

# 谈谈 Web 那些事



扫码试看/订阅  
《玩转 Spring 全家桶》

# Spring MVC 实践

# 编写第一个Spring MVC Controller

# 认识 Spring MVC

## DispatcherServlet

- Controller
- xxxResolver
  - ViewResolver
  - HandlerExceptionResolver
  - MultipartResolver
- HandlerMapping 请求映射处理

# Spring MVC 中的常用注解

- @Controller
  - @RestController
- @RequestMapping controller处理什么请求  
定义在类上面的就是定义在方法上的路径的前缀
  - @GetMapping / @PostMapping
  - @PutMapping / @DeleteMapping
- @RequestBody / @ResponseBody / @ResponseStatus  
请求报文体                      响应报文体                      响应Http代码

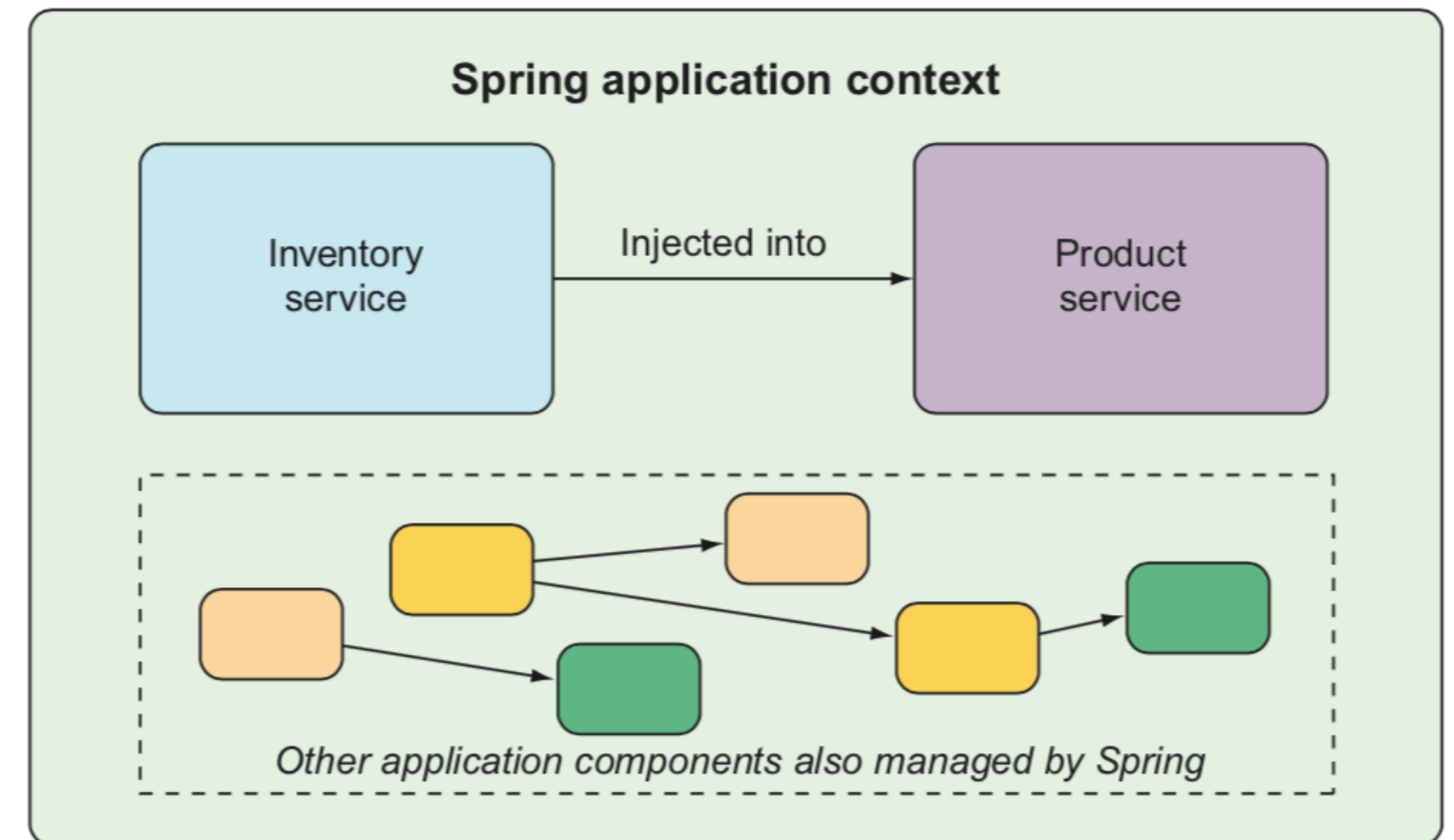
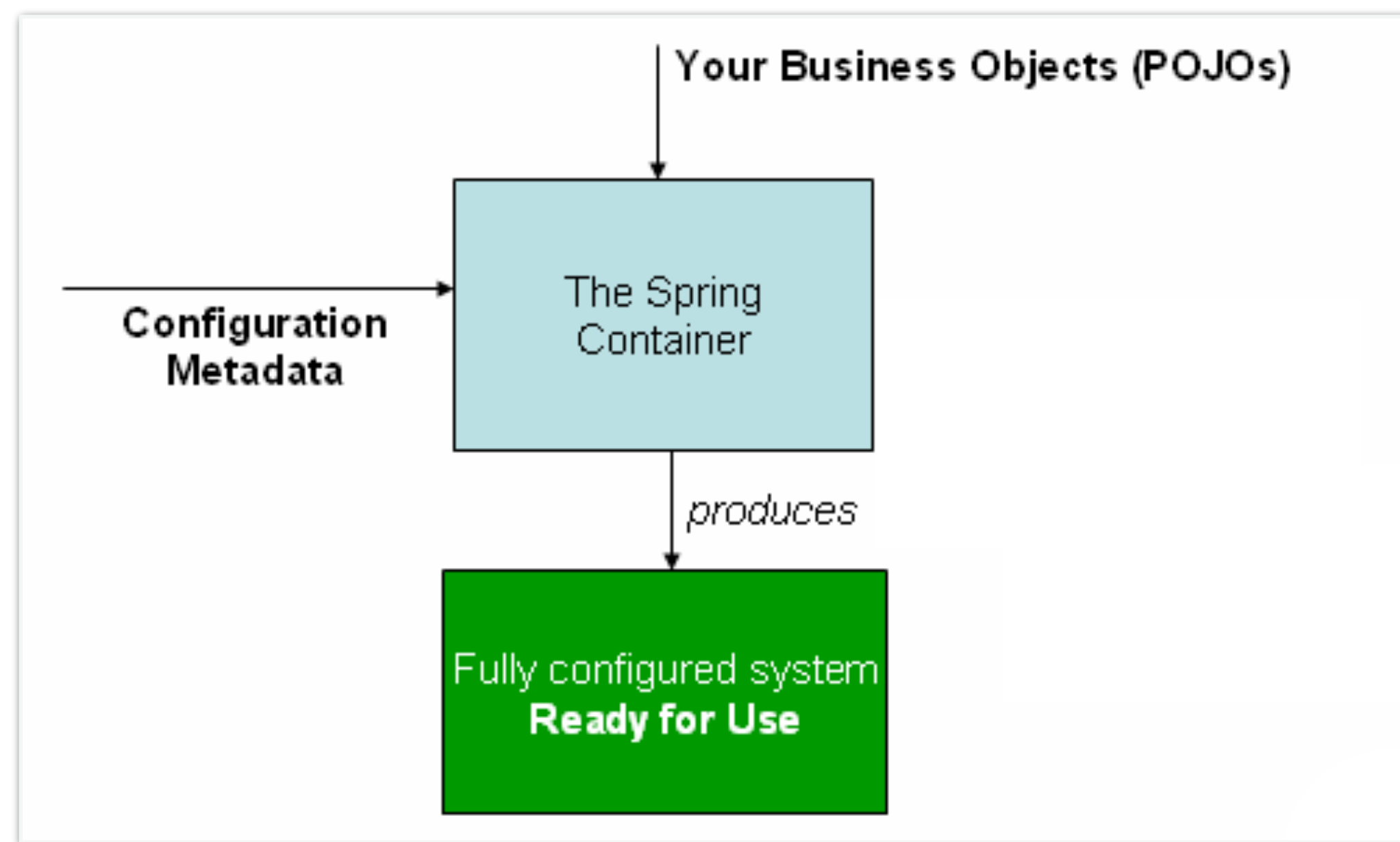
**“Talk is cheap, show me the code.”**

