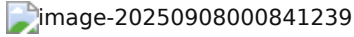


## SSM

□□□□□□

- Spring framework
- SpringMVC
- Mybatis

# webSSM SpringBoot SSMSpring



## SpringMVC

## MVC Model View Controller web

- Model 객체들끼리 상호작용을 할 수 있도록 JavaBean 인터페이스를 Model 객체에 정의
- View 객체는 Controller 객체로부터 html 렌더링을 받음
- Controller 객체는 Model 객체를 조작하고 View 객체에게 렌더링을 요청함



# Spring MVC MVC Web Spring

□□□□□

WebMvcConfigurationSupport

- **Spring MVC**





```

WebMvcConfigurer

```

- Spring Boot MVC
- \*\* WebMvcConfigurer WebMvcAutoConfiguration \*\* Spring Boot

11

- 
- 
- 
- 

## Maven

□□□□□□

555

- clean
- compile
- test
- package



- install

- 
- 
- 

image-20250601214430146

- Maven
- pom.xml

```
<!-- Apache Commons IO -->
<dependency>
  <!-- 1. Group ID:
    /
    Apache -->
  <groupId>commons-io</groupId>

  <!-- 2. Artifact ID:
    Commons IO -->
  <artifactId>commons-io</artifactId>

  <!-- 3. Version:
    2.11.0 -->
  <version>2.11.0</version>

  <!-- 4. Scope:
    compile
    testprovidedJDK -->
  <!-- <scope>compile</scope> -->

  <!-- 5. Optional:
    <exclusions> -->
  <!-- <exclusions>
    <exclusion>
      <groupId></groupId>
      <artifactId></artifactId>
    </exclusion>
```



```
        </exclusions> -->
</dependency>
<!-- 排除XML 依赖 --> </dependency>-->
```

打包方式

打包方式决定了打包后的文件结构

- jar+manifest
- pom.xml 打包方式
- web.xml 打包方式

jar

打包方式决定了打包后的文件结构

image-20250826164719979

打包方式

jar <parent>... </parent>

打包方式

- jar 打包方式
- pom 打包方式
- war 打包方式 Tomcat 打包方式
- <packing> pom </packing>

打包方式

1. pom
2. pom.xml 打包方式
  - <relativePath> </relativePath> 打包方式
3. 打包方式

打包方式

打包方式

打包方式 pom.xml <dependencyManagement> 打包方式

- 打包方式
- 打包方式
- 打包方式

打包方式

- <properties> </properties> 打包方式
- <properties> \${} </properties>

```
<properties>
  <lombok.verison> 1.9.3 </lombok.verison>
</properties>
```



```

<dependencyManagement>
  <dependencies>
    <dependency>
      <version> ${lombok.version}</version>
    </dependency>
  </dependencies>
</dependencyManagement>

```

- `<dependencies>` 라이브러리 의존성 선언
- `<dependencyManagement>` 라이브러리 버전 관리

❏

모듈화 (Modularity)

- 모듈화란 프로그램을 여러 개의 모듈로 나누는 것
- 모듈화하면 pom 파일이 복잡해지지 않음
- 모듈화 <modules> 태그를 사용

```

<modules>
  <!-- 모듈 목록 -->
  <module> 모듈1 </module>
  <module> 모듈2 </module>
</modules>

```

- 모듈화하면 프로젝트 구조가 깔끔해짐
- 모듈화하면 의존성 관리가 쉬워짐

❏

버전 관리

- 버전 관리란 소프트웨어의 버전을 관리하는 것
- 버전 관리하면 릴리스 (Release)와 스냅샷 (Snapshot)을 관리할 수 있음

❏

- 버전 관리 1.0-SNAPSHOT
- RELEASE 버전은 배포할 때만 사용
- SNAPSHOT 버전은 개발할 때만 사용

❏

Maven 설정 (Maven Settings)

- settings.xml 파일

```

<!-- settings.xml -->
<servers>
  <!-- releases -->
  <server>
    <id>maven-releases</id>

```



```

        <username>admin</username>
        <password>admin</password>
    </server>

    <!-- snapshots 快照 -->
    <server>
        <id>maven-snapshots</id>
        <username>admin</username>
        <password>admin</password>
    </server>
</servers>

```

- pom 文件配置

```

<distributionManagement>
    <repository>
        <id>maven-releases</id>
        <url>.....</url>
    </repository>
    <snapshotRepository>
        <id>maven-snapshots</id>
        <url>.....</url>
    </snapshotRepository>
</distributionManagement>

```

- settings 文件配置 mirrors 和 profiles

```

<!-- settings.xml -->
<!-- 配置镜像和快照仓库 -->
<mirrors>
    <mirror>
        <id>maven-public</id>
        <mirrorOf>*</mirrorOf>
        <url>.....</url>
    </mirror>
</mirrors>

```

```

<profile>
    <id>allow-snapshots</id>
    <activation>
        <activeByDefault>true</activeByDefault> <!-- 默认激活的 Profile -->
    </activation>
    <repositories>
        <repository> <!-- 快照仓库 -->
            <id>maven-public</id> <!-- ID -->
            <url>http://192.168.150.101:8081/repository/maven-public/</url> <!-- URL -->
        </repository>
    </repositories>
    <releases> <!-- 是否允许发布 -->
        <enabled>true</enabled> <!-- 是否允许发布 -->
    </releases>

```



```

        </releases>
        <snapshots>                <!-- 快照SNAPSHOT -->
            <enabled>true</enabled> <!-- 是否启用快照 -->
        </snapshots>
    </repository>
</repositories>
</profile>

```

## SpringBoot

- Spring框架的基石Spring Framework

项目依赖lombok

- 使用 @Data 简化getter/setter
- 使用 @AllArgsConstructor 简化构造器
- 使用 @NoArgsConstructor 简化构造器

