

# Spring Container Kabiliyetleri

## 3



# Spring EL Nedir?

- SpEL, Spring'in **expression dilidir**
- **Unified EL**'e benzer fakat ondan daha gelişmiştir
- **Object graph query** işlemini ve **manipülasyonunu** sağlar
- Method invocation, değer set etme, nesne yaratma vs yapılabilir

# Spring EL ile Çalışma Adımları

**SpelExpressionParser** implementasyonu default impl.dır

① *Parser oluştur*

```
ExpressionParser parser = new SpelExpressionParser();
```

② *Expression'ı parse et*

Oluşturulan parser ile String ifade parse edilir ve Expression elde edilir

```
Expression exp = parser.parseExpression("'Hello World'");
```

③ *Expression'ı evaluate et*

```
String message = (String) exp.getValue();
```

getValue metodu expression'ı evaluate ederek değerini döndürür

# Spring EL Expression Örnekleri

```
ExpressionParser parser = new SpelExpressionParser();  
Expression exp = parser.parseExpression("'Hello World'.bytes");  
byte[] bytes = (byte[]) exp.getValue();
```

//getBytes().length metodu cagrilir...

```
Expression exp = parser.parseExpression("'Hello  
World'.bytes.length");  
int length = (Integer) exp.getValue();
```

```
Expression exp = parser.parseExpression("new String('hello  
world').toUpperCase()");  
String message = exp.getValue(String.class);
```

FQN yazılarak herhangi  
Bir sınıftan nesne yaratmak  
Da mümkündür

# Spring EL ve Evaluation Context

- Tanımlanan expression'ları nesneler üzerinde evaluate etmek ve bu nesnelerin property'lerine değer set etmek için kullanılır

```
ExpressionParser parser = new SpelExpressionParser();  
Expression exp = parser.parseExpression("firstName");
```

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

Expression evaluation işleminin üzerinde gerçekleştirildiği nesneye root object denir

```
StandardEvaluationContext context = new StandardEvaluationContext();  
context.setRootObject(owner);
```

```
String firstName = (String) exp.getValue(context);  
exp = parser.parseExpression("address?.city");  
String firstName = (String) exp.getValue(context);  
exp = parser.parseExpression("lastName");  
String lastName = (String) exp.setValue(context, "Yılmaz");
```

# Spring EL Değişkenleri ile Değer Set Etme

```
ExpressionParser parser = new SpelExpressionParser();
```

```
Expression exp = parser.parseExpression("firstName = #newName");
```

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

```
StandardEvaluationContext context = new StandardEvaluationContext();  
context.setRootObject(owner);
```

```
context.setVariable("newName", "Veli");
```

Context içerisinde newName  
İsimli bir değişken ve değeri  
Tanımlanır.

Daha sonra bu değişkenin değeri firstName  
Property'sine assign edilir

```
String firstName = (String) exp.getValue(context);
```

# Spring EL'de Fonksiyon Tanımlama

```
ExpressionParser parser = new SpelExpressionParser();  
StandardEvaluationContext context = new StandardEvaluationContext();  
  
Method method = StringUtils.class.getDeclaredMethod("reverseString",  
new Class[] { String.class })  
context.registerFunction("reverseString", method);
```

Reflection ile elde edilen metot nesnesi register edilip expression içerisinde kullanılabilir

```
String helloWorldReversed =  
parser.parseExpression("#reverseString('hello'  
world)").getValue(context, String.class);
```

# SpEL ve Bean Tanımları

```
<bean id="numberGuess" class="x.y.NumberGuess">
  <property name="randomNumber" value="#{ T(java.lang.Math).random()* 100.0 }" />
</bean>
```

Bean tanımlarındaki property değerlerinde SpEL ifadeleri kullanılabilir  
T() ifadesi ile burada bir java tipi FQN ile ifade edilmiştir

```
<bean id="taxCalculator" class="x.y.TaxCalculator">
  <property name="defaultLocale" value="#{ systemProperties['user.region'] }"/>
</bean>
```

