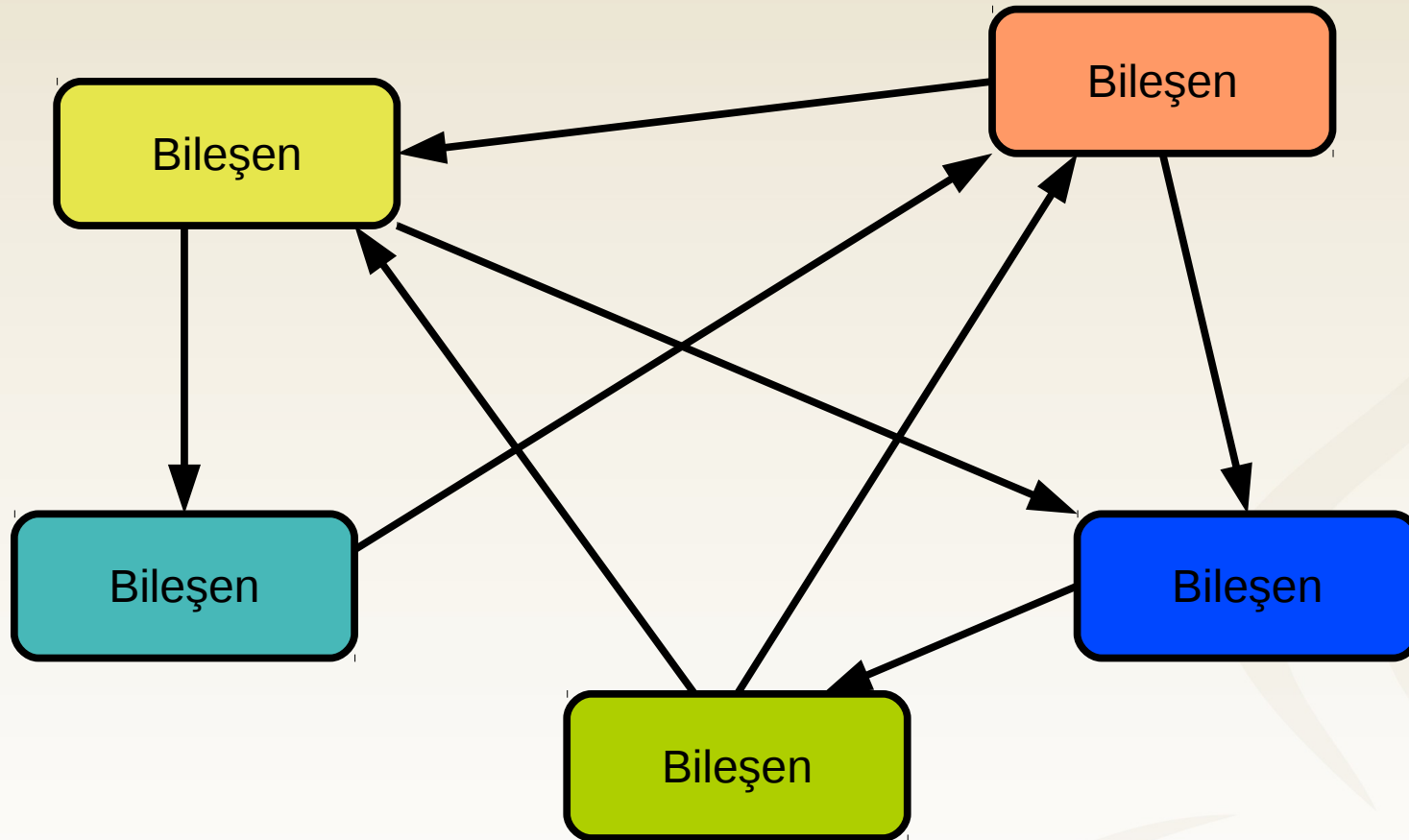


Tasarım Örüntüleri ile Spring Eğitimi 8

Tasarım Örüntülerine Devam

Mediator

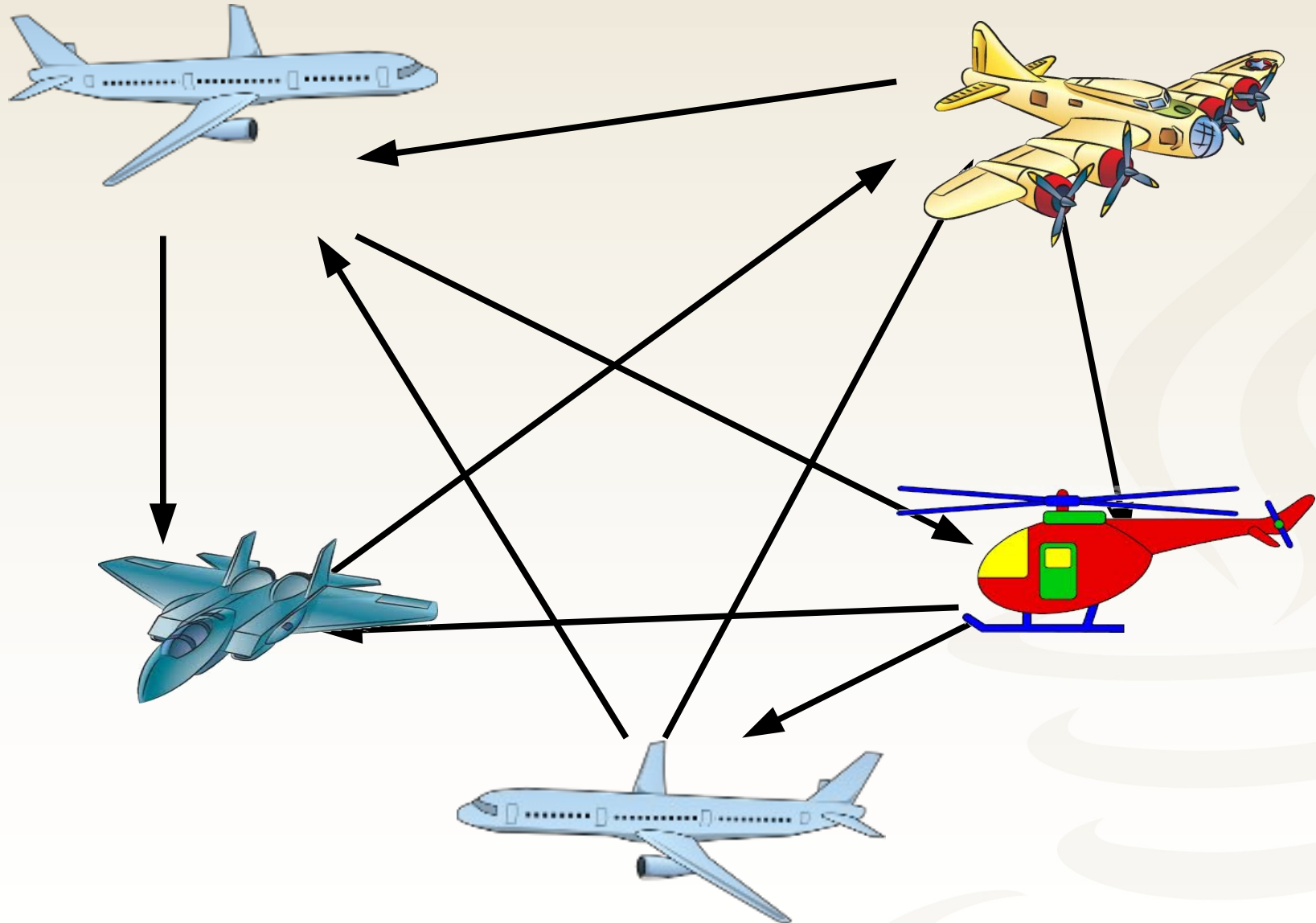


Belirli bir nesne grubunun birbirleri arasında **kompleks bir iletişim sekansı** söz konusu olabilir

