JSR 299: Web Beans

Web Beans Expert Group

Version: Public Review

Table of Contents

Chapter 1. Architecture

Web Beans provides a powerful new set of services to Java EE components. This specification defines:

- The lifecycle and interactions of stateful components bound to well-defined *contexts*, where the set of contexts is extensible
- A sophisticated, typesafe *dependency injection* mechanism, including a facility for choosing between various components that implement the same Java interface at deployment time
- Integration with the Unified Expression Language (EL), allowing any component to be used directly within a JSF or JSP page
- Generalization of the method and component lifecycle *interceptors* defined by EJB 3.0 to other kinds of components, along with an improved approach to binding interceptors to components and a new type of interceptor, called a *decorator*
- An *event notification* model
- A web *conversation* context in addition to the three standard web contexts defined by the Java Servlet specification
- An SPI allowing third-party frameworks that execute in the Java EE environment to integrate cleanly with Web Beans

To take advantage of these facilities, the Java EE component developer provides additional component-level and application-level metadata in the form of Java annotations and/or XML-based deployment descriptors.

A Java EE component with a lifecycle bound to a Web Beans context is called a *Web Bean*. Any Web Bean may be injected into other Java EE components by the Web Beans dependency injection service.

The use of Web Beans significantly simplifies the task of creating Java EE applications by integrating the Java EE web tier with Java EE enterprise services. In particular, Web Beans allows EJB 3 components to be used as JSF managed beans, thus integrating the the component models of EJB and JSF and significantly simplifying the programming model when EJB and JSF are used together. In an environment that supports Web Beans, all EJB 3 session beans are Web Beans—no Web Beans specific metadata is required.

Furthermore, Web Beans makes it easy to use most plain Java classes as Java EE components that may inject, or be injected into, other Java EE components such as EJBs or Servlets. Web Beans promotes plain Java classes to the status of managed Java EE components. In particular, in an environment that supports Web Beans, all JavaBeans are Web Beans—no Web Beans specific metadata is required.

Even when EJB, or JSF, or both, are *not* used, Web Beans may be used to simplify development of the business-logic layer of an application. It is even possible for applications developed using third-party frameworks to take advantage of the services provided by Web Beans via a framework integration SPI.

1.1. Contracts

This specification defines the responsibilities of a user who writes an application that executes inside an environment that supports Web Beans and uses the functionality provided by Web Beans, along with responsibilities of a vendor who implements the functionality defined by this specification and provides a runtime environment in which Web Beans execute — the *container*.

The container may be a Java EE container or an embeddable EJB Lite container.

1.2. Supported environments

A Web Bean application may be designed to execute in either the Java EE 6, Java EE 5 or Java SE environments. When a Web Bean application executes in a Java SE environment, the embeddable EJB Lite container provides Java EE services such as transaction management and persistence.

Any Java EE 5 compliant container may support Web Beans. However, certain functionality defined by this specification is optional for Java EE 5 containers. This is the case only when explicitly noted in this specification.

Java EE 6 and embeddable EJB Lite containers must support all functionality defined by this specification.

1.3. Relationship to other specifications

An application developer using Web Beans creates Java EE components such as EJBs, Servlets and JavaBeans and then provides additional Web Beans metadata that defines additional behavior in terms of the Web Beans context model. These components may take advantage of the services defined by this specification, together with the enterprise and presentational aspects defined by other Java EE platform technologies.

In addition, this specification defines an SPI that allows alternative, non-platform technologies to integrate with Web Beans, for example, alternative web presentation technologies.

Open issue: the Web Beans annotations for dependency injection, scope and interceptor binding are currently defined in the package `javax.webbeans`. To make these annotations more easily consumable by other specifications, should they instead be categorized by concern into packages such as `javax.dependency`, `javax.contexts` and `javax.interceptors`?

1.3.1. Relationship to EJB

EJB defines a programming model for application components that access transactional resources in a multi-user environment. EJB allows concerns such as role-based security, transaction demarcation, concurrency and scalability to be specified declaratively using annotations and XML deployment descriptors and enforced by the EJB container at runtime.

EJB components may be stateful, but are not by nature contextual. References to stateful component instances must be explicitly passed between clients and stateful instances must be explicitly destroyed by the application.

Any EJB bean obtained via the Web Beans dependency injection service is a contextual object. It is bound to a context and available to other Web Beans that execute in that context. The container automatically creates the EJB bean when it is needed by a client. When the context ends, the container automatically destroys the bean.

For any EJB bean, even EJB beans which are not obtained via the Web Beans dependency injection service, the container provides certain services, including injection of Web Bean instances and binding of Web Bean interceptors and decorators.

1.3.2. Relationship to JSF

JavaServer Faces is a web-tier presentation framework that provides a component model for graphical user interface components, a *managed bean* component model for application logic, and an event-driven interaction model that binds the two component models. The managed bean component model is a contextual model where managed beans are bound to one of the three web tier contexts and may hold contextual state.

Any Web Bean may fulfill the role of the managed bean in a JSF application. Thus, a JSF application may take advantage of the more sophisticated context and dependency injection model defined by this specification. Even better, the Web Bean may be an EJB bean, allowing direct use of EJB in any JSF page.

1.3.3. Relationship to Java Servlets

Web Beans may be called by a Servlet or JSP.

Servlets are not themselves Web Beans, because they may not be injected into another object by the Web Beans dependency injection mechanism. However, in the Java EE 6 environment, the container does provide injection of Web Beans into Servlets and binding of Web Bean interceptors and decorators to Servlets.

1.3.4. Relationship to Common Annotations for the Java Platform

Certain functionality defined by Common Annotations for the Java Platform is available to any Web Bean.

1.4. Introductory examples

The following examples demonstrate the Web Beans programming model.

1.4.1. JSF example

The following JSF page defines a login prompt for a web application:

```
<f:view>
    <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>
</f:view>
```

The Unified EL expressions in this page refer to Web Beans named `credentials` and `login`.

The `Credentials` class is a Web Bean whose lifecycle is bound to the JSF request:

```
@Model
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; }

}
```

The `@Model` annotation is a *stereotype* that identifies the `Credentials` class as a Web Bean which acts as a model object in an MVC architecture.

The `Login` class is a Web Bean whose lifecycle is bound to the HTTP session:

```
@SessionScoped @Model
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() {
        if (user==null) {
            throw new NotLoggedInException();
        }
        else {
            return user;
        }
    }

}
```

The `@SessionScoped` annotation is a *scope type* that specifies the lifecycle of instances of `Login`.

The `@Current` annotation is a *binding annotation* and causes the `Credentials` Web Bean to be injected into an instance of `Login` when it is created by the container.

The Common Annotations `@PersistenceContext` annotation causes a JPA `EntityManager` to be injected by the container.

The `@LoggedIn` annotation is also a binding annotation. The method annotated `@Produces` is a *producer method*, which will be called whenever another Web Bean in the system needs the currently logged-in user, for example, whenever the `user` attribute of the `DocumentEditor` class is injected by the container:

```
@Model
public class DocumentEditor {

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

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

}
```

When the login form is submitted, JSF sets the entered username and password onto an instance of the `Credentials` Web Bean that is automatically instantiated and provided by the container. Next, JSF calls the `login()` method on an instance of `Login` that is automatically instantiated and provided by the container. This instance continues to exist for and be available to other requests in the same HTTP session, and provides the `User` object representing the current user to any other Web Bean that requires it (for example, `DocumentEditor`). If the producer method is called before the `login()` method initializes the user object, it throws a `NotLoggedInException`.

1.4.2. EJB example

Our `Login` class may take advantage of the functionality defined by EJB:

```
@Stateful @SessionScoped @Model
public class Login {

    @Current Credentials credentials;
    @PersistenceContext EntityManager userDatabase;

    private User user;

    @TransactionAttribute(REQUIRES_NEW)
    @RolesAllowed("guest")
    public void login() {
        ...
    }

    public void logout() {
        user = null;
    }

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

    @RolesAllowed("user")
    @Produces @LoggedIn User getCurrentUser() {
        ...
    }

}
```

The `@Stateful` annotation specifies that this Web Bean is also an EJB stateful session bean. The `@TransactionAttribute` and `@RolesAllowed` annotations declare the EJB transaction demarcation and security attributes.

1.4.3. Interceptor example

Web Beans *interceptors* allow common, cross-cutting concerns to be applied to Web Beans via custom annotations. Interceptor types may be individually enabled or disabled at deployment time.

The `AuthorizationInterceptor` class defines a custom authorization check:

```
@Secure @Interceptor public class AuthorizationInterceptor {

    @LoggedIn User user;

    @AroundInvoke public void authorize(InvocationContext ic) {
        try {
            if ( !user.isBanned() ) {
                System.out.println("Authorized");
                ic.proceed();
            }
            else {
                System.out.println("Not authorized");
                throw new NotAuthorizedException();
            }
        }
        catch (NotAuthenticatedException nae) {
            System.out.println("Not authenticated");
            throw nae;
        }
    }

}
```

The Web Beans `@Interceptor` annotation identifies the `AuthorizationInterceptor` class as a Web Beans interceptor. The `@Secure` annotation is a custom *interceptor binding type*.

```
@InterceptorBindingType
@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface Secure {}
```

The `@Secure` annotation is used to apply the interceptor to a Web Beans or EJB bean:

```
@Model
public class DocumentEditor {

    @Current Document document;
    @LoggedIn User user;
    @PersistenceContext EntityManager em;

    @Secure
    public void save() {
        document.setCreatedBy(currentUser);
        em.persist(document);
    }

}
```

When the `save()` method is invoked, the `authorize()` method of the interceptor will be called. The invocation will proceed to the `DocumentEditor` class only if the authorization check is successful.

1.4.4. Decorator example

Web Beans *decorators* are similar to interceptors, but apply only to Web Beans of a particular Java interface. Like interceptors, decorators may be easily enabled or disabled at deployment time. Unlike interceptors, decorators are aware of the semantics of the intercepted method.

For example, the `DataAccess` interface might be implemented by many Web Beans:

```
public interface DataAccess {

    public Object load(Object id);
    public Object getId();

    public void save();
    public void delete();

    public Class getDataType();

}
```

The `DataAccessAuthorizationDecorator` class defines the authorization checks:

```
@Decorator public abstract class DataAccessAuthorizationDecorator
    implements DataAccess {

    @Decorates DataAccess delegate;

    @LoggedIn User user;

    public void save() {
        authorize("save");
        delegate.save();
    }

    public void delete() {
        authorize("delete");
        delegate.delete();
    }

    private void authorize(String action) {
        try {
            Object id = delegate.getId();
            Class type = delegate.getDataType();
            if ( user.hasPermission(action, type, id) )
            {
                System.out.println("Authorized for " + action);
            }
            else {
                System.out.println("Not authorized for " + action);
                throw new NotAuthorizedException(action);
            }
        }
        catch (NotAuthenticatedException nae) {
            System.out.println("Not authenticated");
            throw nae;
        }
    }

}
```

The `@Decorator` annotation identifies the `DataAccessAuthorizationDecorator` class as a Web Beans decorator. The `@Decorates` annotation identifies the *delegate attribute*, which the decorator uses to delegate method calls to the container. The decorator applies to any Web Bean that implements `DataAccess`.

The decorator intercepts invocations just like an interceptor. However, unlike an interceptor, the decorator contains functionality that is specific to the semantics of the method being called.

Decorators may be declared abstract, relieving the developer of the responsibility of implementing all methods of the decorated interface. If a decorator does not implement a method of an API type, the decorator will simply not be called when that method is invoked upon the decorated Web Bean.

Chapter 2. Web Bean definition

A *Web Bean* is a Java EE component that bears additional metadata defining its lifecycle and interactions with other components according to the Web Beans context model.

Speaking more abstractly, a Web Bean is a source of contextual objects which define application state and/or logic. These objects are called *instances of the Web Bean*. The container creates and destroys these instances and associates them with the appropriate Web Beans context. Instances of a Web Bean may be injected into other objects (including other Web Bean instances) that execute in the same context, and may be used in EL expressions that are evaluated in the same context.

A Web Bean comprises the following attributes:

- A (nonempty) set of API types
- A (nonempty) set of binding annotation types
- A scope
- A deployment type
- Optionally, a Web Bean name
- A set of interceptor binding types
- A Web Bean implementation

In most cases, a Web Bean developer provides the Web Bean implementation by writing business logic in Java code. The developer then defines the remaining attributes by providing additional Web Beans specific metadata, or by allowing them to be defaulted by the container. In certain other cases, for example JMS endpoints defined in Section 3.6, “JMS endpoints”, the developer provides only the Web Beans specific metadata and the Web Bean implementation is provided by the container.

It is sometimes convenient to use XML instead of annotations to define this metadata. The `web-beans.xml` file format defined in Chapter 10, *XML based metadata* supports XML declaration of Web Beans.

A Web Bean implementation may be a Java class, an EJB session bean class, a producer method or field or a JMS queue or topic, as specified in Chapter 3, *Web Bean implementation*. The other attributes of the Web Bean are either:

- declared explicitly by annotating the implementation class,
- declared explicitly in `web-beans.xml`, or
- defaulted by the container.

The deployment type, API types and binding types of a Web Bean determine where its instances will be injected by the container.

The Web Bean developer may also create Web Beans interceptors and/or decorators or reuse existing interceptors and/or decorators. The interceptor binding types of a Web Bean determine which interceptors will be applied at runtime. The API types and binding types of a Web Bean determine which decorators will be applied at runtime. Interceptors, decorators and interceptor binding types are specified in Chapter 7, *Interceptors and decorators*.

