JAVA 6 EE Pocket Guide [BIGNAMI]

O'REILLY

Indice generale

Introduzione....1
What's New in Java EE 6....3
Managed Beans....4
Lifecycle Callback....4
Servlets....5
Java Persistence API....5
Entities....5
Persistence Unit, Persistence Context and Entity Manager....6
Create, Read, Update and Delete Entities....6
Validating the Entities....8
Transactions and Locking....8
Enterprise JavaBeans....9
Stateful Session Beans....9
Stateless Session Beans....10
Singleton Session Beans....10
Message-Driven Beans....11
Portable Global JNDI Names....11
Transactions....11
Asynchronous....12
Timers....12
Embeddable API....12
EJB.Lite....13
Contexts and Dependency Injection....13
Injection Points....13
Qualifier and Alternative....14
Producer and Disposer....15
Interceptors and Decorators....15
Scopes and Contexts....16
Stereotypes....16
Events....17
Portable Extensions....18
JavaServer Faces....18
Facelets....18
Resource Handling....20
Composite Components....20
Ajax....21
HTTP GET....21
Server and Client Extension Points....21
Navigation Rules....21
SOAP-Based Web Services....22
Web Service Endpoints....22
Provider-Based Dynamic Endpoints....23
Web Service Client....23
Handlers....24
RESTful Web Services....24
Simple RESTful Web Services....25
Binding HTTP Methods....25
Multiple Resource Representations....26
Java Message Service....28
Bean Validation....28
Built-in Constraints....28
Defining a Custom Constraint....29
Validation Groups....30
Integration with JPA....31
Integration with JSF....32

Introduzione

Java EE 6 è costituito dalle specifiche che definiscono i requisti della piattaforma. Si compone anche di specifiche per

componenti:
Web Technologies

- JSR 45: Debugging Support for Other Languages
- JSR 52: Standard Tag Library for JavaServer Pages (JSTL)1.2
- JSR 245: JavaServer Pages (JSP) 2.2 and Expression Language (EL) 1.2
- JSR 314: JavaServer Faces (JSF) 2.0
- JSR 315: Servlet 3.0

Enterprise Technologies

- JSR 250: Common Annotations for the Java Platform 1.1
- JSR 299: Contexts and Dependency Injection (CDI) for the Java EE Platform 1.0
- JSR 303: Bean Validation 1.0
- JSR 316: Managed Beans 1.0
- JSR 317: Java Persistence API (JPA) 2.0
- JSR 318: Enterprise JavaBeans (EJB) 3.1
- JSR 318: Interceptors 1.1
- JSR 322: Java EE Connector Architecture 1.6
- JSR 330: Dependency Injection for Java 1.0
- JSR 907: Java Transaction API (JTA) 1.1
- JSR 914: Java Message Server (JMS) 1.1
- JSR 919: JavaMail 1.4

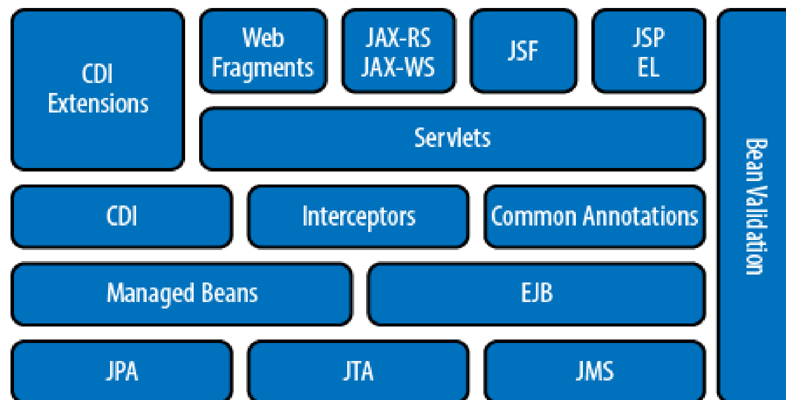
Web Service Technologies

- JSR 67: Java APIs for XML Messaging (JAXM) 1.3
- JSR 93: Java API for XML Registries (JAXR) 1.0
- JSR 101: Java API for XML-based RPC (JAXRPC) 1.1
- JSR 109: Implementing Enterprise Web Services 1.3
- JSR 173: Streaming API for XML (StAX) 1.0
- JSR 181: Web Services Metadata for the Java Platform 2.0
- JSR 222: Java Architecture for XML Binding (JAXB) 2.2
- JSR 224: Java API for XML Web Services (JAXWS) 2.2
- JSR 311: Java API for RESTful Web Services (JAXRS) 1.1

Management and Security Technologies

- JSR 77: J2EE Management API 1.1
- JSR 88: Java Platform EE Application Deployment API 1.2
- JSR 115: Java Authorization Contract and Containers (JACC) 1.3
- JSR 196: Java Authentication Service Provider Inteface for Containers (JASPIC) 1.0

Questi componenti lavorano insieme per fornire uno stack di tecnologie integrate fra loro:



What's New in Java EE 6

Managed Beans
• POJO-based managed component.
• Fornisce un insieme di servizi comuni come lifecycle resource injection, callbacks, and interceptors.

Enterprise JavaBeans
• Uso delle annotazioni (@Stateless, @Stateful, @Singleton) per creare un EJB da un singolo POJO.
• Possibilità di deployare EJBs in un .war per l'accesso locale usando l'annotazione @Local. Uso di ejb-jar per accesso locale e remoto.
• Possibilità di accedere a EJBs usando un portable global JNDI name.
• Un metodo di un session bean può essere marcato per invocazione asincrona (fire-and-forget pattern).
• Possibilità di schedulare temporalmente eventi usando una sintassi cron-like, tramite l'annotazione @Schedule posta sui metodi.
• Embeddable EJB API: permette a codice client e al suo corrispondente enterprise bean di essere eseguiti all'interno della stessa JVM e dello stesso class loader.

Servlets
• Annotation-driven Servlet (@WebServlet), Filter (@WebFilter), and Listener (@WebListener). Il descrittore web.xml diventa opzionale nella maggior parte dei casi.
• Servlets, filters, and listeners possono essere registrati programmaticamente usando ServletContext.
• Possibilità di svolgere operazioni asincone.
• Librerie di framework possono essere integrate in modo modulare usando web-fragment.xml.
• Possibilità di definire la Servlet security tramite annotation (@ServletSecurity, @HttpConstraint, @HttpMethodConstraint) in aggiunta a <security-constraint>.

Java API for RESTful Web Services
• Pubblicazione di RESTful web services tramite POJO e annotazioni.
• Supporto al set standard set di metodi del protocollo HTTP: GET, POST, PUT, and DELETE.
• Ogni risorsa può essere rappresentata in formati multipli, sono supportati anche formati custom.
• Supporto alla content negatiation Client-side usando HTTP Accept: header.

SOAP-Based Web Services
• Pubblicazione di SOAP-based web services tramite POJO e annotazioni. Controllo più fine sui messaggi usando Source, DataSource, e SOAPMessage.
• Client-side API per invocare un web service SOAP-based.
• Punti di estensione ben definiti per il pre/post processamente dei messaggi request/response su client e server.
• Standard Java-to-WSDL and WSDL-to-Java mapping.

JavaServer Faces
• Facelets come linguaggio di template predefinito per le pagine. Permette di definire facilmente composite components.
• Supporto per Ajax usando le APIs JavaScript e in modo dichiarativo usando f:ajax.
• La maggior parte degli elementi definiti nel file faces-config.xml hanno una corrispondete annotation che può essere usata in alternativa. Le regole di navigazione di default sono definite seguendo il principio convention-over-configuration.
• Supporto a HTTP GET e bookmarkable URLs.

• Intergrazione con la Bean Validation.

Java Persistence API
• Miglioramento dell' object/relational mapping.
• Il Metamodel cattura un metamodello del persistent state e delle relazioni fra le classi gestite dalla persistence unit. Questo schema viene per sfruttato per query type-safe tramite Criteria API.
• Supporto al locking pessimistico.
• Opzioni standard di configurazione usando javax.persistence properties.

Interceptors
• Possibilità di frapporsi su invocazioni e eventi del ciclo di vita che avvengono sulla classe target.
• Gli Interceptors supportano sia le annotazioni (@Interceptors) che i deployment descriptor come beans.xml.

Contexts and Dependency Injection
• Standards-based type-safe dependency injection.
• Forte tipizzazione specificando tutte le dipendenze usando Java type system. Fornisce loose coupling con Events, Interceptors, and Decorators.
• Integrazione con Expression Language.
• Definisce un extensible scope e un meccanismo di context management.
• Collegamento tra il livello delle transazioni (EJB) e il livello di presentazione (JSF).

Bean Validation
• Dichiarazione di constraint e validatori direttamente sulle classi per i POJO.
• Insieme di validatori built-in.
• Possibilità di dichiarare validatori custom usando METAINF/validation.xml in aggiunta alle annotazioni.

Managed Beans

Un managed bean è un POJO che è trattato come un managed component dal container Java EE.
Esso fornisce una base comune per i diversi tipi di componenti della piattaforma. Inoltre, la specifica definisce anche un piccolo insieme di servizi di base come resource injection, lifecycle callbacks, e interceptors su questi beans.

Un managed bean è un POJO con un costruttore senza argomenti con l'annotazione class-level javax.annotation.ManagedBean:

```
@javax.annotation.ManagedBean("myBean")
public class MyManagedBean {
//. . .
}
```

Questo bean può essere iniettato in ogni altro managed component in tre differenti modi:

1. Usando la @Resource annotation:

```
@Resource
MyManagedBean bean;
```

2. Usando la @Inject annotation:

```
@Inject
MyManagedBean bean;
```

3. Usando il riferimento JNDI java:app/ManagedBean/myBean oppure java:module/myBean dove ManagedBean è il nome dell'archivio di deploy (.war in questo caso):

```
InitialContext ic = new InitialContext();
MyManagedBean bean = (MyManagedBean)ic.lookup("java:module/myBean");
```

Non c'è un nome di default per un managed bean, quindi è importante fornire esplicitamente un nome per poter usare il riferimento JNDI. Le specifiche EJB e CDI estendono questa regola e forniscono delle regole per il default naming.