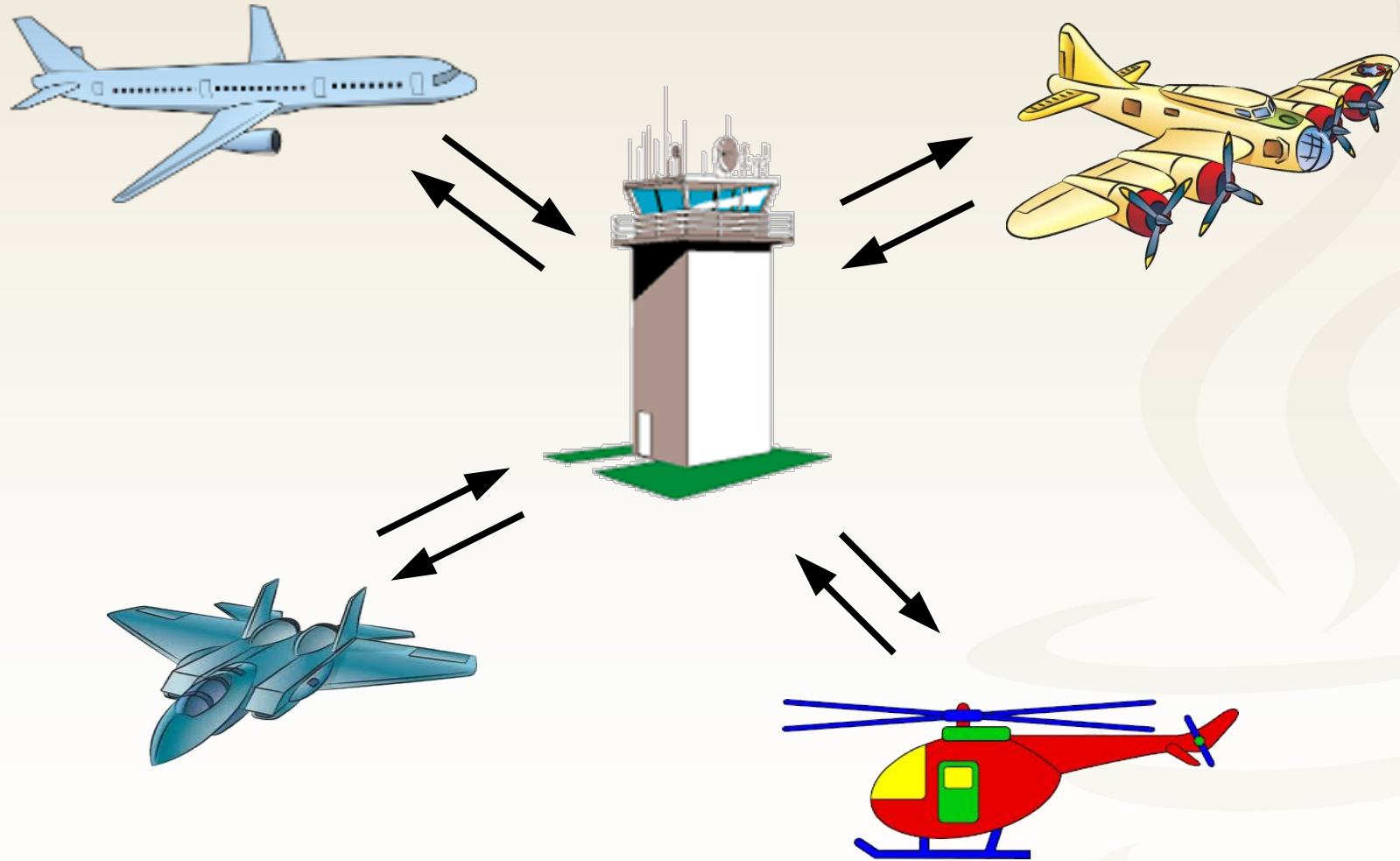
Nesnelerin **birbirleri arasında pek çok ilişki** söz konusudur

Bu iletişim sekansı içinde yer alan nesnelerin kendi davranışları ne kadar yeniden kullanılabilir olsa bile, bu ilişkiler nedeni ile teker teker **başka context'de kullanılmaları oldukça zordur**

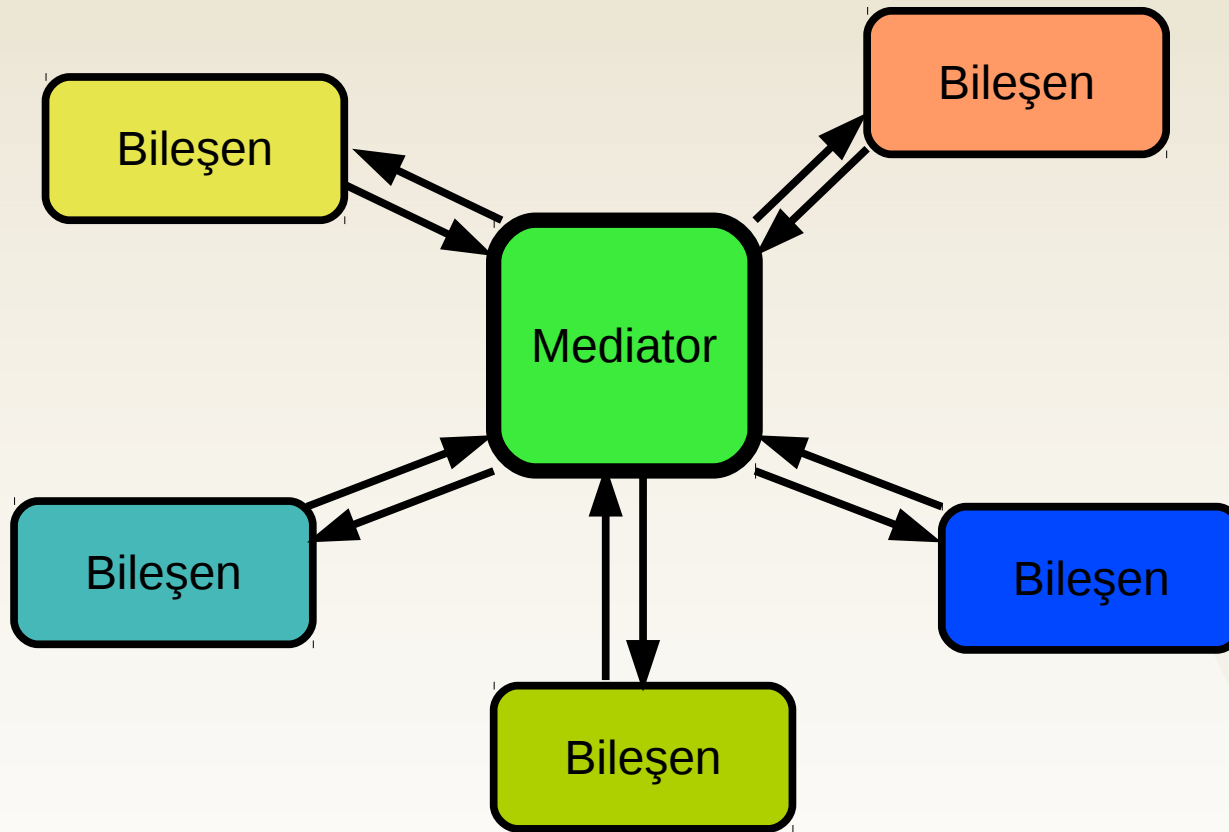
Mediator



Mediator



Mediator

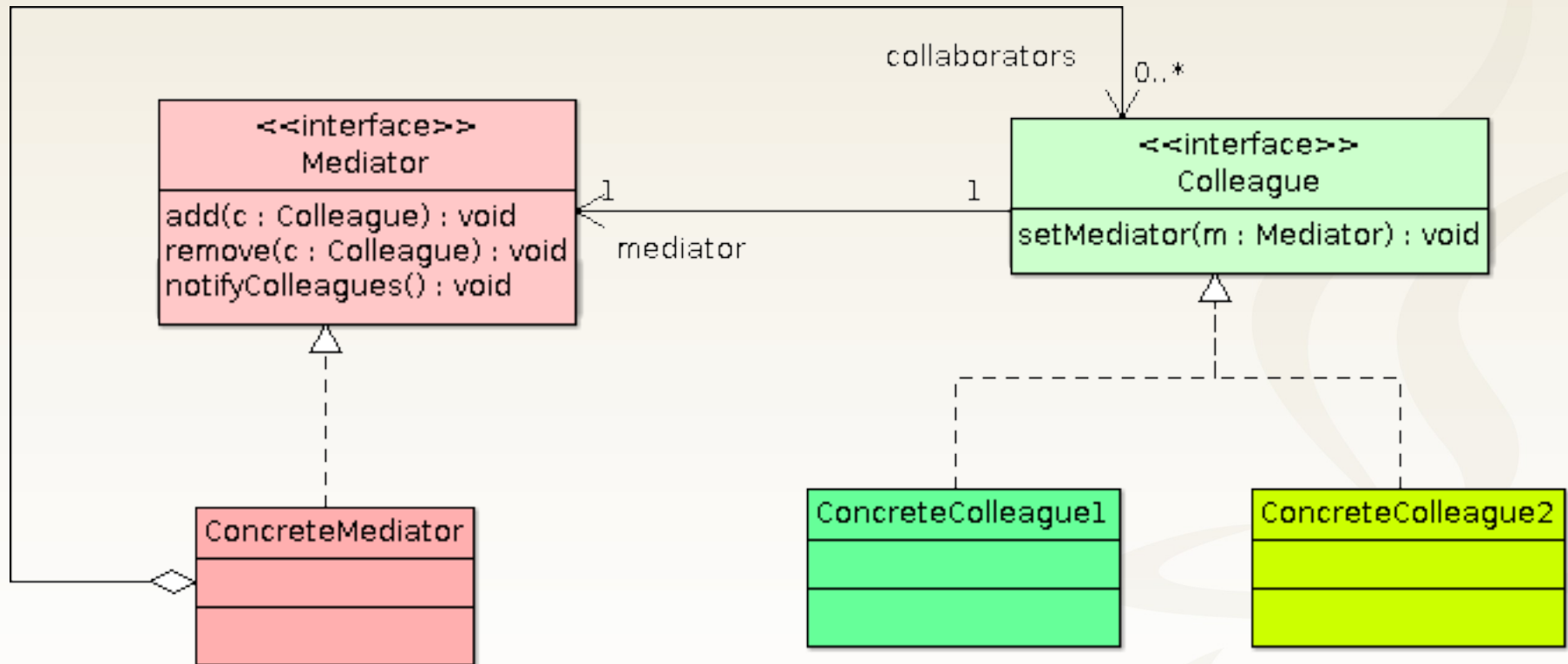


Kompleks nesne etkileşimleri ve ilişkileri ayrı bir nesne içerisinde encapsule edilir

Mediator nesneler arasındaki **koordinasyondan** sorumludur

Grup içerisindeki nesnelerin birbirleri ile doğrudan ilişkiye girmelerini önler, nesneler **sadece Mediator nesnesini bilirler**

