

- Seam, Java ile internet uygulamaları geliştirmek için ortaya çıkmış açık kaynak kodlu bir platformdur. Çıkış amacı Ajax, JSF, Java Persistence (JPA), Enterprise Java Beans (EJB 3.0) ve Business Process Management (jBPM) teknolojilerini bir araya getiren bir çözüm sunmaktır.
- Seam, aşağıdaki prensiplerden ilham almıştır:
 - **One kind of stuff:** Seam uygulamanızdaki tüm iş mantığı için bir bileşen modeli sunar. Bir seam bileşeni stateful ya da stateless olabilir.
 - **Integrate JSF with EJB 3.0:** JSF ve EJB 3.0 Java EE 5'in getirdiği en iyi iki özellik olmasına rağmen JEE 5 bu iki bileşeni entegre etmek için bir standart sunmamıştır. Seam, yazılımcının entegrasyonu düşünmeden sadece iş mantığına odaklanmasını sağlar.
 - **Integrated AJAX:** Seam en iyi iki açık kaynak kodlu JSF tabanlı AJAX çözümleri olan JBoss Richfaces ve IceFaces'i destekler.
 - **Business process as a first class construct:** İsteğe göre Seam, jBPM (Java Business Process Management) ile transparan bir iş süreci yönetimi sağlar. jBPM ve Seam ile karmaşık iş akışlarını ve görev yönetimini düzenlemek çok kolaydır.
 - **Declarative State Management:** Genellikle j2ee uygulamaları state yönetimini Servlet session'larını kullanarak kendileri implement ederler. Bu state yönetimi, session temizleme sırasında veya birden fazla pencere açılarak işlem yapılırken bir çok bug'a ve memory leak'e neden olur. Seam Hibernate veya JPA'da ortaya çıkan lazy ilişkilerdeki hataları (ör: LazyInitializationException) engeller.
 - **Bijection:** "Inversion of Control" veya "Dependency Injection" JSF'te ve EJB 3'te bulunan özelliklerdir. Bu container'ların çoğu stateless servisleri implement eden componentleri inject etmek için kullanılır. Stateful componentlerin injection'ı desteklendiğinde ise (ör: JSF) uygulama state'ini ele almak kullanışlı olmaz çünkü stateful bileşenin scope'u yeteri kadar esnek tanımlanamaz. Bijection'ın Inversion of Control'den farkı dinamik, esnek ve çift yönlü olmasıdır.
 - **Prefere annotations to XML:** JSF'te her bir management bean'i xml'de tanımlamak gerekir. Onlarca management bean olan bir uygulamada XML dosyası hayli büyük boyuta ulaşır. Seam ise tüm management bean tanımlarını ve konfigürasyonlarını annotation'larla yapmayı sağlar.
 - **There's more to a web application than serving HTML pages:** Günümüz web framework'leri çok basit düşünmektedirler. Kullanıcıdan girdiyi alıp bir POJO'ya bind ederler ve kalan işi size bırakırlar. Tam anlamıyla iyi bir web framework'ü Persistence, concurrency, asynchronicity, state management, security, email, messaging, PDF ve chart generation, workflow, wikitext rendering, webservisleri ve caching gibi bir çok problemi adreslemelidir.

SEAM KOMPONENTLERİ:

@Name("xxxxxx") =

Tüm seam komponentleri bir isme ihtiyaç duyar. Bu ismi vermezsek o komponentte hiçbir seam annotation'ını kullanamayız. İsmi hem JSF sayfalarında hem de Java kodunda kullanabiliriz.

Class'a verilen ismi belirler. Bu sayede context'te bu isimle çağırabiliriz. Kullanıcı arayüz'de atama ve tanımlama yaparken componentName yazılan yere ne yazılmışsa onu yazarak nesnelere veya metodlarına ulaşabiliriz. İstenildiği şekilde isimlendirilebilir fakat tek olmalıdır. Class ismiyle aynı olma zorunluluğu yoktur.

@Scope(SESSION) =

Nesnenin ne kadar süre context'te kalacağını yani yaşam süresini belirler.

@Scope (ScopeType . CONVERSATION)

Scope Türleri : EVENT, PAGE, CONVERSATION, SESSION, BUSINESS_PROCESS, APPLICATION, STATELESS.

