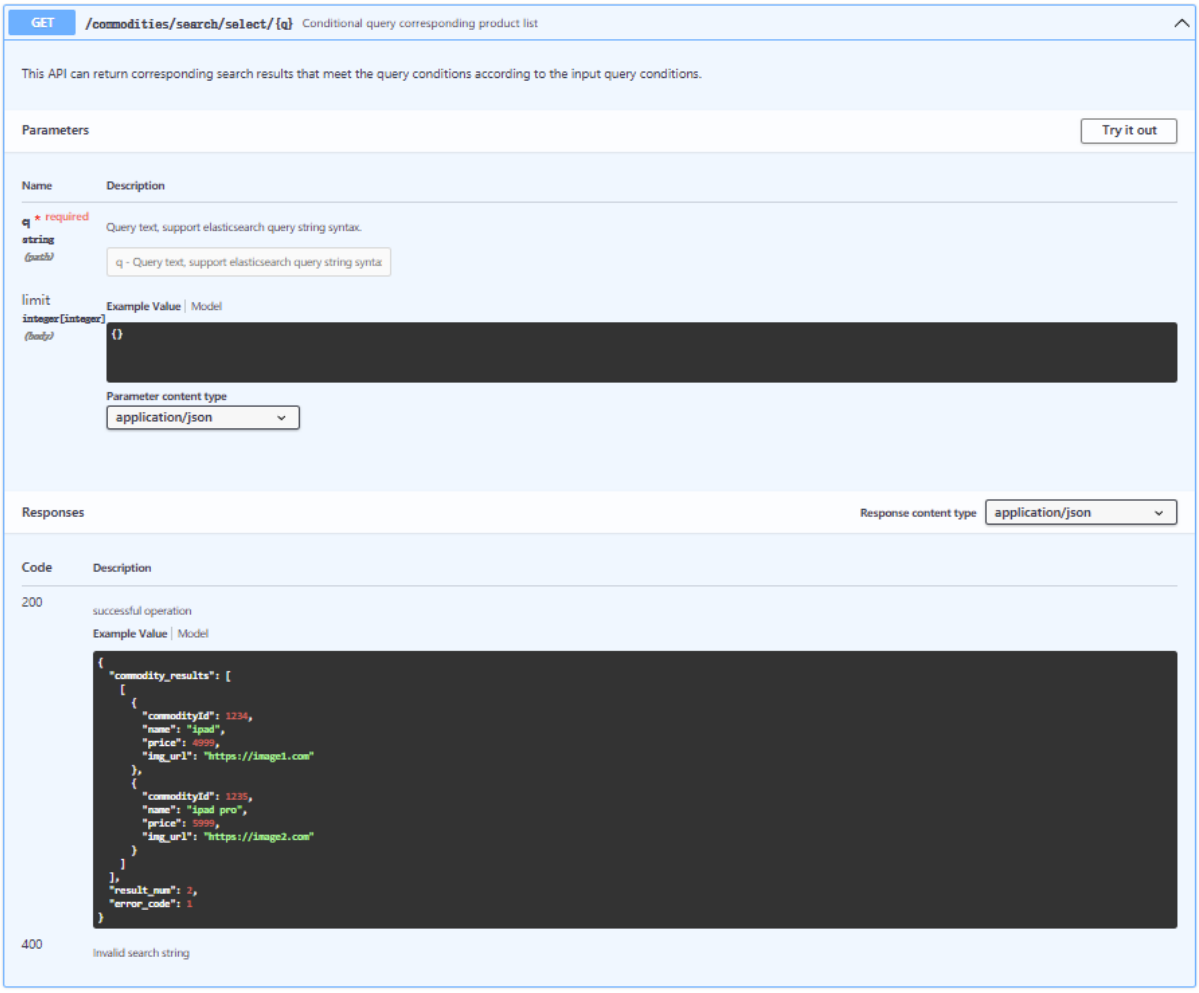

Figure 2.6 API specification for searching product

2.4.2 Create a new second-hand product

Function Introduction:

This API creates a second-hand product data object by submitting a synchronization request and submits it to the back-end database.

Request URL:

POST /api/commodities

Public request parameters:

NAME	TYPE	NECESSITY	DESCRIPTION
name	string	Yes	The name of product.
date	string	Yes	GMT time stipulated in HTTP 1.1,eg:Wed,05 Sep.2012 23:00:00 GMT.

NAME	TYPE	NECESSITY	DESCRIPTION
img_name	Array of string	Yes	The name of the product picture.
img_url	Array of string	Yes	The URL of the product's images.
commodity_description	string	Yes	The description information of the product, the maximum length of the information is 256 characters.
price	integer	Yes	The price of the commodity.
category	string	No	The category to which the product belongs.
campus_name	string	Yes	The campus to which the product belongs.

Public corresponding parameters:

NAME	TYPE	NECESSITY	DESCRIPTION
error_code		Yes	The requested error code, 1 means upload is successful, 405 means invalid input.

Request example:

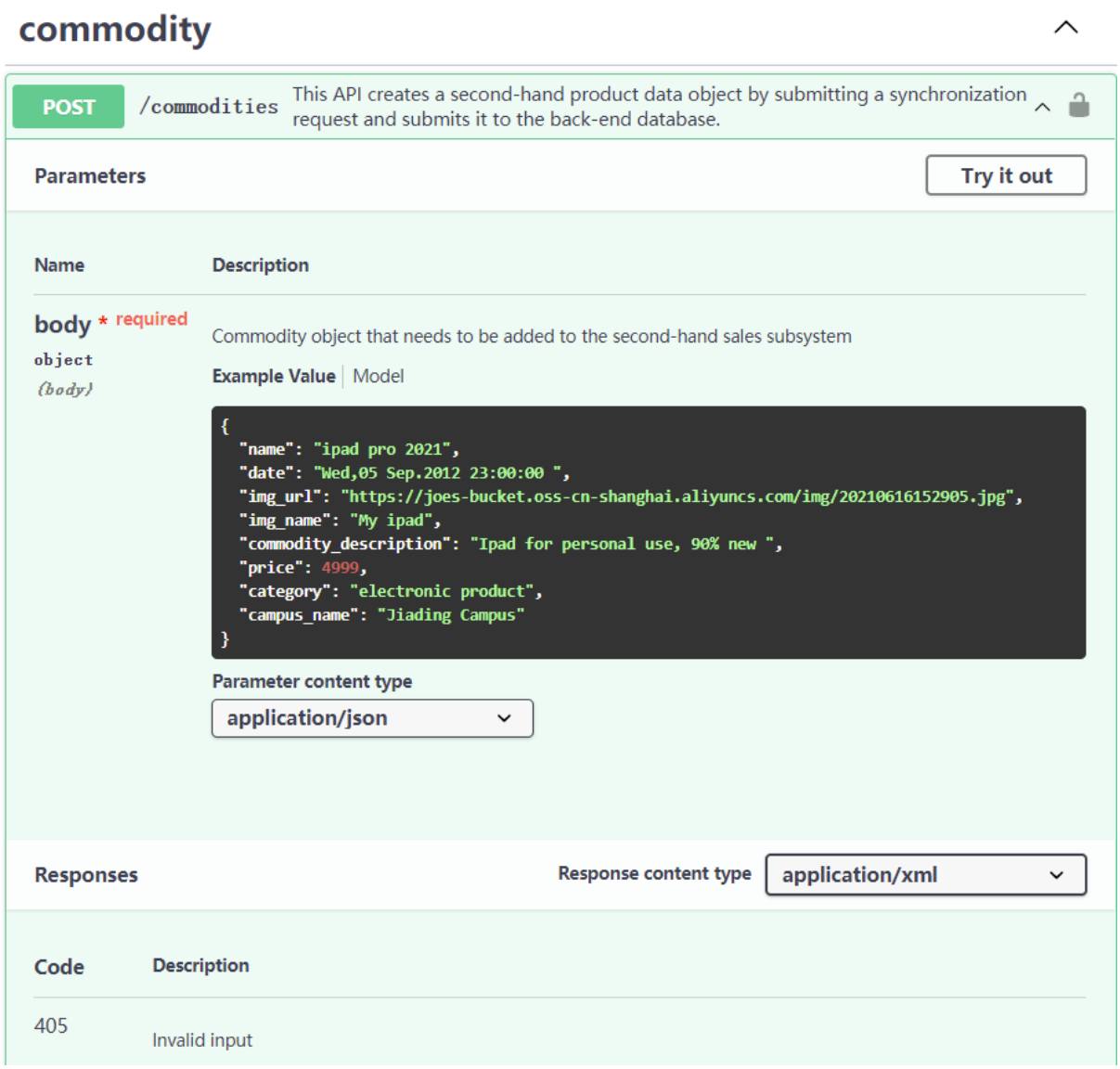

Figure 2.7 API specification for publishing product

2.4.3 Get the product detail information of a Commodity id

Function Introduction:

This API can obtain all the detailed information of the product according to the id field of the product, which is convenient for display on the front-end page.

Request URL:

GET /api/Commodities/{commodityId}

Public request parameters:

NAME	TYPE	NECESSITY	DESCRIPTION

NAME	TYPE	NECESSITY	DESCRIPTION
commodity_id	number	Yes	The id that uniquely identifies a second-hand item.

Public corresponding parameters:

NAME	TYPE	NECESSITY	DESCRIPTION
name	string	Yes	The name of product.
date	string	Yes	GMT time stipulated in HTTP 1.1,eg:Wed,05 Sep.2012 23:00:00 GMT.
img_name	Array of string	Yes	The name of the product picture.
img_url	Array of string	Yes	The URL of the product's images.
commodity_description	string	Yes	The description information of the product, the maximum length of the information is 256 characters.
price	integer	Yes	The price of the commodity.
category	string	No	The category to which the product belongs.
campus_name	string	Yes	The campus to which the product belongs.