A Web Bean implementation may produce or consume events. The Web Beans event notification facility is specified in Chapter 8, *Events*.

2.1. Functionality provided by the container to the Web Bean

A Web Bean is provided by the container with the following capabilities:

- transparent creation and destruction and scoping to a particular Web Beans context, specified in Chapter 6, *Web Bean lifecycle* and Chapter 9, *Scopes and contexts*,

- scoped resolution by API type and binding annotation type when injected into a Java-based client, as defined by Section 5.10, “Instance resolution”,
- scoped resolution by name when used in a Unified EL expression, as defined by Section 5.12, “EL name resolution”,
- lifecycle callbacks and automatic injection of other Web Bean instances, specified in Chapter 3, *Web Bean implementation*,
- method interception, callback interception, and decoration, as defined in Chapter 7, *Interceptors and decorators*, and
- event notification, as defined in Chapter 8, *Events*.

2.2. Web Bean API types

A Web Bean API type defines a client-visible type of the Web Bean. A Web Bean may have multiple API types. For example, the following Web Bean has three API types:

```
public class BookShop
        extends Business
        implements Shop<Book> {
    ...
}
```

The API types are `BookShop`, `Business` and `Shop<Book>`.

Meanwhile, this EJB has only the local interfaces `BookShop` and `Auditable` as API types, since the bean class is not a client-visible type.

```
@Stateful
public class BookShopBean
        extends Business
        implements BookShop, Auditable {
    ...
}
```

The rules for determining the set of API types for a Web Bean are defined in Chapter 3, *Web Bean implementation*.

The API types of a Web Bean are used by the resolution algorithms defined in Chapter 5, *Lookup, dependency injection and EL resolution*.

An API type may be a parameterized type with an actual type parameter. For the purposes of the typesafe resolution algorithm defined in Section 5.10.1, “Typesafe resolution algorithm”, parameterized API types are considered identical by the container only if both the type and the type parameters (if any) are identical.

However, API types may not declare a type variable or wildcard. If the type of an injection point is a parameterized type with a type variable or wildcard, a `DefinitionException` is thrown by the container at deployment time.

Aside from this restriction, almost any Java type may be an API type of a Web Bean:

- An API type may be an interface, a concrete class or an abstract class, and may be declared final or have final methods.
- An API type may be an array type. Two array types are considered identical only if the element type is identical.
- An API type may be a primitive types. Primitive types are considered to be identical to their corresponding wrapper types in `java.lang`.

However, certain additional restrictions are specified in Section 5.4.1, “Unproxyable API types” for Web Beans with a normal scope type, as defined in Section 9.3, “Normal scopes and pseudo-scopes”.

All Web Beans have the API type `java.lang.Object`.

A client of a Web Bean may typecast its reference to any instance of the Web Bean to any API type of the Web Bean. For example, if our simple Web Bean was injected to the following field:

```
@Current Shop<Book> bookShop;
```

Then the following typecast is legal and will not result in an exception:

```
Business biz = (Business) bookShop;
```

Likewise, if our EJB was injected to the following field:

```
@Current BookShop bookShop;
```

Then the following typecast is legal and will not result in an exception:

```
Auditable aud = (Auditable) bookShop;
```

2.3. Binding types

For a given API type, there may be multiple Web Beans which implement the type. For example, an application may have two implementations of the interface `PaymentProcessor`:

```
class SynchronousPaymentProcessor
        implements PaymentProcessor {
    ...
}
```

```
class AsynchronousPaymentProcessor
        implements PaymentProcessor {
    ...
}
```

A client that needs a `PaymentProcessor` that processes payments synchronously needs some way to distinguish between the two different implementations. One approach would be for the client to explicitly specify the class that implements that `PaymentProcessor` interface. However, this approach creates a hard dependence between client and implementation—exactly what use of the interface was designed to avoid!

A Web Beans *binding type* represents some client-visible semantic of an API implementation that is satisfied by some implementations of the API (and not by others). For example, we could introduce binding types representing synchronicity and asynchronicity. In Java code, binding types are represented by annotations.

```
@Synchronous
class SynchronousPaymentProcessor
        implements PaymentProcessor {
    ...
}
```

```
@Asynchronous
class AsynchronousPaymentProcessor
        implements PaymentProcessor {
    ...
}
```

Finally, binding types are applied to injection points to distinguish which implementation is required by the client. For example, when the container encounters the following injected field, an instance of `SynchronousPaymentProcessor` will be injected:

```
@Synchronous PaymentProcessor paymentProcessor;
```

But in this case, an instance of `AsynchronousPaymentProcessor` will be injected:

```
@Asynchronous PaymentProcessor paymentProcessor;
```

The container inspects the binding annotations and type of the injected attribute to determine the Web Bean instance to be injected, according to the resolution algorithm defined in Chapter 5, *Lookup, dependency injection and EL resolution*.

Binding types are also used as event selectors by observers of Web Beans events, as defined in Chapter 8, *Events*, and to bind decorators to Web Beans, as specified in Section 7.3, “Decorators”.

2.3.1. Default binding type

If a Web Bean does not explicitly declare a binding type, the Web Bean has exactly one binding type: `javax.webbeans.Current`. This is called the *default binding type*.

The following declarations are equivalent:

```
@Current
public class Order {}
```

```
public class Order {}
```

The default binding type is also assumed for any injection point that does not explicitly declare a binding type. The following declarations are equivalent:

```
public class Order {
    public Order(@Current OrderProcessor processor) { ... }
}
```

```
public class Order {
    public Order(OrderProcessor processor) { ... }
}
```

2.3.2. Defining new binding types

A binding type is a Java annotation defined as `@Target({METHOD, FIELD, PARAMETER, TYPE})` and `@Retention(RUNTIME)`.

A binding type may be declared by specifying the `@BindingType` meta-annotation.

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

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

Alternatively, the `@BindingType` meta-annotation may be omitted, and the binding type may be declared in `web-beans.xml`.

```
<myapp:Synchronous>
    <BindingType/>
</myapp:Synchronous>
```

A binding annotation may define annotation members.

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

Binding annotation member values are significant to the typesafe resolution algorithm.

2.3.3. Declaring the binding types of a Web Bean using annotations

A Web Bean's binding types are declared by annotating the implementation class or producer method or field with the binding types.

```
@LDAP
class LdapAuthenticator
        implements Authenticator {
    ...
}
```

```
public class Shop {

   @Produces @All
   public List<Product> getAllProducts() { ... }

   @Produces @WishList
   public List<Product> getWishList() { ..... }

   @Produces @ShoppingCart
   public List<Product> getShoppingCart() { ..... }

}
```

Any Web Bean may declare multiple binding types.

```
@Synchronous @Reliable
class SynchronousReliablePaymentProcessor
        implements PaymentProcessor {
    ...
}
```

2.3.4. Declaring the binding types of a Web Bean using XML

If a Web Bean is declared in `web-beans.xml`, binding types may be specified using the binding type names:

```
<myapp:SynchronousPaymentProcessor>
    <myapp:Synchronous/>
    <myapp:Reliable/>
</myapp:SynchronousPaymentProcessor>
```

2.3.5. Using binding annotations on injected fields

Binding annotations are applied to injected fields (see Section 3.7, “Injected fields”) to determine the Web Bean that is injected, according to the typesafe resolution algorithm defined in Section 5.10.1, “Typesafe resolution algorithm”.

```
@LDAP Authenticator authenticator;
```

A Web Bean may only be injected to an injection point if it has all the binding types of the injection point.

```
@Synchronous @Reliable PaymentProcessor paymentProcessor;
```

```
@All List<Product> catalog;
```

```
@WishList List<Product> wishList;
```

```
@ShoppingCart List<Product> cart;
```

For a Web Bean defined in XML, the binding types of a field may be specified using XML:

```
<myapp:paymentProcessor>
    <myapp:PaymentProcessor>
        <myapp:Asynchronous/>
        <myapp:Reliable/>
    </myapp:PaymentProcessor>
</myapp:paymentProcessor>
```

When the binding types of a field are specified using XML, any binding type annotations of the field are ignored.

2.3.6. Using binding annotations on method or constructor parameters

Binding annotations may be applied to parameters of producer methods, initializer methods, disposal methods or Web Bean constructors (see Chapter 3, *Web Bean implementation*) to determine the Web Bean instance that is passed when the method is called by the container. The container uses the typesafe resolution algorithm defined in Section 5.10.1, “Typesafe resolution algorithm” to determine values for these parameters.

For example, when the container encounters the following producer method, an instance of `SynchronousPaymentPro-`

cessor will be passed to the first parameter and an instance of AsynchronousPaymentProcessor will be passed to the second parameter:

```
@Produces
PaymentProcessor getPaymentProcessor(@Synchronous PaymentProcessor sync,
                                     @Asynchronous PaymentProcessor async) {
    return isSynchronous() ? sync : async;
}
```

For a Web Bean defined in XML, the binding types of a method parameter may be specified using XML:

```
<myapp:getPaymentProcessor>
    <Produces/>
    <myapp:PaymentProcessor>
        <myapp:Synchronous/>
    </myapp:PaymentProcessor>
    <myapp:PaymentProcessor>
        <myapp:Asynchronous/>
    </myapp:PaymentProcessor>
</myapp:getPaymentProcessor>
```

When the binding types of a parameter are specified using XML, any binding type annotations of the parameter are ignored.

2.4. Web Bean scopes

Unlike JSF managed beans, Java EE components such as Servlets, EJBs and JavaBeans do not have a well-defined *scope*. These components are either:

- *singletons*, such as EJB singleton beans, whose state is shared between all clients,
- *stateless objects*, such as Servlets and stateless session beans, which do not contain client-visible state, or
- objects that must be explictly created and destroyed by their client, such as JavaBeans and stateful session beans, whose state is shared by explicit reference passing between clients.

Scoped objects, by contrast, exist in a well-defined context:

- they may be automatically created when needed and then automatically destroyed when the context in which they were created ends, and
- their state is automatically shared by clients that execute in the same context.

All Web Beans have a scope. The scope of a Web Bean determines the lifecycle of its instances, and which instances of the Web Bean are visible to instances of other Web Beans, as defined in Chapter 9, *Scopes and contexts*. A scope type is represented by an annotation type.

For example, an object that represents the current user is represented by a session scoped object:

```
@Produces @SessionScoped User getCurrentUser() { ... }
```

An object that represents an order is represented by a conversation scoped object:

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

A list that contains the results of a search screen might be represented by a request scoped object:

```
@Produces @RequestScoped @Named("orders")
List<Order> getOrderSearchResults() { ... }
```

The set of scope types is extensible.

2.4.1. Built-in scope types

There are several standard scope types defined by Web Beans. The `@RequestScoped`, `@ApplicationScoped` and `@SessionScoped` annotations defined in Section 9.6, “Context management for built-in scopes” represent the standard scopes defined by the Java Servlets specification. The `@ConversationScoped` annotation represents the Web Beans conversation scope defined in Section 9.6.4, “Conversation context lifecycle”. In addition, there is the `@Dependent` pseudo-scope for dependent objects, as defined in Section 9.4, “Dependent pseudo-scope”.

2.4.2. Defining new scope types

A Web Beans scope type is a Java annotation defined as `@Target({TYPE, METHOD, FIELD})` and `@Retention(RUNTIME)`. All scope types must also specify the `@ScopeType` meta-annotation.

For example, the following annotation declares a "business process scope":

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

An application or third-party framework might provide a *context* implementation for this custom scope (see Section 9.7, “Context management for custom scopes”).

2.4.3. Declaring the Web Bean scope using annotations

The Web Bean's scope is defined by annotating the implementation class or producer method or field with a scope type.

A Web Bean implementation class or producer method or field may specify at most one scope type annotation. If an implementation class or producer method or field specifies multiple scope type annotations, a `DefinitionException` is thrown by the container at deployment time.

The following examples demonstrate the use of built-in scope types:

```
@RequestScoped
public class ProductList implements DataModel { ... }
```

```
public class Shop {

   @Produces @SessionScoped @WishList
   public List<Product> getWishList() { ..... }

   @Produces @ConversationScoped @ShoppingCart
   public List<Product> getShoppingCart() { ..... }

}
```

Likewise, a Web Bean with the custom business process scope may be declared by annotating it with the `@BusinessProcessScoped` annotation:

```
@BusinessProcessScoped
public class Order {
    ...
}
```

Alternatively, a scope type may be specified using a stereotype annotation, as defined in Section 2.7.2, “Declaring the stereotypes for a Web Bean using annotations”.

2.4.4. Declaring the Web Bean scope using XML

If the Web Bean is declared in `web-beans.xml`, the scope may be specified using the scope annotation type name:

```
<myapp:ProductList>
    <RequestScoped/>
</myapp:ProductList>
```

If more than one scope type is specified in XML, a `DefinitionException` is thrown by the container at deployment time.

Alternatively, a scope type may be specified using a stereotype declared in XML, as defined in Section 2.7.3, “Declaring

the stereotypes for a Web Bean using XML".

2.4.5. Default scope

When no scope is explicitly declared by annotating the implementation class or producer method or field, or by using XML, the scope of a Web Bean is defaulted.

The *default scope* for a Web Bean which does not explicitly declare a scope depends upon its declared stereotypes:

- If the Web Bean does not declare any stereotype with a declared default scope, the default scope for the Web Bean is `@Dependent`.
- If all stereotypes declared by the Web Bean that have some declared default scope have the same default scope, then that scope is the default scope for the Web Bean.
- If there are two different stereotypes declared by the Web Bean that declare different default scopes, then there is no default scope and the Web Bean must explicitly declare a scope. If it does not explicitly declare a scope, a `DefinitionException` is thrown by the container at deployment time.