### Http-相关

- SpringBoot使用Tomcat容器web容器
- Tomcat使用HTTP协议处理请求HttpServletRequest

项目依赖

- 使用 @RestController 简化控制器
  - 使用@Bean注解 @RestController("adminShopController") 标注Bean
- 使用 @RequestMapping 简化请求映射
  - 使用value 或 method 标注
  - @PostMapping 标注post请求
- 使用 @PathVariable 标注路径变量 /depts/{id}
- 使用 @RequestParam 标注请求参数 /depts?id=123
  - 使用@RequestParam 标注
  - 使用 MultipartFile 标注文件
- 使用 @DateTimeFormat(pattern="yyyy-MM-dd") 标注日期格式
- 使用 @Valid 标注验证

```

package com.example.demo;

import jakarta.servlet.http.HttpServletRequest;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class RequestController {
    @RequestMapping("/request")
    public String request(HttpServletRequest request) {
        // 1. 请求参数
    }
}

```



```

String method = request.getMethod();
System.out.println("method:" + method);
// 2. 获取url
String url = request.getRequestURL().toString(); // 获取StringBuffer
System.out.println("url:" + url);

String uri = request.getRequestURI(); // 获取URI
System.out.println("uri:" + uri);
// 3. 获取协议
String protocol = request.getProtocol();
System.out.println("protocol:" + protocol);
// 4. 获取参数
String name = request.getParameter("name");
String age = request.getParameter("age");
System.out.println("name:" + name + ", age:" + age);
// 5. 获取头
String accept = request.getHeader("Accept");
System.out.println("Accept:" + accept);

return "OK";
}
}

```

- `@RequestBody` 接收JSON数据

```

@RestController
public class UserController {

    @PostMapping("/login")
    public ResponseEntity<String> login(@RequestBody User user) {
        // Spring 使用JSON 接收 User 对象
        System.out.println("username: " + user.getUsername());
        System.out.println("password: " + user.getPassword());
        System.out.println("email: " + user.getEmail());
        System.out.println("age: " + user.getAge());

        return ResponseEntity.ok("login success");
    }
}

```

## Http-状态

- 状态码 3xx 表示重定向
- 状态码 200 表示成功
- 状态码 500 表示服务器内部错误
- 状态码 404 表示资源不存在
- 状态码 HTTP 状态码 404 表示资源不存在 `HttpServletResponse` 类



- `ResponseEntity` `body`

```
package com.example.demo;

import jakarta.servlet.http.HttpServletResponse;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.io.IOException;

@RestController
public class ResponseController {
    @RequestMapping("/response")
    public void response(HttpServletResponse response) throws IOException {
        // 1. status
        response.setStatus(401);
        // 2. header
        response.setHeader("name", "whs");
        // 3. body
        response.getWriter().write("<h1>whs</h1>");
    }
    // ...
    @RequestMapping("/response2")
    public ResponseEntity<String> response2(){
        return ResponseEntity.status(301).header("name", "whs").body("<h1>whs</h1>");
    }
}
```

Controller

- Controller
- Service
- Dao

Springboot

Springboot IOC DI

IOC

- Bean
- ...

```
public interface EmailSender {
    void sendEmail(String to, String message);
}
```

@Component



```
public class SmtEmailSender implements EmailSender {
    @Override
    public void sendEmail(String to, String message) {
        // SMTP
    }
}
```

IOC

- `@Component`
- `@Service`

```
public interface EmailSender {
    void sendEmail(String to, String message);
}

@Service
public class XX implements EmailSender {
    @Override
    public void sendEmail(String to, String message) {
    }
}
```

- `@Repository`

```
public interface DataLoader {
}

@Repository
public class XX implements DataLoader {
}
```

- `@RestController`

```
//
@RestController
public class XX {
}
```

DI

- 
- `Service`

-



```

@RestController
public class UserController(){
    private UserService userService;

    @Autowired
    public UserController(UserService userservice){
        this.userService = userservice;
    }
}

```

- setter

```

@RestController
public class UserController(){
    @Autowired
    private UserService userService;
}

```

- setting

```

@RestController
public class UserController(){
    private UserService userService;

    @Autowired
    public void setUserService(UserService userservice){
        this.userService = userservice;
    }
}

```

Annotations

- `@Autowired` automatically injects a Bean
  - `@Component`
- `@Qualifier` specifies the Bean name `@Qualifier(bean_name)`

```

@RestController
public class UserController(){
    @Autowired
    @Qualifier(bean_name)
    private UserService userService;
}

```

- `@Primary` specifies the primary Bean

```

// @Primary
@Primary

```



```
@Service
public class UserServiceImpl2 implements UserService{

}
```

- @Resource 标注 Bean 名称 @Resource(name=bean\_name)

@Resource 与 @Autowired 区别

- @Resource 按照名称/类型匹配 Bean 名称/类型
- @Autowired 按照类型/名称匹配 Bean

异常处理

- 使用 @RestControllerAdvice 标注
- 实现 ExceptionHandler 接口
  - 实现 handleException() 方法

```
// 异常处理
@Slf4j
@RestControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler
    public Result handleException(Exception e){
        log.error(e.getMessage());
        return Result.error(e.getMessage());
    }
}
```

## JDBC

JDBC API

步骤

1. 加载驱动
2. 获取连接
3. 创建SQL语句
4. 执行SQL
5. 关闭连接

```
import java.sql.DriverManager
public class Jdbc{
    public void test(){
        // 1. 加载驱动
        Class.forName("com.mysql.jdbc.Driver");
        // 2. 获取连接
        Connection connection = DriverManager.getConnection(URL,username,password);
        // 3. 创建SQL语句
        Statement statement = connection.createStatement();
        // 4. 执行SQL语句
        statement.executeUpdate(sql);
    }
}
```



```

        // 5. 關閉
        statement.close();
        connection.close();
    }
}

```

## SQL

- `ResultSet` 物件：`ResultSet rs = statement.executeQuery()`
  - `next()` 物件：`boolean`
  - `getXxx()` 物件：

```

String sql = "SELECT * FROM user";

stmt = conn.prepareStatement(sql);

ResultSet rs = stmt.executeQuery();
while(rs.next()){
    User user = new User(
        rs.getInt("id"),
        rs.getString("username"),
        rs.getString("password"),
        rs.getString("name"),
        rs.getInt("age")
    );
    System.out.println(user);
}

