

Spring Application Framework Overview 2

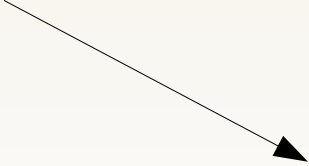


Annotasyon Tabanlı Container Konfigürasyonu

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans ...>
```

```
<context:annotation-config/>
```

```
</beans>
```

 @PostConstruct, @PreDestroy, @Required, @Autowired gibi
annotasyonları devreye sokar

**Bu aşamada henüz bean tanımları XML konfigürasyon
dosyalarında yapılmaya devam etmektedir!**

@Autowired

```
public class Foo {  
    private Bar bar;
```

ApplicationContext'de tanımlı Bar tipin'deki bean'ı enjekte eder

```
    @Autowired  
    public void setBar(Bar bar) {  
        this.bar = bar;  
    }  
  
    // ...  
}
```

```
<beans...>  
    <bean id="bar" class="x.y.Bar"/>  
  
    <bean id="foo" class="x.y.Foo"/>  
</beans>
```

@Autowired

```
public class Foo {  
  
    @Autowired  
    private Bar bar;  
  
    private Baz baz;  
  
    @Autowired  
    public Foo(Baz baz) {  
        this.baz = baz;  
    }  
  
    // ...  
}
```

Field, setter ve constructor'a uygulanabilir

Default olarak **byType** modunda çalışır

Birden fazla aynı tipte bean olması ve bu bean'lardan herhangi birisinin ismi property ile eşleşmediği durumda hata verir

Qualifier ile Aday Bean'ların Sınırlandırılması

```
public class Foo {
```

```
    @Autowired  
    @Qualifier("myBar")  
    private Bar bar;
```

```
    // ...
```

```
}
```

```
<bean id="foo" class="x.y.Foo"/>
```

```
<bean id="bar1" class="x.y.Bar">  
</bean>
```

```
<bean id="bar2" class="x.y.Bar">  
    <qualifier value="myBar"/>  
</bean>
```

Eğer tanımlanmaz ise **default qualifier** değeri olarak bean ismi kabul edilir

@Autowired

```
public class Foo {

    private Bar bar;

    @Autowired(required=false)
    public void setBar(Bar bar) {
        this.bar = bar;
    }

    // ...
}
```

Default **required** attribute değeri **true**'dür
required=true durumunda Spring Container
belirtilen tipte bean bulamadığında
hata verir. Property'nin NULL kalması için
required=false olarak belirtilmelidir.

ApplicationContext'deki Belirli Tipteki Bütün Bean'ları Enjekte Etmek

```
public class Foo {
```

```
    @Autowired  
    private Bar[] bars;
```

```
    // ...
```

```
}
```

→ Bu sayede container'da tanımlı **belirli bir tipteki bütün beanları** bir array'e autowire etmek de mümkündür

Eğer ApplicationContext'de Bar tipinde hiç bean yoksa hata verir


ApplicationContext'deki Belirli Tipteki Bütün Bean'ları Enjekte Etmek

```
public class Foo {  
  
    private Set<Bar> bars;  
  
    @Autowired  
    public void setBars(Set<Bar> bars) {  
        this.bars = bars;  
    }  
  
    // ...  
}
```

Enjekte edilecek bean'lerin tipini java generics'den tespit edebilir

ApplicationContext'i Enjekte Etmek

```
public class Foo {  
  
    @Autowired  
    private ApplicationContext context;  
  
    // ...  
}
```



ApplicationContext kendisini bu bean'a enjekte eder. **ApplicationContextAware** arayüzünü implement etme gereksinimini ortadan kaldırır.

Component Scan İşlemi

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans ...>
```

```
    <context:component-scan  
        base-package ="com.javaegitimleri"/>
```

```
</beans>
```



<context:annotation-config/> elemanını da otomatik olarak aktive eder. Dolayısı ile bu elemanı tanımlaya artık gerek yoktur.

