## Web Beans: Contesti Java e Dependency Injection

# Il nuovo standard Java per la dependency injection e la gestione contestuale dello stato

**Gavin King**
**JSR-299 specification lead**
**Red Hat Middleware LLC**

**Pete Muir**
**Web Beans (JSR-299 Reference Implementation) lead**
**Red Hat Middleware LLC**

**David Allen**
**Traduzione italiana: Nicola Benaglia, Francesco Milesi**
**Traduzione spagnola: Gladys Guerrero**
**Red Hat Middleware LLC**
**Traduzione coreana: Eun-Ju Ki,**

Red Hat Middleware LLC
Traduzione cinese (tradizionale): Terry Chuang
Red Hat Middleware LLC
Traduzione cinese semplificato: Sean Wu
Kava Community

## Nota

JSR-299 ha recentemente cambiato il suo nome da "Web Beans" a "Contesti Java e Dependency Injection". La guida fa comunque riferimento alla JSR-299 come "Web Beans" e alla JSR-299 Reference Implementation come "Web Beans RI". Altre documentazioni, blogs, forum, ecc. potrebbero usare la nuova nomenclatura, includendo il nuovo nome per la JSR-299 Reference Implementation - "Web Beans".

Si vedrÃ  che alcune delle piÃ¹ recenti funzionalitÃ  da specificare mancano (come campi produttori, realizzazione, eventi asincroni, mappatura XML delle risorse EE).

# Parte I. Usare gli oggetti contestuali

La specifica Web Beans (JSR-299) definisce un insieme di servizi per l'ambiente Java EE che rende molto più facile lo sviluppo di applicazioni. Web Beans sovrappone un più ricco modello di interazione e di gestione del ciclo di vita ai tipi di componenti java esistenti, Java Beans and Enterprise Java Beans inclusi. A complemento del tradizionale modello di programmazione Java EE, i servizi Web Beans forniscono:

- una migliore gestione del ciclo di vita dei componenti stateful, associata a *contesti* ben definiti
- un approccio typesafe alla *dependency injection*,
- interazioni basate su una struttura per *la notifica degli eventi*, e
- un migliore approccio nell'associazione degli *interceptors* ai componenti, unitamente all'introduzione di un nuovo tipo di interceptor, chiamato *decoratore*, più adatto ad essere utilizzato nella soluzione di problemi legati alla business logic.

La dependency injection, insieme alla gestione contestuale del ciclo di vita dei componenti, risparmia a chi utilizza un API con cui non ha familiarità la necessità di dover formulare le risposte relative alle seguenti domande:

- qual è il ciclo di vita di questo oggetto?
- quanti client può simultaneamente avere?
- è multithreaded?
- da dove posso ottenerne uno?
- devo distruggerlo esplicitamente?
- dove dovrei tenerne il riferimento quando non lo sto usando direttamente?
- come posso aggiungere un livello di indirezione, in modo che l'implementazione di tale oggetto possa variare a deployment time?
- cosa dovrei fare per condividere questo oggetto con altri oggetti?

Un Web Bean specifica soltanto il tipo e la semantica degli altri Web Beans da cui dipende. Non ha bisogno di essere a conoscenza del reale ciclo di vita, della implementazione, del modello di threading o degli altri client dei Web Bean da cui dipende. Ancor meglio, l'implementazione, il ciclo di vita e il modello di threading di un Web Bean da cui dipende possono variare a seconda dello scenario di deployment, senza avere effetti su nessun client.

Eventi, interceptor e decoratori potenziano l'*accoppiamento debole* (loose-coupling) inerente a questo modello:

- le *notifiche degli eventi* disaccoppiano i produttori di eventi dai consumatori,
- gli *interceptor* disaccoppiano i problemi tecnici dalla business logic, e
- i *decoratori* permettono di compartimentare i problemi di business logic.

Soprattutto, Web Beans fornisce tutti questi strumenti in un modo *typesafe*. Web Beans non usa mai identificatori di tipo stringa per determinare come interagiscono oggetti che collaborano fra di loro. Sebbene resti un'opzione, il linguaggio XML è usato raramente. Invece Web Beans utilizza l'informazione di tipo già presente nel modello a oggetti di Java, insieme ad un nuovo pattern, chiamato *binding annotations*, per assemblare i Web Beans, le loro dipendenze, i loro interceptor e decoratori e i loro consumatori di eventi.

I servizi di Web Beans sono generali e applicabili ai seguenti tipi di componenti che esistono in ambiente Java EE:

- tutti i JavaBeans,
- tutti gli EJB, e
- tutti i Servlets.

Web Beans fornisce anche i necessari punti di integrazione in modo che altri tipi di componenti definiti da future specifiche Java EE o da framework non standard possano essere integrati in modo trasparente con Web Beans, avvantaggiarsi dei suoi servizi, e interagire con qualunque altro tipo di Web Bean.

Web Beans è stata influenzata da un buon numero di framework Java esistenti, inclusi Seam, Guice and Spring. Comunque, Web Beans ha un proprio chiaro carattere distintivo: è più sicuro nell'uso dei tipi (typesafe) di Seam, è più orientato allo stato (stateful) e meno basato su XML di Spring, e più capace di Guice nelle applicazioni web ed enterprise.

Soprattutto, Web Beans è uno standard JCP che si integra in modo trasparente con Java EE, e con qualunque ambiente Java SE dove EJB Lite sia disponibile in modo embeddable.

# Iniziare con Web Beans

Non vedi l'ora di iniziare a scrivere il primo Web Beans? O forse sei un pò scettico e ti domandi quali virtuosismi ti farà fare la specifica Web Beans! La buona notizia è che probabilmente hai già scritto e usato centinaia, forse migliaia di Web Beans. Potresti addirittura non ricordare il primo Web Bean scritto.

## 1.1. Il primo Web Bean

Con alcune eccezioni molto particolari, ogni classe Java con un costruttore che non accetta parametri è un Web Bean. Questo include ogni JavaBean. Inoltre, ogni session bean di stile EJB3 è un Web Bean. Sicuramente i JavaBean e gli EJB3 che si sono sempre scritti non erano in grado di sfruttare i nuovi servizi definiti dalla specifica Web Beans, ma si sarà presto in grado di usare ciascuno di essi come Web Bean # iniettandoli in altri Web Beans, configurandoli tramite strumenti di configurazione XML Web Bean, e perfino aggiungendo a loro interceptor e decoratori # senza toccare il codice esistente.

Si supponga di avere due classi Java esistenti, usate da anni in varie applicazioni. La prima classe esegue il parsing di una stringa in una lista di frasi:

```
public class SentenceParser {
   public List<String
> parse(String text) { ... }
}
```

La seconda classe è un session bean stateless front-end per un sistema esterno capace di tradurre le frasi da una lingua ad un altra:

```
@Stateless
public class SentenceTranslator implements Translator {
   public String translate(String sentence) { ... }
}
```

Dove `Translator` è l'interfaccia locale:

```
@Local
public interface Translator {
   public String translate(String sentence);
}
```

Sfortunatamente non ci sono classi preesistenti che traducano l'intero testo dei documenti. Quindi occorre scrivere un Web Bean che faccia questo lavoro:

```
public class TextTranslator {

   private SentenceParser sentenceParser;
   private Translator sentenceTranslator;

   @Initializer
   TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator) {
      this.sentenceParser = sentenceParser;
      this.sentenceTranslator = sentenceTranslator;
   }

   public String translate(String text) {
      StringBuilder sb = new StringBuilder();
      for (String sentence: sentenceParser.parse(text)) {
         sb.append(sentenceTranslator.translate(sentence));
      }
      return sb.toString();
   }

}
```

Si può ottenere un'istanza di `TextTranslator` iniettandolo in un Web Bean, Servlet o EJB:

```
@Initializer
public setTextTranslator(TextTranslator textTranslator) {
   this.textTranslator = textTranslator;
}
```

In alternativa si può ottenere un'istanza chiamando direttamente un metodo del manager Web Bean:

```
TextTranslator tt = manager.getInstanceByType(TextTranslator.class);
```

Ma `TextTranslator` non ha un costruttore con nessun parametro! E' ancora un Web Bean? Una classe che non ha un costruttore senza parametri può essere un Web Bean se il suo costruttore è annotato con `@Initializer`.

Come hai indovinato, l'annotazione `@Initializer` ha qualcosa che fare con la dependency injection! `@Initializer` può essere applicato ad un costruttore od un metodo di un Web Bean, e dice al manager Web Bean di chiamare quel costruttore o metodo quando si istanzia il Web Bean. Il manager Web Bean inietterà altri Web Bean nei parametri del costruttore o del metodo.

In fase di inizializzazione del sistema, il manager Web Bean deve convalidare che esattamente un solo Web Bean esista e soddisfi ciascun punto di iniezione. Nell'esempio, se nessuna implementazione di `Translator` fosse disponibile # se l'EJB `SentenceTranslator` non venisse deployato # il manager Web Bean lancerebbe una `UnsatisfiedDependencyException`. Se più di un'implementazione di `Translator` fosse disponibile, il manager Web Bean lancerebbe una `AmbiguousDependencyException`.

## 1.2. Cosa è un Web Bean?

Ma cosa è *esattamente* un Web Bean?

Un Web Bean è una classe di un'applicazione che contiene della logica di business. Può essere chiamato direttamente da codice Java, o può essere invocato via Unified EL. Un Web Bean può accedere a risorse transazionali. Le dipendenze tra Web Beans sono gestite automaticamente dal manager Web Bean. La maggior parte dei Web Beans sono *stateful* e *contestuali*. Il ciclo di vita di un Web Bean è sempre gestito da un manager Web Bean.

Torniamo indietro un attimo. Cosa significa veramente essere "contestuale"? Poiché Web Beans può essere stateful, è importante *quale* istanza di bean si ha. Diversamente da un modello a componenti stateless (per esempio, i session bean stateless) o un modello a componenti singleton (come i servlet o i bean singleton) i client di un Web Bean vedono il Web Bean in stati differenti. Lo stato del client visibile dipende dall'istanza del Web Bean alla quale il client ha il riferimento.

Comunque, in modo simile ad un modello stateless o singleton, ma *non come* i session bean stateful, il client non ha il controllo sul ciclo di vita dell'istanza, creandola e distruggendola esplicitamente. Invece, lo *scope* del Web Bean determina:

- il ciclo di vita di ogni istanza del Web Bean e
- quali client condividono una referenza con una particolare istanza del Web Bean.

Per un dato thread in un'applicazione Web Beans, ci può essere un *contesto attivo* associato allo scope del Web Bean. Questo contesto può essere univoco nel thread (per esempio, se il Web Bean è con scope di richiesta), o può essere condiviso con alcuni altri thread (per esempio, se il Web Bean è con scope di sessione) od anche tutti gli altri thread (se è scope di applicazione).

I client (per esempio, altri Web Beans) che sono in esecuzione nello stesso contesto vedranno la stessa istanza del Web Bean. Ma i client in un contesto diverso vedranno un istanza diversa.

Un grande vantaggio del modello contestuale è che consente ai Web Beans stateful di essere trattati come servizi! Il client non deve preoccuparsi di gestire il ciclo di vita del Web Bean che sta utilizzando, *e neppure deve sapere quale sia il ciclo di vita.* Web Beans interagisce passando

i messaggi, e le implementazioni Web Bean definiscono il ciclo di vita del proprio stato. I Web Beans sono debolmente disaccoppiati (loosely coupled) poiché:

- interagiscono tramite delle API pubblica ben-definita
- il loro ciclo di vita è completamente disaccoppiato

Si può sostituire un Web Bean con un diverso Web Bean che implementa la stessa API ed ha un diverso ciclo di vita (un diverso scope) senza alterare l'implementazione dell'altro Web Bean. Infatti Web Beans definisce un meccanismo sofisticato per fare l'override delle implementazioni Web Bean al momento del deploy, come visto in *Sezione 4.2, «Tipi di deploy»*.

Si noti che non tutti i client dei un Web Bean sono Web Bean. Altri oggetti come Servlet o Message-Driven Beans # che sono per natura non iniettabili, oggetti contestuali # possono pure ottenere riferimenti a Web Bean tramite iniezione.

Più formalmente, secondo la specifica:

> Un Web Bean comprende:
>
> - Un set (non vuoto) di tipi di API
> - Un set (non vuoto) di tipi di annotazione di binding
> - Uno scope
> - Un tipo di deploy
> - Opzionalmente un nome Web Bean
> - Un set di tipi di interceptor binding
> - Un implementazione Web Bean

Vediamo cosa significano alcuni di questi termini per lo sviluppatore Web Bean.

### 1.2.1. Tipi di API, tipi di binding e dependency injection

I Web Bean solitamente acquisiscono riferimenti ad altri Web Bean tramite la dependency injection. Ogni attributo iniettato specifica un "contratto" che deve essere soddisfatto dal Web Bean per essere iniettato. Il contratto è:

- un tipo di API, assieme a
- un set di tipi di binding

Un API è una classe o interfaccia definita dall'utente. (Se il Web Bean è un session bean EJB, il tipo di API è l'interfaccia `@Local` o la vista locale della classe-bean). Un tipo di binding rappresenta un semantica del client che è soddisfatta da certe implementazioni dell'API e non da altre.

I tipi di binding sono rappresentati da annotazioni definite dall'utente che sono loro stesse annotate con `@BindingType`. Per esempio, il seguente punto di iniezione ha un tipo di API `PaymentProcessor` ed un tipo di binding `@CreditCard`:

```
@CreditCard PaymentProcessor paymentProcessor
```

Se nessun tipo di binding viene specificato in modo esplicito ad un punto di iniezione, il tipo di binding di default si assume essere `@Current`.

Per ogni punto di iniezione, il manager Web Bean cerca un Web Bean che soddisfi il contratto (che implementi la API, e che abbia tutti i tipi di binding), ed inietta tale Web Bean.

Il seguente Web Bean ha il tipo binding `@CreditCard` e implementa il tipo API `PaymentProcessor`. Può quindi essere iniettato nel punto di iniezione d'esempio:

```
@CreditCard
public class CreditCardPaymentProcessor
   implements PaymentProcessor { ... }
```

Se un Web Bean non specifica esplicitamente un set di tipi di binding, ha esattamente un solo tipo di binding: il tipo di binding di default `@Current`.

Web Beans definisce un *algoritmo di risoluzione* sofisticato ma intuitivo che aiuta il container a decidere cosa fare se più di un Web Bean soddisfa un particolare contratto. Vedremo i dettagli in *Capitolo 4, Dependency injection*.

### 1.2.2. Tipi di deploy

*I tipi di deploy* consentono di classificare i Web Bean secondo uno scenario di deploy. Un tipo di deploy è un'annotazione che rappresenta un particolare scenario di deploy, per esempio `@Mock`, `@Staging` oppure `@AustralianTaxLaw`. Si applica l'annotazione ai Web Bean che dovrebbero essere deployati in tale scenario. Un tipo di deploy consente ad un intero set di Web Bean di essere deployati in modo condizionato, con una sola linea di configurazione.

Molti Web Bean usano soltanto il tipo di deploy di default `@Production`, ed in questo caso non occorre specificare esplicitamente nessun tipo di deploy. Tutti e tre i Web Bean d'esempio hanno ul tipo di deploy `@Production`.

In un ambiente di test è possibile sostituire il Web Bean `SentenceTranslator` con un "oggetto mock":

```
@Mock
public class MockSentenceTranslator implements Translator {
```

```
    public String translate(String sentence) {
        return "Lorem ipsum dolor sit amet";
    }
}
```

In ambiente di test si dovrebbe abilitare il tipo di deploy `@Mock` per indicare che l'uso di `MockSentenceTranslator` ed ogni altro Web Bean annotato con `@Mock`.

Si discuterà questa potente funzionalità con maggior dettaglio in *Sezione 4.2, «Tipi di deploy»*."

### 1.2.3. Scope

Lo *scope* definisce il ciclo di vita e la visibilità delle istanze di Web Bean. Il modello di contesto Web Bean è estensibile e facilita gli scope arbitrari. Comunque alcuni importanti scope sono predefiniti all'internodella specifica e vengono forniti dal manager Web Bean. Uno scope è rapresentato da un tipo di annotazione.

Per esempio un'applicazione web può avere Web Bean con *scope di sessione*

```
@SessionScoped
public class ShoppingCart { ... }
```

Un'istanza di un Web Bean con scope sessione è legato ad una sessione utente ed è condivisa da tutte le richieste che si eseguono nel contesto di tale sessione.

Di default i Web Bean appartengono ad uno speciale scope chiamato *pseudo-scope dipendente*. Web Bean con questo scope sono oggetti puri dipendenti dall'oggetto nel quale vengono iniettati ed il loro ciclo di vita è legato al ciclo di vita di tale oggetto.

Approfondiremo gli scope in *Capitolo 5, Scope e contesti*.

### 1.2.4. Nomi Web Bean e Unified EL

Un Web Bean può avere un *nome* che gli consente di essere usato in un'espressione Unified EL. E' facile specificare il nome del Web Bean:

```
@SessionScoped @Named("cart")
public class ShoppingCart { ... }
```

Ora si può facilmente utilizzare il Web Bean in ogni pagina JSF o JSP:

```
<h:dataTable value="#{cart.lineItems}" var="item">
    ....
```

```
</h:dataTable
>
```

Si può anche lasciare assegnare al manager Web Bean il nome di default:

```
@SessionScoped @Named
public class ShoppingCart { ... }
```

In questo caso il nome di default è `shoppingCart` # il nome della classe non qualificata, con il primo carattere messo in minuscolo.

### 1.2.5. Tipi di interceptor binding

Web Beans supporta la funzionalità di interceptor definita da EJB 3, non solo per i bean EJB, ma anche per classi Java semplici (plain). In aggiunta, Web Beans fornisce un nuovo approccio al binding di interceptor nei confronti di bean EJB e di altri Web Beans.

Rimane la possibilità di specificare direttamente la classe interceptor tramite l'uso dell'annotazione `@Interceptors`.

```
@SessionScoped
@Interceptors(TransactionInterceptor.class)
public class ShoppingCart { ... }
```

Comunque è più elegante ed è considerata una pratica migliore quella di giungere indirettamente ad un interceptor binding tramite un *tipo di interceptor binding*:

```
@SessionScoped @Transactional
public class ShoppingCart { ... }
```

Si discuteranno gli interceptor e i decoratori di Web BEans in *Capitolo 7, Gli interceptor* e *Capitolo 8, Decoratori*.

## 1.3. Quali tipi di oggetti possono essere Web Beans?

Si è già visto che JavaBeans, EJB ed altri tipi di classi Java possono essere Web Bean. Ma esattamente quali tipi di oggetti sono Web Beans?

