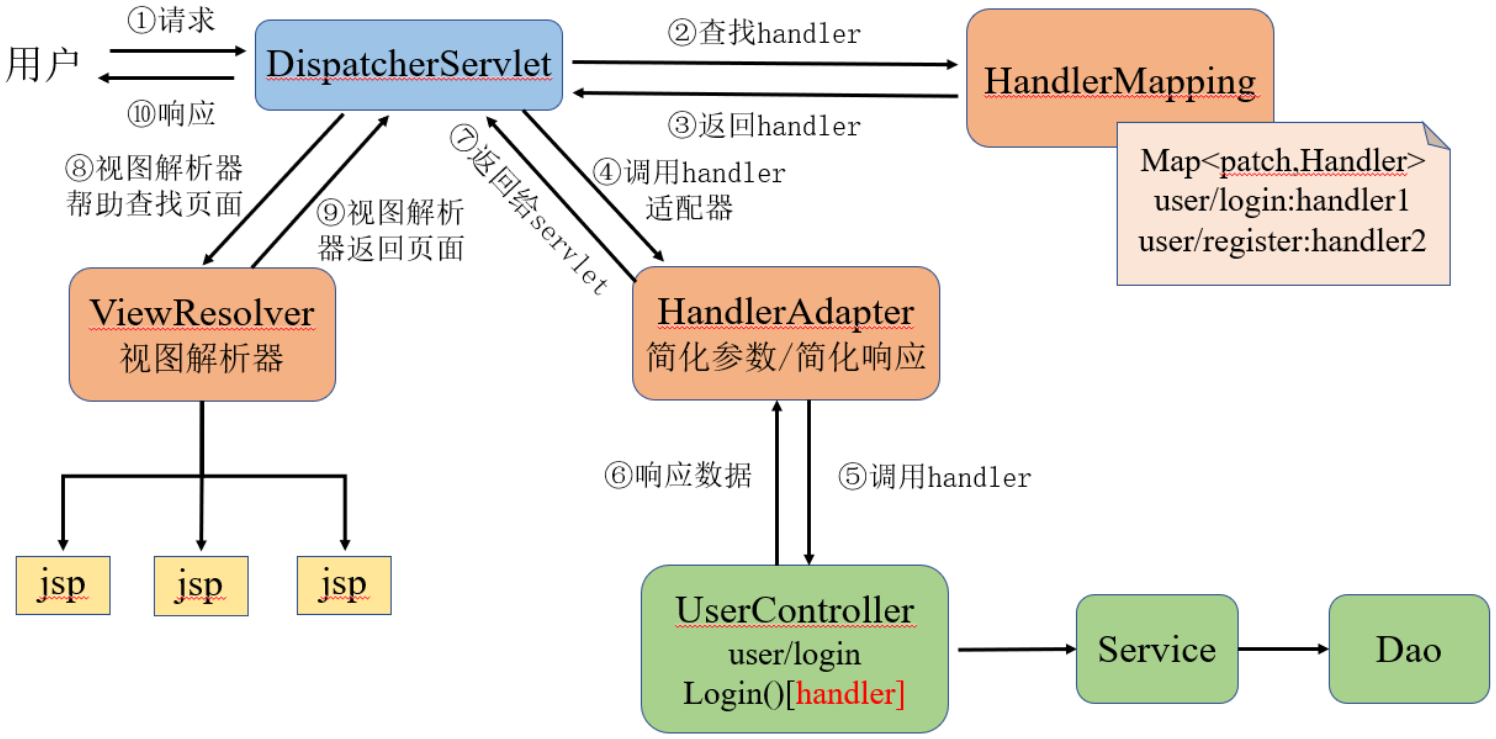


二、SpringMVC 接收数据

1、访问路径设置

@RequestMapping 注解的作用就是将**请求的 URL 地址**和**处理请求的方式**（handler 方法）**关联起来**，建立映射关系。**SpringMVC 接收**到指定的**请求**，就会来**找**到**在映射关系中对应的方法**来**处理这个请求**。

（@RequestMapping 不是必须使用/开头）

1）精准路径匹配（{“/user/login”}）

在@RequestMapping 注解指定 URL 地址时，不使用任何通配符，按照请求地址进行精确匹配。**多个地址时**{“url1”,“url2”}

```
@Controller
public class UserController {
    /**
     * 精准设置访问地址 /user/login
     */
    @RequestMapping(value = {"/user/login"})
    @ResponseBody
    public String login(){
        System.out.println("UserController.login");
        return "login success!!";
    }
}
```

2）模糊路径匹配（单层{“/user/*”}/多层{“/user/**”}）

在@RequestMapping 注解指定 URL 地址时，通过使用通配符，匹配多个类似的地址。

```
@Controller
public class ProductController {

    /**
     *  路径设置为 /product/*
     *    /* 为单层任意字符串  /product/a  /product/aaa 可以访问此handler
     *    /product/a/a 不可以
     *  路径设置为 /product/**
     *   /** 为任意层任意字符串  /product/a  /product/aaa 可以访问此handler
     *   /product/a/a 也可以访问
     */
    @RequestMapping("/product/*")
    @ResponseBody
    public String show(){
        System.out.println("ProductController.show");
        return "product show!";
    }
}
```

【注意】单层匹配和多层匹配：

1> **/*：**只能**匹配** URL 地址中的**一层**，想准确**匹配两层**写“/*/*”。

2> **/**：**可以**匹配** URL 地址中的**多层**。

其中所谓的一层或多层是指一个 URL 地址字符串被“/”划分出来的各个层次。这个知识点虽然对于@RequestMapping 注解来说实用性不大，但是将来配置拦截器的时候也遵循这个规则。

3）类和方法级别区别：

@RequestMapping 注解可以用于类级别和方法级别，它们之间区别如下：

1> 设置到**类级别**：@RequestMapping 注解可以**设置在控制器类上**，用于**映射整个控制器**的**通用请求路径**。这样，如果控制器中的**多个方法**都**需要映射同一请求路径**，就不需要在每个方法上都添加映射路径。

2> 设置到**方法级别**：@RequestMapping 注解也可以单独**设置在控制器方法上**，用于**更细粒度地映射具体的请求路径和处理方法**。当**多个方法处理同一个路径的不同操作**时，可使用方法级别的@RequestMapping 注解进行更精细的映射。