Component Scan İşlemi

- Default olarak aşağıdaki **built-in anotasyonlar** scan edilir
 - @Component
 - @Repository
 - @Service
 - @Controller ve @RestController
 - @ControllerAdvice
 - @Configuration

Annotasyon Tabanlı Bean Tanımlamaya Örnek

```
@Service("securityService")
public class SecurityServiceImpl implements
SecurityService {

    private SecurityDao securityDao;

    @Autowired
    public SecurityServiceImpl(SecurityDao securityDao) {
        this.securityDao = securityDao;
    }
}

@Repository
public class SecurityDaoImpl implements SecurityDao {
    // ...
}
```

Component İçinde Bean Tanımları

- Component'ler business metotları dışında **bean factory metotları** da barındırabilir

@Component

```
public class FooFactory {
```

Metot düzeyinde tanımlama **@Bean** annotasyonu ile gerçekleştirilir

```
@Bean @Qualifier("myFoo")
```

```
public Foo foo() {  
    return new Foo();  
}
```

Metot ismi bean ismi olur,
Ayrıca Qualifier'da tanımlanabilir

```
public void doWork() {  
    // ...  
}
```

Component normal bir bean instance'ıdır ve bean yaratma dışında normal işlemlere de sahip olabilir

```
}
```

@Value

- Built-in Java tipli **property değerlerini** enjekte etmek için @Value anotasyonu kullanılır
- İçerisinde property **placeholder** da kullanılabilir

```
@Component
public class Foo {

    @Value("bar-value")
    private String bar;

    @Value("${foo.baz}")
    private String baz;

    // ...
}
```

@Scope

@Scope anotasyonu bean tanımında metot veya sınıf düzeyinde kullanılabilir

```
@Scope("prototype")
@Component
public class CommandImpl implements Command {
    // ...
}
```

```
@Component
public class CommandFactory {

    @Bean @Scope("prototype")
    public Command createCommand() {
        return new CommandImpl();
    }
}
```

@Lazy

```
@Component
@Lazy
public class FooFactory {

    @Bean @Lazy
    private Foo foo() {
        return new Foo();
    }
}
```

Sınıf veya metot düzeyinde **@Lazy** anotasyonu kullanılarak bean'lerin sadece gerektiği anda yaratılmaları sağlanabilir

XML vs Java Konfigürasyonları

appContextConfig.xml

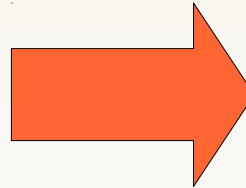
```
<beans...>
```

```
<bean id="foo" class="x.y.Foo">
  <property name="bar">
    <ref bean="bar"/>
  </property>
  <property name="baz"
ref="baz"/>
</bean>
```

```
<bean id="bar"
class="x.y.Bar"/>
```

```
<bean id="baz"
class="x.y.Baz"/>
```

```
</beans>
```



@Configuration

```
public class AppContextConfig {
```

@Bean

```
public Foo foo() {
    Foo foo = new Foo();
    foo.setBar(bar());
    foo.setBaz(baz());
    return foo;
}
```

@Bean

```
public Bar bar() {
    return new Bar();
}
```

@Bean

```
public Baz baz() {
    return new Baz();
}
```

```
}
```

Bağımlılıkların Enjekte Edilmesi

```
@Configuration
public class AConfig {

    @Autowired
    private Bar bar;

    @Bean
    public Foo foo() {
        Foo foo = new Foo();
        foo.setBar(bar);
        return foo;
    }
}
```

```
@Configuration
public class BConfig {

    @Bean
    public Bar bar() {
        return new Bar();
    }
}
```

Bağımlılıkların Enjekte Edilmesi

```
@Configuration  
public class AConfig {
```

```
    @Autowired
```

```
    private BConfig bConfig;
```

```
    @Bean
```

```
    public Foo foo() {  
        Foo foo = new Foo();  
        foo.setBar(bConfig.bar());  
        return foo;  
    }
```

```
}
```

```
@Configuration
```

```
public class BConfig {
```

```
    @Bean
```

```
    public Bar bar() {  
        return new Bar();  
    }
```

```
}
```