### 1.3.1. Web Beans Semplici

La specifica Web Beans dice che una classe concreta Java è un Web Bean *semplice* se:

- Non è un componente gestito da un container EE, come EJB, un Servlet o un entity JPA,
- non è una classe interna statica/non statica,
- non è un tipo parametrizzato, e
- ha un costruttore senza parametro, o un costruttore annotato con `@Initializer`.

Quindi quasi ogni JavaBean è un Web Bean semplice.

Ogni interfaccia implementata direttamente o indirettamente da un Web Bean semplice è un tipo di API di un Web Bean semplice. La classe e le sue superclassi sono anch'essere tipi di API.

### 1.3.2. Web Bean Enterprise

La specifica dice che tutti i bean di sessione stile EJB3 e quelli singleton sono Web Bean *enterprise*. I bean message driven non sono Web Bean # poiché non sono intesi per essere iniettati in altri oggetti # ma possono sfruttare la maggior parte della funzionalità dei Web Bean, inclusi dependency injection e interceptor.

Ogni interfaccia locale di un Web Bean enterprise che non ha un parametro tipo wildcard o variabile tipo, e ciascuna delle sue superinterfacce, è un tipo di API del Web Bean enterprise. Se il bean EJB ha una vista locale di classe bean, la classe bean e ogni sua superclasse è anch'essa un tipo di API.

I session bean stateful dovrebbero dichiarare un metodo remoto senza parametri od un metodo annotato con `@Destructor`. Il manager Web Bean chiama questo metodo per distruggere l'istanza del session bean statefull alla fine del suo ciclo di vita. Questo metodo è chiamato metodo *distruttore* del Web Bean enterprise.

```
@Stateful @SessionScoped
public class ShoppingCart {

   ...

   @Remove
   public void destroy() {}

}
```

Ma allora quando occorre usare un Web Bean enterprise invece di un Web Bean semplice? Quando occorrono servizi enterprise avanzati offerti da EJB, quali:

- gestione delle transazioni a livello di metodo e sicurezza,
- gestione della concorrenza,

- passivazione a livello di istanza per session bean stateful e pooling di istanze per session bean stateless,
- invocazione remota e web service, e
- timer e metodi asincroni,

si dovrebbe usare un Web Bean enterprise. Quando non occorrono queste cose, va bene utilizzare un Web Bean semplice.

Molti Web Bean (inclusi Web Bean con scope di sessione o applicazione) sono disponibili per accessi concorrenti. Quindi la gestione della concorrenza fornita da EJB3.1 è molto utile. La maggior parte dei Web Bean con scope sessione e applicazione dovrebbero essere EJB.

Web Bean che mantengono riferimenti alle risorse pesanti o mantengono molti benefici dello stato interno dal ciclo di vita avanzato, gestito dal container, definito dal modello EJB `@Stateless`/ `@Stateful`/`@Singleton`", con supporto alla passivazione e pooling delle istanze.

Infine è ovvio quando occorre usare la gestione delle transazioni a livello di metodo, la sicurezza a livello di metoto, i timer, i metodi remoti o i metodi asincroni.

E' facile iniziare con un Web Bean semplice e poi volgere a EJB semplicemente aggiungendo l'annotazione: `@Stateless`, `@Stateful` o `@Singleton`.

### 1.3.3. Metodi produttori

Un *metodo produttore* è un metodo che viene chiamato dal manager Web Bean per ottenere un'istanza di un Web Bean quando non esiste alcuna istanza nel contesto corrente. Un metodo produttore lascia all'applicazione il pieno controllo del processo di istanziamento, invece di lasciare l'istanziamento al manager Web Bean. Per esempio:

```
@ApplicationScoped
public class Generator {

   private Random random = new Random( System.currentTimeMillis() );

   @Produces @Random int next() {
      return random.nextInt(100);
   }

}
```

Il risultato del metodo produttore è iniettato come qualsiasi altro Web Bean.

```
@Random int randomNumber
```

Il tipo di ritorno del metodo e tutte le interfacce che estende/implementa direttamente o indirettamente sono tipi di API del metodo produttore. Se il tipo di ritorno è una classe, tutte le superclassi sono anch'esse tipi di API.

Alcuni metodi produttori restituiscono oggetti che richiedono una distruzione esplicita:

```
@Produces @RequestScoped Connection connect(User user) {
   return createConnection( user.getId(), user.getPassword() );
}
```

Questi metodi produttori possono definire corrispondenti *metodi distruttori*:"

```
void close(@Disposes Connection connection) {
   connection.close();
}
```

Il metodo distruttore è chiamato direttamente dal manager Web Bean alla fine della richiesta.

Si parlerà in maggior dettaglio dei metodi produttori in *Capitolo 6, Metodi produttori*.

### 1.3.4. Endpoint JMS

Infine una coda od un topic JMS possono essere Web Bean. Web Beans solleva lo sviluppatore dalla noia della gestione dei cicli di vita di tutti i vari oggetti JMS richiesto per inviare messaggi a code o topic. Si discuteranno gli endpoint JMS in *Sezione 13.4, «Endpoint JMS»*.

# Esempio di applicazione web JSF

Illustriamo queste idee con un esempio completo. Implementiamo il login/logout dell'utente per un'applicazione che utilizza JSF. Innanzitutto definiamo un Web Bean che mantenga username e password digitati durante il login:

```
@Named @RequestScoped
public class Credentials {

   private String username;
   private String password;

   public String getUsername() { return username; }
   public void setUsername(String username) { this.username = username; }

   public String getPassword() { return password; }
   public void setPassword(String password) { this.password = password; }

}
```

Questo Web Bean è associato al login all'interno della seguente form JSF:

```
<h:form>
   <h:panelGrid columns="2" rendered="#{!login.loggedIn}">
      <h:outputLabel for="username"
>Username:</h:outputLabel>
      <h:inputText id="username" value="#{credentials.username}"/>
      <h:outputLabel for="password"
>Password:</h:outputLabel>
      <h:inputText id="password" value="#{credentials.password}"/>
   </h:panelGrid>
   <h:commandButton value="Login" action="#{login.login}" rendered="#{!login.loggedIn}"/>
   <h:commandButton value="Logout" acion="#{login.logout}" rendered="#{login.loggedIn}"/>
</h:form
>
```

Il vero lavoro è fatto da un Web Bean con scope di sessione che mantiene le informazioni sull'utente correntemente loggato ed espone l'entity `User` agli altri Web Beans:

```
@SessionScoped @Named
```

```
public class Login {

   @Current Credentials credentials;
   @PersistenceContext EntityManager userDatabase;

   private User user;

   public void login() {

      List<User
> results = userDatabase.createQuery(
         "select u from User u where u.username=:username and u.password=:password")
         .setParameter("username", credentials.getUsername())
         .setParameter("password", credentials.getPassword())
         .getResultList();

      if ( !results.isEmpty() ) {
         user = results.get(0);
      }

   }

   public void logout() {
      user = null;
   }

   public boolean isLoggedIn() {
     return user!=null;
   }

   @Produces @LoggedIn User getCurrentUser() {
      return user;
   }

}
```

`@LoggedIn` è un'annotazione di binding:

```
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD})
@BindingType
public @interface LoggedIn {}
```

Ora qualsiasi altro Web Bean può facilmente iniettare l'utente corrente:

```
public class DocumentEditor {

    @Current Document document;
    @LoggedIn User currentUser;
    @PersistenceContext EntityManager docDatabase;

    public void save() {
        document.setCreatedBy(currentUser);
        docDatabase.persist(document);
    }

}
```

Quest'esempio è un assaggio del modello di programmazione con Web Bean. Nel prossimo capitolo esploreremo la dependency injection dei Web Bean con maggior profondità.

# Implementazione di riferimento di Web Beans JSR-299

Web Beans viene sviluppato all'indirizzo *the Seam project* [http://seamframework.org/WebBeans]. Si può scaricare l'ultima release di Web Beans dalla *pagina di download* [http://seamframework.org/Download].

Web Beans viene distribuito con due applicazioni deployabili d'esempio, `webbeans-numberguess`, un esempio in formato war, che contiene solo bean semplici, e `webbeans-translator`, un esempio in formato ear, che contiene bean enterprise. Ci sono anche due varianti dell'esempio Indovina Numero, per Tomcat e con jsf2, che può essere usato con JSF2. Per eseguire gli esempi occorre fare le seguenti cose:

- l'ultima release di Web Beans,
- JBoss AS 5.0.1.GA, o
- Apache Tomcat 6.0.x, e
- Ant 1.7.0.

## 3.1. Usare JBoss AS 5

Occorre scaricare JBoss AS 5.0.1.GA da *jboss.org* [http://www.jboss.org/jbossas/downloads/], e scompattarlo. Per esempio:

```
$ cd /Applications
$ unzip ~/jboss-5.0.1.GA.zip
```

Scaricare Web Beans da *seamframework.org* [http://seamframework.org/Download], e scompattarlo. Per esempio

```
$ cd ~/
$ unzip ~/webbeans-$VERSION.zip
```

Quindi, occorre indicare a Web Beans dove è stato installato JBoss. Modificate il file `jboss-as/build.properties` e valorizzate la proprietà `jboss.home`. Per esempio:

```
jboss.home=/Applications/jboss-5.0.1.GA
```

Per installare Web Beans, occorre avere installato Ant 1.7.0, e avere valorizzato la variabile d'ambiente `ANT_HOME`. Per esempio:

```
$ unzip apache-ant-1.7.0.zip
$ export ANT_HOME=~/apache-ant-1.7.0
```

Quindi, è possibile installare gli aggiornamenti. Lo script di aggiornamento userà Maven per scaricare automaticamente Web Beans.

```
$ cd webbeans-$VERSION/jboss-as
$ ant update
```

Ora, siete pronti a fare il deploy del primo esempio!

### Suggerimento

Gli script di build degli esempio offrono una quantità di target per JBoss AS, quali:

- `ant restart` - fa il deploy dell'esempio in formato esploso
- `ant explode` - aggiorna un esempio in formato esploso, senza riavviare il deploy
- `ant deploy` - fa il deploy dell'esempio in formato jar compresso
- `ant undeploy` - rimuove l'esempio dal server
- `ant clean` - ripulisce l'esempio

Per fare il deploy dell'esempio Indovina Numero:

```
$ cd examples/numberguess
ant deploy
```

Avviare JBoss AS:

```
$ /Application/jboss-5.0.0.GA/bin/run.sh
```

**Suggerimento**

Se si usa Windows, si usi lo script `run.bat`.

Attendete che l'applicazione sia installata, e godetevi ore di divertimento all'indirizzo *http://localhost:8080/webbeans-numberguess*!

Web Beans include un secondo semplice esempio che tradurrà i vostri testi in Latino. L'esempio Indovina Numero è in formato war, e usa soltanto bean semplici; l'esempio col traduttore è in formato ear, e include dei bean enterprise, assemblati in un modulo EJB. Per provarlo:

```
$ cd examples/translator
ant deploy
```

Attendete che l'applicazione sia installata, e visitate l'indirizzo *http://localhost:8080/webbeans-translator*!

## 3.2. Usare Apache Tomcat 6.0

Scaricare Tomcat 6.0.18 o successivo da *tomcat.apache.org* [http://tomcat.apache.org/download-60.cgi], e scompattarlo. Per esempio

```
$ cd /Applications
$ unzip ~/apache-tomcat-6.0.18.zip
```

Scaricare Web Beans da *seamframework.org* [http://seamframework.org/Download], e scompattarlo. Per esempio

```
$ cd ~/
$ unzip ~/webbeans-$VERSION.zip
```

Quindi, occorre indicare a Web Beans dove è stato installato Tomcat. Modificate il file `jboss-as/build.properties` e valorizzate la proprietà `tomcat.home`. Per esempio:

```
tomcat.home=/Applications/apache-tomcat-6.0.18
```

### Suggerimento

Gli script di build degli esempi offrono una quantità di target per Tomcat, quali:

- `ant tomcat.restart` - esegue il deploy dell'esempio in formato esploso
- `ant tomcat.explode` - aggiorna un esempio in formato esploso, senza riavviare il deploy
- `ant tomcat.deploy` - esegue il deploy dell'esempio in formato jar compresso
- `ant tomcat.undeploy` - rimuove l'esempio dal server
- `ant tomcat.clean` - ripulisce l'esempio

Per eseguire il deploy dell'esempio Indovina Numero per tomcat:

```
$ cd examples/tomcat
ant tomcat.deploy
```

Avviare Tomcat:

```
$ /Applications/apache-tomcat-6.0.18/bin/startup.sh
```

### Suggerimento

Se si usa Windows, si usi lo script `startup.bat`.

Attendete che l'applicazione sia installata, e godetevi ore di divertimento all'indirizzo *http://localhost:8080/webbeans-numberguess*!

## 3.3. Usare GlassFish

DA FARE

## 3.4. Esempio Indovina Numero

Nell'applicazione Indovina Numero avete a disposizione 10 tentativi per indovinare un numero tra 1 e 100. Dopo ciascun tentativo, siete informati se siete stati troppo alti o troppo bassi.

L'esempio Indovina Numero comprende un certo numero di Web Bean, file di configurazione e pagine JSF, assemblati in un war. Iniziamo dai file di configurazione.

Tutti i file di configurazione di questo esempio si trovano in `WEB-INF/`, che è situato in `WebContent` nell'albero dei sorgenti. Innanzitutto, c'è `faces-config.xml`, in cui JSF viene informata di usare Facelets:

```
<?xml version='1.0' encoding='UTF-8'?>
<faces-config version="1.2"
              xmlns="http://java.sun.com/xml/ns/javaee"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/
javaee/web-facesconfig_1_2.xsd">

   <application>
      <view-handler
>com.sun.facelets.FaceletViewHandler</view-handler>
   </application>

</faces-config
>
```

Vi è un file vuoto `web-beans.xml`, che identifica l'applicazione come applicazione Web Beans.

Infine c'è `web.xml`:

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

<web-app version="2.5"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  http://java.sun.com/xml/ns/javaee/
web-app_2_5.xsd">

  <display-name
>Web Beans Numbergues example</display-name>

  <!-- JSF -->
                                                    (1)

  <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
>*.jsf</url-pattern>
  </servlet-mapping>

  <context-param>
    <param-name
>javax.faces.DEFAULT_SUFFIX</param-name>
    <param-value
>.xhtml</param-value>
  </context-param>

  <session-config>
    <session-timeout
>10</session-timeout>
  </session-config>

</web-app
>
```

(2) (3) (4) (5)

1. Enable and load the JSF servlet
2. Configure requests to `.jsf` pages to be handled by JSF
3. Tell JSF that we will be giving our source files (facelets) an extension of `.xhtml`
4. Configure a session timeout of 10 minutes

**Nota**

Whilst this demo is a JSF demo, you can use Web Beans with any Servlet based web framework.

Let's take a look at the Facelet view:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:ui="http://java.sun.com/jsf/facelets"
   xmlns:h="http://java.sun.com/jsf/html"
   xmlns:f="http://java.sun.com/jsf/core"
   xmlns:s="http://jboss.com/products/seam/taglib">


 <ui:composition template="template.xhtml">                                    (1)
   <ui:define name="content">
     <h1
>Guess a number...</h1>

     <h:form id="NumberGuessMain">                                             (2)
       <div style="color: red">
         <h:messages id="messages" globalOnly="false"/>
           <h:outputText id="Higher" value="Higher!" rendered="#{game.number gt game.guess
 and game.guess ne 0}"/>
         <h:outputText id="Lower" value="Lower!" rendered="#{game.number lt game.guess and
 game.guess ne 0}"/>
       </div>


       <div>                                                                   (3)
         I'm thinking of a number between #{game.smallest} and #{game.biggest}.
         You have #{game.remainingGuesses} guesses.
       </div>

       <div>

         Your guess:                                                           (4)
         <h:inputText id="inputGuess"
                   value="#{game.guess}"
                   required="true"
                   size="3"

                   disabled="#{game.number eq game.guess}">                    (5)
           <f:validateLongRange maximum="#{game.biggest}"
                        minimum="#{game.smallest}"/>
         </h:inputText>                                                        (6)
         <h:commandButton id="GuessButton"
                     value="Guess"
                     action="#{game.check}"
                     disabled="#{game.number eq game.guess}"/>
```

```
        </div>
        <div>
                    <h:commandButton id="RestartButton" value="Reset" action="#{game.reset}"
 immediate="true" />
        </div>
      </h:form>
    </ui:define>
  </ui:composition>
</html
>
```

1. Facelets is a templating language for JSF, here we are wrapping our page in a template which defines the header.
2. There are a number of messages which can be sent to the user, "Higher!", "Lower!" and "Correct!"
3. As the user guesses, the range of numbers they can guess gets smaller - this sentance changes to make sure they know what range to guess in.
4. This input field is bound to a Web Bean, using the value expression.
5. A range validator is used to make sure the user doesn't accidentally input a number outside of the range in which they can guess - if the validator wasn't here, the user might use up a guess on an out of range number.
6. And, of course, there must be a way for the user to send their guess to the server. Here we bind to an action method on the Web Bean.

L'esempio consiste di 4 classi, delle quali le prime due sono tipi di binding. Innanzitutto, c'è il tipo di binding `@Random`, usato per iniettare un numero casuale:

```
@Target( { TYPE, METHOD, PARAMETER, FIELD })
@Retention(RUNTIME)
@Documented
@BindingType
public @interface Random {}
```

C'è anche il tipo di binding `@MaxNumber`, usato per iniettare il numero massimo iniettatabile:

```
@Target( { TYPE, METHOD, PARAMETER, FIELD })
@Retention(RUNTIME)
@Documented
@BindingType
public @interface MaxNumber {}
```

Alla classe `Generator` è affidata la generazione del numero casuale, per mezzo di un metodo produttore. Inoltre essa espone il massimo numero possibile attraverso un metodo produttore:

```
@ApplicationScoped
public class Generator {

  private java.util.Random random = new java.util.Random( System.currentTimeMillis() );

  private int maxNumber = 100;

  java.util.Random getRandom()
  {
    return random;
  }

  @Produces @Random int next() {
    return getRandom().nextInt(maxNumber);
  }

  @Produces @MaxNumber int getMaxNumber()
  {
    return maxNumber;
  }

}
```

E' possibile notare che `Generator` ha scope applicazione; quindi non si ottiene un diverso numero casuale ogni volta.

Il Web Bean finale nell'applicazione è `Game` avente scope di sessione.

