A Shipping and Transportation Web application Development with Spring MVC and More

Assignment 1

学院：软件学院

专业：软件工程

项目成员：

20301093 赵天舒

20301115 祁麟

指导教师：曾立刚

创建时间：2023.4.26

目录

1. 需求分析

本系统是一个航运和运输 Web 应用程序，主要目的是为用户提供货物运输的服务，包括货物的运输、物流跟踪、订单管理等功能。本系统主要面向以下用户：

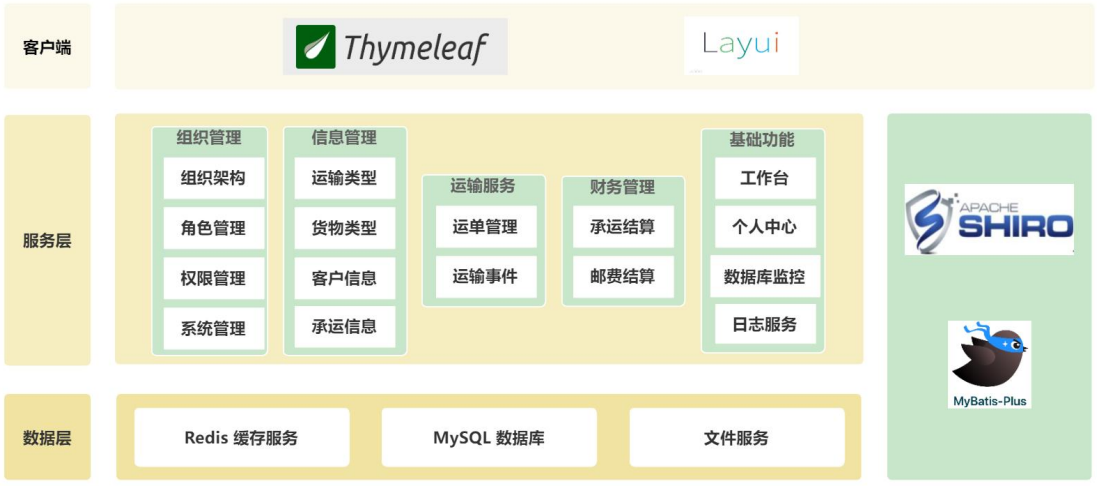
管理员：具备系统所有功能权限；

业务员：负责运输信息管理、运单管理、财务管理等。

在本系统中，管理员需要进行组织管理，包括组织架构信息管理、角色管理、权限管理与系统管理。业务员需要管理运输信息，包括运输类型、货物类型、客户信息、承运信息的管理；需要管理运单，包括运单的创建、状态管理、运输事件管理等；需要管理财务收支，包括承运结算与邮费结算等。

2. 系统架构设计

本系统采用前后端分离的架构，后端使用 SpringBoot + Mybatis-Plus + Apache Shiro + Redis，前端使用 Thymeleaf + Layui。 系统架构图如下所示：



客户端：主要负责用户界面展示和用户交互，与后端通过 RESTful API 进行通信；

服务层：主要负责处理业务逻辑和数据存储，与前端通过 RESTful API 进行通信；

数据层：主要用于存储用户、货物、运单等信息；

3. 数据库设计

本系统中需要保存的数据包括用户信息、货物信息、订单信息等。其中承运信息表、客户信息表、运单表的数据库设计如下所示：

3.1 承运信息表

字段名	数据类型	注释
id	int	承运人 id
name	varchar(100)	公司名
contact_name	varchar(50)	联系人姓名
email	varchar(100)	邮箱
phone	varchar(20)	电话

3.2 客户信息表

字段名	数据类型	注释
id	int	客户 id
name	varchar(100)	姓名
email	varchar(100)	邮箱
phone	varchar(20)	手机号
company_name	varchar(100)	公司名
address	varchar(200)	地址
city	varchar(50)	城市
country	varchar(50)	国家
postal_code	varchar(20)	邮编

3.3 运单信息表

字段名	数据类型	注释
id	int	运单 id
customer_id	int	客户 id
carrier_id	int	承运方 id
send_id	int	发货人 id
shipment_type_id	int	物流类型
origin_address	varchar(255)	发货详细地址
origin_city	varchar(255)	发货地
origin_country	varchar(255)	发货地国家
origin_postal_code	varchar(255)	发货地邮编
destination_address	varchar(255)	收货详细地址
destination_city	varchar(255)	收货地
destination_country	varchar(255)	收货地国家
destination_postal_code	varchar(255)	收货地邮编
des	varchar(255)	备注
goods_id	int	物资类型
num	int	物资数量
weight	decimal(10,2)	总重量
value	decimal(10,2)	物资价值
order_status	varchar(255)	运单状态
pickup_date	datetime	发货日期
delivery_date	datetime	收货日期