Una volta che il bean è stato iniettato, i suoi metodi di business possono essere invocati direttamente.

Lifecycle Callback

Le annotazioni standard javax.annotation.PostConstruct e javax.annotation.PreDestroy possono essere applicate ad ogni metodo di un managed bean per effettuare l'inizializzazione o il cleanup delle risorse:

```
@ManagedBean("myBean")
public class MyManagedBean {
@PostConstruct
public void setupResources() {
//. . .
}
@PreDestroy
public void cleanupResources() {
//. . .
}
public String sayHello() {
return "Hello " + name;
}
}
```

All'interno del metodo setupResources vengono acquisite le risorse necessarie durante l'esecuzione dei metodi di business, e all'interno del metodo cleanupResources queste risorse sono chiuse o rilasciate.
Questi metodi di lifecycle callback sono invocati dopo il costruttore senza argomenti.

Servlets

Una servlet è un web component ospitato in un servlet container che genera contenuti dinamici. I web clients interagiscono con la servlet usando il pattern request/response.
Il servlet container è invece responsabile del ciclo di vita della servlet, riceve le requests e spedisce le responses ed effettua ogni altra codifica/decodifica richiesta.

Una servlet è definita usando la @WebServlet annotation su un POJO, e deve estendere la classe javax.servlet.http.HttpServlet:

```
@WebServlet("/account")
public class AccountServlet
extends javax.servlet.http.HttpServlet {
//. . .
}
```

Java Persistence API

JPA definisce le API per la gestione della persistenza e mapping oggetti/relazioni usando un Java domain model.
JPA definisce un mapping standard fra le tabelle del database e i POJO.
Inoltre definisce tutte le funzionalità necessarie ad una applicazione che accede ad un database, come gestione delle transazioni, caching, validazione.

Entities

L'entity che mappa una o più tabelle è definita tramite un POJO con un costruttore pubblico senza argomenti e annotata con @Entity. Le variabili di istanza, che seguono lo stile JavaBeans, rappresentano lo stato persistente dell'entity.

```
@Entity
public class Student implements Serializable {
@Id
private int id;
private String name;
private String grade;
@Embedded
private Address address;
@ElementCollection
@CollectionTable("StudentCourse")
List<Course> courses;
//. . .
}
```

La classe impementa l'interfaccia Serializable e questo le consente di essere passata per valore attraverso una interfaccia remota.
L'entità può ereditare da una superclasse che le fornisca lo stato persistente e le informazioni di mappatura, ma tale

superclasse può essere o non essere a sua volta una entity. Per specificare l'eredità da una entity superclasse si usano le annotation @Inheritance e @Discriminator. Mentre per una nonentity superclasse si usa l'annotation @MappedSuperclass.
Le relazioni fra entità sono definite usando le annotazioni @OneToOne, @OneToMany, @ManyToOne, e @ManyToMany sul campo corrispondente all'entità referenziata.

Persistence Unit, Persistence Context and Entity Manager

Le entity sono gestite all'interno di un persistence context. Ogni entity ha una istanza unica per ogni persistent entity identity all'interno del contesto.
All'interno del persistence context, le istanze delle entity e il loro ciclo di vita sono gestita dall'entity manager.
L'entity manager può essere container-managed o application-managed.

Java EE:
Entity manager di tipo container-managed. E' ottenuto dall'applicazione direttamente attraverso dependency injection o tramite JNDI:

```
@PersistenceContext
EntityManager em;
```

Il persistence context è propagato attraverso transazioni multiple per entity manager di tipo container-managed, e il contaier è responsabile della gestione del ciclo di vita delle entity gestite.

Java SE:
Entity manager di tipo application-managed. E' ottenuto dall'applicazione da un entity manager factory:

```
@PersistenceUnit
EntityManagerFactory emf;
//. . .
EntityManager em = emf.createEntityManager();
```

Viene creato un nuovo persistence context isoltato quando viene richiesta una nuova entity e l'applicazione è responsabile della gestione del ciclo di vita dell'entity.

Gli entity managers, le loro informazioni di configurazione, l'insieme delle entity gestite e i metadati che specificano il mapping fra classi e database sono pacchettizzate insieme come una persistence unit.
Una persistence unit è definita dal file persistence.xml ed è contenuta all'interno di un ejb-jar, un .war, un .ear, o un application-client JAR.

Un semplice persistence.xml può essere definito come segue:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation=
"http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
<persistence-unit name="MyPU" transaction-type="JTA">
<provider>
org.eclipse.persistence.jpa.PersistenceProvider
</provider>
<jta-data-source>jdbc/sample</jta-data-source>
<exclude-unlisted-classes>
false
</exclude-unlisted-classes>
<properties>
<property name="eclipselink.ddl-generation"
value="create-tables"/>
</properties>
</persistence-unit>
</persistence>
```

L'elemento jta-data-source definisce il global JNDI name del data source JTA definito nel container. Di default, un persistence context di tipo container-managed persistence context ha uno scope a singola transazione e le entity sono detached alla fine della transazione.

Create, Read, Update and Delete Entities

La specifica JPS permette di effettuare operazioni di CRUD in diverse modalità:

Java Persistence Query Language (JPQL)
I metodi EntityManager.createNamedXXX sono usati per creare statements JPQL.

Criteria API
Le Criteria API permettono solo di effettuare query sulle entity.

Native SQL statement
I metodi EntityManager.createNativeXXX sono usati per creare query native.
Una nuova entity può essere persista nel database usando un entity manager:

```
Student student = new Student();
student.setId(1234);
//. . .
em.persist(student);
```

In questo codice, em è un entity manager. L'entity è persistita nel database al commit della transazione.

Esempio di statement JPQL statement per selezionare tutti gli studenti dalla rispettiva entity:

```
em.createNamedQuery("SELECT s FROM Student s").
getResultList();
```

@NamedQuery e @NamedQueries sono annotazioni usati per definire un mapping fra statements JPQL statici e un nome simbolico. Questo permette di seguire il design pattern “Don’t Repeat Yourself” e di centralizzare gli statements JPQL:

```
@NamedQuery(
name="findStudent"
value="SELECT s FROM Student s WHERE p.grade = :grade")
//. . .
Query query = em.createNamedQuery("findStudent");
List<Student> list = (List<Student>)query
.setParameter("grade", "4")
.getResultList();
```

JPA permette anche la costruzione di query dinamiche in modo typesage attraverso le Criteria API:

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery criteria = builder.createQuery
(Student.class);
Root<Student> root = criteria.from(Student.class);
criteria.select(root);
TypedQuery<Student> query = em.createQuery(criteria);
List<Student> list = query.getResultList();
```

Le @NamedQuery statiche sono più appropriate per use cases semplici. may be more appropriate for simple use cases. In una query complessa dove SELECT, FROM, WHERE e altre clausole sono definite in fase di esecuzione, le query JPQL dinamiche possono essere più soggetto ad errori, in genere a causa di una concatenazione di stringhe.
Le type-safe Criteria API offrono un modo più robusto per trattare con questi tipi query: tutte le clausole possono essere facilmente specificate in modo type-safe, fornendo il vantaggio della validazione a compile-time delle query.

Per aggiornare una entity esistente, è necessario prima recuperarla, poi applicare le necessarie modifiche al valore dei campi e infine chiamare il metodo EntityManager.merge:

```
Student student = (Student)query
.setParameter("id", "1234")
.getSingleResult();
//. . .
student.setGrade("5");
em.merge(student);
```

Una entity può essere aggiornata usando JPQL:

```
Query query = em.createQuery("UPDATE Student s"
+ "SET s.grade = :grade WHERE s.id = :id");
query.setParameter("grade", "5");
query.setParameter("id", "1234");
```

```
query.executeUpdate();
```

Per rimuovere una entity, è necessario recuperarla e quindi chiamare il metodo EntityManager.remove:

```
Student student = em.find(Student.class, 1234);
em.remove(student);
The entity may be deleted using JPQL:
Query query = em.createQuery("DELETE FROM Student s"
+ "WHERE s.id = :id");
query.setParameter("id", "1234");
query.executeUpdate();
```

Rimuovendo una entity, viene rimosso il corrispondente record sulla base dati.

Validating the Entities

Bean Validation 1.0 è una nuova specifica nella piattaforma Java EE 6 e permetto di specificare metadati per la validazione direttamente su JavaBeans.
Per JPA, tutte le classi gestite (entities, managed superclasses, and embeddable classes) possono essere configurate per includere i vincoli di Bean Validation. Questi vincoli sono applicati quando l'entity è persistita, aggiornata o rimossa dal database.

Bean Validation contiene alcuni vincoli pre-definiti come @Min, @Max, @Pattern, e @Size.

Si possono comunque facilmente definire vincoli custom.

La validazione automatica si ottiene delegando la validazione all'implementazione del Bean Validation nei metodi di pre-persist, pre-update e pre-remove delle lifecycle callback. In alternativa, la validazione può essere anche ottenuta espressamente dall'applicazione chiamando il metodo Validator.validate su una istanza di una managed class.
Di default, la validazione degli entity beans è automaticamente attivata. Questo comportamento può essere controllato modificando l'elemento validation-mode nel file persistence.xml.

Si possono definire dei gruppi di validazione, dichiarando una nuova interfaccia:

```
public interface MyGroup { }
```

Un campo dell'entity Student può essere associato al gruppo di validazione:

```
@Entity
public class Student implements Serializable {
@Id @NotNull int id;
@AssertTrue(groups=MyGroup.class)
private boolean canBeDeleted;
}
```

Il persistence.xml deve definire la seguente proprietà:

```
//. . .
<property
name="javax.persistence.validation.group.pre-remove"
value="org.sample.MyGroup"/>
```

Transactions and Locking

I metodi EntityManager.persist, .merge, .remove, e .refresh devono essere invocati all'interno di un transaction context quando viene usato un entity manager con un transaction-scoped persistence context.
Le transazioni sono controllate attraverso JTA oppure attraverso l'uso di EntityTransaction API locali.

Un entity manager di tipo container-managed deve usare JTA e questo è il tipico modo per ottenre un comportamento transazionale in un container Java EE.