*Chapter 6 / simple-controller-demo*

# 理解 Spring 的应用上下文



# Spring 的应用程序上下文



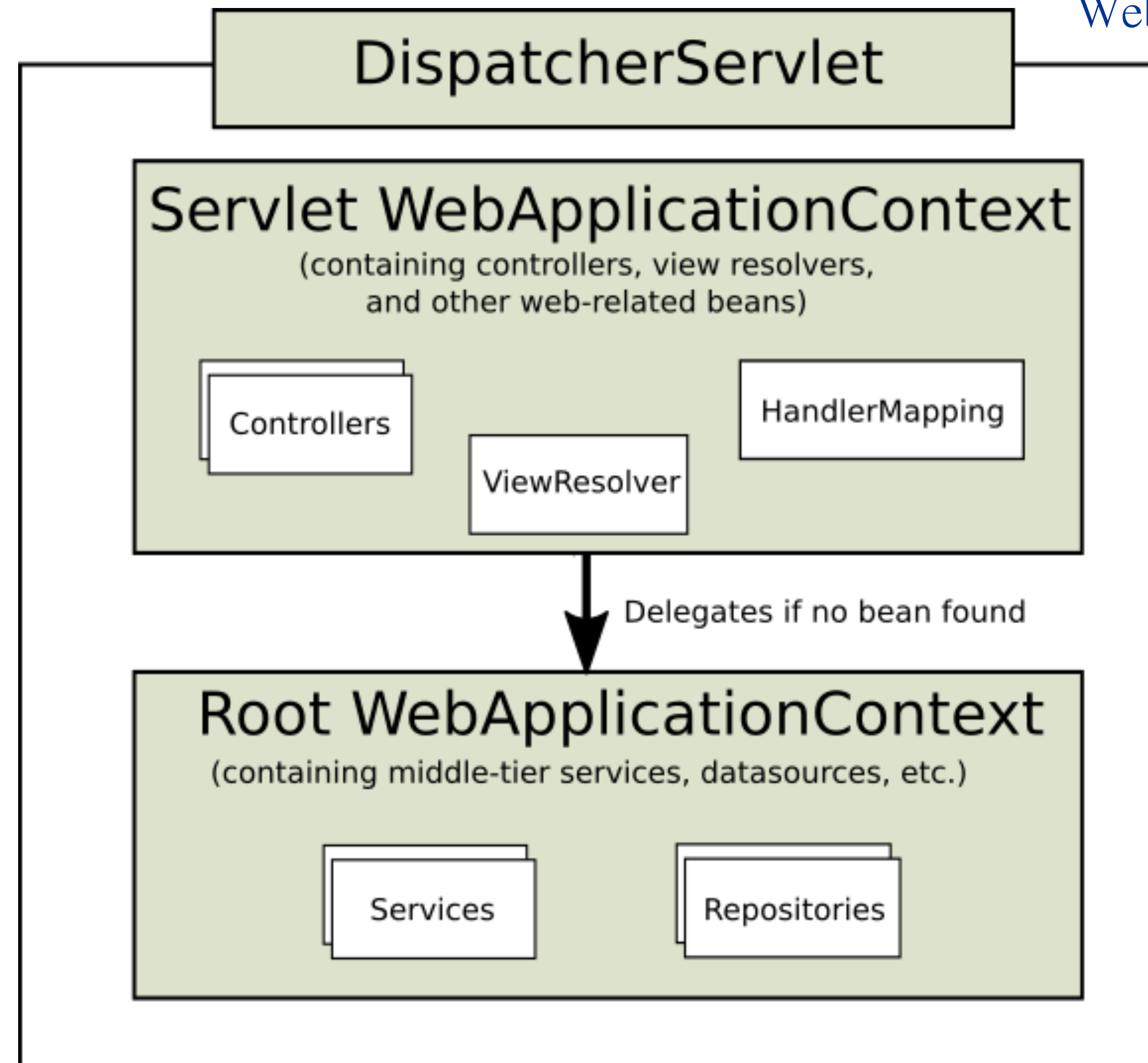
# Spring 的应用程序上下文

## 关于上下文常用的接口

- BeanFactory      一般不用这个，没有特殊原因请使用ApplicationContext
  - DefaultListableBeanFactory
- ApplicationContext
  - ClassPathXmlApplicationContext
  - FileSystemXmlApplicationContext
  - AnnotationConfigApplicationContext
- WebApplicationContext

# Web 上下文层次

如果AOP拦截在servlet web  
ApplicationContext里面，但这个时候  
并不会拦截到root  
WebApplicationContext



# Web 上下文层次

基于注解

```
<web-app>

  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-
class>
    root config
  </listener>

  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/app-context.xml</param-value>
  </context-param>

  <servlet>
    Servlet config
    <servlet-name>app</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value></param-value>应用service,dao由contextLoaderListener加载
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>app</servlet-name>
    <url-pattern>/app/*</url-pattern>
  </servlet-mapping>

</web-app>
```

```
public class MyWebAppInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {

  @Override
  protected Class<?>[] getRootConfigClasses() {
    return new Class<?>[] { RootConfig.class };
  }

  @Override
  protected Class<?>[] getServletConfigClasses() {
    return new Class<?>[] { App1Config.class };
  }

  @Override
  protected String[] getServletMappings() {
    return new String[] { "/app1/*" };
  }
}
```