```

## SQL

- 物件：
- SQL
- 物件：

## Mybatis

物件DAO物件JDBC

物件

- `application.properties` 物件
  - 物件

物件

- 物件mybatis物件@Mapper 物件
  - 物件Mapper物件Spring物件Bean
  - 物件
- 物件sql物件mybatis物件



```
// 测试
@Mapper
public interface UserMapper{
    @Select("select * from user")
    public List<User> findAll();
}

// 测试
public class Test{
    @Autowired
    private UserMapper usermapper;
    public void testQuery(){
        List<User> list = usermapper.findAll();
    }
}
```

- application.properties
- sqlIDEA

□□□□□□□□

- `@Results` & `@Result` annotations
  - `Mapper` annotations

```
@Results({
    @Result(column="update_time",properties="updateTime")
})
```

- ☐ ☐ ☐ ☐ ☐ ☐

```
@Select("select id,name,create_time createTime,update_time updateTime from dept order by update_time desc")
```

- `create_time` `createTime`

```
map-underscore-to-camel-case: true
```

□ □ □ □ □

- jdbcConnection
- mybatis  
◦ mybatis

□ □ □ □ □ □ □ □ □ □

- **Datasource** `DataSource`



- `getConnection()` 메서드

예제

- Spring에서 **Hikari** 데이터베이스 연결 풀을 사용

예제

예제

- `#{id}` 형식의 파라미터를 사용
- `Integer` 타입의 파라미터를 DML 쿼리에서 사용

```
@Mapper
public interface UserMapper{
    @Delete("delete from user where id = #{id}")
    public void deleteById(Integer id);
}
```

예제

- 데이터베이스 연결 풀을 사용

```
@Mapper
public interface UserMapper{
    @Insert("insert into user(username,password,name) values(#{username},#{password},#{name})")
    public void insert(User user);
}

public class Test{
    @Autowired
    private UserMapper userMapper;

    public testInsert(){
        User user = new User("1",123,"2");
        userMapper.insert(user);
    }
}
```

예제

- 데이터베이스 연결 풀을 사용

```
@Mapper
public interface UserMapper{
    @Update("update user set username = #{username},password = #{password},name = #{name} where id = #{id}")
    public void update(User user,Integer id);
}

public class Test{
```



11

- ```
// 测试
@Mapper
public interface UserMapper{
    @Select("select * from user where username=#{username} and password=#{password}")
    public User find(@Param("username")String username,@Param("password")String password);
}

// 测试
public class Test{
    @Autowired
    private UserMapper usermapper;
    public void testQuery(){
        User user = usermapper.find("root","123456");
    }
}
```

```
select e.*,d.name deptName from emp e left join dept d on d.id = e.dept_id
where
    e.name like concat('%',{name},%')
    and e.gender = {gender}
    and e.entry_date between {begin} and {end}
order by update_time desc
```

□□□□□□□□ / □□□□□□□□



```
// 测试
@Mapper
public interface UserMapper{
    public List<User> findAll();
}
```

- 测试数据库连接
- id 为SQL语句的标识符
- resultType 返回结果的数据类型

```
<mapper namespace="com.zxb.mapper.UserMapper">
    <select id = "findAll" resultType="com.zxb.pojo.User">
        select id, username, password,name,age from user
    </select>
</mapper>
```

测试

测试XML配置文件

- application.properties 配置XML配置文件
  - mybatis.mapper-locations=classpath:mapper/\*.xml 指定mapper文件XML文件
  - 测试 src/main/resources 指定 src/main/java 指定 .java 文件
- 测试MyBatis与Java的交互
  - type-aliases-package=com.sky.entit 指定 com.sky.entity 指定数据库表

测试 mybatisx 测试

SQL

测试SQL语句

- <if> 测试true SQL
  - test 测试
- <where> 测试 where 测试 and 测试 or
  - > 测试
  - < 测试 xiao'yu
- <foreach> 测试
  - collection 测试
  - item 测试
  - separator 测试
  - open 测试
  - close 测试
- <set> 测试set 测试



테이블

테이블을 생성하는 방법

- `@Options` 옵션
  - `useGeneratedKeys` 옵션 `true`
  - `keyProperty` 옵션 지정

```
@Options(useGeneratedKeys = true, keyProperty = "id")
// 테이블 emp에 id를 생성
void insert(Emp emp);
```

테이블

테이블에 `user` 테이블에 `department` 테이블

테이블

- 테이블 생성
- 테이블에 foreign key를 생성
  - 테이블 생성
  - 테이블 생성
  - 테이블 생성
  - 테이블 생성
  - 테이블 생성

```
-- 테이블 생성
-- ALTER TABLE 테이블 ADD CONSTRAINT 테이블 FOREIGN KEY (테이블) REFERENCES 테이블(테이블);

-- constraint 테이블 foreign key (테이블) references 테이블(테이블)
alter table user add constraint tmp_dept_id foreign key (dept_id) references
department(id);

-- 테이블 생성
-- 테이블 생성
alter table user drop foreign key tmp_dept_id;
```

테이블

- 테이블 생성
- 테이블에 unique 제약 조건

```
-- 테이블 생성

-- 테이블 card
create table card(
    id int primary key,
    card varchar(18)
```




```
);

-- 用户表
create table user(
    id int primary key,
    -- 姓名
    -- 身份证号
    card_id int not null unique,
    constraint user_card_id foreign key (card_id) references card(id)
);
```

用户表

- 用户表
- 用户表与部门表的关系
  - 用户表
- 用户表

 image-20250825145449821


用户表

- 用户表与部门表的关系

```
-- 用户表
select * from user,department;
```

- 用户表与部门表的关系

用户表

 image-20250825135942870

- 用户表
  - 用户表

```
select * from user as a inner join department as b on a.dept_id =
b.id;
```

- inner join 用户表 on 部门表
- 用户表 where 部门表
- 用户 as 部门

- 用户表

```
select * from user as a,department as b where a.dept_id = b.id;
```



- 左外连接A到B

- 左外连接A到B

```
select * from user a left outer join department b on a.dept_id = b.id;
```

- 右外连接B到A

```
select * from user a right outer join department b on a.dept_id = b.id;
```