Si noti anche che è stata usata l'annotazione `@Named`, in modo che sia possibile usare il bean in espressioni EL presenti nelle pagine JSF. Infine, si è utilizzata l'iniezione del costruttore per inizializzare il gioco con un numero casuale. E naturalmente, è necessario dire al giocatore se ha vinto, informazione di feedback che viene fornita con un `FacesMessage`.

```
package org.jboss.webbeans.examples.numberguess;


import javax.annotation.PostConstruct;
import javax.faces.application.FacesMessage;
import javax.faces.context.FacesContext;
import javax.webbeans.AnnotationLiteral;
```

```
import javax.webbeans.Current;
import javax.webbeans.Initializer;
import javax.webbeans.Named;
import javax.webbeans.SessionScoped;
import javax.webbeans.manager.Manager;

@Named
@SessionScoped
public class Game
{
  private int number;

  private int guess;
  private int smallest;
  private int biggest;
  private int remainingGuesses;

  @Current Manager manager;

  public Game()
  {
  }

  @Initializer
  Game(@MaxNumber int maxNumber)
  {
    this.biggest = maxNumber;
  }

  public int getNumber()
  {
    return number;
  }

  public int getGuess()
  {
    return guess;
  }

  public void setGuess(int guess)
  {
    this.guess = guess;
  }
```

```
  public int getSmallest()
  {
    return smallest;
  }

  public int getBiggest()
  {
    return biggest;
  }

  public int getRemainingGuesses()
  {
    return remainingGuesses;
  }

  public String check()
  {
    if (guess
>number)
    {
      biggest = guess - 1;
    }
    if (guess<number)
    {
      smallest = guess + 1;
    }
    if (guess == number)
    {
      FacesContext.getCurrentInstance().addMessage(null, new FacesMessage("Correct!"));
    }
    remainingGuesses--;
    return null;
  }

  @PostConstruct
  public void reset()
  {
    this.smallest = 0;
    this.guess = 0;
    this.remainingGuesses = 10;
    this.number = manager.getInstanceByType(Integer.class, new AnnotationLiteral<Random
>(){});
  }
```

```
}
```

### 3.4.1. Esempio Indovina Numero per Tomcat

L'Indovina Numero di Tomcat differisce in un paio di punti. Innanzitutto Web Beans dovrebbe essere deployato come libreria Web Application in `WEB-INF/lib`. Per comodità viene fornito un singolo jar `webbeans-tomcat.jar` adatto per Web Beans in Tomcat.

**Suggerimento**

Certamente occorre anche includere JSF e EL, e le annotazioni comuni (`jsr250-api.jar`) che un server JEE include di default.

In secondo luogo, occorre specificare esplicitamente il servlet listener di Tomcat (usato per avviare Web Beans, e controllare la sua interazione con le richieste) in `web.xml`:

```
<listener>
  <listener-class
>org.jboss.webbeans.environment.tomcat.Listener</listener-class>
</listener
>
```

## 3.5. Esempio Traduttore

L'esempio Traduttore prende le frasi che vengono inserite e le traduce in latino.

L'esempio Traduttore è assemblato in un ear, e contiene EJB. Di conseguenza, la sua struttura è più complessa di quella dell'esempio Indovina Numero.

**Nota**

EJB 3.1 and Java EE 6 permettono di assemblare gli EJB in un war, cosa che rende questa struttura molto più semplice!

Innanzitutto, diamo un'occhiata all'aggregatore ear, che è situato nel modulo `webbeans-translator-ear`. Maven genera automaticamente il file `application.xml`:

```
<plugin>
  <groupId
>org.apache.maven.plugins</groupId>
```

```
  <artifactId
>maven-ear-plugin</artifactId>
  <configuration>
    <modules>
      <webModule>
        <groupId
>org.jboss.webbeans.examples.translator</groupId>
        <artifactId
>webbeans-translator-war</artifactId>
        <contextRoot
>/webbeans-translator</contextRoot>
      </webModule>
    </modules>
  </configuration>
</plugin
>
```

Qua viene impostato il context path, in modo da avere un url gradevole (*http://localhost:8080/webbeans-translator*).

**Suggerimento**

Se non si sta usando Maven per generare questi file, sarebbe necessario avere il file `META-INF/application.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns="http://java.sun.com/xml/ns/javaee"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  http://
java.sun.com/xml/ns/javaee/application_5.xsd"
             version="5">
  <display-name
>webbeans-translator-ear</display-name>
  <description
>Ear Example for the reference implementation of JSR 299: Web Beans</
description>

  <module>
    <web>
      <web-uri
>webbeans-translator.war</web-uri>
      <context-root
>/webbeans-translator</context-root>
```

```
    </web>
  </module>
  <module>
    <ejb
>webbeans-translator.jar</ejb>
  </module>
</application
>
```

Quindi, esaminiamo il war. Proprio come nell'esempio Indovina Numero, abbiamo un `faces-config.xml` (per abilitare Facelets) e un `web.xml` (per abilitare JSF) in `WebContent/WEB-INF`.

Più interessante è il facelet usato per tradurre il testo. Proprio come nell'esempio Indovina Numero c'è un template, che circoscrive la form (qui omessa per brevità):

```
<h:form id="NumberGuessMain">

  <table>
    <tr align="center" style="font-weight: bold" >
      <td>
        Your text
      </td>
      <td>
        Translation
      </td>
    </tr>
    <tr>
      <td>
      <h:inputTextarea id="text" value="#{translator.text}" required="true" rows="5" cols="80" />
      </td>
      <td>
        <h:outputText value="#{translator.translatedText}" />
      </td>
    </tr>
  </table>
  <div>
    <h:commandButton id="button" value="Translate" action="#{translator.translate}"/>
  </div>

</h:form
>
```

L'utente può inserire del testo nell'area di testo sulla sinistra e premere il pulsante di traduzione per vedere il risultato sulla destra.

Infine, si esamini il modulo ejb, `webbeans-translator-ejb`. In `src/main/resources/META-INF` si trova un file vuoto `web-beans.xml`, usato per marcare l'archivio come contenente Web Beans.

Abbiamo lasciato per ultimo il boccone più prelibato, il codice! Il progetto ha due bean semplici, `SentenceParser` e `TextTranslator` e due bean enterprise, `TranslatorControllerBean` e `SentenceTranslator`. Dovreste ormai essere piuttosto familiari all'aspetto di un Web Bean, così ci limiteremo a evidenziare le parti più interessanti.

Sia `SentenceParser` che `TextTranslator` sono bean dependenti, e `TextTranslator` usa l'inizializzazione via costruttore:

```
public class TextTranslator {
  private SentenceParser sentenceParser;
  private Translator sentenceTranslator;

  @Initializer
  TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator)
  {
    this.sentenceParser = sentenceParser;
    this.sentenceTranslator = sentenceTranslator;
```

`TextTranslator` è un bean stateless (con un'interfaccia business locale), dove avviene la magia - naturalmente, non potevamo sviluppare un traduttore completo, ma gli abbiamo dato un buon avvio!

Infine, vi è un controller orientato all'UI, che raccoglie il testo dall'utente, e lo invia al traduttore. Questo è un bean stateful di sessione, dotato di nome, con scope richiesta, in cui viene iniettato il traduttore.

```
@Stateful
@RequestScoped
@Named("translator")
public class TranslatorControllerBean implements TranslatorController
{

  @Current TextTranslator translator;
```

Il bean possiede pure dei metodi getter e setter per tutti i campi della pagina.

Poichè si tratta di un bean stateful di sessione, è necessario un metodo di rimozione (remove method):

```
@Remove
public void remove()
{

}
```

Il manager Web Beans chiamerà il metodo di rimozione quando il bean verrà distrutto; in questo caso al termine della richiesta.

Ciò conclude il nostro breve tour degli esempi della RI di Web Beans. Per saperne di più, o per trovare ulteriore aiuto, per favore visitate *http://www.seamframework.org/WebBeans/Development*.

Abbiamo bisogno di aiuto in tutte le aree - soluzione dei bug, scrittura di nuove caratteristiche ed esempi e traduzione di questa guida.

# Dependency injection

Web Beans supporta tre meccanismi primari per la dependency injection:

Iniezione dei parametri del costruttore

```
public class Checkout {

   private final ShoppingCart cart;

   @Initializer
   public Checkout(ShoppingCart cart) {
      this.cart = cart;
   }

}
```

Iniezione dei parametri del *metodo inizializzatore (initializer method)*:

```
public class Checkout {

   private ShoppingCart cart;

   @Initializer
   void setShoppingCart(ShoppingCart cart) {
      this.cart = cart;
   }

}
```

Iniezione diretta degli attributi

```
public class Checkout {

   private @Current ShoppingCart cart;

}
```

La dependency injection avviene sempre quando il Web Bean viene istanziato la prima volta.

- Innanzitutto, il Web Bean manager chiama il costruttore, per ottenere un'istanza del Web Bean.
- Quindi, il Web Bean manager inizializza i valori di tutti i campi del Web Bean soggetti ad iniezione.
- Quindi, il Web Bean manager chiama tutti i metodi inizializzatori del Web Bean.
- Infine, se ne esiste uno, viene chiamato il metodo del Web Bean annotato con `@PostConstruct`.

L'iniezione dei parametri del costruttore non è supportata per gli EJB, poiché gli EJB sono istanziati dal container EJB, non dal manager Web Bean.

I parametri dei costruttori e dei metodi di inizializzazione non devono essere annotati esplicitamente quando il tipo del binding è `@Current`, quello predefinito. I campi iniettati, comunque, *devono* specificare il tipo del binding, anche quando il tipo del binding è quello predefinito. Se il campo non specifica il tipo del binding, non verrà iniettato.

I metodi produttori supportano anche l'iniezione dei parametri:

```
@Produces Checkout createCheckout(ShoppingCart cart) {
   return new Checkout(cart);
}
```

Infine, i metodi observer (che vedremo in *Capitolo 9, Eventi*), i metodi disposal e i metodi distruttori supportano tutti l'iniezione dei parametri.

Le specifiche Web Beans definiscono una procedura, chiamata *typesafe resolution algorithm (algoritmo di risoluzione sicura rispetto ai tipi)*, che il manager Web Bean segue quando deve identificare il Web Beanda iniettare in punto di iniezione. Questo algoritmo di primo acchito sembra complesso, ma una volta che lo si è compreso, in realtà, risulta piuttosto intuitivo. La risoluzione sicura dei tipi viene eseguita durante l'inizializzazione del sistema (system initialization time), il che significa che il manager Web Bean informerà immediatamente un utente se le dipendenze di un Web Bean non possono essere soddisfatte, lanciando una `UnsatisfiedDependencyException` o una `AmbiguousDependencyException`.

Lo scopo di questo algoritmo è di permettere a più Web Bean di implementare la stessa tipo definito dall'API e:

- permettere al client di selezionare l'implementazione richiesta usando le *binding annotations*,
- permettere all'installatore dell'applicazione (deployer) di selezionare quale implementazione è appropriata per un particolare deploy, senza cambiamenti al client, abilitando o disabilitando *i tipi di deploy*, o
- permette ad un'implementazione della API di fare l'override di un'altra implementazione della stessa API a deployment time, senza apportare modifiche al client, usando *la precedenza fra tipi di deploy (deployment type precedence)*.

Indaghiamo come il manager di Web Beans individua un Web Bean da iniettare.

## 4.1. Annotazioni di binding

Se esiste più di un Web Bean che implementa un particolare tipo di API, il punto di iniezione può specificare esattamente quale Web Bean dovrebbe essere iniettato usando una binding annotation. Per esempio, ci potrebbero essere due implementazioni di `PaymentProcessor`:

```
@PayByCheque
public class ChequePaymentProcessor implements PaymentProcessor {
   public void process(Payment payment) { ... }
}
```

```
@PayByCreditCard
public class CreditCardPaymentProcessor implements PaymentProcessor {
   public void process(Payment payment) { ... }
}
```

Dove `@PayByCheque` e `@PayByCreditCard` sono binding annotation:

```
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
@BindingType
public @interface PayByCheque {}
```

```
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
@BindingType
public @interface PayByCreditCard {}
```

Lo sviluppatore di un Web Bean client usa la binding annotation per specificare esattamente quale Web Bean debba essere iniettato.

Utilizzando l'iniezione a livello di campo:

```
@PayByCheque PaymentProcessor chequePaymentProcessor;
@PayByCreditCard PaymentProcessor creditCardPaymentProcessor;
```

Utilizzando l'iniezione a livello di metodo inizializzatore:

```
@Initializer
public void setPaymentProcessors(@PayByCheque PaymentProcessor
 chequePaymentProcessor,
                          @PayByCreditCard PaymentProcessor creditCardPaymentProcessor) {
  this.chequePaymentProcessor = chequePaymentProcessor;
  this.creditCardPaymentProcessor = creditCardPaymentProcessor;
}
```

O usando l'iniezione a livello di costruttore:

```
@Initializer
public Checkout(@PayByCheque PaymentProcessor chequePaymentProcessor,
            @PayByCreditCard PaymentProcessor creditCardPaymentProcessor) {
  this.chequePaymentProcessor = chequePaymentProcessor;
  this.creditCardPaymentProcessor = creditCardPaymentProcessor;
}
```

### 4.1.1. Annotazioni di binding con membri

Le binding annotation possono avere dei membri:

```
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
@BindingType
public @interface PayBy {
   PaymentType value();
}
```

Nel qual caso, il valore del membro è significativo:

```
@PayBy(CHEQUE) PaymentProcessor chequePaymentProcessor;
@PayBy(CREDIT_CARD) PaymentProcessor creditCardPaymentProcessor;
```

E' possibile indicare al manager Web Bean di ignorare un membro di un tipo di binding annotation annotando il membro con `@NonBinding`.

### 4.1.2. Combinazioni di binding annotation

Un punto di iniezione può anche specificare più binding annotation:

```
@Asynchronous @PayByCheque PaymentProcessor paymentProcessor
```

In questo caso, soltanto un Web Bean che ha *entrambe* le binding annotation sarebbe candidato ad essere iniettato.

### 4.1.3. Binding annotation e metodi produttori

Anche i metodi produttori possono specificare binding annotation:

```
@Produces
@Asynchronous @PayByCheque
PaymentProcessor createAsyncPaymentProcessor(@PayByCheque PaymentProcessor
 processor) {
   return new AsynchronousPaymentProcessor(processor);
}
```

### 4.1.4. Il tipo di binding predefinito

Web Beans definisce un tipo di binding `@Current` che è il tipo di binding predefinito per ogni punto di iniezione o per ogni Web Bean che non specifichi esplicitamente un tipo di binding.

Vi sono due circostanze comuni in cui è necessario specificare esplicitamente l'annotazione `@Current`:

- su un campo, allo scopo di dichiararne l'iniezione con il tipo di binding predefinito, and
- su un Web Bean che ha un tipo di binding aggiuntivo rispetto al tipo di binding predefinito.

## 4.2. Tipi di deploy

Tutti i Web Bean hanno un *tipo di deployment (deployment type)*. Ogni tipo di deployment identifica un insieme di Web Bean che dovrebbe essere installato in modo condizionale in corrispondenza ad alcuni deploy del sistema.

Per esempio, potremmo definire un tipo di deploy denominato `@Mock`, che identifichi i Web Bean da installare soltanto quando il sistema è posto in esecuzione in un ambiente di test integrato:

```
@Retention(RUNTIME)
 @Target({TYPE, METHOD})
 @DeploymentType
```

```
public @interface Mock {}
```

Supponiamo di avere alcuni Web Bean che interagiscano con un sistema di pagamenti esterno:

```
public class ExternalPaymentProcessor {

   public void process(Payment p) {
      ...
   }

}
```

Dal momento che questo Web Bean non specifica esplicitamente un tipo di deploy, ha il tipo di deploy predefinito `@Production`.

Per le attività di test (d'unità o integrazione), il sistema esterno è lento o non disponibile. Così sarebbe necessario creare un oggetto mock:

```
@Mock
public class MockPaymentProcessor implements PaymentProcessor {

   @Override
   public void process(Payment p) {
      p.setSuccessful(true);
   }

}
```

Ma in che modo il manager Web Bean determina quale implementazione usare con un particolare deploy?

### 4.2.1. Abilitazione dei tipi di deploy

Web Beans definisce due tipi di deploy percostituiti: `@Production` e `@Standard`. Di default, sono abilitati soltanto i Web Bean con i tipi di deploy precostituiti quando si procede al deploy del sistema. E' possibile abilitare dei tipi di deploy aggiuntivi per un particolare deploy elencandoli in `web-beans.xml`.

Tornando al nostro esempio, quando si fa il deploy dei test di integrazione, si vuole che tutti gli oggetti `@Mock` vengano installati:

```
<WebBeans>
```

```
   <Deploy>
      <Standard/>
      <Production/>
      <test:Mock/>
   </Deploy>
</WebBeans
>
```

Ora il manager Web Bean identificherà ed installerà tutti i Web Bean annotati con `@Production`, `@Standard` o `@Mock` a deployment time.

Il tipo di deploy `@Standard` è usato soltanto per dei Web Bean speciali definiti nelle specifiche. Non è possibile utilizzarlo per i propri Web Bean, e non è possibile disabilitarlo.

Il tipo di deploy `@Production` è quello predefinito per i Web Bean che non dichiarano esplicitamente un tipo di deploy, e può essere disabilitato.

### 4.2.2. Precedenza del tipo di deploy

Se avete prestato attenzione, vi state probabilmente chiedendo come il manager Web Bean decida quale implementazione scegliere # `ExternalPaymentProcessor` o `MockPaymentProcessor` # Si consideri cosa succede quando il manager incontra questo punto di iniezione:

```
@Current PaymentProcessor paymentProcessor
```

Vi sono ora due Web Bean che soddisfano l'interfaccia di `PaymentProcessor`. Naturalmente, non è possibile utilizzare una binding annotation per eliminare l'ambiguità, poiché le binding annotation sono cablate nel sorgente in corrispondenza al punto di iniezione, e noi vogliamo che il manager sia in grado di decidere a deployment time!

La soluzione a questo problema sta nel fatto che ciascun tipo di deploy ha una diversa *precedenza*. La precedenza dei tipi di deploy è determinata dall'ordine con cui appaiono in `web-beans.xml`. Nel nostro esempio, `@Mock` compare dopo `@Production` cosicché ha una precedenza più alta.

Ogni volta che il manager scopre che più di un Web Bean potrebbe soddisfare il contratto (interfaccia più binding annotation) specificato da un punto di iniezione, passa a considerare la precedenza relativa dei Web Bean. Se uno ha una precedenza superiore a quella degli altri, questo viene scelto per essere iniettato. Così, nel nostro esempio, il manager Web Bean inietterà `MockPaymentProcessor` quando viene eseguito nel nostro ambiente di test (che è esattamente ciò che vogliamo).

E' interessante confrontare questa funzionalità con le architetture di gestone oggi in voga. Vari container "lightweight" permettono il deploy condizionale di classi che esistono nel classpath, ma le classi che devono essere installate devono essere elencate esplicitamente ed individualmente

nel codice di configurazione o in qualche file XML di configurazione. Web Beans supporta certo la definizione e configurazione dei Web Bean attraverso l'XML, ma nei casi comuni in cui non si renda necessaria una configurazione complicata, i tipi di deploy permettono di abilitare un insieme completo di Web Bean con una sola linea di XML. Al contempo, uno sviluppatore che esamini il codice, potrà facilmente identificare gli scenari di deploy in cui il Web Bean sarà utilizzato.

### 4.2.3. Esempio dei tipi di deploy

I tipi di deploy sono utili in molte situazioni. Di seguito riportiamo alcuni esempi:

- i tipi di deploy individuati da `@Mock` e `@Staging` per i test
- il tipo di deploy `@AustralianTaxLaw` per i Web Bean di un'applicazione specifica
- i tipi di deploy `@SeamFramework` e `@Guice` per framework di terze parti basati su Web Beans
- `@Standard` per Web Bean standard definiti dalle specifiche di Web Beans

Sono sicuro che siate in grado di escogitare altre applicazioni...

## 4.3. Risoluzione di dipendenze non soddisfatte

L'algoritmo di risoluzione sicura rispetto ai tipi fallisce quando, dopo avere considerato le binding annotation e i tipi di deploy di tutti i Web Bean che implementano il tipo di un punto di iniezione, il manager Web Bean non è in grado di identificare esattamente uno ed un solo Web Bean da iniettare.

Di solito è semplice porre rimedio a un'eccezione `UnsatisfiedDependencyException` o `AmbiguousDependencyException`.

Per rimediare ad una `UnsatisfiedDependencyException`, si deve semplicemente fornire un Web Bean che implementi il tipo dell'API in uso e abbia gli stessi tipi di binding del punto di iniezione #151; o si deve abilitare il tipo di deploy di un Web Bean che già implementa il tipo dell'API in uso e possiede i tipi di binding in esame.

Per porre rimedio a una `AmbiguousDependencyException`, si deve introdurre un tipo di binding per distinguere tra le due implementazioni del tipo delle API, o si deve cambiare il tipo di deploy di una delle implementazione in modo che il manager Web Bean possa usare la precedenza dei tipi di deploy per scegliere fra di essi. Una `AmbiguousDependencyException` può verificarsi soltanto se due Web Bean condividono il tipo di binding e hanno esattamente lo stesso tipo di deploy.

Vi è un ulteriore questione di cui occorre essere a conoscenza quando si usa la dependency injection in Web Beans.

## 4.4. Client proxy

I client di un Web Bean che sono stati iniettati solitamente non hanno un riferimento diretto all'istanza del Web Bean.

Immaginiamo che un Web Bean associato allo scope applicazione tenga un riferimento diretto a un Web Bean associato allo scope richiesta. Il Web Bean con scope applicazione è condiviso fra molte diverse richieste. Comunque, ciascuna richiesta dovrebbe vedere una diversa istanza del Web bean con scope richiesta!

Immaginiamo ora che un Web Bean con scope sessione abbia un riferimento diretto a un Web Bean con scope applicazione . Ogni tanto, il contesto della sessione viene serializzato su disco in modo da usare la memoria in modo più efficiente. Comunque, l'istanza del Web Bean con scope applicazione non dovrebbe essere serializzato insieme al Web Bean con scope sessione!

Quindi, a meno che un Web Bean abbia lo scope predefinito `@Dependent`, il manager Web Bean deve rendere indiretti tutti i riferimenti al Web Bean iniettati attraverso un oggetto proxy. Questo *client proxy* ha la responsabilità di assicurare che l'istanza del Web Bean su cui viene invocato un metodo sia l'istanza associata al contesto corrente. Il client proxy, inoltre, permette ai Web Bean associati a contesti come quello di sessione di essere salvati su disco senza serializzare ricorsivamente altri Web Beans che siano stati iniettati.

Purtroppo, a causa di limitazioni del linguaggio Java, alcuni tipi Java non possono essere gestiti tramite un proxy dal manager Web Bean. Quindi, il manager Web Bean lancia un'eccezione `UnproxyableDependencyException` se il tipo di un punto di iniezione non può essere gestito tramite proxy.

I seguenti tipi Java non possono essere gestiti tramite proxy dal manager Web Bean:

- classi dichiarate `final` o che abbiano un metodo `final`,
- classi che non abbiano costruttori non privati (non-private) senza parametri, e
- array e tipi primitivi.

Di solito è molto facile rimediare a una `UnproxyableDependencyException`. Si deve semplicimente aggiungere un costruttore privo di parametri alla classe iniettata, introdurre un'interfaccia, o modificare lo scope del Web Bean iniettato a `@Dependent`.

## 4.5. Ottenere un riferimento a un Web Bean via codice

L'applicazione può ottenere un'istanza dell'interfaccia `Manager` attraverso iniezione:

```
@Current Manager manager;
```

L'oggetto `Manager` fornisce un insieme di metodi per ottenere l'istanza di un Web Bean via codice.

```
PaymentProcessor p = manager.getInstanceByType(PaymentProcessor.class);
```

Le binding annotation possono essere specificate come sottoclassi della classe helper `AnnotationLiteral`, poiché è altrimenti difficile istanziare un tipo annotazione in Java.

```
PaymentProcessor p = manager.getInstanceByType(PaymentProcessor.class,
                                    new AnnotationLiteral<CreditCard
>(){});
```

Se il tipo di binding ha un membro annotazione, non è possibile utilizzare una sottoclasse anonima di `AnnotationLiteral` # sarà invece necessario creare una sottoclasse non anonima:

```
abstract class CreditCardBinding
   extends AnnotationLiteral<CreditCard
>
   implements CreditCard {}
```

```
PaymentProcessor p = manager.getInstanceByType(PaymentProcessor.class,
                                    new CreditCardBinding() {
                                       public void value() { return paymentType; }
                                    } );
```

## 4.6. Chiamare al ciclo di vita, `@Resource`, `@EJB` e `@PersistenceContext`

I Web Beans di tipo enterprise supportano tutte le callback del ciclo di vita definite dalle specifiche EJB: `@PostConstruct`, `@PreDestroy`, `@PrePassivate` e `@PostActivate`.

Semplici Web Bean supportano soltanto le callback `@PostConstruct` e `@PreDestroy`.

Sia i Web Bean semplici che quelli enterprise supportano l'uso di `@Resource`, `@EJB` e `@PersistenceContext` per l'iniezione rispettivamente di risorse Java EE, di EJB e di contesti di persistenza JPA. I Web Bean semplici non supportano l'uso di `@PersistenceContext(type=EXTENDED)`.

La callback `@PostConstruct` viene sempre eseguita dopo che tutte le dipendenze sono state iniettate.

## 4.7. L'oggetto `InjectionPoint`

Certi tipi di oggetti dipendenti # Web Bean con scope `@Dependent` # hanno bisogno di avere informazioni riguardo l'oggetto o il punto in cui sono iniettati per fare quello che devono. Per esempio:

- La categoria di log per un `Logger` dipende dalla classe dell'oggetto che lo contiene.

- L'iniezione di un parametro o di un header HTTP dipende dal nome del parametro o dello header specificato nel punto di iniezione.
- L'iniezione del risultato di una espressione EL dipende dall'espressione specificata nel punto di iniezione.

Un Web Bean con scope `@Dependent` può essere iniettato con un'istanza di `InjectionPoint` e accedere i metadati riguardanti il punto di iniezione cui appartiene.

Vediamo un esempio. Il codice seguente è prolisso e vulnerabile a problemi di refactoring:

```
Logger log = Logger.getLogger(MyClass.class.getName());
```

Questo piccolo e intelligente metodo produttore permette di iniettare un `Logger` JDK senza specificare esplicitamente la categoria di log:

```
class LogFactory {

  @Produces Logger createLogger(InjectionPoint injectionPoint) {
    return Logger.getLogger(injectionPoint.getMember().getDeclaringClass().getName());
  }

}
```

Ora è possibile scrivere:

```
@Current Logger log;
```

Non siete convinti? Eccovi un secondo esempio. Per iniettare parametri HTTP, è necessario definire un tipo di binding:

```
@BindingType
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface HttpParam {
  @NonBinding public String value();
}
```

Potremmo usare questo tipo di binding in corrispondenza ai punti di iniezione in questo modo:

```
@HttpParam("username") String username;
@HttpParam("password") String password;
```

Il seguente metodo produttore esegue il lavoro:

```
class HttpParams

  @Produces @HttpParam("")
  String getParamValue(ServletRequest request, InjectionPoint ip) {
    return request.getParameter(ip.getAnnotation(HttpParam.class).value());
  }

}
```

(Occorre notare che il membro `value()` dell'annotazione `HttpParam` viene ignorato dal manager Web Bean poiché è annotato con `@NonBinding`.)

Il manager Web Bean fornisce un Web Bean di sistema che implementa l'interfaccia `InjectionPoint`:

```
public interface InjectionPoint {
  public Object getInstance();
  public Bean<?> getBean();
  public Member getMember():
  public <T extends Annotation
> T getAnnotation(Class<T
> annotation);
  public Set<T extends Annotation
> getAnnotations();
}
```

# Scope e contesti

Finora si sono visti pochi esempi di *annotazioni di tipi di scope*. Gli scope di un Web Bean determinano il ciclo di vita del Web Bean. Lo scope determina anche quali client fanno riferimento a quali istanze di Web Bean. Secondo la specifica Web Bean, uno scope determina:

- Quando una nuova istanza di un Web Bean con tale scope viene creata
- Quando un'istanza esistente di un Web Bean con tale scope viene distrutta
- Quali riferimenti iniettati puntano a istanze di un Web Bean con tale scope

Per esempio, se esiste un Web Bean con scope di sessione, `CurrentUser`, tutti i Web Bean che vengono chiamati nel contesto della medesima `HttpSession` vedranno la stessa istanza di `CurrentUser`. Quest'istanza verrà automaticamente creata la prima volta che in tale sessione occorre `CurrentUser`, e verrà distrutta automaticamente quando la sessione termina.

## 5.1. Tipi di scope

Web Bean fornisce un *modello di contesto estensibile*. E' possibile definire nuovi scope creando una nuova annotazione di un tipo di scope.

```
@Retention(RUNTIME)
@Target({TYPE, METHOD})
@ScopeType
public @interface ClusterScoped {}
```

Sicuramente questa è la parte facile del lavoro. Affinché questo tipo di scope sia utile, avremo bisogno di definire un oggetto `Contesto` che implementi lo scope! Implementare un `Contesto` è compito molto tecnico, inteso solo per lo sviluppo di framework.

Si può applicare un'annotazione con un tipo scope ad una classe di implementazione Web Bean per specificare lo scope del Web Bean:

```
@ClusterScoped
public class SecondLevelCache { ... }
```

Solitamente si userà uno degli scopi predefiniti di Web Beans.

## 5.2. Scope predefiniti

Web Beans definisce quattro scope predefiniti:

- `@RequestScoped`
- `@SessionScoped`
- `@ApplicationScoped`
- `@ConversationScoped`

Per un'applicazione web che impiega Web Beans:

- qualsiasi richiesta servlet ha accesso a scope attivi di richiesta, sessione e applicazione e, in aggiunta
- qualsiasi richiesta JSF ha accesso allo scope attivo di conversazione

Gli scope di richiesta ed applicazione sono pure attivi:

- durante le invocazioni a metodi remoti EJB,
- durante i timeout EJB,
- durante la consegna dei messaggi a bean message-drive, e
- durante invocazioni web service.

Se l'applicazione prova ad invocare un Web Bean con scope che non ha un contesto attivo, una `ContextNotActiveException` viene lanciata a runtime dal manager Web Bean.

Tre dei quattro scope predefiniti dovrebbero essere estremamente familiari ad ogni sviluppatore Java EE, quindi non si procede oltre nella discussione. Uno degli scope è comunque nuovo.

## 5.3. Lo scope conversazione

Lo scope di conversazione di Web Beans è un pò come il tradizionale scope di sessione in cui viene mantenuto lo stato associato all'utente del sistema, e vengono create richiest multiple al server. Comunque, a differenza dello scope di sessione, lo scope di conversazione:

- è demarcato esplicitamente dall'applicazione, e
- mantiene lo stato associato ad un particolare tab del browser in un'applicazione JSF

Una conversazione rappresenta un task, un'unità di lavoro dal punto di vista dell'utente. Il contesto di conversazione mantiene uno stato associato all'utente che sta lavorando. Se l'utente sta facendo più cose contemporaneamente ci saranno più conversazioni.

Il contesto di conversazione è attivo durante ogni richiesta JSF. Comunque, la maggior parte delle conversazioni vengono distrutte alla fine della richiesta. Se una conversazione deve mantenere lo stato nel corso richieste multiple, deve esplicitamente essere promossa a *conversazione long-running*.

### 5.3.1. Demarcazione della conversazione

Web Beans fornisce un Web Bean predefinito per controllare il ciclo di vita delle conversazioni in un'applicazione JSF. QUesto Web Bean può essere ottenuto per iniezione:

```
@Current Conversation conversation;
```

Per promuovere a long-running la conversazione associata alla richiesta corrente, occorre chiamare il metodo `begin()` dal codice dell'applicazione. Per schedulare la distruzione del contesto attuale della conversazione long-running, si chiami `end()`.

Nel seguente esempio un Web Bean con scope di conversazione controlla la conversazione alla quale è associato:

```
@ConversationScoped @Stateful
public class OrderBuilder {

   private Order order;
   private @Current Conversation conversation;
   private @PersistenceContext(type=EXTENDED) EntityManager em;

   @Produces public Order getOrder() {
      return order;
   }

   public Order createOrder() {
      order = new Order();
      conversation.begin();
      return order;
   }

   public void addLineItem(Product product, int quantity) {
      order.add( new LineItem(product, quantity) );
   }

   public void saveOrder(Order order) {
      em.persist(order);
      conversation.end();
   }

   @Remove
   public void destroy() {}
```

```
}
```

Questo Web Bean è capace di controlla il proprio ciclo di vita attraverso l'uso della API `Conversation`. Ma altri Web BEan hanno un ciclo di vita che dipende completamente da un altro oggetto.

## 5.3.2. Propagazione della conversazione

Il contesto di conversazione viene automaticamente propagato con ogni richiesta JSF faces (invio di form JSF). Non si propaga in modo automatico con richiesta non-faces, per esempio, navigazione tramite un link.

E' possibile forzare la conversazione da propagare tramite richiesta non-faces attraverso l'inclusione di un identificatore univoco della conversazione come parametro di richiesta. La specifica Web Bean riserva un parametro di richiesta chiamato `cid` per tale uso. L'identificatore univoco della conversazione può essere ottenuto dall'oggetto `Conversation`, che ha il nome Web Beans `conversation`.

Quindi il seguente link propaga la conversazione:

```
<a href="/addProduct.jsp?cid=#{conversation.id}"
>Add Product</a
>
```

Il manager Web Bean deve propagare le conversazioni attraverso i redirect, anche se la conversazione non è marcata long-running. Questo rende facile implementare il comune pattern POST-then-redirect, senza impiegare fragili costrutti quali oggetti "flash". In questo caso il manager Web Bean aggiunge automaticamente un parametro di richiesta all'URL di redirect.

## 5.3.3. Timeout della conversazione

Al manager Web Bean è permesso di distruggere una conversazione e tutto lo stato mantenuto nel contesto in qualsiasi momento al fine di preservare le risorse. Un'implementazione di un manager Web Bean eseguirà questo sulla base di un qualche timeout # sebbene non sia richiesto dalla specifica Web Beans. Il timeout è il periodo di inattività prima che la conversazione venga distrutta.

L'oggetto `Conversation` fornisce un metodo per impostare iltimeout. Questo è solo un suggerimento al manager Web Bean, che è libero di ignorare quest'impostazione.

```
conversation.setTimeout(timeoutInMillis);
```

## 5.4. Pseudo-scope dipendente

In aggiunta ai quattro scope predefiniti, Web Bean fornisce il cosiddetto *pseudo-scope dipendente*. Questo è lo scope di default per un Web Bean che non dichiara esplicitamente un tipo di scope.

Per esempio questo Web Bean ha uno scope di tipo `@Dependent`:

```
public class Calculator { ... }
```

Quando un punto di iniezione di un Web Bean risolve verso un Web Bean dipendente, viene creata una nuova istanza di Web Bean dipendente ogni volta che il primo Web Bean viene istanziato. Le istanze dei Web Beans dipendenti non vengono mai condivise tra Web Bean differenti o punti di iniezione differenti. Sono *oggetti dipendenti* di altre istanze Web Bean.

Istanze Web Bean dipendenti vengono distrutte quando viene distrutta l'istanza da cui dipendono.

Web Beans facilità l'ottenimento di un'istanza dipendente di una classe Java o bean EJB, anche se la classe o bean EJB sono già dichiarati come Web Bean con qualche altro tipo di scope.

### 5.4.1. Annotazione `@New`

L'annotazione predefinita di binding `@New` consente la definizione*implicita* di un Web Bean dipendente in un punti di iniezione. Si supponga di dichiarare il seguentecampo iniettato:

```
@New Calculator calculator;
```

Allora viene implicitamente definito il Web Bean con scope `@Dependent`, tipo di binding `@New`, tipo di API `Calculator`, classe di implementazione `Calculator` e tipo di deploy `@Standard`.

Questo è vero se `Calculator` è *già* dichiarato con un tipo di scope differente, per esempio:

```
@ConversationScoped
public class Calculator { ... }
```

Quindi i seguenti attributi iniettati ricevono ciascuno un'istanza di `Calculator`:

```
public class PaymentCalc {

   @Current Calculator calculator;
   @New Calculator newCalculator;
```

```
}
```

Il campo `calculator` ha iniettata un'istanza con scope conversazionale di `Calculator`. Il campo `newCalculator` ha iniettata un nuova istanza di `Calculator`, con ciclo di vita che è legato a `PaymentCalc`.

Questa caratteristica è particolarmente utile con i metodi produttori, come si vedrà nel prossimo capitolo.

# Metodi produttori

I metodi produttori consentono di superare alcune limitazioni che sorgono quando il manager Web Bean è responsabile dell'istanziamento degli oggetti al posto dell'applicazione. Questi metodi sono anche il modo migliore per integrare gli oggetti che non sono Web Beans dentro l'ambiente Web Beans. (Si incontrerà un secondo approccio in *Capitolo 12, Definire i Web Beans tramite XML*.)

Secondo la specifica:

> A Web Beans producer method acts as a source of objects to be injected, where:
>
> - the objects to be injected are not required to be instances of Web Beans,
> - the concrete type of the objects to be injected may vary at runtime or
> - the objects require some custom initialization that is not performed by the Web Bean constructor

For example, producer methods let us:

- expose a JPA entity as a Web Bean,
- expose any JDK class as a Web Bean,
- define multiple Web Beans, with different scopes or initialization, for the same implementation class, or
- vary the implementation of an API type at runtime.

In particular, producer methods let us use runtime polymorphism with Web Beans. As we've seen, deployment types are a powerful solution to the problem of deployment-time polymorphism. But once the system is deployed, the Web Bean implementation is fixed. A producer method has no such limitation:

```
@SessionScoped
public class Preferences {

   private PaymentStrategyType paymentStrategy;

   ...

