

Tasarım Örüntüleri ile Spring Eğitimi 7

Spring MVC & REST

REST Nedir?

- HTTP protokolü üzerinden **HTTP metotları ile gerçekleştirilen** web servis yaklaşımıdır
- **WSDL** tabanlı web servisleri HTTP'yi asıl tasarlandığı **uygulama düzeyi protokol** olarak değil, **transport protokol** olarak kullanmaktadır
- REST ile HTTP sadece transport protokol olarak değil, **uygulama düzeyi protokol** olarak da kullanılmaktadır

Uygulama Protokolü Olarak HTTP

- Transport protokol verinin istemci ve sunucu arasında **nasıl taşınacağını** belirler
- Veri bölümlenir, encode/decode edilir, istemci – sunucu arasında transfer edilir vs.
- Ancak gerçekleştirilecek olan **işlem veya servisle ilgili** herhangi bir **anlam içermez**
- Uygulama düzeyi protokolde ise bir takım komutlar veya ifadeler ile **sunucu tarafında yapılması istenen işlem** de belirtilmektedir

Resource Nedir?

- Resource **herhangi bir büyüklükteki veridir**
 - Text dosya, resim, ses, video gibi **statik** dosya olabilir
 - Yada **dinamik** olarak üretilen bir veri olabilir
 - **URL ile tespit edilebilmelidir**

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**)

POST ve PUT Metot Çağrılarları Arasındaki Fark

- Hem POST hem de PUT metotları sunucu tarafında **yeni bir resource yaratmak** (INSERT) için kullanılabilir
- Ancak **yaratılacak resource'un sunucudaki lokasyonu bilindiği zaman** PUT metodunu kullanmak anlamlıdır
- Bu da yeni bir nesne yaratılıyorsa bu **nesne'nin ID'sini istemcinin bilmesi** veya karar vermesine karşılık gelir

POST ve PUT Metot Çağrılarları Arasındaki Fark

- Eğer ID sunucu tarafından üretilen bir değer ise **POST**'u kullanmak doğru olacaktır
- Genel kural olarak yeni bir resource yaratma işlemleri için **POST** metodunu, mevcut resource'u güncellemek için ise **PUT** metodunu kullanmak doğrudur

REST Mimarisi

- **Stateless bir mimari** söz konusudur, çünkü HTTP'de stateless bir protokoldür
- State yönetimi gerekiyorsa **istemciler tarafından** yapılmalıdır
- Resource'ların **değişik formatlarda** gösterimi olabilir
 - XML, JSON, HTML vb.
- HTTP **header ve statü kodları** ile operasyonlarla ilgili bilgi istemci ve sunucu arasında paylaşılır

Spring ve REST

- RESTful uygulamalar geliştirmek için Java EE6 standardı **JAX-RS**'dir
- JAX-RS'in Jersey, CXF, Restlet gibi gerçekleştirimleri mevcuttur
- Spring MVC 3.1 ile birlikte **RESTful** Web servisleri ve uygulamalar geliştirmek mümkün hale gelmiştir
- Fakat Spring MVC bir **JAX-RS** gerçekleştirimi değildir

Spring ve REST

- REST servisleri **@Controller** bean'leri üzerinden hayata geçirilir
- Spring MVC ile REST servisleri oluşturmak için **aşağıdaki anotasyonlar** kullanılır
 - @RequestMapping
 - @PathVariable
 - @RequestBody
 - @ResponseBody
 - @ResponseStatus

@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 {
    ...
}
```

Spring ve REST API Örneği

```
@RequestMapping(value="/owners",  
                method=RequestMethod.GET)
```

```
@ResponseStatus(HttpStatus.OK)  
@ResponseBody Owners  
petClinicService.findOwners()
```

```
@RequestMapping(value="/owners/{id}",  
                method=RequestMethod.GET)
```

```
@ResponseStatus(HttpStatus.OK)  
@ResponseBody Owner petClinicService.  
loadOwner(@PathVariable long id)
```

```
@RequestMapping(value="/owner",  
                method=RequestMethod.POST)
```

