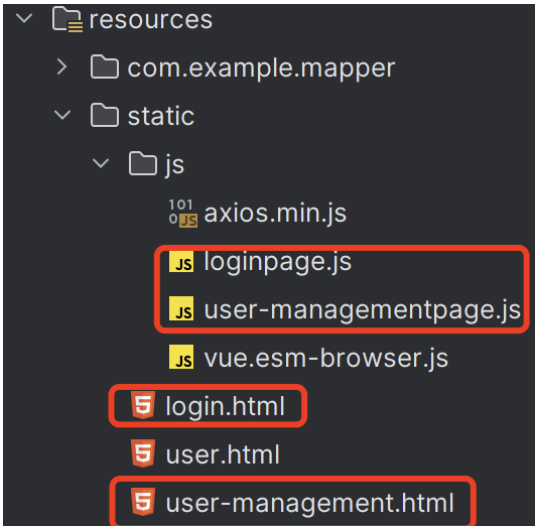
1.准备前端页面及JS文件及 安装插件MyBatisX



2.参考接口文档

https://share.note.youdao.com/s/XxIM4jO7

3.熟悉Restful风格

REST（Representational State Transfer），表述性状态转换，它是一种软件架构风格。

传统URL风格如下：

- http://localhost:8080/user/getById?id=1 GET：查询id为1的用户
- http://localhost:8080/user/saveUser POST：新增用户
- http://localhost:8080/user/updateUser POST：修改用户
- http://localhost:8080/user/deleteUser?id=1 GET：删除id为1的用户

【不足】

原始的传统URL，定义比较复杂，而且将资源的访问行为对外暴露出来了。

对于开发人员来说，每一个开发人员都有自己的命名习惯，就拿根据id查询用户信息来说的，不同的开发人员定义的路径可能是这样的：getById，selectById，queryById，loadById... 。

每一个人都有自己的命名习惯，如果都按照各自的习惯来，一个项目组，几十号或上百号人，那最终开发出来的项目，将会变得难以维护，没有一个统一的标准。

基于REST风格URL如下：

- http://localhost:8080/users/1 GET：查询id为1的用户

- http://localhost:8080/users　　POST：新增用户
- http://localhost:8080/users　　PUT：修改用户
- http://localhost:8080/users/1　　DELETE：删除id为1的用户

通过URL定位要操作的资源，通过HTTP动词(请求方式)来描述具体的操作。

在REST风格的URL中，通过四种请求方式，来操作数据的增删改查。

- GET： 查询
- POST： 新增
- PUT： 修改
- DELETE： 删除

如果是基于REST风格，定义URL，URL将会更加简洁、更加规范、更加优雅。

4.准备统一响应结果类Result

```
package com.example.pojo;import lombok.Data;/** * 后端统一返回结果 */@Datapublic class Result {

    private Integer code; //编码：1成功，0为失败
    private String msg; //错误信息
    private Object data; //数据
    public static Result success() {
        Result result = new Result();
        result.code = 1;
        result.msg = "success";
        return result;
        }

    public static Result success(Object object) {
        Result result = new Result();
        result.data = object;
        result.code = 1;
        result.msg = "success";
        return result;    }

    public static Result error(String msg) {
        Result result = new Result();
        result.msg = msg;
        result.code = 0;
        return result;
        }
}
```

5.查询所有用户

6.用户查询（带条件查询）

7.用户查询（根据ID查询）

8.新增用户

9.批量删除用户

10.更新用户信息

11.修改用户密码

修改信息类

12.登录

13.JWT

添加依赖

```
implementation 'io.jsonwebtoken:jjwt-api:0.12.3'
runtimeOnly 'io.jsonwebtoken:jjwt-impl:0.12.3'
runtimeOnly 'io.jsonwebtoken:jjwt-jackson:0.12.3'
```

测试

准备JWT工具类

service实现类中获取token

TokenFilter过滤器

前端代码改造

TokenInterceptor拦截器

注册配置拦截器

配置类 WebConfig

14.AOP （Aspect Oriented Programming）

AOP：Aspect Oriented Programming（面向切面编程、面向方面编程），即面向切面编程就是面向特定方法编程。

AOP的优势主要体现在以下四个方面：

- 减少重复代码：不需要在业务方法中定义大量的重复性的代码，只需要将重复性的代码抽取到AOP程序中即可。
- 代码无侵入：在基于AOP实现这些业务功能时，对原有的业务代码是没有任何侵入的，不需要修改任何的业务代码。
- 提高开发效率
- 维护方便

14.1添加依赖

```
implementation 'org.springframework.boot:spring-boot-starter-aop'
```