If a Web Bean explicitly declares a scope, any default scopes declared by stereotypes are ignored.

2.5. Deployment types

In many applications, there are various implementations of a particular API, and the implementation used at runtime varies between different deployments of the system. Web Beans allows the developer to associate a particular implementation of an API with a certain deployment scenario.

A Web Beans *deployment type* represents a deployment scenario. Web Beans may be classified by deployment type, and thereby associated with various deployment scenarios.

Deployment types allow the container to identify which Web Beans should be *enabled* for use in a particular deployment of the system. The deployment type also determines the *precedence* of a Web Bean, used by the resolution algorithms specified in Chapter 5, *Lookup, dependency injection and EL resolution*.

The set of deployment types is extensible.

2.5.1. Built-in deployment types

There are two standard deployment types defined by Web Beans: `@Production` and `@Standard`.

All standard Web Beans defined by this specification, and provided by the container, are defined using the `@Standard` deployment type. For example, the `Conversation` object defined in Section 9.6.4, "Conversation context lifecycle" and the `Manager` object defined in Section 5.8, "The Manager object" have this deployment type. No Web Bean may be declared with the `@Standard` deployment type unless explicitly required by this specification.

Application Web Beans may be defined using the `@Production` deployment type.

2.5.2. Defining new deployment types

A Web Beans deployment type is a Java annotation defined as `@Target({TYPE, METHOD, FIELD})` and `@Retention(RUNTIME)`. All deployment types must also specify the `@DeploymentType` meta-annotation.

Applications and third-party frameworks may define their own deployment types. For example, the following deployment type might identify Web Beans which are used only at a particular site at which the application is deployed:

```
@DeploymentType
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface Australian {}
```

This deployment type might be used by a third-party framework that extends Web Beans:

```
@DeploymentType
```

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

This deployment type might be used to define mock objects for integration testing:

```
@DeploymentType
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface Mock {}
```

2.5.3. Declaring the deployment type of a Web Bean using annotations

The deployment type of the Web Bean is declared by annotating the implementation class or producer method or field.

An implementation class or producer method or field may specify at most one deployment type. If multiple deployment type annotations are specified, a `DefinitionException` is thrown by the container at deployment time.

Open issue: is this too restrictive? We could allow multiple deployment types to be specified, and ignore all but the highest-precedence enabled deployment type.

This Web Bean has the deployment type `@Production`:

```
@Production
public class Order {}
```

This Web Bean has the deployment type `@Mock`:

```
@Mock
public class MockOrder extends Order {}
```

By default, if no deployment type annotation is explicitly specified, a producer method or field inherits the deployment type of the Web Bean in which it is defined.

This producer method has the deployment type `@Production`:

```
@Production
public class Login {

   @Produces
   public User getUser() { ... }

}
```

This producer method has the deployment type `@Australian`:

```
@Production
public class TaxPolicies {

   @Produces @Australian
   public TaxPolicy getAustralianTaxPolicy() { ... }

}
```

Alternatively, a deployment type may be specified using a stereotype annotation, as defined in Section 2.7.2, “Declaring the stereotypes for a Web Bean using annotations”.

2.5.4. Declaring the deployment type of a Web Bean using XML

When a Web Bean is declared in `web-beans.xml`, the deployment type may be specified using a tag with the annotation type name:

```
<myapp:AustralianTaxPolicy>
    <deployment:Australian/>
</myapp:AustralianTaxPolicy>
```

If more than one deployment type is specified in XML, a `DefinitionException` is thrown by the container at deployment

time.

Alternatively, a deployment type may be specified using a stereotype declared in XML, as defined in Section 2.7.3, “Declaring the stereotypes for a Web Bean using XML”.

2.5.5. Default deployment type

When no deployment type is explicitly declared by annotating the implementation class or producer method or field, or by use of XML, the deployment type is defaulted.

The *default deployment type* for a Web Bean which does not explicitly declare a deployment type depends upon its declared stereotypes:

- If a Web Bean does not declare any stereotype with a declared default deployment type, then the default deployment type is `@Production`.
- Otherwise, the default deployment type for the Web Bean is the highest-precedence default deployment type declared by any stereotype declared by the Web Bean.

Thus, the following declarations are equivalent:

```
@Production
public class Order {}
```

```
public class Order {}
```

If a Web Bean explicitly declares a deployment type, any default deployment type declared by stereotypes are ignored.

2.5.6. Enabled deployment types

In a particular deployment, only some deployment types are *enabled*. Web Beans declared with a deployment type that is not enabled are not available to the resolution algorithms defined in Chapter 5, *Lookup, dependency injection and EL resolution*.

The container inspects the deployment type of each Web Bean that exists in a particular deployment (see Section 11.1, “Web Bean discovery”) to determine whether the Web Bean is *enabled* in this deployment. If the deployment type is enabled, an instance of the Web Bean may be obtained by lookup, injection or EL resolution. Otherwise, the Web Bean is never instantiated by the container.

By default, only the built-in deployment types are enabled. To enable a custom deployment type, a `<Deploy>` element must be included in a `web-beans.xml` file and the deployment type must be declared using the annotation type name.

```
<WebBeans>
    <Deploy>
        <Standard/>
        <Production/>
        <myfwk:DaoFramework/>
        <deployment:Australian/>
        <myfwk:Mock/>
    </Deploy>
</WebBeans>
```

If a `<Deploy>` element is specified, only the explicitly declared deployment types are enabled. The `@Standard` deployment type must be declared. If the `@Standard` deployment type is not declared, a `DeploymentException` is thrown by the container at deployment time.

If no `<Deploy>` element is specified in any `web-beans.xml` file, only the `@Standard` and `@Production` deployment types are enabled.

If the `<Deploy>` element is specified in more than one `web-beans.xml` document, a `DeploymentException` is thrown by the container at deployment time.

2.5.7. Deployment type precedence

In a particular deployment, all enabled deployment types are strongly ordered in terms of *precedence*. The precedence of a

deployment type is used by the resolution algorithms defined in Chapter 5, *Lookup, dependency injection and EL resolution*.

If a `<Deploy>` element is specified, the order of the deployment type declarations determines the deployment type precedence. Deployment types which appear later in this list have a higher precedence than deployment types which appear earlier. The `@Standard` deployment type must appear first and always has the lowest precedence of any deployment type.

If no `<Deploy>` element is specified, the `@Production` deployment type has a higher precedence than the `@Standard` deployment type.

2.6. Web Bean names

A Web Bean may have a *Web Bean name*. A Web Bean with a name may be referred to by its Web Bean name in Unified EL expressions. A valid Web Bean name is a period-separated list of valid EL identifiers.

There is no relationship between the Web Bean name of an EJB bean and the EJB name of the bean.

In certain circumstances, multiple Web Beans may share the same name.

Names are used by the EL name resolution algorithm defined in Section 5.10.1, “Typesafe resolution algorithm”. This allows a Web Bean to be used directly in a JSP or JSF page.

For example, a Web Bean with the name `products` could be used like this:

```
<h:outputText value="#{products.total}"/>
```

JMS endpoints do not have names.

2.6.1. Declaring the Web Bean name using annotations

To specify the name of a Web Bean, the `@Named` annotation is applied to the implementation class or producer method or field. This Web Bean is named `products`:

```
@Named("products")
public class ProductList implements DataModel { ... }
```

If the `@Named` annotation does not specify the `value` member, the default name is assumed.

2.6.2. Declaring the Web Bean name using XML

If the Web Bean is declared in `web-beans.xml`, the name may be specified using `<Named>`:

```
<myapp:ProductList>
    <Named>products</Named>
</myapp:ProductList>
```

If the `<Named>` element is empty, the default name is assumed.

2.6.3. Default Web Bean names

In the following circumstances, a *default name* must be assigned by the container:

- An implementation class or producer method or field of a Web Bean defined using annotations declares a `@Named` annotation and no name is explicitly specified by the `value` member.
- An empty `<Named>` element is specified by a Web Bean defined in XML.
- A Web Bean declares a stereotype that declares an empty `@Named` annotation, and the Web Bean does not explicitly specify a name.

The default name for a Web Bean depends upon the Web Bean implementation. The rules for determining the default name for a Web Bean are defined in Chapter 3, *Web Bean implementation*.

2.6.4. Web Beans with no name

If neither `<Named>` nor `@Named` is specified, by the Web Bean or its stereotypes, a Web Bean has no name.

2.7. Stereotypes

In many systems, use of architectural patterns produces a set of recurring Web Bean roles. A *stereotype* allows a framework developer to identify such a role and declare some common metadata for Web Beans with that role in a central place.

A stereotype encapsulates any combination of:

- a default deployment type,
- a default scope type,
- a restriction upon the Web Bean scope,
- a requirement that the Web Bean implement or extend a certain type, and
- a set of interceptor binding annotations.

A stereotype may also specify that all Web Beans with the stereotype have defaulted Web Bean names.

A Web Bean may declare zero, one or multiple stereotypes.

2.7.1. Defining new stereotypes

A Web Beans stereotype is a Java annotation defined as `@Target({TYPE, METHOD, FIELD})`, `@Target(TYPE)`, `@Target(METHOD)`, `@Target(FIELD)` or `@Target({METHOD, FIELD})` and `@Retention(RUNTIME)`.

A stereotype may be declared by specifying the `@Stereotype` meta-annotation.

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

Alternatively, the `@Stereotype` meta-annotation may be omitted, and the stereotype may be declared in `web-beans.xml`.

```
<myfwk:Action>
    <Stereotype/>
</myfwk:Action>
```

A stereotype may not declare any binding type annotation. If a stereotype declares a binding type annotation, a `DefinitionException` is thrown by the container at deployment time.

2.7.1.1. Declaring the default scope and deployment type for a stereotype

A stereotype may declare at most one scope type. If a stereotype declares more than one scope type, a `DefinitionException` is thrown by the container at deployment time.

A stereotype may declare at most one deployment type. If a stereotype declares more than one deployment type, a `DefinitionException` is thrown by the container at deployment time.

For example, the following stereotype might be used to identify action classes in a web application:

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

```
<myfwk:Action>
    <RequestScoped/>
    <Production/>
```

```
    <Stereotype/>
</myfwk:Action>
```

Then actions would have scope `@RequestScoped` and deployment type `@Production` unless the scope or deployment type explicitly specified by the Web Bean.

2.7.1.2. Specifying interceptor bindings for a stereotype

A stereotype may declare zero, one or multiple interceptor binding types, as defined in Section 7.2.4.2, “Interceptor bindings for stereotypes”.

We may specify interceptor bindings that apply to all actions:

```
@RequestScoped
@Secure
@Transactional
@Production
@Stereotype
@Target(TYPE)
@Retention(RUNTIME)
public @interface Action {}
```

```
<myfwk:Action>
    <RequestScoped/>
    <myfwk:Secure/>
    <myfwk:Transactional/>
    <Production/>
    <Stereotype/>
</myfwk:Action>
```

2.7.1.3. Specifying name defaulting for a stereotype

A stereotype may declare an empty `@Named` annotation. If a stereotype declares a non-empty `@Named` annotation, a `DefinitionException` is thrown by the container at deployment time.

We may specify that every Web Bean with the stereotype has a defaulted name when a name is not explicitly specified by the Web Bean:

```
@RequestScoped
@Secure
@Transactional
@Named
@Production
@Stereotype
@Target(TYPE)
@Retention(RUNTIME)
public @interface Action {}
```

```
<myfwk:Action>
    <RequestScoped/>
    <Named/>
    <myfwk:Secure/>
    <myfwk:Transactional/>
    <Production/>
    <Stereotype/>
</myfwk:Action>
```

2.7.1.4. Restricting Web Bean scopes and types using a stereotype

If all actions are request scoped, we can make this restriction explicit:

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

We may even require that all actions extend some `ActionBase` class:

```
@RequestScoped
@Secure
@Transactional
@Production
@Stereotype(requiredTypes=ActionBase.class)
@Target(TYPE)
@Retention(RUNTIME)
public @interface Action {}
```

Scope and type restrictions may not be specified when a stereotype is declared in XML.

2.7.1.5. Stereotypes with additional stereotypes

A stereotype may declare other stereotypes.

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

```
<myfwk:AuditableAction>
    <Stereotype/>
    <myfwk:Auditable/>
    <myfwk:Action/>
</myfwk:AuditableAction>
```

Stereotype declarations are transitive—a stereotype declared by a second stereotype is inherited by all Web Beans and other stereotypes that declare the second stereotype.

Stereotypes declared `@Target(TYPE)` may not be applied to stereotypes declared `@Target({TYPE, METHOD, FIELD})`, `@Target(METHOD)`, `@Target(FIELD)` or `@Target({METHOD, FIELD})`.

2.7.2. Declaring the stereotypes for a Web Bean using annotations

Stereotype annotations may be applied to a Web Bean implementation class or producer method or field.

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

The default deployment type and default scope declared by the stereotype may be overridden by the Web Bean:

```
@Mock @ApplicationScoped @Action
public class MockLoginAction extends LoginAction { ... }
```

Multiple stereotypes may be applied to the same Web Bean:

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

2.7.3. Declaring the stereotypes for a Web Bean using XML

If the Web Bean is declared in `web-beans.xml`, stereotypes may be declared using the stereotype annotation type name:

```
<myapp:LoginAction>
    <myfwk:Action/>
</myapp:LoginAction>
```

2.7.4. Stereotype restrictions

A stereotype may place certain restrictions upon the Web Beans that declare the stereotype.