Request example:

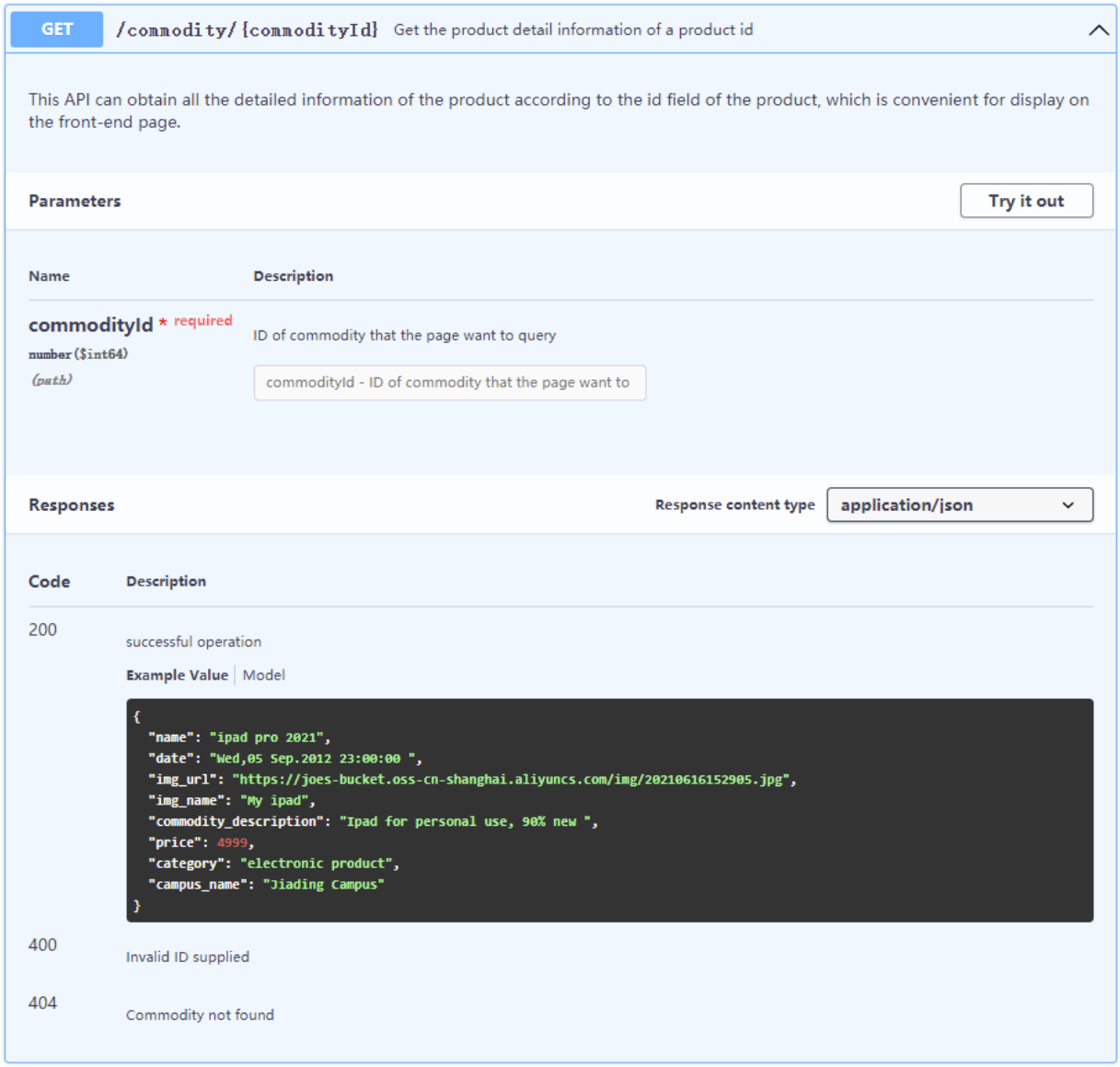

Figure 2.8 API specification for obtaining product information

2.4.4 Add a specific second-hand product to a specific user's collections.

Function Introduction:

This API adds a specific product infomation to a specific user's favorate and submits it to the back-end database.

Request URL:

POST /api/user/{userId}/collections/{commodityId}

Public request parameters:

NAME	TYPE	NECESSITY	DESCRIPTION

NAME	TYPE	NECESSITY	DESCRIPTION
commodity_name	string	Yes	The name of commodity.
commodity_id	number	Yes	The id of the commodity.
add_date	string	Yes	GMT time stipulated in HTTP 1.1,eg:Wed,05 Sep.2012 23:00:00 GMT. The time when the product was addded to the collections.
img_name	Array	Yes	The name of the product picture.
img_url	Array	Yes	The URL of the product's images.
commodity_url	string	Yes	the URL of the commodity's details page.

Public corresponding parameters:

NAME	TYPE	NECESSITY	DESCRIPTION
error_code	integer	Yes	The requested error code, 1 means upload is successful, 405 means invalid input.
favorate_full	string	Yes	The requested code,"false" means the favorate is full,"notfull" means is not full.

Request example:

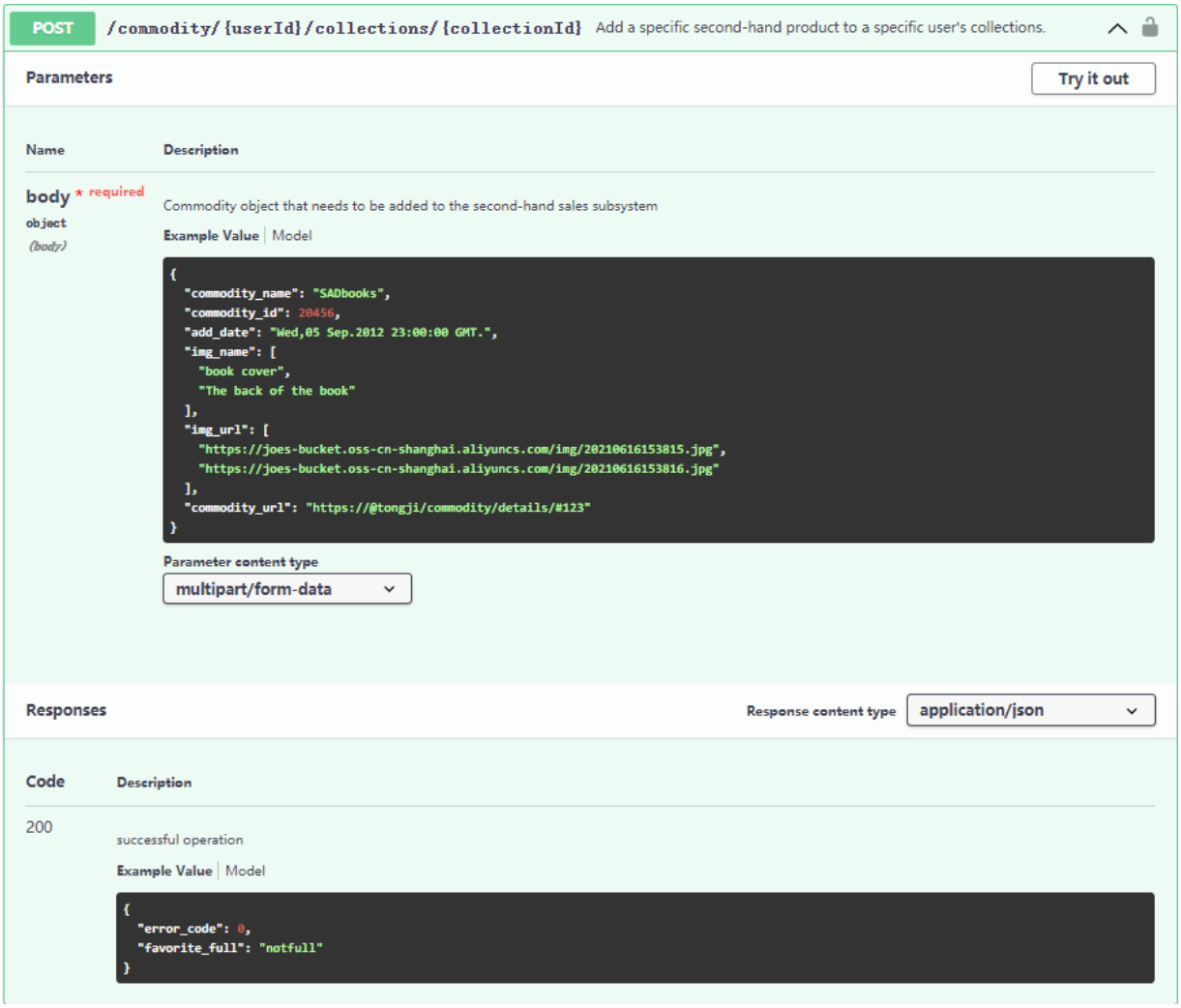

Figure 2.9 API specification for adding goods to collections

3 Design Mechanism

3.1 Data Persistence

Data persistence is the general term for converting an in-memory data model to a storage model, and a storage model to an in-memory data model. The data model can be any data structure or object model, and the storage model can be a relational model, XML, binary stream, etc. However, due to the extensive application of object model and relational model, data persistence is often mistaken for the transformation of object model to relational database. In this system, we use the relational database management system (**RDBMS**). A data persistence framework is a kind of middle-ware that assists and automates the storage of program data into databases, especially relational databases. At this level of abstraction between the application and the database, the program indirectly accesses the database through session control.