Bağımlılıkların Enjekte Edilmesi

```
@Configuration
public class AConfig {

    @Bean
    public Foo foo(Bar bar) {
        Foo foo = new Foo();
        foo.setBar(bar);
        return foo;
    }
}
```

```
@Configuration
public class BConfig {

    @Bean
    public Bar bar() {
        return new Bar();
    }
}
```

ApplicationContext Yaratılması

```
AnnotationConfigApplicationContext applicationContext =  
new AnnotationConfigApplicationContext();  
  
applicationContext.register(AConfig.class,BConfig.class);  
  
applicationContext.refresh();  
  
Foo foo = applicationContext.getBean(Foo.class);
```

@ImportResource ve @Import

@ImportResource herhangi bir XML
spring bean definition dosyasının
yüklenmesini sağlar

```
@Configuration
@ImportResource("classpath:/appcontext/beans-config.xml")
public class AConfig {

}
```

```
@Configuration
@Import(AConfig.class)
public class BConfig {

}
```

@Import herhangi bir başka configuration
sınıfının diğer bir configuration sınıfı
tarafından yüklenmesini sağlar

@ComponentScan

@Component ve türevi
anotasyonların scan edileceği
paketleri tanımlar

```
@Configuration
@ComponentScan("com.javaegitimleri.petclinic")
public class AConfig {

}
```

Web Uygulamalarında ApplicationContext Oluşturma

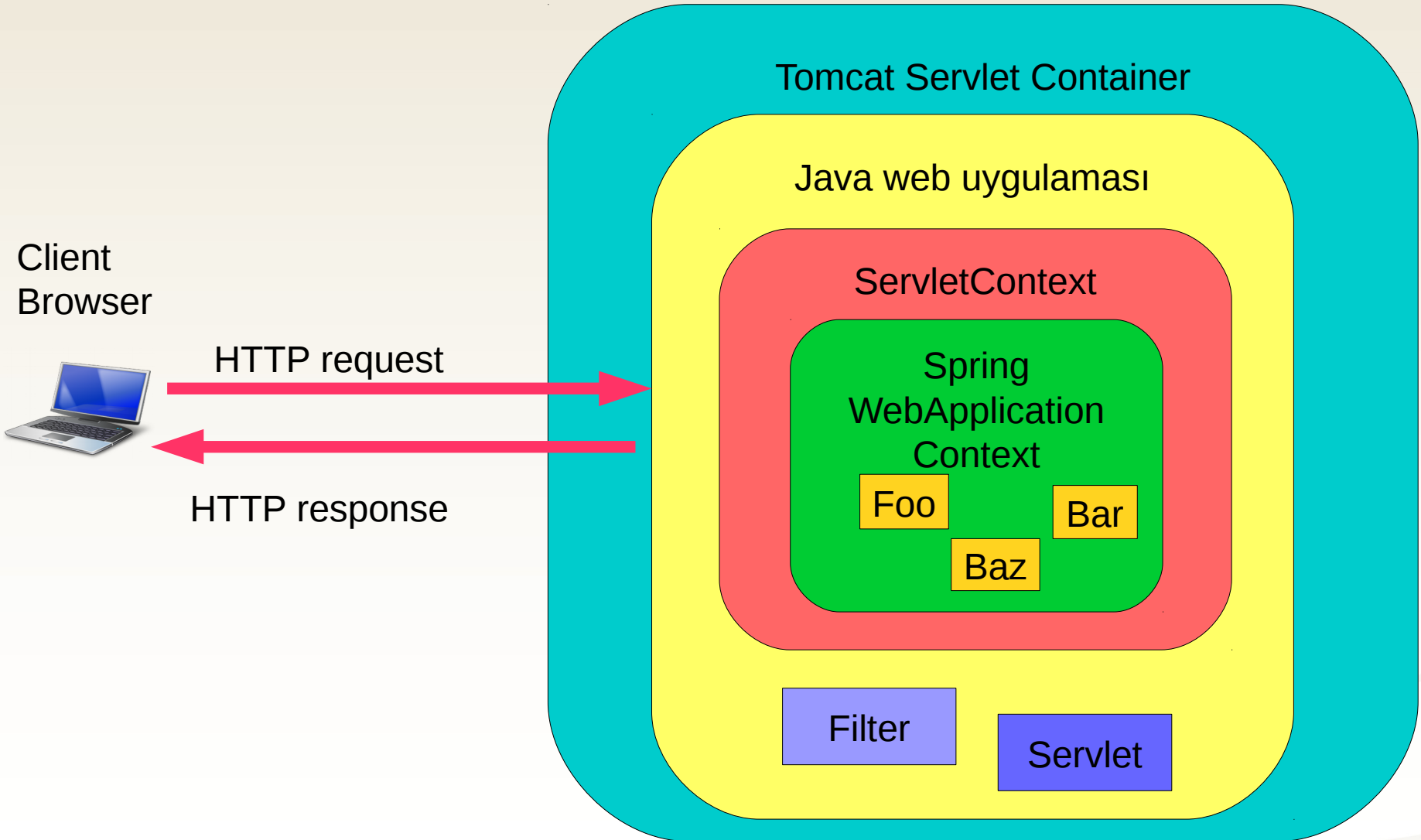
web.xml

```
<web-app>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      classpath*: /appcontext/beans-*.xml
    </param-value>
  </context-param>

  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>
</web-app>
```