- on 连接条件

- where 过滤条件

SQL

SQL select 语句

- 基本语法

```
-- 查询
-- 查询 salary
select salary from user where name = "张三";

-- 查询
-- 查询 salary
select * from user where salary = ?;

-- 查询
select * from user where salary = (select salary from user where name = "张三");
```

- 连接查询

```
-- 查询
-- 查询 dept
-- 查询 dept_id
select id from dept where name = "研发" or name = "市场";

-- 查询
-- 查询 dept_id
select * from user where dept_id = ?;

-- 查询
select * from user where dept_id in (select id from dept where name = "研发" or name = "市场");
```



- 查詢指定員工

```
-- 查詢
-- 查詢指定員工
select salary,job from user where name = "SCOTT";

-- 查詢
-- 查詢指定員工的薪水
select * from user where salary = ? and job = ?;

-- 查詢
select * from user where salary = (select salary from user where name = "SCOTT")
and job = (select job from user where name = "SCOTT");

select * from user where (salary,job) = (select salary,job from user where
name = "SCOTT");
```

- 查詢指定部門

- 查詢指定部門的薪水

```
-- 查詢
-- 查詢指定部門
select dept_id ,max(salary) from user group by dept_id;

-- 查詢
-- 查詢指定部門的薪水
-- 查詢
select * from user a, (select dept_id ,max(salary) max_salary from user group
by dept_id) b where a.dept_id = b.dept_id and a.salary = b.salary;

-- 查詢
select * from user a inner join (select dept_id ,max(salary) max_salary from
user group by dept_id) b on a.dept_id = b.dept_id where a.salary = b.salary;
```

SQL

查詢指定部門的薪水

- 開始事務 start transaction 或 begin
- 提交事務 commit
- 回滾事務 rollback

```
begin
-- 開始SQL
```



```
-- sql语句

-- 提交
commit;

-- 回滚
rollback;
```

## Spring事务管理

- 事务管理相关注解 `@Transactional`
  - `rollbackFor` 指定在哪些异常情况下回滚
    - 默认情况下 `RuntimeException` 和 `Error` 类型

```
public class Test{
    @Transactional(rollbackFor = {Exception.class})
    public void save(){

    }
}
```

- `propagation` 传播模式
  - 事务的传播模式决定了当前方法是否需要开启事务
  - `REQUIRED` 如果当前没有事务，就新建一个，如果有事务，就用该事务
  - `REQUIRES_NEW` 如果当前没有事务，就新建一个，如果有事务，就新建一个

```
@Service
public class WorkParent{

    @Autowired
    private WorkChild workchild;

    @Transactional
    public void save(){
        try{
            /**
             *
             */
        }
        finally{
            // 提交
            workchild.save();
        }
    }
}

@Service
// WorkChild
public class WorkChild{
```



```

// 传播
@Transactional(propagation = Propagation.REQUIRES_NEW)
public void save(){
    /**
     *
     */
}
}

```

- 数据库事务管理

```

logging:
  level:
    org.springframework.jdbc.support.JdbcTransactionManager: debug

```

数据库

- **Spring** 数据库事务管理

数据库

PageHelper 数据库

数据库

```

<dependency>
  <groupId>com.github.pagehelper</groupId>
  <artifactId>pagehelper-spring-boot-starter</artifactId>
  <version>2.1.0</version>
</dependency>

```

- Mapper 数据库

```

@Select("select e.*,d.name deptName from emp e left join dept d on d.id = e.dept_id
order by e.update_time desc ")
List<Emp> list();

```

- Service 数据库 PageHelper 数据库
  - 数据库
  - .getTotal() 数据库
  - .getResult() 数据库
  - 数据库 Select 数据库

```

@Override
public PageResult<Emp> page(Integer page, Integer pageSize){
    // 1. 数据库
    PageHelper.startPage(page, pageSize);

    // 2. 数据库

```



```

        List<Emp> rows = empMapper.list();

        // 3.返回结果
        Page<Emp> p = (Page<Emp>) rows;
        return new PageResult<Emp>(p.getTotal(),p.getResult());
    }

```

数据库

Mybatis数据库操作框架

Mapper.xml配置文件

- CRUD操作 resultMap 数据库表结构
- resultMap 数据库表结构
  - id 数据库主键 type 数据库 pojo 数据库表名
    - id 数据库主键
    - result 数据库表名
      - column 数据库列名
      - property 数据库属性名

SQL

```

<resultMap id="empResultMap" type="com.zxb.pojo.Emp">
    <id column="id" property="id"/>
    <result column="username" property="username"/>
    <result column="password" property="password"/>
    <result column="name" property="name"/>
    <result column="gender" property="gender"/>
    <result column="image" property="image"/>
    <result column="entry_date" property="entryDate"/>
    <result column="dept_id" property="deptId"/>
    <result column="create_time" property="createTime"/>
    <result column="update_time" property="updateTime"/>

<!-- 数据库表结构 -->
    <collection property="exprList" ofType="com.zxb.pojo.EmpExpr">
        <id column="ee_id" property="id"/>
        <result column="ee_company" property="company"/>
        <result column="ee_job" property="job"/>
        <result column="ee_begin" property="begin"/>
        <result column="ee_end" property="end"/>
        <result column="ee_empid" property="empId"/>
    </collection>
</resultMap>

<select id="findById" resultMap="empResultMap">
    select
    e.*,

```



```

        ee.id ee_id,
        ee.company ee_company,
        ee.job ee_job,
        ee.begin ee_begin,
        ee.end ee_end,
        ee.emp_id ee_empid
    from emp e left join emp_expr ee on e.id = ee.emp_id
    where e.id = #{id}
</select>

```

案例

案例

案例 case 语句

- when 语句
- then 语句

```

select
    (case when job=1 then '男'
        when job=2 then '男'
        when job=3 then '男'
        when job=4 then '男'
        when job=5 then '男'
        else '男' end) pos,
    count(*) num
from emp group by job order by num

```

案例

案例 if 语句

```

select
    if(gender=1,'男','女') name,
    count(*) value
from emp group by gender

```

## SpringBoot

案例 properties , yml , yaml

- properties key value 格式 . 文件
- yml 格式
  - 格式
  - 格式
  - 格式 0 格式 ' ' 格式 0 格式 yml 格式
  - yml 格式 @Value 格式



```
@Value("${user.name}")
private String name;
```