4. 接口设计

本系统主要采用 RESTful API 进行通信，共包含两类接口，分别用于系统管理（sys）与运单管理（shipping），基于增删改查实现一系列系统管理的基础功能。

系统集成 knife4j 生成接口文档，启动项目后可访问 http://localhost:8080/doc.html 进行查看。

5. 系统测试

本系统目前完成了对运单管理服务、财务管理服务的单元测试，采用 JUnit+mockito+Jacoco 实现，测试用例如下表所示：

测试用例编号	Shipment -01	用例类型	服务实现层
用例目的	测试统计承运总金额的功能		
测试步骤			
1. 创建一个 Shipment 对象，并设置其各个属性的值。 2. 调用 shipmentService.saveShipment() 方法，将 Shipment 对象保存到数据库中，并获取保存后的 Shipment 对象。 3. 验证保存后的 Shipment 对象是否符合预期，包括其各个属性的值是否正确。 4. 使用 JUnit 的 Assert 断言方法，验证保存后的 Shipment 对象是否符合预期，包括其 id、originAddress、destinationAddress、weight、value、pickupDate、orderStatus 等属性的值是否正确。			
代码覆盖分析			

```
34      public Shipment saveShipment(Shipment shipment) {
35          LambdaQueryWrapper<Customer> q1 = Wrappers.lambdaQuery();
36          q1.eq(Customer::getId, shipment.getSendId());
37          Customer send = customerMapper.selectOne(q1);
38          shipment.setOriginAddress(send.getAddress());
39          shipment.setOriginCountry(send.getCountry());
40          shipment.setOriginCity(send.getCity());
41          shipment.setOriginPostalCode(send.getPostalCode());
42          LambdaQueryWrapper<Customer> q2 = Wrappers.lambdaQuery();
43          q2.eq(Customer::getId, shipment.getCustomerId());
44          Customer customer = customerMapper.selectOne(q2);
45          shipment.setDestinationAddress(customer.getAddress());
46          shipment.setDestinationCountry(customer.getCountry());
47          shipment.setDestinationCity(customer.getCity());
48          shipment.setDestinationPostalCode(customer.getPostalCode());
49          LambdaQueryWrapper<Goods> q3 = Wrappers.lambdaQuery();
50          q3.eq(Goods::getId, shipment.getGoodsId());
51          Goods goods = goodsMapper.selectOne(q3);
52          shipment.setWeight(goods.getWeight().multiply(shipment.getNum()));
53          shipment.setValue(goods.getPrice().multiply(shipment.getNum()));
54          shipment.setPickupDate(new Date());
55          shipment.setOrderStatus(0);
56          shipmentMapper.insert(shipment);
57          return shipment;
58      }
```

测试用例编号	CarrierBilling-01	用例类型	服务实现层
用例目的	测试根据运单创建承运结算记录的功能		

测试步骤

1. 创建一个 Shipment 对象，并设置其属性值。
2. 创建一个 ShipmentType 对象，并设置其属性值。
3. 使用 Mockito 框架模拟 shipmentTypeMapper 的 selectOne 方法，使其返回上一步中创建的 ShipmentType 对象。
4. 使用 Mockito 框架模拟 carrierBillingMapper 的 insert 方法，使其返回 1，表示插入数据成功。
5. 调用 carrierBillingService 的 saveCarrierBilling 方法，并传入第 1 步中创建的 Shipment 对象作为参数。
6. 使用断言方法 assertTrue 判断 saveCarrierBilling 方法返回值是否为 true。

代码覆盖分析

```
31      public boolean saveCarrierBilling(Shipment shipment) {
32          CarrierBilling carrierBilling = new CarrierBilling();
33          carrierBilling.setCarrierId(shipment.getCarrierId());
34          carrierBilling.setOrderId(shipment.getId());
35          carrierBilling.setCreateDate(new Date());
36          LambdaQueryWrapper<ShipmentRate> queryWrapper = Wrappers.lambdaQuery();
37          queryWrapper.eq(ShipmentRate::getShipmentTypeId, shipment.getShipmentTypeId());
38          queryWrapper.eq(ShipmentRate::getRelatedId, shipment.getCarrierId());
39          ShipmentRate shipmentRate = shipmentRateMapper.selectOne(queryWrapper);
40          carrierBilling.setFreightCharge(shipment.getWeight().multiply(shipmentRate.getPricePerKg()));
41          return SqlHelper.retBool(this.getBaseMapper().insert(carrierBilling));
42      }
```