If a stereotype declares a `requiredType`, and the Web Bean API types do not include the type, a `DefinitionException` is thrown by the container at deployment time.

If a stereotype explicitly declares a set of scope types using `supportedScopes`, and the Web Bean scope is not in that set, a `DefinitionException` is thrown by the container at deployment time.

If a Web Bean declares multiple stereotypes, it must satisfy every restriction declared by every declared stereotype.

2.7.5. Built-in stereotypes

Web Beans provides a built-in stereotype. The `@Model` stereotype is intended for use with Web Beans that define the *model* layer of an MVC web application architecture such as JSF:

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

In addition, the special-purpose `@Interceptor` and `@Decorator` stereotypes are defined in Chapter 7, *Interceptors and decorators*.

Chapter 3. Web Bean implementation

A Web Bean implementation implements the API types of the Web Bean. The developer must follow certain rules when defining a Web Bean implementation. However, the rules depend upon what kind of Web Bean it is. The container provides built-in support for the following kinds of Web Bean:

- Simple Web Beans (Java classes)
- Enterprise Web Beans (EJB session beans)
- Producer methods and fields
- JMS endpoints (topics and queues)

An application or third-party framework may support other kinds of Web Beans by implementing the `Bean` interface and registering the implementation with the container, as defined in Section 11.2, “Web Bean registration”.

Open issue: should the functionality that is currently defined for JMS endpoints be generalized to support other Java EE resources such as Web Service endpoints, connectors and JDBC datasources?

3.1. Restriction upon Web Bean instantiation

Most Web Beans are implemented by an annotated Java class, possibly an EJB bean class, called the *implementation class* of the Web Bean. Implementation classes are defined in Section 3.2, “Simple Web Beans” and Section 3.3, “Enterprise Web Beans”.

This specification places very few restrictions upon the implementation class of a Web Bean. In particular, the class is a concrete class and is not required to implement any special interface or extend any special superclass. Therefore, Web Bean implementation classes are easy to instantiate and unit test.

However, if the application directly instantiates an implementation class of a Web Bean, instead of letting the container perform instantiation, the capabilities listed in Section 2.1, “Functionality provided by the container to the Web Bean” will not be available to that particular class instance. In a deployed application, it is the Web Beans implementation that is responsible for instantiating Web Beans and initializing their dependencies.

If the application requires full control over instantiation of a Web Bean, a *producer method* may be used. A producer method is just an annotated method of another Web Bean that is invoked by the container to instantiate the Web Bean. Producer methods are defined in Section 3.4, “Producer methods”. However, a similar restriction exists for producer methods: if the application calls the producer method directly, instead of letting the container call it, the capabilities listed in Section 2.1, “Functionality provided by the container to the Web Bean” will not be available to the returned instance.

3.2. Simple Web Beans

A *simple Web Bean* is a Web Bean that is implemented by a Java class. This class is called the *implementation class* of the simple Web Bean.

The implementation class of a simple Web Bean may not be a non-static inner class or a parameterized type.

The implementation class of a simple Web Bean may not be an abstract class, unless the simple Web Bean is a Web Beans decorator.

If the implementation class of a simple Web Bean is annotated with both the `@Interceptor` and `@Decorator` stereotypes, a `DefinitionException` is thrown by the container at deployment time.

Note that multiple simple Web Beans may share the same implementation class. This occurs when Web Beans are defined using XML. Only one simple Web Bean per implementation class may be defined using annotations.

If a simple Web Bean has a public field, it must have scope `@Dependent`. If a simple Web Bean with a public field declares any scope other than `@Dependent`, a `DefinitionException` is thrown by the container at deployment time.

3.2.1. Which Java classes are simple Web Beans?

A top-level Java class is a simple Web Bean if it meets the following conditions:

- It is not a parameterized type.
- It is not a non-static inner class.
- It is a concrete class, or is annotated `@Decorator`.
- It is not annotated with any of the following annotations:
  - the JPA `@Entity` annotation,
  - the EJB component-defining annotations.
- It does not implement any of the following interfaces:
  - `javax.servlet.Servlet`
  - `javax.servlet.Filter`
  - `javax.servlet.ServletContextListener`
  - `javax.servlet.http.HttpSessionListener`
  - `javax.servlet.ServletRequestListener`
  - `javax.ejb.EnterpriseBean`
- It does not extend `javax.faces.component.UIComponent`.
- It is not declared as an EJB bean class in `ejb-jar.xml`.
- It is not declared as a JPA entity in `orm.xml`.
- It has an appropriate constructor—either:
  - the class has a constructor with no parameters, or
  - the class declares a constructor annotated `@Initializer`.

All Java classes that meet these conditions are simple Web Beans and thus no special declaration is required to define a simple Web Bean. Additional simple Web Beans for the class may be defined using XML, by specifying the class in `web-beans.xml`.

3.2.2. API types of a simple Web Bean

The set of API types for a simple Web Bean contains the implementation class, every superclass and all interfaces it implements directly or indirectly.

Note the additional restrictions upon API types of Web Beans with normal scope types defined in Section 5.4.1, “Unproxyable API types”.

3.2.3. Declaring a simple Web Bean using annotations

A simple Web Bean with a constructor that takes no parameters does not require any special annotations. The following classes are Web Beans:

```
public class Shop { .. }
```

```
class PaymentProcessorImpl implements PaymentProcessor { ... }
```

An implementation class may also specify a scope type, name, deployment type, stereotypes and/or binding annotations:

```
@ConversationScoped @Current
public class ShoppingCart { ... }
```

A simple Web Bean may extend another simple Web Bean:

```
@Named("loginAction")
public class LoginAction { ... }
```

```
@Mock
@Named("loginAction")
public class MockLoginAction extends LoginAction { ... }
```

The second Web Bean is a "mock object" that overrides the implementation of `LoginAction` when running in an embedded EJB Lite based integration testing environment.

3.2.4. Declaring a simple Web Bean using XML

Simple Web Beans may be declared in `web-beans.xml` using the implementation class name.

```
<myapp:Order>
    <deployment:Staging/>
    <ConversationScoped/>
    ...
</myapp:Order>
```

A simple Web Bean may even be declared at any injection point declared in XML, as defined in Section 10.7, “Inline Web Bean declarations”, in which case no binding types are specified.

If the implementation class of a simple Web Bean defined in XML is a parameterized type or a non-static inner class, a `DefinitionException` is thrown by the container at deployment time.

If the implementation class of a simple Web Bean defined in XML is an abstract class, and the simple Web Bean is not a Web Beans decorator, a `DefinitionException` is thrown by the container at deployment time.

If the implementation class of a simple Web Bean defined in XML is annotated `@Interceptor`, then the Web Bean must be explicitly declared as an interceptor in XML, as defined in Section 7.2.5.2, “Declaring a Web Beans interceptor using XML”. If a simple Web Bean defined in XML has an implementation class annotated `@Interceptor` and is not declared as an interceptor in XML, a `DefinitionExceptionis` thrown by the container at deployment time.

If the implementation class of a simple Web Bean defined in XML is annotated `@Decorator`, then the Web Bean must be explicitly declared as a decorator in XML, as defined in Section 7.3.2, “Declaring a decorator using XML”. If a simple Web Bean defined in XML has an implementation class annotated `@Decorator` and is not declared as a decorator in XML, a `DefinitionException` is thrown by the container at deployment time.

3.2.5. Web Bean constructors

When the container instantiates a simple Web Bean, it calls the *Web Bean constructor*. The Web Bean constructor is a constructor of the implementation class.

The application may call Web Bean constructors directly. However, if the application directly instantiates the Web Bean, no parameters are passed to the constructor by the container; the returned object is not bound to any context; no dependencies are injected by the container; and the lifecycle of the new instance is not managed by the container.

3.2.5.1. Declaring a Web Bean constructor using annotations.

The Web Bean constructor may be identified by annotating the constructor `@Initializer`.

```
@SessionScoped
public class ShoppingCart {

   private User customer;

   @Initializer
   public ShoppingCart(User customer) {
       this.customer = customer;
   }
```

```
   public ShoppingCart(ShoppingCart original) {
       this.customer = original.customer;
   }

   ShoppingCart() {}

   ...

}
```

```
@ConversationScoped
public class Order {

   private Product product;
   private User customer;

   @Initializer
   public Order(@Selected Product product, User customer) {
       this.product = product;
       this.customer = customer;
   }

   public Order(Order original) {
       this.product = original.product;
       this.customer = original.customer;
   }

   Order() {}

   ...

}
```

If a simple Web Bean defined using annotations does not explicitly declare a constructor using `@Initializer`, the constructor that accepts no parameters is the Web Bean constructor.

If a simple Web Bean defined using annotations has more than one constructor annotated `@Initializer`, a `DefinitionException` is thrown by the container at deployment time.

If a Web Bean constructor has a parameter annotated `@Disposes`, or `@Observes`, a `DefinitionException` is thrown by the container at deployment time.

3.2.5.2. Declaring a Web Bean constructor using XML.

For a simple Web Bean defined using XML, the Web Bean constructor may be specified by listing the parameter types of the constructer, in order, as direct children of the element that declares the Web Bean.

```
<myapp:ShoppingCart>
    <ConversationScoped/>
    <myapp:User/>
</myapp:ShoppingCart>
```

```
<myapp:Order>
    <ConversationScoped/>
    <myapp:Product>
        <Selected/>
    </myapp:Product>
    <myapp:User/>
</myapp:Order>
```

If a simple Web Bean defined using XML does not explicitly declare constructor parameters in XML, the constructor that accepts no parameters is the Web Bean constructor.

If a simple Web Bean declared in XML does not have a constructor with the parameter types declared in XML, a `NonexistentConstructorException` is thrown by the container at deployment time.

When a Web Bean constructor is declared in XML, the container ignores binding annotations applied to Java constructor parameters.

Open issue: should it default to use the constructor annotated `@Initializer`?

3.2.5.3. Web Bean constructor parameters

If the Web Bean constructor has parameters, the container calls the method `Manager.getInstanceByType()` defined in Section 5.10, “Instance resolution” to determine a value for each parameter and calls the constructor with those parameter values.

3.2.6. Specializing a simple Web Bean

If an implementation class of a simple Web Bean X defined using annotations is annotated `@Specializes`, then the implementation class of X must directly extend the implementation class of another simple Web Bean Y defined using annotations. Then:

- X inherits all binding types of Y, and
- if Y has a name, X has the same name as Y.

We say that X *directly specializes* Y, and we can be certain that Y will never be instantiated or called by the container if X is enabled.

If the implementation class of X does not directly extend the implementation class of another simple Web Bean, a `DefinitionException` is thrown by the container at deployment time.

For example, `MockLoginAction` directly specializes `LoginAction`:

```
public class LoginAction { ... }
```

```
@Mock @Specializes
public class MockLoginAction extends LoginAction { ... }
```

If a simple Web Bean X defined in XML declares the `<Specializes>` element, then the implementation class of X must be the implementation class of another simple Web Bean Y defined using annotations. Then:

- X inherits all binding types of Y, and
- if Y has a name, X has the same name as Y.

We say that X *directly specializes* Y, and we can be certain that Y will never be instantiated or called by the container if X is enabled.

3.2.7. Default name for a simple Web Bean

The default name for a simple Web Bean is the unqualified class name of the Web Bean implementation class, after converting the first character to lower case.

For example, if the implementation class is named `ProductList`, the default Web Bean name is `productList`.

3.3. Enterprise Web Beans

An *enterprise Web Bean* is a Web Bean that is implemented by an EJB 3-style session bean. The bean class is called the *implementation class* of the enterprise Web Bean.

An EJB stateless session bean must belong to the `@Dependent` pseudo-scope. An EJB singleton bean must belong to either the `@ApplicationScoped` scope or to the `@Dependent` pseudo-scope. If an enterprise Web Bean specifies an illegal scope, a `DefinitionException` is thrown by the container at deployment time.

Note that multiple enterprise Web Beans may share the same implementation class. This occurs when Web Beans are defined using XML. Only one Web Bean per implementation class may be defined using annotations.

However, in any deployment, there may be at most one most specialized enabled enterprise Web Bean for any particular EJB enterprise bean. Therefore, for each distinct EJB name in a module, there is at most one Web Bean that may be called at runtime. If there is more than one most specialized enabled enterprise Web Bean for a particular EJB enterprise bean, a `DeploymentException` is thrown by the container at deployment time. This restriction exists because the container is not aware of the binding types of the client injection point when the container intercepts the lifecycle callbacks of the EJB

bean, as defined in Section 6.9, “Lifecycle of EJB beans”.

If the implementation class of an enterprise Web Bean is annotated `@Interceptor` or `@Decorator`, a `DefinitionException` is thrown by the container at deployment time.

3.3.1. EJB remove methods of enterprise Web Beans

If an enterprise Web Bean is a stateful session bean:

- If the scope is `@Dependent`, the application *may* call any EJB remove method of an instance of the enterprise Web Bean.
- Otherwise, the application *may not* directly call any EJB remove method of any instance of the enterprise Web Bean.

If the application directly calls an EJB remove method of an instance of an enterprise Web Bean that is a stateful session bean and declares any scope other than `@Dependent`, an `UnsupportedOperationException` is thrown.

If the application directly calls an EJB remove method of an instance of an enterprise Web Bean that is a stateful session bean and has scope `@Dependent` then no parameters are passed to the method by the container. Furthermore, the container ignores the instance instead of destroying it when `Bean.destroy()` is called, as defined in Section 6.4, “Lifecycle of stateful session enterprise Web beans”.