Mediator Sınıf Diagramı



Mediator

Örüntüsünün Sonuçları

- Nesneler arası **kompleks etkileşim tek bir yerden** yönetilir
- **Nesnelerin birbirlerinden bağımsız** biçimde değişmelerini ve farklı context'lerde yeniden kullanılabilmelerini sağlar
- Mediator örüntüsü hemen her zaman **Observer örüntüsü ile birlikte** kullanılır

Spring ve Event Yönetimi

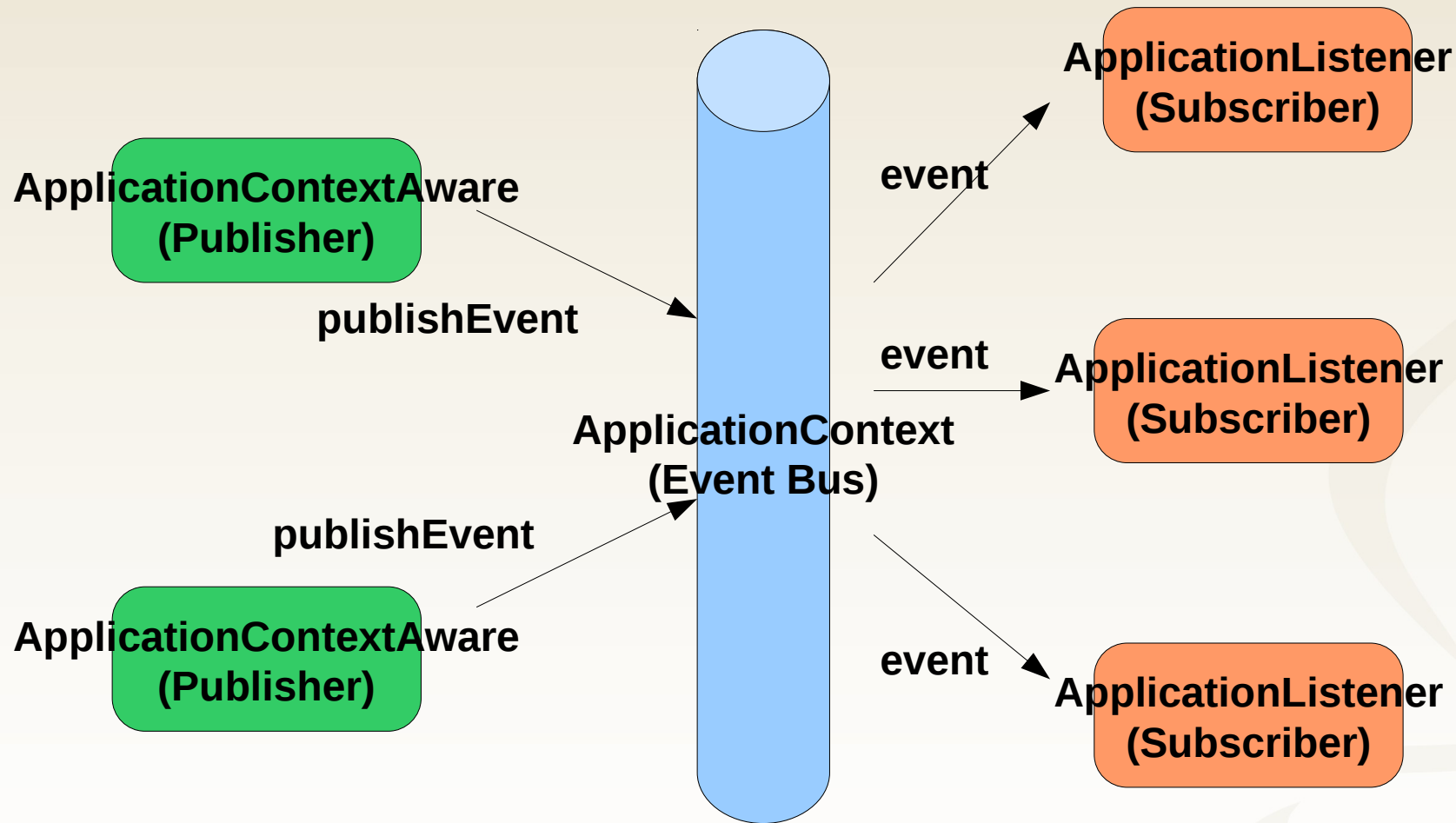
ApplicationContext ve Event'ler

- ApplicationContext üzerinden belirli durumlarda **event'ler fırlatılabilir**
- Fırlatılan event'ler **ApplicationEvent** tipinde nesnelerdir
- Event fırlatma işi
ApplicationContext.**publishEvent** metodu ile gerçekleştirilir
- Event fırlatacak Spring managed bean'lerine **ApplicationContext enjekte edilmelidir**

ApplicationEvent'lerin Yakalanması

- Diğer Spring managed bean'ları da bu **event'leri yakalayıp** birtakım işlemler gerçekleştirebilirler
- Event handler bean sınıflarının ise **ApplicationListener** arayüzünü implement etmesi gerekir

ApplicationContext Event Mimarisi



Burada **Observer** ve **Mediator** örüntüleri birlikte kullanılmaktadır

ApplicationContext'de Event Yönetimi

- ApplicationListener nesnelerine event'ler **senkron** biçimde verilir
- Bu durumda publishEvent() bütün listener'lar çalışana değin **süreci bloklar**
- İstenirse **ApplicationEventMulticaster** arayüzü implement edilerek event yönetimi **asenkron biçimde** de gerçekleştirilebilir

Built-in Event Tipleri

- Spring Container **built-in event'ler** fırlatmaktadır
 - ContextRefreshedEvent
 - ContextClosedEvent
 - RequestHandledEvent
- Ayrıca Spring Security gibi diğer framework'lerin de **built-in event'leri** mevcuttur
 - AuthenticationSuccessEvent
 - AbstractAuthenticationFailureEvent

Uygulamaya Özel Event Tipleri

- Uygulamaya özel event'ler de tanımlanıp, ApplicationContext üzerinden fırlatılabilir
- Event sınıfları **ApplicationEvent** sınıfından türetilmelidir
- Bu event'leri yaratıp fırlatmak için de **ApplicationContext.publishEvent()** metodu kullanılır

Uygulamaya Özel Event Tipleri

```
public class EmailSender implements ApplicationContextAware {  
  
    private List blacklist;  
    private ApplicationContext appContext;  
  
    public void sendEmail(String address, String text) {  
        if (blacklist.contains(address)) {  
            BlackListEvent event = new BlackListEvent(address, text);  
            appContext.publishEvent(event);  
            return;  
        }  
        // email gönderme işlemi...  
    }  
}
```


Uygulamaya Özel Event Tipleri

```
public class BlackListNotificationHandler
    implements ApplicationListener {

    private String notificationAddress;

    ...

    public void onApplicationEvent(ApplicationEvent
event) {
        if (event instanceof BlackListEvent) {
            //sistem yoneticisi vs. bilgilendirilir...
        }
    }
}
```

Uygulamaya Özel Event Tipleri (Java Generics)

```
public class BlackListNotificationHandler
    implements ApplicationListener<BlackListEvent> {

    private String notificationAddress;

    ...

    public void onApplicationEvent(BlackListEvent event) {
        //sistem yoneticisi vs. bilgilendirilir...
    }
}
```

Uygulamaya Özel Event Tipleri

```
<bean id="emailSender" class="example.EmailSender">  
  <property name="blackList">  
    <list>  
      <value>black@list.org</value>  
      <value>white@list.org</value>  
      <value>john@doe.org</value>  
    </list>  
  </property>  
</bean>
```

```
<bean id="blackListListener"  
class="example.BlackListNotificationHandler">  
  <property name="notificationAddress" value="spam@list.org"/>  
</bean>
```

Annotasyon Tabanlı Event Listener Tanımı

- Spring 4.2 ile birlikte **ApplicationListener** arayüzünü **implement etmeden** de event handler tanımlamak mümkün hale gelmiştir
- Bunun için **metot düzeyinde @EventListener anotasyonu** kullanılır
- Bu sayede bir bean içerisinde **birden fazla farklı türde event'i yakalayan metotlar** yazmak mümkündür

Annotasyon Tabanlı Event Listener Tanımı

```
@Component
public class MyApplicationEventListener {

    @EventListener
    public void handle(ContextRefreshedEvent event) {
        System.out.println("ApplicationContext is ready to serve...");
    }

    @EventListener
    public void handle(ContextClosedEvent event) {
        System.out.println("ApplicationContext shutdown...");
    }
}
```


Transactional Event Listener Kabiliyeti

- **@TransactionalEventListener** anotasyonu event'lerin sadece **transactional bir metot içerisinde** fırlatıldıklarında yakalanmalarını sağlar
- Eğer event transactional bir metot içerisinde fırlatılmamış ise **göz ardı** edilir
- Metodun çağırılma zamanı da **TransactionPhase** ile belirlenebilir
- Böylece metodun TX **commit öncesi** veya **sonrası**, yada sadece **rollback** olduğunda invoke edilmesi sağlanabilir

Transactional Event Listener Kabiliyeti

```
@Component
public class MyApplicationEventListener {

    @TransactionalEventListener(
        phase=TransactionPhase.AFTER_COMMIT)
    public void handle(PetCreatedEvent event) {
        //handle pet created event here...
    }
}
```



BEFORE_COMMIT
AFTER_COMMIT
AFTER_ROLLBACK
AFTER_COMPLETION

Ayrıca **fallbackExecution=true** ifadesi ile transactional context dışında fırlatılan bir event'in de yakalanması sağlanabilir