JVM sistem değişkenlerine erişmek de mümkündür

```
<bean id="shapeGuess" class="org.springframework.samples.ShapeGuess">
  <property name="initialShapeSeed" value="#{ numberGuess.randomNumber }"/>
</bean>
```

Expression içerisinde başka bir bean'ın property değerine  
Erişmek de mümkündür



# SpEL ve Annotasyon Tabanlı Konfigürasyon

**@Component**

```
public class FooBean {
```

```
    @Value("#{ systemProperties['user.region'] }")  
    private String defaultLocale;
```

```
    public void setDefaultLocale(String defaultLocale) {  
        this.defaultLocale = defaultLocale;  
    }
```

```
    ...  
}
```

Metot ve field düzeyinde kullanılabilir

Metot input parametre tanımlarında da kullanılabilir

# Spring ve Metot Düzeyinde Validasyon

- Controller/Servis metotlarının **input parametreleri** ve **return değeri** Spring vasıtası ile deklaratif biçimde validate edilebilir

**@Validated**

@Service

```
public class UserService {  
    public @NotNull Long createUser(@Valid User user) {  
        ...  
    }  
}
```

Validate edilecek metotları içeren servis bean'ı @Validated anotasyonu ile işaretlenmelidir

Bunun yanında JSR-303 validasyon constraint'leri doğrudan input parametre ve return değerinde de kullanılabilir

Validasyona tabi tutulacak input parametre veya return tipi içerisinde JSR 303 constraint'leri barındıran bir domain sınıf ise @Valid anotasyonu ile işaretlenir

# Spring ve Metot Düzeyinde Validasyon

```
public class User {  
    @NotEmpty  
    private String username;  
    @Email  
    private String email;  
    @Min(18) @Max(64)  
    private int age;  
}
```

Buradaki @NotEmpty, @Email, @Min, @Max anotasyonları JSR-303 constraint'leridir Spring'den bağımsızdır

# JSR Bean Validation API ve Spring

- Spring validasyon işlemini JSR-303/JSR-349 **Bean Validation API**' üzerine bina etmiştir
- Servis metotlarına ve domain model sınıflarına **validasyon anotasyonları** yerleştirilir
- Runtime'da da otomatik tespit edilip yüklenen **JSR Validator bean**'i ile bu constraint'ler denetlenir

# Servis Metot Düzeyinde Validasyon Konfigürasyonu

@Validated anotasyonuna sahip bean'ların metot parametrelerini ve return değerlerini validate eden proxy bean'lar üretir

```
<bean  
class="org.springframework.validation.beanvalidation.  
MethodValidationPostProcessor">  
    <property name="validator" ref="validator"/>  
</bean>
```

```
<bean id="validator"  
class="org.springframework.validation.beanvalidation.  
LocalValidatorFactoryBean"/>
```

classpath'deki JSR-303 bean validation provider'ı devreye sokar

# JSR-303 Validator API'nin Doğrudan Kullanılması

```
public class FooService {  
    private javax.validation.Validator validator;  
  
    public void setValidator(Validator validator) {  
        this.validator=validator  
    }  
  
    public void doWork() {  
        validator.validate(...);  
    }  
}
```

JSR-303 implementasyonu Spring managed bean olarak uygulama içindeki diğer bean'lere **Validator nesnesi** olarak enjekte edilerek doğrudan da kullanılabilir

```
<bean id="fooService" class="x.y.FooService">  
    <property name="validator" ref="validator"/>  
</bean>
```

```
<bean id="validator"  
class="org.springframework.validation.beanvalidation  
.LocalValidatorFactoryBean" />
```