- Default: /WEB-INF/applicationContext.xml
- Eğer parametre mevcut ise birden fazla dosya path'i belirtilebilir
- Path'ler virgül, noktalı virgül veya boşluk ile ayrılabilir
- Ant-style path pattern'ları da desteklenir

ServletContext & WebApplicationContext İlişkisi

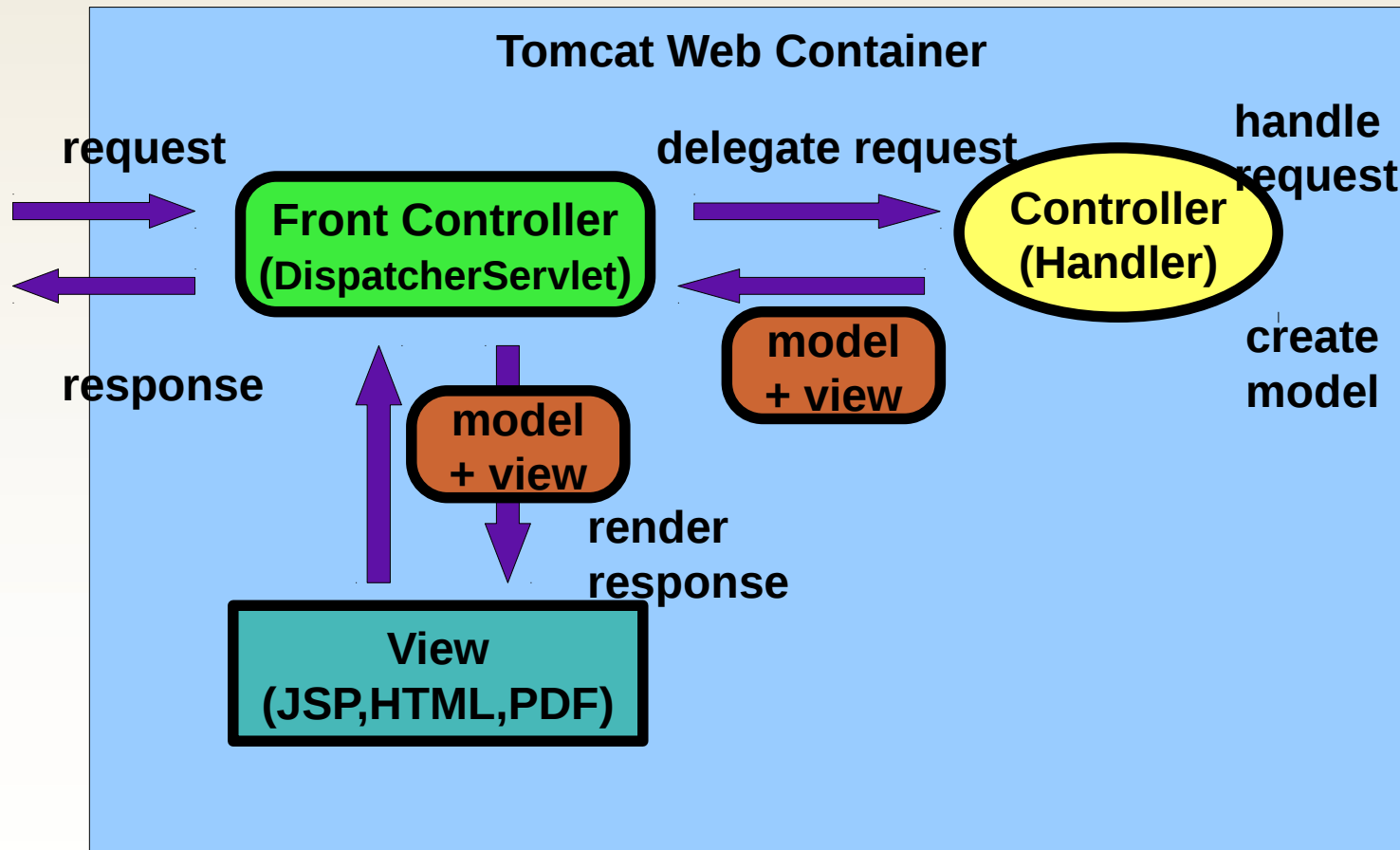


WebApplicationContext'e Nasıl Erişilir?

- ServletContext'e bind edilen WebApplicationContext'e erişmek için Spring **WebApplicationContextUtils** helper sınıfını sunar

```
public class PetClinicServlet extends HttpServlet {  
    @Override  
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)  
        throws ServletException, IOException {  
        WebApplicationContext wac = WebApplicationContextUtils  
            .getRequiredWebApplicationContext(getServletContext());  
        PetClinicService pcs = wac.getBean(PetClinicService.class);  
        ...  
    }  
}
```

Front Controller & DispatcherServlet



DispatcherServlet Konfigürasyonu

- DispatcherServlet'in aktivasyonu için **web.xml** içerisinde tanım yapılarak, burada **hangi requestleri ele alacağı** belirtilir

web.xml

```
<web-app>
  <servlet>
    <servlet-name>DispatcherServlet</servlet-name>
    <servlet-class>
org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>DispatcherServlet</servlet-name>
    <url-pattern>/mvc/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Spring MVC Özelliklerinin Aktivasyonu

- `<mvc:annotation-driven />` elemanı ise aşağıdaki kabiliyetleri devreye sokar:
 - Built-in **HttpMessageConverter** nesnelerini register eder
 - **Formatter** ve **Conversion** servislerini devreye alır
 - Controller handler metot parametrelerindeki **@NumberFormat**, **@DateTimeFormat** anotasyonları devreye sokar

@RequestMapping Annotasyonu

@Controller

```
public class HelloWorldController {
```

```
    @RequestMapping("/hello")
```

```
    public ModelAndView helloWorld() {
```

```
        ModelAndView mav = new ModelAndView();  
        mav.setViewName("/hello.jsp");  
        mav.addObject("message", "Hello World!");
```

```
        return mav;
```

```
    }
```

```
}
```

URL requestinin controller metotları ile eşleştirilmesini sağlar

Controller handler metodunun hem model, hem de view bilgisini tek bir return değeri olarak dönmesini sağlar

@RequestMapping Annotasyonu

Sınıf düzeyinde tanımlandığı takdirde metod düzeyindeki tanımlar relatif hale gelir

```
@Controller
```

```
@RequestMapping("/greet")
```

```
public class HelloWorldController {
```

```
    @RequestMapping("/hello")
```

```
    public ModelAndView hello() {
```

```
        // ...
```

```
    }
```

```
    @RequestMapping("/bye")
```

```
    public ModelAndView bye() {
```

```
        // ...
```

```
    }
```

```
}
```

hello() metodu eğer URI /greet/hello şeklinde olursa çağrılacaktır

bye() metodu eğer URI /greet/bye şeklinde olursa çağrılacaktır

Controller Metot

Return Tipi: String

@Controller