   @Produces @Preferred
   public PaymentStrategy getPaymentStrategy() {
```

```
        switch (paymentStrategy) {
            case CREDIT_CARD: return new CreditCardPaymentStrategy();
            case CHEQUE: return new ChequePaymentStrategy();
            case PAYPAL: return new PayPalPaymentStrategy();
            default: return null;
        }
    }

}
```

Consider an injection point:

```
@Preferred PaymentStrategy paymentStrat;
```

This injection point has the same type and binding annotations as the producer method, so it resolves to the producer method using the usual Web Beans injection rules. The producer method will be called by the Web Bean manager to obtain an instance to service this injection point.

## 6.1. Scope di un metodo produttore

Lo scope dei metodi produttori è di default impostato a `@Dependent`, e quindi verrà chiamato *ogni volta* che il manager Web Bean inietta questo campo o qualsiasi altro campi che risolve lo stesso metodo produttore. Quindi ci potrebbero essere istanze multiple dell'oggetto `PaymentStrategy` per ogni sessione utente.

Per cambiare questo comportamento si può aggiungere un'annotazione `@SessionScoped` al metodo.

```
@Produces @Preferred @SessionScoped
public PaymentStrategy getPaymentStrategy() {
    ...
}
```

Ora, quando il metodo produttore viene chiamato, il `PaymentStrategy` restituito verrà associato al contesto di sessione. Il metodo produttore non verrà più chiamato nella stessa sessione.

## 6.2. Iniezione nei metodi produttori

C'è un potenziale problema con il codice visto sopra. Le implementazioni di `CreditCardPaymentStrategy` vengono istanziate usando l'operatore Java `new`. Gli oggetti istanziati direttamente dall'applicazione non possono sfruttare la dependency injection e non hanno interceptor.

Se questo non è ciò che si vuole, è possibile usare la dependency injection nel metodo produttore per ottenere istanze Web Bean:

```
@Produces @Preferred @SessionScoped
public PaymentStrategy getPaymentStrategy(CreditCardPaymentStrategy ccps,
                                          ChequePaymentStrategy cps,
                                          PayPalPaymentStrategy ppps) {
   switch (paymentStrategy) {
      case CREDIT_CARD: return ccps;
      case CHEQUE: return cps;
      case PAYPAL: return ppps;
      default: return null;
   }
}
```

Ma cosa succede se `CreditCardPaymentStrategy` è un Web Bean con scope di tipo richiesta? Il metodo produttore ha l'effetto di "promuovere" l'istanza corrente con scope di tipo richiesta a scope di tipo sessione. Questo è quasi certamente un bug! L'oggetto con scope richiesta verrà distrutto dal manager Web Bean prima che la sessione termini. Quest'errore *non* verrà rilevato dal manager Web Bean, quindi si faccia attenzione quando si restituiscono istanze Web Bean dai metodi produttori!

Ci sono almeno 3 modi per correggere questo bug. Si può cambiare lo scope dell'implementazione di `CreditCardPaymentStrategy`, ma questo non influenzerebbe gli altri client di questo Web Bean. Un'opzione migliore sarebbe quella di cambiare lo scope del metodo produttore a `@Dependent` o `@RequestScoped`.

Ma una soluzione più comune è quella di usare la speciale annotazione di binding `@New`.

## 6.3. Uso di `@New` con i metodi produttori

Si consideri il seguente metodo produttore:

```
@Produces @Preferred @SessionScoped
public PaymentStrategy getPaymentStrategy(@New CreditCardPaymentStrategy ccps,
                                          @New ChequePaymentStrategy cps,
                                          @New PayPalPaymentStrategy ppps) {
   switch (paymentStrategy) {
      case CREDIT_CARD: return ccps;
      case CHEQUE: return cps;
      case PAYPAL: return ppps;
      default: return null;
   }
```

```
}
```

Quindi verrebbe creata una nuova istanza *dipendente* di `CreditCardPaymentStrategy`, passata al metodo produttore, ritornata al metodo produttore ed infine associata al contesto di sessione. L'oggetto dipendente non verrebbe distrutto finché l'oggetto `Preferences` non viene distrutto, cioè a fine sessione.

# Parte II. Sviluppare codice debolmente-accoppiato

Il primo tema saliente di Web Beans è l'*accoppiamento debole (loose coupling)*. Abbiamo già visto tre modi per realizzarlo:

- i *tipi di deployment* rendono possibile il polimorfismo a deployment time,
- i *metodi produttori* rendono possibile il polimorfismo a runtime, e
- *la gestione contestuale del ciclo di vita* disaccoppia i cicli di vita dei Web Bean

Queste tecniche servono a realizzare l'accoppiamento debole (loose coupling) di client e server. Il client non è più strettamente legato all'implementazione di una API, né è tenuto a gestire il ciclo di vita dell'oggetto server. Questo approccio permette *agli oggetti stateful di interagire come se fossero servizi*.

L'accoppiamento debole (loose coupling) rende un sistema più *dinamico*. Il sistema può rispondere ai cambiamenti in un modo ben definito. In passato, i framework che hanno cercato di fornire le funzionalità e gli strumenti sopraelencati, l'hanno puntualmente fatto a discapito della sicurezza dei tipi (type safety). Web Beans è la prima tecnologia a raggiungere questo livello di puntualmente accoppiamento debole (loose coupling) in modo sicuro rispetto all'uso dei tipi.

Web Beans fornisce tre strumenti extra importanti che ampliano l'obiettivo del loose coupling:

- gli *interceptor* disaccoppiano i problemi tecnici dalla business logic,
- i *decoratori* possono essere usati per disaccoppiare alcuni problemi relativi alla business logic, e
- le *notifiche degli eventi* disaccoppiano i produttori di eventi dai consumatori

Innanzitutto esploriamo gli interceptor.

# Gli interceptor

Web Beans riutilizza l'architettura base degli interceptor di EJB3.0, estendendo la funzionalità in due direzioni:

- Qualsiasi Web Bean può avere interceptor, non solo i session bean.
- Web Bean fornisce un più sofisticato approccio basato su annotazioni per associare interceptor ai Web Beans.

"La specifica Web Bean definisce due tipi di punti di intercettazione:

- intercettazione del metodo di business, e
- intercettazione della chiamata del ciclo di vita

Un *interceptor di un metodo di business* si applica alle invocazioni di metodi del Web Bean da parte di client del Web Bean:

```
public class TransactionInterceptor {
   @AroundInvoke public Object manageTransaction(InvocationContext ctx) { ... }
}
```

Un *interceptor di chiamata del ciclo di vita* si applica alle invocazioni delle chiamate del ciclo di vita da parte del container:

```
public class DependencyInjectionInterceptor {
   @PostConstruct public void injectDependencies(InvocationContext ctx) { ... }
}
```

Una classe interceptor può intercettare entrambi le chiamate del ciclo di vita ed i metodi di business.

## 7.1. Interceptor bindings

Si supponga di voler dichiarare transazionali alcuni Web Beans. La prima cosa necessaria è un'*annotazione di interceptor binding* per specificare esattamente quali Web Beans sono interessati:

```
@InterceptorBindingType
@Target({METHOD, TYPE})
```

```
@Retention(RUNTIME)
public @interface Transactional {}
```

Ora è facilmente possibile specificare che `ShoppingCart` è un oggetto transazionale:

```
@Transactional
public class ShoppingCart { ... }
```

O se si preferisce, si può specificare che solo un metodo sia transazionale:

```
public class ShoppingCart {
   @Transactional public void checkout() { ... }
}
```

## 7.2. Implementare gli interceptor

Bene, ma da qualche parte è necessario implementare l'interceptor che fornisce l'aspect di gestione della transazione. Occore quindi creare un interceptor EJB standard e annotarlo con `@Interceptor` e `@Transactional`."

```
@Transactional @Interceptor
public class TransactionInterceptor {
   @AroundInvoke public Object manageTransaction(InvocationContext ctx) { ... }
}
```

Tutti gli interceptor dei Web Beans sono Web Beans semplici e possono sfruttare la dependency injection e la gestione del ciclo di vita contestuale.

```
@ApplicationScoped @Transactional @Interceptor
public class TransactionInterceptor {

   @Resource Transaction transaction;

   @AroundInvoke public Object manageTransaction(InvocationContext ctx) { ... }

}
```

Diverso interceptor possono usare lo stesso tipo di interceptor binding.

## 7.3. Abilitare gli interceptor

Infine occorre *abilitare* l'interceptor in `web-beans.xml`.

```
<Interceptors>
   <tx:TransactionInterceptor/>
</Interceptors
>
```

Ma perché viene usato ancora XML, quando Web Beans non dovrebbe utilizzarlo?

La dichiarazione XML risolve due problemi:

- Ci consente di specificare un ordinamento totale per tutti gli interceptor del sistema, assicurando un comportamente deterministico, e
- consente di abilitare o disabilitare le classi di interceptor a deployment time

Per esempio è possibile specificare che l'interceptor di sicurezza venga eseguito prima di `TransactionInterceptor`."

```
<Interceptors>
   <sx:SecurityInterceptor/>
   <tx:TransactionInterceptor/>
</Interceptors
>
```

Oppure si può disattivarli entrambi dal proprio ambiente di test!

## 7.4. Interceptor binding con membri

Si supponga di voler aggiungere qualche informazione extra all'annotazione `@Transactional`:

```
@InterceptorBindingType
@Target({METHOD, TYPE})
@Retention(RUNTIME)
public @interface Transactional {
   boolean requiresNew() default false;
}
```

Web Beans utilizzerà il valore di `requiresNew` per scegliere tra due diversi interceptor, `TransactionInterceptor` e `RequiresNewTransactionInterceptor`.

```
@Transactional(requiresNew=true) @Interceptor
public class RequiresNewTransactionInterceptor {
   @AroundInvoke public Object manageTransaction(InvocationContext ctx) { ... }
}
```

Ora è possibile usare `RequiresNewTransactionInterceptor` in questo modo:

```
@Transactional(requiresNew=true)
public class ShoppingCart { ... }
```

Ma cosa succede se si ha solo un interceptor e si vuole che il manager ignori il valore di `requiresNew` quando si associa l'interceptor? Si può usare l'annotazione `@NonBinding`:

```
@InterceptorBindingType
@Target({METHOD, TYPE})
@Retention(RUNTIME)
public @interface Secure {
   @NonBinding String[] rolesAllowed() default {};
}
```

## 7.5. Annotazioni per interceptor binding multipli

Solitamente si usano combinazioni di tipi di interceptor binding per associare pià interceptor ad un Web Bean. Per esempio, la seguente dichiarazione verrebbe impiegata per associare `TransactionInterceptor` e `SecurityInterceptor` allo stesso Web Bean:

```
@Secure(rolesAllowed="admin") @Transactional
public class ShoppingCart { ... }
```

Comunque in casi molto complessi un interceptor da solo potrebbe specificare alcune combinazioni di tipi di interceptor binding:

```
@Transactional @Secure @Interceptor
public class TransactionalSecureInterceptor { ... }
```

Allora quest'interceptor potrebbe venire associato al metodo `checkout()` usando una delle seguenti combinazioni:

```
public class ShoppingCart {
   @Transactional @Secure public void checkout() { ... }
}
```

```
@Secure
public class ShoppingCart {
   @Transactional public void checkout() { ... }
}
```

```
@TransactionI
public class ShoppingCart {
   @Secure public void checkout() { ... }
}
```

```
@Transactional @Secure
public class ShoppingCart {
   public void checkout() { ... }
}
```

## 7.6. Ereditarietà del tipo di interceptor binding

Una limitazione del supporto del linguaggio Java per le annotazioni è la mancanza di ereditarietà delle annotazioni. In verità le annotazioni dovrebbero avere il riutilizzo predefinito per consentire che questo avvenga:

```
public @interface Action extends Transactional, Secure { ... }
```

Fortunatamente Web Beans provvede a questa mancanza di Java. E' possibile annotare un tipo di interceptor binding con altri tipi di interceptor binding. Gli interceptor binding sono transitivi # qualsiasi Web Bean con il primo interceptor binding eredita gli interceptor binding dichiarati come meta-annotazioni.

```
@Transactional @Secure
@InterceptorBindingType
@Target(TYPE)
@Retention(RUNTIME)
```

```
public @interface Action { ... }
```

Ogni Web Bean annotato con `@Action` verrà legato ad entrambi `TransactionInterceptor` e `SecurityInterceptor`. (E anche `TransactionalSecureInterceptor`, se questo esiste.)

## 7.7. Uso di `@Interceptors`

L'annotazione `@Interceptors` definita dalla specifica EJB è supportata per entrambi i Web Bean semplici ed enterprise, per esempio:

```
@Interceptors({TransactionInterceptor.class, SecurityInterceptor.class})
public class ShoppingCart {
   public void checkout() { ... }
}
```

Comunque, quest'approccio soffre dei seguenti difetti:

- l'implementazione degli interceptor è codificata nel codice di business,
- gli interceptor possono non essere facilmente disabilitati a deployment time, e
- l'ordinamento degli interceptor è non-globale # è determinata dall'ordine in cui gli interceptor sono elencati al livello di classe.

Quindi si raccomanda l'uso di interceptor binding di stile Web Beans.

# Decoratori

Gli interceptor sono un potente modo per catturare e separare i concern (N.d.T. un concern è un particolare concetto o area di interesse) che sono *ortogonali* al sistema tipo. Qualsiasi interceptor è capace di intercettare le invocazioni di qualsiasi tipo Java. Questo li rende perfetti per risolvere concern tecnici quali gestione delle transazioni e la sicurezza. Comunque, per natura, gli interceptor non sono consapevoli dell'attuale semantica degli eventi che intercettano. Quindi gli interceptor non sono il giusto strumento per separare i concern di tipo business.

Il contrario è vero per i *decoratori*. Un decoratore intercetta le invocazioni solamente per una certa interfaccia Java, e quindi è consapevole della semantica legata a questa. Ciò rende i decoratori uno strumento perfetto per modellare alcuni tipi di concern di business. E significa pure che un decoratore non ha la generalità di un interceptor. I decoratori non sono capaci di risolvere i concern tecnici che agiscono per diversi tipi.

Supponiamo di avere un'interfaccia che rappresenti degli account:

```
public interface Account {
   public BigDecimal getBalance();
   public User getOwner();
   public void withdraw(BigDecimal amount);
   public void deposit(BigDecimal amount);
}
```

Parecchi Web Beans del nostro sistema implementano l'interfaccia `Account`. Abbiamo come comune requisito legale, per ogni tipo di account, che le transazioni lunghe vengano registrate dal sistema in uno speciale log. Questo è un lavoro perfetto per un decoratore.

Un decorator è un semplice Web Beans che implementa il tipo che decora ed è annotato con `@Decorator`."

```
@Decorator
public abstract class LargeTransactionDecorator
      implements Account {

   @Decorates Account account;

   @PersistenceContext EntityManager em;

   public void withdraw(BigDecimal amount) {
      account.withdraw(amount);
      if ( amount.compareTo(LARGE_AMOUNT)
>0 ) {
```

```
            em.persist( new LoggedWithdrawl(amount) );
        }
    }

    public void deposit(BigDecimal amount);
        account.deposit(amount);
        if ( amount.compareTo(LARGE_AMOUNT)
>0 ) {
            em.persist( new LoggedDeposit(amount) );
        }
    }

}
```

Diversamente dai semplici Web Beans, un decoratore può essere una classe astratta. Se un decoratore non ha niente da fare per un particolare metodo, allora non occorre implementare quel metodo.

## 8.1. Attributi delegate

Tutti i decoratori hanno un *attributo delegato*. Il tipo ed i tipi di binding dell'attributo delegato determinano a quali Web Beans è legato il decoratore. Il tipo di attributo delegato deve implementare o estendere tutte le interfacce implementate dal decoratore.

Quest'attributo delegate specifica che ildecorator è legao a tutti i Web Beans che implementano `Account`:

```
@Decorates Account account;
```

Un attributo delegato può specificare un'annotazione di binding. E quindi il decoratore verrà associato a Web Beans con lo stesso binding.

```
@Decorates @Foreign Account account;
```

Un decorator è legato ad un qualsiasi Web Bean che:

- ha il tipo di attributo delegate come un tipo API, e
- ha tutti i tipi di binding che sono dichiarati dall'attributo delegate.

Il decoratore può invocare l'attributo delegate, il ché ha lo stesso effetto come chiamare `InvocationContext.proceed()` da un interceptor.

## 8.2. Abilitare i decoratori

Occorre *abilitare* il decoratore in `web-beans.xml`.

```
<Decorators>
   <myapp:LargeTransactionDecorator/>
</Decorators
>
```

Per i decoratori questa dichiarazione provvede alle stesse finalità di quanto la dichiarazione `<Interceptors>` fa per gli interceptor.

- Consente di specificare un ordinamento totale per tutti i decoratori del sistema, assicurando un comportamento deterministico, e
- consente di abilitare o disabilitare le classi decorato durante la fase di deploy.

Gli interceptor per un metodo sono chiamati prima dei decoratori che vengono applicati a tali metodo.

# Eventi

Il sistema di notifica a eventi di Web Beans consente a Web Beans di interagire in maniera totalmente disaccoppiata. I *produttori* di eventi sollevano eventi che vengono consegnati agli *osservatori* di eventi tramite il manager Web Bean. Lo schema base può suonare simile al familiare pattern observer/observable, ma ci sono un paio di differenze:

- non solo i produttori di eventi sono disaccoppiati dagli osservatori; gli osservatori sono completamente disaccoppiati dai produttori,
- gli osservatori possono specificare una combinazione di "selettori" per restringere il set di notifiche di eventi da ricevere, e
- gli osservatori possono essere notificati immediatamente, o possono specificare che la consegna degli eventi venga ritardata fino alla fine della transazione conrrente

## 9.1. Osservatori di eventi

Un *metodo osservatore* è un metodo di un Web Bean con un parametro annotato `@Observes`.

```
public void onAnyDocumentEvent(@Observes Document document) { ... }
```

Il parametro annotato viene chiamato *parametro evento*. Il tipo di parametro evento è il *tipo evento* osservato. I metodi osservatori possono anche specificare dei "selettori", che sono solo istanze di tipi di binding di Web Beans. Quando un tipo di binding viene usato come selettore di eventi viene chiamato *tipo binding di evento*.

```
@BindingType
@Target({PARAMETER, FIELD})
@Retention(RUNTIME)
public @interface Updated { ... }
```

Specifichiamo i binding di evento del metodo osservatore annotando il parametro evento:

```
public void afterDocumentUpdate(@Observes @Updated Document document) { ... }
```

Un metodo osservatore non ha bisogno di specificare alcun binding di evento # in questo caso è interessato a *tutti* gli eventi di un particolare tipo. Se specifica dei binding di evento, è solamente interessato agli eventi che hanno anche gli stessi binding di evento.

Il metodo osservatore può avere parametri addizionali che vengono iniettati secondo la solita semantica di iniezione del parametro del metodo Web Beans.