Bir komponentin scope'unu (context) belirleme yaparız.

EVENT : Default değerdir. Tek bir istek boyunca yaşar. web request'inin sonunda yok olur

PAGE : sayfa açık kaldığı sürece yaşar, devam eder.

CONVERSATION : Birkaç sayfadaki bilgiler alınacaksa o sayfalar boyunca yaşar.

SESSION : Oturum kapanıncaya kadar yaşar (Login-Logout)

APPLICATION : uygulama kapanıncaya kadar yaşar.

STATELESS : State'i olmayan komponentlerdir ve nesne yönelime uygun komponentler değildir. Yine de seam uygulamasının önemli bir parçasını implement etmek için kullanılabilirler.

BUSINESS_PROCESS : Bu context uzun süreli bir iş işleminin state'ini tutar. Bu state BPM motoru tarafından yönetilir. (JBoss jBPM engine) Bu işlem birden fazla kullanıcının paylaştığı state'ler için kullanılır.

FacesMessages =

FacesMessages komponenti kullanıcıya mesaj göstermek için kullanılan bir built-in komponent örneğidir

```
FacesMessages.getInstance().add("Welcome back, #{user.name}!");
```

@In = inject edilecek değeri belirler

@Out annotation'ı ise outject edilecek değeri belirler.

- Çoğu Seam uygulaması, JSF eylem dinleyicileri olarak session beans kullanır
- tam olarak bir JSF eylemimiz ve buna bağlı bir session bean yöntemimiz var.
- Bu durumda, eylemimizle ilişkili tüm states User BEAN tarafından tutulduğundan, stateless bir session bean kullanacağız .

```
@Entity
@Name("xxx")
@Scope(SESSION)
@Table(name="xxx")
@Stateless
@Logger = Seam @Logger açıklaması, bileşenin Log örneğini enjekte etmek için
kullanılır .
@Logger
private Log log;
```

@Local = session bean local interface ihtiyacı var.

MVC

Controller	View	Model
Faces Servlet	xhtml	Pojo

```
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>*.seam</url-pattern>
</servlet-mapping>
```

components.xml

Dosyayı, Seam'e JNDI'da EJB bileşenlerimizi nasıl bulacağını anlatmak için kullanacağız.

Java Naming Directory Interface

web.xml

uygulamanın sunum katmanı. web dağıtım tanımlayıcısıdır

faces-config.xml

Çoğu Seam uygulaması, sunum katmanı olarak JSF görünümlerini kullanır. Görünümlerimizi tanımlamak için Facelets kullanacağız, bu yüzden JSF'ye, Facelets'i şablon motoru olarak kullanmasını söylememiz gerekiyor

Bellekteki mesajların listesini server requests arasında ön belleğe almak istiyoruz, bu yüzden bunu stateful session bean yaparız

@Stateful

@Scope(Session)

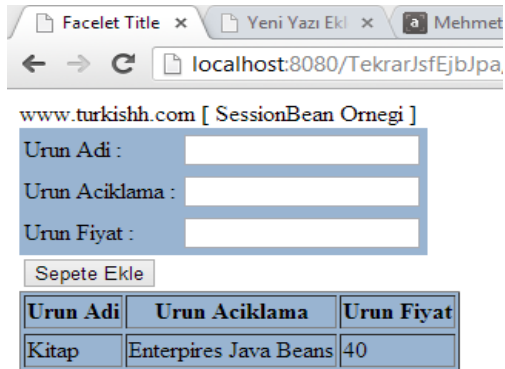
@DataModel

@DataModel anotasyonu Jsf sayfalarının bir instance örneği olan javax.faces.model.DataModel

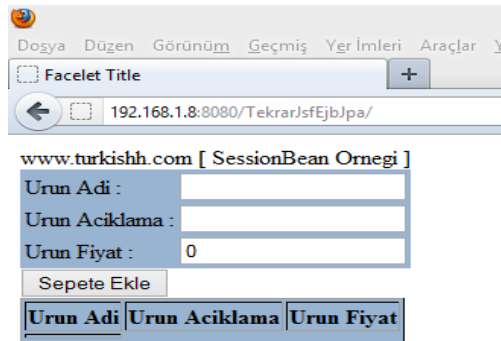
List tipinde bir attribute maruz kalır. Her link satırı için tıklanabilir JSF <h:dataTable> liste kullanmasına izin verir. Bu durumda DataModel adlandırılmış mevcut messagelist session context içerikleri yapılır

@Stateful

Simdi arkadaşlar bu ornekte notasyonumuz olarak @Stateful kullandik bu notasyonun nasıl davrandığını tarayicimiz üzerinden bakalım..İlk önce chrome ile acıyorum ve bir tane ürün ekliyorum ,ekledikten sonra mozillada acıyorum sayfayı ve şu şekilde görünüyor.