- 通过@ConfigurationProperties 标注 prefix 的 yml 配置项  
注入 Bean 中

```
@Data
@Component
@ConfigurationProperties(prefix = "aliyun.oss")
public class AliyunOSSProperties {
    private String endpoint;
    private String bucketName;
    private String region;
}
```

```
## list/set
hobby:
- java
- game
- sport

## map
user:
  name: 小明
  age: 18
  password: 123456
```

## SpringBoot 配置

SpringBoot 默认使用 properties 格式配置，yml 格式配置需要配置

配置

配置 SpringBoot 使用 yml 格式配置

- 在 application-{profile}.yml 或 application-{profile}.properties 中
- 在 application.yml 中配置

```
spring:
  profiles:
    active: dev # 指定 dev 环境
```

配置

配置

- 配置 Nginx
- 配置 Tomcat



- 数据库驱动

## Restful

REST 数据库驱动

数据库驱动

Rest Url	方法	说明
http://localhost:8080/users/1	GET	获取id为1的用户
http://localhost:8080/users/1	DELETE	删除id为1的用户
http://localhost:8080/users	POST	新增用户
http://localhost:8080/users	PUT	更新用户

## Apifox/Postman

数据库驱动 Mock 数据库 API 接口

数据库

数据库 LogBack

- 数据库驱动 logback.xml
- 数据库驱动
- 数据库驱动
  - 数据库 @Slf4j 或 Lombok 注解

```
import lombok.extern.slf4j.Slf4j;

@Slf4j
public class UserService {
    public void processUser(int age) {
        log.info("数据库{}", age);
    }
}
```

- 数据库 final 数据库
  - getLogger 数据库驱动

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
public class UserService{
    private static final Logger logger =
    LoggerFactory.getLogger(UserService.class);
    public void processUser(int age){
        logger.info("数据库{}", age);
    }
}
```



```
}
}
```

- `Logger` 的 `add` 方法

```
import lombok.extern.slf4j.Slf4j;

@Slf4j
public class UserService {
    public void processUser(int age) {
        log.info("UserService: processUser({}), age: {}", age);
    }
}
```

配置

- `logger.level` 配置日志级别
  - `off` 关闭日志
  - `all` 记录所有日志
  - 其他 `trace` `debug` `info` `warn` `error`
- 输出到 `STDOUT` 或 `FILE`
- 配置 `logger` 的 `appender` 属性
- 示例

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <!-- 控制台 -->
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} -
            %msg%n</pattern>
        </encoder>
    </appender>

    <!-- 文件 -->
    <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <file>logs/application.log</file>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>logs/application.%d{yyyy-MM-dd}.log</fileNamePattern>
            <maxHistory>30</maxHistory>
        </rollingPolicy>
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} -
            %msg%n</pattern>
        </encoder>
    </appender>

    <!-- 根日志 -->
    <root level="INFO">
        <appender-ref ref="STDOUT" />
    </root>
</configuration>
```



```

        <appender-ref ref="FILE" />
    </root>

    <!-- 配置日志输出 -->
    <logger name="com.example.service" level="DEBUG" />
    <logger name="org.springframework" level="WARN" />
</configuration>


```

配置

配置web容器

配置

## cookie

 image-20250825143543584

- 配置
- 配置cookie Set-Cookie:cookie 配置cookie
  - 配置 cookie 配置
- cookie配置

配置URL

```

@RestController
public class AuthController {

    @GetMapping("/login")
    public ResponseEntity<String> login(HttpServletResponse response) {
        // 配置Cookie
        Cookie cookie = new Cookie("auth_token", "user123");
        cookie.setMaxAge(3600); // 1小时
        cookie.setHttpOnly(true); // 防止XSS
        cookie.setPath("/"); // 路径

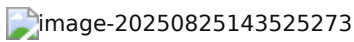
        response.addCookie(cookie);
        return ResponseEntity.ok("登录成功");
    }

    @GetMapping("/profile")
    public String profile(@CookieValue("auth_token") String token) {
        return "Token: " + token;
    }
}

```

## session





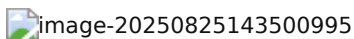
- 浏览器cookie
- 浏览器session Set-Cookie 浏览器session——  
sessionId
- 浏览器sessionId
- 浏览器Nginx
- 浏览器 HttpSession 的 setAttribute 和 getAttribute

```
@RestController
@SessionAttributes("user")
public class SessionController {

    @GetMapping("/setSession")
    public String setSession(HttpSession session) {
        session.setAttribute("username", "用户");
        session.setAttribute("loginTime", new Date());
        return "Session成功";
    }

    @GetMapping("/getSession")
    public String getSession(HttpSession session) {
        String username = (String) session.getAttribute("username");
        Date loginTime = (Date) session.getAttribute("loginTime");
        return "用户名: " + username + ", 登录时间: " + loginTime;
    }
}
```

JWT



JWT

JWT

- JWT
- JWT json

JWT

- JWT Header
- JWT Payload
- JWT Signature token

JWT

- Header Payload Base64URL



- Signature는 Header와 Payload를 합친 것을 암호화한 것

예제

- Header와 Payload를 생성
- Header와 Payload를 합친 것을 암호화

예제

- jwt 라이브러리
- JwtUtils 클래스
  - JwtBuilder를 사용하여 JWT를 생성
  - signWith 메서드로 암호화
  - setClaims 메서드로 클레임 설정
  - setExpiration 메서드로 만료일 설정 (Date 타입)
  - compact 메서드로 compact 형식 생성
- JwtParser 클래스
  - parser 메서드로 JwtParser 생성
  - setSigningKey 메서드로 암호화 키 설정
  - parseClaimsJws 메서드로 JWT를 파싱
    - Claims를 반환
  - getBody() 메서드로 payload를 반환

```
public class JwtUtils{
    // 암호화 키
    private static final SecretKey SECRET_KEY =
Keys.secretKeyFor(SignatureAlgorithm.HS256);
    // 만료 시간
    private static final long EXPIRATION_TIME = 3600 * 1000;

    public static String generateToken(Map<String, Object> claims){
        return Jwts.builder()
            .signWith(SignatureAlgorithm.HS256, SECRET_KEY)
            .setClaims(claims)
            .setExpiration(new Date(System.currentTimeMillis() +
EXPIRATION_TIME))
            .compact();
    }

    public static Claims parseToken(String token){
        return Jwts.parser()
            .setSigningKey(SECRET_KEY)
            .parseClaimsJws(token)
            .getBody();
    }
}
```