```
public void afterDocumentUpdate(@Observes @Updated Document document, User user) { ... }
```

## 9.2. Produttori di eventi

Il produttore dell'evento può ottenere tramite iniezione un oggetto *notificatore d'evento*:

```
@Observable Event<Document
> documentEvent
```

L'annotazione `@Observable` definisce implicitamente un Web Bean con scope `@Dependent` e tipo di deploy `@Standard`, con un'implementazione fornita dal manager Web Bean.

Un produttore solleva eventi chiamando il metodo `fire()` dell'intefaccia `Event`, passando un *oggetto evento*:

```
documentEvent.fire(document);
```

Un oggetto evento può essere un'istanza di una classe Java che non ha variabili tipo o parametri tipo wildcard. L'evento verrà consegnato ad ogni metodo osservatore che:

- ha un parametro evento a cui l'oggetto evento è assegnabile, e
- non specifica binding d'evento.

Il manager Web Bean chiama semplicemente tutti i metodi osservatori, passando l'oggento evento come valore del parametro evento. Se il metodo osservatore lancia un'eccezione, il manager Web Bean smette di chiamare i metodi osservatori, e l'eccezione viene rilanciata dal metodo `fire()`.

Per specificare un "selettore" il produttore d'evento può passare un'istanza del tipo di binding d'evento al metodo `fire()`:

```
documentEvent.fire( document, new AnnotationLiteral<Updated
>(){} );
```

La classe helper `AnnotationLiteral` rende possibile istanziare inline i tipi di binding, dato che questo risulta difficile da fare in Java.

L'evento verrà consegnato ad ogni metodo osservatore che:

- ha un parametro evento a cui l'oggetto evento è assegnabile, e
- non specifica alcun event binding *tranne* per gli event binding passati a `fire()`.

In alternativa gli event binding possono essere specificati annotando il punto di iniezione del notificato d'evento:

```
@Observable @Updated Event<Document
> documentUpdatedEvent
```

Quindi ciascun evento sollevato tramite quest'istanza di `Event` ha annotato l'event binding. L'evento verrà consegnato ad ogni metodo osservatore che:

- ha un parametro evento a cui l'oggetto evento è assegnabile, e
- non specifica alcun event binding *tranne* per gli event binding passati a `fire()` o per gli event binding annotati del punto di iniezione del notificatore eventi.

## 9.3. Registrare dinamicamente gli osservatori

E' spesso utile registrare un osservatore d'evento in modo dinamico. L'applicazione può implementare l'interfaccia `Observer` e registrare un'istanza con un notificatore d'evento chiamando il metodo `observe()`.

```
documentEvent.observe( new Observer<Document
>() { public void notify(Document doc) { ... } } );
```

I tipi di event binding possono essere specificati dal punto di iniezione del notificatore d'eventi o passando istance del tipo di event binding al metodo `observe()`:

```
documentEvent.observe( new Observer<Document
>() { public void notify(Document doc) { ... } },
                                    new AnnotationLiteral<Updated
>(){} );
```

## 9.4. Event binding con membri

Un tipo di event binding può avere membri annotati:

```
@BindingType
@Target({PARAMETER, FIELD})
@Retention(RUNTIME)
```

```
public @interface Role {
   RoleType value();
}
```

Il valore del membro è usato per restringere i messaggi consegnati all'osservatore:

```
public void adminLoggedIn(@Observes @Role(ADMIN) LoggedIn event) { ... }
```

I membri del tipo di eventbinding possono essere specificati staticamente dal produttore di eventi tramite annotazioni nel punto di iniezione del notificatore d'evento:

```
@Observable @Role(ADMIN) Event<LoggedIn
> LoggedInEvent;}}
```

Alternativamente il valore del membro del tipo di event binding può esserre dinamicamente determinato dal produttore di eventi. Iniziamo scrivendo una sottoclasse astratta di `AnnotationLiteral`:

```
abstract class RoleBinding
   extends AnnotationLiteral<Role
>
   implements Role {}
```

Il produttore di eventi passa un'istanza di questa classe a `fire()`:

```
documentEvent.fire( document, new RoleBinding() { public void value() { return user.getRole();
 } } );
```

## 9.5. Event binding multipli

I tipi di event binding possono essere combinati, per esempio:

```
@Observable @Blog Event<Document
> blogEvent;
...
if (document.isBlog()) blogEvent.fire(document, new AnnotationLiteral<Updated
>(){});
```

Quando si genera un evento, tutti i seguenti metodi osservatori verranno notificati:

```
public void afterBlogUpdate(@Observes @Updated @Blog Document document) { ... }
```

```
public void afterDocumentUpdate(@Observes @Updated Document document) { ... }
```

```
public void onAnyBlogEvent(@Observes @Blog Document document) { ... }
```

```
public void onAnyDocumentEvent(@Observes Document document) { ... }}}
```

## 9.6. Osservatori transazionali

Gli osservatori transazionali ricevono notifiche d'evento prima o dopo la fase di completamento della transazione, nella quale l'evento viene sollevato. Per esempio, il seguente metodo osservatore ha bisogno di aggiornare il set di risultati della query memorizzato nel contesto dell'applicazione, ma solo quando hanno successo le transazioni che aggiornano l'albero `Category`.

```
public void refreshCategoryTree(@AfterTransactionSuccess @Observes CategoryUpdateEvent
 event) { ... }
```

Ci sono tre tipi di osservatori transazionali:

- gli osservatori `@AfterTransactionSuccess` vengono chiamati dopo la fase di completamento della transazione, ma solo se questa si completa con successo
- gli osservatori `@AfterTransactionFailure` vengono chiamati dopo la fase di completamento della transazione, ma solo se questa fallisce e quindi non completa con successo
- gli osservatori `@AfterTransactionCompletion` vengono chiamati dopo la fase di completamento della transazione
- gli osservatori `@BeforeTransactionCompletion` vengono chiamati prima della fase di completamento della transazione

Gli osservatori transazionali sono molto importanti in un modello ad oggetto stateful come Web Beans, poiché lo stato è spesso mantenuto per un tempo più lungo di una singola transazione atomica.

Si immagini di avere cachato un risultato di query JPA nello scope di applicazione:

```
@ApplicationScoped @Singleton
public class Catalog {

   @PersistenceContext EntityManager em;

   List<Product
> products;

   @Produces @Catalog
   List<Product
> getCatalog() {
      if (products==null) {
         products = em.createQuery("select p from Product p where p.deleted = false")
            .getResultList();
      }
      return products;
   }

}
```

Di tanto in tanto un `Product` viene creato o cancellato. Quando questo avviene occorre aggiornare il catalogo del `Product`. Ma si dovrebbe aspettare che la transazione *abbia completato* con successo prima di eseguire l'aggiornamento!

Il Web Bean che crea o cancella `Product` può sollevare eventi, per esempio:

```
@Stateless
public class ProductManager {

   @PersistenceContext EntityManager em;
   @Observable Event<Product
> productEvent;

   public void delete(Product product) {
      em.delete(product);
      productEvent.fire(product, new AnnotationLiteral<Deleted
>(){});
   }

   public void persist(Product product) {
      em.persist(product);
      productEvent.fire(product, new AnnotationLiteral<Created
>(){});
```

```
    }

    ...

}
```

E ora `Catalog` può osservare gli eventi dopo il completamento (con successo) della transazione:

```
@ApplicationScoped @Singleton
public class Catalog {

    ...

    void addProduct(@AfterTransactionSuccess @Observes @Created Product product) {
        products.add(product);
    }

    void addProduct(@AfterTransactionSuccess @Observes @Deleted Product product) {
        products.remove(product);
    }

}
```

# Parte III. Realizzare una tipizzazione piÃ¹ forte

Il secondo tema saliente di Web Beans è lo *tipizzazione forte (strong typing)*. Le informazioni riguardanti dipendenze, interceptor e decorator di un Web Bean, e le informazioni sui i consumatori relativi ad un produttore di eventi, sono contenute in costrutti Java sicuri rispetto ai tipi (typesafe) che possono essere validati dal compilatore.

Non si vedono identificatori di tipo stringa nel codice basato su Web Beans, non perché il framework li nasconde usando regole intelligenti nell'assegnamento dei valori di default # la cosiddetta "configurazione per convenzione (configuration by convention)" # ma semplicemente perché non ci sono stringhe, tanto per cominciare!

L'ovvio beneficio di questo approccio è che *qualunque* IDE può fornire autocompletamento, validazione e refactoring senza che sia necessario realizzare dei tool appositi. Ma c'è un secondo beneficio meno immediatamente ovvio. Si scopre che quando si incomincia a pensare di identificare oggetti, eventi o interceptor usando annotazioni invece di nomi, si ha l'opportunità di elevare il livello semantico del proprio codice.

Web Beans incoraggia a sviluppare annotazioni che modellano concetti, per esempio,

- `@Asynchronous`,
- `@Mock`,
- `@Secure` or
- `@Secure` or

invece di usare nomi composti come

- `asyncPaymentProcessor`,
- `mockPaymentProcessor`,
- `SecurityInterceptor` or
- `DocumentUpdatedEvent`.

Le annotazioni sono riutilizzabili. Aiutano a descrivere caratteristiche comuni di parti diverse del sistema. Ci aiutano a categorizzare e comprendere il nostro stesso codice. Ci aiutano ad affrontare i concern comuni in un modo comune. Rendono il nostro codice più elegante e comprensibile.

Gli *stereotipi (stereotypes)* di Web Beans fanno fare un ulteriore passo in avanti a questa idea. Uno stereotipo descrive un *ruolo* comune nell'architettura di un'applicazione. Incapsula in un unico

pacchetto riutilizzabile varie proprietà del ruolo stesso, inclusi lo scope, gli interceptor bindings, il tipo di deployment, etc, .

Persino i metadati XML di Web Beans sono fortemente tipizzati (strongly typed)! Non esistendo un compilatore XML, Web Beans si basa sugli schemi XML per validare i tipi Java e gli attributi che compaiono nell'XML. Questo approccio finisce col rendere il codice XML più informato, proprio come le annotazioni rendono il codice Java + informato.

Ora siamo pronti ad incontrare alcune caratteristiche più avanzate di Web Beans. Tenete a mente che tali caratteristiche esistono sia per rendere il nostro codice più facile da validare che per renderlo più comprensibile. La maggior parte delle volte non è necessario usare tali carattestiche, ma, se usate con accortezza, si arriverà ad apprezzerne l'efficacia.

# Stereotipi

Secondo la specifica Web Beans:

> In molti sistemi l'uso di pattern architetturali produce un set di ruoli Web Bean ricorrenti. Uno stereotipo consente allo sviluppatore di framework di identificare tale ruolo e di dichiarare alcuni metadati comuni per i Web Bean con tale ruolo come parte principale.
>
> Uno stereotipo incapsula qualsiasi combinazione di:
>
> - un tipo di deploy di default,
> - un tipo di scope di default,
> - una restrizione sullo scope del Web Bean,
> - un requisito che il Web Bean implementi o estenda un certo tipo, e
> - un set di annotazioni di interceptor binding.
>
> Uno stereotipo può anche specificare che tutti i Web Beans con tale stereotipo abbiano nomi Web Bean di default.
>
> Un Web Bean può dichiarare zero, uno o più stereotipi.

Uno stereotipo è un tipo di annotazione Java. Questo stereotipo identifica le classi di azione in alcuni framework MVC:

```
@Retention(RUNTIME)
@Target(TYPE)
@Stereotype
public @interface Action {}
```

Lo stereotipo viene impiegato applicando l'annotazione al Web Bean.

```
@Action
public class LoginAction { ... }
```

## 10.1. Scope di default e tipo di deploy per uno stereotipo

Uno stereotipo può specificare uno scope di default e/o un tipo di deploy di default per Web Bean con tale stereotipo. Per esempio, se il tipo di deploy `@WebTier` identifica Web Bean

che dovrebbero essere deployati quando il sistema viene eseguito come applicazione web, si potrebbero specificare i seguenti valori di default per le classi d'azione:

```
@Retention(RUNTIME)
@Target(TYPE)
@RequestScoped
@WebTier
@Stereotype
public @interface Action {}
```

Certamente una particolare azione può comunque fare l'override di questi valore se necessario:

```
@Dependent @Mock @Action
public class MockLoginAction { ... }
```

Se si vuole forzare tutte le azioni ad un particolare scope, è possibile.

## 10.2. Restringere lo scope ed il tipo con uno stereotipo

Si supponga di voler prevenire le azioni dal dichiarare certi scope. Web Beans consente esplicitamente di specificare un set di scope consentiti per Web Bean con certi stereotipi. Per esempio:

```
@Retention(RUNTIME)
@Target(TYPE)
@RequestScoped
@WebTier
@Stereotype(supportedScopes=RequestScoped.class)
public @interface Action {}
```

Se una particolare classe d'azione tenta di specificare uno scope diverso dallo scope di richiesta Web Bean, verrà lanciata un'eccezione dal manager Web Bean in fase di inizializzazione.

Tutti i Web Bean con certi stereotipi possono venire forzati ad implementare un'interfaccia o ad estendere una classe:

```
@Retention(RUNTIME)
@Target(TYPE)
@RequestScoped
@WebTier
```

```
@Stereotype(requiredTypes=AbstractAction.class)
public @interface Action {}
```

Se una particolare classe d'azione non estende la classe `AbstractAction`, verrà lanciata un'eccezione dal manager Web Bean in fase di inizializzazione.

## 10.3. Interceptor binding per gli stereotipi

Uno stereotipo può specificare un set di interceptor binding da ereditare da tutti i Web Bean con quello stereotipo.

```
@Retention(RUNTIME)
@Target(TYPE)
@RequestScoped
@Transactional(requiresNew=true)
@Secure
@WebTier
@Stereotype
public @interface Action {}
```

Questo aiuta ad ottenere concern tecnici anche lontano dal codice di business!

## 10.4. Assegnare nomi di default con gli stereotipi

Infine è possibile specificare che tutti i Web Bean con certi stereotipi abbiamo un certo nome Web Bean, messo di default dal manager Web Bean. Le azioni sono spesso referenziate nelle pagine JSP, e quindi sono un caso d'uso perfetto per questa funzionalità. Basta aggiungere un'annotazione vuota `@Named`:

```
@Retention(RUNTIME)
@Target(TYPE)
@RequestScoped
@Transactional(requiresNew=true)
@Secure
@Named
@WebTier
@Stereotype
public @interface Action {}
```

Ora, `LoginAction` avrà nome `loginAction`.

## 10.5. Stereotipi standard

Si sono già visti due stereotipi standard definiti dalla specifica Web Bean: `@Interceptor` e `@Decorator`.

Web Bean definisce un ulteriore stereotipo standard:

```
@Named
@RequestScoped
@Stereotype
@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface Model {}
```

Questo stereotipo è inteso per l'uso con JSF. Invece di usare bean gestiti da JSF, basta annotare un Web Bean con `@Model`, e usarlo direttamente nelle pagine JSF.

# Specializzazione

Si è già visto come il modello di dependency injection di Web Beans consenta l'*override* dell'implementazione di un API a deployment time. Per esempio, il seguente Web Bean enterprise fornisce un'implementazione di un API `PaymentProcessor` in produzione:

```
@CreditCard @Stateless
public class CreditCardPaymentProcessor
      implements PaymentProcessor {
   ...
}
```

Ma in quest'ambiente di prova si procede con l'override dell'implementazione di `PaymentProcessor` con un Web Bean differente:

```
@CreditCard @Stateless @Staging
public class StagingCreditCardPaymentProcessor
      implements PaymentProcessor {
   ...
}
```

Quello che si è cercato di fare con `StagingCreditCardPaymentProcessor` è sostituire completamente `AsyncPaymentProcessor` in un particolare deployment del sistema. In tale deployment, il tipo di deploy `@Staging` avrebbe più alta priorità del tipo di deploy di default `@Production`, e quindi client con il seguente punto di iniezione:

```
@CreditCard PaymentProcessor ccpp
```

riceverebbero un'istanza di `StagingCreditCardPaymentProcessor`.

Sfortunatamente ci sono parecchie trappole in cui è facile cadere:

- il Web Bean con più alta priorità potrebbe non implementare tutti i tipi di API del Web Bean di cui tenta di fare override,
- il Web Bean con più alta priorità potrebbe non dichiare tutti i tipi di binding del Web Bean di cui tenta di fare override,
- il Web Bean con più alta priorità potrebbe non avere lo stesso nome del Web Bean di cui tenta di fare override, oppure

- il Web Bean di cui tenta di fare override potrebbe dichiarare un metodo produttore, metodo distruttore o metodo osservatore.

In ciascuno di questi casi, il Web Bean di cui si tenta l'override potrebbe ancora venire chiamato a runtime. Quindi, l'override è qualcosa di incline all'errore per lo sviluppatore.

Web Beans fornisce una funzionalità speciale chiamata *specializzazione*, che aiuta lo sviluppatore ad evitare queste trappole. La specializzazione sembra inizialmente un pò esoterica, ma in pratica è facile da usare, e si apprezzerà veramente la sicurezza extra che fornisce.

## 11.1. Usare la specializzazione

La specializzazione è una funzionalità specifica dei Web Bean semplici ed enterprise. Per fare uso della specializzazione il Web Bean a più alta priorità deve:

- essere un diretta sottoclasse del Web Bean di cui fa l'override, e
- essere un semplice Web Bean se il Web Bean di cui fare override è un semplice Web Bean o un Web Bean Enterprise se il Web Bean di cui fa override è un Web Bean Enterprise, e
- essere annotato con `@Specializes`.