3.3.2. Which EJBs are enterprise Web Beans?

All session beans exposing an EJB 3.x client view and declared via an EJB component defining annotation on the EJB bean class are Web Beans, and thus no special declaration is required. Additional enterprise Web Beans for these EJBs may be defined using XML, by specifying the bean class in `web-beans.xml`.

All session beans exposing an EJB 3.x client view and declared in `ejb-jar.xml` are also Web Beans. Additional enterprise Web Beans for these EJBs may be defined using XML, by specifying the bean class and EJB name in `web-beans.xml`.

3.3.3. API types of an enterprise Web Bean

The set of API types for an enterprise Web Bean contains all local interfaces of the bean that do not have wildcard type parameters or type variables and their superinterfaces. If the EJB bean has a bean class local view and the bean class is not a parameterized type, the set of API types contains the bean class and all superclasses. In addition, `java.lang.Object` is an API type of every enterprise Web Bean.

Remote interfaces are not included in the set of API types.

3.3.4. Declaring an enterprise Web Bean using annotations

An enterprise Web Bean does not require any special annotations. The following EJBs are Web Beans:

```
@Singleton
class Shop { .. }
```

```
@Stateless
class PaymentProcessorImpl implements PaymentProcessor { ... }
```

An implementation class may also specify a scope type, name, deployment type, stereotypes and/or binding annotations:

```
@ConversationScoped @Stateful @Current @Model
public class ShoppingCart { ... }
```

An enterprise Web Bean implementation class may extend another Web Bean implementation class:

```
@Stateless
@Named("loginAction")
public class LoginActionImpl implements LoginAction { ... }
```

```
@Stateless
@Mock
@Named("loginAction")
```

```
public class MockLoginActionImpl extends LoginActionImpl { ... }
```

3.3.5. Declaring an enterprise Web Bean using XML

Enterprise Web Beans may be declared in `web-beans.xml` using the bean class name (for EJBs defined using a component-defining annotation) or bean class and EJB name (for EJBs defined in `ejb-jar.xml`).

```
<myapp:OrderBean>
    <deployment:Staging/>
    <ConversationScoped/>
    ...
</myapp:OrderBean>
```

```
<myapp:OrderBean ejbName="RushOrder">
    <myapp:Rush/>
    <ConversationScoped/>
    ...
</myapp:OrderBean>
```

The `ejbName` attribute declares the EJB name of an EJB defined in `ejb-jar.xml`.

Enterprise Web Beans may not be message-driven beans. If an enterprise Web Bean declared in XML is a message-driven bean, a `DefinitionException` is thrown by the container at deployment time.

3.3.6. Specializing an enterprise Web Bean

If an implementation class of an enterprise Web Bean X defined using annotations is annotated `@Specializes`, then the implementation class of X must directly extend the implementation class of another enterprise Web Bean Y defined using annotations. Then:

- X inherits all binding types of Y, and
- if Y has a name, X has the same name as Y.

Furthermore:

- X must support all local interfaces supported by Y, and
- if Y supports a bean-class local view, X must also support a bean-class local view.

Otherwise, a `DefinitionException` is thrown by the container at deployment time.

We say that X *directly specializes* Y, and we can be certain that Y will never be instantiated or called by the container if X is enabled.

If the implementation class of X does not directly extend the implementation class of another enterprise Web Bean, a `DefinitionException` is thrown by the container at deployment time.

For example, `MockLoginActionBean` directly specializes `LoginActionBean`:

```
@Stateless
public class LoginActionBean implements LoginAction { ... }
```

```
@Stateless @Mock @Specializes
public class MockLoginActionBean extends LoginActionBean { ... }
```

If an enterprise Web Bean X defined in XML declares the `<Specializes>` element, then the implementation class of X must be the implementation class of another enterprise Web Bean Y defined using annotations. Then:

- X inherits all binding types of Y, and
- if Y has a name, X has the same name as Y.

We say that X *directly specializes* Y, and we can be certain that Y will never be instantiated or called by the container if X

is enabled.

3.3.7. Default name for an enterprise Web Bean

The default name for an enterprise Web Bean is the unqualified class name of the Web Bean implementation class, after converting the first character to lower case.

For example, if the bean class is named `ProductList`, the default Web Bean name is `productList`.

3.3.8. Enterprise bean proxies

EJB local object references do not implement all local interfaces of the EJB. A local object reference may not be typecast to different local interface type, as required by Section 2.2, “Web Bean API types”. Therefore, the container proxies the local object reference. An enterprise bean proxy implements all local interfaces of the EJB.

When the proxy object is invoked, the proxy obtains the appropriate EJB local object reference and delegates the invocation to the local object reference.

When an enterprise Web Bean is invoked via the enterprise bean proxy, the interface returned by `SessionContext.getInvokedBusinessInterface()` will be specific to the container implementation. Portable applications should not rely upon the interface returned by this method.

3.4. Producer methods

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, or
- 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.

A producer method must be a method of a simple Web Bean implementation class or enterprise Web Bean implementation class. A producer method may be either static or non-static. If the Web Bean is an enterprise Web Bean, the producer method must be either a business method of the EJB or a static method of the bean class.

If a producer method sometimes returns a null value, then the producer method must have scope `@Dependent`. If a producer method returns a null value at runtime, and the producer method declares any other scope, an `IllegalProductException` is thrown by the container. This restriction allows the container to use a client proxy, as defined in Section 5.4, “Client proxies”.

If the producer method return type is a parameterized type, it must specify actual type parameters for each type parameter. If a producer method return type contains a wildcard type parameter or type variable, a `DefinitionException` is thrown by the container at deployment time.

The application may call producer methods directly. However, if the application calls a producer method directly, no parameters will be passed to the producer method by the Web Beans implementation; the returned object is not bound to any context; and its lifecycle is not managed by the container.

A Web Bean may declare multiple producer methods.

3.4.1. API types of a producer method

The API types of a producer method depend upon the method return type:

- If the return type is an interface, the set of API types contains the return type, all interfaces it extends directly or indirectly and `java.lang.Object`.
- If a return type is primitive or is a Java array type, the set of API types contains exactly two types: the method return type and `java.lang.Object`.
- If the return type is a class, the set of API types contains the return type, every superclass and all interfaces it imple-

ments directly or indirectly.

Note the additional restrictions upon API types of Web Beans with normal scope types defined in Section 5.4.1, “Unproxyable API types”.

3.4.2. Declaring a producer method using annotations

A producer method may be declared by annotating a method with the `@Produces` annotation.

```
public class Shop {

   @Produces PaymentProcessor getPaymentProcessor() { ... }

   @Produces List<Product> getProducts() { ... }

}
```

A producer method may also specify scope, name, deployment type, stereotypes and/or binding annotations.

```
public class Shop {

   @Produces @ApplicationScoped @Catalog @Named("catalog")
   List<Product> getProducts() { ... }

}
```

If a producer method is annotated `@Initializer` or `@Destructor`, has a parameter annotated `@Disposes`, or has a parameter annotated `@Observes`, a `DefinitionException` is thrown by the container at deployment time.

3.4.3. Declaring a producer method using XML

For a Web Beans defined in XML, a producer method may be declared using the method name, the `<Produces>` element, the return type, and the parameter types of the method:

```
<myapp:Shop>

    <myapp:getProducts>
        <Produces>
            <ApplicationScoped/>
            <util:List>
                <myapp:Product/>
                <myapp:Catalog/>
            </util:List>
            <Named>catalog</Named>
        </Produces>
    </myapp:getProducts>

</myapp:Shop>
```

When a producer method is declared in XML, the container ignores binding annotations applied to the Java method or method parameters.

If the implementation class of a Web Bean declared in XML does not have a method with the name and parameter types declared in XML, a `NonexistentMethodException` is thrown by the container at deployment time.

3.4.4. Producer method parameters

If the producer method has parameters, the container calls the method `Manager.getInstanceByType()` defined in Section 5.10, “Instance resolution” to determine a value for each parameter and calls the producer method with those parameter values.

```
public class OrderFactory {

   @Produces @ConversationScoped
   public Order createCurrentOrder(@New Order order, @Selected Product product)
   {
       order.setProduct(product);
       return order;
   }
```

```
}
```

```
<myapp:OrderFactory>

    <myapp:createCurrentOrder>

        <Produces>
            <ConversationScoped/>
            <myapp:Order/>
        </Produces>

        <myapp:Order>
            <New/>
        </myapp:Order>

        <myapp:Product>
            <myapp:Selected/>
        </myapp:Product>

    </myapp:createCurrentOrder>

</myapp:OrderFactory>
```

3.4.5. Specializing a producer method

If a producer method X is annotated `@Specializes`, then it must be non-static and directly override another producer method Y. Then:

- X inherits all binding type of Y, and
- if Y has a name, X has the same name as Y.

We say that X *directly specializes* Y, and we can be certain that Y will never be called by the container if X is enabled.

If the method is static or does not directly override another producer method, a `DefinitionException` is thrown by the container at deployment time.

For example:

```
@Mock
public class MockShop extends Shop {

   @Override @Specializes
   @Produces
   PaymentProcessor getPaymentProcessor() {
      return new MockPaymentProcessor();
   }

   @Override @Specializes
   @Produces
   List<Product> getProducts() {
      return PRODUCTS;
   }

   ...

}
```

3.4.6. Disposal methods

A disposal method allows the application to perform customized cleanup of an object returned by a producer method.

A disposal method must be a method of a simple Web Bean implementation class or enterprise Web Bean implementation class. A disposal method may be either static or non-static. If the Web Bean is an enterprise Web Bean, the disposal method must be a business method of the EJB or a static method of the bean class.

A Web Bean may declare multiple disposal methods.

3.4.7. Disposed parameter of a disposal method

Each disposal method must have exactly one *disposed parameter*, of the same type as the corresponding producer method return type. When searching for disposal methods for a producer method, the container considers the type and binding types of the disposed parameter. If a disposed parameter resolves to a producer method according to the typesafe resolution algorithm, the container must call this method when destroying an instance returned by that producer method.

If the disposed parameter does not resolve to any producer method according to the typesafe resolution algorithm, an `UnsatisfiedDependencyException` is thrown by the container at deployment time.

3.4.8. Declaring a disposal method using annotations

A disposal method may be declared using annotations by annotating a parameter `@Disposes`. That parameter is the disposed parameter.

```
public class UserDatabaseEntityManager {

    @Produces @ConversationScoped @UserDatabase
    public EntityManager create(EntityManagerFactory emf) {
        return emf.createEntityManager();
    }

    public void close(@Disposes @UserDatabase EntityManager em) {
        em.close();
    }

}
```

If a method has more than one parameter annotated `@Disposes`, a `DefinitionException` is thrown by the container.

If a disposal method is annotated `@Produces`, `@Initializer` or `@Destructor`, or has a parameter annotated `@Observes`, a `DefinitionException` is thrown by the container at deployment time.

3.4.9. Declaring a disposal method using XML

For a Web Beans defined in XML, a disposal method may be declared using the method name, the `<Disposes>` element, and the parameter types of the method:

```
<myfwk:UserDatabaseEntityManager>

    <myfwk:create>
        <Produces>
            <ConversationScoped/>
            <jpa:EntityManager>
                <myapp:UserDatabase/>
            </jpa:EntityManager>
        </Produces>
        <jpa:EntityManagerFactory/>
    </myfwk:create>

    <myfwk:close>
        <Disposes>
            <jpa:EntityManager>
                <myapp:UserDatabase/>
            </jpa:EntityManager>
        </Disposes>
    </myfwk:close>

</mmyfwk:UserDatabaseEntityManager>
```

When a disposal method is declared in XML, the container ignores binding annotations applied to the Java method parameters.

If the implementation class of a Web Bean declared in XML does not have a method with the name and parameter types declared in XML, a `NonexistentMethodException` is thrown by the container at deployment time.

3.4.10. Disposal method parameters

In addition to the disposed parameter, a disposal method may declare additional parameters, which may also specify binding types. The container calls `Manager.getInstanceByType()` to determine a value for each parameter of a disposal method and calls the disposal method with those parameter values.

```
public void close(@Disposes @UserDatabase EntityManager em, @Logger Log log) { ... }
```

```
<myfwk:close>

    <Disposes>
        <jpa:EntityManager>
            <myapp:UserDatabase/>
        </jpa:EntityManager>
    </Disposes>

    <myfwk:Log>
        <myfwk:Logger/>
    <myfwk:Log>

</myfwk:close>
```

3.4.11. Disposal method resolution

When searching for disposal methods for a producer method, the container searches for disposal methods which satisfy the following rules:

- The disposal method must be declared by an enabled Web Bean.
- The disposed parameter must resolve to the producer method, according to the typesafe resolution algorithm.

If there are multiple disposal methods for a producer method, a `DefinitionException` is thrown by the container at deployment time.

3.4.12. Default name for a producer method

The default name for a producer method is the method name, unless the method follows the JavaBeans property getter naming convention, in which case the default name is the JavaBeans property name.

For example, this producer method is named `products`:

```
public class Shop {

   @Produces @Named
   public List<Product> getProducts() { ... }

}
```

This producer method is named `paymentProcessor`:

```
public class Shop {

   @Produces @Named
   public PaymentProcessor paymentProcessor() { ... }

}
```

3.5. Producer fields

A Web Beans producer field is a slightly simpler alternative to a producer method.