14.2示例：统计业务处UserServiceImpl中方法执行耗时

新建aop包-新建RecordTimeAspect切面类

14.3通过面向切面编程 记录业务层方法对应操作日志数据

- 创建数据库表operate_log

```sql
-- 操作日志表
create table operate_log(
    id int unsigned primary key auto_increment comment 'ID',
    operate_user_id int unsigned comment '操作人ID',
    operate_time datetime comment '操作时间',
    class_name varchar(100) comment '操作的类名',
    method_name varchar(100) comment '操作的方法名',
    method_params varchar(1000) comment '方法参数',
    return_value varchar(2000) comment '返回值, 存储json格式',
    cost_time int comment '方法执行耗时, 单位:ms'
) comment '操作日志表';
```

- pojo包准备的实体类

```java
package com.example.pojo;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import java.time.LocalDateTime;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class OperateLog {
    private Integer id; //ID
    private Integer operateUserId; //操作人ID
    private LocalDateTime operateTime; //操作时间
    private String className; //操作类名
    private String methodName; //操作方法名
    private String methodParams; //操作方法参数
    private String returnValue; //操作方法返回值
    private Long costTime; //操作耗时
}
```

- mapper包中新建日志操作Mapper接口 OperateLogMapper

```java
package com.example.mapper;

import com.example.pojo.OperateLog;
import org.apache.ibatis.annotations.Insert;
import org.apache.ibatis.annotations.Mapper;

@Mapper
public interface OperateLogMapper {
    //插入日志数据
    @Insert("insert into operate_log (operate_user_id, operate_time, class_name, method_name, method_params, return_value, cost_time) " +
            "values (#{operateUserId}, #{operateTime}, #{className}, #{methodName}, #{methodParams}, #{returnValue}, #{costTime});")
    public void insert(OperateLog log);

}
```

新建anno包-自定义接口-注解 @LogOperation

```java
/**
 *  自定义注解，用于标识哪些方法需要记录日志
 * @Target是原注解，修饰注解的注解。后面参数代表在方法上生效。
 * @Retention也是原注解，代表这个注解什么时候生效。后面参数代表是在运行是生效。
 * 注解内不需要定义任何属性，仅仅起到标识方法的作用
 */
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface LogOperation {
}
```

utils包中新建工具类 CurrentHolder

用于获取、设置、删除操作用户的id

```java
package com.itheima.utils;

public class CurrentHolder {

    private static final ThreadLocal<Integer> CURRENT_LOCAL = new ThreadLocal<>();

    public static void setCurrentId(Integer employeeId) {
        CURRENT_LOCAL.set(employeeId);
    }

    public static Integer getCurrentId() {
        return CURRENT_LOCAL.get();
    }

    public static void remove() {
        CURRENT_LOCAL.remove();
    }
}
```

在TokenFilter中，解析完当前登录员工ID，将其存入ThreadLocal（用完之后需将其删除）

aop包中新建AOP记录日志的切面类

```java
import com.example.anno.LogOperation;
import com.example.mapper.OperateLogMapper;
import com.example.pojo.OperateLog;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import java.time.LocalDateTime;
import java.util.Arrays;

@Aspect
@Component
public class OperationLogAspect {

    @Autowired
    private OperateLogMapper operateLogMapper;

    // 环绕通知
    @Around("@annotation(log)")
    public Object around(ProceedingJoinPoint joinPoint, LogOperation log) throws Throwable {
        // 记录开始时间
        long startTime = System.currentTimeMillis();
        // 执行方法
        Object result = joinPoint.proceed();
        // 当前时间
        long endTime = System.currentTimeMillis();
        // 耗时
        long costTime = endTime - startTime;

        // 构建日志对象
        OperateLog operateLog = new OperateLog();
        operateLog.setOperateUserId(getCurrentUserId()); // 需要实现 getCurrentUserId 方法
        operateLog.setOperateTime(LocalDateTime.now());
        operateLog.setClassName(joinPoint.getTarget().getClass().getName());
        operateLog.setMethodName(joinPoint.getSignature().getName());
        operateLog.setMethodParams(Arrays.toString(joinPoint.getArgs()));
        operateLog.setReturnValue(result.toString());
```

```
        operateLog.setCostTime(costTime);

        // 插入日志
        operateLogMapper.insert(operateLog);
        return result;
    }

    // 示例方法，获取当前用户ID
    private int getCurrentUserId() {
        // 这里应该根据实际情况从认证信息中获取当前登录用户的ID
        return 1; // 示例返回值
    }
}
```

在需要记录的日志的Controller层的方法上，加上注解 @LogOperation

16.用户信息统计

17.导入文件

18.导出文件