Spring Security

Spring ve Güvenlik

- Spring Security Framework, **web uygulamalarına** yönelik bir güvenlik framework'üdür
- Web uygulamalarının **kimliklendirme** (authentication) ve **yetkilendirme** (authorization) ihtiyaçlarını karşılar
- Spring üzerine kuruludur ve Servlet **Filter tabanlıdır**

Spring Security'nin Özellikleri

- Pek çok değişik **kimliklendirme** (authentication) yöntemini desteklemektedir
 - Form tabanlı, LDAP, basic, digest, NTLM, Kerberos, JAAS...
- Kullanıcı bilgisinin **değişik 'realm'lerden** (Memory, LDAP, JDBC...) alınmasını sağlar
- CAS, OpenID, OpenAuth, Siteminder gibi **SSO çözümleri ile entegre** çalışabilmektedir

Spring Security'nin Özellikleri

- URL (web resource'ları), servis metotları ve domain nesneleri (DB kayıtları) düzeyinde **yetkilendirme** sağlar
- Aynı kullanıcının farklı noktalardan sisteme erişim sayısını, **kullanıcı oturumlarını** yönetir
- “**Beni hatırla**” (remember me) desteği sunar
- Güvenlikli bir HTTP iletişimi (**HTTPS ile erişim**) kurmaya yardımcı olur
- Şifreleri **kriptolayarak** tutmaya yardımcı olur

Spring Security Filter Zinciri

- Spring Security **Servlet Filter** tabanlı bir framework'tür
- Her bir güvenlik gereksinimi **ayrı bir Filter** tarafından ele alınır
- Bu Filter'lar web isteği üzerinde **birbirleri ardı sıra** işlem yaparlar
- Dolayısı ile aralarındaki **sıralama önemlidir**

Spring Security

Konfigürasyonu: web.xml

```
<web-app>
  <filter>
    <filter-name>
      springSecurityFilterChain
    </filter-name>
    <filter-class>
      org.springframework.web.filter.DelegatingFilterProxy
    </filter-class>
  </filter>

  <filter-mapping>
    <filter-name>
      springSecurityFilterChain
    </filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
</web-app>
```

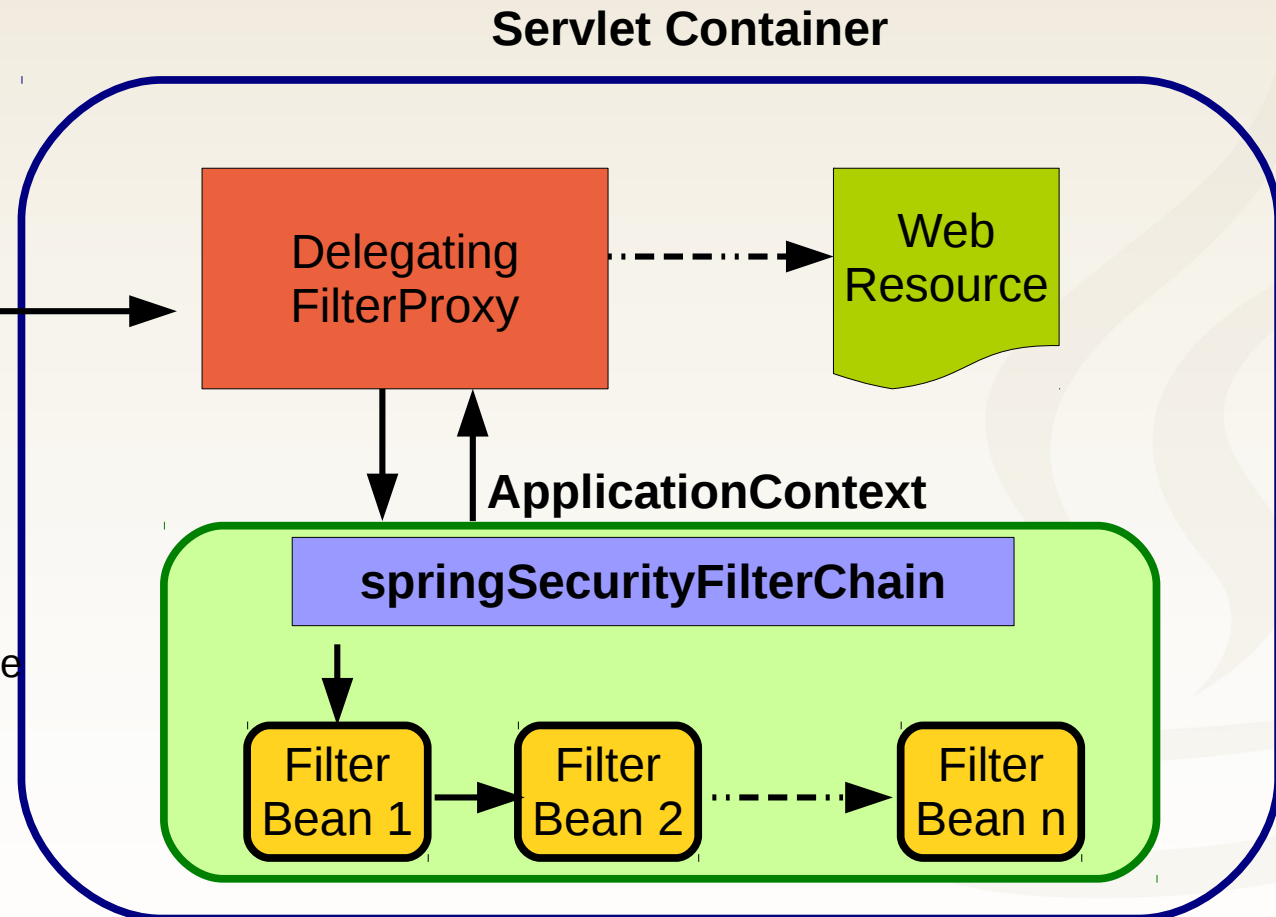
web.xml içerisinde
DelegatingFilterProxy ile
bir Servlet Filter tanımı
yapılır

Spring Security Filter Mimarisi

DelegatingFilterProxy sınıfı, web.xml içerisindeki tanım ile ApplicationContext içinde tanımlı Filter bean'ları arasında **köprü vazifesi** görür



Web client



DelegatingFilterProxy işi Spring Container'da tanımlı **FilterChainProxy** tipindeki bean'e (springSecurityFilterChain) delege eder. Böylece **tek bir filter** ile bütün security filter zincirinin tanımlanması sağlanır.

Spring Security

Konfigürasyonu: Filter Zinciri

Otomatik olarak bir login form render eder, form authentication'ın yanında basic authentication'ı da devreye alır, default logout url'i ve logout servislerini register eder

```
<security:http auto-config="true">
```

```
<security:intercept-url pattern="/**" access="hasRole('ROLE_USER')" />
```

```
</security:http>
```

springSecurityFilterChain id'li
FilterChainProxy tipinde bean tanımını
Spring Container'a otomatik olarak ekler

Bu bean içerisinde de her bir http elemanına
karşılık gelen bir SecurityFilterChain bean'i
eklenir

Uygulamadaki web kaynaklarına
Erişimi yetkilendirir

Konfigürasyonu: AuthenticationManager

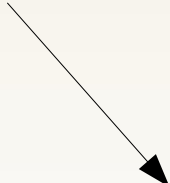
Kimliklendirme işlemini gerçekleştirmesi için **authenticationManager** ve **authenticationProvider** konfigürasyonu yapılır

```
<security:authentication-manager>  
  <security:authentication-provider  
    user-service-ref="userService">  
  
  </security:authentication-provider>  
</security:authentication-manager>
```

AuthenticationProvider, kullanıcı bilgilerine erişebilmek için **UserDetailsService** bean'ine ihtiyaç duyar

Konfigürasyonu: UserDetailsService

```
<security:user-service id="userService"  
    properties="classpath:/users.properties">  
</security:user-service>
```



Herhangi bir user realm'i kullanarak **userDetailsService** tanımlanır. Burada test amaçlı olarak **in-memory realm** kullanılmıştır