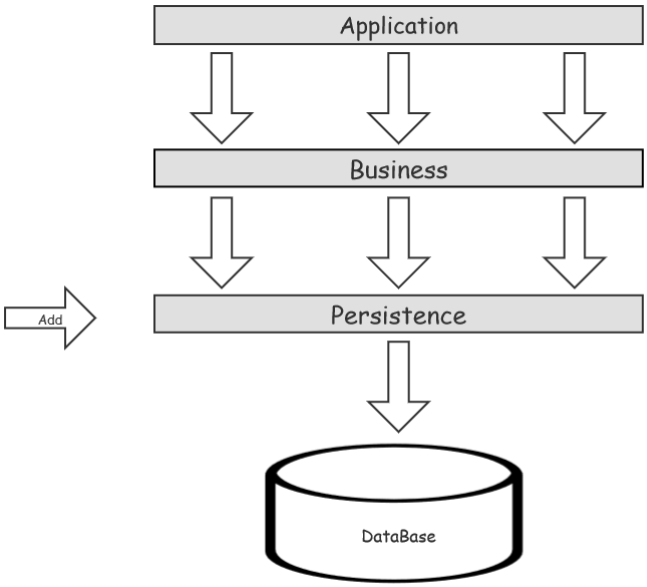

Figure 3.1 Schematic diagram of data persistence layer

Specifically, we use **ORM** ,which is a kind of data persistence technique, which is mainly to establish a relationship between the object model, such as JavaBean objects and the tables of the relational database, and provides a mechanism to manipulate the tables of the database through JavaBean objects. In effect, it creates a "virtual object database" that can be used in a programming language. Common ORM frameworks include **Hibernate** and **MyBatis.**

In addition to information transmission, the system also needs to provide users with persistent information services, that is, data should be stored in the system persistently. For the user information, product information, order information, course information, club information and so on that appear in our system, we should take effective ways to save them.

Regarding how to save the data, we have chosen **MySQL** as the relational database management system to save the data. After selecting the database, we also need to choose the tools to obtain data in the system, the most commonly used are JDBC, Hibernate, MyBatis and so on. After carefully weighing the advantages and disadvantages of different tools, we finally chose MyBatis.

JDBC is the original tool provided by Java. It is powerful but complex because programmers must load their own drivers, obtain connections, close resources… Therefore, we need another automatic or semi-automatic tool that allows the development process to focus on writing SQL.

MyBatis is a semi-automatic middle-ware. When using MyBatis, we can directly write SQL. And because of the existence of MyBatis, we do not need to configure the connection with the database. It implements a layer to maintain the connection between the business layer and the database, that is, the data persistence layer. In traditional JDBC mode, we need to close the connection after each query and reopen it when we start the next query, which takes more time. Native JDBC operations also have a lot of repetitive code, and the role of the framework is to encapsulate that tedious code so that programmers can focus on the SQL statements themselves.

Benefits of using data persistence mechanism

- The program code reuse, even if the replacement of the database, only need to change the configuration file, do not have to rewrite the program code.
- Business logic code readability, in the code will not have a large number of SQL language, improve the readability of the program.
- The persistence technology can be automatically optimized to reduce the access to the database, improve the efficiency of the program.

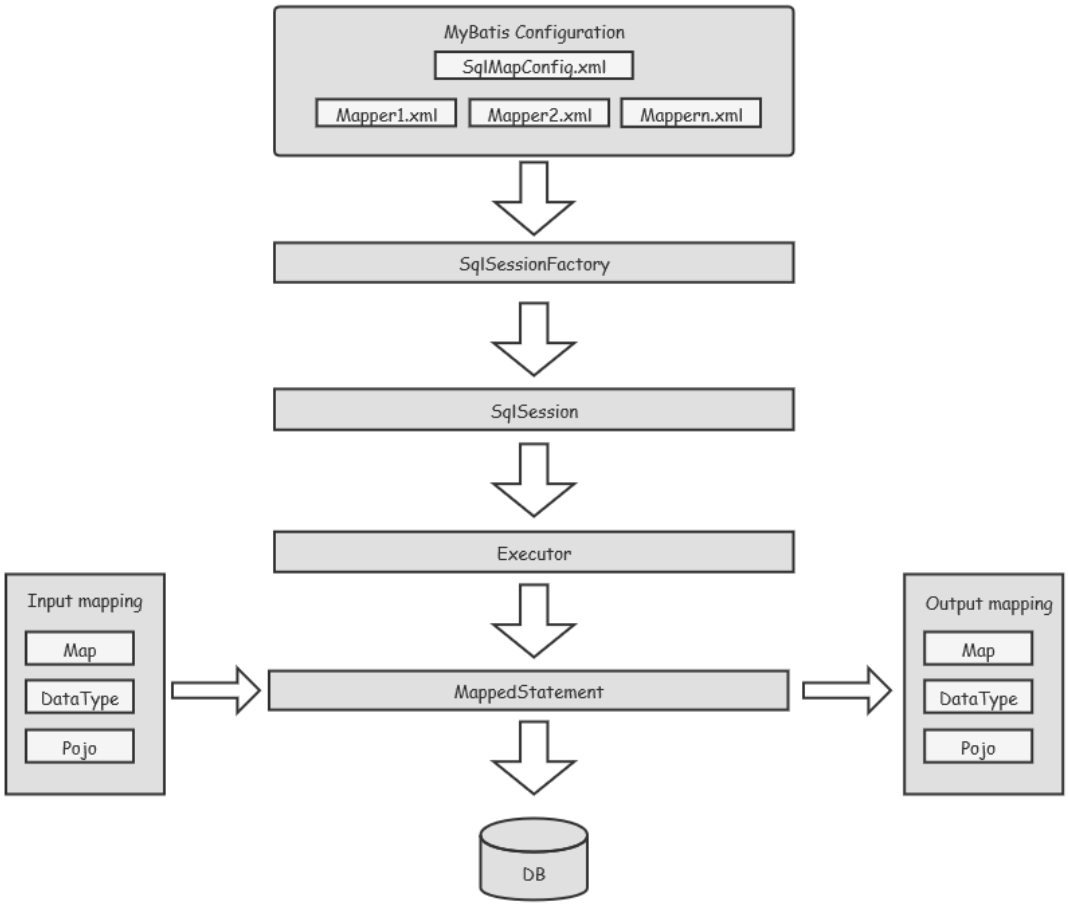

Figure 3.2 Functional Architecture Diagram of using MyBatis

The following will introduce the design idea of using MyBatis in the data persistence layer.

The functional architecture of MyBatis:

The functional architecture of MyBatis is generally divided into three layers:

- **API interface layer**: Interface APIs for external use, through which developers manipulate the database. As soon as the interface layer receives the call request, it calls the data processing layer to complete the specific data processing.
- **Data processing layer**： Responsible for specific SQL lookup, SQL parsing, SQL execution and execution result mapping processing, etc. Its main purpose is to complete a database operation based on the invoked request.
- **Base support layer**： Responsible for the most basic functional support, including connection management, transaction management, configuration loading and caching processing, these are common things, they are extracted as the most basic components. For the upper data processing layer to provide the most basic support.

The Main Execution Process of MyBatis

- MyBatis configuration file, including MyBatis global configuration file and MyBatis mapping file, in which the global configuration file configures data sources, transactions and other information; The mapping file configures information related to SQL execution.
- MyBatis reads configuration file information (global configuration file and mapping file), constructs **SQLSessionFactory**, namely session factory.
- With SQLSessionFactory, you can create **SQLSession** (session). MyBatis operates the database through SQLSession.
- SQLSession itself does not operate directly on the database. It operates on the database through the underlying **Executor** interface. The Executor interface has two implementation classes, a normal Executor and a cache Executor (the default).
- The SQL information to be processed by the Executor is encapsulated in an underlying object, **Mapped Statement.**

3.2 Distribution

Our analysis mechanism is distribution. The purpose of using **Distribution** is to integrate the instant messaging framework. Because it can decouple the server and the client, the caller and the callee at both ends of the network can be barrier-free communication. so it can expand the business scale. We want to split the business into many services and make the services deployed independently, to make low coupling. After splitting, the service can be started more quickly because less dependent libraries and less code are used.

When our business develops rapidly, there will be more and more developer. The division of services is conducive to the division of labor between teams, so that each team can be responsible for the corresponding business. When a service has a large number of visitors, we just need to expand this service, and do not need to expand other services together.

In order to facilitate the call of services between the client and the server, we decided to design api interface according to certain standards to facilitate the later expansion of business. so we choose RESTful.

All of our interface designs conform to Restful's constraints. The specific operation types of resources are all represented by **HTTP verbs**. In our interface, after processing the request, the server needs to return the status code and prompt information to the client. At the same time, in our interface, if the server has an error or the resource is unreachable, we will return an error message to the user. Our API data description is simple, and complex data is generally exchanged into **JSON.**

The advantages of our API design

- Our interface is resource-oriented, clear at a glance and self-explanatory.
- Our interface is directly based on HTTP, no other message protocols are needed, and the semantics of the HTTP protocol itself are fully utilized.
- When calling our interface to access and manipulate resources, you don’t need to consider the context or the current state, which greatly reduces the complexity.
- Our interface can be achieved with the help of rich HTTP-related middle-ware, whether it is caching or resource operation, it can be realized with the help of business-independent middle-ware.

Our Implementation Mechanism:

The data exchange between our interfaces is **JSON of RFC 4627 Standard**. At the same time, we use **SpringBoot** to write the interface and return the package results to achieve the back-end interface and the front-end page.

Spring Boot is a new framework provided by Pivotal team, which is designed to simplify the initial construction and development process of new Spring applications.

The advantage of of using SpringBoot：

Spring Boot, makes coding, configuration, deployment, and monitoring simple. It can be used to build projects quickly and integrate the mainstream development framework without configuration. At the same time, the project can run independently without external dependency on the **Servlet** container. It can also provide application monitoring at runtime. This greatly improves the efficiency of development and deployment.