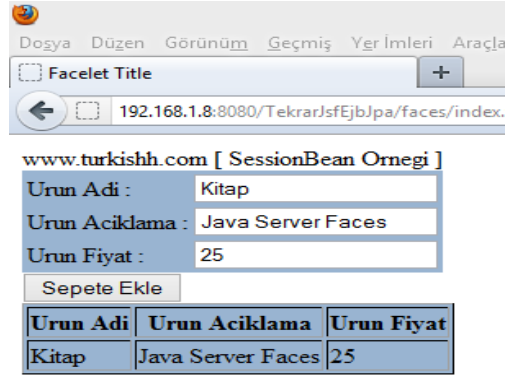
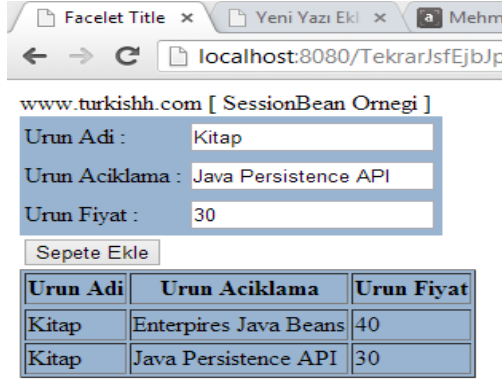


Urun Adi	Urun Aciklama	Urun Fiyat
Kitap	Enterpires Java Beans	40

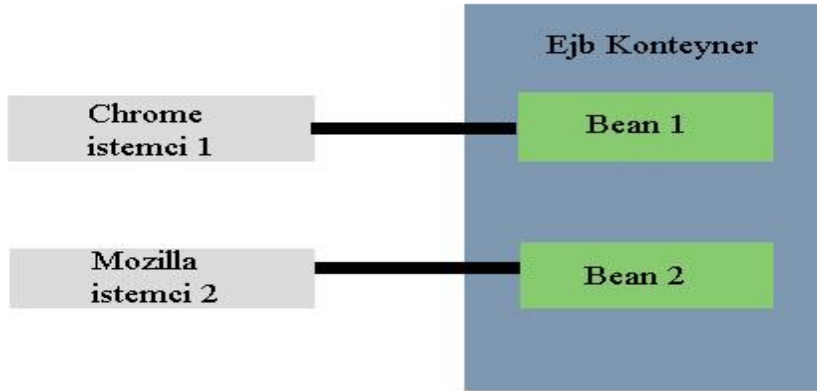


Urun Adi	Urun Aciklama	Urun Fiyat
Kitap	Enterpires Java Beans	40

Simdi 2 tarayiciyada ayrı ayrı birer tane ürün daha ekliyorum ve çıktısı şöyle oluyor..

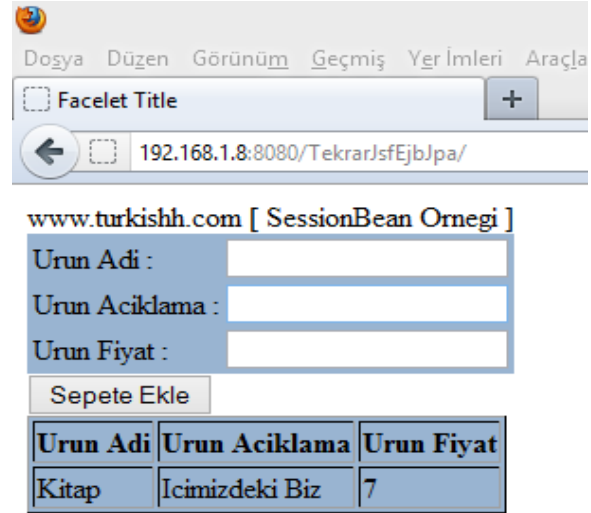


Goruldugu gibi her istemciye ayri ayri bean olusturuluyor . Stateful notasyonunu daha iyi anlayabilmeniz icin resimde gostermeye calisalim ..