基于xml的

**“Talk is cheap, show me the code.”**

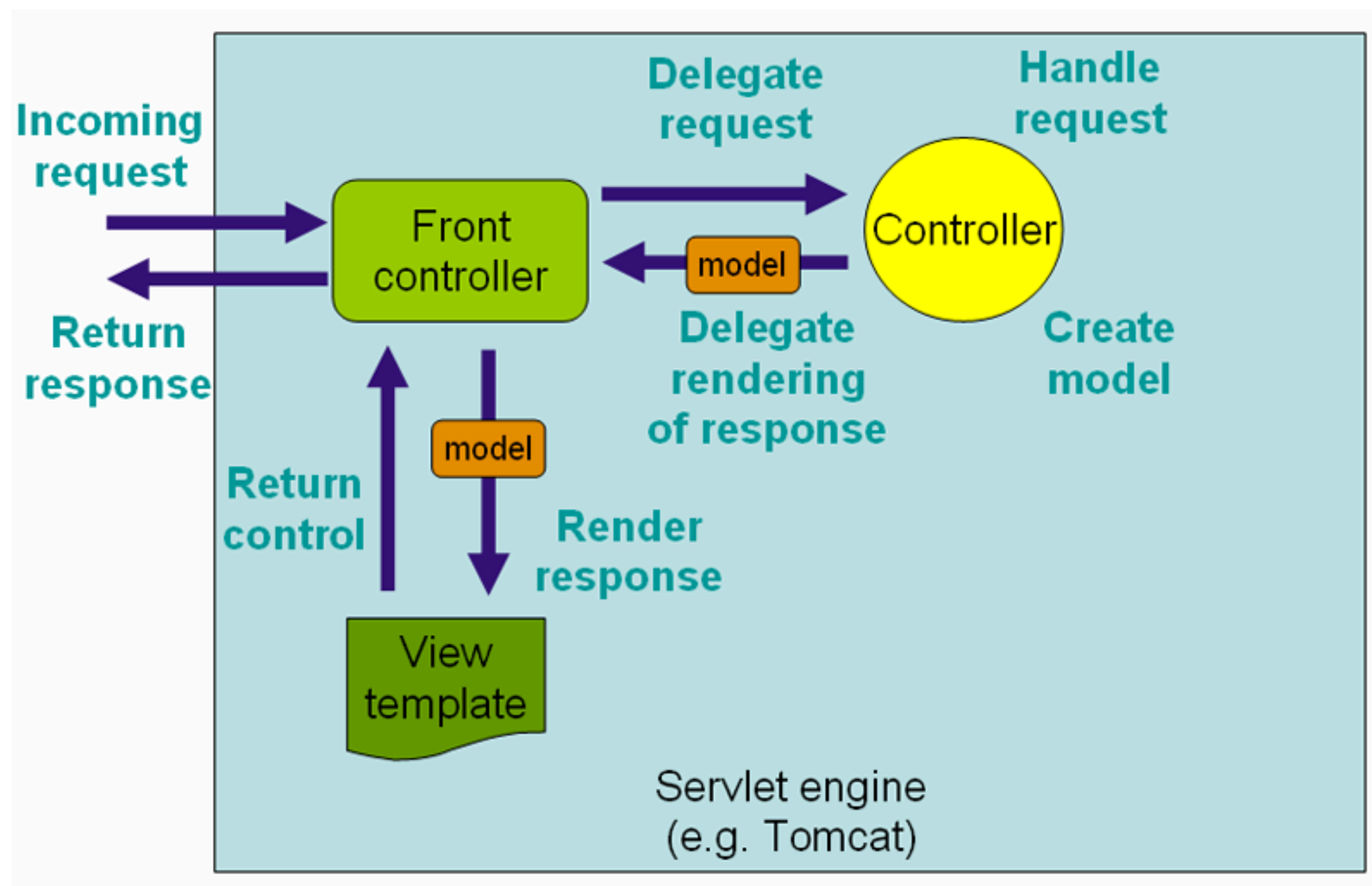
*Chapter 6 / context-hierarchy-demo*

# Spring MVC 中的各种机制

## 请求处理



## Spring MVC 的请求处理流程



# 一个请求的大致处理流程

## 绑定一些 Attribute

- WebApplicationContext / LocaleResolver / ThemeResolver

## 处理 Multipart

- 如果是，则将请求转为 MultipartHttpServletRequest

## Handler 处理

- 如果找到对应 Handler，执行 Controller 及前后置处理器逻辑

## 处理返回的 Model，呈现视图



# 如何定义处理方法

# 定义映射关系

**@Controller**

**@RequestMapping**

- path / method 指定映射路径与方法
- params / headers 限定映射范围
- consumes / produces 限定请求与响应格式

一些快捷方式

- @RestController
- @GetMapping / @PostMapping / @PutMapping / @DeleteMapping / @PatchMapping

## 定义处理方法

- `@RequestBody` / `@ResponseBody` / `@ResponseStatus`
- `@PathVariable` / `@RequestParam` / `@RequestHeader`
- `HttpEntity` / `ResponseEntity`

# 定义处理方法

## 详细参数

- <https://docs.spring.io/spring/docs/5.1.5.RELEASE/spring-framework-reference/web.html#mvc-ann-arguments>

## 详细返回

- <https://docs.spring.io/spring/docs/5.1.5.RELEASE/spring-framework-reference/web.html#mvc-ann-return-types>

## 方法示例

```
@PostMapping(path = "/", consumes = MediaType.APPLICATION_JSON_VALUE,  
              produces = MediaType.APPLICATION_JSON_UTF8_VALUE)  
@ResponseStatus(HttpStatus.CREATED)  
public CoffeeOrder create(@RequestBody NewOrderRequest newOrder) {  
    log.info("Receive new Order {}", newOrder);  
    Coffee[] coffeeList = coffeeService.getCoffeeByName(newOrder.getItems())  
        .toArray(new Coffee[] {});  
    return orderService.createOrder(newOrder.getCustomer(), coffeeList);  
}
```