**Filter**

**JavaWeb**

예제



- 实现 Filter 接口实现 init 和 doFilter 和 destroy
  - 实现 doFilter 方法接收 HttpServletRequest 和 HttpServletResponse
  - 调用 filterChain.doFilter() 继续下一个 Filter
    - filterChain 是 FilterChain 对象
- 实现 @WebFilter 注解
  - 指定 urlPatterns 匹配规则
    - 匹配 /login
    - 匹配 /\*
    - 匹配 exp/\*
- 使用 @ServletComponentScan 扫描 @WebFilter 注解
- 使用 @WebFilter 注解实现 Filter

```
import javax.servlet.*;

@Slf4j // 日志
@WebFilter("/*") // 匹配规则
public class LoginFilter implements Filter {
    @Override
    public void init(FilterConfig filterConfig) {
        log.info("init.....");
    }

    @Override
    public void doFilter(ServletRequest httprequest, ServletResponse httpresponse,
        FilterChain filterChain) {
        log.info("doFilter");
        HttpServletRequest request = (HttpServletRequest) httprequest;
        HttpServletResponse response = (HttpServletResponse) httpresponse;

        // 获取请求URI
        String requestURI = request.getRequestURI();

        // 获取token
        String token = request.getHeader("token");

        // 判断token是否存在
        if(token == null) {
            log.info("token不存在");
            // 返回401
            response.setStatus(401);
            return;
        }

        // 解析token
        try {
            JwtUtils.parseToken(token); // 解析token
        } catch (Exception e) {
            log.info("token解析失败");
        }
    }
}
```



```

        // 返回
        response.setStatus(401);
        return;
    }

    // 放行
    filterChain.doFilter(request, response);
}

@Override
public void destroy(){
    log.info("销毁....");
}
}

```

总结

- 过滤器 init 方法
- 过滤器 doFilter 方法
- 过滤器 destroy 方法

## Interceptor 拦截器

Spring 拦截器 Spring 拦截器

拦截器拦截器拦截器拦截器拦截器

 image-20250825140013326

总结

- 拦截器 @Component 拦截器 Bean
- 拦截器 HandlerInterceptor 拦截器
  - preHandle 拦截器返回 True 放行
  - postHandle 拦截器
  - afterCompletion 拦截器
- 拦截器配置
  - 拦截器
  - 拦截器 @Configuration 拦截器
  - 拦截器 WebMvcConfigurer 拦截器 addInterceptors 拦截器 registry.addInterceptor 拦截器 addPathPatterns 拦截器 excludePathPatterns 拦截器
    - /\*\* 拦截器
    - /\*\* 拦截器
    - 拦截器 .order() 拦截器

总结

```

@Slf4j
@Component
public class LoginInterceptor implements HandlerInterceptor{
    @Override

```



```

    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler){
        /*
        拦截器
        */
        return true; // 通过
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response,
Object handler, ModelAndView modelAndView){
        log.info("Controller拦截");
    }

    /**
     * 拦截器
     */
    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) {
        log.info("拦截器");
    }
}

```

拦截器

```

@Configuration
public class Webconfig implements WebMvcConfigurer{
    @Autowired
    private LoginInterceptor interceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry){
        // 拦截器/login拦截

        registry.addInterceptor(interceptor).addPathPatterns("/**").excludePathPatterns("/login")
    }
}

```

## SpringAOP

AOP拦截器拦截器拦截器

拦截器拦截器AOP拦截器

- 拦截器
- 拦截器
- 拦截器
- 拦截器

SpringAOP



- 使用SpringAOP
- AOP的基本概念
- 使用SpringAOP进行切面编程

## 什么是AOP

- 使用 @Component 和 @Aspect 进行切面编程
- 使用 ProceedingJoinPoint 进行切面编程
- 使用 @Around 进行切面编程

```
// 使用Around
@Aspect
@Component
public class TestAop {
    // 使用com.zxb.project进行切面编程
    @Around("execution(* com.zxb.project.*(..))")
    public Object test(ProceedingJoinPoint pj) throws Throwable{
        System.out.println("测试");
        // 使用ProceedingJoinPoint
        Object result = pj.proceed();
        return result;
    }
}
```

## 为什么

- 使用AOP进行切面编程
- 使用AOP进行切面编程
- 使用AOP进行切面编程
- 使用AOP进行切面编程
- 使用AOP进行切面编程

## 使用SpringAOP进行切面编程

## 使用SpringAOP

切面	切点
@Around	使用AOP进行切面编程
@After	使用AOP进行切面编程
@Before	使用AOP进行切面编程
@AfterReturning	使用AOP进行切面编程
@AfterThrowing	使用AOP进行切面编程

- 使用 @Around 进行切面编程

```
@Component
@Aspect
public class TestAop{
    // 使用Around
```



```

    @Around("execution(* com.zxb.project0.*(..))")
    public Object testAround(ProceedingJoinPoint pj) throws Throwable{
        System.out.println("[]");
        Object result = pj.proceed();
        return result;
    }
    // []Before[]
    // [][][][][][][][][]
    @Before("execution(* com.zxb.project1.*(..))")
    public void testBefore(){
        System.out.println("[]");
    }

    // []After[]
    // [][][][][][][][][]
    @After("execution(* com.zxb.project2.*(..))")
    public void testAfter(){
        System.out.println("[]");
    }
}

```

[][]

[][][][][][][][][][][][][][][][][][][][]

- [][][][][][][][][]
  - [][][][][][][][][]
  - [][][][][][][][][]
- [][][][]
  - [][][] @Order [][][]
  - @Before [][][]
  - @After [][][]

```

@Component
@Aspect
@Order(3)
public class AopOrder0{
    @Before("execution(* com.zxb.project.*(..))")
    public void testBefore() {
        System.out.println("AopOrder0-[]");
    }
    @After("execution(* com.zxb.project.*(..))")
    public void testAfter() {
        System.out.println("AopOrder0-[]");
    }
}