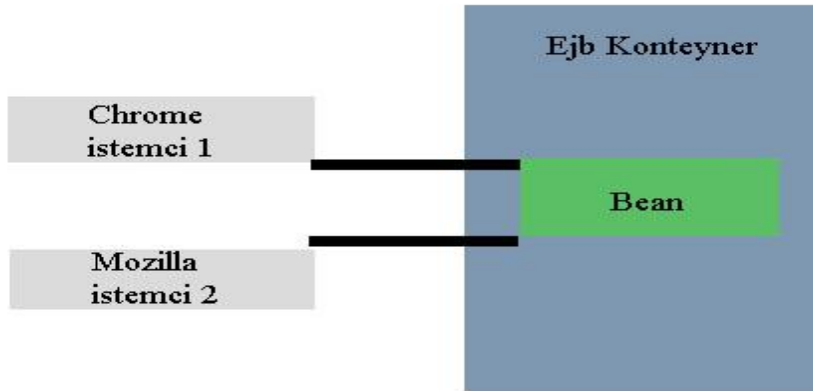


@Singleton

Bir baska notasyonumuz @Singleton nasil davrandigina deyinecek olursak singleton notasyonu bir tane sabit bean tutar ve her istemciye ayni bean yollar .. Simdi kodumuza geri donerek @Stateful notasyonunu @Singleton olarak degistirelim ve tarayicilarimizda nasil farklılıklar var görelim .Ayni islemi tekrarlıyalim chrome ile bir tane kitap ismi ekleyelim ve mozilla ile eklemiyelim bakalim ne olacak..



Bu sefer isler degisti 😊 mozilladan urun eklemedigim halde bana ejb konteyner ayni bean'i gonderdi .. Yani durum assagidaki resimde anlattigim gibi oldu ..



@Stateless

Session Bean ihtiyaca yönelik yazılmış ve özel ifadeler kullanan Java sınıflarıdır. Kendi içerisinde "stateless" ve "stateful" olmak üzere iki alt başlığa ayrılır. Stateless Bean, istemci hakkında bilgi tutmaz. Stateful Bean ise tam tersi istemci hakkında bilgi tutar ve gerektiğinde istemcinin bir kopyası sunucu üzerinde tutulur.

Stateless Bean iki parçadan meydana gelir. İlki istemcinin görmesini istediğimiz metotları içeren

bir arayüz, ikinci parça ise bu arayüzü uygulayan ve dolayısıyla metotlarını "override" eden sınıf. EJB'de mantık biraz farklıdır. İstemci kodlara direkt sınıf üzerinden değil de, arayüz üzerinden ulaşır. Yani aslında istemci kodlamada arayüzü görür.

Stateful Session Bean,

- Client ile bean arasındaki ilişkide sadece o "client"a özgü bilgi akışı varsa,
- Bean'in metot çağrıları sırasında istemciye ait bilgileri taşıması gerekiyorsa,
- Bean, istemci ile diğer sunucu bileşenleri arasında bir aracı görevi ile kullanılıyorsa kullanılabilir.

Stateless Session Bean,

- Bean, istemciye ait bilgi taşımıyorsa,
- Bean'deki metotların uyarımıyla tüm istemcilere sağlanan jenerik bir hizmet veriliyorsa (örneğin online siparişin tamamlandığını belirten bir mail.)
- Bean bir web servisi gerçekleştiriyorsa kullanılabilir.

@Stateless: Arayüzü uygulayan ve metotları "override" eden sınıfın başına bu tag eklenir. Böylece sunucu otomatik olarak bunun bir stateless EJB olduğunu algılar.

@Local: Metotların tanımlandığı arayüzün giriş kısmına eklenir.

@Remote: Metotların tanımlandığı arayüzün giriş kısmına eklenir.