测试用例编号	CarrierBilling -02	用例类型	服务实现层
用例目的	测试统计承运总金额的功能		
测试步骤			
1. 定义一个 expected 变量，用于存储预期的结果值。 2.调用 carrierBillingService 对象的 carrierCount() 方法，并将返回值赋值给 actual 变量，用于存储实际的结果值。 3.使用 Assert.assertEquals() 方法，将预期结果值和实际结果值进行比较。如果两个值相等，则测试通过，否则测试失败。			
代码覆盖分析			

```
public Double carrierCount() {
    LambdaQueryWrapper<CarrierBilling> queryWrapper = Wrappers.lambdaQuery();
    queryWrapper.eq(CarrierBilling::getState, 1);
    List<CarrierBilling> list = this.baseMapper.selectList(queryWrapper);
    Double total = 0.0;
    for(CarrierBilling c:list) {
        total += Double.parseDouble(c.getFreightCharge().toString());
    }
    System.out.println(total);
    return Double.parseDouble(total.toString());
}
```

测试用例编号	CustomerBilling-01	用例类型	服务实现层
用例目的	测试根据运单创建邮费结算记录的功能		
测试步骤			
1. 创建一个 Shipment 对象，并设置其属性值。 2. 创建一个 ShipmentType 对象，并设置其属性值。 3. 使用 Mockito 框架模拟 shipmentTypeMapper 的 selectOne 方法，使其返回上一步中创建的 ShipmentType 对象。 4. 使用 Mockito 框架模拟 customerBillingMapper 的 insert 方法，使其返回 1，表示插入数据成功。			

5. 调用 customerBillingService 的 saveCustomerBilling 方法，并传入第 1 步中创建的 Shipment 对象作为参数。 6. 使用断言方法 assertTrue 判断 saveCustomerBilling 方法返回值是否为 true。
代码覆盖分析

```
25      public boolean saveCustomerBilling(Shipment shipment) {
26          CustomerBilling customerBilling = new CustomerBilling();
27          customerBilling.setSendId(shipment.getSendId());
28          customerBilling.setOrderId(shipment.getId());
29          customerBilling.setCreateDate(new Date());
30          LambdaQueryWrapper<ShipmentType> queryWrapper = Wrappers.lambdaQuery();
31          queryWrapper.eq(ShipmentType::getId, shipment.getShipmentTypeId());
32          ShipmentType shipmentType = shipmentTypeMapper.selectOne(queryWrapper);
33          customerBilling.setPaymentAmount(shipmentType.getPrice().multiply(shipment.getWeight()));
34          return SqlHelper.retBool(this.getBaseMapper().insert(customerBilling));
35      }
```

测试用例编号	CustomerBilling-02	用例类型	服务实现层
用例目的	测试统计邮费总金额的功能		
测试步骤			
1. 定义一个 expected 变量，用于存储预期的结果值。 2.调用 customerBillingService 对象的 customerCount() 方法，并将返回值赋值给 actual 变量，用于存储实际的结果值。 3.使用 Assert.assertEquals() 方法，将预期结果值和实际结果值进行比较。如果两个值相等，则测试通过，否则测试失败。			
代码覆盖分析			

```
37      public Double customerCount() {
38          LambdaQueryWrapper<CustomerBilling> queryWrapper = Wrappers.lambdaQuery();
39          queryWrapper.eq(CustomerBilling::getState, 1);
40          List<CustomerBilling> list = this.baseMapper.selectList(queryWrapper);
41          Double total = 0.0;
42          for(CustomerBilling c:list) {
43              total += Double.parseDouble(c.getPaymentAmount().toString());
44          }
45          System.out.println(total);
46          return Double.parseDouble(total.toString());
47      }
```

6. 安全设计

本系统采用 Apache Shiro 实现 Token 角色权限认证，具体实现如下：

- 用户登录时，系统生成 Token 并保存在 Redis 中，并将 Token 返回给客户端；

- 客户端在后续请求中携带 Token，后端通过 Token 验证用户身份和权限；
- Token 具有一定的有效期，过期后需要重新登录获取新的 Token。

7. 性能优化

本系统采用 Redis 缓存 Token 信息，提高系统的性能和安全性。同时，系统还可以采用以下方式进一步优化性能：

- 使用 Nginx 等反向代理服务器，提高系统的并发处理能力；
- 使用 Redis 集群，提高系统的可用性和性能；
- 使用分布式文件系统（如 FastDFS）存储文件，提高文件上传和下载的性能；
- 对数据库进行优化，如建立索引、分表等，提高数据库的查询性能。

8. 部署方案

本系统可采用以下部署方案：

前端部署：前端页面可以部署到静态文件服务器上，如 Nginx；

后端部署：后端可以采用 Docker 容器化部署，方便快捷；

数据库部署：数据库可以采用 MySQL 或者 PostgreSQL，也可以采用 Docker 容器化部署。

9. 总结

本系统采用前后端分离的架构，后端使用 SpringBoot + Mybatis-Plus + Apache Shiro + Redis 实现 Token 角色权限认证，前端使用 Thymeleaf + Layui。系统具有良好的性能和安全性，目前处于开发第一阶段，可以基本满足用户的需求。