## 方法示例

```
@RequestMapping(path =("/{id}", method = RequestMethod.GET,  
    produces = MediaType.APPLICATION_JSON_UTF8_VALUE)  
@ResponseBody  
public Coffee getById(@PathVariable Long id) {  
    Coffee coffee = coffeeService.getCoffee(id);  
    return coffee;  
}  
  
@GetMapping(path = "/", params = "name")  
@ResponseBody  
public Coffee getName(@RequestParam String name) {  
    return coffeeService.getCoffee(name);  
}
```

**“Talk is cheap, show me the code.”**

*Chapter 6 / complex-controller-demo*

# 定义类型转换

自己实现 WebMvcConfigurer

- Spring Boot 在 WebMvcAutoConfiguration 中实现了一个
- 添加自定义的 Converter
- 添加自定义的 Formatter



# 定义校验

- 通过 Validator 对绑定结果进行校验
  - Hibernate Validator
- @Valid 注解
- BindingResult

# Multipart 上传

- 配置 `MultipartResolver`
  - Spring Boot 自动配置 `MultipartAutoConfiguration`
- 支持类型 `multipart/form-data`
- `MultipartFile` 类型

**“Talk is cheap, show me the code.”**

*Chapter 6 / more-complex-controller-demo*

# Spring MVC 中的各种机制

## 视图处理

# 视图解析的实现基础

## ViewResolver 与 View 接口

- AbstractCachingViewResolver
- UrlBasedViewResolver
- FreeMarkerViewResolver
- ContentNegotiatingViewResolver
- InternalResourceViewResolver

# DispatcherServlet 中的视图解析逻辑

- `initStrategies()`
  - `initViewResolvers()` 初始化了对应 `ViewResolver`
- `doDispatch()`
  - `processDispatchResult()`
    - 没有返回视图的话, 尝试 `RequestToViewNameTranslator`
    - `resolveViewName()` 解析 `View` 对象

# DispatcherServlet 中的视图解析逻辑

使用 @ResponseBody 的情况

- 在 `HandlerAdapter.handle()` 的中完成了 Response 输出
- `RequestMappingHandlerAdapter.invokeHandlerMethod()`
- `HandlerMethodReturnValueHandlerComposite.handleReturnValue()`
- `RequestResponseBodyMethodProcessor.handleReturnValue()`

# 重定向

## 两种不同的重定向前缀

- redirect:
- forward:



# Spring MVC 中的常用视图

# Spring MVC 支持的视图

## 支持的视图列表

- <https://docs.spring.io/spring/docs/5.1.5.RELEASE/spring-framework-reference/web.html#mvc-view>
- Jackson-based JSON / XML
- Thymeleaf & FreeMarker

### Template Engines

- ☐ Thymeleaf  
Thymeleaf templating engine
- ☐ Freemarker  
FreeMarker templating engine
- ☐ Mustache  
Mustache templating engine
- ☐ Groovy Templates  
Groovy templating engine

## 配置 MessageConverter

- 通过 WebMvcConfigurer 的 configureMessageConverters()
- Spring Boot 自动查找 HttpMessageConverters 进行注册

```
public class WebConfiguration implements WebMvcConfigurer {  
  
    @Override  
    public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {  
        Jackson2ObjectMapperBuilder builder = new Jackson2ObjectMapperBuilder()  
            .indentOutput(true)  
            .dateFormat(new SimpleDateFormat("yyyy-MM-dd"))  
            .modulesToInstall(new ParameterNamesModule());  
        converters.add(new MappingJackson2HttpMessageConverter(builder.build()));  
        converters.add(new  
MappingJackson2XmlHttpMessageConverter(builder.createXmlMapper(true).build()));  
    }  
}
```

# Spring Boot 对 Jackson 的支持

- JacksonAutoConfiguration
  - Spring Boot 通过 @JsonComponent 注册 JSON 序列化组件
  - Jackson2ObjectMapperBuilderCustomizer
- JacksonHttpMessageConvertersConfiguration
  - 增加 jackson-dataformat-xml 以支持 XML 序列化

**“Talk is cheap, show me the code.”**

*Chapter 6 / json-view-demo*

“Thymeleaf is a modern server-side Java template engine for both web and standalone environments.”