Yukarıdaki iki "tag"da arayüzlerin üzerine uygulanır. Aralarındaki fark istemci ve EJB'nin çalıştığı JVM'lerden kaynaklanır. Eğer istemci ile EJB aynı JVM üzerinde çalışıyorsa "@Local" ile tanımlı arayüz kullanır, farklı JVM üzerinde çalışıyorlarsa "@Remote" ile tanımlı arayüz kullanılır. Farklı JVM, aynı bilgisayar üzerinde de bulunabilir. Yani bir bilgisayardaki farklı iki JVM aynı ortam kabul edilemez. Bunlardan başka iki geri çağırımı "tag"ımız daha var. Bunlar otomatik olarak tetiklenir.

Component names

```
@Name("loginAction")
```

```
@Stateless
```

```
public class LoginAction implements Login {
```

```
...
```

```
<h:commandButton type="submit" value="Login"
```

```
action="#{com.jboss.myapp.loginAction.login}"/>
```

Defining the component scope

```
@Name("user")

@Entity

@Scope(SESSION)

public class User {

    ...

}
```

@Scope annotation kullanarak bir bileşenin varsayılan kapsamını (bağlamını) geçersiz kılabiliriz .

Components with multiple roles

Bazı Seam component sınıfları, sistemde birden fazla rolü yerine getirebilir. Örneğin User, genellikle mevcut kullanıcıyı temsil eden session-scoped component bir bileşen olarak kullanılan ancak kullanıcı yönetim ekranlarında conversation-scoped component olarak kullanılan bir sınıfa sahibiz

@Role annotation bize farklı bir scope ile, bir bileşen için ek named rol tanımlamanıza izin verir - bu bizi bağlamak different scope değişkenleri aynı component sınıfı sağlar. Herhangi bir Seam component *örneği*, birden çok context değişkenine bağlı olabilir, ancak bu, bunu sınıf düzeyinde yapmamıza ve otomatik örneklemeden yararlanmamıza olanak tanır

```
@Entity

@Scope(CONVERSATION)

@Role(name="currentUser", scope=SESSION)

public class User {

    ...

}
```

@Roles annotation istediğimiz kadar bize birçok ek rolleri olarak belirtmenizi sağlar.


```
@Entity
@Scope(CONVERSATION)
@Roles({@Role(name="currentUser", scope=SESSION),
        @Role(name="tempUser", scope=EVENT)})
public class User {
    ...
}
```

Bijection

```
@Name("loginAction")
@Stateless
public class LoginAction implements Login {
    @In User user;
    ...
}
```

@In Bir değer ya da bir örnek değişkene, enjekte edilmesi gerektiğini belirtir. Context'te inject edilen nesne var ise onun bir instance'ını getirir. Eğer yoksa null döndürür.

@In(create=true)

Inject edilen nesne null olduğunda hata verir. Eer create=true denilirse SEAM o nesnenin bir instance'ını oluşturur.

```

@Name("loginAction")

@Stateless

public class LoginAction implements Login {

    User user;

    @In

    public void setUser(User user) {

        this.user=user;

    }

    ...

}

```

Bijection Inversion of Control nazaran contextli, çift yönlü ve dinamiktir. @In annotation'ı inject edilecek değeri belirler. @Out annotation'ı ise outject edilecek değeri belirler.

```

private Person person;
private List <Person> fans;

@In
public void setPerson (Person person) {
    this.person = person;
}
@Out
public Person getPerson () {
    return person;
}
@Out
public List <Person> getFans () {
    return fans;
}

```

in = setter ; out = getter mantığı gibi düşünülebilir

Başına @Out annotation'ı konulan nesne context'te gönderilir ve başka bir yerden aynı nesne @In ile inject edilebilir.

@PostConstruct: Bu "tag"ı sınıfta tanımlı herhangi bir metot üzerine koyabiliriz. Böylece EJB'nin bir örneği oluştuktan ve varsa gerekli bağımlılıklar ayarlandıktan sonra bu metot otomatik olarak çağrılır ve içeriği işletilir.

@PreDestroy: @PostConstruct ile uygulaması aynıdır; yalnız EJB'nin örneği sunucudan kaldırılmadan önce işletilir.

```
private Vector kalanOgrenciler;

@PostConstruct
private void listeyiOlustur() {
    kalanOgrenciler = new Vector();
}

@PreDestroy
private void listeyiSil() {
    if(kalanOgrenciler != null){
        kalanOgrenciler.clear();
    }
}
```

@Name("messageSender")

@Install(precedence=MOCK)

public class MockMessageSender **extends** MessageSender {

public void sendMessage() {

 //do nothing!