4 Use Case Realization Based On Design Mechanisms

Combined with the **design mechanism** described above and the improved architecture, two use cases, order and refund are selected here for use case realization. Different from the sequence diagram and class diagram made in the last assignment, this time the form of boundary class, control class and entity class is not adopted, but the interface class and subsystem class are used to reflect the system architecture.

4.1 Place an Order

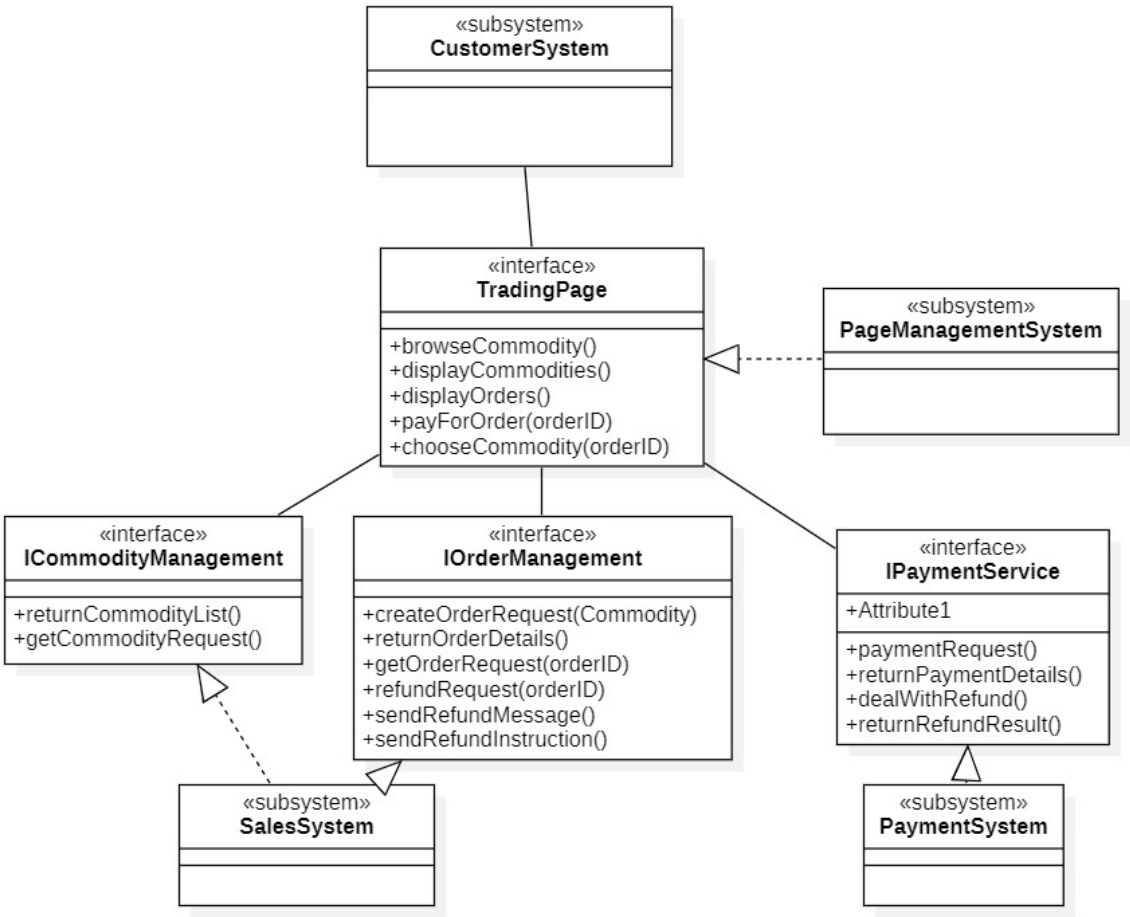

Figure 4.1 Class Diagram of the Use Case "Place an Order"

In the system, we use **JSON** messages and use **REST** and **WebSocket** services to interact in the system, and all messages will be JSON messages. When the user wants to buy second-hand goods, the user browses the desired goods in the second-hand transaction interface. The system generates the JSON message corresponding to the browsing request and sends the request to the commodity management system. The commodity management system accesses the commodity information through the **MyBatis** data persistence framework and returns the commodity list to the transaction interface. When the user wants to buy a certain product, the order management system creates an order containing the product for the user. The data persistence mechanism mentioned above is also used here, and the MyBatis is used. Finally, the user has to pay for the order, using an external Alipay or WeChat payment tool. Finally, the order information is returned to the transaction interface.

Based on the above analysis, we have drawn the sequence diagram of the use case “Place an Order" as shown in the figure below.

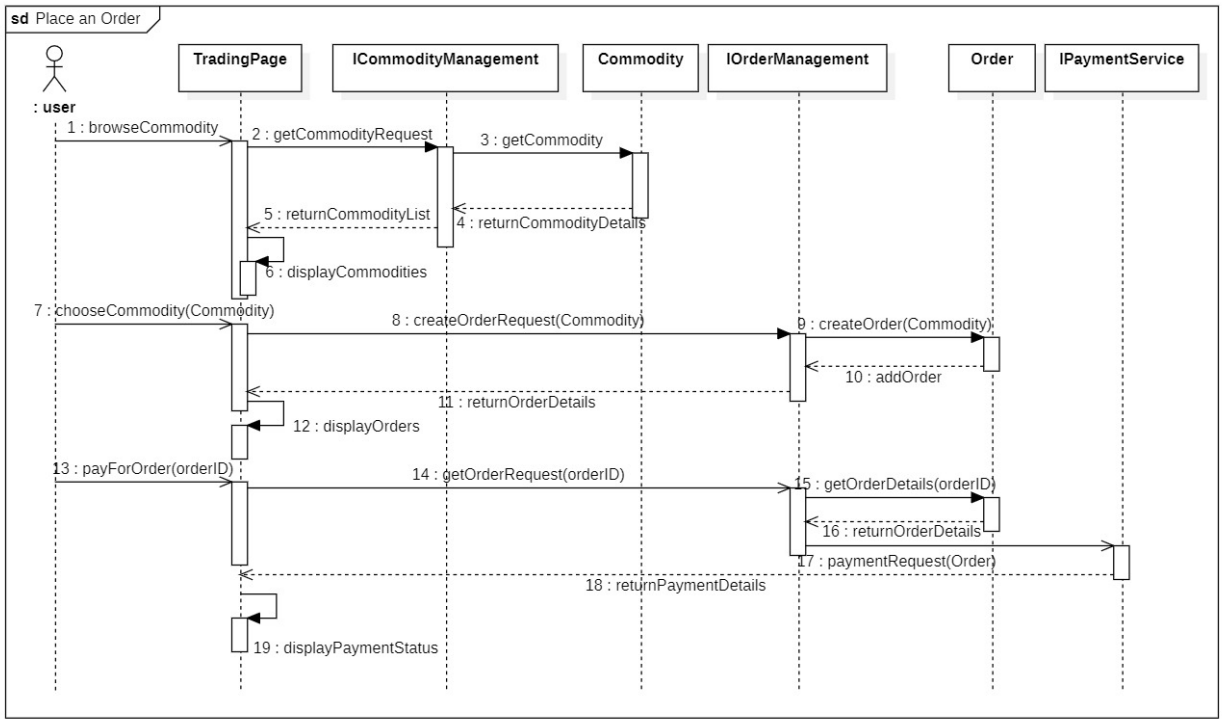

Figure 4.2 Sequence Diagram of the Use Case "Place an Order"

In addition, we have additionally drawn a communication diagram of the use case.

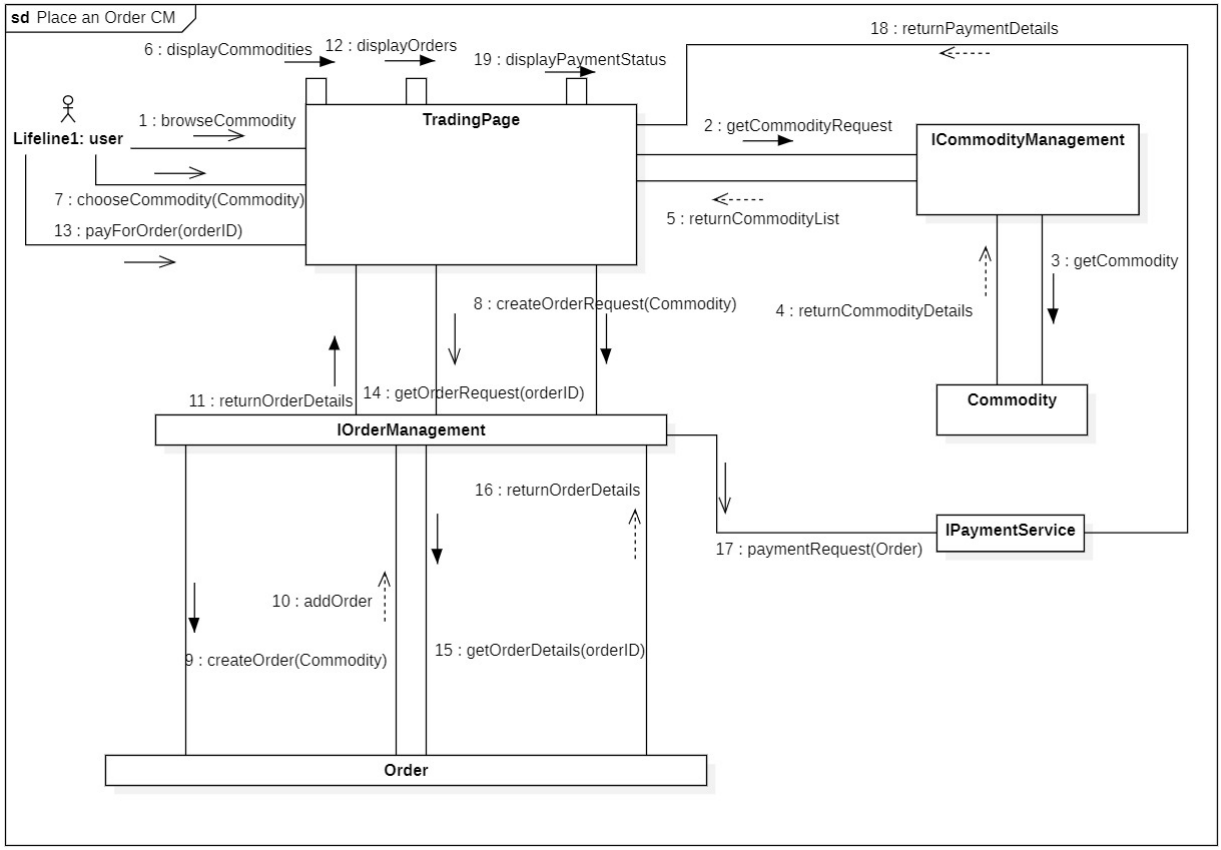

Figure 4.3 Communication Diagram of the Use Case "Place an Order"