# Spring MVC Controller'ları ve Validasyon


- **@Controller** metotlarındaki **input parametreler** ve **return değerleri** de otomatik olarak validate edilebilir
- Controller bean'larında validasyonun aktive olması için **<mvc:annotation-driven/>** elemanına ihtiyaç vardır
- Metot invokasyonu sırasında meydana gelen **validasyon hataları** da yine Controller içerisinde **@ExceptionHandler** ile ele alınabilir

# Spring MVC Controller'ları ve Validasyon

Validasyon hataları **BindingResult** nesnesinde toplanır. BindingResult kontrol edilerek iş akışı yönlendirilebilir.

```
@Controller
public class PetClinicRestController {

    @RequestMapping(value="/owner",method=RequestMethod.POST)
    @ResponseStatus(code=HttpStatus.CREATED)
    public void create(@RequestBody @Valid Owner owner,
                     BindingResult bindingResult,
                     HttpServletResponse response)
        throws Exception {
        if(bindingResult.hasErrors()) {
            //...
            response.setHeader("Location", "/ownerCreateFailed");
        } else {
            //...
            response.setHeader("Location", "/ownerCreateSuccess");
        }
    }
}
```





# Spring MVC Controller'ları ve Validasyon

Eğer metod parametresinin validasyonunda herhangi bir sorun olursa  
MethodArgumentNotValidException fırlatılır

```
@RestController
public class PetClinicRestController {

    @Autowired
    private PetClinicService petClinicService;

    @RequestMapping(value="/owner",method=RequestMethod.POST)
    @ResponseStatus(code=HttpStatus.CREATED)
    public Long createOwner(@RequestBody @Valid Owner owner) {
        petClinicService.create(owner);
        return owner.getId();
    }

    @ExceptionHandler
    public void handle(MethodArgumentNotValidException exception,
        HttpServletResponse response) {
        response.setStatus(HttpStatus.PRECONDITION_REQUIRED.value());
    }
}
```

# Spring MVC Controller'ları ve Validasyon

@RestController

**@Validated**

public class PetClinicRestController {

@Autowired

private PetClinicService petClinicService;

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

public Owner findOwnerByEmail(@PathVariable **@Email** String email) {  
 return petClinicService.findOwner(email);  
}

@ExceptionHandler

public void handle **ConstraintViolationException** exception,  
HttpServletResponse response) {  
 response.setStatus(HttpStatus.PRECONDITION\_REQUIRED.value());  
}

JSR-303 validasyon kısıtları da denetlenebilir. Bunun için sınıf düzeyinde @Validated anotasyonu kullanılmalıdır. Hata durumunda ConstraintViolationException fırlatılır

# JSR-303 Custom Validation Constraints ve Spring Ent.

- Spring JSR-303 API ile **custom validation constraint** yazımı için de kolaylık sağlar
- Örneğin **@CheckIfApproved** gibi bir constraint yazılabilir
- Validation constraint anotasyonunu process eden **ConstraintValidator** sınıfına Spring dependency injection yapabilir
- Böylece ConstraintValidator sınıfları **Spring ekosistemindeki imkanlardan** faydalanabilir

# JSR-303 Custom Validation Constraints ve Spring Ent.

```
public class VisaRequest {
```

```
@CheckIfApproved
```

```
private Document doc;
```

```
//...
```

```
}
```

Custom validation constraint'dir

Custom validation constraint'i process eden **ConstraintValidator** sınıfıdır

```
@Target({ElementType.METHOD, ElementType.FIELD})
```

```
@Retention(RetentionPolicy.RUNTIME)
```

```
@Constraint(validatedBy=ApprovedConstraintValidator.class)
```

```
public @interface CheckIfApproved {
```

```
}
```