 }

}

Öncelik, Seam'in Class her iki bileşeni de bulduğunda hangi versiyonu kullanacağına karar vermesine yardımcı olur. Başka hangi bileşenlerin kurulu olduğuna ve sınıf yolunda hangi sınıfların mevcut olduğuna bağlı olarak hangi bileşenlerin kurulacağına karar verebilmek istiyorum. @Install annotation da bu işlevi kontrol eder.

```
@Logger private Log log;
```

```
public Order createOrder(User user, Product product, int quantity) {
```

```
    log.debug("Creating new order for user: #0 product: #1 quantity: #2", user.userName(), product.name(), quantity);
```

```
    return new Order(user, product, quantity);
```

```
}
```

Seam, bu kodu önemli ölçüde basitleştiren bir Log kaydı API'si sağlar. Ayrıca Seam, Log'u hangi component enjekte ettiğini bildiğinden, genellikle log kategorisini açıkça belirtmemize gerek yoktur.

```
@Factory(scope=CONVERSATION)
```

```
public List<Customer> getCustomerList() {
```

```
    return ... ;
```

```
}
```

Bir factory method, named context variable hiçbir değer bağlı olmadığında çağrılır ve context variable değerini başlatması beklenir. Bir yöntemi, bağlam değişkeni için fabrika yöntemi olarak işaretler.

Metodlar üzerinde kullanılır. İlgili componentten bağımsız olarak kullanılabilir. Yani herhangi bir yerden verilen isim ile çağrılabilir.

İki tür fabrika yöntemi vardır.

Void dönüş türüne sahip fabrika yöntemleri, bağlam değişkenine bir değer atamaktan sorumludur.

```
@DataModel List<Customer> customerList;

@Factory("customerList")
public void initCustomerList() {
    customerList = ... ;
}
```

Seam döndürülen değeri belirtilen scope bağlayacağından, bir değer döndüren factory method değeri açıkça belirtmesi gerekmez.

```
@Factory(scope=CONVERSATION)
public List<Customer> getCustomerList() {
    return ... ;
}
```

Fabrika yöntemi, bağlam değişkenine bir değer bağlar ve bağlı değerın kapsamını belirler.

Herhangi bir kapsam açıkça belirtilmezse, @Factory yöntemiyle bileşenin kapsamı kullanılır (bileşen durumsuz olmadığı sürece, bu durumda EVENT bağlamı kullanılır).

autoCreate - @In, create = true değerini belirtmese bile, **değişken istendiğinde bu fabrika yönteminin otomatik olarak çağırılması** gerektiğini belirtir.

Burada yönetilen bileşen, temeldeki nesneyi değiştiren birçok olayı gözlemler. Bileşen bu eylemleri kendisi yönetir ve nesne her erişimde sarmalanmadığı için tutarlı bir görünüm sağlar.

```
@Observer({"chickBorn", "chickenBoughtAtMarket"})
```

```
public addHen()
```

```
{
```

```
    hens.add(hen);
```

```
}
```

```
@Observer("chickenSoldAtMarket")
```

```
public removeHen()
```

```
{
```

```
    hens.remove(hen);
```

```
}
```

```
@Observer("foxGetsIn")
```

```
public removeAllHens()
```

```
{
```

```
    hens.clear();
```

```
}
```

jBPM

jBPM (Java Business Process Management) ile transparan bir iş süreci yönetimi sağlar. JBPM ve Seam ile karmaşık iş akışlarını ve görev yönetimini düzenlemek çok kolaydır.

Herhangi bir java programında gömülü olarak veya tek başına bir sunucu olarak çalıştırılabilen esnek ve geliştirilebilir bir süreç motorudur.