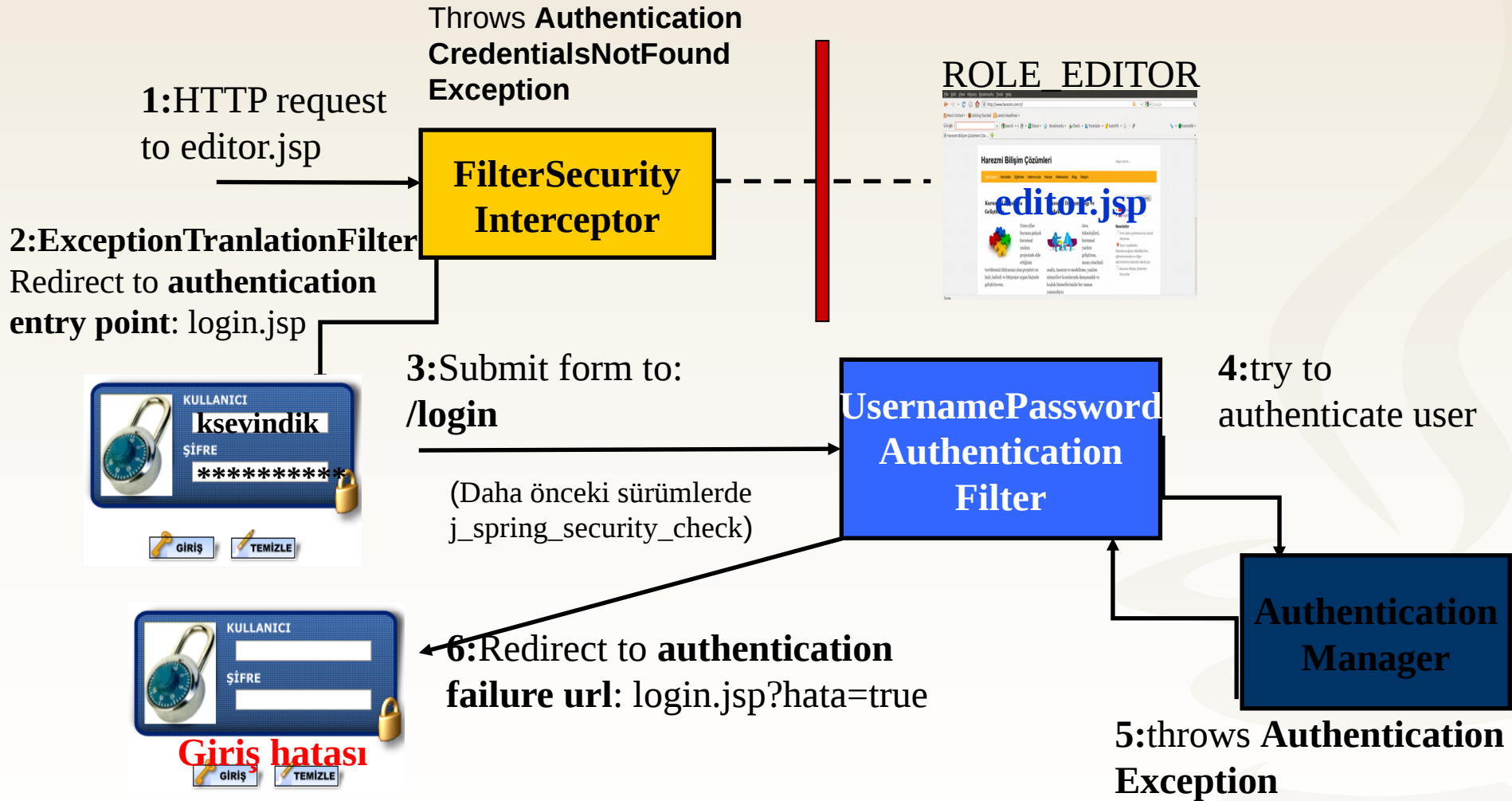
Spring UserDetailsService'in JDBC ve LDAP gerçekleştirmeleri için de built-in namespace elemanları sağlar

Kimliklendirme (Authentication)

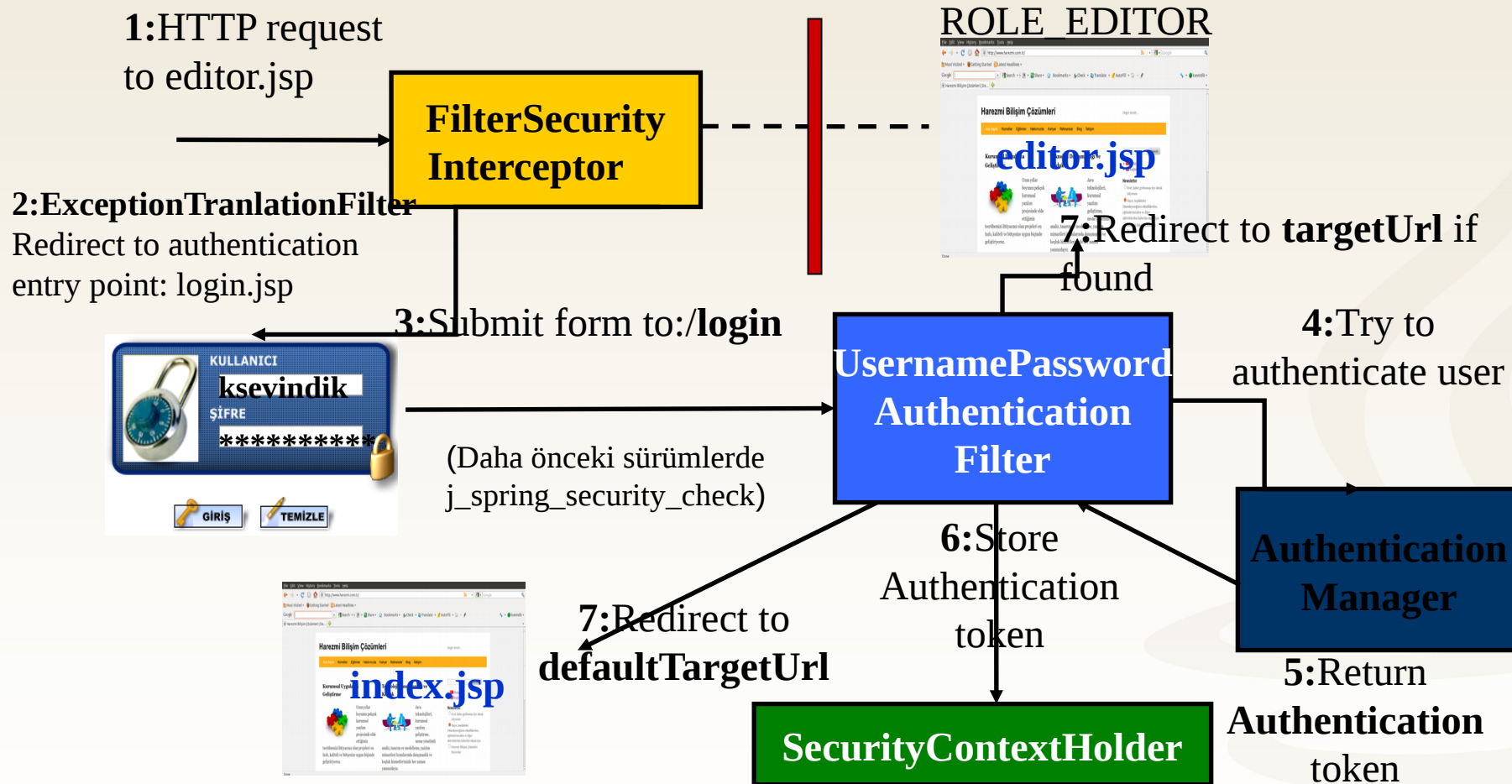
Kimliklendirme Nedir?

- Sistemdeki herhangi bir kaynağa erişmeye çalışan **kullanıcı veya harici sistemin kimliğinin denetlenmesi** işlemidir
- Öncelikle kullanıcı veya harici sistem belirttiği kimlik bilgisini kanıtlayan **gizli veya benzersiz bir bilgiyi** sunucuya iletir
- Sunucu bu bilgiyi kullanarak kullanıcının **kimlik bilgisini denetler**
- İşlem başarılı ise kullanıcı **sisteme giriş** yapabilir