4.2 Refund

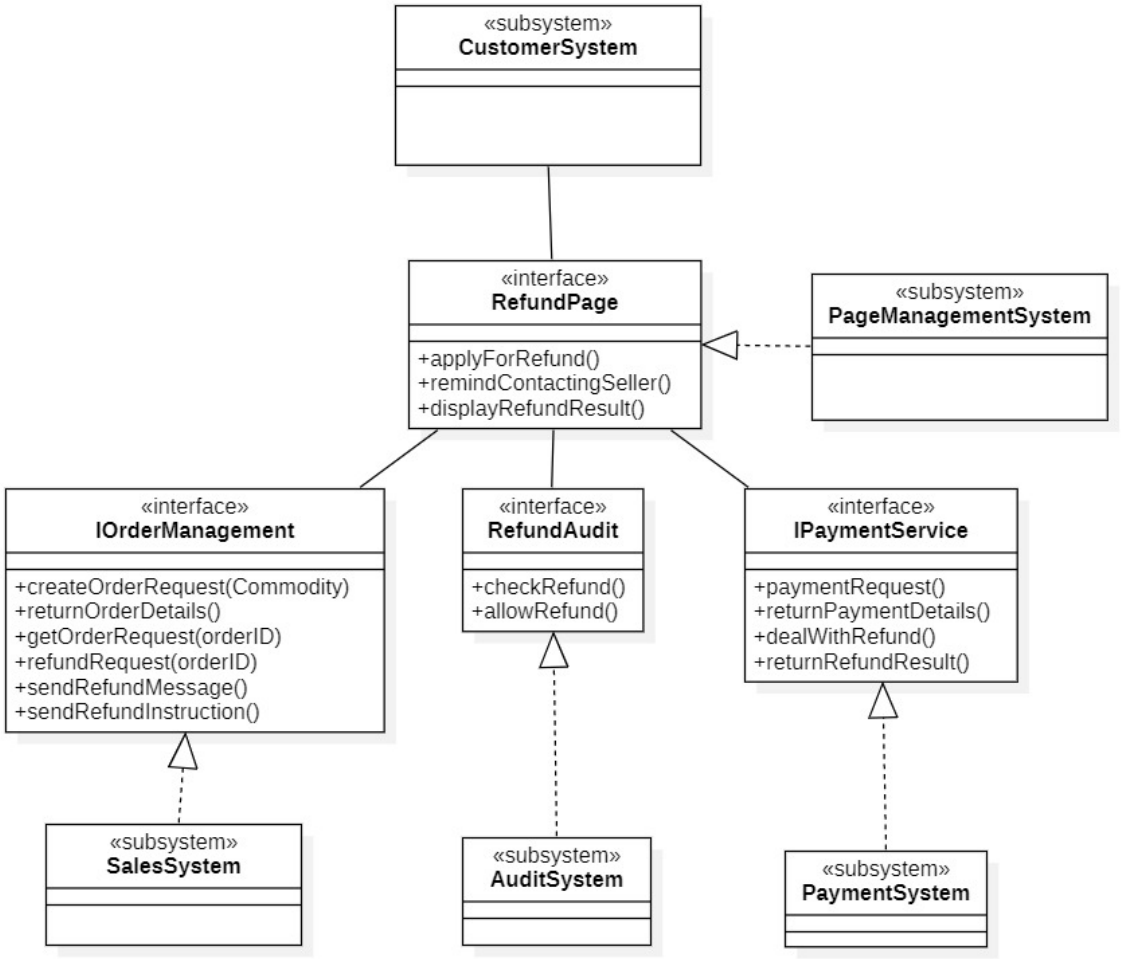

Figure 4.4 Class Diagram of the Use Case "Refund"

When a user applies for a refund, the refund page first obtains the ID of the order to be refunded, and then obtains the details of the order through the **Order Management System**. If the order has been received by the user, the system reminds the user to communicate with the seller. Otherwise, the order information is sent to the **Audit System** for review. If the basic requirements for a refund are met, the system refunds the money to the user through the **Payment System**.

Based on the above analysis, we have drawn the sequence diagram of the use case “Refund" as shown in the figure below.

In addition, we have additionally drawn a communication diagram of the use case.

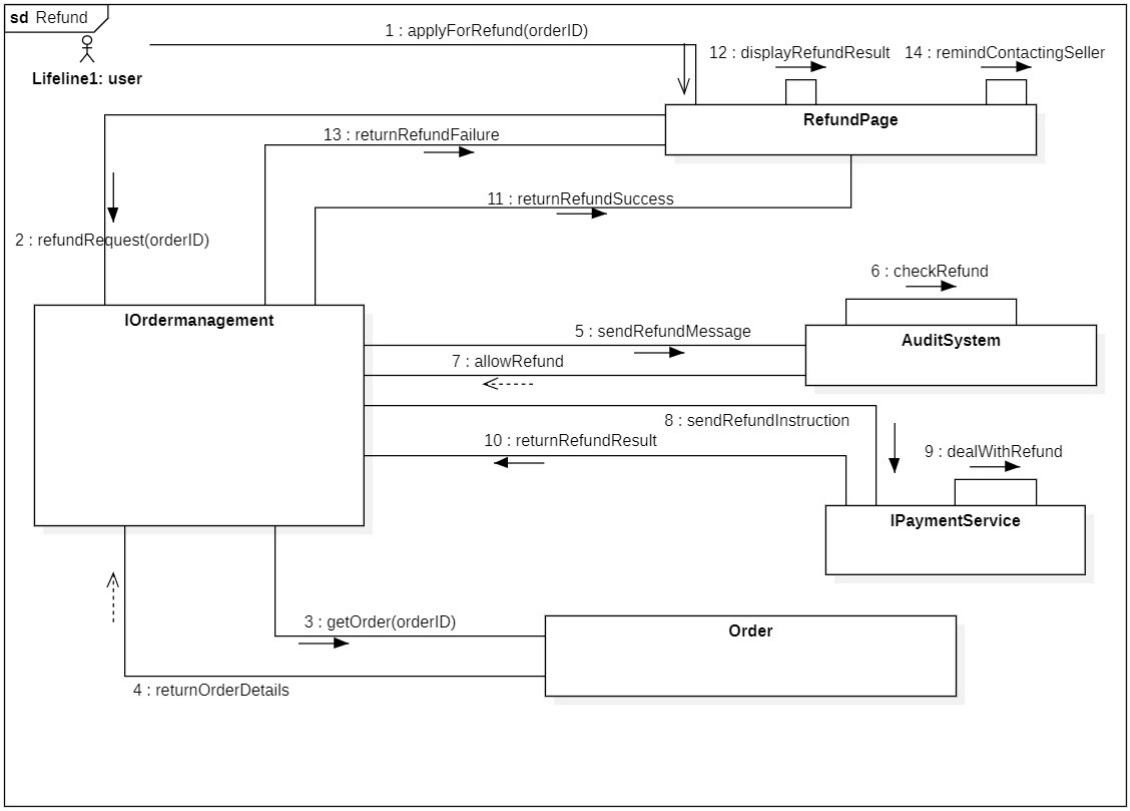

Figure 4.6 Sequence Diagram of the Use Case "Refund"

5 Architectural Style and Critical Design Decisions

5.1 Architectural Styles

We have adopted different architectural styles in different architectures. The deployment architecture adopts the **micro-service architecture style.** The microservice architecture advocates dividing a single application into a set of small services, and the services coordinate and cooperate with each other to provide users with ultimate value. Each service runs in its own process, and a lightweight communication mechanism is used between services to communicate with each other (HTTP-based RESTful API). Each service is built around a specific business, and can be independently deployed to the production environment, with the characteristics of single responsibility, lightweight communication, and independence. Since this system involves several different development teams, the technology stack used by each development team is different. In order to achieve service integration between the various subsystems, in the actual development process, each development group only needs to focus on what it is responsible for. The realization of the functional API of the microservice module can

realize the interconnection and intercommunication between services, which greatly improves the development efficiency.

In the logical architecture, we adopt the **Ports & Adapters** Architecture style and the microservice architecture style. The system interacts with the outside through the adapter, encapsulating application services and domain services inside the system. The application interacts with the outside through the "port". In the traditional layered architecture, it is easy to cross the boundaries between layers and penetrate business logic into other layers. The Ports & Adapters style solves the problem of decoupling between technology and business systems, technology and technology. This architecture comes from the design pattern, starting from the business entity service, and simplify the interface-oriented design, and the service itself realizes completeness and integrity and independence.

The **hierarchical architecture style** is adopted in the development architecture. The hierarchical system is organized into a hierarchical structure, characterized in that each layer provides services for the upper layer and uses the services of the next layer. Layers are used to isolate different concern points to cope with changes in different requirements, so that such changes can be carried out independently. In addition, business complexity and technical complexity are also isolated. It effectively simplifies the design, makes the designed system structure clearer, and facilitates the improvement of reuse ability and system maintenance ability.

