Spring Boot 的整体结构是基于 分层架构 来设计的，每一层负责不同的职能，从而实现代码的模块化、可维护性、可扩展性。以下是常见的 Spring Boot 应用的主要层次结构：

**1. 控制层（Controller Layer）**

控制层负责接收和处理 HTTP 请求，并将请求转发到适当的业务逻辑层。控制层的主要职责是：

* 处理用户请求（如 GET、POST 等）
* 调用业务逻辑层（Service）进行处理
* 返回响应（通常是视图或 JSON 数据）

主要组件：@Controller、@RestController、@RequestMapping、@GetMapping、@PostMapping 等。

示例：

@Controller

public class MyController {

@Autowired

private MyService myService;

@GetMapping("/greeting")

public String greet(@RequestParam String name, Model model) {

String message = myService.greet(name);

model.addAttribute("message", message);

return "greeting";

}

}

注解：

* @GetMapping：用于处理 GET 请求，通常用于查询数据。
* @PostMapping：用于处理 POST 请求，通常用于创建数据。
* @PutMapping：用于处理 PUT 请求，通常用于更新数据。
* @DeleteMapping：用于处理 DELETE 请求，通常用于删除数据。
* @RequestMapping：是通用注解，可以处理所有 HTTP 请求类型，通过 method 属性指定请求方法。

**2. 业务逻辑层（Service Layer）**

服务层负责封装应用的业务逻辑，调用数据访问层（DAO）进行数据库操作。它是控制层和数据访问层之间的中介。服务层的主要职责是：

* 实现应用的业务逻辑
* 处理复杂的业务规则
* 调用数据访问层（Repository）进行数据处理

主要组件：@Service、@Transactional（用于处理事务）

示例：

@Service

public class MyService {

@Autowired

private MyRepository myRepository;

public String greet(String name) {

// 业务逻辑处理

return "Hello, " + name;

}

}

注解：

@Service: 标识 Service 类，Spring 会将其注册为 Bean。

@Transactional: 事务管理，确保操作的一致性和完整性。

@Autowired: 自动注入依赖。

@PostConstruct: 初始化方法，在 Bean 初始化后执行。

@PreDestroy: 清理方法，在 Bean 销毁前执行。

@Cacheable: 方法缓存，避免重复计算。

@Async: 异步方法执行，不阻塞主线程。

@Value: 注入配置文件中的值。

@Qualifier: 指定注入的具体 Bean。

**3. 数据访问层（Repository Layer）**（约等于mybatis的mapper接口）

数据访问层负责与数据库进行交互，通常包括 持久化 和 查询 等操作。Spring Boot 提供了方便的 Spring Data JPA 来简化数据库操作。数据访问层的主要职责是：

* 与数据库交互
* 执行 CRUD（创建、读取、更新、删除）操作
* 通过 JPA、JdbcTemplate 等进行数据访问

主要组件：@Repository、JpaRepository、CrudRepository、@Query 等。

示例：

@Repository

public interface MyRepository extends JpaRepository<MyEntity, Long> {

List<MyEntity> findByName(String name);

}

**4. 模型层（Model Layer）**（一般为实体类，对应数据库内部的表）

模型层负责封装应用的数据结构。通常它们映射到数据库表，称为 实体类。这些类使用 JPA 注解来定义表结构和字段映射，通常用于数据的持久化。

主要组件：@Entity、@Table、@Id、@Column、@OneToMany 等。

示例：

@Entity

public class MyEntity {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Column(name = "name")

private String name;

// getters and setters

}

注解：

* @Entity: 标记实体类。
* @Table: 指定实体类对应的数据库表名。
* @Id: 标识主键字段。
* @GeneratedValue: 定义主键的生成策略。
* @Column: 映射数据库表中的列。
* @OneToMany, @ManyToOne, @ManyToMany, @OneToOne: 定义实体间的关系。
* @JoinColumn: 配置外键列。
* @Transient: 标记不需要持久化的字段。
* @Enumerated: 定义枚举的存储方式。
* @Lob: 存储大字段（如文本或二进制数据）。

**5. 配置层（Configuration Layer）**

配置层通常包含 Spring Boot 应用的各类配置和设置。Spring Boot 通过 application.properties 或 application.yml 配置文件来设置应用的参数，如数据库连接、端口、日志级别等。

此外，可以通过 @Configuration 注解的类来进行 Java 配置。

主要组件：@Configuration、@PropertySource、@Bean、@Profile、@EnableAutoConfiguration 等。

示例：

@Configuration

public class MyConfiguration {

@Bean

public MyBean myBean() {

return new MyBean();

}

}

**6. 过滤器层（Filter Layer）**

过滤器用于对请求和响应进行处理，可以在控制器方法执行前后对请求和响应进行预处理。常见的操作有日志记录、权限验证、请求计时等。

主要组件：Filter、@WebFilter、@Component。

示例：

@Component

public class MyFilter implements Filter {

@Override

public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)

throws IOException, ServletException {

// 过滤逻辑

chain.doFilter(request, response);

}

@Override

public void init(FilterConfig filterConfig) throws ServletException {}

@Override

public void destroy() {}

}

**7. 拦截器层（Interceptor Layer）**

Spring 的拦截器机制与过滤器相似，都是用来在请求和响应的生命周期中执行特定操作。拦截器通常用于日志、权限、性能监控等。

主要组件：HandlerInterceptor、@Component。

示例：

@Component

public class MyInterceptor implements HandlerInterceptor {

@Override

public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)

throws Exception {

// 拦截逻辑

return true; // 继续执行

}

@Override

public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,

ModelAndView modelAndView) throws Exception {}

@Override

public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler,

Exception ex) throws Exception {}

}

**8. 异常处理层（Exception Handling Layer）**

Spring Boot 提供了全局异常处理机制，通常使用 @ControllerAdvice 和 @ExceptionHandler 来集中处理异常。这有助于统一管理应用中的异常，简化错误处理。

主要组件：@ControllerAdvice、@ExceptionHandler、@ResponseStatus。

示例：

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(ResourceNotFoundException.class)

@ResponseStatus(HttpStatus.NOT\_FOUND)

public String handleResourceNotFoundException(ResourceNotFoundException ex) {

return ex.getMessage();

}

}

**9. 安全层（Security Layer）**

Spring Security 用于保护 Spring 应用的安全性。它提供了身份验证、授权、CSRF 防护、会话管理等功能。你可以自定义认证机制，管理用户角色，或者进行权限控制。

主要组件：@EnableWebSecurity、WebSecurityConfigurerAdapter、@PreAuthorize 等。

示例：

@Configuration

@EnableWebSecurity

public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http.authorizeRequests()

.antMatchers("/admin/\*\*").hasRole("ADMIN")

.anyRequest().authenticated()

.and()

.formLogin();

}

}

Spring Boot 典型的分层结构总结：

* Controller Layer：处理 HTTP 请求，调用业务逻辑层，返回响应。
* Service Layer：实现具体的业务逻辑，调用数据访问层进行操作。
* Repository Layer：负责数据持久化和数据库操作。
* Model Layer：数据结构，通常是数据库实体类。
* Configuration Layer：应用的配置，设置 Bean、环境参数等。
* Filter/Interceptor Layer：对请求和响应进行预处理、后处理。
* Exception Handling Layer：处理应用中的异常，统一响应错误信息。
* Security Layer：保护应用的安全，进行身份验证和授权。

这种分层架构可以使得代码更加模块化，层次清晰，并且具有良好的可维护性和扩展性。不同的开发人员可以专注于不同的层次进行开发。