```
public class HelloWorldController {
```

```
    @RequestMapping("/hello")
```

```
    public String helloWorld(ModelMap model) {
```

```
        model.addAttribute("message", "Hello World!");
```

```
        return "/hello.jsp";
```

```
    }
```

```
}
```



Controller metodunun return tipi String ise, bu “**logical view**” ismine karşılık gelir

Controller Metot

Return Tipi: void

@Controller

```
public class HelloWorldController {
```

```
    @RequestMapping("/hello")
```

```
    public void helloWorld(HttpServletResponse response) {  
        response.getWriter().write("Hello World!");  
    }
```

```
}
```



Return tipi void ise, bu DispatcherServlet'e “**response'u controller metodu üretecek, sen herhangi bir şey yapma!**” demektir

@ResponseBody

Annotasyonunun İşlevi

```
@RequestMapping("/hello")  
@ResponseBody  
public String helloWorld() {  
    return "Hello World";  
}
```



Metot return değeri http response body'sini oluşturur

@RequestBody

Annotasyonunun İşlevi

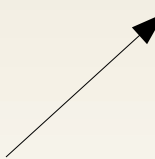
```
@RequestMapping(value="/printRequestBody",method=RequestMethod.POST)
public void handleRequest(@RequestBody String body, Writer writer)
throws IOException {
    writer.write(body);
}
```



Http request body'si @RequestBody ile işaretlenen metot parametresine atanır

@RequestParam

Request parametresinin değerini
metot parametresine atar

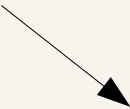


```
@RequestMapping("/pet")
public String displayPet(@RequestParam("petId") int petId,
    ModelMap model) {
    Pet pet = this.clinic.loadPet(petId);
    model.addAttribute("pet", pet);
    return "petForm";
}
```

<http://localhost:8080/petclinic/pet?petId=123>

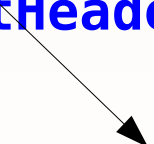
@CookieValue ve @RequestHeader

```
@RequestMapping("/displaySessionId")
public void displaySessionId(
    @CookieValue("JSESSIONID") String cookie) {
    //...
}
```



Http cookie değerini metod parametresine bind eder

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



Http request header değerini metod parametresine bind eder

Controller Bean'ları ve Exception'lar

@Controller

```
public class HelloController {
```

```
    @RequestMapping("/hello")
```

```
    public String helloWorld(ModelMap model) {
```

```
        model.addAttribute("message", "Hello World!");
```

```
        if(true) throw new RuntimeException("error!!!");
```

```
        return "/hello.jsp";
```

```
    }
```

@ExceptionHandler(RuntimeException.class)

```
public void handle(RuntimeException ex, Writer writer) {
```

```
    writer.write("Error handled :" + ex);
```

```
}
```

```
}
```

Birden fazla exception tipi alabilir
Değer olarak normal handler metotlar gibi view dönebilir,
yada response'u kendisi üretebilir

@ControllerAdvice ve @ExceptionHandler

```
@ControllerAdvice
public class GlobalErrorHandler {

    @ExceptionHandler(IOException.class)
    public String handleException(IOException ex,
HttpServletRequest request) {
        request.setAttribute("exception", ex);
        return "/error.jsp";
    }
}
```

URI Template Kabiliyeti

URL içerisindeki bölümlere controller metodu içerisinde erişmeyi sağlar, her bir bölüm bir değişkene karşılık gelir. Bu değişkenlerin requestdeki karşılıkları controller metod parametrelerine aktarılır.

```
@RequestMapping("/owners/{ownerId}")
public String findOwner(@PathVariable("ownerId") Long
id, Model model) {
    // ...
}
```

<http://localhost/owners/1> --> ownerId=1

@PathVariable ile değişken değeri metod parametresine aktarılır

Metod parametresi herhangi bir basit tip olabilir: **int, long, String**

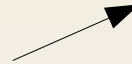
URI Template Kabiliyeti

URI template variable isimleri ile metot parametre isimlerinin eşleştirilmesi **derleme işleminin debug özelliği** açıksa mümkün olur. Aksi takdirde @PathVariable'a **variable ismi** verilmelidir