```
@ResponseStatus(HttpStatus.CREATED)  
@ResponseBody Long petClinicService  
.createOwner(@RequestBody Owner owner)
```

```
@RequestMapping(value="/owner/{id}",  
                method=RequestMethod.PUT)
```

```
@ResponseStatus(HttpStatus.OK)  
void petClinicService.updateOwner(  
@RequestBody Owner owner, @PathVariable long id)
```

```
@RequestMapping(value="/owner/{id}",  
                method=RequestMethod.DELETE)
```

```
@ResponseStatus(HttpStatus.OK)  
void petClinicService.deleteOwner(  
@PathVariable long id)
```

HttpMessageConverter ile Text – Nesne Dönüşümü

- **HttpMessageConverter**'lar ile REST çağrılarında transfer edilen nesnelerin **text-nesne dönüşümleri** gerçekleştirilir
- **<mvc:annotation-driven/>** elemanı tarafından değişik converter implementasyonları **built-in register** edilmektedir
- Başka converter'lar da **ilave olarak** register edilebilir

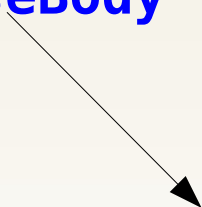
@RequestBody ile HTTP Mesajının Nesneye Dönüşümü

```
@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

Response Content Tipinin Belirlenmesi

- Web isteği sonucunda dönülecek içeriğin formatı default durumda önce **request URI'in uzantısına**, yoksa **format request parametresine**, diğer durumda ise **Accept request header'ına** bakılarak belirlenir
- Bu ayarlar `<mvc:annotation-driven/>` elemanı built-in **ContentNegotiationManager** bean'ı register etmektedir

Response Content Tipinin Belirlenmesi: URI Path Ext.



```
GET /petclinic/pets/123.json  
...
```

```
HTTP/1.1 200 OK  
...  
Content-Length:...  
Content-Type: application/json  
  
{  
    pet: {id:"123", ...}  
}
```

Response Content Tipinin Belirlenmesi: Request Parameter



```
GET /petclinic/pets/123?format=json  
...
```

```
HTTP/1.1 200 OK  
...  
Content-Length:...  
Content-Type: application/json  
  
{  
    pet: {id:"123", ...}  
}
```

Response Content Tipinin Belirlenmesi: Accept Header

```
GET /petclinic/pets/123
...
Accept: application/json
...
```

```
HTTP/1.1 200 OK
...
Content-Length:...
Content-Type: application/json

{
    pet: {id:"123", ...}
}
```

ContentNegotiationManager Konfigürasyonu

```
<mvc:annotation-driven content-negotiation-manager="contentNegotiationManager"/>
```

```
<bean id="contentNegotiationManager"  
class="org.springframework.web.accept.ContentNegotiationManagerFactoryBean">
```

```
  <property name="favorPathExtension" value="true"/>
```

```
  <property name="favorParameter" value="false"/>
```

```
  <property name="ignoreAcceptHeader" value="false"/>
```

```
  <property name="mediaTypes">
```

```
    <map>
```

```
      <entry key="json" value="application/json"/>
```

```
      <entry key="xml" value="application/xml"/>
```

```
      <entry key="html" value="text/html"/>
```

```
    </map>
```

```
  </property>
```

```
  <property name="useJaf" value="false"/>
```

```
  <property name="defaultContentType" value="text/html"/>
```

```
</bean>
```

1. request URI uzantısı'na bakılır
2. Accept header değerine bakılır
3. "format" request param değerine bakılır

MediaTypes mevcut değil ise Java Activation Framework Kütüphanesi classpath'de mevcut ise mime type belirlemek için kullanılabilir

Response'un Farklı Tiplerde Üretilmesine Örnek

```
@RequestMapping(value="/owners/{id}", produces={"application/json", "application/xml"})
@ResponseBody
public Owner getOwner(@PathVariable Long id) {
    return petClinicService.getOwner(id);
}
```

http://localhost:8080/petclinic/mvc/owners/1.json -> application/json
http://localhost:8080/petclinic/mvc/owners/1.xml -> application/xml

http://localhost:8080/petclinic/mvc/owners/1.html -> text/html

```
@RequestMapping("/owners/{id}")
@ResponseBody
public String getOwnerAsHtml(@PathVariable Long id) {
    StringBuilder builder = new StringBuilder();
    Owner owner = petClinicService.getOwner(id);
    builder.append("<html><body>");
    builder.append(owner.getFirstName() + " " + owner.getLastName() + "<br>");
    builder.append("</body></html>");
    return builder.toString();
}
```