Di default, viene usata la optimistic concurrency. L'annotazione @Version posta su un campo di una entity viene usata dal persistence provider per eseguire l'optimistic locking.

Enterprise JavaBeans

Gli Enterprise JavaBeans sono usati per sviluppare e deployare applicazioni distribuite di tipo component-based che hanno caratteristiche di scalabilità, transazionalità e sicurezza.

Un EJB tipicamente contiene la business logic che opera sui dati enterprise.

Le informazioni sul servizio, come gli attributi riguardo a transazioni e sicurezza, possono essere specificati tramite annotation o in un XML deployment descriptor.

Una istanza di un bean è gestita a runtime dal container. Questo permette agli sviluppatori di focalizzarsi sulla logica di business senza preoccuparsi di transazioni di basso livello e dettagli di gestione dello stato, servizi remoti, concorrenza, il multithreading, il pool di connessioni, o altre complesse API di basso livello.

Ci sono tre tipi di enterprise beans:
• Session beans
• Message-driven beans
• Entity beans

Gli Entity beans sono contrassegnati per l'eliminazione nella specifica EJB 3.1

Stateful Session Beans

Un stateful session bean contine lo stato conversazionale per uno specifico client. Lo stato è memorizzato nei valori dei campi di istanza del session bean.

Un stateful session bean può essere denifito usando:

```
@Stateful:
package org.sample;
@Stateful
public class Cart {
List<String> items;
public ShoppingCart() {
items = new ArrayList<Item>();
}
public void addItem(String item) {
items.add(item);
}
public void removeItem(String item) {
items.remove(item);
}
public void purchase() {
//. . .
}
@Remove
public void remove() {
items = null;
}
}
```

Questo è un POJO contrassegnato con l'annotazione @Stateful.
Il metodo contrassegnato con l'annotazione @Remove è chiamato quando il bean viene rimosso. Questo metodo è chiamato dal container quando il bean viene rimosso e non è previsto che venga chiamato dal client. Rimuovere uno stateful session bean significa che lo specifico stato dell'istanza per quel cliente è andato.

Questo stile di dichiarazione dei bean è chiamato a no-interface view. Questo tipo di bean è accessibile soltanto localmente ai client pacchettizzati nello stesso archivio. Se il bean deve essere accedibile da remoto, è necessario definire una interfaccia di business separata annotata con @Remote:

```
@Remote
public interface Cart {
public void addItem(String item);
public void removeItem(String item);
public void purchase();
}
@Stateful
public class CartBean implements Cart {
```

```
public float addItem(String item) {
//. . .
}
public void removeItem(String item) {
//. . .
}
//. . .
}
```

In questo modo il bean può essere iniettato usando l'interfaccia:

```
@EJB Cart cart;
```

Accesso da parte del client:

```
@EJB ShoppingCart cart;
cart.addItem("Apple");
cart.addItem("Mango");
cart.addItem("Kiwi");
cart.purchase();
```

Per stateful session beans sono forniti anche i metodi di lifecycle callback PostConstruct e PreDestroy.

Stateless Session Beans

Uno stateless session bean non contiene alcun stato conversazionale per uno specifico client. Tutte le istanze di stateless bean sono equivalenti, il container può scegliere di delegare una invocazione di metodo da parte del client ad ognuna delle istanze.

Uno stateless session bean può essere definito come:

```
@Stateless:
package org.sample;
@Stateless
public class AccountSessionBean {
public float withdraw() {
//. . .
}
public void deposit(float amount) {
//. . .
}
}
```

E' un POJO marcato con l'annotation @Stateless.
Questo stateless session bean può essere acceduto usando l'annotation @EJB.

Se è necessario abilitare l'accesso remoto, è necessario definire una interfaccia di business separata, annotata con @Remote.

Sono supportati i metodi di lifecycle callbacks PostConstruct e PreDestroy.
Il metodo PostConstruct è invocato dopo il costruttore senza argomenti e permetto di iniettare tutte le dipendenze necessarie prima che venga invocato il primo metodo di business.
Il metodo PreDestroy è chiamato prima che l'istanza venga rimossa dal container. In questo metodo tutte le risorse acquisite vengono rilasciate.

Singleton Session Beans

Un singleton session bean è un session bean che è istanziato una volta per applicazioni e fornisce un facile accesso a uno stato condiviso.
Se il container è distribuito su multiple JVM, ogni applicazione avrà una istanza del singleton per ogni JVM. Un singleton session bean è esplicitamente progettato per essere condiviso e supportare la concorrenza.

Un singleton session bean può essere definito usando:

```
@Singleton:
@Singleton
public class MySingleton {
//. . .
}
```

Un singleton bean supporta sempre accessi concorrenti. Di default un singleton bean è settato per una gestione della concorrenza di tipo container-managed, ma può essere settato in alternativa per una concorrenza bean-managed.

Message-Driven Beans

Un message-driven bean (MDB) è un bean container-managed usato per processare messaggi in modo asincrono. Questi bean sono stateless e sono invocati dal container quando un messaggio JMS arriva al destinatario. Un session bean può ricevere un messaggio JMS in modo sincrono, ma un message-driven bean può riceverlo in modo asincrono.

Un POJO può essere convertito in un message-driven bean usando:

```
@MessageDriven:
@MessageDriven(mappedName = "myDestination")
public class MyMessageBean implements MessageListener {
@Override
public void onMessage(Message message) {
try {
// process the message
} catch (JMSException ex) {
//. . .
}
}
}
```

Un message-driven bean può anche spedire messaggi JMS.
Un messaggio è spedito ad un message-driven bean all'interno di un transaction context, questo implica che tutte le operazioni all'interno del metodo onMessage sono parte di una singola transazione.

Portable Global JNDI Names

Un bean locale o no-interface pacchettizzato in un .war è accessibile solo agli altri componenti contenuti nel medesimo .war, ma un bean marcato con l'annotation @Remote è accedibile in modo remoto indipendetemente dalla sua pacchetizzazione. Il file ejb-jar può essere deployato da solo o pacchettizzato all'interno un archivio .ear. I bean pacchettizzati in questo ejb-jar possono essere acceduti in remoto.
Questo EJB può inoltre essere acceduto usando il portable global JNDI name usando la sintassi:

```
java:global[/<app-name>]
/<module-name>
/<bean-name>
[!<fully-qualified-interface-name>]
```

Transactions

Un bean può usare transazioni programmate nel codice del bean, questo viene chiamato bean-managed transaction bean. In alternativa, la modalità di gestione delle transazioni può essere specificata in modo dichiarativo quando le transazioni sono gestite dal container, questo viene chiamato container-managed transaction bean.
La gestione Container-managed è quella di default.

Possibili valori del @TransactionAttribute:

Valore	Descrizione
MANDATORY	Sempre chiamato nel contesto della transazione del client. Se il client chiama dentro un transaction context allora si comporta come REQUIRED. Se il client chiama senza un transaction context, allora il container solleva una eccezione di tipo javax.ejb.EJBTransactionRequiredException.
REQUIRED	Se il client chiama con un transaction context, questo è propagato al bean. Altrimenti il container inizializza una nuova transazione prima di chiamare il metodo di business e tenta di committare la transazione quando il metodo di business è completato.
REQUIRES_NEW	Il container crea sempre un nuovo transaction context prima di chiamare il metodo di business e tenta di

	committare la transazioni al completamento del metodo. Se il client chiama con un transaction context, la transazione viene sospesa e poi ripresa dopo che la nuova transazione è stata committata.
SUPPORTS	Se il client chiama con un transaction context, si compora come REQUIRED. Altrimenti si comporta come NOT_SUPPORTED.
NOT_SUPPORTED	Se il client chiama con un transaction context, il container sospende la transazione e la rilancia prima e dopo l'invocazione del metodo di business. Se il client chiama senza un transaction context, non viene creata nessuna nuova transazione.
NEVER	E' richiesto che il client chiami senza un transaction context. Se il client chiama con un transaction context, il container solleva una eccezione javax.ejb.EJBException. Se il client chiama senza un transaction context, si comporta come NOT_SUPPORTED.

Asynchronous

Ogni metodo di un session bean di default è invocato in modo sincrono. Operazioni asincrono devono avere come valore di ritorno void o Future<V>.

Per definire un metodo asincrono si usa l'annotazione @Asynchronous, se posta a livello di classe indica che tutti i metodi della classe sono asincroni.

In EJB 2.1 è stata introditta la nuova classe AsyncResult che wrappa il risultato di un metodo asincrono come un oggetto di tipo Future. I metodi dell'API Future API sono usati per interrogare la disponibilità di un risultato con isDone o per cancellare l'eseguzione con cancel(boolean mayInterruptIfRunning).

Timers

Un EJB Timer Serviceè un servizio container-managed che permette di schedulare callbacks per eventi time-based.
Un Timers può essere creato in un stateless session beans, in un singleton session beans e in un message-driven beans, ma non in un stateful session beans.
I Timers sono persistenti di default e vanno eventualmente resi non persistenti in modo programmatico settando TimerConfig.setPersistent(false) o automaticamente aggiungendo persistent=false su @Schedule.

Embeddable API

L'API Embeddable EJB permetto al codice client e al corrispondente enterprise beans di essere eseguiti all'interno della stessa JVM e dello stesso class loader.
L'esempio di codice mostra come scrivere un test case che inizializza un container embeddable EJB, recupera l'EJB usando il Portable Global JNDI Name, e invoca un suo metodo:

```
public void testEJB() throws NamingException {
EJBContainer ejbC = EJBContainer.createEJBContainer();
Context ctx = ejbC.getContext();
MyBean bean = (MyBean) ctx.lookup
("java:global/classes/org/sample/MyBean");
assertNotNull(bean);
//. . .
ejbC.close();
}
```

L'embeddable EJB container usa la JVM del classpath per caricare il modulo EJB da caricare. Il client può sovrascrivere questo comportamente durante il setup specificando un set alternativo di moduli target:

```
Properties props = new Properties();
props.setProperty(EJBContainer.EMBEDDABLE_MODULES_PROPERTY,
"bar");
EJBContainer ejbC = EJBContainer.createEJBContainer(props);
```

Questo codice caricherà solo il modulo EJB “bar” all'interno dell'embeddable container.