A producer field must be a field of a simple Web Bean implementation class or enterprise Web Bean implementation class. A producer field may be either static or non-static.

If a producer field sometimes contains a null value when accessed, then the producer field must have scope `@Dependent`. If a producer method contains a null value at runtime, and the producer field declares any other scope, an `IllegalProductException` is thrown by the container. This restriction allows the container to use a client proxy, as defined in Section 5.4, “Client proxies”.

If the producer field return type is a parameterized type, it must specify actual type parameters for each type parameter. If a producer field return type contains a wildcard type parameter or type variable, a `DefinitionException` is thrown by the container at deployment time.

The application may access producer fields directly. However, if the application accesses a producer field directly, the returned object is not bound to any context; and its lifecycle is not managed by the container.

A Web Bean may declare multiple producer fields.

3.5.1. API types of a producer field

The API types of a producer field depend upon the field type:

- If the field type is an interface, the set of API types contains the field type, all interfaces it extends directly or indirectly and `java.lang.Object`.
- If a field type is primitive or is a Java array type, the set of API types contains exactly two types: the field type and `java.lang.Object`.
- If the field type is a class, the set of API types contains the field type, every superclass and all interfaces it implements directly or indirectly.

Note the additional restrictions upon API types of Web Beans with normal scope types defined in Section 5.4.1, “Unproxyable API types”.

3.5.2. Declaring a producer field using annotations

A producer field may be declared by annotating a field with the `@Produces` annotation.

```
public class Shop {

   @Produces PaymentProcessor paymentProcessor = ....;

   @Produces List<Product> products = ....;

}
```

A producer field may also specify scope, name, deployment type, stereotypes and/or binding annotations.

```
public class Shop {

   @Produces @ApplicationScoped @Catalog @Named("catalog")
   List<Product> products = ....;

}
```

3.5.3. Declaring a producer field using XML

For a Web Beans defined in XML, a producer field may be declared using the field name, the `<Produces>` element, and the type:

```
<myapp:Shop>

    <myapp:products>
        <Produces>
            <ApplicationScoped/>
            <util:List>
                <myapp:Product/>
                <myapp:Catalog/>
            </util:List>
            <Named>catalog</Named>
        </Produces>
    </myapp:products>

</myapp:Shop>
```

When a producer field is declared in XML, the container ignores binding annotations applied to the Java field.

If the implementation class of a Web Bean declared in XML does not have a field with the name and type declared in XML, a `NonexistentMethodException` is thrown by the container at deployment time.

3.5.4. Default name for a producer field

The default name for a producer field is the field name.

For example, this producer field is named `products`:

```
public class Shop {

   @Produces @Named
   public List<Product> products = ...;

}
```

3.6. JMS endpoints

Web Beans that send JMS messages must interact with at least two different objects defined by the JMS API:

- to send a message to a queue, the Web Bean must interact with a `QueueSession` and the `QueueSender`, or
- to send a message to a topic, the Web Bean must interact with a `TopicSession` and the `TopicPublisher`.

A Web Beans *JMS endpoint* is a Web Bean that represents a JMS queue or topic. JMS endpoints may be declared in `web-beans.xml`, and allow direct injection of any of the following JMS objects:

- For a queue, the `Queue`, `QueueConnection`, `QueueSession` and/or `QueueSender` may be injected.
- For a topic, the `Topic`, `TopicConnection`, `TopicSession` and/or `TopicPublisher` may be injected.

The lifecycles of the injected objects are managed by the container, and therefore the application need not explicitly `close()` any injected JMS object. If the application calls `close()` on an instance of a JMS endpoint, an `UnsupportedOperationException` is thrown by the container.

For example:

```
@PaymentProcessor QueueSender paymentSender;
@PaymentProcessor QueueSession paymentSession;

public void sendMessage() {
    MapMessage msg = paymentSession.createMapMessage();
    ...
    paymentSender.send(msg);
}
```

```
@Prices TopicPublisher pricePublisher;
@Prices TopicSession priceSession;

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

A JMS endpoint must belong to the `@Dependent` pseudo-scope. If a JMS endpoint specifies any other scope, a `DefinitionException` is thrown by the container at deployment time.

Web Beans JMS endpoints must explicitly declare at least one binding type, and must not declare the `@Current` binding type.

A JMS endpoint may not declare a Web Beans name.

JMS endpoints are always declared using XML.

3.6.1. API types of a JMS endpoint

The API types of a JMS endpoint depend upon whether it represents a queue or topic.

- If the JMS endpoint represents a queue, the API types are `Queue`, `QueueConnection`, `QueueSession` and `QueueSender`.

- If the JMS endpoint represents a topic, the API types are `Topic`, `TopicConnection`, `TopicSession` and `TopicPublisher`.

In addition, `java.lang.Object` is an API type of every JMS endpoint.

3.6.2. Declaring a JMS endpoint using XML

A JMS endpoint may be declared using the `<Topic>` or `<Queue>` elements in `web-beans.xml`. The JNDI name of the queue or topic must be specified using `<destination>` and the JNDI name of the JMS connection factory must be specified using `<connectionFactory>`.

```
<Queue>
    <destination>java:comp/env/jms/PaymentQueue</destination>
    <connectionFactory>java:comp/env/jms/QueueConnectionFactory</connectionFactory>
    <myapp:PaymentProcessor/>
</Queue>
```

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

Open issue: do we need to allow specification of `transacted` and `acknowledgeMode` for the session?

3.7. Injected fields

An *injected field* is a non-static, non-final field of a Web Bean implementation class, of a Servlet, or of any EJB session or message driven bean class.

Injected fields are initialized by the container immediately after instantiation and before any methods of the instance are invoked. The container calls the method `Manager.getInstanceByType()` defined in Section 5.10, “Instance resolution” to determine a value for each injected field.

Any EJB session or message driven bean running in the context of a Web Beans application may declare injected fields and have those fields injected by the container. The EJB bean is not required to be the implementation class of a Web Bean to take advantage of this functionality.

Open issue: are injected fields allowed to be declared transient? If so, should they be reinjected after deserialization (activation)?

If a field is a producer field or a decorator delegate attribute, it is not an injected field.

3.7.1. Declaring an injected field using annotations

An injected field may be declared by annotating the field with any binding type.

```
@ConversationScoped
public class Order {

   @Selected Product product;
   @Current User customer;

}
```

3.7.2. Declaring an injected field using XML

For a Web Beans defined in XML, an injected field may be declared using the field name and a child element representing the type of the field:

```
<myapp:Order>
    <ConversationScoped/>

    <myapp:product>
        <myapp:Product>
            <myapp:Selected/>
```

```
        </myapp:Product/>
    </myapp:product>

    <myapp:customer>
        <myapp:User/>
    </myapp:customer>

</myapp:Order>
```

When an injected field is declared in XML, the container ignores binding annotations applied to the Java field.

If the type element does not declare any binding type, the default binding type `@Current` is assumed.

If the implementation class of a Web Bean declared in XML does not have a field with the name and type declared in XML, a `NonexistentFieldException` is thrown by the container at deployment time.

3.8. Initializer methods

An *initializer method* is a non-static method of a Web Bean implementation class, of a Servlet, or of any EJB session or message driven bean class.

Initializer methods are called by the container immediately after injected fields have been initialized by the container and before any other methods of the instance are invoked.

If the Web Bean is an enterprise Web Bean, the initializer method is *not* required to be a business method of the session bean.

Method interceptors are never called when the container calls an initializer method.

A Web Bean implementation class may declare multiple (or zero) initializer methods.

The application may call initializer methods directly, but then no parameters will be passed to the method by the container.

Any EJB session or message driven bean running in the context of a Web Beans application may declare initializer methods and have the methods called by the container. The EJB bean is not required to be the implementation class of a Web Bean to take advantage of this functionality.

3.8.1. Declaring an initializer method using annotations

An initializer method may be declared by annotating the method `@Initializer`.

```
@ConversationScoped
public class Order {

   private Product product;
   private User customer;

   @Initializer
   void setProduct(@Selected Product product)
   {
       this.product = product;
   }

   @Initializer
   public void setCustomer(User customer)
   {
       this.customer = customer;
   }

}
```

If an initializer method is annotated `@Produces` or `@Destructor`, has a parameter annotated `@Disposes`, or has a parameter annotated `@Observes`, a `DefinitionException` is thrown by the container at deployment time.

3.8.2. Declaring an initializer method using XML

For a Web Beans defined in XML, an initializer method may be declared using the method name, the `<Initializer>` element and the parameter types of the method.

```
<myapp:Order>
    <ConversationScoped/>

    <myapp:setProduct>
        <Initializer/>
        <myapp:Product>
            <myapp:Selected/>
        </myapp:Product>
    </myapp:setOrder>

    <myapp:setCustomer>
        <Initializer/>
        <myapp:User/>
    </myapp:setCustomer>

</myapp:Order>
```

When an initializer method is declared in XML, the container ignores binding annotations applied to the Java method parameters.

If the implementation class of a Web Bean declared in XML does not have a method with the name and parameter types declared in XML, a `NonexistentMethodException` is thrown by the container at deployment time.

3.8.3. Initializer method parameters

An initializer method may have any number of parameters. If the initializer method has parameters, the container calls `Manager.getInstanceByType()` to determine a value for each parameter and calls the initializer method with those parameter values.

3.9. The `@New` binding type

Sometimes, the scope of a Web Bean is inconvenient for a particular usecase. One solution to this problem is to obtain an independent instance of the Web Bean implementation and inject it as a dependent object of some other Web Bean, or even bind it to a different context using a producer method.

When the built-in binding type `@New` is applied to an injection point, a Web Bean is implicitly defined with:

- scope `@Dependent`,
- deployment type `@Standard`,
- `@New` as the only binding annotation,
- no Web Bean name,
- no stereotypes, and such that
- the implementation class is the declared type of the injection point.

If the parameter type satisfies the definition of a simple Web Bean implementation class, Section 3.2.1, “Which Java classes are simple Web Beans?”, then the Web Bean is a simple Web Bean. If the parameter type satisfies the definition of an enterprise Web Bean implementation class, Section 3.3.2, “Which EJBs are enterprise Web Beans?”, then the Web Bean is an enterprise Web Bean.

Furthermore, this Web Bean:

- has the same Web Bean constructor, initializer methods and injected fields as a Web Bean defined using annotations—that is, it has any Web Bean constructor, initializer method or injected field declared by annotations that appear on the implementation class,
- has no observer methods, producer methods or fields or disposal methods,
- has the same interceptors as a Web Bean defined using annotations—that is, it has all the interceptor binding types declared by annotations that appear on the implementation class, and
- has no decorators.

The `@New` annotation or `<New>` element may be applied to any field of a Web Bean implementation class or to any parameter of a producer method, initializer method, disposal method or Web Bean constructor where the type of the field or parameter is a concrete Java type which satisfies the requirements of a simple Web Bean implementation class or enterprise Web Bean implementation class.

```
@Produces @RequestScoped
Payment createPayment(@New Payment payment, Order order) {
    payment.setOrder(order);
    return payment;
}
```

```
<myapp:createPayment>

    <Produces>
        <RequestScoped/>
        <myapp:Payment/>
    </Produces>

    <myapp:Payment>
        <New/>
    </myapp:Payment>

    <myapp:Order/>

</myapp:createPayment>
```

In this example, the `Payment` is created as a dependent object of the producer method Web Bean and is bound to the request context when returned by the method. It is now the current instance of the producer method Web Bean. When the request context ends, the `Payment` is passed to the corresponding disposal method (if any), and then finally destroyed when all dependent objects of the producer method are destroyed.

The `@New` annotation and `<New>` element may not appear in conjunction with any other binding type. They may not be applied to a field or method parameter of a type which does not satisfy the definition of a simple Web Bean implementation class or enterprise Web Bean implementation class.

If the `@New` binding type appears in conjunction with some other binding type, or is specified for a field or parameter of a type which does not satisfy the definition of a simple Web Bean implementation class or enterprise Web Bean implementation class, a `DefinitionException` is thrown by the container at deployment time.

No Web Bean defined using annotations or XML may explicitly declare `@New` as a binding type.

3.10. Support for Common Annotations

In addition to the capabilities defined by this specification, simple Web Beans also support certain functionality defined by the Common Annotations for the Java Platform and Enterprise JavaBeans specifications.

The following functionality is provided by the container when annotations are applied to the implementation class of a simple Web Bean:

- dependency injection via `@EJB`, `@PersistenceContext` and `@Resource`
- JNDI lookup of resource references declared via `@Resource` and `@Resources`
- `@PostConstruct` and `@PreDestroy` callbacks
- interception, as defined in `javax.interceptor`

`@PersistenceContext(type=EXTENDED)` is not supported for simple Web Beans.

Open issue: should `@PrePassivate` and `@PostActivate` be supported for simple Web Beans?

Open issue: what restrictions exist upon invoking dependencies from `@PreDestroy`?

Support for `@EJB`, `@PersistenceContext`, `@Resource` and `@Resources` is not required when a plugin container is used in a Java EE 5 environment.

This simple Web Bean makes use of annotations defined by the Common Annotations and EJB specifications:

```
@SessionScoped
@Interceptors(MyTransactionInterceptor.class)
public class ShoppingCart {

    private User customer;
    private Order order;
    private @Resource Connection connection;
    private @EJB PaymentProcessor paymentProcessor;
    private @PersistenceContext(type=EXTENDED) EntityManager entityManager;

    @Initializer
    ShoppingCart(User customer) {
        this.customer = customer;
    }

    @PostConstruct
    void retrieveOrder() {
        order = entityManager.find( Order.class, customer.getId() );
    }

    ...