# JSR-303 Custom Validation Constraints ve Spring Ent.

```
import javax.validation.ConstraintValidator;

public class ApprovedConstraintValidator
    implements ConstraintValidator {

    @Autowired
    private ApprovalCheckingService approvalCheckingService;

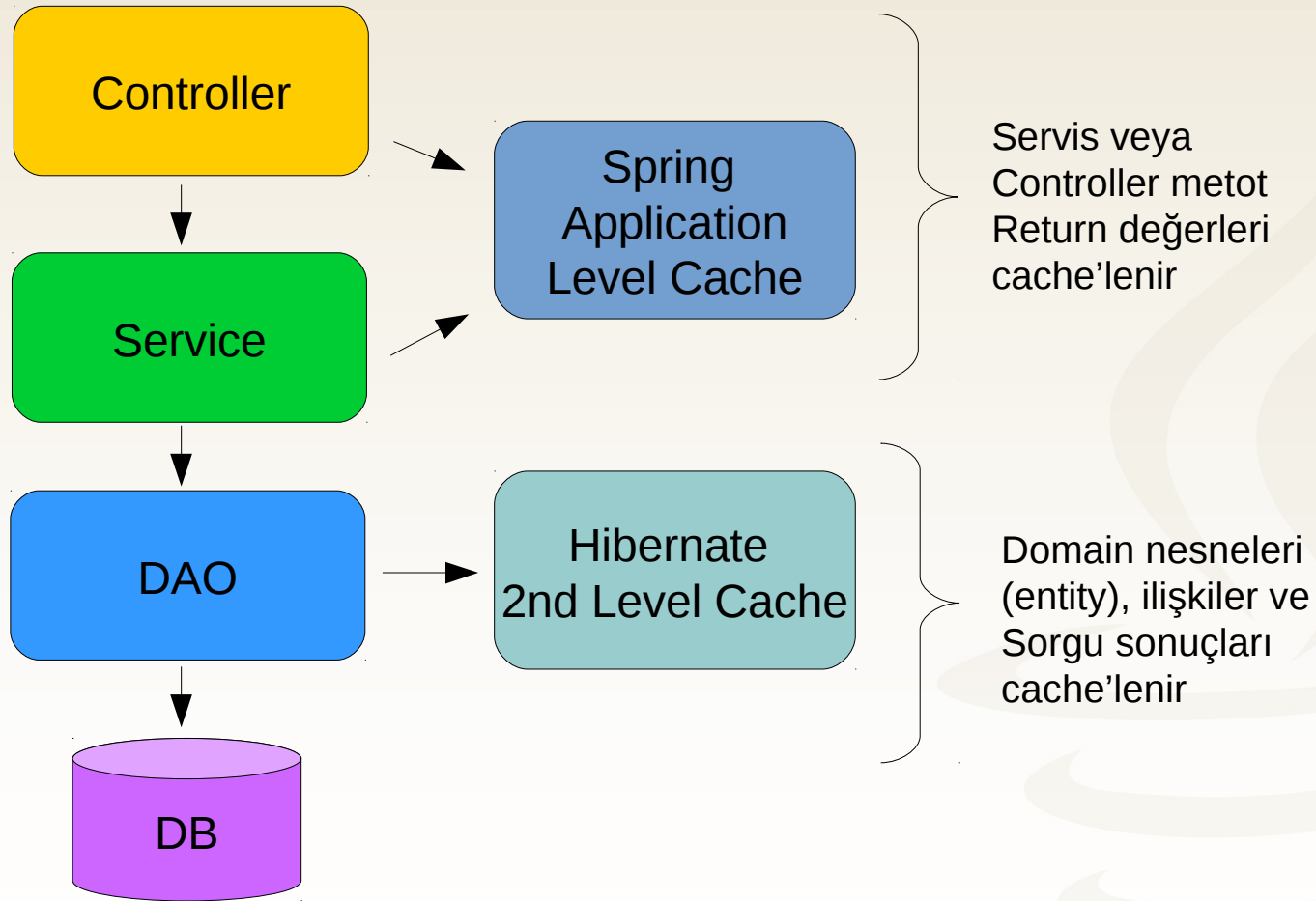
    //...
}
```