EJB.Lite

EJB.Lite è un sottoinsieme delle funzioanlità EJB.

Questo permette di usare le API EJB in applicazioni che possono avere installazioni più piccole rispetto ad una tipica applicazione full Java EE.

Differenze fra EJB 3.1 Lite e EJB 3.1 Full API:

	EJB 3.1 Lite	EJB 3.1 Full API
Session beans	✔	✔
Message-Driven beans	✘	✔
2.x/1.x/CMP/BMP Entity beans	✘	✔
Java persistence 2.0	✔	✔
Local / No-interface	✔	✔
3.0 Remote	✘	✔
2.x Remote / Home component	✘	✔
JAX-WS Web service endpoint	✘	✔
JAX-RPC Web service endpoint	✘	✔
EJB Timer service	✘	✔
Asynchronous session bean invocations	✘	✔
Interceptors	✔	✔
RMI-IIOP Interoperability		✔
Container-managed transactions / Bean-managed transactions	✔	✔
Declarative and programmatic security	✔	✔
Embeddable API	✔	✔

Le funzionalità definite da EJB.Lite sono disponibili su application server di tipo Java EE web profile–compliant.
Per il set completo di funzionalità è invece richiesto un application server full Java EE–compliant.

Contexts and Dependency Injection

CDI definisce un meccanismo di type-safe dependency injection nella piattaforma Java EE.
Queste funzionalità rispecchiano il pattern “strong typing, loose coupling” e rendono il codice più facilmente mantenibile.
Il bean così iniettato ha un ben definito ciclo di vita ed è legato ad un lifecycle contexts. Il bean iniettato è anche chiamato contextual instance in quanto è sempre ineittato in un context.

Pressochè ogni POJO può essere iniettato come CDI bean.

CDI permetto a componenti EJB di essere usati come JSF managed beans, colmando in tal modo il gap tra il livello transazionale e quello web. Inoltre è integrato un Unified Expression Language (UEL), che permette di iniettare ogni contextual object direttamente all'interno di pagine JSF o JSP.

Injection Points

Un bean può essere iniettato in un campo, un metodo o un costrutture utilizzando @Inject.

Questa è la sequenza di inizializzazione:

1. Costruttore di default o quello annotato con @Inject.

2. Tutti i campi del bean annotati con @Inject.
3. Tutti i metodi del bean annotati con @Inject.
3. Il metodo di @PostConstruct method, se presente.

Qualifier and Alternative

I qualificatori permettono di specificare in modo unico il bean che deve essere iniettata fra le sue molteplici implementazioni.
Per esempio, questo codice dichiara un nuovo qualificatore, @Fancy:

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

Questo definisce una implementazione dell'interfaccia Greeting:

```
@Fancy
public class FancyGreeting implements Greeting {
public String greet(String name) {
return "Nice to meet you, hello" + name;
}
}
```

si può iniettare un bean di tipo generico Greeting ma specificare che implementa l'interfaccia desiderata @Fancy:

```
@Stateless
public class GreetingService {
@Inject @Fancy Greeting greeting;
public String sayHello(String name) {
return greeting.greet(name);
}
}
```

Questo elimina ogni dipendenza diretta da particolari implementazioni delle interfaccie.
I qualificatori accettano parametri che permettono una ulteriore discriminazione.

Qualificatori CDI Built-in:

Qualifier	Description
@Named	Qualificatore String-based, richiesto per l'uso in Expression Language
@Default	Qualificatore di default per tutti i bean senza un qualificatore esplicito, eccetto @Named
@Any	Qualificatore di default per tutti i bean eccetto @New
@New	Permette all'applicazione di ottenre una nuova istanza indipendentemente dallo scopo dichiarato

*l'uso di @Qualifier è una implementazione del pattern Strategy

I bean marcati con l'annotation @Alternative NON sono disponibili per injection, lookup, o EL resolution.
E' necessario abilitarli espressamente nel beans.xml usando <alternatives>.

```
@Alternative
public class SimpleGreeting implements Greeting {
//. . .
}
@Fancy @Alternative
public class FancyGreeting implements Greeting {
//. . .
}
```

Con queste annotazioni, la seguente injection darà un errore di unresolved dependency:

```
@Inject Greeting greeting;
```

in quanto entrambi i bean sono disabilitati per l'injection. Questo errore può essere risolto esplicitando l'abilitazione nel beans.xml:

```
<beans
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/beans_1_0.xsd">
<alternatives>
<class>org.sample.FancyGreeting</class>
</alternatives>
</beans>
```

@Alternative permette di pacchettizzare multiple implementazioni di un bean con lo stesso qualificatore in un unico .war e abilitarli selettivamente modificando il deployment descriptor in base all'ambiente in cui si sta effettuando il deploy. Questo permetto un polimorfismo di tipo deployment-type.

Producer and Disposer

Un Producer è un metodo annotato con @Produces che restituisce una istanza di un bean, e quando cercheremo di fare injection di quel tipo non sarà più il container a creare l'istanza ma il suddetto metodo.
I metodi producer forniscono poliforfismo a runtime.

Interceptors and Decorators

Interceptor: permette di catturare e separare aspetti ortogonali alla logica applicativa. E' perfetto quindi per risolvere questioni tecniche come la gestione delle transazioni, dei log e della sicurezza. Per natura quindi gli interceptor non sono a conoscenza del contesto che intercettano: semplicemente permettono di eseguire operazioni prima e/o dopo la chiamata ad un metodo di business, senza entrare in merito alla logica del metodo chiamato.
La specifica CDI definisce un meccanismo type-safe per associare gli interceptors ai bean target. E' necessario definire un tipo interceptor binding e può essere fatto tramite la meta-annotation @InterceptorBinding:

```
@InterceptorBinding
@Retention(RUNTIME)
@Target({METHOD,TYPE})
public @interface Logging {
}
```

Di default, tutti gli interceptors sono disabilitati e devono essere esplicitamente abilitati nel file beans.xml:

```
<beans 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/beans_1_0.xsd">
<interceptors>
<class>org.sample.LoggingInterceptor</class>
</interceptors>
</beans>
```

Gli interceptors sono invocati nell'ordine con cui sono specificati dentro l'elemento <interceptors>.
Supportano inoltre la dependency injection.

Decorator: in CDI, un decoratore intercetta la chiamate relative a specifiche interfacce java dell'applicazione, entrando quindi in merito alla logica di business: possiamo pensarli come wrapper del nostro stato applicativo nel quale possiamo modellare e delegare specifiche funzionalità. In sostanza, sono complementari agli interceptors, dai queli si differenziano per specificità d'impiego e conoscenza semantica del contesto in cui agiscono.
Un decorator è un bean che implementa il bean che decora ed è annotato con @Decorator:

```
@Decorator
class MyDecorator implements Greeting {
public String greet(String name) {
//. . .
}
}
```

La classe decorator può essere astratta, oppure non implementare tutti i metodi del bean.

In ordine di esecuzione, gli interceptors per un metodo sono chiamati prima dei decoratori dichiarati per lo stesso metodi.

Scopes and Contexts

Ogni bean è dentro uno scope ed è associato ad un context.

Il runtime assicura che per il giusto scope il bean venga create, se richiesto (il client quindi non deve essere conscio della gestione degli scope).

Ci sono quattro scopes predefiniti e uno di default:

Scope	Description
@RequestScoped	Il bean è legato ad una request. Il bean è disponibile durante una singola request e distrutto quando la request è completata.
@SessionScoped	Il bean è legato ad una sessione. Lo stesso bean è condiviso fra tutte le request all'interno della stessa sessione HTTP, mantiene lo stato a livello di sessione ed è distrutta quando la sessione HTTP raggiunge il times out o è invalidata.
@ApplicationScoped	Il bean è legato ad una applicazione. Il bean è creato allo start dell'applicazione, mantiene lo stato a livello di applicazione ed è distrutta allo shut down dell'applicazione.
@ConversationScoped	Il bean è legato ad una consersazione ed è di due tipi: transient o long-running. Di default, è di tipo transient: è creato con una request JSF ed è distrutto alla fine della request. Una conversazione transient può essere convertita in una long-running usando Conversation.begin. Questa conversazione long-running può essere terminata usando Conversation.end. Tutte le conversazioni long-running sono legate ad una particolare sessione HTTP servlet e può essere propagata as altre request JSF. Dentro una sessione possono essere eseguite conversazioni multiple parallele, ognuno è identificato da un identificativo stringa che è settato dall'applicazione o più generalmente dal container.
@Dependent	Il bean appartiene ad uno pseudoscope. Questo è lo scopo di default per i bean che non dichiarano esplicitamente uno scope.

Stereotypes

Uno stereotype incapsula un pattern architetturale o dei metadati comuni per bean che ricoprono un ruolo ricorrente .
Incapsula lo scope, gli interceptor bindings, i qualifiers, e altre proprietà del ruolo.
Uno stereotype è una meta-annotation annotata con @Stereotype:

```
@Stereotype
@Retention(RUNTIME)
@Target(TYPE)
//. . .
public @interface MyStereotype { }
```

Uno stereotype che aggiunge un comportamento transazionale può essere definito come:

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

Events

Events fornisce un event model annotation-based basato sul pattern observer.
Gli Event producers lanciano eventi che sono consumati dagli observers. L'oggetto evento, tipicamente un POJO, trasporta lo stato dal producer al consumer. Il producer e l'observer sono completamente disacoppiati e comunicano solo usando lo stato.
Un producer bean scatena un evento usando l'interfaccia Event:

```
@Inject @Any Event<Customer> event;
//. . .
event.fire(customer);
```

L'observer bean con la seguente method signature riceverà l'evento:

```
void onCustomer(@Observes Customer event) {
//. . .
}
```

In questo codice, Customer trasporta lo stato dell'evento.
Il producer bean può specificare un insieme di qualificatori quando inietta l'evento:

```
@Inject @Any @Added Event<Customer> event;
```

La signature del metodo che dovrà ricevere l'evento sull'observer bean deve riportare esattamente lo stesso insieme di qualificatori:

```
void onCustomer(@Observes @Added Customer event) {
//. . .
}
```

Possono essere aggiunti anche parametri per i qualificatori, per specificare ulteriormento lo scope del bean observer.