– <https://www.thymeleaf.org/>

# 使用 Thymeleaf

## 添加 Thymeleaf 依赖

- `org.springframework.boot:spring-boot-starter-thymeleaf`

## Spring Boot 的自动配置

- `ThymeleafAutoConfiguration`
  - `ThymeleafViewResolver`

# Thymeleaf 的一些默认配置

- `spring.thymeleaf.cache=true`
- `spring.thymeleaf.check-template=true`
- `spring.thymeleaf.check-template-location=true`
- `spring.thymeleaf.enabled=true`
- `spring.thymeleaf.encoding=UTF-8`
- `spring.thymeleaf.mode=HTML`
- `spring.thymeleaf.servlet.content-type=text/html`
- `spring.thymeleaf.prefix=classpath:/templates/`
- `spring.thymeleaf.suffix=.html`



**“Talk is cheap, show me the code.”**

*Chapter 6 / thymeleaf-view-demo*

# 静态资源与缓存

# Spring Boot 中的静态资源配置

## 核心逻辑

- `WebMvcConfigurer.addResourceHandlers()`

## 常用配置

- `spring.mvc.static-path-pattern=/**`
- `spring.resources.static-locations=classpath:/META-INF/resources/,classpath:/resources/,classpath:/static/,classpath:/public/`

# Spring Boot 中的缓存配置

常用配置（默认时间单位都是秒）

- `ResourceProperties.Cache`
- `spring.resources.cache.cachecontrol.max-age=时间`
- `spring.resources.cache.cachecontrol.no-cache=true/false`
- `spring.resources.cache.cachecontrol.s-max-age=时间`

## Controller 中手工设置缓存

```
@GetMapping("/book/{id}")
public ResponseEntity<Book> showBook(@PathVariable Long id) {

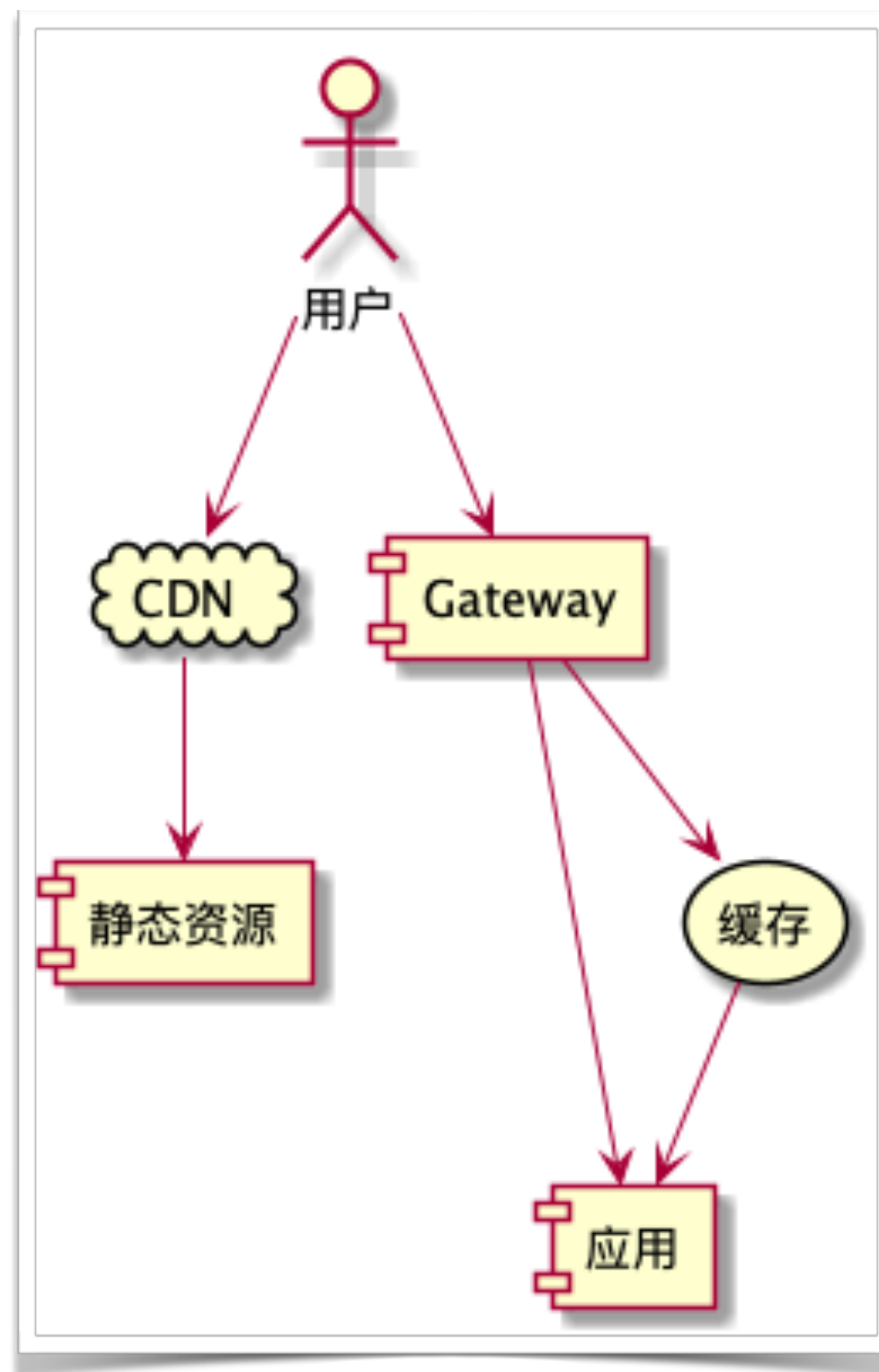
    Book book = findBook(id);
    String version = book.getVersion();

    return ResponseEntity
        .ok()
        .cacheControl(CacheControl.maxAge(30, TimeUnit.DAYS))
        .eTag(version) // lastModified is also available
        .body(book);
}
```