JSR Validator bean'ini tanımlamak için **LocalValidatorFactoryBean** kullanıldığı takdirde **ConstraintValidator** nesnelerine dependency injection yapmak mümkündür

# Spring ve Metot Düzeyinde Caching

- Spring cache **web veya remote metot çağrılar**ından dönen sonuçların cache'lenmesi için düşünülmüştür
- **Controller** veya **servis** katmanında yer alan ve **cache ihtiyacı** olan bean'lerde kullanılır
- ORM 2<sup>nd</sup> level cache **persistent domain nesnelerinin** cache'lenmesi içindir
- Spring cache ise **metot return değerlerini** cache'ler

# Cache Katmanları



# Spring ve Metot Düzeyinde Caching

- Sınıf veya metot düzeyinde **@Cacheable** anotasyonu ile tanımlanır
- **@Cacheable**
  - Sınıf düzeyinde bütün metotlarda caching'i devreye alır
  - Metot düzeyinde ise metot spesifik tanımlar yapılabilir
- Ayrıca **@CacheEvict** ve **@CachePut** anotasyonları ile cache içerisindeki veri yönetilebilir



# @Cacheable Kullanımı

```
public class FooService {
```

```
@Cacheable("fooWithNameRegion")
```

```
public Foo findFoo(String name) {  
    //...  
}
```

Cache bölümünün ismidir, Cache'lenen bilgi fiziksel cache içerisinde bu bölümde tutulacaktır

```
@Cacheable(  
    value="defaultRegion",key="#date.time")  
public Foo findFoo(Date date) {  
    //...  
}
```

Cache **key** default durumda **metot parametrelerinden** elde edilir, **SpEL expression** yardımı ile de elde edilebilir

```
@Cacheable(  
    value="defaultRegion",condition="#i>10")  
public Foo findFoo(int i) {  
    //...  
}
```

Cache işleminin ne zaman devreye gireceğini belirlemek için SpEL yardımı ile "**condition**" da tanımlanabilir

```
}
```

# @CacheEvict ve @CachePut Kullanımı

```
public class FooService {
```

```
    @CacheEvict("fooWithNameRegion")  
    public void updateFoo(String name) {  
        //...  
    }
```

Key değerine karşılık gelen entry  
cache'den çıkarılır

```
    @CachePut("fooWithNameRegion")  
    public Foo insertFoo(String name) {  
        //...  
    }
```

Metot return değeri cache'e entry olarak eklenir  
Default durumda metot input parametreleri ile key  
değeri belirlenir

```
}
```

# Spring ve Metot Düzeyinde Caching Konfigürasyonu

- `<cache:annotation-driven/>` elemanı ile devreye girer
- Container'da tanımlı **CacheManager** arayüzünü implement eden bir bean'a ihtiyaç duyar
- Built-in **EhCache** ve **ConcurrentHashMap** tabanlı implemantasyonları mevcuttur
- Diğer cache provider'lar da **entegre** edilebilir

# Spring ve Metot Düzeyinde Caching Konfigürasyonu

Test ortamları için uygun  
basit bir CacheManager  
implementasyonudur

```
<cache:annotation-driven/>
```

```
<bean id="cacheManager"  
class="org.springframework.cache.support.SimpleCacheManager">  
  <property name="caches">  
    <set>  
      <bean  
class="org.springframework.cache.concurrent.ConcurrentMapCacheFact  
oryBean">  
        <property name="name" value="defaultRegion" />  
      </bean>  
      <bean  
class="org.springframework.cache.concurrent.ConcurrentMapCacheFact  
oryBean">  
        <property name="name" value="fooWithNameRegion" />  
      </bean>  
    </set>  
  </property>  
</bean>
```