    @PreDestroy
    void updateOrder() {
        entityManager.merge(order);
    }

}
```

Of course, enterprise Web Beans may take advantage of all functionality defined by the EJB specification.

3.11. The `Bean` object for a Web Bean

The abstract class `Bean` provides everything the container needs to manage instances of a certain Web Bean.

```
public abstract class Bean<T>
        implements Contextual<T> {

    private final Manager manager;

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

    protected Manager getManager() {
        return manager;
    }

    public abstract Set<Type> 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();

}
```

Note that concrete subclasses of `Bean` must implement the operations defined by the `Contextual` interface defined in Section 9.1, “The Contextual interface”.

An instance of `Bean` exists for every enabled Web Bean in a deployment.

An application or third party framework may add support for new kinds of Web Beans beyond those defined by the Web Beans specification (simple Web Beans, enterprise Web Beans, producer methods and fields and JMS endpoints) by extending `Bean` and registering Web Beans with the container, using the mechanism defined in Section 11.2, “Web Bean registration”.

Chapter 4. Inheritance, specialization and realization

Multiple Web Beans may share the same implementation. The implementation of one Web Bean may be shared by a second Web Bean in two different ways:

- The implementation of the second Web Bean may extend the implementation of the first Web Bean using Java inheritance
- The second Web Bean may be declared to have the same implementation using XML

In either case, there are three possible reasons for reusing the implementation of the first Web Bean. Either:

- The second Web Bean *specializes* the first Web Bean in a particular deployment scenario. In that deployment, the second Web Bean completely replaces the first, fulfilling the same role in the system.
- The implementation of the first Web Bean is generic, and was designed to fulfill multiple roles in the system. The second Web Bean *realizes* one of these roles. Other Web Beans may also share the implementation of the first Web Bean, and fulfill other roles.
- The second Web Bean is simply reusing the Java implementation, and otherwise bears no relation to the first Web Bean. The first Web Bean may not even have been designed for use as a contextual object.

The three cases are quite dissimilar.

By default, Java implementation reuse is assumed. In this case, the producer, disposal and observer methods of the first Web Bean are not inherited by the second Web Bean.

The Web Bean developer may explicitly specify that the second Web Bean specializes or realizes the first through use of an annotation.

In the case of specialization, the specialized Web Bean receives all invocations, including producer, disposal and observer method invocations that would have been received by the first Web Bean. In a particular deployment, there may be only one Web Bean that fulfills the specific role. The specialized Web Bean inherits, and may not override, the binding types and name of the first Web Bean.

In the case of realization, the second Web Bean inherits the producer, disposal and observer methods of the generic Web Bean, but in this case, the inherited members have a distinct identity, since the second Web Bean has its own role in the system, distinct from all the other Web Beans that share the implementation of the generic Web Bean. The second Web Bean must declare a distinct set of binding types and name (if any).

However, in all three cases, the inheritance of type-level metadata is controlled via use of the Java `@Inherited` meta-annotation.

4.1. Inheritance of type-level metadata

Suppose a class X is extended directly or indirectly by the implementation class of a simple or enterprise Web Bean Y.

- If X is annotated with a Web Beans binding type, stereotype or interceptor binding type Z then Y inherits the annotation if and only if Z declares the `@Inherited` meta-annotation and neither Y nor any intermediate class that is a subclass of X and a superclass of Y declares an annotation of type Z.
- If X is annotated with a Web Beans scope type Z then Y inherits the annotation if and only if Z declares the `@Inherited` meta-annotation and neither Y nor any intermediate class that is a subclass of X and a superclass of Y declares a scope type.
- If X is annotated with a Web Beans deployment type Z then Y inherits the annotation if and only if Z declares the `@Inherited` meta-annotation and neither Y nor any intermediate class that is a subclass of X and a superclass of Y declares a deployment type.

Scope types and deployment types explicitly declared by and inherited from the class X take precedence over default scope types and deployment types declared by stereotypes.

Suppose a class X is the implementation class of a simple or enterprise Web Bean Y declared using XML.

- If X is annotated with a Web Beans binding type, stereotype or interceptor binding type Z then Y inherits the annotation if and only if Z declares the `@Inherited` meta-annotation and Y does not explicitly declare an annotation of type Z using XML.
- If X is annotated with a Web Beans scope type Z then Y inherits the annotation if and only if Z declares the `@Inherited` meta-annotation and Y does not explicitly declare a scope type using XML.
- If X is annotated with a Web Beans deployment type Z then Y inherits the annotation if and only if Z declares the `@Inherited` meta-annotation and Y does not explicitly declare a deployment type using XML.

Scope types and deployment types explicitly declared by and inherited from the class X take precedence over default scope types and deployment types declared by stereotypes.

For annotations defined by the Web Bean specification:

- all built-in scope types are declare `@Inherited`,
- all built-in stereotypes are declared `@Inherited`,
- no built-in binding type is declared `@Inherited`, and
- the built-in deployment type is not declared `@Inherited`.

For annotations defined by the application or third-party extensions, it is recommended that:

- scope types should be declared `@Inherited`,
- binding types should not be declared `@Inherited`,
- deployment types should not be declared `@Inherited`,
- interceptor binding types should be declared `@Inherited`, and
- stereotypes may be declared `@Inherited`, depending upon the semantics of the stereotype.

However, in special circumstances, these recommendations may be ignored.

4.2. Inheritance of member-level metadata

Suppose a class X is extended directly or indirectly by the implementation class of a simple or enterprise Web Bean Y.

- If X declares an injected field `x` then Y inherits `x`.
- If X declares an initializer method, `@PostConstruct` method or `@PreDestroy` method `x()` then Y inherits `x()` if and only if neither Y nor any intermediate class that is a subclass of X and a superclass of Y overrides the method `x()`.
- If X declares a non-static method `x()` annotated with an interceptor binding type Z then Y inherits the binding if and only if neither Y nor any intermediate class that is a subclass of X and a superclass of Y overrides the method `x()`.
- If X declares a non-static producer, disposal, or observer method `x()` then Y does not inherit this method unless Y is explicitly declared to specialize or realize X.
- If X declares a non-static producer field `x` then Y does not inherit this field unless Y is explicitly declared to specialize or realize X.
- If Y is a decorator and X declares a delegate attribute `x` then Y inherits `x` if and only if neither Y nor any intermediate class that is a subclass of X and a superclass of Y defines a delegate attribute.

Suppose a class X is the implementation class of a simple or enterprise Web Bean Y declared using XML.

- If X declares an injected field `x` then Y inherits `x`, unless Y explicitly declares `x` using XML.
- If X declares an initializer method, `@PostConstruct` method or `@PreDestroy` method `x()` then Y inherits `x()`, unless Y explicitly declares `x()` using XML.
- If X declares a non-static method `x()` annotated with an interceptor binding type Z then Y inherits the binding, unless Y explicitly declares `x()` using XML.
- If X declares a non-static producer, disposal, or observer method `x()` then Y does not inherit this method, unless Y is explicitly declared to specialize or realize X.
- If X declares a non-static producer field `x` then Y does not inherit this method, unless Y is explicitly declared to specialize or realize X.
- If Y is a decorator and X declares a delegate attribute `x` then Y inherits `x`, unless Y explicitly declares a delegate attribute using XML.

4.3. Specialization

If two Web Beans both support a certain API type, and share at least one binding type, then they are both eligible for injection to any injection point with that declared type and binding type. The container will choose the Web Bean with the highest priority enabled deployment type.

Consider the following Web Beans:

```
@Current @Asynchronous
public class AsynchronousService implements Service{
    ...
}
```

```
@Mock @Current
public class MockAsynchronousService extends AsynchronousService {
    ...
}
```

Suppose that the deployment type `@Mock` is enabled:

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

Then the following attribute will receive an instance of `MockAsynchronousService`:

```
@Current Service service;
```

However, if the Web Bean with the lower priority deployment type declares a binding annotation that is not declared by the Web Bean with the higher priority deployment type, then the Web Bean with the higher priority deployment type will not be eligible for injection to an injection point with that binding type.

Therefore, the following attribute will receive an instance of `AsynchronousService` even though the deployment type `@Mock` is enabled:

```
@Current @Asynchronous Service service;
```

This is a useful feature in many circumstances, however, it is not always what is intended by the developer.

The only way one Web Bean can completely override a lower-priority Web Bean at all injection points is if it implements all the API types and declares all the binding types of the lower-priority Web Bean. However, if the lower-priority Web Bean declares a producer method, then even this is not enough to ensure that the lower-priority Web Bean is never called!

To help prevent developer error, the first Web Bean may:

- directly extend the implementation class of the lower-priority Web Bean, in the case of a Web Bean declared using annotations, or
- declare the same implementation class as the lower-priority Web Bean, in the case of a Web Bean declared using XML, or
- directly override the lower-priority producer method, in the case of a producer method Web Bean, and then

explicitly declare that it *specializes* the lower-priority Web Bean.

4.3.1. Using specialization

A Web Bean declared using annotations may declare that it specializes a lower-priority Web Bean using the `@Specializes` annotation. A Web Bean declared using XML may declare that it specializes a lower-priority Web Bean using the `<Specializes>` element.

Then the first Web Bean will inherit the binding types and name of the lower-priority Web Bean:

- The binding types of a Web Bean X that specializes a lower-priority Web Bean Y include all binding types of Y, together with all binding types declared explicitly by X.
- If a Web Bean X specializes a lower-priority Web Bean Y with a name, the name of X is the same as the name of Y. If X declares a name explicitly, a `DefinitionException` is thrown by the container at deployment time.

For example, the following Web Bean would have the inherited binding types `@Current` and `@Asynchronous`:

```
@Mock @Specializes
public class MockAsynchronousService extends AsynchronousService {
    ...
}
```

If `AsynchronousService` declared a name:

```
@Current @Asynchronous @Named("asyncService")
public class AsynchronousService implements Service{
    ...
}
```

Then the name would also automatically be inherited by `MockAsynchronousService`.

When an enabled Web Bean specializes a lower-priority Web Bean, we can be certain that the lower-priority Web Bean is never instantiated or called by the container. Even if the lower-priority Web Bean defines a producer method, the method will be called upon an instance of the first Web Bean.

Specialization applies only to simple Web Beans, as defined in Section 3.2.6, “Specializing a simple Web Bean”, enterprise Web Beans, as defined in Section 3.3.6, “Specializing an enterprise Web Bean” and producer methods, as defined in Section 3.4.5, “Specializing a producer method”.

4.3.2. Direct and indirect specialization

The `@Specializes` annotation or `<Specializes>` XML element is used to indicate that one Web Bean *directly specializes* another Web Bean.

Formally, a Web Bean X is said to *specialize* another Web Bean Y if either:

- X directly specializes Y, or
- a Web Bean Z exists, such that X directly specializes Z and Z specializes Y.

If X specializes Y but does not directly specialize Y, we say that X *indirectly specializes* Y.

If, in a particular deployment, a Web Bean with a certain API type and set of binding types is not specialized by any other enabled Web Bean, we call it the *most specialized Web Bean* for that combination of type and binding types in that deployment.

Any non-static producer methods (see Section 3.4, “Producer methods”), producer fields (see Section 3.5, “Producer fields”), disposal methods (see Section 3.4.6, “Disposal methods”) or observer methods (see Section 8.5, “Observer methods”) of any Web Bean are invoked upon an instance of the most specialized enabled Web Bean that specializes the Web Bean, as defined by Section 6.6, “Lifecycle of producer methods”, Section 6.7, “Lifecycle of producer fields” and Section 8.4, “Observer invocation”.

4.3.3. Inconsistent specialization

If, in a particular deployment, either

- some enabled Web Bean X specializes another enabled Web Bean Y and X does not have a higher precedence than Y, or
- more than one enabled Web Bean directly specializes the same Web Bean

we say that *inconsistent specialization* exists, and an `InconsistentSpecializationException` is thrown by the container at deployment time.

4.4. Realization

Third-party frameworks and libraries often define generic classes that are intended for reuse by the application.

Consider the following generic class that defines a producer method, a disposal method and an observer method:

```
@ApplicationScoped
public abstract class PersistenceContext {

    protected abstract EntityManager createEntityManager();

    @Produces @ConversationScoped EntityManager getEntityManager() {
        return createEntityManager();
    }

    void closeEntityManager(@Disposes EntityManager em) {
        em.close();
    }

    void beforeDirectJdbcQuery(@Observes @Before DirectJdbcQuery event, EntityManager em) {
        em.flush();
    }