5.2 Critical Design Decisions

1. Use a distributed database

If a shared database pattern is adopted, the entire architecture will become more and more rigid and lose the significance of the microservice architecture. Therefore, split the database, that is, adopt the form of distributed database. We refined it into 6 databases with different functions, among which all persistence layers of User Database are isolated from each other, and each service is responsible for it. There are many popular MQs, such as Kafka, Rabbitmq, etc. Compared with these two components, RocketMQ supports transactional messages and has the characteristics of high performance, high reliability, and distribution. In particular, it meets the management needs of high-frequency news of consumer clusters in the second-hand transaction service of our system. Therefore, under the premise of facing a large number of message requests, in order to improve the real-time performance of the system, we chose to join the RocketMQ message queue mechanism to realize the problem of timely response to messages. For the technology stack of the data layer, the first is the database management system DBMS. The current mainstream DBMSs include Oracle and MySQL. Since our system is oriented to schools, the data volume is not too large, and the use of an enterprise-level DBMS such as Oracle

may seem heavy, so we choose the lightweight MySQL as the DBMS.After selecting the database, we also need to select the tools to obtain data in the system, the most commonly used are JDBC, Hibernate, MyBatis, etc. Because MyBatis does not need to consider many details, the learning cost is smaller than Hibernate, the development mode is very small from the traditional JDBC, and the optimization is relatively easy, so it is very convenient to get started and develop projects. In addition, we are a student-oriented system, the system complexity is not very large, so we finally chose MyBatis. The following figure shows an embodiment of our use of this distributed database:

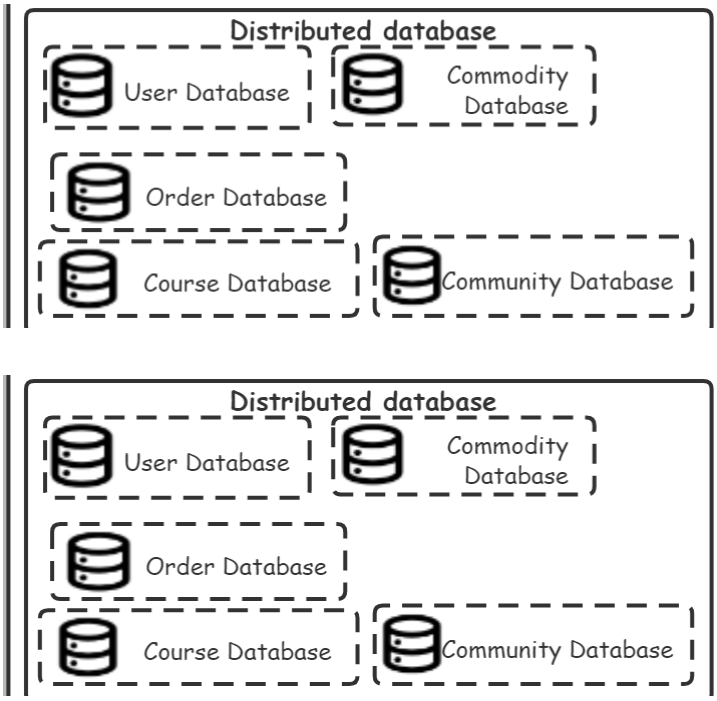

Figure 5.1 Schematic Diagram of Distributed Database (1)

In order to prevent the database from becoming a performance bottleneck and better deal with the service dependency problem caused by the implementation of the microservice framework, we adopted the distributed database system DDBS. Some databases with high access frequency introduce caching mechanisms such as MEMcache and Redis. At the same time, Redis exporter and MySQL exporter are used as indicator interfaces for cache and database, and Prometheus is used as the indicator collector.

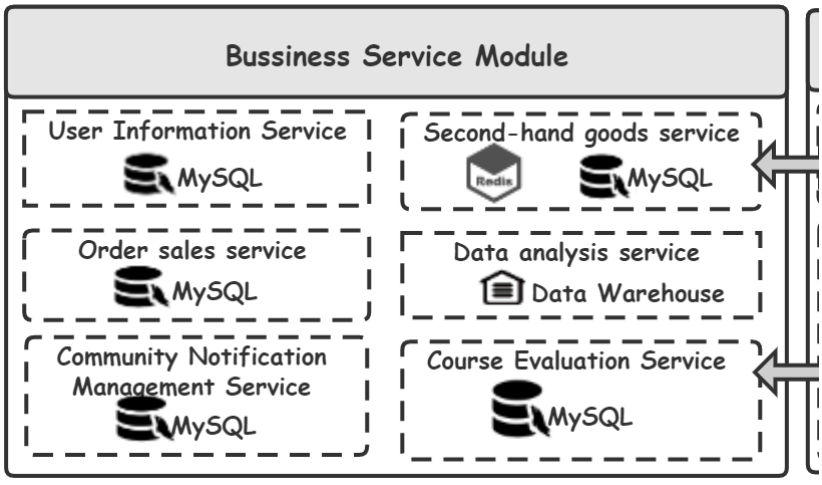

Figure 5.2 Schematic Diagram of Distributed Database (2)

2.Use microservice architecture instead of monolithic application architecture

In the deployment architecture, we use the microservice architecture for implementation. The Spring Boot framework is used for all six micro-services, and they are all registered in Spring Eureka, which is a service discovery component. Since our system adopts a microservice architecture, a very important requirement for adopting micro-services is the decoupling between various services. Using RocketMQ can reduce the avalanche phenomenon caused by the collapse of a certain function. Except for data analysis services, all microservice databases are MySQL, because we only need a lightweight database. Considering the massive amount of data that analyzes user behavior to make business decisions, we apply Redshift to implement data analysis services.

For our system, the main body of information required by users—second-hand transaction information, course information, association information, etc.—is complicated. At the same time, the information and services required by different student users need to be diversified. Therefore, Spring Cloud Netflflix, which has a finer service granularity and is more conducive to resource reuse, is more suitable for our system. In terms of improving the interface call problems brought by the microservice framework, we build the corresponding API gateway as a unified platform for providing service interfaces. For this, we use Eureka (service discovery), Hystrix (circuit breaker), Zuul (intelligent routing) and Ribbon (client load balancing) services provided by Spring Cloud Netflix.

6 Design Patterns

In the previous stage, we carried out a simple use case realization. At this stage, we used several design patterns to optimize the design in the use case realization part. Our classes and interfaces mainly use **Strategy Pattern**, **Proxy Pattern**, and **Bridge Pattern**.

6.1 Strategy Pattern

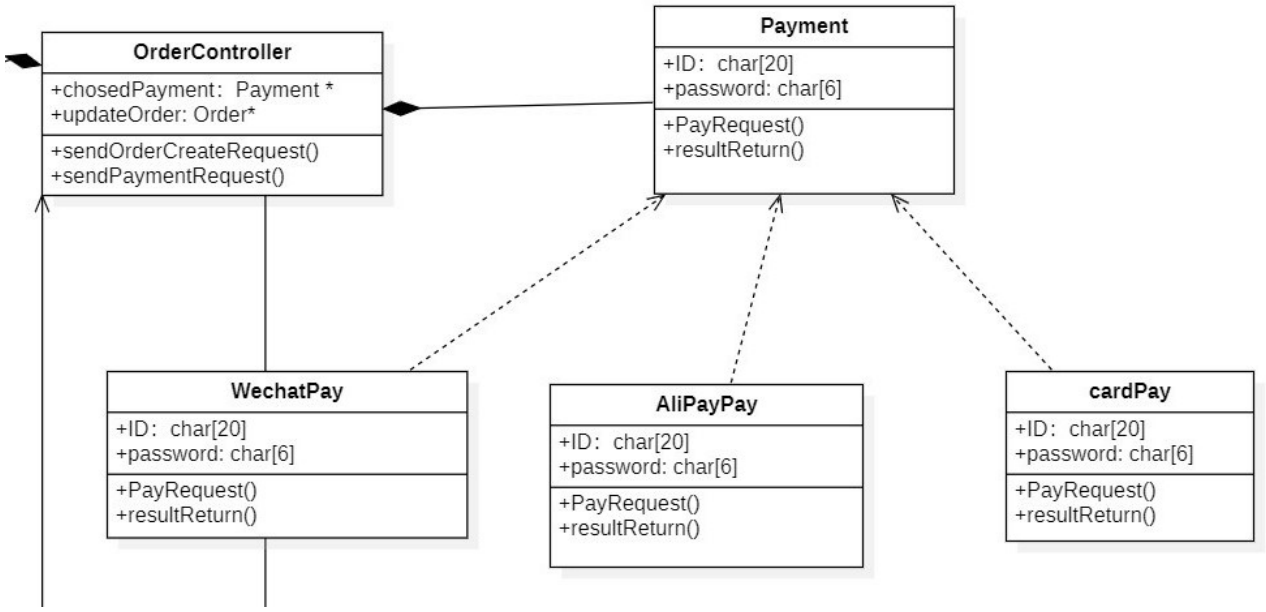

Figure 6.1 Optimization of Class Diagrams in the Mode of Strategy pattern

Strategy Pattern defines a series of algorithms, and **encapsulates** each algorithm so that they can be replaced with each other, and the change of the algorithm will not affect the customers who use the algorithm. The strategy mode belongs to the object behavior mode. It encapsulates the algorithm, separates the business using the algorithm and the algorithm, and constructs different objects to manage these algorithms.