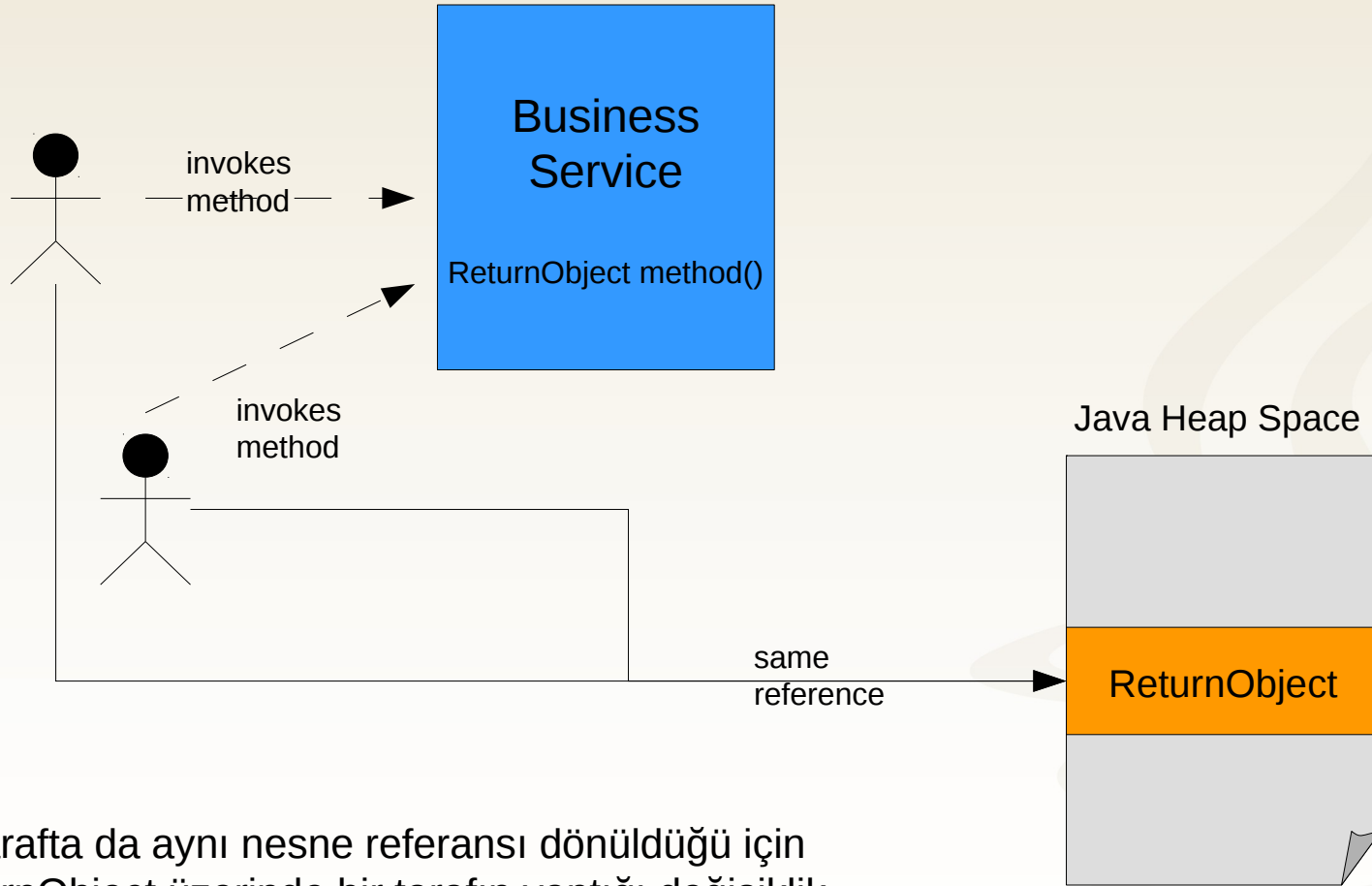
Her bir **cache region** için bir bean tanımlanır.  
Nesneler bu cache bölümleri içerisinde **key'e göre**  
**gruplanarak** saklanır