REST ve Statü Kodları

- RESTful servisler, istemcilerle haberleşmek için **http statü kodlarından** da yararlanırlar
- **@ResponseStatus** annotasyonu ile **HttpServletResponse**'a müdahale etmeden response'un statü kodu set edilebilir

HTTP Statü Kodları

Statü Kod Grubu	Mesaj Kategorisi
1xx	Info
2xx	Success
3xx	Redirect
4xx	Client Error
5xx	Server Error

@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>

RestTemplate Kullanım

Örneği: GET

```
RestTemplate restTemplate = new RestTemplate();
```

```
Vet vet = restTemplate.getForObject(  
    "http://localhost:8080/petclinic/mvc/vets/1",  
    Vet.class);
```

RestTemplate Kullanım

Örneği: GET

```
RestTemplate restTemplate = new RestTemplate();
```

```
ResponseEntity<String> entity =  
    restTemplate.getForEntity(  
        "http://localhost:8080/petclinic/mvc/vets/1/lastName",  
        String.class);
```

```
String body = entity.getBody();
```

```
MediaType contentType =  
    entity.getHeaders().getContentType();
```

```
HttpStatus statusCode = entity.getStatusCode();
```

RestTemplate Kullanım

Örneği: POST

```
RestTemplate restTemplate = new RestTemplate();
```

```
Vet vet = new Vet("Kenan", "Sevindik", 1999);
```

```
restTemplate.postForObject("http://localhost:8080  
/petclinic/mvc/vet", vet, Void.class);
```

RestTemplate Kullanım

Örneği: POST

```
RestTemplate restTemplate = new RestTemplate();

Owner owner = new Owner();
owner.setFirstName("Ali");
owner.setLastName("Yücel");

URI location =
restTemplate.postForLocation("http://localhost:8080/petclinic/mvc/owners/create", owner);
```


RestTemplate Kullanım

Örneği: POST

```
RestTemplate restTemplate = new RestTemplate();
```

```
Vet vet = new Vet("Kenan", "Sevindik", 1999);
```

```
ResponseEntity<Long> entity =  
    restTemplate.postForEntity(  
        "http://localhost:8080/petclinic/mvc/vet", vet,  
        Long.class);
```

```
Long id = entity.getBody();
```

```
MediaType contentType =  
    entity.getHeaders().getContentType();
```

```
HttpStatus statusCode = entity.getStatusCode();
```

RestTemplate Kullanım

Örneği: Exchange

```
RestTemplate restTemplate = new RestTemplate();

Vet vet = new Vet("Kenan", "Sevindik", 1999);

RequestEntity<Vet> requestEntity =
    RequestEntity.post(
        new URI("http://localhost:8080/petclinic/mvc/vet"))
        .header("Authorization", "Basic aHR0cHdhbGNoOmY=")
        .accept(MediaType.APPLICATION_JSON)
        .body(vet);

ResponseEntity<Long> responseEntity =
    restTemplate.exchange(requestEntity, Long.class);

Long id = responseEntity.getBody();
```

RestTemplate Kullanım

Örneği: Execute

```
RequestCallback requestCallback = new RequestCallback() {

    @Override
    public void doWithRequest(ClientHttpRequest request)
        throws IOException {
        request.getHeaders()
            .set("Authorization", "Basic aHR0cHdhdGNoOmY=");
    }
};

ResponseExtractor<Vet> responseExtractor = new ResponseExtractor<Vet>() {

    @Override
    public Vet extractData(ClientHttpResponse response)
        throws IOException {
        HttpStatus statusCode = response.getStatusCode();
        InputStream inputStream = response.getBody();
        //...
        return vet;
    }
};

Vet vet = restTemplate.execute("http://localhost:8080/mvc/vets/1",
    HttpMethod.GET, requestCallback, responseExtractor);
```