**“Talk is cheap, show me the code.”**

*Chapter 6 / cache-demo*

## 建议的资源访问方式



# Spring MVC 中的各种机制

异常处理



# Spring MVC 的异常解析

## 核心接口

- `HandlerExceptionResolver`

## 实现类

- `SimpleMappingExceptionHandlerResolver`
- `DefaultHandlerExceptionHandlerResolver`
- `ResponseStatusExceptionHandlerResolver`
- `ExceptionHandlerExceptionHandlerResolver`

# 异常处理方法

## 处理方法

- `@ExceptionHandler`

## 添加位置

- `@Controller` / `@RestController`
- `@ControllerAdvice` / `@RestControllerAdvice`

**“Talk is cheap, show me the code.”**

*Chapter 6 / exception-demo*

# 了解 Spring MVC 的切入点

# Spring MVC 的拦截器

## 核心接口

- HandlerInteceptor
  - `boolean preHandle()`
  - `void postHandle()`
  - `void afterCompletion()`

# Spring MVC 的拦截器

针对 **@ResponseBody** 和 **ResponseEntity** 的情况

- `ResponseBodyAdvice`

针对异步请求的接口

- `AsyncHandlerInterceptor`
  - `void afterConcurrentHandlingStarted()`

# 拦截器的配置方式

## 常规方法

- `WebMvcConfigurer.addInterceptors()`

## Spring Boot 中的配置

- 创建一个带 `@Configuration` 的 `WebMvcConfigurer` 配置类
- 不能带 `@EnableWebMvc` (想彻底自己控制 MVC 配置除外)

**“Talk is cheap, show me the code.”**

*Chapter 6 / springbucks*



# SpringBucks 进度小结

## 本章小结

- 解释了什么是 Spring 的 ApplicationContext
- 了解了 Spring MVC 的基本使用
- 理解 Spring MVC 的多种机制

# SpringBucks 进度小结

- 拆分了 waiter-service
- 增加了更多 REST 方法
- 增加了缓存、性能日志与异常处理



扫码试看/订阅  
《玩转 Spring 全家桶》