```
@RequestMapping("/{ownerId}")  
public String findOwner(@PathVariable Long ownerId, Model  
model) {  
    // ...  
}
```

URI Template Kabiliyeti ve Wildcard Kullanımı

Ant stili örüntüleri de destekler



```
@RequestMapping("/owners/*/pets/{petId}")  
public String findPet(@PathVariable Long petId, Model model) {  
  
    Pet pet = petService.getPet(petId);  
    model.addAttribute("pet", pet);  
  
    return "displayPet";  
}
```

```
@RequestMapping("/owners/**/pets/{petId}")
```



Herhangi derinlikteki path'leri kapsar

@RestController

- REST Controller bean'larını tanımlamak için kullanılır
- **@Controller** ve **@ResponseBody** anotasyonlarını bir araya getirir
- Her bir handler metoduna **@ResponseBody** eklemekten kurtarır
- Sadece kolaylık sağlar

```
@RestController
public class PetClinicRestController {
    ...
}
```

HTTP Protokolünün Metotları

- HTTP protokolünde sunucu tarafında yapılacak iş **HTTP metotları** ile belirtilir
 - **GET** : Mevcut bir resource'a erişim sağlar (**SELECT**)
 - **POST** : Yeni bir resource yaratır (**INSERT**)
 - **PUT** : Mevcut bir resource'u yaratmayı veya bütün olarak güncellemeyi sağlar (**INSERT/UPDATE**)
 - **PATCH** : Mevcut bir resource'u kısmen güncellemeyi sağlar (**UPDATE**)
 - **DELETE** : Mevcut bir resource'u siler (**DELETE**)

@RequestBody ile HTTP Mesajının Nesneye Dönüşümü

```
@RequestMapping(value="/owners", method=RequestMethod.POST)
public void createOwner(@RequestBody Owner owner) {
    // ...
}
```



Request ile gelen String verinin nesneye çevrimi için kullanılır. Nesne dönüşümü için kullanılacak `HttpMessageConverter` gelen **request'in content type'ına bakılarak** tespit edilir

@ResponseBody ile Return Değerinin Mesaja Dönüşümü

```
@RequestMapping( value = "/pets/{petId}",  
                  method=RequestMethod.GET)  
public @ResponseBody Pet findPet(@PathVariable Long  
petId) {  
    // ...  
}
```



Nesne'nin response ile gönderilecek veriye dönüştürülmesini sağlar

Dönüşüm için uygun `HttpMessageConverter` gelen **request URI**'na, request **accept header**'ına veya spesifik bir **request parametresine** bakılarak otomatik olarak tespit edilir

@ResponseStatus

```
@RequestMapping(value="/{ownerId}/addPet",method=RequestMethod.POST)
```

```
@ResponseStatus(HttpStatus.CREATED)
```

```
public void addPet(@RequestBody Pet pet, @PathVariable Long ownerId) {  
    // ...  
}
```

Metot başarılı sonlandığı vakit Spring MVC DispatcherServlet response statü kodu olarak 201 CREATED değerini set edecektir

@ResponseStatus

```
@ResponseStatus(HttpStatus.NOT_FOUND)
public class OwnerNotFoundException extends
RuntimeException {
    // ...
}
```

Bu exception herhangi bir controller metodundan fırlatıldığı zaman response statü kodu 404 NOT_FOUND olacaktır

RestTemplate

- Spring MVC, programatik REST istemci geliştirmek için **RestTemplate** desteği sunar
- RestTemplate sınıfı Spring'in klasik **template yaklaşımına** sahiptir
- GET,POST,PUT ve DELETE türündeki **HTTP** çağrıları için değişik metotlar barındırır

RestTemplate Kullanımı

HTTP Metodu	RestTemplate Metodu
GET	<code>getForObject(URI url, Class<T> responseType):T</code>
POST	<code>postForObject(URI url, Object request, Class<T> responseType):T</code>
PUT	<code>put(URI url, Object request):void</code>
DELETE	<code>delete(URI url):void</code>

İletişim

- **Harezmi** Bilişim Çözümleri
- Kurumsal Java Eğitimleri
- <http://www.java-egitimleri.com>
- info@java-egitimleri.com