# Metot Düzeyinde Caching

## Kullanırken Dikkat Edilecekler

- Metot **return değerleri** cache'lendiği takdirde bu değerlerin **immutable** (read-only) olması önemlidir
- Aksi takdirde cache'den dönen aynı nesne referansı üzerinden hareket edildiği için **istenmeyen yan etkiler** ortaya çıkabilir
- Bu nedenle caching kullanımı **controller katmanı** için daha uygundur

# Metot Düzeyinde Caching Kullanırken Dikkat Edilecekler



İki tarafta da aynı nesne referansı dönlüdüğü için  
ReturnObject üzerinde bir tarafın yaptığı değişiklik  
Diğer tarafı da etkileyecektir

# Spring ve E-Posta Desteği

- Spring, e-posta gönderme işlemini sistemin **Java Mail altyapısından soyutlanmasını** sağlayan yardımcı sınıflar sunar
- Alt düzey sistem hatalarını daha anlamlı **üst düzey e-posta hatalarına** dönüştürür
- Alt düzeyde **resource'ların ele alınmasını** da kolaylaştırır

# Spring ve E-Posta Desteği

- Spring e-posta desteğinin temel yapı taşı **MailSender** arayüzüdür
- **JavaMailSender** arayüzü de MIME mesaj gönderimi gibi ilave kabiliyetler sunar
- Her ikisinde de alt tarafta **Java Mail API**'si kullanılmaktadır



# Mail Mesajları

- **MailMessage** bütün e-posta mesajları için ortak arayüzdür
- Farklı gerçekleştirmeleri mevcuttur
- **SimpleMailMessage** sınıfı ile from, to, cc, subject, text body gibi bölümleri içeren basit e-posta mesajları oluşturulabilir
- **MimeMailMessage** sınıfı ile de MIME mesajları oluşturulabilir
- **MimeMessagePreparator** sınıfı ile mime mesajlarının hazırlanması gerçekleştirilir

# MailSender Konfigürasyonu

```
<bean id="mailSender"
class="org.springframework.mail.javamail.JavaMailSenderImpl">
    <property name="host" value="smtp.mycompany.com"/>
    <property name="port" value="25"/>
    <property name="username" value="user1"/>
    <property name="password" value="secret"/>
    <property name="javaMailProperties">
        <props>
            <prop key="mail.smtp.auth">true</prop>
        </props>
    </property>
</bean>
```

# Text İçerikli Mesaj Gönderme

```
SimpleMailMessage msg = new SimpleMailMessage();
msg.setFrom("me@example.com");
msg.setTo("you@example.com");
msg.setText("Hello world!");

mailSender.send(msg);
```

# Mime Mesaj Gönderme: 1.yol

```
MimeMessagePreparator preparator = new MimeMessagePreparator() {  
  
    public void prepare(MimeMessage mimeMessage) throws Exception {  
        mimeMessage.setRecipient(Message.RecipientType.TO,  
            new InternetAddress("you@example.com"));  
        mimeMessage.setFrom(new InternetAddress("mail@mycompany.com"));  
        mimeMessage.setText("Hello world!");  
    }  
};  
  
javaMailSender.send(preparator);
```

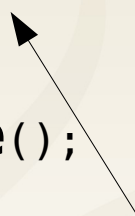
# Mime Mesaj Gönderme: 2.yol

```
MimeMessage message = javaMailSender.createMimeMessage();  
  
MimeMessageHelper helper = new MimeMessageHelper(message);  
  
helper.setTo("test@host.com");  
helper.setText("<html><body>Thank you!</body></html>", true);  
  
javaMailSender.send(message);
```

↓  
İçeriğin html olduğu  
belirtilebilir

# Attachment Gönderme

Multi part mesaj olduğu  
boolean flag ile belirtilir



```
MimeMessage message = javaMailSender.createMimeMessage();

MimeMessageHelper helper = new MimeMessageHelper(message, true);
helper.setTo("test@host.com");

helper.setText("Check out this image!");

FileSystemResource file = new FileSystemResource(
    new File("c:/Sample.jpg"));
helper.addAttachment("CoolImage.jpg", file);

javaMailSender.send(message);
```