RestTemplate ve ClientHttpRequestInterceptor

- **ClientHttpRequestInterceptor** nesneleri vasıtası ile istemci tarafında HTTP request öncesi ve HTTP response döndükten sonra ilave işlemler yapılabilir
- Örneğin **BasicAuthorizationInterceptor** yardımı ile HTTP isteğine basic authentication header değerleri yerleştirilebilir

BasicAuthorization Interceptor Örneği

```
RestTemplate restTemplate = new RestTemplate();

BasicAuthorizationInterceptor basicAuthorizationInterceptor =
    new BasicAuthorizationInterceptor("user1", "secret");
restTemplate.setInterceptors(
    Arrays.asList(basicAuthorizationInterceptor));

Owner owner = restTemplate
    .getForObject("http://localhost/owners/1", Owner.class);
```

Web Uygulamaları ve Entegrasyon Birim Testleri

Web Uygulamalarının Test Edilmesi

- Spring standalone testler içerisinde **WebApplicationContext**'in oluşturulmasını sağlar
- Böylece **Controller katmanı** da entegrasyon testlerine dahil edilebilir

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration("webapp")
@ContextConfiguration("/appcontext/beans-*.xml")
public class SpringWebAppTests {
    ...
}
```

ContextRoot dizinini belirler
Default **src/main/webapp**'dir

Mock Servlet API Kullanımı

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration("webapp")
@ContextConfiguration("/appcontext/beans-*.xml")
public class SpringWebAppTests {

    @Autowired
    private WebApplicationContext wac; // cached

    @Autowired
    private MockServletContext servletContext; // cached

    @Autowired
    private MockHttpSession session;

    @Autowired
    private MockHttpServletRequest request;

    @Autowired
    private MockHttpServletResponse response;

}
```

Testler için oluşturulan
WebApplicationContext
nesnesi

Test metotları içerisinde
erişilebilecek built-in
mock nesnelerdir

Spring'in Servlet API
Mock sınıflarıdır

MockMvc Kullanarak Controller Testi

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration("webapp")
@ContextConfiguration("/appcontext/beans-*.xml")
public class WebApplicationContextMvcTests {

    @Autowired
    private WebApplicationContext wac; // cached

    private MockMvc mockMvc;

    @Before
    public void setup() {
        this.mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).build();
    }

    ...
}
```

MockMvc Kullanarak Controller Testi

```
@Test
public void testOwners() throws Exception {
    MockHttpServletRequestBuilder requestBuilder =
        MockMvcRequestBuilders.get("/owners");

    ResultActions resultActions = mockMvc.perform(requestBuilder)

    MvcResult mvcResult = resultActions.andReturn();

    ModelAndView mav = mvcResult.getModelAndView();

    MatcherAssert.assertThat(mav.getViewName(),
                             Matchers.equalTo("owners"));
    MatcherAssert.assertThat(mav.getModel().containsKey("owners"),
                             Matchers.equalTo(true));
}
```

MockMvc ve ResultMatcher Kullanımı

ResultMatcher :Dönen response üzerinde bir “expectation”ın kontrol edilmesini sağlar
MockMvcResultMatchers ile kullanılır

```
@Test
public void testOwners() throws Exception {

    mockMvc.perform(MockMvcRequestBuilders.get("/owners")
                                                    .param("lastName", "Yücel"))

        .andExpect(MockMvcResultMatchers.view().name("ownersList"))

        .andExpect(MockMvcResultMatchers.model()
                    .attributeExists("owners"));

}
```

MockMvc ve ResultHandler Kullanımı

ResultHandler :Dönen response üzerinde bir action execute etmeyi sağlar
MockMvcResultHandlers ile kullanılır

```
@Test
public void testOwners() throws Exception {

    mockMvc.perform(MockMvcRequestBuilders.get("/owners"))

        .andDo(MockMvcResultHandlers.print());