JPMN, çoklu süreçleri destekleyen PVM (Process Virtual Machine)Tabanlıdır.

aşağıdaki components.xml dosyası jBPM'yi yükler:

```
<components xmlns="http://jboss.org/schema/seam/components"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xmlns:bpm="http://jboss.org/schema/seam/bpm">
    <bpm:jbpm/>
</components>
```

Yada

```
<components>
    <component class="org.jboss.seam.bpm.Jbpm"/>
</components>
```

Events, interceptors and exception handling

Seam events

JSF **EL yöntemi** bağlama ifadeleri aracılığıyla Seam bileşenlerine eşlenir. Bir JSF olayı için bu, JSF şablonunda tanımlanır:

```
<h:commandButton value="Click me!" action="#{helloWorld.sayHello}"/>
```

Page actions

Bir Seam sayfa eylemi, bir sayfayı oluşturmada hemen önce gerçekleşen bir olaydır. WEB-INF'de sayfa actions açıklıyoruz

```
<pages>
  <page view-id="/hello.xhtml" action="#{helloWorld.sayHello}"/>
</pages>
```

Or

```
<pages>
  <page view-id="/hello/*" action="#{helloWorld.sayHello}"/>
</pages>
```

or

```
<pages>

  <page view-id="/hello.xhtml">
    <action execute="#{helloWorld.sayHello}" if="#{not validation.failed}"/>
    <action execute="#{hitCount.increment}"/>
  </page>

</pages>
```

Sizi JSF API'sinin ayrıntı düzeyinden kurtarmak için Seam, aynı sonucu çok daha az yazarak elde etmenizi sağlayan yerleşik bir koşul sunar. Geri göndermede bir sayfa eylemini yalnızca geri göndermeyi yanlış olarak ayarlayarak devre dışı bırakabilir. Geriye dönük uyumluluk nedenleriyle, `postback` özneliğinin varsayılan değeri `true`, ancak bunun tersi ayarı daha sık kullanacaksınız.

```
<pages>
  <page view-id="/dashboard.xhtml">
    <action execute="#{dashboard.loadData}" on-postback="false"/>
  </page>
</pages>
```

Mapping request parameters to the model

Görünüm kimliği için yüz olmayan (GET) bir istek gerçekleştiğinde, Seam, adlandırılmış istek parametresinin değerini, uygun tür dönüştürmeleri gerçekleştirdikten sonra model nesnesine ayarlar.


```
<pages>

  <page view-id="/hello.xhtml" action="#{helloWorld.sayHello}">
    <param name="firstName" value="#{person.firstName}"/>
    <param name="lastName" value="#{person.lastName}"/>
  </page>

</pages>
```

Yalnızca name özniteliği belirtilirse, istek parametresi PAGE context kullanılarak yayılır (model özelliğine eşlenmez).

```
<pages>

  <page view-id="/hello.xhtml" action="#{helloWorld.sayHello}">
    <param name="firstName" />
    <param name="lastName" />
  </page>

</pages>
```

URL rewriting with page parameters

Bu durumda, / home için gelen herhangi bir istek /home.xhtml adresine gönderilecektir. Daha da ilginç, normalde /home.seam'e işaret eden herhangi bir bağlantı, bunun yerine / home olarak yeniden yazılacaktır. Yeniden yazma modelleri, URL'nin yalnızca sorgu parametrelerinden önceki kısmıyla eşleşir. Dolayısıyla, /home.seam?conversationId=13 ve /home.seam?color=red, bu yeniden yazma kuralı tarafından eşleştirilecektir.

Yeniden yazma kuralları, aşağıdaki kurallarda gösterildiği gibi bu sorgu parametrelerini dikkate alabilir.

```
<page view-id="/home.xhtml">

  <rewrite pattern="/home/{color}" />
  <rewrite pattern="/home" />

</page>
```

Bu durumda, / home / red için gelen bir istek, /home.seam?color=red için bir istekmiş gibi sunulacaktır. Benzer şekilde, renk bir sayfa parametresiye, normalde /home.seam?color=blue olarak gösterilen bir giden URL, bunun yerine / home / blue olarak çıkarılır. Kurallar sırayla işlenir, bu nedenle daha genel kurallardan önce daha özel kuralları listelemek önemlidir.

Varsayılan Seam sorgu parametreleri, Seam'in parmak izlerini gizlemek için başka bir seçeneğe izin vererek URL yeniden yazma kullanılarak da eşlenebilir. Aşağıdaki örnekte /search.seam?conversationId=13, / search-13 olarak yazılır.

```
<page view-id="/search.xhtml">  
  <rewrite pattern="/search-{conversationId}" />  
  <rewrite pattern="/search" />  
</page>
```