【注意】如果**类上加**@RequestMapping(“/user”)，**方法上必须要加**@RequestMapping；才能访问到/user。

4）附带请求方式限制（method = RequestMethod.XXX）

HTTP 协议定义了八种请求方式，在 SpringMVC 中封装到这个枚举类：

```
public enum RequestMethod {
  GET, HEAD, POST, PUT, PATCH, DELETE, OPTIONS, TRACE
}
```

默认情况下：@RequestMapping(“/logout”) **任何请求方式都可以访问。**

如果需要可以使用 **method** 特定指定，**违背请求方式时**会出现 **405 异常**。

```
@Controller
public class UserController {

    /**
     * 精准设置访问地址 /user/login
     * method = RequestMethod.POST 可以指定单个或者多个请求方式!
     * 注意:违背请求方式会出现405异常!
     */
    @RequestMapping(value = {"/user/login"} , method = RequestMethod.POST)
    @ResponseBody
    public String login(){
        System.out.println("UserController.login");
        return "login success!!";
    }

    /**
     * 精准设置访问地址 /user/register
     */
    @RequestMapping(value={"/user/reg"},method={RequestMethod.POST,RequestMethod.GET})
    @ResponseBody
    public String register(){
        System.out.println("UserController.register");
        return "register success!!";
    }
}
```

5）进阶注解（HTTP 方法特定快捷方式变体，**只能加在方法上**）

@RequestMapping 的 HTTP 方法特定快捷方式变体：@GetMapping、@PostMapping、@PutMapping、@DeleteMapping、@PatchMapping。

【注】@RequestMapping(value=”/login”,method=RequestMethod.GET) = @GetMapping(value=”/login”);

进阶注解**只能添加到 handler 方法上，无法添加到类上。**

6）常见配置问题

```
There is already 'demo03MappingMethodHandler' bean method
com.atguigu.mvc.handler.Demo03MappingMethodHandler#empGet() mapped.
```

出现原因：**多个 handler 方法映射了同一个地址**，导致 SpringMVC 在接收到这个地址的请求时该找哪个 handler 方法处理。

2、接收参数

1）param 和 json 参数比较

1> 参数**编码**：

param 类型的参数会被编码为 **ASCII 码**。

（例：假设 name=john doe，则会被编码为 name=john%20doe。）

JSON 类型参数会被编码为 **UTF-8**。

2> 参数**顺序**：

param 类型的参数**没有顺序限制**。

JSON 类型的参数是**有序的**。JSON 采用**键值对**的形式**进行传递**，其中**键值对**是**有序排列**的。

3> 数据**类型**：

param 类型的参数仅支持**字符串**类型、**数值**类型和**布尔**类型等简单数据类型。**JSON 类型**的参数则支持**更复杂的数据类型**，如**数组**、**对象**等。

4> **嵌套性**：

param 类型的参数**不支持嵌套**。但是，**JSON 类型**的参数**支持嵌套**，可以传递更为复杂的数据结构。

5> **可读性**：

param 类型的参数格式比 JSON 类型的参数更加简单、易读。但是，JSON 格式在传递嵌套数据结构时更加清晰易懂。

总的来说，**param 类型**的参数适用于**单一的数据传递**，而 **JSON 类型**的参数则更适用于**更复杂的数据结构传递**。根据具体的业务需求，需要选择合适的参数类型。在实际开发中，常见的做法是：在 **GET 请求**中采用 **param 类型**的参数，而在 **POST 请求**中采用 **JSON 类型**的参数传递。

2）param 参数接收（四种场景）

1> 直接接值

handler 接收参数，只要**形参名和类型**与**传递参数相同**即可自动接收。

```
@Controller
@RequestMapping("param")
public class ParamController {


    /**
     * 前端请求: http://localhost:8080/param/value?name=xx&age=18
     *
     * 可以利用形参列表,直接接收前端传递的param参数!
     *     要求: 参数名 = 形参名
     *           类型相同
     * 出现乱码正常, json接收具体解决! !
     * @return 返回前端数据
     */
    @GetMapping(value="/value")
    @ResponseBody
    public String setupForm(String name,int age){
        System.out.println("name = " + name + ", age = " + age);
        return name + age;
    }
}
```

2> @RequestParam 注解（value、required、defaultValue）

可以使用**@RequestParam 注释**将 **Servlet 请求参数**（即查询参数或表单数据）**绑定到控制器中的方法参数**。

@RequestParam 属性：**value** 指定绑定的请求参数名、**required** 要求请求参数必须传递、**defaultValue** 为请求参数提供默认值。

```
@GetMapping(value="/data")
@ResponseBody
public Object paramForm(@RequestParam("name") String name,
                        @RequestParam(required = false,defaultValue = "18") int age){
    System.out.println("name = " + name + ", age = " + age);
    return name+age;
}
```

3> 特殊场景接值(一名多值，集合接收)

多选框，提交数据的时候一个 key 对应多个值，使用**集合**进行**接收**。

```
/**
 * 前端请求: http://localhost:8080/param/mul?hbs=吃&hbs=喝
 *
 *  一名多值,可以使用集合接收即可!但是需要使用@RequestParam注解指定
 */
@GetMapping(value="/mul")
@ResponseBody
public Object mulForm(@RequestParam List<String> hbs){
    System.out.println("hbs = " + hbs);
    return hbs;
}
```

4> 实体对象接收（属性名必须等于参数名）

Spring MVC 是 Spring 框架提供的 Web 框架，它允许开发者**使用实体对象来接收 HTTP 请求中的参数**。通过这种方式，可以在**方法内部**直接**使用对象的属性**来**访问请求参数**，而不需要每个参数都写一遍。

【注意】代码中将请求参数 name 和 age 映射到实体类属性上，要求**属性名必须等于参数名**，否则无法映射。

定义一个用于接收参数的实体类：

```
public class User {
  private String name;
  private int age = 18;
  // getter 和 setter 略
}
```

在控制器中，使用实体对象接收：

```
@Controller
@RequestMapping("param")
public class ParamController {

    @RequestMapping(value = "/user", method = RequestMethod.POST)
    @ResponseBody
    public String addUser(User user) {
        // 在这里可以使用 user 对象的属性来接收请求参数
        System.out.println("user = " + user);
        return "success";
    }
}
```