```

```

@Component
@Aspect
@Order(5)
public class AopOrder1{

```



--	--	--	--	--	--	--	--

- 

- `execution`
  - 
  - `execution(??? ? ??.?.??.(???) throws ??)`
  - `?? ?`
    1. `???`
    2. `?.?.`
    3. `throws ??`
  - 
  - `*`
  - `..`

- @annotation []
  - [] @annotation []



```
// 测试类

// 测试
@Target(ElementType.METHOD)
public @interface LogOperation{
}

// 测试类
public class Test{
    @LogOperation
    public void test(){
        System.out.println("hello");
    }
}

// 测试
@Aspect
@Component
public class TestAop {
    // 测试LogOperation测试类test
    @Before("@annotation(com.zxb.project.LogOperation)")
    public void test(){
        System.out.println("测试");
    }
}
```

111

- `com.zxb.project` `com.zxb.project.*(..)`
- `com.zxb.project` `com.zxb.project..*(..)`
  - `..*` `com.zxb.project`

□□□□□□□□

```
00 @Pointcut 0000000000
```

- `@Pointcut` 注解
- `private` 注解
- `" "` 注解

```
@Component
@Aspect
public class AopOrder{

    // 切面
    @Pointcut("execution(* com.zxb.project.*(..))")
    private void all(){}
```



```

    @Before("all()")
    public void testBefore() {
        System.out.println("AopOrder-前置");
    }

    @After("all()")
    public void testAfter() {
        System.out.println("AopOrder-后置");
    }
}

```

接口

Spring 的 `JoinPoint` 接口

- 通过 `@Around` 注解使用 `ProceedingJoinPoint`
- 通过 `JoinPoint` 接口

接口方法

- `getTarget()` 返回目标对象
- `getTarget().getClass().getName()` 返回目标对象的类名
- `getSignature()` 返回目标方法签名
- `getArgs()` 返回目标方法的参数

```

@Component
@Aspect
public class AopTest{
    @Before("execution(* com.zxb.project..*.*(..))")
    public void test(JoinPoint jp){
        // 打印目标对象
        Object target = jp.getTarget();

        // 打印类名
        String name = jp.getTarget().getClass().getName();

        // 打印方法名
        String methodName = jp.getSignature().getName();

        // 打印参数
        Object[] args = jp.getArgs();
    }
}

```

## Bean

Spring IOC 容器中的 Bean

Spring IOC 容器中的 **Bean** 是指

Spring IOC

- 通过 `ApplicationContext` 接口，Spring Framework 实现了 **Springboot** 接口



- 测试

- `getBean()` 返回Bean对象

```
@Component
public class SimpleService {
    public void sayHello() {
        System.out.println("Hello from SimpleService!");
    }
}

@Component
public class Test{
    // ApplicationContext
    @Autowired
    private ApplicationContext applicationContext;
    public void test(){
        System.out.println(applicationContext.getBean("simpleService"));
    }
}
```

- 测试

- 使用IOC容器 @Lazy 测试
- 使用Bean容器测试IOC

## Bean容器

- `singleton` 单例bean
- `prototype` 多例bean

使用 @Scope 测试IOC

```
@Scope("prototype")
@Component
public class TestScope{
    public void test(){
        System.out.println("测试");
    }
}

// 测试
public class Test{
    // 测试
    @Autowired
    private TestScope testScope;
}
```

测试bean

- 测试bean singleton



- 配置bean的scope
- 配置bean的scope prototype
  - 配置bean的scope

配置 @Bean 配置

- 配置 @Configuration 类
- 配置 @Bean 类
  - 配置 @Bean
  - 配置 @Bean
  - 配置 new 配置
  - 配置 @Bean 类

```
public class ThirdWay{
    /*
    配置
    */
}

@Configuration
public class Config{
    @Bean
    public ThirdWay thirdWay(){
        return new ThirdWay();
    }
}

// 配置
@Component
public class MainProcess{
    @Autowired
    private ThirdWay thirdWay;
}
```

## SpringBoot

配置

SpringBoot 配置 maven 配置

配置

配置 spring 配置 bean 配置 IOC 配置

- 配置 ApplicationContext 配置
- SpringBoot 配置

配置 Springboot 配置 @ComponentScan 配置 basePackages 配置

- 配置 Springboot 配置 IOC 配置



```

@ComponentScan(basePackages={"com.example"})
@SpringBootApplication
public class Begin{

}

```

ImportSelector @Import

ImportSelector

ImportSelector @Import

```

public class ImportConfig implements ImportSelector{
    public String[] selectImports(AnnotationMetadata importClassMetadata){
        return new String[]{"com.example.Hello","com.example.Goodbye"};
    }
}

@Import(ImportConfig.class)
@SpringBootApplication
public class Begin{
    @Autowired
    private Hello hello; //
}

```

@EnableXXX

Springboot

Springboot @SpringBootApplication

- @SpringBootConfiguration @Configuration
- @ComponentScan
- @EnableAutoConfiguration
  - @Import ImportSelector SelectImports
  - 
  - 
  - Springboot @Configuration Bean IOC
  - @Bean IOC

```

@Configuration
public class Config{
    @Bean
    public ThirdWay thirdWay(){
        return new ThirdWay();
    }
}

```



```

}

// 测试
@Component
public class MainProcess{
    @Autowired
    private ThirdWay thirdWay;
}

```

看看 @Conditional 注解

- 注解在类上，表示该类在什么条件下被 Spring IOC 容器管理
- 注解在方法上
  - @ConditionalOnClass 表示在什么条件下被 Spring IOC 容器管理
    - 参数 name 表示要检查的类名
  - @ConditionalOnMissingBean 表示在什么条件下被 Spring IOC 容器管理
    - 参数 name 表示要检查的 bean 名称
  - ConditionalOnProperty 表示在什么条件下被 Spring IOC 容器管理
    - 参数 name 表示要检查的属性名，havingValue 表示要检查的值