```
@Stateless @Staging @Specializes
public class StagingCreditCardPaymentProcessor
      extends CreditCardPaymentProcessor {
   ...
}
```

Si dice che il Web Bean a più alta priorità *specializza* la sua superclasse.

## 11.2. Vantaggi della specializzazione

Quando viene usata la specializzazione:

- i tipi di binding della superclasse vengono automaticamente ereditati dal Web Bean annotato con `@Specializes`, e
- il nome del Web Bean della superclasse è automaticamente ereditato dal Web Bean annotato con `@Specializes`, e
- i metodi produttori, metodi distruttori e metodi osservatori dichiarati dalla superclasse vengono chiamati da un'istanza del Web Bean annotato con `@Specializes`.

Nel nostro esempio, il tipo di binding `@CreditCard` di `CreditCardPaymentProcessor` viene ereditato da `StagingCreditCardPaymentProcessor`.

Inoltre il manager Web Bean validerà che:

- tutti i tipi di API della superclasse sono tipi di API del Web Bean annotato con `@Specializes` (tutte le interfacce locali del bean enterprise della superclasse sono anch'essi interfacce della sottoclasse),
- il tipo di deploy di un Web Bean annotato con `@Specializes` ha una precedenza più alta di un tipo di deploy della superclasse, e
- non esiste nessun altro Web Bean che specializza la superclasse.

Se una di queste condizioni viene violata, il manager Web Bean lancia un'eccezione in fase di inizializzazione.

Quindi, si può essere certi che la superclasse non verrà *mai* chiamata in alcun deploy del sistema in cui viene deployato ed abilitato il Web Bean annotato con `@Specializes`.

# Definire i Web Beans tramite XML

Finora si sono visti molti esempi di Web Bean dichiarati usando annotazioni. Comunque ci sono varie occasioni in cui non è possibile usare le annotazioni per definire un Web Bean:

- quando la classe d'implementazione proviene da qualche libreria preesistente, o
- quando devrebbero esserci Web Beans multipli con la stessa classe d'implementazione.

In entrambi i casi Web Beans fornisce due opzioni:

- scrivere un metodo produttore, o
- dichiarare il Web Bean usando XML.

Molti framework utilizzano XML per scrivere metadati relazionati alle classi. Web Beans usa un approccio molto diverso rispetto agli altri framework per specificare i nomi delle classi Java, dei campi o dei metodi. Invece di scrivere i nomi delle classi o dei membri come valori stringa di elementi e atttributi XML, Web Beans consente di utilizzare il nome della classe o del membro come nome dell'elemento XML.

Il vantaggio di quest'approccio è che risulta possibile scrivere uno schemaXML che previene gli errori di scrittura nei documenti XML. E' comunque possibile per un tool generare lo schema XML in modo automatico dal codice Java compilato. Oppure un ambiente di sviluppo integrato può eseguire la stessa validazionesenza il bisogno di un passo di generazione intermedio ed esplicito.

## 12.1. Dichiarare classi Web Bean

Per ogni pacchetto Java, Web Beans definisce un corrispondente namespace XML. Ilnamespace è formato aggiungendo il prefisso `urn:java:` al nomedel pacchetto Java. Per il pacchetto `com.mydomain.myapp`, il namespace XML è `urn:java:com.mydomain.myapp`.

I tipi Java appartenenti al pacchetto vengono riferiti usando un elemento XML nel namespace corrispondente al pacchetto. Il nome dell'elemento è un nome di tipo Java. I campi e metodi del tipo vengono specificati dagli elementi figlio nello stesso namespace. Se il tipo è un'annotazione, i membri sonospecificati dagli attributi dell'elemento.

Per esempio l'elemento `<util:Date/>` nel seguente frammento XML si riferisce alla classe `java.util.Date`:

```
<WebBeans xmlns="urn:java:javax.webbeans"
      xmlns:util="urn:java:java.util">

   <util:Date/>
```

```
</WebBeans
>
```

E questo è tutto il codice per dichiarare che `Date` è un Web Bean semplice! Un'istanza di `Date` ora può essere iniettata da qualsiasi altro Web Bean:

```
@Current Date date
```

## 12.2. Dichiarare metadati Web Bean

E' possibile dichiarare lo scope, il tipo di deploy ed i tipi di binding degli interceptor usando elementi figli diretti della dichiarazione Web Bean:

```
<myapp:ShoppingCart>
   <SessionScoped/>
   <myfwk:Transactional requiresNew="true"/>
   <myfwk:Secure/>
</myapp:ShoppingCart
>
```

Si utilizza esattamente lo stesso approccio per specificare i nomi ed il tipo di binding:

```
<util:Date>
   <Named
>currentTime</Named>
</util:Date>

<util:Date>
   <SessionScoped/>
   <myapp:Login/>
   <Named
>loginTime</Named>
</util:Date>

<util:Date>
   <ApplicationScoped/>
   <myapp:SystemStart/>
   <Named
>systemStartTime</Named>
</util:Date
```

```
>
```

Dove `@Login` e `@SystemStart` sono tipi di annotazioni di binding.

```
@Current Date currentTime;
@Login Date loginTime;
@SystemStart Date systemStartTime;
```

Di nuovo un Web Bean può supportare tipi di binding multipli:

```
<myapp:AsynchronousChequePaymentProcessor>
   <myapp:PayByCheque/>
   <myapp:Asynchronous/>
</myapp:AsynchronousChequePaymentProcessor
>
```

Interceptor e decoratori sono solo eb Bean semplici, e quindi possono essere dichiarati come qualsiasi altro Web Bean semplice:

```
<myfwk:TransactionInterceptor>
   <Interceptor/>
   <myfwk:Transactional/>
</myfwk:TransactionInterceptor
>
```

## 12.3. Dichiarare membri Web Bean

DA FARE!

## 12.4. Dichiarazione inline dei Web Beans

I Web Beans consentono di definire un Web Bean ad un certo punto di iniezione. Per esempio:

```
<myapp:System>
   <ApplicationScoped/>
   <myapp:admin>
      <myapp:Name>
         <myapp:firstname
>Gavin</myapp:firstname>
```

```
        <myapp:lastname
>King</myapp:lastname>
        <myapp:email
>gavin@hibernate.org</myapp:email>
      </myapp:Name>
   </myapp:admin>
</myapp:System
>
```

L'elemento `<Name>` dichiara un Web Bean semplice con scope `@Dependent` e classe `Name`, con un set di valori di campo iniziali. Questo Web Bean ha uno speciale binding generatore dal container e quindi è iniettabile solo allo specifico punto di iniezione al quale è stato dichiarato.

Questa caratteristica semplice e potente consente formato XML di Web Bean di essere usato per specificare l'intero grafo di oggetti Java. Non è ancora una completa soluzione di databinding, ma ci si avvicina!

## 12.5. Uso di uno schema

Se si vuole che il formato di documento XML sia scritto da persone che non sono sviluppatori Java o che non hanno accesso al codice, occorre fornire uno schema. There's nothing specific to Web Beans about writing or using the schema.

```
<WebBeans xmlns="urn:java:javax.webbeans"
      xmlns:myapp="urn:java:com.mydomain.myapp"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:java:javax.webbeans http://java.sun.com/jee/web-beans-1.0.xsd
                        urn:java:com.mydomain.myapp http://mydomain.com/xsd/myapp-1.2.xsd">

   <myapp:System>
      ...
   </myapp:System>

</WebBeans
>
```

Scrivere unoschema XML è abbastanza noiso. Quindi il progetto Web Beans RI fornirà uno strumento per generare automaticamente lo schema XML dal codice Java compilato.

# Parte IV. Web Beans e l'ecosistema Java EE

Il terzo tema di Web Beans è *l'integrazione*. Web Beans è stata progettata per funzionare in armonia con altre tecnologie, per aiutare lo sviluppatore a far funzionare assieme le altre tecnologie. Web Beans è una tecnologia aperta. Costituisce una parte dell'ecosistema Java EE, ed è essa stessa la fondazione di un nuovo ecosistema di estensioni portabili e di integrazioni con framework e tecnologie esistenti.

Abbiamo già visto come Web Beans aiuti ad integrare EJB con JSF, permettendo agli EJB di essere legati direttamente alle pagine JSF. Questo non è che l'inizio. Web Beans offre le stesse potenzialità ad altre e diverse tecnologie, quali i motori di gestione di processi di business, altri Framework Web, e modelli a componenti di terze parti. La piattaforma Java EE non potrà mai standardizzare tutte le interessanti tecnologie utilizzate nel mondo dello sviluppo di applicazioni Java, ma Web Beans rende più facile usare in modo trasparente all'interno dell'ambiente Java EE tecnologie che non fanno ancora parte di tale ambiente.

Stiamo per scoprire come sfruttare appieno la piattaforma Java EE in applicazioni che usano Web Beans. Scorreremo anche velocemente un insieme di SPI (Service Provider Interface) fornite per permettere la realizzazione di estensioni portabili di Web Beans. Potrebbe non essere mai necessario usare queste SPI direttamente, ma è opportuno sapere che esistono, qualora possano servire. Soprattutto, le si sfrutterà in modo indiretto ogni volta che si utilizzeranno estensioni di terze parti.

# Integrazione Java EE

Web Beans è pienamente integrata nell'ambiente Java EE. Web Beans ha accesso alle risorse Java EE ed ai contesti di persistenza JPA. I Web Beans possono essere usati in espressioni Unified EL dentro pagine JSF e JSP. Possono anche essere iniettati negli oggetti, come Servlet e Message-Driven Beans, che non sono Web Beans.

## 13.1. Iniettare risorse Java EE in un Web Bean

Tutti i Web Beans sia semplici che enterprise si avvantaggiano della dependency injection di Java EE usando `@Resource`, `@EJB` e `@PersistenceContext`. Abbiamo già visto un paio di esempi a riguardo, sebbene non ci siamo soffermati molto a suo tempo.

```
@Transactional @Interceptor
public class TransactionInterceptor {

   @Resource Transaction transaction;

   @AroundInvoke public Object manageTransaction(InvocationContext ctx) { ... }

}
```

```
@SessionScoped
public class Login {

   @Current Credentials credentials;
   @PersistenceContext EntityManager userDatabase;

   ...

}
```

Le chiamate Java EE `@PostConstruct` e `@PreDestroy` vengono anche supportate per tutti i Web Beans (semplici e enterprise). Il metodo `@PostConstruct` viene chiamato dopo che *tutta* l'injection è stata eseguita.

C'è una restrizione di cui essere informati: `@PersistenceContext(type=EXTENDED)` non è supportato per i Web Beans semplici.

## 13.2. Chiamare un Web Bean da un servlet

E' facile utilizzare i Web Beans da un Servlet in Java EE 6. Semplicemente si inietti il Web Bean utilizzando l'injection del campo Web Bean o del metodo inizializzatore.

```
public class Login extends HttpServlet {

   @Current Credentials credentials;
   @Current Login login;

   @Override
   public void service(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
      credentials.setUsername( request.getAttribute("username") ):
      credentials.setPassword( request.getAttribute("password") ):
      login.login();
      if ( login.isLoggedIn() ) {
         response.sendRedirect("/home.jsp");
      }
      else {
         response.sendRedirect("/loginError.jsp");
      }
   }

}
```

Il client proxy Web Beans si occupa di instradare le invocazioni dei metodi da un Servlet alle corrette istanze di `Credentials` e `Login` per la richiesta corrente e la sessione HTTP.

## 13.3. Chiamare un Web Bean da un Message-Driven Bean

L'injection dei Web Beans si applica a tutti gli EJB3, perfino quando non sono sotto il controllo del manager Web Bean (per esempio se sono stati ottenuti da ricerca JNDI diretta, o injection usando `@EJB`) In particolaresi può usare l'injection di Web Beans nei Message-Driven Beans, che non sono considerati Web Beans poiché non possono essere iniettati.

Si possono perfino associare degli interceptor Web Beans ai Message-Driven Beans.

```
@Transactional @MessageDriven
public class ProcessOrder implements MessageListener {
```

```
    @Current Inventory inventory;
    @PersistenceContext EntityManager em;

    public void onMessage(Message message) {
        ...
    }

}
```

Quindi ricevere i messaggi è veramente facile in ambiente Web Beans. Ma attenzione che non è disponibile alcun contesto di sessione o conversazione quando il messaggio viene consegnato al Message-Driven Bean. Solamente i Web Beans `@RequestScoped` and `@ApplicationScoped` sono disponibili.

E' anche molto facile spedire messaggi usando Web Beans.

## 13.4. Endpoint JMS

La spedizione dei messaggi usando JMS può essere abbastanza complessa, a causa del numero di oggetti differenti da trattare. Per le code si hanno `Queue`, `QueueConnectionFactory`, `QueueConnection`, `QueueSession` e `QueueSender`. Per i topic si hanno `Topic`, `TopicConnectionFactory`, `TopicConnection`, `TopicSession` e `TopicPublisher`. Ciascuno di questi oggetti ha il proprio ciclo di vita e modello di thread di cui bisogna (pre)occuparsi.

I Web Beans si prendono cura di tutto questo per noi. Tutto ciò che occorre fare è dichiarare la coda od il topic in `web-beans.xml`, specificando un

```
<Queue>
    <destination
>java:comp/env/jms/OrderQueue</destination>
    <connectionFactory
>java:comp/env/jms/QueueConnectionFactory</connectionFactory>
    <myapp:OrderProcessor/>
</Queue
>
```

```
<Topic>
    <destination
>java:comp/env/jms/StockPrices</destination>
    <connectionFactory
>java:comp/env/jms/TopicConnectionFactory</connectionFactory>
    <myapp:StockPrices/>
```

```
</Topic
>
```

Ora è possibile iniettare `Queue`, `QueueConnection`, `QueueSession` o `QueueSender` per una coda, oppure `Topic`, `TopicConnection`, `TopicSession` o `TopicPublisher` per un topic.

```
@OrderProcessor QueueSender orderSender;
@OrderProcessor QueueSession orderSession;

public void sendMessage() {
   MapMessage msg = orderSession.createMapMessage();
   ...
   orderSender.send(msg);
}
```

```
@StockPrices TopicPublisher pricePublisher;
@StockPrices TopicSession priceSession;

public void sendMessage(String price) {
   pricePublisher.send( priceSession.createTextMessage(price) );
}
```

Il ciclo di vita degli oggetti JMS iniettati è interamente controllato dal manager Web Bean.

## 13.5. Packaging and deployment

Web Beans non definisce nessuno archivio speciale per il deploy. Si può impacchettare i Web Beans in JAR, EJB-JAR o WAR # qualsiasi locazione di deploy nel classpath dell'applicazione. Comunque ciascun archivio che contiene Web Beans devi includere un file chiamato `web-beans.xml` nella directory `META-INF` o `WEB-INF`. Il file può essere vuoto. I Web Beans collocati negli archivi che non hanno un file `web-beans.xml` non saranno disponibili per l'uso nell'applicazione.

Per l'esecuzione in Java SE, Web Beans può essere deployato in un qualsiasi posto nel quale gli EJB siano stati messi per essere eseguito da un embeddable EJB Lite container. Di nuovo ogni locazioni deve contenere un file `web-beans.xml`.

# Estendere i Web Beans

Web Beans è inteso essere una piattaforma per framework, estensioni e integrazione con altre tecnologie. Quindi Web Beans espone un set di SPI (Service Provider Interface) per l'uso da parte degli sviluppatori di estensioni portabili a Web Beans. Per esempio, i seguentitipi di estensione sono state prese in considerazione dai progettisti di Web Beans:

- Integrazione con i motori di Gestione dei Processi di Business,
- integrazione con framework di terze-parti quali Spring, Seam, GWT o Wicket, e
- nuova tecnologia basata sul modello di programmazione di Web Beans.

Il nervo centrale per l'estensione di Web Beans è l'oggetto `Manager`.

## 14.1. L'oggetto `Manager`

L'interfaccia `Manager` consente di registrare ed ottenere programmaticamente interceptor, decoratori, osservatori e contesti di Web Beans.

```
public interface Manager
{

  public <T
> Set<Bean<T
>
> resolveByType(Class<T
> type, Annotation... bindings);

  public <T
> Set<Bean<T
>
> resolveByType(TypeLiteral<T
> apiType,
      Annotation... bindings);

  public <T
> T getInstanceByType(Class<T
> type, Annotation... bindings);

  public <T
> T getInstanceByType(TypeLiteral<T
> type,
      Annotation... bindings);
```

```
   public Set<Bean<?>
> resolveByName(String name);

   public Object getInstanceByName(String name);

   public <T
> T getInstance(Bean<T
> bean);

   public void fireEvent(Object event, Annotation... bindings);

   public Context getContext(Class<? extends Annotation
> scopeType);

   public Manager addContext(Context context);

   public Manager addBean(Bean<?> bean);

   public Manager addInterceptor(Interceptor interceptor);

   public Manager addDecorator(Decorator decorator);

   public <T
> Manager addObserver(Observer<T
> observer, Class<T
> eventType,
         Annotation... bindings);

   public <T
> Manager addObserver(Observer<T
> observer, TypeLiteral<T
> eventType,
         Annotation... bindings);

   public <T
> Manager removeObserver(Observer<T
> observer, Class<T
> eventType,
         Annotation... bindings);

   public <T
> Manager removeObserver(Observer<T
> observer,
         TypeLiteral<T
```

```
> eventType, Annotation... bindings);

   public <T
> Set<Observer<T
>
> resolveObservers(T event, Annotation... bindings);

   public List<Interceptor
> resolveInterceptors(InterceptionType type,
         Annotation... interceptorBindings);

   public List<Decorator
> resolveDecorators(Set<Class<?>
> types,
         Annotation... bindings);

}
```

Possiamo ottenere un'istanza di `Manager` via iniezione:

```
@Current Manager manager
```

## 14.2. La classe `Bean`

Istanze della classe astratta `Bean` rappresentano i Web Beans. C'è un'istanza di `Bean` registrata con l'oggetto `Manager` per ogni Web Bean dell'applicazione.

```
public abstract class Bean<T> {

   private final Manager manager;

   protected Bean(Manager manager) {
      this.manager=manager;
   }

   protected Manager getManager() {
      return manager;
   }

   public abstract Set<Class> getTypes();
   public abstract Set<Annotation> getBindingTypes();
   public abstract Class<? extends Annotation> getScopeType();
```

```
    public abstract Class<? extends Annotation> getDeploymentType();
    public abstract String getName();

    public abstract boolean isSerializable();
    public abstract boolean isNullable();

    public abstract T create();
    public abstract void destroy(T instance);

}
```

E' possibile estendere la classe `Bean` e registrare le istanze chiamando `Manager.addBean()` per fornire supporto a nuovi tipi di Web Beans, oltre a quelli definiti dalla specifica Web Beans (semplici ed enterprise, metodi produttori e endpoint JMS). Per esempio possiamo usare la classe `Bean` per consentire ad oggetti gestiti da altri framework di essere iniettati nei Web Beans.

Ci sono due sottoclassi di `Bean` definite dalla specifica Web Beans: `Interceptor` e `Decorator`.

## 14.3. L'interfaccia `Context`

L'interfaccia `Context` supporta l'aggiunta di nuovi scope ai Web Beans, o l'estensione di scope esistenti a nuovi ambienti.