Di default, nel contesto corrente, per consegnare l'evento, è usata una istanza esistente del bean o ne viene creata una nuova. Questo comportamente può essere modificato in modo che l'evento sia consegnato solo se il bean già esiste nello scope corrente:

```
void onCustomer(
@Observes(
notifyObserver= Reception.IF_EXISTS)
@Added Customer event){
//. . .
}
```

I metodi transazionali degli observer ricevono le notifiche degli eventi durant le fasi di before o after completion della transazioni in cui l'evento è sollevato. Questo comportamento può essere modificato attraverso gli identificatori TransactionPhase:

Transactional observers	Descrizione
IN_PROGRESS	Comporameto di default, gli observers sono chiamati immediatamente
BEFORE_COMPLETION	Gli observers sono chiamati durante la fase di before completion della transazione
AFTER_COMPLETION	Gli observers sono chiamati durante la fase di after completion della transazione
AFTER_FAILURE	Gli observers sono chiamati durante la fase di after completion, solo quando la transazione fallisce
AFTER_SUCCESS	Gli observers sono chiamati durante la fase di after completion, solo quando la transazione ha successo

Portable Extensions

CDI espone una Service Provider Interface (SPI) che permette di estendere le funzionalità del container tramite portable extensions. Una portable extension può integrare il container nei seguenti modi:

- Fornendo dei propri beans, interceptors, e decorators al container
- Iniettando dipendenze dentro gli oggetti del container usando il servizio di dependency injection
- Fornendo una implementazione contestuale per uno scope costum
- Aumentando o sovrascrivendo i metadati annotation-based con metadati da altre fonti

JavaServer Faces

JavaServer Faces è un framework per l'implementazione di server-side user interface (UI) per Java-based web applications.
JSF permette di:

- Creare una pagina web con un set di componenti UI riusabili, seguendo il pattern Model-View-Controller (MVC) .
- Lega i compomenti a un modello server-side. Questo permette una comunicazione a due vie dei dati applicativi della UI.
- Gestire la navigazione fra pagine in risposta ad eventi UI e modellare le interazioni.
- Gestire lo stato dei componenti UI attraverso le request del server.
- Fornire un modello semplice per legare gli eventi generati dal cliente al codice applicativo lato server.
- Costruire e riusare facilmente componenti UI custom.

Una applicazione JSF è composta da:

- Un set di pagine web contenenti i componenti UI.
- Un set di managed beans. Un insieme di bean legano i compomenti ad un modello server-side (tipicamenti CDI beans o Managed Beans) e un altro insieme fa da Controller (tipicamente EJB o CDI beans).
- Un deployment descriptor opzionale, web.xml.
- Un file di configurazione opzionale, faces-config.xml.
- Un insieme opzionale di oggetti custom come converters e listeners, creati dallo sviluppatore.

Facelets

Facelets è il linguaggio di dichiarazione della view (view handler) per JSF.
Le pagine facelets sono create utilizzando XHTML 1.0 e Cascading Style Sheets (CSS).
Una semplice pagina facelets può essere definira usando XHTML:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!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:h="http://java.sun.com/jsf/html">
<h:head>
<title>My Facelet Page Title</title>
</h:head>
<h:body>
Hello from Facelets
</h:body>
</html>
```

Facelets è integrato con Expression Language (EL). Questo permetto un data binding two-way fra i backing beans e l'UI:

```
Hello from Facelets, my name is #{name.value}!
```

In questo codice, #{name} è una EL che fa riferimento al campo di un request-scoped CDI bean:

```
@Named
@RequestScoped
public class Name {
private String value;
//. . .
}
```

E' importante aggiungere @Named sul CDI bean per abilitare la sua injection in una espressione EL. E' altamente raccomandato usare CDI-compatible beans invece di bean annotati con @javax.faces.bean.ManagedBean.

Allo stesso modo, si può iniettare un EJB:

```
@Stateless
```

```
@Named
public class CustomerSessionBean {
public List<Name> getCustomerNames() {
//. . .
}
}
```

CustomerSessionBean è uno stateless session bean e ha un metodo di business che ritorna una lista di nomi di customer. L'annotation @Named lo rende iniettabile in una EL e quindi può essere usato in una pagina Facelets:

```
<h:dataTable value="#{customerSessionBean.customerNames}"
var="c">
<h:column>#{c.value}</h:column>
</h:dataTable>
```

Facelets fornisce un potente sistema di templating che permette di creare un look-and-feel consistente fra le diverse pagine dell'applicazione.

Common Facelets tags for a template

Tag	Descrizione
ui:composition	Definisce il layout di una pagina che opzionalmente usa un template. Se viene usato l'attributo template allora i tag figli definiscono il layout del template, altrimenti, è solo un gruppo di elementi che può essere inserito ovunque. Il contenuto fuori da questi tag viene ignorato.
ui:insert	Usato in una pagina di template, definisce il placeholder per inserire contenuto nel template. Il contenuto è preso dal tag ui:define corrispondente nella pagina template client.
ui:define	Usato nella pagina template client: definisce il contenuto che rimpiazzerà il content definito nel template con il corrispondente ui:insert tag.
ui:component	Inserisce un nuovo UI component nel JSF component tree. Ogni componente o frammento di content fuori da questo tag è ignorato.
ui:fragment	Simile a ui:component, ma il contenuto fuori dal tag non viene ignorato.
ui:include	Include il documento referenziato dall'attributo “src” come parte della pagina corrente.

Esempio di pagina template:

```
<h:body>
<div id="top">
<ui:insert name="top">
<h1>Facelets are Cool!</h1>
</ui:insert>
</div>
<div id="content" class="center_content">
<ui:insert name="content">Content</ui:insert>
</div>
<div id="bottom">
<ui:insert name="bottom">
<center>Powered by GlassFish</center>
</ui:insert>
</div>
</h:body>
```

In questo codice, la pagina definisce la struttura usando <div> e CSS. ui:insert definisce il content che verrà sostituito dalla pagina template client.

Pagina template client:

```
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:ui="http://java.sun.com/jsf/facelets"
xmlns:h="http://java.sun.com/jsf/html">
```

```
<body>
<ui:composition template="./template.xhtml">
<ui:define name="content">
<h:dataTable
value="#{customerSessionBean.customerNames}"
var="c">
<h:column>#{c.value}</h:column>
</h:dataTable>
</ui:define>
</ui:composition>
</body>
</html>
```

In questo codice, non sono definite le sezioni ui:insert per top e bottom, quindi verrano usate le corrispondeti sezioni definite nella pagina di template. E' invece presente l'elemento ui:define con un nome che corrisponde all'elemento ui:insert del template, quindi il content viene sostituito.

Resource Handling

JSF definisce una modalità standard per gestire le risorse come immagini, CSS, o file JavaScript.
Queste risorse possono essere pacchettizzate nella directory /resources della web application o in /META-INF/resources nel classpath. Le risorse possono anche essere localizzate, versionate e raccolte in librerie.
Una risorsa può essere referenziata in una EL:

```
<a href="#{resource['header.jpg']}">click here</a>
```

Composite Components

Un composite component è un compomente composto da ono o più componenti JSF definito in un Facelets markup file.
Questo file .xhtml viene posto dentro ad una resource library.
Il composite component è defnito nella defining page e usato nella using page. La defining page defnisce i metadata (o parameters) usando <cc:interface> e l'implementazione usando <cc:implementation>, cc:interface definisce i metadata che descrivono le caratteristiche del component, come gli attributi supportati, e punti di aggancio per gli event listeners. cc:implementation contiene il contains i sostituti del markup per il composite component.
Per esempio, diciamo che il codice necessita di passare diverse value expressions (invece di #{user.name}) e invocare diversi metodi (invece di #{userService.register}) quando il bottone submit è cliccato in diverse using page.
La defining page può quindi passare i valori:

```
<!-- INTERFACE -->
<cc:interface>
<cc:attribute name="name"/>
<cc:attribute name="password"/>
<cc:attribute name="actionListener"
method-signature=
"void action(javax.faces.event.Event)"
targets="ccForm:loginButton"/>
</cc:interface>
<!-- IMPLEMENTATION -->
<cc:implementation>
<h:form id="ccForm">
<h:panelGrid columns="3">
<h:outputText value="Name:" />
<h:inputText value="#{cc.attrs.name}" id="name"/>
<h:message for="name" style="color: red" />
<h:outputText value="Password:" />
<h:inputText value="#{cc.attrs.password}"
id="password"/>
<h:message for="password" style="color: red" />
</h:panelGrid>
<h:commandButton id="loginButton"
action="status"
value="submit"/>
</h:form>
</cc:implementation>
```

In questo codice, tutti i parametri sono esplicitamente specificati in cc:interface. Il terzo parametro ha un targets attribute che fa riferimento a ccForm:loginButton.
Nella cc:implementation:

- h:form ha l'attributo id necessario perchè il bottono possa essere espressamente referenziato.
- h:inputText usa #{cc.attrs.xxx} come default EL expression e permette di accedere agli attributi del composite component. In questo caso #{cc.attrs} ha nome e password definiti come attributi.
- ActionListener è il punto di ingresso per l'event listener. E' definito come una method-signature e descrive la

signature del metodo.
- h:commandButton ha un attributo id così può essere chiaramente identificato dentro a h:form.

User, password, e actionListener sono quindi passati come attributi required nella using page:

```
<ez:login
name="#{user.name}"
password="#{user.password}"
actionListener="#{userService.register}"/>
```

Ora la using page può passare differenti backing beans, e possono essere invocati diversi business methods al click del bottone di submit.

Complessivamente, i composite component forniscono i seguenti vantaggi:
- Seguono il pattern Don't Repeat Yourself (DRY) e permettono di tenere il codice che può essere ripetuto i un singolo file.
- Permetto agli sviluppatore di creare nuovi componenti senza bisogno di codice Java a configurazioni XML.

Ajax

JSF fornisce un supporto nativo per l'aggiunta di funzionalità Ajax alle pagine web.
Questo permette il partial view processing, quando solo alcuni dei compomenti della pagina sono necessari per precessare la response. Permette anche il partial page rendering, cioè il rendering di solo alcuni componenti della pagina.
Ci sono due modi per attivare queste funzionalità:
- Programmaticamente usando risorse JavaScript
- Dichiarativamente usando f:ajax. Questo tag può essere innestato in un singolo componente o può essere "wrapped" attorno ad un gruppo di components.

HTTP GET

JSF fornisce supporto per il mapping dei parametri in URL in HTTP

Server and Client Extension Points

Converters, validators, e listeners sono attached object server-side che aggiungono funzionalità ai componenti delle pagine.
Behaviors sono client-side extension points che possono migliorare il rendering del contenuto di un componente con un behavior-defined scripts.

JSF fornisce anche un integrazione built-in con i constraints definiti usando la Bean Validation. Non è richiesto codice aggiuntivo oltre alla definizione delle annotation constraints sui bean.

A differenza di converters, validators, e listeners, un behavior migliora le funzionalità client-side di un componente dichiarando script da attaccarci. Per esempio, f:ajax è definito come un clientside behavior. Questo pemette anche ti effettuare validazioni e logging client-side, mostrare tooltips, e altre funzionalità simili.
Si possono definire Custom behaviors estendendo ClientBehaviorBase e usando l'annotazione @FacesBehavior.

Navigation Rules

JSF definisce navigation rules implicite ed esplicite.
Le navigation rules implicite cernao l'outcome per una azione, se viene trovata una pagina Facelets che corrisponde all'outcome, viene renderizzata:

```
<h:commandButton action="login" value="Login"/>
```

In questo codide, cliccando sul bottone verrà renderizzata la pagina login.xhtml contenuta nella stessa directory.

Regole di navigazione esplicite possono essere specificate usando il tag <navigation-rule> nel file faces-config.xml. La navigazione condizionale può essere specificata usando il tag <if>:

```
<navigation-rule>
<from-view-id>/index.xhtml</from-view-id>
<navigation-case>
<from-outcome>success</from-outcome>
<to-view-id>/login.xhtml</to-view-id>
<if>#{user.isPremium}</if>
```

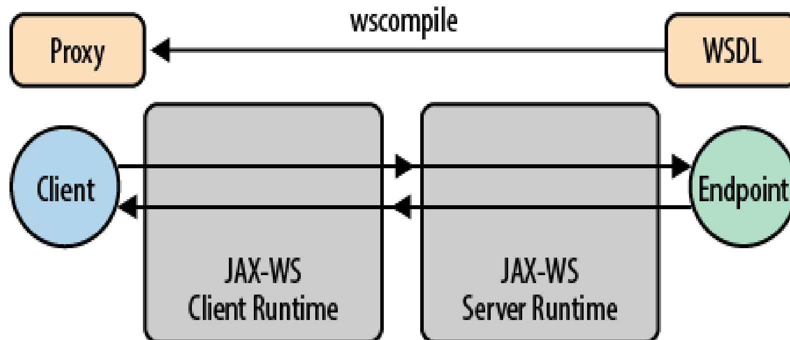
```
</navigation-case>
</navigation-rule>
```

In questo codice, la navigazione da index.xhtml a login.xhtml avviene solo se lo user è un premium customer.

SOAP-Based Web Services

SOAP è un XML-based messaging protocol usato come formato di dati per scambiare informazioni fra web services. La specifica SOAP definisce una busta (envelope) che rappresenta il contenuto del messaggio SOAP e regole di encoding per i tipi di dati. Definisce inoltre come il messaggio SOAP può essere spedito su diversi protocolli di trasporto, come lo scambio di messaggi come payload di un HTTP POST.
Il protocollo SOAP fornisce un modo per comunicare fra applicazioni eseguite su diversi sistemi operativi, con differenti tecnologie e linguaggi di programmazione.

La Java API for XML-Based Web Services (JAX-WS) nasconde la complessita del protocollo SOAP protocol e fornisce API più semplici:



Il Data mapping fra Java e XML è definito usando Java API for XML Binding (JAXB).

Web Service Endpoints

Un POJO può essere convertito in un SOAP-based web service endpoint aggiungendo l'annotation @WebService:

```
@WebService
public class SimpleWebService {
public String sayHello(String name) {
return "Hello " + name;
}
}
```

Tutti i metodi public della classe sono esposti come operazioni del web service. Questo è chiamato Service Endpoint Interface (SEI)–based endpoint.

@WebService attributes

Attributes	Description
endpointInterface	Fully qualified class name dell'interfaccia del service endpoint che definisce il nome del contratto di servizio astratto per questo web service (wsdl:portType)
portName	Port name del web service (wsdl:port)
serviceName	Namespace del web service (targetNamespace)
targetNamespace	Service name del web service (wsdl:service)
wsdlLocation	Posizione del WSDL predefinito che descrive il servizio

L'annotation @WebMethod può essere usata sui singoli metodi per sovrascrivere i valori di default:

```
@WebMethod(operationName="hello")
public String sayHello(String name) {
return "Hello " + name;
}
```

In questo modo viene sovrascritto il nome di default presente in wsdl:operation per questo metodo.
Se un metodo della classe è annotato con @WebMethod, tutti gli altri metodi della classe sono implicitamente non disponibilo come SEI endpoint. Ogni altro metodo da esporre deve essere a sua volta annotato.

Il mapping fra i tipi Java e l'XML è delegato a JAXB che segue il mapping di default Java-to-XML e XML-to-Java mapping per ogni parametri e tipo di ritorno.

Di defualt, un messaggio segue il patter request response (una response ricevuta per ogni request). Si può modificare questo comportamente e far seguire ad un metodo il patter fire and forget specificando l'annotation @Oneway, così può essere spedita la request dal message ma non viene ricevuta la response.

Provider-Based Dynamic Endpoints

I Provider-based endpoint sono una alternativa dinamica ai SEI-based endpoint. Invece di mappare tipi Java, si possono usare direttamente gli elementi del protocollo come Source, DataSource, o SOAPMessage nell'endpoint.
Il messaggio di response deve essere preparato anch'esso usando questa API.

Web Service Client

Il contratto tra il web service endpoint e il client è definito nel.
Un high-level web service client può essere generato importanto il WSDL. Il tool segue il mapping WSDL-to-Java defiito dalla specifica JAX-WS e genera la classi corrispondenti.

WSDL-to-Java mappings

WSDL element	Java class
wsdl:service	Service class che estende javax.xml.ws.Service; fornisce al client la visione del web service.
wsdl:portType	Service endpoint interface.
wsdl:operation	Metodi Java del corrispondente SEI.
wsdl:input	Parametri del metodo Java (Wrapper o nonwrapper style).
wsdl:output	Valore di ritorno del metodo Java (Wrapper o nonwrapper style).
wsdl:fault	Service-specific exception.
XML schema elements in wsdl:types	Come definito nel mapping XML-to-Java nella specifica JAXB.

Una nuova istanza del proxy può essere generata chiamando uno dei metodi getPort della classe Service generata:

```
@WebServiceClient(name="...",
targetNamespace="...",
wsdlLocation="...")
public class SimpleWebServiceService
extends Service {
URL wsdlLocation = ...
QName serviceQName = ...
public SimpleWebService() {
super(wsdlLocation, serviceQName);
}
//. . .
public SimpleWebService getSimpleWebServicePort() {
return super.getPort(portQName,
SimpleWebService.class);
}
}
```

Il client può quindi invocare il metodi di business del web service:

```
SimpleWebServiceService service =
new SimpleWebServiceService();
SimpleWebServicePort port =
service.getSimpleWebServicePort();
port.sayHello("Duke");
```

Può anche essere usato un metodo getPort più generico:

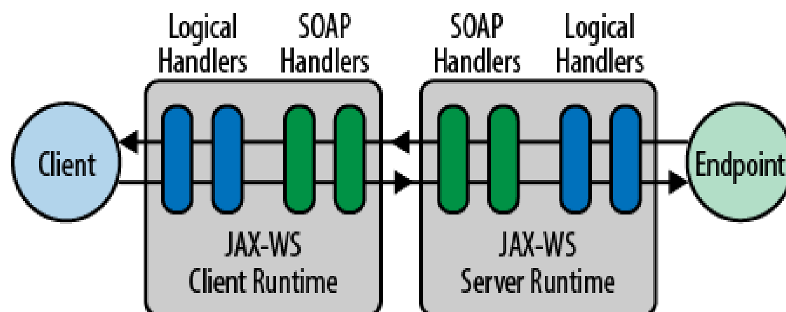
```
SimpleWebServiceService service =
new SimpleWebServiceService();
SimpleWebServicePort port = service.getPort(
SimpleWebService.class);
port.sayHello("Duke");
```

Handlers

Gli Handlers sono punti di estensione ben definiti che effettuano operazioni aggiuntive sui messaggi di request e response.

Si dividono in logical handler e protocol handel.
I Logical handlers sono indipendenti dal protocollo usato e non possono cambiare nessuna parte del messaggio che dipenda dal protocollo.
I Protocol handlers sono specifici per un protocollo e possono accedere o cambiare aspetto protocol-specific del messaggio.

JAX-WS logical and SOAP handlers:



RESTful Web Services

REST è uno stile architetturale per creare servizi che utilizzano standard web. I web service progettati usando REST sono chiamati RESTful web service.
I principi fondamentali dei RESTful Web Services sono:

- Ogni cosa può essere identificata come una risorsa e ogni risorsa è identificabile univocamente usando un URI.
- Una risorsa può essere rappresentata in diversi formati, definiti dal tipo di media. Il tipo di media fornirà le informazioni su come la request deve essere generata. Sono definiti metodi standard per il client e il server per nagoziare sul content type della risorsa.
- Uso di metodi standard HTTP per interagire con le risorse:
  - GET per recuperare una risorsa,
  - POST per creare una risorsa,
  - PUT per aggiornare una risorsa,
  - DELETE per cancellare una risorsa.
- La comunicazione fra client ed endpoint è stateless. Tutti gli stati associati richiesti dal server devono essere passati dal client ad ogni invocazione.

La Java API for RESTful web services (JAX-RS) definisce una API standard annotation-driven che aiuta gli sviluppatori a

creare RESTful web service in Java

Simple RESTful Web Services

Un semplice RESTful web service può essere definito usando @Path:

```
@Path("orders")
public class Orders {
@GET
public List<Order> getAll() {
//. . .
}
@GET
@Path("{oid}")
public Order getOrder(@PathParam("oid")int id) {
//. . .
}
}
@XmlRootElement
public class Order {
int id;
//. . .
}
```

Tipicamene, una risorsa RESTful è impacchettata in un .war insieme alle classi e risorse. La classe Application e l'annotation @ApplicationPath sono usate per specificare il path di base per tutte le risorse RESTful presenti nell'archivio. La classe Application fornisce anche metadati aggiuntivi riguardo all'applicazione.

Diciamo che un POJO è pacchettizzato nel file store.war file, deployato su localhost:8080, e la Application class è definita come:

```
@ApplicationPath("webresources")
public class ApplicationConfig extends Application {
}
```

Una lista di tutti gli ordini accessibili tramite una richiesta di GET sarà:

```
http://localhost:8080/store/webresources/orders
```

Uno specifico ordine può essere recuperato aggiungendo un parametro alla GET:

```
http://localhost:8080/store/webresources/orders/1
```

Il valore 1 sarà passato al metodo getOrder' come parametro.
Avere l'annotazione @XmlRootElement assicura che sarà ritornata una rappresentazione XML della risorsa.

Un URI può passare parametri HTTP usando coppie nome/valore. Queste possono essere mappate come parametri dei metodi della risorsa o come campi usando l'annotation @QueryParam.

```
public List<Order> getAll(@QueryParam("start")int from,
@QueryParam("page")int page) {
//. . .
}
```

Questa risorsa sarà acceduta come:

```
http://localhost:8080/store/webresources/orders?
start=10&page=20
```

Binding HTTP Methods

JAX-RS fornisce supporto per i metodi standard HTTP GET, POST, PUT, DELETE, HEAD, e OPTIONS usando corrispondenti annotation:

HTTP method	JAX-RS annotation
GET	@GET
POST	@POST
PUT	@PUT

DELETE	@DELETE
HEAD	@HEAD
OPTIONS	@OPTIONS

Consideriamo la seguente form HTML, che prende l'identificatore dell'ordine e il nome del customer e crea un ordine postando il form a webresources/orders/create:

```
<form method="post" action="webresources/orders/create">
Order Number: <input type="text" name="id"/><br/>
Customer Name: <input type="text" name="name"/><br/>
<input type="submit" value="Create Order"/>
</form>
```

La risorsa:

```
@POST
@Path("create")
@Consumes("application/x-www-form-urlencoded")
public Order createOrder(@FormParam("id")int id,
@FormParam("name")String name) {
Order order = new Order();
order.setId(id);
order.setName(name);
return order;
}
```

L'annotation @FormParam associa il valore di un parametro di una form HTML al parametro di un metodo di una risorsa, o ad un suo campo. Il nome dell'attributo e il valore dell'annotation @FormParam devono essere identici. Cliccando il bottone submit viene ritornata una rappresentazione XML del nuovo ordine creato.

Comportamente analogo per il gli altri metodi e corrispondenti annotazioni.

Il metodo HTTP HEAD è identico al GET ad eccezione del fatto che non viene tornato nessun body nella response.
Il metodo HTTP OPTIONS effettua una requests per ottenere le opzioni di comunicazione disponibile sulla request/response identificata dall'URI.

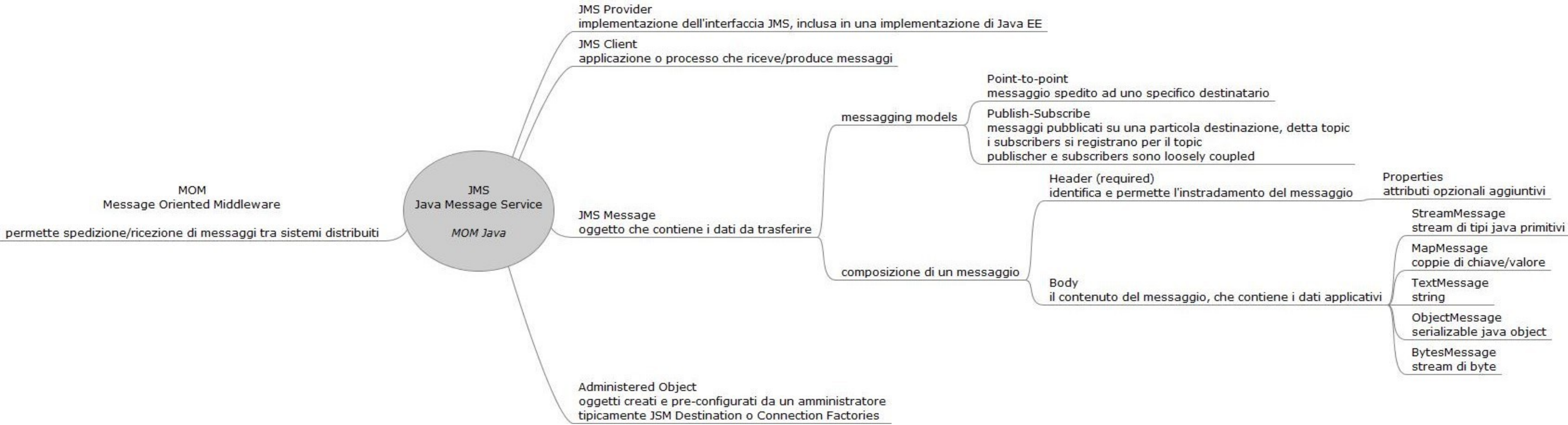
Multiple Resource Representations

Di default, una risorsa RESTful è pubblicata o consumato con il MIME type */*.
Una risorsa RESTful può restringere l'insieme di media supportati dalle request e dalle response usando rispettivamente le annotation @Consumes e @Produces. Queste annotazioni possono essere specificate sulla classe risorsa o sui singoli metodi della risorsa (le annotazioni specificate su un metodo sovrascrivono quelle a livello di classe).

Un esempio che mostra come la risorsa Order possa essere pubblicata usando MIME types multipli:

```
@GET
@Path("{oid}")
@Produces({"application/xml", "application/json"})
public Order getOrder(@PathParam("oid")int id) { . . . }
```

Il metodo getOrder può generare una rappresentazione di Order XML o JSON. L'esatto tipo di ritorno è determinato dall'header HTTP Accept nella request.


MOM
Message Oriented Middleware
permette spedizione/ricezione di messaggi tra sistemi distribuiti
JMS
Java Message Service
MOM Java
JMS Provider
implementazione dell'interfaccia JMS, inclusa in una implementazione di Java EE
JMS Client
applicazione o processo che riceve/produce messaggi
JMS Message
oggetto che contiene i dati da trasferire
messaging models
Point-to-point
messaggio spedito ad uno specifico destinatario
Publish-Subscribe
messaggi pubblicati su una particola destinazione, detta topic
i subscribers si registrano per il topic
publischer e subscribers sono loosely coupled
composizione di un messaggio
Header (required)
identifica e permette l'instradamento del messaggio
Properties
attributi opzionali aggiuntivi
Body
il contenuto del messaggio, che contiene i dati applicativi
StreamMessage
stream di tipi java primitivi
MapMessage
coppie di chiave/valore
TextMessage
string
ObjectMessage
serializable java object
BytesMessage
stream di byte
Administered Object
oggetti creati e pre-configurati da un amministratore
tipicamente JSM Destination o Connection Factories

Java Message Service

Bean Validation

La Bean Validation permette di dichiarare constraint a livello di classe per facilitare la validazione dei dati.
I constraints possono essere definiti con annotazioni poste su campi, property, parametri di metodi o classi. I constraints possono essere definiti su interfaccie o superclassi.
Validation constraints e informazioni di configurazione possono anche essere definiti tramite file XML validation.xml posto in META-INF/. I descrittori sovrascrivono ed estendono i metadati definiti usando le annotation.

Built-in Constraints

Tutti i validatori built-in sono definiti nel package javax.validation.constraints:

@Null	L'elemento annotato deve essere nulla. Può essere applicato ad ogni tipo.	`@Null` `String httpErrorCode;`
@NotNull	L'elemento annotato non può essere nullo. Può essere applicato ad ogni tipo. Scatena un errore di validazione se la variabile si istanza è assegnata ad un valore nullo.	@NotNull String name;
@AssertTrue	L'elemento annotato deve essere true. Applicabile solamente ai tipi boolean o Boolean.	@AssertTrue boolean isConnected;
@AssertFalse	L'elemento annotato deve essere false. Applicabile solamente ai tipi boolean e Boolean.	@AssertFalse Boolean isWorking;
@Min, @DecimalMin	L'elemento annotato deve essere un numero il cui valore è maggiore o uguale al minimo specificato. Supporta i tipi byte, short, int, long, Byte, Short, Integer, Long, BigDecimal, e BigInteger	@Min(10) int quantity;
@Max, @DecimalMax	L'elemento annotato deve essere un numero il cui valore è minore o uguale al massimo specificato. Supporta i tipi byte, short, int, long, Byte, Short, Integer, Long, BigDecimal, e BigInteger. Si possono definire più vincoli per lo stesso campo.	@Max(20) int quantity; @Min(10) @Max(20) int quantity;
@Size	La lunghezza dell'elemento deve essere compresa nei limiti definiti. Supporta i tipi String, Collection, Map, Array. Di default min è 0 e max è 2147483647.	@Size(min=5, max9) String zip; @Size(min=1) @List<Item> items;
@Digits	L'elemento annotato deve essere un numero all'interno di un range stabilito. Supporta i tipi byte, short, int, long, Byte, Short, Integer, Long, BigDecimal, BigInteger, e String.	@Digits(integer=3,fraction=0) int age; @Min(18) @Max(25) @Digits(integer=3,fraction=0) int age;
@Past	L'elemento annotato deve essere una data passata. Il tempo presente è definito come il tempo corrente della virtual machine. Supporta i tipi Date e Calendar.	@Past Date dob;
@Future	L'elemento annotato deve essere una	@Future

	data futura. Il tempo presente è definito come il tempo corrente della virtual machine. Supporta i tipi Date e Calendar.	Date retirementDate;
@Pattern	La string annotata deve matchare una regular expression.	@Pattern(regexp="[0-9]*") String zip; @Pattern(regexp="[0-9]*") @Size(min=5, max=5) String zip;

Ogni constraint declaration può anche sovrascrivere i campi message, group, e payload. Message è usato per sovrascrivere il messaggio di errore di default, ritornato quando viene violato il vincolo. Group è usato per sovrascrivere il default validation group. Payload è usato per associare metadati al vincolo.

Defining a Custom Constraint

Si possono definire custom constraints combinando annotation e implementazioni di custom validation.
Esempio di codice che crea un custom constraint per validare uno zip code:

```
@Documented
@Target({ ElementType.ANNOTATION_TYPE,
ElementType.METHOD,
ElementType.FIELD,
ElementType.CONSTRUCTOR,
ElementType.PARAMETER })
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy=ZipCodeValidator.class)
@Size(min=5, message="{org.sample.zipcode.min_size}")
@Pattern(regexp="[0-9]*")
@NotNull(message="{org.sample.zipcode.cannot_be_null}")
public @interface ZipCode {
String message() default
"{org.sample.zipcode.invalid_zipcode}";
Class<?>[] groups() default {};
Class<? extends Payload>[] payload() default {};
Country country() default Country.US;
public enum Country {
US,
CANADA,
MEXICO,
BRASIL
}
}
```

In questo codice:

- @Target definisce che questo vincolo può essere applicato su tipi, metodi, campi, costruttori e parametri di metodi.
- @Constraint definisce che l'annotation è la definizione di un constraint. Crea anche un link con la sua implementazione, definita dall'attributo validatedBy. La classe ZipCodeValidator.class fornisce l'implementazione del validatore. Possono essere specificate diverse implementazioni usando in array di classi.
- @Size, @Pattern, e @NotNull sono vincoli primitivi usati per comporre il vincolo custom. Annotare un elemento con @ZipCode (l'annotazione composta) è equivalente ad annotarla con @Size, @Pattern, e @NotNull. Di default, ogni violazione di una annotazione composta solleva un report di errore individuale. Tutti i reports sono riuniti insieme e viene riportata ogni violazione. Comunque, può essere usata l'annotation @ReportAsSingleViolation per sopprimere i singoli report di errore generati dagli elementi che compongono l'annotazione composta, in questo caso è generato un error report direttamente dall'annotazione composta. In questo codice, il messaggio è legato ad una chiave per permettere l'internazionalizzazione.
- Group specifica il validation group. Questo è usato per effettuare una validazione parziale del bean o per controllare l'ordine con il quale le validazioni sono effettuate. Di default, il valore è un array vuoto e l'appartenenza è al Default group.
- payload è usato per associare informazioni di metadati al vincolo.
- country è definito come un elemento aggiuntivo per parametrizzare il vincolo. Il set di valori possibili per questo parametro è definito come un enum ed è anche specificato un valore di default (Country.US).

Implementazione del validatore:

```
public class ZipCodeValidator
implements ConstraintValidator<ZipCode, String> {
List<String> zipcodes;
@Override
public void initialize(ZipCode constraintAnnotation) {
zipcodes = new ArrayList<String>();
switch (constraintAnnotation.country()) {
case US:
zipcodes.add("95054");
zipcodes.add("95051");
zipcodes.add("94043");
break;
case CANADA:
//
break;
case MEXICO:
//
break;
case BRASIL:
//
break;
}
}
@Override
public boolean isValid(
String value,
ConstraintValidatorContext context) {
return zipcodes.contains(value);
}
}
```

In questo codice:

- L'implementazione del validatore implementa l'interfaccia ConstraintValidator. Nella definizione del vincoli si definisce che può essere applicato a più tipi Java, questo richiede di definire una implementazione per ogni tipo. Questo validatore può essere solo applicato a tipi String.
- Il metodo initialize inizializza tutte le risorso o le strutture dati da usare per la validazione. E' garantito che questo metodo venga chiamato prima dell'uso dell'stanza per la validazione.
- Il metodo isValid implementa la logica di validazione. Ritorna true se il vincolo è rispettato, false altrimenti. Il parametro value è l'oggetto da validare e ConstraintValidatorContext fornisce il constesto in cui la validazione è eseguita. L'implementazione di questo metodo deve essere thread-safe.

Se un bean X contiene un field di tipo Y, di default la validazione del tipo X non scatena la validazione del tipo Y.
Questo comportamente può essere modificato annotando il fiels con @Valid, questo farà in modo che la validazione sia effettuata a cascata anche sul tipo Y.

@Valid fornisce anche polimorfismo per la validazione: se il campo Y è una interfaccia o una classe astratta, allora il vincoli di validazione applicato a runtimer è quello della reale implementazione o sottotipo.

Ogni campo e properties iterabile può essere decorato con @Valid per fare in modo che tutti gli elementi dell'iteratore vengano validati. @Valid è applicato ricorsivamento, così ogni elemento dell'iteratore viene a sua volta validato:

```
public class Order {
@Pattern(...)
String orderId;
@Valid
private List<OrderItem> items;
}
```

In questo codice, la lista di order items è validata ricorsivamente insieme al campo orderId.
Se non fosse specificato @Valid sarebbe validato solo il campo orderId alla validazione del bean.

Validation Groups

Di default, tutti i constraints sono definiti nel Default validation group.
Un constraint può essere definito in un diverso validation group, esplicitamente creato per effettuare una validazione parziale del bean o per controllare l'ordine con il quale i vincoli sono valutati.

Un validation group è definito come una interfaccia:

```
public interface ZipCodeGroup {
}
```

A questo validation group può ora essere assegnata la definizione di un vincolo:

```
@ZipCode(groups=ZipCodeGroup.class)
String zip;
```

In questo codice, zip sarà validato solo quando il target per la validazione è lo ZipCodeGroup validation group.
Di default, il Default validation group non è incluso se viene specificato un set specifico di gruppi:

```
@ZipCode(groups={Default.class,ZipCodeGroup.class})
String zip;
```

I grouppi possono ereditare a altri gruppi usando l'ereditarietà fra interfacce.
Può essere definito un nuovo gruppo composto dai gruppi Default e ZipCodeGroup:

```
public interface DefaultZipCodeGroup
extends Default, ZipCodeGroup {
}
```

Questo nuovo gruppo può essere specificato come parte di un vincolo ed è semanticamente equivalente allo specificare i due gruppi separatamente:

```
@ZipCode(groups=DefaultZipCodeGroup.class)
String zip;
```

Integration with JPA

Le JPA-managed classes (entities, mapped superclasses, e embeddable classes) possono essere configurate per includere validation constraints.
Di default, tutti i constraints sono validati durante le fasi pre-persist, pre-update, e pre-remove.

Definizione di una JPA entity con vincoli di validazione:

```
@Entity
public class Name {
@NotNull
@Size(4)
private String name;
@Min(16)
@Max(25)
private int age;
//. . .
}
```

Il comportamente di default del validatore può essere cambiato specificando l'elemento validation-mode nel file persistence.xml.
Values for validation-mode in persistence.xml:

validation-mode	Description
auto	Validazione automatica delle entities; comportamento di default. Non viene effettuata nessuna validazione se non è fornito un Bean Validation provider.
callback	Validazione nel Lifecycle. E' sollevato un errore se non è fornito un Bean Validation provider.
none	Nessuna validazione.

Questi attibuti possono essere specificati nel persistence.xml:

```
<persistence-unit name="MySamplePU" transaction-type="JTA">
<jta-data-source>jdbc/sample</jta-data-source>
<exclude-unlisted-classes>
false
</exclude-unlisted-classes>
<validation-mode>CALLBACK</validation-mode>
<properties/>
</persistence-unit>
```

Questi valori possono anche essere specificati usando la property javax.persistence.validation.mode se l'entity manager factory è creata usando Persistence.createEntityManagerFactory:

```
Map props = new HashMap();
props.put("javax.persistence.validation.mode", "callback");
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("MySamplePU",
props);
```

Di default, ogni entity è messa nel Default validation group.
Il Default group è il target degli eventi di pre-persist e pre-update, nessun gruppo è targer per gli eventi di pre-remove. Quindi i vincoli sono validati quando una entity è creata o aggiornata, ma non quando viene cancellata.

Possono essere definiti diversi gruppo di validazione per gli eventi del ciclo di vita specificando le properties:

- javax.persistence.validation.group.per-persist
- javax.persistence.validation.group.pre-update

Questo properties sono usate nel persistence.xml:

- javax.persistence.validation.group.pre-remove

```
<persistence-unit name="BeanValidationPU"
transaction-type="JTA">
<jta-data-source>jdbc/sample</jta-data-source>
<exclude-unlisted-classes>
false
</exclude-unlisted-classes>
<validation-mode>CALLBACK</validation-mode>
<properties>
<property name=
"javax.persistence.validation.group.pre-persist"
value="org.sample.MyPrePersistGroup"/>
<property name=
"javax.persistence.validation.group.pre-update"
value="org.sample.MyPreUpdateGroup"/>
<property name=
"javax.persistence.validation.group.pre-remove"
value="org.sample.MyPreRemoveGroup"/>
</properties>
</persistence-unit>
```

Queste proprietà sono anche passate al Persistence.createEntityManagerFactory in una mappa.
Se un vincolo è violato, la transazione corrente è marcata per il rollback.

Integration with JSF

Una applicazione JSF tipicamente consiste di multiple pagine Facelets e i corrispondenti backing beans per catturare i dati da queste pagine. Ogni vincolo definito su un backing bean è automaticamente processato durante la fase di process validations.
La validazione standard javax.faces.Bean assicura anche che ogni violazione del vincolo venga wrappata in un FacesMessage e aggiunta al FacesContext. Questo messaggio è visualizzato all'utente come gli altri messaggi di validazione.