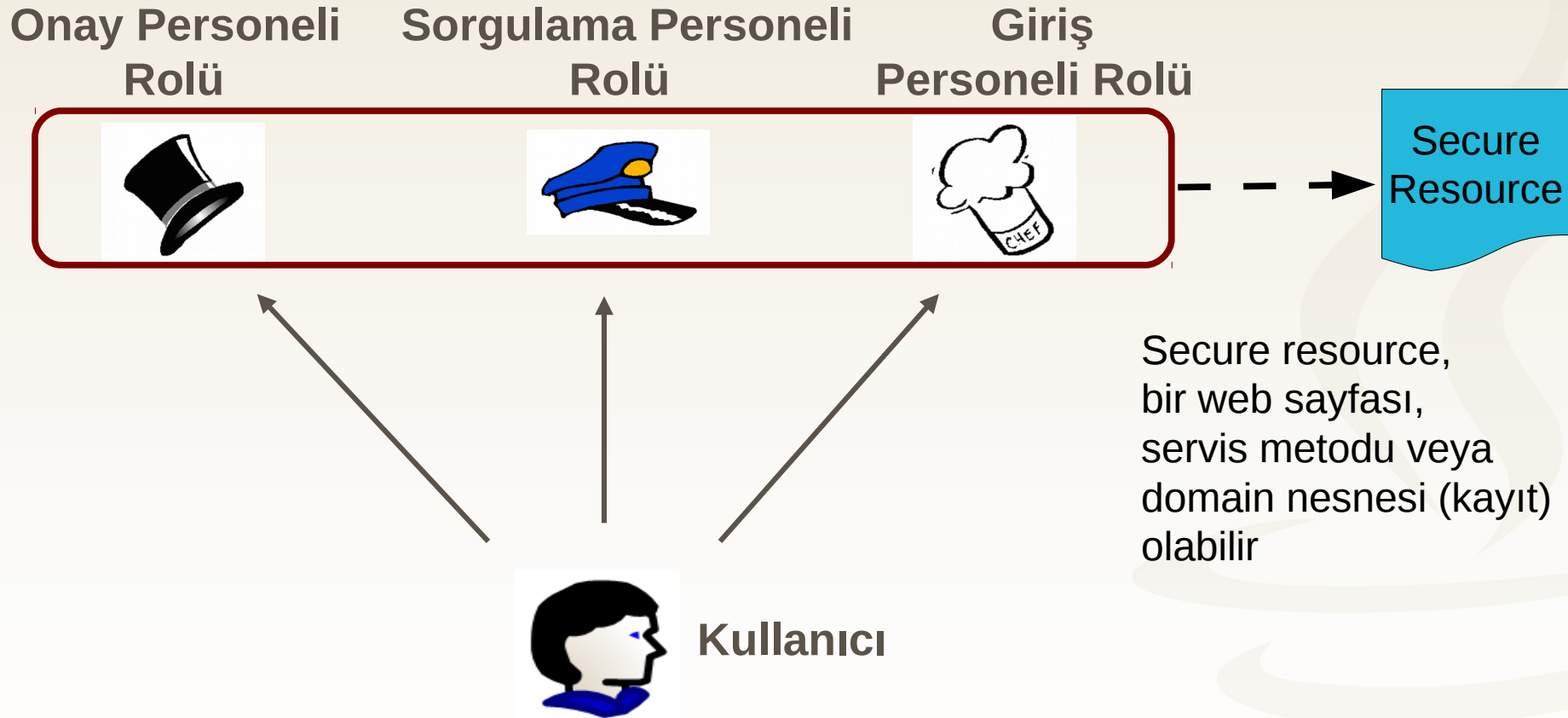
Başarısız Bir Login Akış Örneği



Başarılı Bir Login Akış Örneği



Spring Security Kullanıcı – Rol İlişkisi



Kimlik Bilgisinin İki Web Requesti Arasında Saklanması

- Web uygulamalarında principal bilgisi request'ler arasında **HttpSession** içerisinde saklanır
- Request başlangıcında **SecurityContext** HttpSession'dan alınır ve **SecurityContextHolder**'a set edilir
- Request sonunda da SecurityContextHolder temizlenir
- Request boyunca uygulama tarafında herhangi bir sınıf içerisinde **SecurityContextHolder** üzerinden **SecurityContext**'e erişmek mümkündür

Kimliklendirme Bilgilerine Erişim

- **SecurityContextHolder** üzerinden istediğimiz herhangi bir yerden **Authentication** bilgilerine erişebiliriz

```
SecurityContext securityContext = SecurityContextHolder.getContext();
```

```
UsernamePasswordAuthenticationToken authenticationToken =  
(UsernamePasswordAuthenticationToken) securityContext.getAuthentication();
```

```
String username = authenticationToken.getName();
```

```
User principal = (User) authenticationToken.getPrincipal();
```

```
Collection<GrantedAuthority> authorities =  
    authenticationToken.getAuthorities();
```

```
boolean authenticated = authenticationToken.isAuthenticated();
```


Login Sayfasının Özelleştirilmesi

Spring 4 öncesi login processing url
j_spring_security_check idi

```
<form action="login" method="post">  
  Username:<input name="username" type="text"/><br/>  
  Password:<input name="password" type="password"/><br/>  
  Remember:<input name="remember-me" type="checkbox">  
  
  <input type="hidden" name="${_csrf.parameterName}"  
  value="${_csrf.token}">  
  
  <input type="submit" value="Login">  
</form>
```

Beni hatırla kabiliyeti
aktif ise gereklidir

Spring 4 öncesi username
parametresi j_username,
password parametresi
j_password, remember-me
ise
_spring_security_remember_me
idi

Spring Security 4 ile birlikte csrf koruması
default olarak aktif gelmeye başladı. Dolayısı ile CSRF
aktif ise login sayfası içerisinde de csrf token'ın yönetilmesi
için bir gizli input alanın eklenmesi gerekir

Login Sayfasının Özelleştirilmesi

```
<security:http>
```

```
<security:form-login login-page="/login.jsp"  
  authentication-failure-url="/login.jsp?error=true"/>
```

```
<security:intercept-url pattern="/login.jsp"  
access="permitAll"/>
```

```
<security:intercept-url pattern="/**"  
access="hasRole('ROLE_USER')"/>
```

```
</security:http>
```


Anonim kimliklendirme yerine **permitAll** ifadesi de kullanılabilir

/login.jsp request URL'inin intercept edilmemesi gerekir. Aksi takdirde login.jsp sayfasında yetkilendirmeye tabi tutulacağı için sayfa düzgün biçimde render edilemeyecek ve **"The page isn't redirecting properly"** şeklinde bir hata alınacaktır.

Login Sayfasının Özelleştirilmesi

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

```
<security:http pattern="/login.jsp" security="none"/>
```



csrf koruma
aktif ise security none
kullanımı
uygun değildir!

Bu kullanım şekli daha
çok Spring Security 4
öncesi dönemde karşımıza
çıkmaktadır

```
<security:http>
```

```
<security:form-login login-page="/login.jsp"  
authentication-failure-url="/login.jsp?error=true"/>
```

```
<security:intercept-url pattern="/**"  
access="hasRole('ROLE_USER')" />
```

```
</security:http>
```

```
</beans>
```

Logout İşlemi

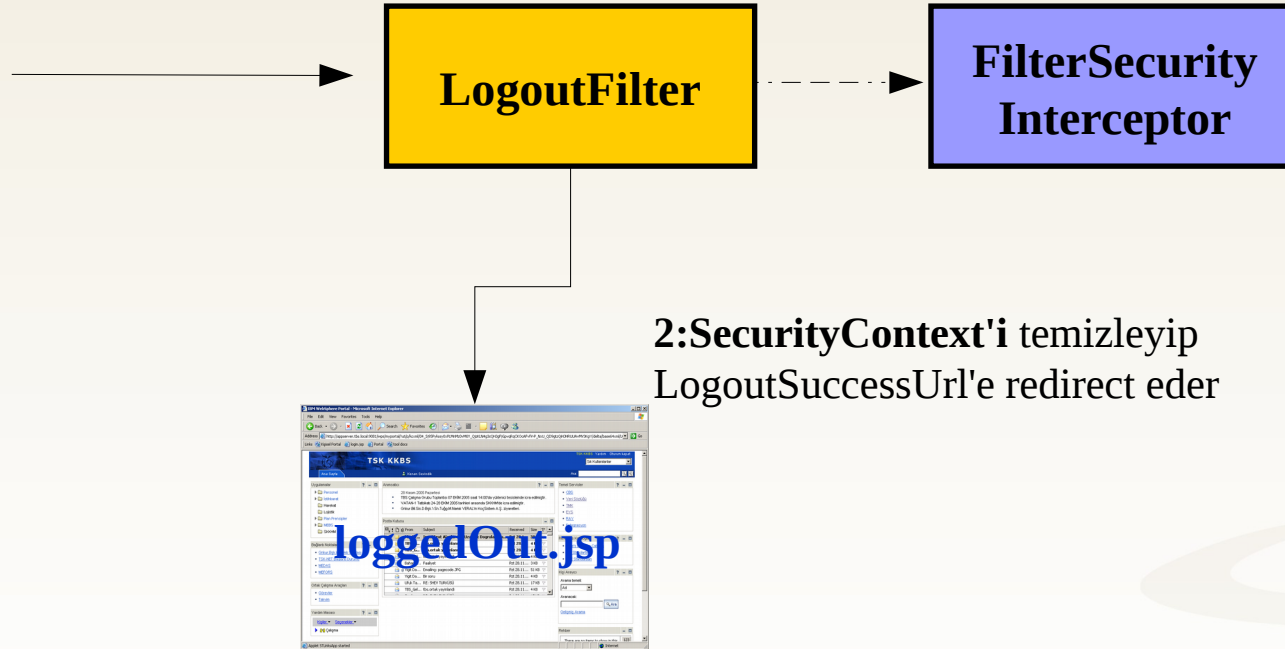
- Logout işlemi **LogoutFilter** tarafından yönetilir
- **Logout URL**'ine yapılmış istekler logout işlemini tetikler
- Default logout URL **/logout** şeklindedir
- CSRF koruması nedeni ile LogoutFilter **sadece POST** request'lerinde devreye girmektedir
- Dolayısı ile logout işlemi için de bir **form post metodu ile submit** edilmelidir

Logout işlemi

Spring Security 4 öncesi logout için request url: j_spring_security_logout

1: HTTP request to **/logout**

Http elemanı içerisinde **</logout>** elemanını tanımlamak yeterlidir



Logout İşlemi

```
<security:http>

    <security:logout logout-success-url="/loggedOut.jsp"/>

        <security:intercept-url pattern="/loggedOut.jsp"
access="permitAll"/>

        <security:intercept-url pattern="/**"
access="hasRole('ROLE_USER')" />

</security:http>
```

Logout İşlemi

- LogoutFilter logout sırasında bir grup **LogoutHandler** nesnesini çalıştırır
- **TokenBasedRememberMeServices** ve **SecurityContextLogoutHandler** en yaygın kullanılan iki LogoutHandler'dır
- Logout işlemi sonucu uygulama bir **logout-success-url** sayfasına **redirect** edilir

CSRF Saldırısı Nedir?

- Kullanıcı, tarayıcısı üzerinden **authenticated bir servise** erişim halinde iken, saldırganların kullanıcının authenticated durumundan yararlanarak bu site üzerinde **bir işlem gerçekleştirebildiği** saldırı şeklidir
- Açığın temeli authentication bilgilerinin istemci tarafında **cookie şeklinde tutulması ve her istekte sunucuya geri gönderilmesidir**

Örnek Bir CSRF Saldırısı

- Kullanıcı **WEBMAIL** hesabına **login** olur
- Bir süre hesabında vakit geçirdikten sonra **logout olmadan** tarayıcıda **yeni bir tab** açar
- Burada da **farklı bir sistemin sayfasına** erişir
- Bu sayfada **WEBMAIL sunucusuna kötü amaçlı istek gönderen bir HTML tag** yer alır
- Örneğin, bu istekte bir mesajı silme, kullanıcı ayarlarını değiştirme veya başka birilerine spam mail gönderme **komutu** olabilir
- Kullanıcının **WEBMAIL oturumu hala açık** olduğu için bu **istek kullanıcının yetkileri dahilinde işletilir**

CSRF Nasıl Önlenir?