# Inline Resource Gönderme

```
MimeMessage message = javaMailSender.createMimeMessage();

MimeMessageHelper helper = new MimeMessageHelper(message, true);
helper.setTo("test@host.com");

helper.setText("<html><body><img src=cid:id1234></body></html>", true);

FileSystemResource res = new FileSystemResource(
    new File("c:/Sample.jpg"));
helper.addInline("id1234", res);

javaMailSender.send(message);
```



Önce text mesajın ardından da inline resource'ların eklenmesi gerekir. Önce inline resource'lar eklenirse mesaj düzgün biçimde gönderilmeyecektir

# Mesaj Şablonları

- Kurumsal uygulamaların çoğunda mesajların içeriği, yapısı ve nasıl görüldüğü **şablonlar** vasıtası ile uygulama kodu dışında yönetilir
- Spring, FreeMarker, Velocity gibi “**template kütüphaneleri**”ni destekler
- Önce **şablon üzerinden mesaj içeriği** elde edilir ve ardından mesaj gönderilir



# Mesaj Şablonları Örnek Kullanım

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

```
<bean id="templateEngine"  
class="org.springframework.ui.velocity.VelocityEngineFactoryBean">  
  <property name="velocityProperties">  
    <value>  
      resource.loader=class  
      class.resource.loader.class=  
      org.apache.velocity.runtime.resource.loader.ClasspathResourceLoader  
    </value>  
  </property>  
</bean>  
  
</beans>
```

Spring Container içerisinde **template engine bean** olarak tanımlanmış olmalıdır

# Mesaj Şablonları Örnek Kullanım

```
<html>
  <body>
    <h3>Hi ${user.userName}, welcome to our system!</h3>
    <div>
      Your email address is <a href="mailto:${user.emailAddress}">${user.emailAddress}</a>.
    </div>
  </body>
</html>
```

Şablon mesaj içeriği hazırlanır, classpath'de, file system'de veya başka bir yerde saklanır

# Mesaj Şablonları Örnek Kullanım

```
MimeMessagePreparator preparator = new MimeMessagePreparator() {  
    public void prepare(MimeMessage mimeMessage) throws Exception {  
        MimeMessageHelper message = new MimeMessageHelper(mimeMessage);  
        message.setTo(user.getEmailAddress());  
        message.setFrom("me@example.com");  
        Map modelMap = new HashMap();  
        modelMap.put("user", user);  
  
        String text = VelocityEngineUtils.mergeTemplateIntoString(  
            templateEngine, "com/example/template.vm", modelMap);  
  
        message.setText(text, true);  
    }  
};
```

```
javaMailSender.send(preparator);
```

templateEngine bean'i, şablon dosyası ve şablon input'unu içeren model map kullanılarak mesaj içeriği elde edilir

# Dumbster ile E-Mail Gönderme Testi

```
private SimpleSmtpServer smtpServer;
```

```
@Before
```

```
public void setUp() {  
    smtpServer = SimpleSmtpServer.start();  
}
```

Dumbster isimli bir kütüphane ile Standalone ortamda test amaçlı bir SMTP sunucu çalıştırılabilir

```
@Test
```

```
public void testEmail() {  
    // mail gönderme işlemleri...  
    int size = smtpServer.getReceivedEmailSize();  
    if (size > 0) {  
        Iterator receivedEmails = smtpServer.getReceivedEmail();  
        while (receivedEmails.hasNext()) {  
            Object message = receivedEmails.next();  
            // mesaj üzerinde işlemler...  
        }  
    }  
}
```

```
@After
```

```
public void tearDown() {  
    smtpServer.stop();  
}
```

Test sonunda SMTP sunucunun kapatılması gerekir

# İletişim

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