Take our **OrderController class** as an example. One of the functions of the OrderController class is to pay for orders. There are three traditional payment methods for orders, **cardPay, WechatPay, and AliPayPay.** These three methods all perform the payment function, so three of them are inherited from the Payment class, and the Payment pointer is added to the data member of the OrderController class. This structure conforms to the strategy model, and there is no need to change the existing classes when updating new payment methods in the future. This makes the code optimized, in line with the principles of opening and closing in the design pattern and the principle of composite reuse.

In our system, the main advantages of using the strategy mode are as follows:

- Multiple conditional statements can be avoided, which means the code is easier to maintain.
- The strategy pattern provides a series of reusable algorithm families. A reasonable structure can maximize the transfer of the common code of the algorithm family to the parent class, thereby avoiding duplicate codes and reducing the workload of development.
- The strategy pattern can provide different realizations of the same behavior, and customers can choose different methods according to different time or space requirements.

- The strategy mode provides perfect support for the principle of opening and closing, and can flexibly add new algorithms without modifying the original code.
- The strategy mode puts the use of the algorithm in the environment class, and the realization of the algorithm moves to the specific strategy class, realizing the separation of the two.

6.2 Proxy Pattern

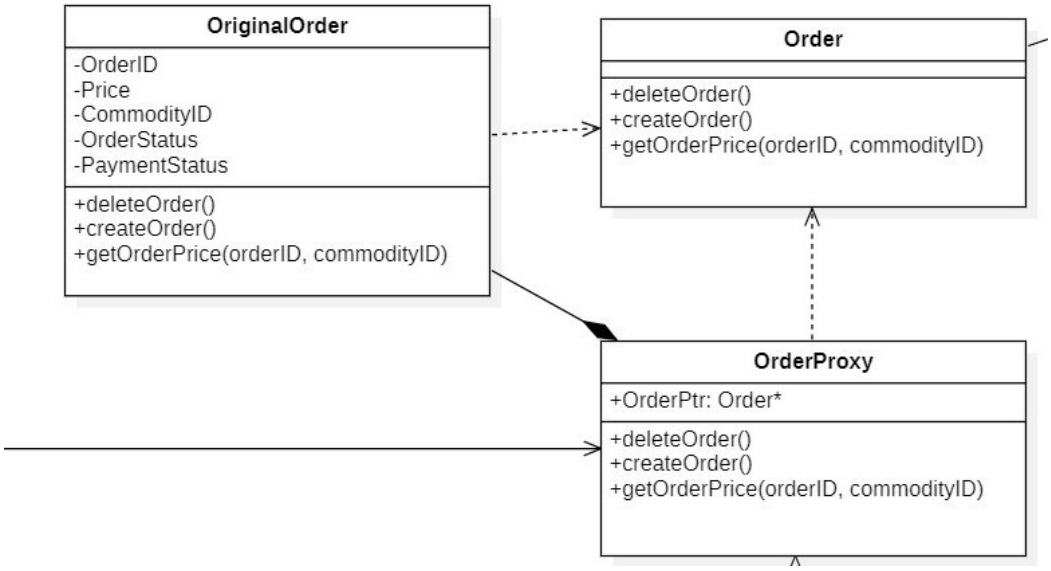

Figure 6.2 Optimization of Class Diagrams in the Mode of Proxy pattern

We use **Proxy Pattern** to provide users with access control to confirm the user's authority and data security.

Our order class wants to achieve the following functions: Once an order is created, only the person who created the order can modify the data in the order, and other people cannot modify it. Then we need to control external access to orders. Only users who meet the conditions can access, and those who do not meet the conditions cannot access. We use OrderProxy to proxy OriginalOrder. This makes it necessary to pass the OrderProxy when accessing OriginalOrder from outside, and the authorization verification is realized in the OrderProxy.

The idea of using the Proxy Pattern in the class diagram has the advantage that the Proxy Pattern can separate the proxy object from the real target object being called. To a certain extent, the coupling degree of the system is reduced, and the scalability of the system is improved. It can play a role in protecting the target object. It can also enhance the function of the target object.

6.3 Bridge Pattern

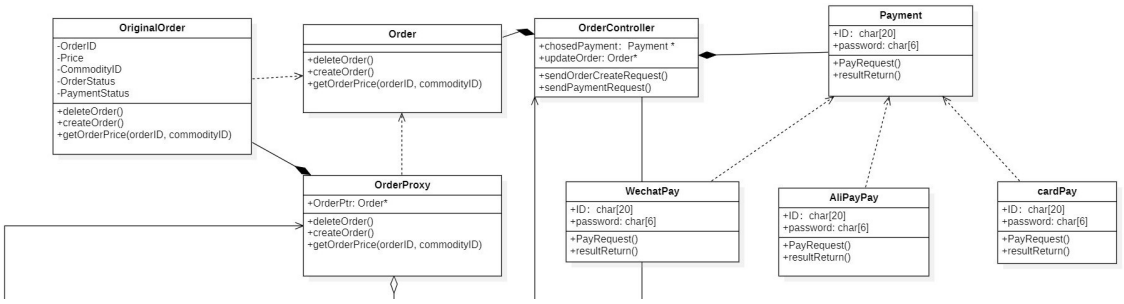

Figure 6.3 Optimization of Class Diagrams in the Mode of Bridge pattern

We all know that inheritance can omit the writing of a large part of the code. But when our code code uses inheritance a lot, it often leads to a class that often has many reasons that can cause its change.

Take our OrderController class as an example. The OrderController class has two reasons for its changes, namely Order and Payment. Separate these two factors and combine them with the OrderController class, which implements the bridge pattern. After using the bridge mode, the causes of class changes are separated, so that the number of new classes that need to be written is greatly reduced. The code of the same function does not need to be written multiple times, which is convenient for other programmers to read and understand. When a new function needs to be added, it has no effect on the original function.

The advantages of using the bridge pattern in our class diagram are: the bridge pattern conforms to the "single responsibility principle", and the number of classes is relatively small compared to multiple inheritance. The bridge pattern is a better solution than the multi-layer inheritance scheme. It greatly reduces the number of subcategories. At the same time, the bridging mode improves the scalability of the system. Any expansion of one of the two changing dimensions does not require modification of the original system and conforms to the "opening and closing principle."

7 Prototypes——Demo of Spring Security

For the functions we mentioned earlier, we took a use case and implemented them roughly. In our system, there are many different types of users, such as users and administrators, and users also have different identities, such as sellers, buyers, etc. We should ensure that everyone has normal access to their services and resources, and the system should provide a comprehensive user-friendly access control mechanism for the entire web-based application. Generally in a system, security consists of two main operations. The first is called "authentication" and establishes a body for the user to

declare. A principal is generally a user, device, or other system that can perform an action in the system. The second is called "authorization", which refers to whether a user can perform an action in the application, before reaching the authorization judgment, the identity of the body has been established by the authentication process. Our overall system is based on a micro-services architecture and uses the Spring Boot framework, so we chose Spring Security as the authentication and access control framework. The login use case is the user's entry point to the system. Since none of the members had any previous knowledge of Java, in order to help us familiarize ourselves with Spring Boot and Spring Security, we implemented a simple implementation according to the main flow of the login use case we designed earlier.