3）路径参数接收（@PathVariable，可令 value=“动态标识”）

路径传递参数是一种在 **URL 路径中传递参数**的方式。在 **RESTful** 的 Web 应用程序中，经常**使用路径传递参数**来**表示资源的唯一标识符**或**更复杂的表示方式**。

@PathVariable 注解允许将 **URL 中的占位符**映射到**控制器方法中的参数**。例如，如果我们想将**/user/{id}路径下**的**{id}映射到控制器方法的一个参数中**，则可以使用@PathVariable 注解来实现。

【注意】形参名=动态标识，则自动赋值；否则令 value=“动态标识”

```
/**
 * 动态路径设计: /user/{动态部分}/{动态部分}   动态部分使用{}包含即可! {}内部动态标识!
 * 形参列表取值: @PathVariable Long id  如果形参名 = {动态标识} 自动赋值!
 *             @PathVariable("动态标识") Long id 如果形参名!= {动态标识} 可以通过指定动态标识赋值!
 *
 * 访问测试:  /param/user/1/root  -> id = 1  uname = root
 */
@GetMapping("/user/{id}/{name}")
@ResponseBody
public String getUser(@PathVariable Long id,
                      @PathVariable("name") String uname) {
    System.out.println("id = " + id + ", uname = " + uname);
    return "user_detail";
}
```

4）json 参数接收

前端传递 JSON 数据时，Spring MVC 框架可以使用**@RequestBody 注解**来将 **JSON 数据转换**为 **Java 对象**。

@RequestBody 注解表示**当前方法参数的值**应该**从请求体中获取**，并且**需要指定 value 属性**来指示**请求体**应该**映射**到**哪个参数上**。

1> 前端发送 JSON 数据的示例：

```
{
  "name": "张三",
  "age": 18,
  "gender": "男"
}
```

2> 定义一个用于接收 JSON 数据的 Java 类：

```
public class Person {
  private String name;
  private int age;
  private String gender;
  // getter 和 setter 略
}
```

3> 在控制器中，使用@RequestBody 注解来接收 JSON 数据，并将其转换为 Java 对象：

```
@PostMapping("/person")
@ResponseBody
public String addPerson(@RequestBody Person person) {

  // 在这里可以使用 person 对象来操作 JSON 数据中包含的属性
  return "success";
}
```

在上述代码中，@RequestBody 注解将请求体中 JSON 数据映射到 Person 类型的 person 参数上，并将其作为一个对象来传递给 addPerson()方法进行处理。

【注意】

springmvc handlerAdpater 配置 json 转化器,配置类需要明确：

```
//TODO: SpringMVC对应组件的配置类 [声明SpringMVC需要的组件信息]

//TODO: 导入handlerMapping和handlerAdapter的三种方式
 //1.自动导入handlerMapping和handlerAdapter [推荐]
 //2.可以不添加,springmvc会检查是否配置handlerMapping和handlerAdapter,没有配置默认加载
 //3.使用@Bean方式配置handlerMapper和handlerAdapter
@EnableWebMvc  //json数据处理,必须使用此注解,因为他会加入json处理器
@Configuration
@ComponentScan(basePackages = "com.atguigu.controller") //TODO: 进行controller扫描

//WebMvcConfigurer springMvc进行组件配置的规范,配置组件,提供各种方法! 前期可以实现
public class SpringMvcConfig implements WebMvcConfigurer {
```

pom.xml 加入jackson依赖

```xml
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.15.0</version>
</dependency>
```

4）接收 Cookie 数据（@CookieValue（“key”））

可以使用**@CookieValue 注解**将 HTTP Cookie 的值绑定到控制器中的方法参数。

考虑使用以下 cookie 的请求：

```
JSESSIONID=415A4AC178C59DACE0B2C9CA727CDD84
```

下面的示例演示如何获取 cookie 值：

```
@GetMapping("/demo")
public void handle(@CookieValue("JSESSIONID") String cookie) {
  //...
}
```

5）接收请求头数据（@RequestHeader（“key”））

可以使用**@RequestHeader** 注解将请求标头绑定到控制器中的方法参数。

请考虑以下带有标头的请求：

```
Host                    localhost:8080
Accept                  text/html,application/xhtml+xml,application/xml;q=0.9
Accept-Language         fr,en-gb;q=0.7,en;q=0.3
Accept-Encoding         gzip,deflate
Accept-Charset          ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive              300
```

下面的示例获取 `Accept-Encoding` 和 `Keep-Alive` 标头的值：

```
@GetMapping("/demo")
public void handle(
    @RequestHeader("Accept-Encoding") String encoding,
    @RequestHeader("Keep-Alive") long keepAlive) {
  //...
}
```

Controller method argument 控制器方法参数	Description
`jakarta.servlet.ServletRequest`, `jakarta.servlet.ServletResponse`	请求/响应对象
`jakarta.servlet.http.HttpSession`	强制存在会话。因此，这样的参数永远不会为 `null` 。
`java.io.InputStream`, `java.io.Reader`	用于访问由 Servlet API 公开的原始请求正文。
`java.io.OutputStream`, `java.io.Writer`	用于访问由 Servlet API 公开的原始响应正文。
`@PathVariable`	接收路径参数注解
`@RequestParam`	用于访问 Servlet 请求参数，包括多部分文件。参数值将转换为声明的方法参数类型。
`@RequestHeader`	用于访问请求标头。标头值将转换为声明的方法参数类型。
`@CookieValue`	用于访问Cookie。Cookie 值将转换为声明的方法参数类型。
`@RequestBody`	用于访问 HTTP 请求正文。正文内容通过使用 `HttpMessageConverter` 实现转换为声明的方法参数类型。
`java.util.Map`, `org.springframework.ui.Model`, `org.springframework.ui.ModelMap`	共享域对象，并在视图呈现过程中向模板公开。
`Errors`, `BindingResult`	验证和数据绑定中的错误信息获取对象！