}
```

Web Uygulamalarında Java Tabanlı Konfigürasyon

Web Uygulamalarında Java Tabanlı Container Konfigürasyonu

- **ContextLoaderListener** default olarak **WebApplicationContext** sınıfından bir Container oluşturur, bu da XML konfigürasyon dosyaları ile çalışır
- Java tabanlı konfigürasyon sınıflarının yüklenmesi isteniyorsa **contextClass** isimli bir context-param ile **AnnotationConfigWebApplicationContext**'i kullanması söylenmelidir
- **contextConfigLocation** context-param'a ise bundan böyle değer olarak FQN şeklinde konfigürasyon sınıfları verilebilir

Web Uygulamalarında Java Tabanlı Container Konfigürasyonu

```
<web-app>
  <context-param>
    <param-name>contextClass</param-name>
    <param-value>
      org.springframework.web.context.support.AnnotationConfigWebApplicationContext
    </param-value>
  </context-param>

  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      com.javaegitimleri.petclinic.config.AppContextConfig
    </param-value>
  </context-param>

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


Web.xml'siz ApplicationContext Konfigürasyonu

- **WebApplicationInitializer** arayüzünü implement ederek web uygulamasında **ApplicationContext bootstrap işlemi tamamen web.xml'siz** gerçekleştirilebilir
- **Servlet 3.0+** spesifikasyonu ile uyumlu uygulama sunucular için geçerlidir
- Uygulamada aynı anda **birden fazla WebApplicationInitializer** mevcut olabilir

Web.xml'siz ApplicationContext Konfigürasyonu

```
public class PetClinicWebApplicationInitializer
    extends AbstractContextLoaderInitializer {

    @Override
    protected WebApplicationContext createRootApplicationContext() {
        AnnotationConfigWebApplicationContext wac =
            new AnnotationConfigWebApplicationContext();
        wac.register(DaoBeansConfig.class, ServiceBeansConfig.class);
        wac.refresh();
        return wac;
    }
}
```



WebApplicationInitializer arayüzünden türer. ServletContext'de ContextLoaderListener'in konfigürasyonunu sağlar

Birden fazla WebApplicationInitializer olması durumunda **Ordered** arayüzü ile aralarında sıralama yapmak mümkündür

Web.xml'siz DispatcherServlet Konfigürasyonu

```
public class PetClinicSpringMvcWebApplicationInitializer
    extends AbstractDispatcherServletInitializer {

    @Override
    protected WebApplicationContext createServletApplicationContext() {
        AnnotationConfigWebApplicationContext wac =
            new AnnotationConfigWebApplicationContext();
        wac.setEnvironment(new StandardServletEnvironment());
        wac.register(ControllerBeansConfig.class);
        return wac;
    }

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

    @Override
    protected WebApplicationContext createRootApplicationContext() {
        AnnotationConfigWebApplicationContext wac =
            new AnnotationConfigWebApplicationContext();
        wac.register(DaoBeansConfig.class, ServiceBeansConfig.class);
        return wac;
    }
}
```

Web.xml'siz DispatcherServlet Konfigürasyonu

- Classpath'deki WebApplicationInitializer sınıflarını tespit edip yükleyen **SpringServletContainerInitializer** sınıfıdır
- **ServletContainerInitializer** SPI arayüzünden türer
- **spring-web.jar:/META-INF/services/javax.servlet.ServletContainerInitializer** dosyası içerisinde tanımlıdır

Web.xml'siz DispatcherServlet Konfigürasyonu

- DispatcherServlet ayrı bir WAC oluşturmak yerine parent WAC'ı da kullanabilir

```
public class PetClinicSpringMvcWebApplicationInitializer
    extends AbstractDispatcherServletInitializer {

    private AnnotationConfigWebApplicationContext wac;

    @Override
    protected WebApplicationContext createRootApplicationContext() {
        wac = new AnnotationConfigWebApplicationContext();
        wac.register(DaoBeansConfig.class, ServiceBeansConfig.class,
            ControllerBeansConfig.class);
        return wac;
    }

    @Override
    protected WebApplicationContext createServletApplicationContext() {
        return wac;
    }

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

İletişim

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



harezmi
bilişim çözümleri

JAVA
Eğitimleri 