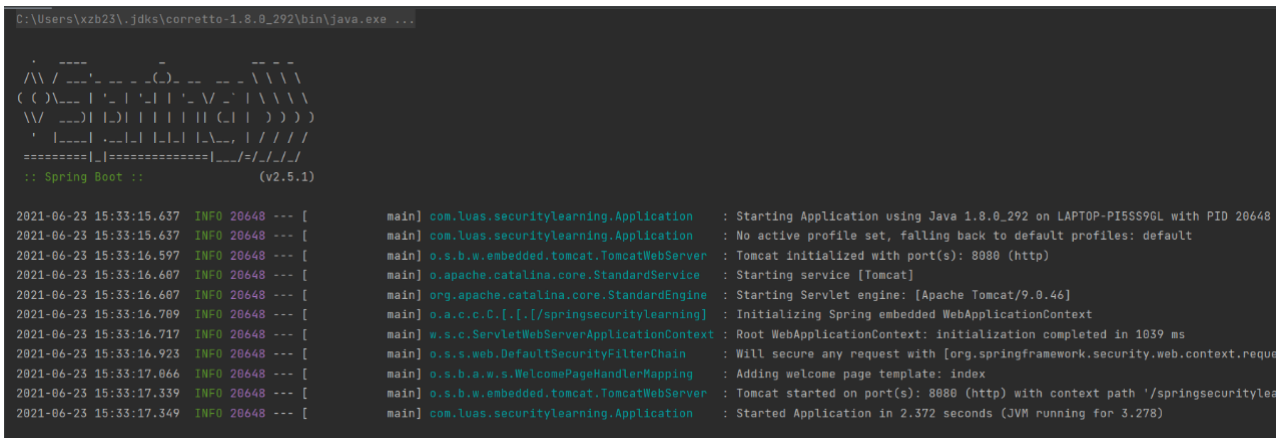

Figure 7.1 Demo of Spring Security

Simple login Page after using Spring Security

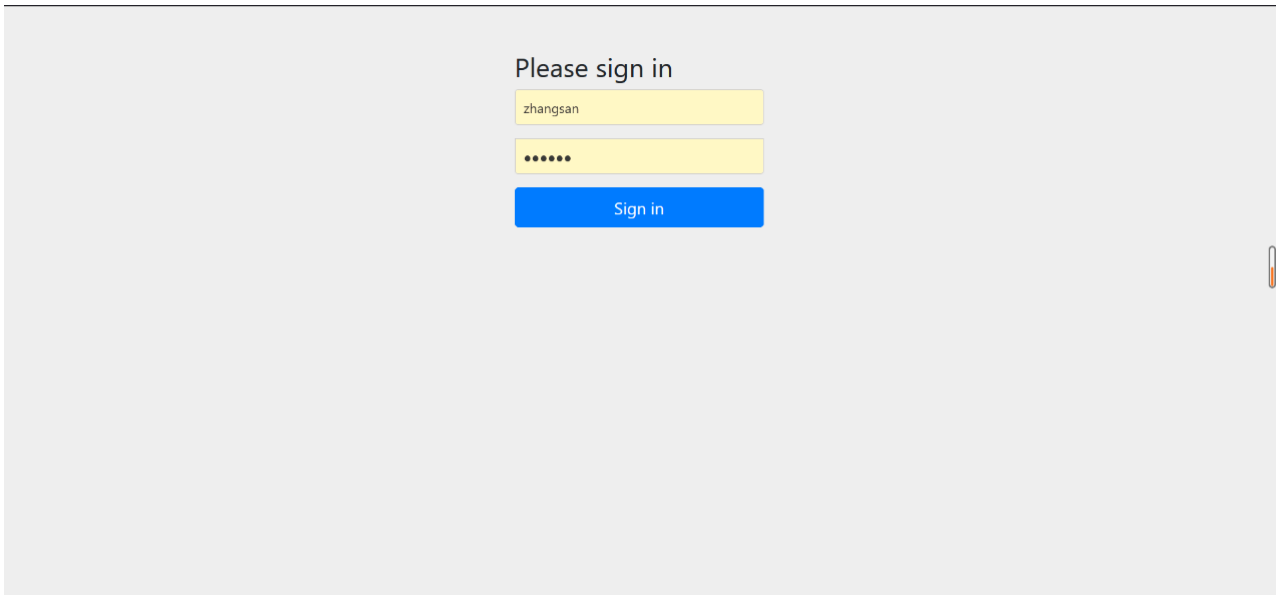

Figure 7.2 The Demo Page of Successfully Login In

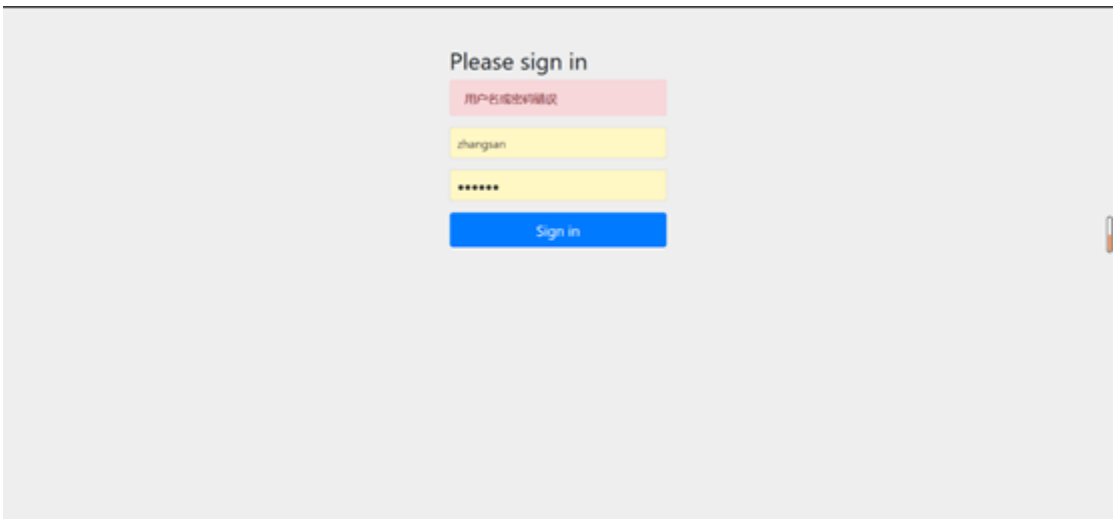

Figure 7.3 The Demo Page of Unsuccessfully Login In

Below is a screen shot of part of our core code:

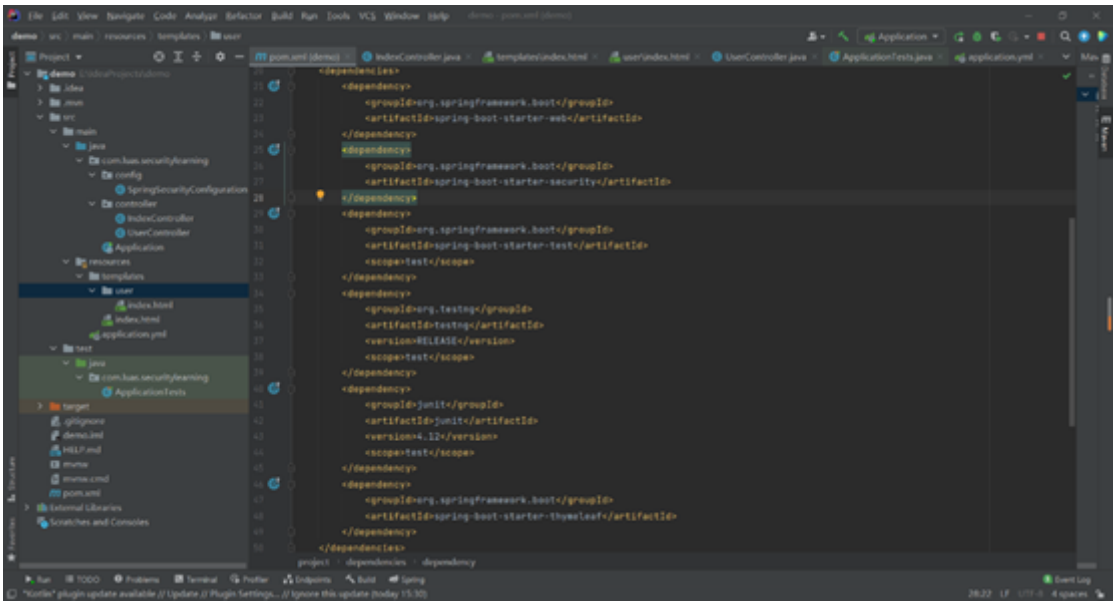

Figure 8.4 Part of the Code

8 Open Issues of the Design Model

Regarding the problems existing in our current design model, we will elaborate on the following aspects：

8.1 Micro-Services

- **Increased resource usage**: The initial investment to run these micro-service architecture applications will be relatively large, because all micro-services should have their own running containers, which also requires more CPU and memory. In addition, the use of middle-ware will also bring a relatively large base investment.
- **Increased network communication overhead**: The network overhead generated by the distributed system is much more than that of the stand-alone application, and the internal call cannot be simply changed to the distributed call, which suffers a lot. Under the microservice architecture, components that run independently need to communicate with each other over the network. In this system, a more reliable and fast network connection is required.
- **Encoding and decoding**: If you use micro-services, then each independent service needs to exchange information, which is inseparable from encoding and decoding. This will not only create a security problem, but also a hindrance to performance.
- **Testing**: Testing the application of the microservice architecture is definitely much more difficult than the monolithic application. I have a deep understanding that the integration testing process is a nightmare.
- The complexity of the distributed system requires a high technical threshold

8.2 Design Patterns

Some problems with the strategy mode:

The client must understand the difference between all strategy algorithms in order to select the appropriate algorithm class at the right time. The abuse of the strategy pattern can easily cause multiple inheritance, affecting more than one change dimension of each class, increasing the difficulty of system maintenance

9 Project Self-reflections

First, summarize the **overall progress** of our project. First of all, starting from **the analysis of project requirements**, we brainstormed to determine the project theme and core business areas as a campus community platform aimed at campus student groups, with the idle sale of second-hand goods as the core business, supplemented by several sub-businesses. After determining the requirements, we conducted a requirement analysis based on UML, including use case analysis, drawing use case diagrams and use case specifications, activity diagrams, etc., and finally drawn a simple prototype page of

the system. After completing the demand analysis, our project has advanced to the analysis stage. We drew sequence diagrams of several use cases, obtained several analysis classes, drew more abstract class diagrams, and conducted preliminary architectural analysis. Finally, we analyzed the design model, further optimized the architecture, and designed the API interface specification. During the design phase, we steadily advanced, proposed several design mechanisms and key design decisions, and adopted several design patterns throughout the process to further optimize Use case realization.

The whole process is based on teamwork, so we will reflect on ourselves from the perspective of the team. The first is the architectural part of the project. Due to the lack of development framework, computer network and other related knowledge, we often get into a bottleneck for many times due to lack of professional knowledge in the architecture part. Therefore, we "stand on the shoulders of giants" by consulting teachers, checking information online, etc., and finally design a set of architecture systems that are most suitable for us. But because the whole process is done by a team, we will inevitably encounter the problem of **information mismatch**. What's more serious is that in the analysis reports of different aspects, we may have a number of mismatches in the technology stack. But the advantage in this regard is that we finally reached the development framework and design decisions that are most suitable for our system after discussion. This is commendable.

Followed by the UML part. We have adopted a variety of UML forms in this process, such as use case diagrams, activity diagrams, sequence diagrams, communication diagrams, class diagrams, etc. In this process, each member participates in the learning and development of each UML form. Finally, a unique style of UML expression form was formed in the project, and at the same time, I have a relatively proficient grasp of the UML language specification.

10. Contribution of Team members

STUDENT NAME	STUDENT NUMBER	CONTRIBUTIONS
Qiao Liang	1853572	1. Update the logical structure and draw the development structure 2. API interface Specification 3. Description of framework and platform 4. Document layout

STUDENT NAME	STUDENT NUMBER	CONTRIBUTIONS
Zhibo Xu	1854062	1. Data persistence design mechanism 2. Use case realization based on design mechanism 3. Prototype design of Spring Security 4. Participate in the design of API interface
Zhenyu Li	1854117	1. Update of deployment architecture 2. Specification of Third-party interface call
Yixi Zheng	1953803	1. Writing architectural styles and design decisions 2. Participate in the design of API interface
Wenchao Chen	1954106	1. Design Patterns 2. Distributed design mechanism 3. Consider the deficiencies in system design