6）原生 Api 对象操作

如果想要**获取请求或者响应对象**，或者**会话等**，可以直接在形参列表传入，并且**不分先后顺序**。

【注意】接收原生对象，并**不影响参数接收**。

获取原生对象示例：

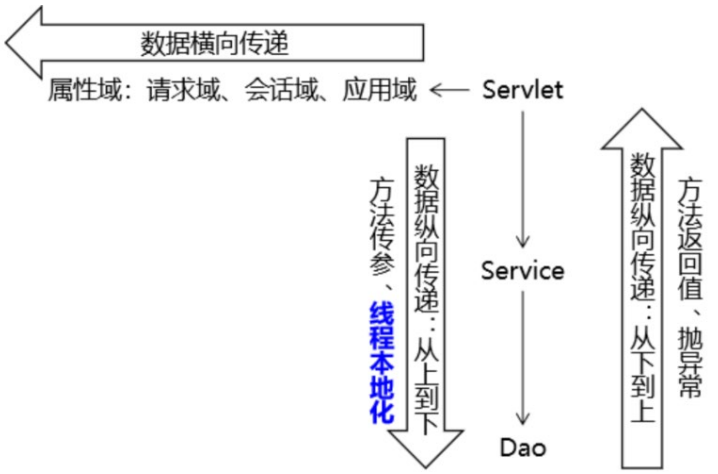
```
/**
 * 如果想要获取请求或者响应对象,或者会话等,可以直接在形参列表传入,并且不分先后顺序!
 * 注意: 接收原生对象,并不影响参数接收!
 */
@GetMapping("api")
@ResponseBody
public String api(HttpSession session , HttpServletRequest request,
                  HttpServletResponse response){
    String method = request.getMethod();
    System.out.println("method = " + method);
    return "api";
}
```

7）共享域对象操作

1）属性（共享）域作用

在 JavaWeb 中，共享域指的是在 Servlet 中存储数据，以便在**同一 Web 应用程序的多个组件之间进行共享和访问**。常见的共享域有四种：ServletContext、HttpSession、HttpServletRequest、PageContext。

- **ServletContext 共享域：**ServletContext 对象可以在整个 Web 应用程序中共享数据，是**最大的共享域**。一般用于保存整个 Web 应用程序的**全局配置信息**，以及**【所有用户】都共享的数据**。在 ServletContext 中保存的数据是**线程安全**的。
- **HttpSession 共享域：**HttpSession 对象可以在**【同一用户】**发出的**多个请求之间共享数据**，但只能**在同一个会话中**使用。比如，可以将用户**登录状态**保存在 HttpSession 中，让用户在多个页面间保持登录状态。
- **HttpServletRequest 共享域：**HttpServletRequest 对象可以在**【同一个请求】**的**多个处理器方法之间**共享数据。比如，可以将**请求的参数和属性**存储在 HttpServletRequest 中，让**处理器方法之间**可以**访问这些数据**。
- **PageContext 共享域：**PageContext 对象是在 **JSP 页面 Servlet 创建时**自动创建的。它可以在 JSP 各个作用域中共享数据，包括 pageScope、requestScope、sessionScope、applicationScope 等作用域。



2）Request 级别属性（共享）域

- 使用 **Model 类型**的形参

在形参位置声明 Model 类型变量，用于存储模型数据到请求域；model.addAttribute()存入模型数据。

```
@RequestMapping("/attr/request/model")
@ResponseBody
public String testAttrRequestModel(Model model) {

    // 在形参位置声明Model类型变量，用于存储模型数据
    // 我们将数据存入模型，SpringMVC 会帮我们把模型数据存入请求域，也被称为暴露到请求域
    model.addAttribute("requestScopeMessageModel","i am very happy[model]");
    return "target";
}
```

- 使用 **ModelMap 类型**的形参

在形参位置声明 ModelMap 类型变量，用于存储模型数据到请求域；modelMap.addAttribute()存入模型数据。

```
@RequestMapping("/attr/request/model/map")
@ResponseBody
public String testAttrRequestModelMap(ModelMap modelMap) {

    // 在形参位置声明ModelMap类型变量，用于存储模型数据
    // 我们将数据存入模型，SpringMVC 会帮我们把模型数据存入请求域,也被称为暴露到请求域
    modelMap.addAttribute("requestScopeMessageModelMap","i am very happy[model map]");
    return "target";
}
```

- 使用 **Map 类型**的形参
在形参位置声明 Map 类型变量，用于存储模型数据到请求域；
map.put()存入模型数据。

```
@RequestMapping("/attr/request/map")
@ResponseBody
public String testAttrRequestMap(Map<String, Object> map) {

    // 在形参位置声明Map类型变量，用于存储模型数据
    // 我们将数据存入模型，SpringMVC 会帮我们把模型数据存入请求域,被称为暴露到请求域
    map.put("requestScopeMessageMap", "i am very happy[map]");
    return "target";
}
```

- 使用**原生 request 对象**
拿到原生对象，就可以调用原生方法执行各种操作；
request.setAttribute()存入模型数据。

```
@RequestMapping("/attr/request/original")
@ResponseBody
public String testAttrOriginalRequest(HttpServletRequest request) {

    // 拿到原生对象，就可以调用原生方法执行各种操作
    request.setAttribute("requestScopeMessageOriginal", "i am very happy[original]");
    return "target";
}
```

- 使用 **ModelAndView 对象**
> 创建 ModelAndView 对象；
> modelAndView.addObject()存入模型数据；
> 设置视图名称。

```
@RequestMapping("/attr/request/mav")
public ModelAndView testAttrByModelAndView() {

    // 1.创建ModelAndView对象
    ModelAndView modelAndView = new ModelAndView();
    // 2.存入模型数据
    modelAndView.addObject("requestScopeMessageMAV", "i am very happy[mav]");
    // 3.设置视图名称
    modelAndView.setViewName("target");

    return modelAndView;
}
```

3> Session 级别属性（共享）域

```
@RequestMapping("/attr/session")
@ResponseBody
public String testAttrSession(HttpSession session) {
    //直接对session对象操作,即对会话范围操作!
    return "target";
}
```

4> Application 级别属性（共享）域
springmvc 会在初始化容器的时候,将 servletContext 对象存储到 ioc 容器中。

```
@Autowired
private ServletContext servletContext;

@RequestMapping("/attr/application")
@ResponseBody
public String attrApplication() {

    servletContext.setAttribute("appScopeMsg", "i am hungry...");
    return "target";
}
```