- Sunucu her web requesti için **benzersiz ve gizli bir token** üretir (synchronizer token)
- Bu token'ı kullanıcıya gönderir
- Kullanıcı **bir sonraki request'i bu token ile birlikte** gerçekleştirir
- Token sunucu tarafında **her seferinde kontrol** edilir
- Eğer geçerli ise işleme izin verilir

Spring Security CSRF Konfigürasyonu

- **<http>** elemanı altında **<csrf/>** elemanı ile konfigüre edilir

```
<security:http>  
  <security:csrf/>  
</security:http>
```

token-repository-ref attribute'u default durumda **HttpSessionCsrfTokenRepository** sınıfından bir bean kullanmaktadır

request-matcher-ref attribute'u request'de CSRF kontrolünün uygulanıp uygulanmayacağına karar verir. Default durumda GET, TRACE, HEAD, OPTIONS metotları haricinde uygulanır

Spring Security CSRF Konfigürasyonu

- GET metodu ile yapılan herhangi bir istekte **yeni bir CSRF token** üretilir ve istemciye dönülür
- İstemciden gelecek istekte de **bu token yer almalıdır**
- Bu CSRF tokenın sonraki **web request'lerinde yer alması** sağlanmalıdır
- Bunun için bütün POST, PUT, DELETE, PATCH metotlarında bir şekilde (örneğin bir hidden input alan ile) **CSRF token'ı web request'ine eklenmelidir**

Spring Security CSRF Konfigürasyonu

```
<form action="${actionUrl}"
  method="post">
  <input type="submit"
    value="Click It!" />
  <input type="hidden"
    name="${_csrf.parameterName}"
    value="${_csrf.token}" />
</form>
```

- **GET isteği tekrarlandığı** vakit CSRF token'ı HTTP Session'da saklandığı için **yeni bir token yerine aynı token** dönmektedir

- Spring Security, geçersiz bir CSRF token ile karşılaşıldığında **InvalidCsrfTokenException** fırlatır
- Bu exception **AccessDeniedHandler** tarafından ele alınır ve default olarak **403 access denied hatası** üretilir

Kriptolu Şifrelerin Kullanılması

- Spring Security şifrelerin DB'de salt metin değilde, **kriptolu olarak** saklanmasını sağlar
- Şifreleri kriptolamak için **değişik algoritmalar** desteklenir
- Bunlar **tek yönlü** algoritmalarıdır
- Kimliklendirme sırasında kullanıcının girdiği şifre algoritmaya göre kriptolanarak **DB'deki kriptolu değer** ile karşılaştırılır

Kriptolu Şifrelerin Kullanılması

- **authentication-provider** elemanı altında tanımlı **password-encoder**'ın **hash attribute**'u ile kripto algoritması belirtilir

```
<security:authentication-manager>
```

```
  <security:authentication-provider  
    user-service-ref="userService">
```

```
    <security:password-encoder hash="bcrypt"/>
```

```
  </security:authentication-provider>
```

```
</security:authentication-manager>
```

bcrypt, plaintext, sha, sha-256, md4, md5 gibi hashing algoritmaları desteklenmektedir. Spring 4 ile **bcrypt** kullanılması önerilir

Kriptolu Şifrelerin Kullanılması

- Kriptolu şifreleri oluşturmak için seçilen **algoritmaya karşılık gelen encoder sınıf** kullanılmalıdır

```
BCryptPasswordEncoder passwordEncoder = new BCryptPasswordEncoder();
```

```
String encodedPasswd = passwordEncoder.encode("plain text passwd");
```


Beni Hatırla Kabiliyeti

- Kullanıcının, tarayıcısını kapatıp açtığında belirli bir süre tekrar **login olmadan sisteme doğrudan girebilmesini** sağlar
- Web uygulamalarında **çok yaygın** bir özelliktir
- Username bilgisi **remember-me cookie** içerisinde saklanmaktadır
- Tarayıcı kapatılıp açıldığında bu cookie sunucuya gönderilir, sunucu da **cookie'yi validate ederek** kimliklendirmeyi gerçekleştirir

Beni Hatırla Kabiliyeti

```
<security:http>
    <security:remember-me user-service-
ref="userService" />
</security:http>
```

Basic Authentication Nedir?

- **Browser authentication** olarak da bilinir
- Remoting protokollerinde ve web servislerde **yaygın** biçimde kullanılmaktadır
- Tarayıcının açtığı bir **login/password dialog**'u üzerinden kullanıcı credential bilgileri girilir
- Credential bilgileri HTTP **Authorization request header**'ı ile sunucuya gönderilir
- Gönderim sırasında **username:password** token'ı **Base64** ile encode edilmelidir
- **HTTPS** ile birlikte kullanılması gerekir

Basic Authentication Konfigürasyonu

```
<security:http pattern="/mvc/rest/**"  
    create-session="stateless">
```

```
<security:http-basic />
```

```
<security:intercept-url pattern="/**/vets/list"  
Access="hasRole('ROLE_USER')" requires-channel="https"/>
```

```
<security:intercept-url pattern="/**/vets/new"  
access="hasRole('ROLE_EDITOR')" requires-channel="https"/>
```

```
</security:http>
```

Basic Authentication İsteğinin Yapısı

- Basic authentication sunucudan gelen aşağıdaki gibi bir **401 statü kodlu HTTP cevabı** ile başlar

```
HTTP/1.1 401 Unauthorized  
WWW-Authenticate: Basic realm="<realm>"
```

↙

Tarayıcı tarafından görüntülenecek login dialog penceresinde görüntülenen kullanıcının neresi için username, password gireceğini anlamasını sağlayan bir mesajdır

Basic Authentication İsteğinin Yapısı

- İstemci tarafından gönderilecek HTTP isteğinde yer alması gereken **Authorization header değeri** de aşağıdaki gibi bir yapıya sahiptir

```
Authorization: Basic <credentials>
```

- Credentials ise aşağıdaki gibi üretilmiş olmalıdır

```
<credentials> := base64(username + ":" + password)
```

Basic Auth İsteyen REST Servislerine Erişim

- RestTemplate'in **interceptor** kabiliyeti ile **basic authentication header bilgisini** web isteğine ekleyerek isteklerimizi gönderebiliriz

```
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);
```

Kullanıcı Bilgilerine JDBC ile Erişilmesi

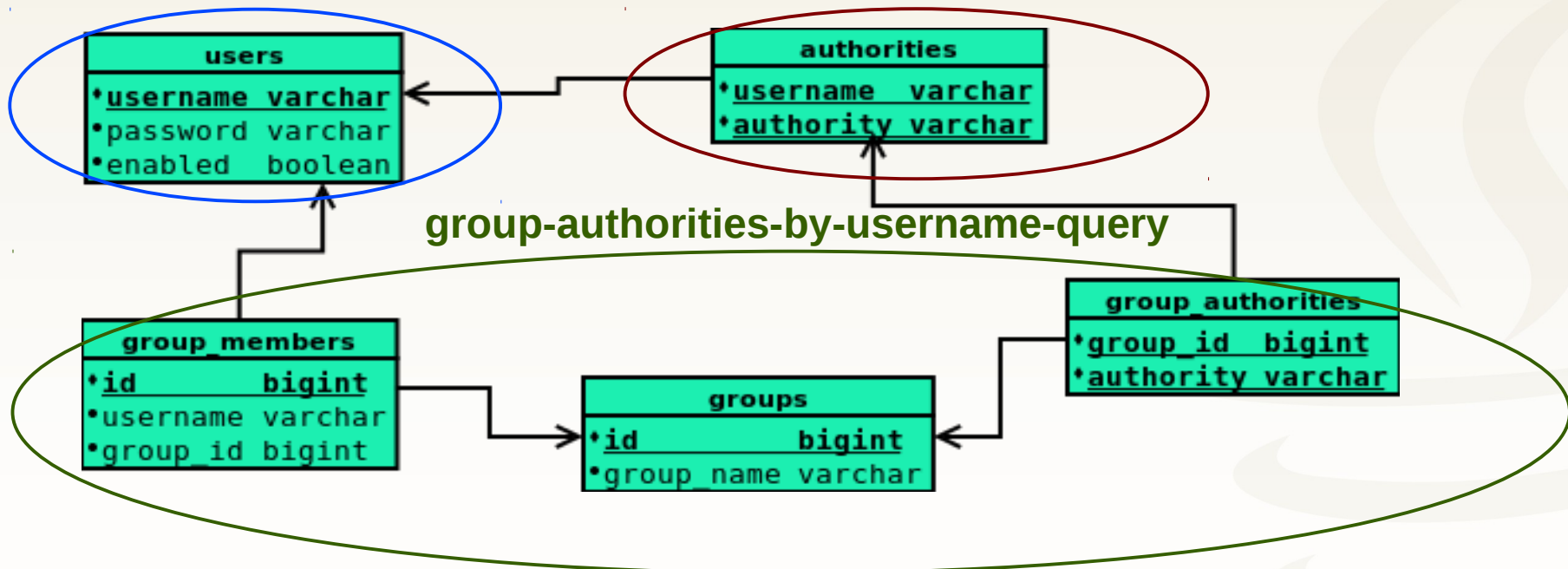
- Kullanıcı kimlik, rol ve grup bilgileri çoğunlukla **ilişkisel veritabanlarında** yönetilir
- Spring Security, JDBC ile kimlik bilgilerinin erişilmesini sağlayan bir **UserDetailsService** gerçekleştirimine sahiptir
- Bu servis `<security:jdbc-user-service>` elemanı ile uygulamaya göre özelleştirilebilir