```
public interface Context {

    public Class<? extends Annotation> getScopeType();

    public <T> T get(Bean<T> bean, boolean create);

    boolean isActive();

}
```

Per esempio possiamo implementare `Context` per aggiungere uno scope di tipo business process a Web Beans, o per aggiungere il supporto allo scope di conversazione ad un'applicazione che impiega Wicket.

# Prossimi passi

Poiché Web Beans è una tecnologia nuova, non è ancora disponibile molta informazione online.

La specifica Web Beans è sicuramente la migliore fonte per avere informazioni su Web Beans. La specifica è lunga circa 100 pagine, circa quest'articolo e per lo più leggibile altrettanto facilmente. Ma sicuramente copre molti dettagli che sono stati saltati nel presente documento. La specifica è disponibile al seguente indirizzo `http://jcp.org/en/jsr/detail?id=299`.

L'implementazione della documentazione Web Beans è stata sviluppata in `http://seamframework.org/WebBeans`. Il team di sviluppo di RI ed il blog per la specifica Web Beans si trova in `http://in.relation.to`. Quest'articolo è sostanzialmente basato su una serie di articoli pubblicati sul blog.

# Parte V. Web Beans Reference

Web Beans è la reference implementatio di JSR-299, ed è usata da JBoss AS e Glassfish per fornire i servizi JSR-299 per le applicazioni Java Enterprise Edition. Web Beans va oltre gli ambienti e le API definite dalla specifica JSR-299 e fornisce supporto ad un numero di altri ambienti (quali servlet container quali Tomcat o Java SE) ed API e moduli addizionali (quali il logging la generazione XSD per i descrittori di deploy XML JSR-299).

Se si vuole velocemente iniziare ad usare Web Beans con JBoss AS o Tomcat e provare con un esempio, si guardi in *Capitolo 3, Implementazione di riferimento di Web Beans JSR-299*. Altrimenti continuare a leggere la discussione esaustiva per usare Web Beans in tutti gli ambienti e gli application server che vengono supportati, così come le estensioni Web Beans.

# Application Server ed ambienti supportati da Web Beans

## 16.1. Usare Web Beans con JBoss AS

Non occorre alcuna configurazione speciale dell'applicazione oltre all'aggiunta di `META-INF/beans.xml` o `WEB-INF/beans.xml`.

Se si usa JBoss AS 5.0.1.GA allora occorre installare Web Beans come extra. Innanzitutto occorre dire a Web Beans dove si trova JBoss. Modificare `jboss-as/build.properties` ed impostare la proprietà `jboss.home`. Per esempio:

```
jboss.home=/Applications/jboss-5.0.1.GA
```

Ora installiamo Web Beans:

```
$ cd webbeans-$VERSION/jboss-as
$ ant update
```

**Nota**

Un nuovo deploer - `webbeans.deployer` - viene aggiunto a JBoss AS. Questo aggiunge a JBoss AS il supporto ai deploy JSR-299, e consente a Web Beans di interrogare il container EJB3 per scoprire quali EJB sono installati nell'applicazione.

Web Beans è incluso in tutte le release di JBoss AS da 5.1 in avanti.

## 16.2. Glassfish

DA FARE

## 16.3. Tomcat (or any plain Servlet container)

Web Beans può essere usato in Tomcat 6.0.

### Nota

Web Beans doesn't support deploying session beans, injection using `@EJB`, or `@PersistenceContext` or using transactional events on Tomcat.

Web Beans should be used as a web application library in Tomcat. You should place `webbeans-tomcat.jar` in `WEB-INF/lib`. `webbeans-tomcat.jar` is an "uber-jar" provided for your convenience. Instead, you could use its component jars:

- `jsr299-api.jar`
- `webbeans-api.jar`
- `webbeans-spi.jar`
- `webbeans-core.jar`
- `webbeans-logging.jar`
- `webbeans-tomcat-int.jar`
- `javassist.jar`
- `dom4j.jar`

You also need to explicitly specify the Tomcat servlet listener (used to boot Web Beans, and control its interaction with requests) in `web.xml`:

```
<listener>
  <listener-class
>org.jboss.webbeans.environment.servlet.Listener</listener-class>
</listener
>
```

Tomcat ha un JNDI read-only, e quindi Web Beans non può automaticamente associare il Manager. Per associare il Manager a JNDI occorre aggiungere a `META-INF/context.xml`:

```
<Resource name="app/Manager"
        auth="Container"
        type="javax.inject.manager.Manager"
        factory="org.jboss.webbeans.resources.ManagerObjectFactory"/>
```

e renderlo disponibile nel proprio deploy aggiungendo questo a `web.xml`:

```
<resource-env-ref>
  <resource-env-ref-name>
    app/Manager
  </resource-env-ref-name>
  <resource-env-ref-type>
    javax.inject.manager.Manager
  </resource-env-ref-type>
</resource-env-ref
>
```

Tomcat only allows you to bind entries to `java:comp/env`, so the Manager will be available at `java:comp/env/app/Manager`

Web Beans also supports Servlet injection in Tomcat. To enable this, place the `webbeans-tomcat-support.jar` in `$TOMCAT_HOME/lib`, and add the following to your `META-INF/context.xml`:

```
<Listener className="org.jboss.webbeans.environment.tomcat.WebBeansLifecycleListener" />
```

## 16.4. Java SE

Apart from improved integration of the Enterprise Java stack, Web Beans also provides a state of the art typesafe, stateful dependency injection framework. This is useful in a wide range of application types, enterprise or otherwise. To facilitate this, Web Beans provides a simple means for executing in the Java Standard Edition environment independently of any Enterprise Edition features.

When executing in the SE environment the following features of Web Beans are available:

- `Simple Web Beans (POJOs)`
- `Typesafe Dependency Injection`
- `Application and Dependent Contexts`
- `Binding Types`
- `Stereotypes`
- `Decorators`
- `(TODO: Interceptors ?)`
- `Typesafe Event Model`

### 16.4.1. Web Beans SE Module

To make life easy for developers Web Beans provides a special module with a main method which will boot the Web Beans manager, automatically registering all simple Web Beans found on the classpath. This eliminates the need for application developers to write any bootstrapping code. The entry point for a Web Beans SE applications is a simple Web Bean which observes the standard `@Deployed Manager` event. The command line paramters can be injected using either of the following:

```
@Parameters List<String
> params;
@Parameters String[] paramsArray; // useful for compatability with existing classes
```

Here's an example of a simple Web Beans SE application:

```
@ApplicationScoped
public class HelloWorld
{
   @Parameters List<String
> parameters;

   public void printHello( @Observes @Deployed Manager manager )
   {
      System.out.println( "Hello " + parameters.get(0) );
   }
}
```

Web Beans SE applications are started by running the following main method.

```
java org.jboss.webbeans.environments.se.StartMain <args
>
```

If you need to do any custom initialization of the Web Beans manager, for example registering custom contexts or initializing resources for your beans you can do so in response to the `@Initialized Manager` event. The following example registers a custom context:

```
public class PerformSetup
{

   public void setup( @Observes @Initialized Manager manager )
```

```
  {
     manager.addContext( ThreadContext.INSTANCE );
  }
}
```

### Nota

The command line parameters do not become available for injection until the `@Deployed Manager` event is fired. If you need access to the parameters during initialization you can do so via the `public static String getParameters()` method in `StartMain`.

# Estensioni JSR-299 disponibili come parte di Web Beans

**Importante**

Questi moduli sono utilizzabili su qualsiasi implementazione JSR-299, non solo Web Beans!

## 17.1. Web Beans Logger

DA FARE

## 17.2. Generatore XSD per descrittori di deploy XML JSR-299

DA FARE

# Appendice A. Integrazione di Web Beans RI in altri ambienti

Attualmente Web Bean RI funziona solo in JBoss AS 5; l'integrazione di RI in altri ambienti EE (per esempio in un application server come Glassfish), in un servlet container (come Tomcat), o con un'implementazione EJB3.1 Embedded è abbastanza facile. In questo appendice si discuterà brevemente dei passi necessari.

**Nota**

Dovrebbe essere possibile far funzionare Web Beans in un ambiente SE, ma occorre molto lavoro per aggiungere i propri contesti ed il ciclo di vita. Web Beans RI attualmente non espone punti di estensione del ciclo di vita, così occorre codificare direttamente nelle classi Web Beans RI.

## A.1. Web Beans RI SPI

The Web Beans SPI is located in `webbeans-spi` module, and packaged as `webbeans-spi.jar`. Some SPIs are optional, if you need to override the default behavior, others are required.

Tutte le interfacce in SPI supportano il pattern decorator e forniscono una classe `Forwarding`.

### A.1.1. Web Bean Discovery

```
public interface WebBeanDiscovery {
  /**
   * Gets list of all classes in classpath archives with web-beans.xml files
   *
   * @return An iterable over the classes
   */
  public Iterable<Class<?>
> discoverWebBeanClasses();

  /**
   * Gets a list of all web-beans.xml files in the app classpath
   *
   * @return An iterable over the web-beans.xml files
   */
  public Iterable<URL
> discoverWebBeansXml();
```

```
}
```

L'analisi dei file delle classi Web Bean e di `web-bean.xml` è molto istruttiva (l'algoritmo è descritto nella sezione 11.1 della specifica JSR-299 e non viene qua ripetuto).

## A.1.2. Servizi EJB

Web Beans RI delega al container la rilevazione dei bean EJB3 e quindi risulta non essere necessario eseguire lo scan delle annotazioni EJB3 o fare il parsing di `ejb-jar.xml`. Per ciascun EJB nell'applicazione dovrebbe essere rilevato un EJBDescriptor:

```
public interface EjbServices
{

  /**
   * Gets a descriptor for each EJB in the application
   *
   * @return The bean class to descriptor map
   */
  public Iterable<EjbDescriptor<?>
> discoverEjbs();
```

```
public interface EjbDescriptor<T
> {

  /**
   * Gets the EJB type
   *
   * @return The EJB Bean class
   */
  public Class<T
> getType();

  /**
   * Gets the local business interfaces of the EJB
   *
   * @return An iterator over the local business interfaces
   */
  public Iterable<BusinessInterfaceDescriptor<?>
> getLocalBusinessInterfaces();

  /**
```

```
    * Gets the remote business interfaces of the EJB
    *
    * @return An iterator over the remote business interfaces
    */
   public Iterable<BusinessInterfaceDescriptor<?>
> getRemoteBusinessInterfaces();

   /**
    * Get the remove methods of the EJB
    *
    * @return An iterator over the remove methods
    */
   public Iterable<Method
> getRemoveMethods();

   /**
    * Indicates if the bean is stateless
    *
    * @return True if stateless, false otherwise
    */
   public boolean isStateless();

   /**
    * Indicates if the bean is a EJB 3.1 Singleton
    *
    * @return True if the bean is a singleton, false otherwise
    */
   public boolean isSingleton();

   /**
    * Indicates if the EJB is stateful
    *
    * @return True if the bean is stateful, false otherwise
    */
   public boolean isStateful();

   /**
    * Indicates if the EJB is and MDB
    *
    * @return True if the bean is an MDB, false otherwise
    */
   public boolean isMessageDriven();

   /**
```

```
   * Gets the EJB name
   *
   * @return The name
   */
  public String getEjbName();


}
```

Il `EjbDescriptor` è abbastanza auto-esplicatorio e dovrebbe restituire i metadati rilevanti definiti nella specifica EJB. In aggiunta a queste due interfacce, vi è `BusinessInterfaceDescriptor` a rappresentare un'interfaccia locale di business (che incapsula la classe d'interfaccia ed il nome jndi usato per la ricerca di una istanza EJB).

La risoluzione di `@EJB` e `@Resource` è delegata al container. Occorre fornire un'implementazione di `org.jboss.webbeans.ejb.spi.EjbServices` che rende disponibili queste operazioni. Web Beans passa nel `javax.inject.manager.InjectionPoint` la risoluzione, anche come `NamingContext`, che è in uso per ogni richiesta di risoluzione.

### A.1.3. Servizi JPA

Solo come risoluzione di `@EJB` è delegata al container, e quindi è la risoluzione di `@PersistenceContext`.

PROBLEMA APERTO: Web Beans richiede anche che il container fornisca una lista di entity nel deploy, cosicché non siano rilevati come semplici bean.

### A.1.4. Servizi di transazione

Web Beans RI deve delegare le attività JTA al container. SPI fornisce un paio di modi per ottenere ciò tramite l'interfaccia `TransactionServices`.

```
public interface TransactionServices
{
  /**
   * Possible status conditions for a transaction. This can be used by SPI
   * providers to keep track for which status an observer is used.
   */
  public static enum Status
  {
    ALL, SUCCESS, FAILURE
  }

  /**
```

```
    * Registers a synchronization object with the currently executing
    * transaction.
    *
    * @see javax.transaction.Synchronization
    * @param synchronizedObserver
    */
   public void registerSynchronization(Synchronization synchronizedObserver);

   /**
    * Queries the status of the current execution to see if a transaction is
    * currently active.
    *
    * @return true if a transaction is active
    */
   public boolean isTransactionActive();
}
```

La enumeration `Status` serve agli implementatori per poter essere in grado di tracciare se una sincronizzazione deve notificare un osservatore solo quando la transazione ha avuto successo, o dopo un errore, o indipendentemente dallo stato della transazione.

Qualsiasi implementazione di `javax.transaction.Synchronization` può essere passata al metodo `registerSynchronization()` e l'implementazione SPI deve immediatamente registrare la sincronizzazione con il gestore della transazione JTA usato per EJB.

Per facilitare la determinazione se o no una transazione è attualmente attiva per il thread di richiesta, può essere usato il metodo `isTransactionActive()`. L'implementazione SPI deve interrogare lo stesso gestore della transazione JTA usato per EJB.

## A.1.5. Il contesto applicazione

Web Beans si aspetta che l'Application Server od un altro container fornisca la memorizzazione per ogni contesto applicazione. `org.jboss.webbeans.context.api.BeanStore` dovrebbe essere implementato per fornire uno storage con scope applicazione. Si può trovare molto utile `org.jboss.webbeans.context.api.helpers.ConcurrentHashMapBeanStore`.

## A.1.6. Bootstrap e spegnimento

L'interfaccia `org.jboss.webbeans.bootstrap.api.Bootstrap` definisce il bootstrap per Web Beans. Per avviare Web Beans occorre ottenere un'istanza di `org.jboss.webbeans.bootstrap.WebBeansBootstrap` (che implementa `Boostrap`), e comunicare le SPI in uso, e poi chiedere che il container venga avviato.

Il bootstrap è suddiviso in più fasi, inizializzazione del bootstrap, bootstrap e shutdown. L'inizializzazione creerà un manager, e aggiungerà i contesti standard (definiti dalla specifica).

Bootstrap scoprirà EJB, classi e XML; aggiungerà i bean definiti con le annotazioni; aggiungerà i bean definiti con XML; e validerà tutti i bean.

Il bootstrap supporta più ambienti. Diversi ambienti richiedono diversi servizi presenti (per esempio servlet non richiede i servizi di transazione, EJB o JPA). Di default viene assunto un ambiente EE, ma si può impostare un ambiente chiamando `bootstrap.setEnvironment()`.

Per inizializzare il bootstrap si chiama `Bootstrap.initialize()`. Prima della chiamata di `initialize()` occorre registrare i servizi richiesti dal proprio ambiente. Si può fare questo chiamando `bootstrap.getServices().add(JpaServices.class, new MyJpaServices())`. Occorre anche fornire l'application context bean store.

Dopo aver chiamato `initialize()`, il `Manager` può essere ottenuto chiamando `Bootstrap.getManager()`.

Per avviare il container chiamare `Bootstrap.boot()`.

Per spegnere il container si chiama `Bootstrap.shutdown()`. Questo consente al container di eseguire ogni pulizia necessaria delle operazioni.

### A.1.7. JNDI

Web Beans RI implementa la ricerca e l'associazione JNDI secondo gli standard, ma può capitare di dover modificare la ricerca e l'associazione (per esempio in un ambiente dove JNDI non è disponibile). Per fare questo occorre implementare `org.jboss.webbeans.resources.spi.NamingContext`:

```
public interface NamingContext extends Serializable {

  /**
   * Typed JNDI lookup
   *
   * @param <T
> The type
   * @param name The JNDI name
   * @param expectedType The expected type
   * @return The object
   */
  public <T
> T lookup(String name, Class<? extends T
> expectedType);

  /**
   * Binds an item to JNDI
   *
   * @param name The key to bind under
```

```
   * @param value The item to bind
   */
  public void bind(String name, Object value);

}
```

## A.1.8. Caricamento risorse

Web Beans RI deve caricare le classi e le risorse dal classpath in vari momenti. Di default vengono caricati dallo stesso classloader usato per caricare RI, comunque questo potrebbe non essere corretto in alcuni ambienti. Se è questo il caso si può implementare `org.jboss.webbeans.spi.ResourceLoader`:

```
      public interface ResourceLoader {

  /**
   * Creates a class from a given FQCN
   *
   * @param name The name of the clsas
   * @return The class
   */
  public Class<?> classForName(String name);

  /**
   * Gets a resource as a URL by name
   *
   * @param name The name of the resource
   * @return An URL to the resource
   */
  public URL getResource(String name);

  /**
   * Gets resources as URLs by name
   *
   * @param name The name of the resource
   * @return An iterable reference to the URLS
   */
  public Iterable<URL
> getResources(String name);

}
```

### A.1.9. Iniezione dei servlet

Java EE / Servlet non fornisce alcun hook da usare per fornire l'iniezione nei Servlet, quindi Web Beans fornisce un'API per consentire al container di richiedere l'iniezione JSR-299 per un Servlet.

Per soddisfare la JSR-299 il container deve richiedere l'iniezione servlet per ogni nuovo servlet istanziato dopo che il costruttore ritorni e prima che il servlet sia messo in servizio.

Per eseguire l'iniezione su un servlet si chiami `WebBeansManager.injectServlet()`. Il manager può essere ottenuto da `Bootstrap.getManager()`.

## A.2. Il contratto con il container

Per il corretto funzionamento al di fuori dell'implementazione delle API, ci sono un numero di requisiti che Web Beans RI pone nel container.

Isolamento del classloader
: Se si integra Web Beans in un ambiente che supporta il deploy di applicazioni, occorre abilitare, automaticamente o attraverso la configurazione utente, l'isolamento del classloader per ogni applicazione Web Beans.

Servlet listener e filtri
: Se si integra Web Beans in un ambiente Servlet occorre registrare `org.jboss.webbeans.servlet.WebBeansListener` come Servlet listener, o automaticamente, o attraverso una configurazione utente, per ciascuna applicazione Web Beans che usa Servlet.

  Se si integra Web Beans in un ambiente Servlet occorre registrare `org.jboss.webbeans.servlet.ConversationPropagationFilter` come Servlet listener, o automaticamente, o attraverso una configurazione utente, per ciascuna applicazione Web Beans che usa JSF. Questo filtro può venir registrato in modo sicuro per tutti i deploy dei servlet.

Session Bean Interceptor
: Se si integra Web Beans in un ambiente EJB occorre registrare `org.jboss.webbeans.ejb.SessionBeanInterceptor` come interceptor EJB per ogni EJB dell'applicazione, o automaticamente o attraverso una configurazione utente, per ciascuna applicazione Web Beans che utilizza bean enterprise.

**Importante**

Occorre registrare il `SessionBeanInterceptor` come interceptor più interno allo stack per tutti gli EJB.

`webbeans-core.jar`

Se si integra Web Beans in un ambiente che supporta il deploy di applicazioni, occorre inserire `webbeans-core.jar` nel classloader isolato delle applicazioni. Non può essere caricato da un classloader condiviso.