3、SpringMVC 响应数据
1）理解 handler 方法的作用和组成：
一个 controller 的方法是控制层的一个处理器,我们称为 handler。handler 需要使用@RequestMapping/@GetMapping 系列,声明路径,在 HandlerMapping 中注册,供 DS 查找。
handler 作用总结：
1> **handler 形参列表→**接收请求参数(param,json,pathVariable 等)
2> **调用业务逻辑→**{方法体 可以向后调用业务方法 service.xx()}
3> **return 返回结果→**响应前端数据(页面,json,转发和重定向等)

```
@GetMapping
public Object handler(简化请求参数接收){
    调用业务方法
    返回的结果 (页面跳转，返回数据 (json) )
    return 简化响应前端数据;
}
```

【注意】在 Web 开发中有两种主要开发模式：前后端分离和混合开发：
1> **前后端分离模式：**
指将**前端的界面**和**后端的业务逻辑**通过**接口分离开发**的一种方式。开发人员使用**不同的技术栈和框架**，**前端**开发人员主要负责**页面的呈现**和**用户交互**，**后端**开发人员主要负责**业务逻辑**和**数据存储**。**前后端通信**通过 **API 接口**完成，**数据格式**一般使用 **JSON 或 XML**。前后端分离模式可以提高开发效率，同时也有助于代码重用和维护。
2> **混合开发模式：**
指将**前端和后端的代码集成在同一个项目**中，**共享相同的技术栈和框架**。这种模式在小型项目中比较常见，可以减少学习成本和部署难度。但是，在大型项目中，这种模式会**导致代码耦合性很高**，维护和升级难度较大。对于**混合开发**，我们就需要使用**动态页面技术**，**动态展示 Java 的共享域数据**。
2）转发和重定向（return “关键字: /路径”）
在 Spring MVC 中，Handler 方法返回值来实现快速转发，可以使用 **redirect** 或者 **forward 关键字**来实现重定向。
【注意】将**方法的返回值**设置为 **String 类型**。
【补充】**重定向和转发的区别：**
* **重定向**是指当浏览器请求一个 URL 时，**服务器返回**一个**重定向指令**，告诉浏览器地址已经变了，麻烦使用新的 URL 再**重新发送新请求**。
> **重定向有两种：**一种是 **302 响应**，称为**临时**重定向，一种是 **301 响应**，称为**永久重定向**。**两者的区别**是，如果服务器发送 301 永久重定向响应，浏览器会缓存/hi 到/hello 这个重定向的关联，**下次**请求/hi 的时候，浏览器就**直接发送/hello 请求**了。
> **重定向目的**是当 Web 应用升级后，如果**请求路径发生变化**，可以**将原来的**路径**重定向到新路径**，从而**避免浏览器请求**原路径**找不到资源**。
* **转发**是指**在 Web 服务器内部**转发。当一个 Servlet 处理请求的时候，它可以决定自己不继续处理，而是转发给另一个 Servlet 处理，**只发出了一个 HTTP 请求**。

```
@RequestMapping("/redirect-demo")
public String redirectDemo() {
    // 重定向到 /demo 路径
    return "redirect:/demo";
}

@RequestMapping("/forward-demo")
public String forwardDemo() {
    // 转发到 /demo 路径
    return "forward:/demo";
}
```

3）返回 JSON 数据（重点）
1> 前置准备
导入 jackson 依赖 **jackson-databind**

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.15.0</version>
</dependency>
```

【**注意**】添加 **json 数据转化器--@EnableWebMvc**；springboot 中不用
2> @ResponseBody
在**前后端分离**的项目中,可以**在方法上用@ResponseBody 注解**，用于将**方法返回的对象*序列化***为 **JSON 或 XML 格式的数据**，并**发送给客户端**。具体来说，**@ResponseBody 注解**可以用来**标识方法或者方法返回值**，表示**方法的返回值是要直接返回给客户端的数据**，而**不是由视图解析器**来解析并渲染生成响应体（viewResolver 没用）。

```
@RequestMapping(value = "/user/detail", method = RequestMethod.POST)
@ResponseBody
public User getUser(@RequestBody User userParam) {
    System.out.println("userParam = " + userParam);
    User user = new User();
    user.setAge(18);
    user.setName("John");
    //返回的对象,会使用jackson的序列化工具,转成json返回给前端!
    return user;
}
```

3> @RestController（给类中的每个方法都加了@ResponseBody 注解）
如果**类中每个方法上都标记了**@ResponseBody 注解,那么这些注解就可以**提取到类上**。类上的@ResponseBody 注解可以@Controller 注解合并为@RestController 注解。所以使用了@RestController 注解就相当于给类中的每个方法都加了@ResponseBody 注解。

4> 返回静态资源处理

- 静态资源概念:资源本身已经是**可以直接拿到浏览器上使用**的程度了，**不需要在服务器端**做任何运算、**处理**。典型的静态资源包括：纯 HTML 文件、图片、CSS 文件、JavaScript 文件等。
- **访问静态资源问题**：直接**浏览器通过路径访问**出现 **404 访问不到**。
- **问题分析**：

* DispatcherServlet 的 url-pattern 配置的是“/”；
* **url-pattern 配置为“/”**表示**整个 Web 应用范围内所有请求都由 SpringMVC 来处理**；
* 对 SpringMVC 来说，**必须有对应的@RequestMapping 才能找到**处理请求的方法，而现在 **images/mi.jpg 请求没有对应的@RequestMapping** 所以返回 404。

- 问题解决：在 SpringMVC 配置配置类：当找不到 handler 方法时，继续找静态资源。

```
@EnableWebMvc  //json数据处理,必须使用此注解,因为他会加入json处理器
@Configuration
@ComponentScan(basePackages = "com.atguigu.controller") //TODO: 进行controller扫描
//WebMvcConfigurer springMvc进行组件配置的规范,配置组件,提供各种方法! 前期可以实现
public class SpringMvcConfig implements WebMvcConfigurer {