Kullanıcı Bilgilerine JDBC ile Erişilmesi

```
<security:jdbc-user-service data-source-ref="dataSource" id="jdbcUserDetailsService"/>
```

users-by-username-query

authorities-by-username-query



Kullanıcı Bilgilerine JDBC ile Erişilmesi

```
<security:jdbc-user-service    id="userService"  
                               data-source-ref="dataSource"  
  
    users-by-username-query="select username,password,enabled  
                             from users where username = ?"  
  
    authorities-by-username-query="select username,authority  
                                   from authorities where username = ?"  
  
    group-authorities-by-username-query="select g.id,  
g.group_name, ga.authority  
                                         from groups g, group_members gm, group_authorities ga  
                                         where gm.username = ? and g.id = ga.group_id and g.id  
= gm.group_id"/>
```

Custom UserDetails ve UserDetailsService

- Uygulamalar **kendilerine ait bir User sınıfını** kullanmak isteyebilir
- Ya da Authentication bilgisinin bir kısmı DB'den diğer bir kısmı LDAP veya başka bir realm'den gelebilir
- Bu gibi durumlarda **UserDetails ve UserDetailsService** arayüzlerinin implement edilmesi gerekecektir

Custom UserDetails ve UserDetailsService

```
public class CustomUser implements UserDetails {
    ...
}

@Service
@Transactional
public class CustomUserDetailsService implements UserDetailsService {

    @PersistenceContext
    private EntityManager entityManager;

    @Override
    public UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException {
        try {
            CustomUser user = entityManager.createQuery(
                "from User u where u.username = :username", CustomUser.class)
                .setParameter("username", username).getSingleResult();
            return user;
        } catch (Exception ex) {
            throw new UsernameNotFoundException(
                "User not found with username :" + username, ex);
        }
    }
}
```

Custom UserDetails ve UserDetailsService

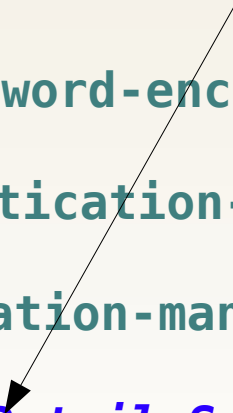
```
<beans>
  <security:authentication-manager>

    <security:authentication-provider
      user-service-ref="customUserDetailsService">

      <security:password-encoder hash="sha" />

    </security:authentication-provider>
  </security:authentication-manager>

  <bean id="customUserDetailsService"
    class="x.y.z.CustomUserDetailsService">
    ...
  </bean>
</beans>
```



Yetkilendirme (Authorization)

Yetkilendirme Nedir?

- Kimliklendirmesi yapılmış herhangi bir **principal**'ın (kullanıcı veya harici sistem) uygulama içerisindeki **nesneler üzerinde hangi işlemleri** gerçekleştirebileceğinin **kontrolüne yetkilendirme** denir
- Erişim denetimine tabi tutulan herhangi bir nesneye “**secure object**” veya “**secure resource**” adı verilir

Uygulandığı Resource Türleri

- Aşağıdakilerden herhangi birisi secure object olabilir
 - **Web istekleri:** web sayfalarına veya dosyalara erişim
 - **Metot çağrıları:** web servis metot çağrılarına veya servis katmanındaki nesnelerin metotlarına erişim
 - **Domain nesneleri:** veritabanındaki kayıtlara erişim, bu kayıtlar üzerinde herhangi bir işlem

Yetkisiz Erişim Örneği

1: HTTP request from User with ROLE_READER to editor.jsp

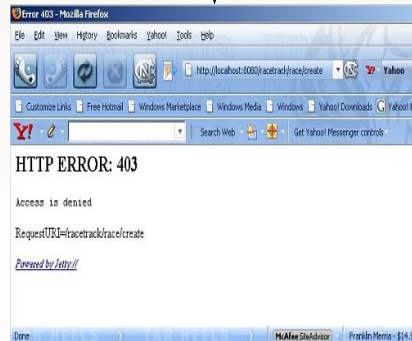
Throws **AccessDeniedException**

**FilterSecurity
Interceptor**

ROLE EDITOR



3: **ExceptionTranlationFilter**
Redirect to accessDenied page



accessDenied.jsp

**AccessDecision
Manager**

Tek tek Voter nesnelere kullanıcının editor.jsp'ye secure resource'una erişip erişemeyeceği sorulur, **ACCESS_DENIED** cevabı Alındığında **AccessDeniedException** fırlatılır

**AccessDecision
Voter**

Web Kaynaklarının Yetkilendirilmesi

- `<security:http>` elemanı içerisinde tanımlanan `<security:intercept-url/>` elemanları ile hangi web kaynağına hangi rollerin erişebileceği tanımlanır

```
<security:http>
```

```
<security:access-denied-handler error-page="/accessDenied.jsp"/>
```


```
<security:intercept-url pattern="/editor.jsp"  
                        access="hasRole('ROLE_EDITOR')" />
```

```
<security:intercept-url pattern="/accessDenied.jsp" access="permitAll"/>
```

```
<security:intercept-url pattern="/**" access="hasRole('ROLE_USER')"/>
```

```
</security:http>
```

Yetkisiz erişim durumunda
gösterilecek olan hata sayfası



Expression Tabanlı Yetkilendirme

- Access attribute içerisindeki yetki tanımlarında **Spring EL** kullanılabilir
- Spring Security 4 ile birlikte expression tabanlı yetkilendirme **default yöntem**dir

```
<security:http use-expressions="true">  
    <security:intercept-url pattern="/admin*" access="hasRole('ROLE_ADMIN') and  
        hasIpAddress('192.168.1.0/24')"/>  
</security:http>
```

Expression Tabanlı Yetkilendirme

- Web ve metot yetkilendirme için bazı **built-in tanımlı** fonksiyonlar:
 - `hasRole('ROLE_USER')`
 - `hasAnyRole('ROLE_USER','ROLE_EDITOR')`
 - `hasAuthority('ROLE_USER')`
 - `hasAnyAuthority('ROLE_USER','ROLE_EDITOR')`
 - `hasIpAddress('192.168.1.1')`
 - `isAnonymous()`, `isRememberMe()`, `isAuthenticated()`, `isFullyAuthenticated()`
- `permitAll`, `denyAll` gibi keyword'ler de kullanılabilir

Metot Erişiminin Yetkilendirilmesi

- Spring tarafından yönetilen bean'lerin metotları **invoke edilmeden önce veya sonra yetki kontrolü** yapılabilir
- Metot argümanlarının ve return değerlerinin **filtrelenmesi** de mümkündür (ACL)
- Devreye girmesi için aşağıdaki elemanın tanımlanması yeterlidir

```
<security:global-method-security  
    pre-post-annotations="enabled"/>
```

Pre-Post Annotasyonlar ile Metot Erişiminin Yetkilendirilmesi

- PreAuthorize, PostAuthorize, PreFilter ve PostFilter annotasyonları ile yetkilendirme yapılır
- Metoda erişimden önce yetkilendirme için **PreAuthorize** anotasyonu kullanılır

```
public class BusinessService {  
  
    @PreAuthorize("hasRole('ROLE_EDITOR')")  
    public void secureMethod() {  
        //...  
    }  
}
```

PreFilter, PostFilter ve PostAuthorize annotasyonları genellikle ACL tarafında ve metot return değerinin yetkilendirilmesinde kullanılır

Spring Security ve Entegrasyon Testleri

- Spring ApplicationContext'in de yer aldığı **entegrasyon birim testlerinde** servis metot çağrılarının yetkilendirilmeleri de kontrol edilebilir
- Bunun için öncelikle test metodu çalışmadan evvel kimliklendirme adımının gerçekleşmiş ve SecurityContext'de uygun rollere sahip **geçerli bir Authentication token**'in mevcut olması gerekir

Spring Security ve Entegrasyon Testleri

- Spring Security için herhangi bir biçimde **SecurityContext**'de geçerli bir **Authentication** token konması yeterlidir
- Bu nesnenin kim tarafından **nasıl** **konulduğu önemli değildir**
- SecurityContext'in içeriğinin uygulama içerisinde **doğrudan programatik olarak set edilmesi** mümkündür

Spring Security ve Entegrasyon Testleri

- **TestingAuthenticationToken** ile valid bir authentication token oluşturulabilir
- Bu token doğrudan **SecurityContext'e** set edilebilir
- Bu token'ın doğrulanması için sistemde **TestingAuthenticationProvider**'in authentication provider olarak tanıtılmış olması gerekir

Spring Security ve Entegrasyon Testleri

```
<beans>
```

```
<security:authentication-manager>
```

```
...
```

```
<security:authentication-provider  
  ref="testingAuthenticationProvider" />
```

```
</security:authentication-manager>
```

```
<bean id="testingAuthenticationProvider"  
  class="org.springframework.security.authentication.T  
  estingAuthenticationProvider" />
```

```
</beans>
```

AuthenticationManager
bean'ine Testing
AuthenticationProvider
da register edilmelidir

Spring Security ve Entegrasyon Testleri

```
Authentication auth = new TestingAuthenticationToken  
    ("user1", "secret", "ROLE_USER");
```

```
SecurityContextHolder.getContext().setAuthentication(auth);
```

...

```
businessService.secureMethod();
```

İletişim

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



harezmi
bilişim çözümleri

JAVA
Eğitimleri 