看看 starter

- starter 注解
- 在 AutoConfiguration 类上使用 XXXAutoConfiguration 注解
- 在 META-INF/spring/... 目录下创建文件
- starter 注解

看看

- POJO 类
  - Entity 类
  - DTO 类
  - VO 类

看看 BeanUtils.copyProperties(DTO, Entity) 方法

看看 @Builder 注解

```

Employee employee = Employee.builder()
    .id(id)
    .status(status)
    .build();

```

## Swagger

看看 Knife4j

1. 添加 knife4j 依赖
2. 配置 knife4j 注解



### 3. Spring Boot API 문서 생성

#### Swagger Bean

- Docket 생성
- Swagger UI 생성
- Swagger API 생성

```
@Bean
public Docket docket() {
    ApiInfo apiInfo = new ApiInfoBuilder()
        .title("Swagger API")
        .version("2.0")
        .description("API 문서")
        .build();
    Docket docket = new Docket(DocumentationType.SWAGGER_2)
        .apiInfo(apiInfo)
        .select()
        // Swagger UI
        .apis(RequestHandlerSelectors.basePackage("com.sky.controller"))
        .paths(PathSelectors.any())
        .build();
    return docket;
}
```

#### Swagger UI

```
protected void addResourceHandlers(ResourceHandlerRegistry registry) {

    registry.addResourceHandler("/doc.html").addResourceLocations("classpath:/META-INF/resources/");

    registry.addResourceHandler("/webjars/**").addResourceLocations("classpath:/META-INF/resources/webjars/");
}
```

#### Swagger API

- @Api Controller 생성
  - tag 생성
- @ApiModel entity DTO VO
  - description 생성
- @ApiModelProperty 생성
- @ApiOperation Controller 생성
  - value 생성

Swagger API 생성

## ThreadLocal

ThreadLocal Thread Thread



- 可透過 Setter 方法來存取物件
- 可透過 Getter 方法來存取物件

範例

- `public void set(T value)` 可透過 Setter 方法來存取物件
- `public T get()` 可透過 Getter 方法來存取物件
- `public void remove()` 可透過 Setter 方法來存取物件

## SpringCache

SpringCache 是一個基於 Spring 的缓存框架

SpringCache 支援以下缓存實現

- EHCache
- Caffeine
- Redis

SpringCache 的用法

- `@EnableCaching` 啟用 SpringCache
- `@Cacheable` 方法執行結果會存入缓存
  - `cacheNames` 指定缓存名稱
  - `key` 指定缓存 key
    - 指定缓存 key 的表达式 #
    - 指定 #result 返回缓存 key
  - 指定缓存名稱 `cacheNames::key`
- `@CachePut` 方法執行結果會存入缓存
  - `cacheNames` 指定缓存名稱
  - `key` 指定缓存 key
    - 指定缓存 key 的表达式 #
    - 指定 #result 返回缓存 key
  - 指定缓存名稱 `cacheNames::key`
- `@CacheEvict` 方法執行結果會清除缓存
  - `cacheNames` 指定缓存名稱
  - `key` 指定缓存 key
    - 指定缓存 key 的表达式 #
    - 指定 #result 返回缓存 key
  - 指定缓存名稱 `cacheNames::key`
  - 指定缓存 key 的表达式 `key` 指定 `allEntries=true`

## SpringTask

SpringTask 是一個基於 Spring 的任務框架

SpringTask 的用法

1. 添加 maven 依賴
2. 添加 `@EnableScheduling` 注解
3. 添加 `@Component` 注解
  - 指定 `@Component` 的類名



- `@Scheduled` `@Scheduled` `@Scheduled`

## cron

`cron` `cron`

`6 7`

- `6 7`
- `6 7`


`6 7`

- `*`
- `?`
- `-`
- `,`
- `/`
- `L`
- `W`
- `#`

## WebSocket

`TCP`

- `TCP`
- `TCP`
- `TCP`

 image-20250913103324320

`WebSocket`

1. `WebSocketServer`
2. `WebSocketConfiguration` `WebSocketServer`
  - `@Configuration`
  - `Bean` `ServerEndpointExporter`
    - `ServerEndpointExporter` `@ServerEndpoint` `Websocket`

`WebSocket`

- `@ServerEndpoint` `WebSocket` `URL`
- `@OnOpen`
- `@OnMessage`



- `@OnClose` 關閉時執行
- `@PathParam` 指定WebSocket路徑

## Apache POI

POI是Microsoft Office檔案操作工具包

POI是Java操作Microsoft Office檔案工具包

POI是

- 使用maven

```
<dependency>
  <groupId>org.apache.poi</groupId>
  <artifactId>poi</artifactId>
  <version>3.16</version>
</dependency>
<dependency>
  <groupId>org.apache.poi</groupId>
  <artifactId>poi-ooxml</artifactId>
  <version>3.16</version>
</dependency>
```

- POI操作excel檔案IO操作
- POI操作excel
- POI excel檔案操作 XSSFWorkbook 操作excel檔案

POI

- XSSFWorkbook 操作excel檔案
- XSSFSheet 操作sheet
- `.createSheet(sheetname)` 創建Excel檔案sheet
- `.createRow(rownum)` 在sheet中創建 rownum 為第rownum sheet
- `.createCell(index)` 在sheet中創建 index 為第rownum
- `.setCellValue()` 設置值
- `.write(IO)` 寫入檔案
- `.close()` 關閉
- `.getSheetAt(index)` 獲取 index + 1 sheet
- `.getLastRowNum()` 獲取sheet中最後一行
- `getCell()` 獲取單元格
- `getStringCellValue()` 獲取字符串值



cors

cors MVC

cors

- cors addCorsMappings corsRegistry registry
- cors registry cors
  - .addMapping() cors
  - .allowedOrigins() cors
  - .allowedMethods() cors
  - .allowedHeaders() cors

```
@Configuration
public class MvcConfiguration implements WebMvcConfigurer{
    @Override
    public void addCorsMappings(CorsRegistry registry){
        registry.addMapping("/**")
            .allowedOrigins("*")
            .allowedMethods("GET")
            .allowedHeaders("*");
    }
}
```