    //开启静态资源处理 <mvc:default-servlet-handler/>
    @Override
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer
configurer) {
        configurer.enable();
    }
}
```

【补充】springboot 静态资源处理(static-locations 属性指定位置)

在 WEB 开发中我们需要引入一些静态资源,例如:HTML,CSS,JS,图片等,如果是**普通的项目静态资源**可以**放在项目的 webapp 目录下**。现在使用 **Spring Boot 做开发**，项目中没有 webapp 目录，我们的**项目是一个 jar 工程**，那么就没有 webapp，一般**默认的静态资源路径有**：

- classpath:/META-INF/resources/
- classpath:/resources/
- classpath:/static/
- classpath:/public/

我们只要静态资源**放在这些目录中任何一个**,SpringMVC 都会帮我们处理，我们习惯会把静态资源放在 classpath:/static/目录下。

【注意】可以**设置 static-locations 属性**来**配置静态资源的位置**

```
spring:
  web:
    resources:
      # 配置静态资源地址,如果设置,会覆盖默认值
      static-locations: classpath:/webapp
```

4、RESTFul 风格设计

1）特点：

1> **每一个 URI 代表 1 种资源**（URI 是**名词**）；

2> 客户端使用 **GET**、**POST**、**PUT**、**DELETE** 4 个表示操作方式的动词对服务端资源进行操作：GET 用来**获取资源**，POST 用来**新建资源**（也可以用于**更新资源**），PUT 用来**更新资源**，DELETE 用来**删除资源**；

3> **资源的表现形式**是 **XML 或者 JSON**；

4> **客户端与服务端之间**的**交互**在**请求之间是无状态的**，从**客户端到服务端的每个请求**都***必须***包含理解请求所必需的信息**。

总结：根据接口的具体动作，选择具体的 HTTP 协议请求方式；路径设计从原来携带动标识，改成名词，对应资源的唯一标识即可！

功能	接口和请求方式	请求参数	返回值
分页查询	GET /user	page=1&size=10	{响应数据}
用户添加	POST /user	{user 数据}	{响应数据}
用户详情	GET /user/1	路径参数	{响应数据}
用户更新	PUT /user	{user 更新数据}	{响应数据}
用户删除	DELETE /user/1	路径参数	{响应数据}
条件模糊	GET /user/search	page=1&size=10&keywork=关键字	{响应数据}

2）一般使用方式：

1> 对**查询**用户详情，使用**路径传递参数**是因为这是一个**单一资源的查询**，即查询一条用户记录。**使用路径参数**可以明确指定所请求的资源。

2> 对**多条件模糊查询**，使用**请求参数传递参数**是因为这是一个**资源集合的查询**，即**查询多条用户记录**。**使用请求参数**可以通过组合不同参数来限制查询结果，**路径参数**的组合和排列可能会很多。

3> **路径参数**应该用于指定资源的**唯一标识**或者 **ID**，而**请求参数**应该用于指定**查询条件**或者**操作参数**。请求参数**不宜过多**。对于**敏感信息**，最好使用 **POST** 和**请求体**来传递参数。

```
package com.atguigu.pojo;

/**
 * projectName: com.atguigu.pojo
 * 用户实体类
 */
public class User {

    private Integer id;
    private String name;

    private Integer age;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
```

```
/**
 * projectName: com.atguigu.controller
 * description: 用户模块的控制器
 */
@RequestMapping("user")
@RestController
public class UserController {
    /**
     * 模拟分页查询业务接口
     */
    @GetMapping
    public Object queryPage(@RequestParam(name = "page",required = false,
                                          defaultValue = "1")int page,
                            @RequestParam(name = "size",required = false,
                                          defaultValue = "10")int size){
        System.out.println("page = " + page + ", size = " + size);
        System.out.println("分页查询业务!");
        return "{'status':'ok'}";
    }
```

```
    /**
     * 模拟用户保存业务接口
     */
    @PostMapping
    public Object saveUser(@RequestBody User user){
        System.out.println("user = " + user);
        System.out.println("用户保存业务!");
        return "{'status':'ok'}";
    }

    /**
     * 模拟用户详情业务接口
     */
    @PostMapping("/{id}")
    public Object detailUser(@PathVariable Integer id){
        System.out.println("id = " + id);
        System.out.println("用户详情业务!");
        return "{'status':'ok'}";
    }
```

```
    /**
     * 模拟用户更新业务接口
     */
    @PutMapping
    public Object updateUser(@RequestBody User user){
        System.out.println("user = " + user);
        System.out.println("用户更新业务!");
        return "{'status':'ok'}";
    }
```

```
    /**
     * 模拟条件分页查询业务接口
     */
    @GetMapping("search")
    public Object queryPage(@RequestParam(name = "page",required = false,
                                          defaultValue = "1")int page,
                            @RequestParam(name = "size",required = false,
                                          defaultValue = "10")int size,
                            @RequestParam(name="keyword",required=false)String keyword){
        System.out.println("page=" + page + ",size=" + size + ",keyword=" + keyword);
        System.out.println("条件分页查询业务!");
        return "{'status':'ok'}";
    }
}
```

5、全局异常处理机制

1）对于异常的处理，一般分为**两种方式**：

- **编程式**异常处理：是指在代码中显式地编写处理异常的逻辑。它通常涉及到对异常类型的检测及其处理，例如使用 **try-catch 块**来**捕获异常**，然后在 **catch 块**中编写**特定的处理代码**，或者在 **finally 块**中**执行一些清理操作**。在编程式异常处理中，开发人员**需要显式地进行异常处理，异常处理代码**混杂在业务代码中，导致代码可读性较差。
- **声明式**异常处理：则是将**异常处理的逻辑**从**具体的业务逻辑**中**分离**出来，通过**配置等方式**进行统一的管理和处理。在声明式异常处理中，开发人员**只需要为方法或类标注相应的注解（如@Throws 或@ExceptionHandler），**就可以处理特定类型的异常。相较于编程式异常处理，声明式异常处理可以使代码更加简洁、易于维护和扩展。

2）基于注解异常声明异常处理（三步）

1> **声明异常处理控制器类**：**@ControllerAdvice** 异常处理控制类，统一定义异常处理 handler 方法。

```
/**
 * @RestControllerAdvice = @ControllerAdvice + @ResponseBody
 * @ControllerAdvice 代表当前类的异常处理controller!
 */
@RestControllerAdvice
public class GlobalExceptionHandler {

}
```

2> **声明异常处理 hander 方法**：@ExceptionHandler(异常名.class)

*** 相同点：**

异常处理 handler 方法和**普通的 handler 方法**参数**接收**和**响应**都**一致**。

*** 区别在于：**

> **异常处理 handler 方法**要映射异常，**发生对应的异常**会调用；

> **普通的 handler 方法**要使用@RequestMapping 注解映射路径，**发生对应的路径**调用。

```
    * @param e 获取异常对象!
    * @return 返回handler处理结果!
    */
@ExceptionHandler(HttpMessageNotReadableException.class)
public Object handlerJsonDateException(HttpMessageNotReadableException e){

    return null;
}
```

```
/**
 * 当发生空指针异常会触发此方法!
 * @param e
 * @return
 */
@ExceptionHandler(NullPointerException.class)
public Object handlerNullException(NullPointerException e){

    return null;
}
```

```
/**
 * 所有异常都会触发此方法!但是如果有具体的异常处理Handler!
 * 具体异常处理Handler优先级更高!
 * 例如: 发生NullPointerException异常!
 *       会触发handlerNullException方法,不会触发handlerException方法!
 * @param e
 * @return
 */
@ExceptionHandler(Exception.class)
public Object handlerException(Exception e){

    return null;
}
```

【注意】**所有异常都会触发此方法** handlerException()，但是如果有具体的异常处理 Handler，**具体异常处理 Handler 优先级更高**。

3> 配置文件扫描控制器类配置：**确保异常处理控制类被扫描**

```
 <!-- 扫描controller对应的包,将handler加入到ioc-->
@ComponentScan(basePackages={"com.atguigu.controller","com.atguigu.exceptionhandler"})
```

6、拦截器

1）拦截器 Springmvc *VS* **过滤器 javaWeb：**

- **相似点：**

拦截：必须先把请求拦住，才能执行后续操作；

过滤：拦截器或过滤器存在的意义就是对请求进行统一处理；

放行：对请求执行了必要操作后，放请求过去，让它访问的资源。

- **不同点：**

1> **工作平台**不同

过滤器：工作在 **Servlet 容器**中；

拦截器：工作在 **SpringMVC** 的基础上；

2> 拦截的**范围**

过滤器：能够拦截到的**最大范围是整个 Web 应用**；

拦截器：能够拦截到的**最大范围是整个 SpringMVC 负责的请求**；

3> **IOC 容器支持**

过滤器：想得到 IOC 容器**需要调用专门的工具方法**，是间接的；

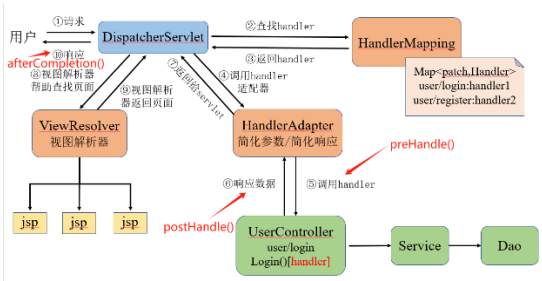
拦截器：它**自己就在 IOC 容器中**，所以可以**直接从 IOC 容器中装配组件**，也就是可以直接得到 IOC 容器的支持。

2）拦截器的使用

1> 创建拦截器类

```
public class Process01Interceptor implements HandlerInterceptor {

    // if( ! preHandler()){return;}
    // 在处理请求的目标 handler 方法前执行
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler) throws Exception {
        System.out.println("request = " + request + ", response = " + response + ",
handler = " + handler);
        System.out.println("Process01Interceptor.preHandle");

        // 返回true: 放行
        // 返回false: 不放行
        return true;
    }

    // 在目标 handler 方法之后, handler报错不执行!
    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response,
Object handler, ModelAndView modelAndView) throws Exception {
        System.out.println("request = " + request + ", response = " + response + ",
handler = " + handler + ", modelAndView = " + modelAndView);
        System.out.println("Process01Interceptor.postHandle");
    }

    // 渲染视图之后执行(最后),一定执行!
    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) throws Exception {
        System.out.println("request = " + request + ", response = " + response + ",
handler = " + handler + ", ex = " + ex);
        System.out.println("Process01Interceptor.afterCompletion");
    }
}
```

拦截类方法拦截的位置：



2> 修改配置类添加拦截器（springboot 中同样需要此操作）

```
@EnableWebMvc  //json数据处理,必须使用此注解,因为他会加入json处理器
@Configuration
@ComponentScan(basePackages =
{"com.atguigu.controller","com.atguigu.exceptionhandler"}) //TODO: 进行controller扫描
//WebMvcConfigurer springMvc进行组件配置的规范,配置组件,提供各种方法! 前期可以实现
public class SpringMvcConfig implements WebMvcConfigurer {

    //添加拦截器
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        //将拦截器添加到Springmvc环境,默认拦截所有Springmvc分发的请求
        registry.addInterceptor(new Process01Interceptor());
    }
}
```

3> **配置详解**：默认拦截全部、精准/模糊配置 addPathPatterns()、排除配置 excludePathPatterns()

```
//添加拦截器
@Override
public void addInterceptors(InterceptorRegistry registry) {

    //将拦截器添加到Springmvc环境,默认拦截所有Springmvc分发的请求
    registry.addInterceptor(new Process01Interceptor());

    //精准匹配,设置拦截器处理指定请求 路径可以设置一个或者多个,为项目下路径即可
    //addPathPatterns("/common/request/one") 添加拦截路径
    //也支持 /* 和 /** 模糊路径。 * 任意一层字符串 ** 任意层 任意字符串
    registry.addInterceptor(new Process01Interceptor())
            .addPathPatterns("/common/request/one","/common/request/two");

    //排除匹配,排除应该在匹配的范围内排除
    //addPathPatterns("/common/request/one") 添加拦截路径
    //excludePathPatterns("/common/request/tow"); 排除路径,排除应该在拦截的范围内
    registry.addInterceptor(new Process01Interceptor())
            .addPathPatterns("/common/request/one","/common/request/two")
            .excludePathPatterns("/common/request/tow");
}
```

4> 多个拦截器执行顺序

- preHandle()方法：SpringMVC 会把所有拦截器收集到一起，然后按照**配置顺序**调用各个 preHandle()方法。
- postHandle()方法：SpringMVC 会把所有拦截器收集到一起，然后按照**配置相反的顺序**调用各个 postHandle()方法。
- afterCompletion()方法：SpringMVC 会把所有拦截器收集到一起，然后按照**配置相反的顺序**调用各个 afterCompletion()方法。

【总结】：像洋葱，先声明的优先级高，优先级高的在外层， *一刀切*。

7、参数校验

在 Web 应用三层架构体系中，**表述层**负责**接收**浏览器提交的数据，**业务逻辑层**负责数据的**处理**。为了能够让业务逻辑层**基于正确的数据**进行处理，需要**在表述层检查数据**，将错误数据隔绝在业务逻辑层外。

1）校验概述

(JSR 303、Hibernate Validator、LocalValidatorFactoryBean)

JSR 303 是 Java 为 Bean 数据合法性校验提供的标准框架，它已经包含在 JavaEE 6.0 标准中。JSR 303 通过在 Bean 属性上标注类似于 @NotNull、@Max 等标准的注解指定校验规则，并通过标准的验证接口对 Bean 进行验证。

注解	规则
@Null	标注值必须为 null
@NotNull	标注值不可为 null
@AssertTrue	标注值必须为 true
@AssertFalse	标注值必须为 false
@Min(value)	标注值必须大于或等于 value
@Max(value)	标注值必须小于或等于 value
@DecimalMin(value)	标注值必须大于或等于 value
@DecimalMax(value)	标注值必须小于或等于 value
@Size(max,min)	标注值大小必须在 max 和 min 限定的范围内
@Digits(integer,fratction)	标注值值必须是一个数字，且必须在可接受的范围内
@Past	标注值只能用于日期型，且必须是过去的日期
@Future	标注值只能用于日期型，且必须是将来的日期
@Pattern(value)	标注值必须符合指定的正则表达式

JSR 303 只是一套标准，**需要提供其实现**才可以使用。**Hibernate Validator** 是 JSR 303 的**一个参考实现**，它还支持以下的扩展注解：

注解	规则
@Email	标注值必须是格式正确的 Email 地址
@Length	标注值字符串大小必须在指定的范围内
@NotEmpty	标注值字符串不能是空字符串
@Range	标注值必须在指定的范围内

Spring 4.0 版本已经拥有自己**独立的数据校验框架**，同时**支持 JSR 303** 标准的校验框架。Spring 的 LocalValidatorFactoryBean 既实现了 **Spring 的 Validator 接口**，也实现了 **JSR 303 的 Validator 接口**。Spring 本身并**没有提供 JSR 303 的实现**，所以必须**将 JSR 303 的实现者的 jar 包放到类路径下。**

在 SpringMVC 中，**配置@EnableWebMvc 后**，SpringMVC 会**默认装配好**一个 LocalValidatorFactoryBean，通过**在处理方法的形参上标注 @Validated 注解**即可让 SpringMVC 在完成数据绑定后执行数据校验。

2）易混总结

@NotNull、**@NotEmpty**、**@NotBlank** 都是用于在数据校验中**检查字段值是否为空**的注解，但是它们的用法和校验规则有所不同。

1> **@NotNull**（**包装类型**不为 null）

@NotNull 注解是 **JSR 303 规范中**定义的注解，当被标注的字段值为 null 时，会认为校验失败而抛出异常。该注解**不能用于字符串类型**的校验，若要对字符串进行校验，应该使用@NotBlank 或@NotEmpty 注解。

2> **@NotEmpty**(**集合类型**长度大于 0，全空格不是空字符串)

@NotEmpty 注解同样是 **JSR 303 规范中**定义的注解，对于 CharSequence、Collection、Map 或者数组对象类型的属性进行校验，校验时会检查该**属性是否为 Null** 或者 **size()==0**，如果是的话就会校验失败。但是**对于其他类型的属性，该注解无效**。需要注意的是只校验空格前后的字符串，如果该**字符串中间只有空格**，**不会被认为是空字符串**，校验不会失败。

3> **@NotBlank**（只用于**字符串**，null，或 “” 或全是空格的字符串）

@NotBlank 注解是 **Hibernate Validator 附加**的注解，对于**字符串类型的属性进行校验**，校验时会检查该属性是否为 Null 或 “” 或者只包含空格，如果是的话就会校验失败。需要注意的是，@NotBlank 注解只能用于字符串类型的校验。

3）使用

1> 导入依赖

```
<!-- 校验注解 -->
<dependency>
    <groupId>jakarta.platform</groupId>
    <artifactId>jakarta.jakartaee-web-api</artifactId>
    <version>9.1.0</version>
    <scope>provided</scope>
</dependency>

<!-- 校验注解实现-->
<!-- https://mvnrepository.com/artifact/org.hibernate.validator/hibernate-validator -->
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>8.0.0.Final</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.hibernate.validator/hibernate-validator-annotation-processor -->
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator-annotation-processor</artifactId>
    <version>8.0.0.Final</version>
</dependency>
```

2> **实体属性上**应用校验注解

```
import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.Min;
import org.hibernate.validator.constraints.Length;

/**
 * projectName: com.atguigu.pojo
 */
public class User {
    //age   1 <=  age < = 150
    @Min(10)
    private int age;

    //name 3 <= name.length <= 6
    @Length(min = 3,max = 10)
    private String name;

    //email 邮箱格式
    @Email
    private String email;
```

3> **handler 方法上**标记和绑定错误收集

【注意】在**实体类参数**和 **BindingResult 之间**不能有任何其他参数，**BindingResult** 可以接受错误信息，避免信息抛出

```
@RestController
@RequestMapping("user")
public class UserController {

    /**
     * @Validated 代表应用校验注解！ 必须添加！
     */
    @PostMapping("save")
    public Object save(@Validated @RequestBody User user,
                       //在实体类参数和 BindingResult 之间不能有任何其他参数，
                         BindingResult可以接受错误信息,避免信息抛出!
                       BindingResult result){
        //判断是否有信息绑定错误！ 有可以自行处理！
        if (result.hasErrors()){
            System.out.println("错误");
            String errorMsg = result.getFieldError().toString();
            return errorMsg;
        }
        //没有,正常处理业务即可
        System.out.println("正常");
        return user;
    }
}
```