}
```

This class is intended to fulfill multiple roles in the system—for every database in use by the application, there should be a Web Bean that extends this class and provides an implementation of `createEntityManager()`. However, each Web Bean that extends `PersistenceContext` needs to define a different set of binding types for the producer and disposal methods. Furthermore, the observer method should be inherited by all Web Beans that extend this class.

However, it is not necessary to force all subclasses to override the producer and disposal methods just in order to override the binding types.

Instead, any Web Bean that extends a generic class may:

- directly extend the generic class, in the case of a Web Bean declared using annotations, or
- declare that the generic class is the implementation class, in the case of a Web Bean declared using XML, and then

explicitly declare that it *realizes* the generic class.

4.4.1. Using realization

A Web Bean declared using annotations may declare that it realizes a generic class by annotating the implementation class with the `@Realizes` annotation. A Web Bean declared using XML may declare that it realizes a generic class using the `<Realizes>` element.

Then the first Web Bean will inherit producer, disposal and observer methods declared by the generic class:

- If a generic class Y declares a non-static producer method or field with a certain combination of scope, stereotypes, binding types and interceptor bindings, then every Web Bean X that realizes Y also has a producer method or field with the same scope, stereotypes and interceptor bindings. The binding types for this inherited producer method or field consist of all binding types declared by the producer method or field of Y, excluding all binding types of Y, together with the binding types declared explicitly by X. The deployment type of the inherited producer method or field is the deployment type of X.
- If a generic class Y declares a non-static disposal method with a disposed parameter with a certain combination of binding types, then every Web Bean X that realizes Y also has a disposal method. The binding types of the disposed parameter of this inherited disposal method consist of all binding types declared by the disposed parameter of the disposal method of Y, excluding all binding types of Y, together with the binding types declared explicitly by X.
- If a generic class Y declares a non-static observer method with an event parameter with a certain combination of event binding types, then every Web Bean X that realizes Y also has an observer method. The event binding types of the event parameter of this inherited observer method consist of all event binding types declared by the event parameter of the observer method of Y.

Open issue: do we need a way to inherit the binding types of X to the event binding types of the observer method?

For example, the following Web Bean would have a producer method with binding type `@CustomerDatabase`, scope type `@ConversationScoped` and deployment type `@Staging`, a disposal method with binding type `@CustomerDatabase` and an observer method with event type `DirectJdbcQuery` and event binding type `@Before`:

```
@Staging @CustomerDatabase @Realizes
public class CustomerDatabasePersistenceContext extends PersistenceContext {

    @Override protected EntityManager createEntityManager() { ... }

}
```

Realization applies only to simple Web Beans and enterprise Web Beans.

Chapter 5. Lookup, dependency injection and EL resolution

Web Beans defines the following *injection points*:

- Any injected field of a Web Bean implementation class
- Any parameter of a Web Bean constructor, initializer method, producer method or disposal method
- Any parameter of an observer method, except for the event parameter

Web Bean instances may also be obtained by evaluating EL expressions which refer to the Web Bean by name.

In general, an API type or Web Bean name does not uniquely identify a Web Bean. When resolving a Web Bean at an injection point, the container considers API type, binding annotations and Web Bean deployment type precedence. When resolving a Web Bean name in EL, the container considers name and deployment type precedence. This allows Web Bean developers to decouple type from implementation.

The container is required to ensure that any injected reference to a Web Bean instance may be cast to any API type of the Web Bean.

5.1. Unsatisfied and ambiguous dependencies

An *unsatisfied dependency* exists at an injection point when no enabled Web Bean has the API type and binding types declared by the injection point.

An *ambiguous dependency* exists at an injection point when in the set of enabled Web Beans with the API type and binding types declared by the injection point there exists no unique Web Bean with a higher precedence than all other Web Beans in the set.

The container must validate all injection points of all enabled Web Beans at deployment time to ensure that there are no unsatisfied or ambiguous dependencies. If an unsatisfied or ambiguous dependency exists, an `UnsatisfiedDependencyException` or `AmbiguousDependencyException` is thrown by the container at deployment time, as defined in Section 5.10, “Instance resolution”.

5.2. Primitive types and null values

For the purposes of typesafe resolution and dependency injection, primitive types and their corresponding wrapper types in the package `java.lang` are considered identical and assignable. If necessary, the container performs boxing or unboxing when it injects a value to a field or parameter of primitive or wrapper type.

However, if an injection point of primitive type resolves to a Web Bean that may be null, such as a producer method with a non-primitive return type or a producer field with a non-primitive type, a `NullableDependencyException` is thrown by the container at deployment time.

The method `Bean.isNullable()` may be used to detect if a Web Bean has null values.

5.3. Injected reference validity

References to Web Bean instances are *valid* only for a certain period of time. The application should not invoke a method of an invalid reference.

The validity of an injected reference depends upon whether the scope of the injected Web Bean is a normal scope or a pseudo-scope.

- Any reference to a Web Bean with a normal scope is valid as long as the application maintains a hard reference to it. However, it may only be invoked when the context associated with the normal scope is active. If it is invoked when the context is inactive, a `ContextNotActiveException` is thrown by the container.
- Any reference to a Web Bean with a pseudo-scope (such as `@Dependent`) is valid until the Web Bean instance to which it refers is destroyed. It may be invoked even if the context associated with the pseudo-scope is not active. If the ap-

plication invokes a method of a reference to an instance that has already been destroyed, the behavior is undefined.

5.4. Client proxies

Clients of a Web Bean with a normal scope type, as defined in Section 9.3, “Normal scopes and pseudo-scopes”, do not hold a direct reference to the instance of the Web Bean (the object returned by `Bean.create()`). Instead, their reference is to a *client proxy* object. A client proxy implements/extends all API types of the Web Bean and delegates all method calls to the current instance (as defined in Section 9.3, “Normal scopes and pseudo-scopes”) of the Web Bean.

There are a number of reasons for this indirection:

- The container must guarantee that when any valid injected reference to a Web Bean of normal scope is invoked, the invocation is always processed by the current instance of the injected Web Bean. In certain scenarios, for example if a request scoped Web Bean is injected into a session scoped Web Bean, or into a Servlet, this rule requires an indirect reference. (Note that the `@Dependent` pseudo-scope is not a normal scope.)
- The container may use a client proxy when creating Web Beans with circular dependencies. This is only necessary when the circular dependencies are initialized via a simple Web Bean constructor or producer method parameter. (Web Beans with scope `@Dependent` never have circular dependencies.)
- Finally, client proxies are serializable, even when the Web Bean itself is not. Therefore the container must use a client proxy whenever a Web Bean with normal scope is injected into a Web Bean with a passivating scope, as defined in Section 9.5, “Passivating scopes and serialization”. (On the other hand, Web Beans with scope `@Dependent` must be serialized along with their client.)

Client proxies are never required for a Web Bean whose scope is a pseudo-scope such as `@Dependent`.

All client proxies must be serializable.

Client proxies may be shared between multiple injection points. For example, a particular container might instantiate exactly one client proxy object per Web Bean. (However, this strategy is not required by the Web Beans specification.)

5.4.1. Unproxyable API types

Certain legal API types cannot be proxied by the container:

- classes without a non-private constructor with no parameters,
- classes which are declared final or have final methods,
- primitive types,
- and array types.

If an injection point whose declared type cannot be proxied by the container resolves to a Web Bean with a normal scope type, an `UnproxyableDependencyException` is thrown by the container at deployment time.

5.4.2. Client proxy invocation

Every time a method of the Web Bean is invoked upon a client proxy, the client proxy must:

- obtain the context object by calling `Manager.getContext()`, passing the Web Bean scope, then
- obtain an instance of the Web Bean by calling `Context.get()`, passing the `Bean` instance representing the Web Bean and `true` as the value of the `create` parameter, and
- invoke the method upon the Web Bean.

The behavior of all methods declared by `java.lang.Object`, except for `toString()`, is undefined for a client proxy. Portable applications should not invoke any method declared by `java.lang.Object`, except for `toString()`, on a client proxy.

5.5. The default binding type at injection points

If an injection point declares no binding type, the default binding type `@Current` is assumed.

The following are equivalent:

```
@ConversationScoped
public class Order {

   private Product product;
   private User customer;

   @Initializer
   public void init(@Selected Product product, User customer)
   {
       this.product = product;
       this.customer = customer;
   }

}
```

```
@ConversationScoped
public class Order {

   private Product product;
   private User customer;

   @Initializer
   public void init(@Selected Product product, @Current User customer)
   {
       this.product = product;
       this.customer = customer;
   }

}
```

As are the following:

```
<myapp:Order>
    <ConversationScoped/>

    <myapp:init>
        <Initializer/>
        <myapp:Product>
            <myapp:Selected/>
        </myapp:Product>
        <myapp:User/>
    </myapp:init>

</myapp:Order>
```

```
<myapp:Order>
    <ConversationScoped/>

    <myapp:init>
        <Initializer/>
        <myapp:Product>
            <myapp:Selected/>
        </myapp:Product>
        <myapp:User>
            <Current/>
        </myapp:User>
    </myapp:init>

</myapp:Order>
```

The following definitions are equivalent:

```
public class Payment {

   public Payment(BigDecimal amount) { ... }

   @Initializer Payment(Order order) {
      this(order.getAmount();
   }

}
```

```
public class Payment {

   public Payment(BigDecimal amount) { ... }

   @Initializer Payment(@Current Order order) {
      this(order.getAmount();
   }

}
```

As are the following:

```
<myapp:Payment>
    <myapp:Order/>
</myapp:Payment>
```

```
<myapp:Payment>
    <myapp:Order>
        <Current/>
    </myapp:Order>
</myapp:Payment>
```

5.6. Generic type literals

The Java language does not currently support a literal syntax for parameterized types. Therefore, Web Beans provides the following helper class to allow inline instantiation of an object that represents a parameterized type.

```
public abstract class TypeLiteral<T> {

        protected TypeLiteral() {
                if (!(getClass().getSuperclass() == TypeLiteral.class)) {
                        throw new RuntimeException("Not a direct subclass of TypeLiteral");
                }
                if (!(getClass().getGenericSuperclass() instanceof ParameterizedType)) {
                        throw new RuntimeException("Missing type parameter in TypeLiteral");
                }
        }

        public final Type getType() {
                ParameterizedType parameterized = (ParameterizedType) getClass()
                                .getGenericSuperclass();
                return parameterized.getActualTypeArguments()[0];
        }

        @SuppressWarnings("unchecked")
        public final Class<T> getRawType() {
                Type type = getType();
                if (type instanceof Class) {
                        return (Class<T>) type;
                } else if (type instanceof ParameterizedType) {
                        return (Class<T>) ((ParameterizedType) type).getRawType();
                } else if (type instanceof GenericArrayType) {
                        return (Class<T>) Object[].class;
                } else {
                        throw new RuntimeException("Illegal type parameter in TypeLiteral");
                }
        }

}
```

An object that represents any parameterized type may be obtained by subclassing `TypeLiteral`.

```
TypeLiteral type = new TypeLiteral<List<String>>() {};
```

This object may be passed to Web Beans APIs that perform typesafe resolution.

5.7. Annotation type literals

The Java language does not currently support a literal syntax for inline instantiation of annotation values. Therefore, Web Beans provides the following helper class to allow inline instantiation of annotation type instances.

```
public abstract class AnnotationLiteral<T extends Annotation>
        implements Annotation {
```

```
        protected AnnotationLiteral() {
                if (!(getClass().getSuperclass() == AnnotationLiteral.class)) {
                        throw new RuntimeException(
                                        "Not a direct subclass of AnnotationLiteral");
                }
                if (!(getClass().getGenericSuperclass() instanceof ParameterizedType)) {
                        throw new RuntimeException(
                                        "Missing type parameter in AnnotationLiteral");
                }
        }

        @SuppressWarnings("unchecked")
        public final Class<T> annotationType() {
                ParameterizedType parameterized = (ParameterizedType) getClass()
                                .getGenericSuperclass();
                return (Class<T>) parameterized.getActualTypeArguments()[0];
        }
}
```

An instance of an annotation type may be obtained by subclassing `AnnotationLiteral`.

```
public abstract class PayByBinding
    extends AnnotationLiteral<PayBy>
    implements PayBy {}
```

```
PayBy payby = new PayByBinding() { public value() { return CHEQUE; } };
```

Annotation values are often passed to Web Beans APIs that perform typesafe resolution.

5.8. The `Manager` object

Occasionally, the application or third-party framework must interact directly with the container via programmatic API call. This is useful in generic framework code—when we need to obtain a Web Bean instance, but the type or binding types vary dynamically, for example. Thus, the `Manager` interface provides operations for resolving a Web Bean by type or name. The container provides an implementation of this interface to the application.

The container provides a built-in Web Bean with API type `javax.webbeans.Manager`, scope `@Dependent`, deployment type `@Standard` and binding type `@Current`. Thus, any Web Bean may obtain an instance of `Manager` by injecting it:

```
@Current Manager manager;
```

Alternatively, the application may obtain the `Manager` object from JNDI. The container must register an instance of `Manager` with name `java:comp/Manager` in JNDI at deployment time.

A contextual instance of a Web Bean may be obtained by calling `Manager.getInstance()`, passing the `Bean` object representing the Web Bean.

```
public interface Manager {

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

    ...

}
```

`Manager.getInstance()` returns a Web Bean instance or client proxy.

- If the given `Bean` instance represents a Web Bean with a normal scope, as defined in Section 9.3, “Normal scopes and pseudo-scopes”, `Manager.getInstance()` must return a client proxy.
- Otherwise, if the `Bean` instance represents a Web Bean with a pseudo-scope, as defined in Section 9.3, “Normal scopes and pseudo-scopes”, `Manager.getInstance()` must:
    - obtain the context object by calling `Manager.getContext()`, passing the Web Bean scope, then
    - obtain an instance of the Web Bean by calling `Context.get()`, passing the `Bean` instance representing the Web Bean and `true` as the value of the `create` parameter, and return it.

5.9. Dynamic lookup

In certain situations, injection is not the most convenient way to obtain a reference to a Web Bean instance. For example, it may not be used when:

- the bindings vary dynamically at runtime, or
- depending upon the deployment, there may be no Web Bean which satisfies the type and bindings.

In these situations, the application may directly call `Manager.getInstanceByType()`.

Alternatively, an instance of the `Instance` interface may be injected via use of the `@Obtains` binding annotation:

```
@Obtains Instance<PaymentProcessor> paymentProcessor;
```

Additional binding annotations may be specified at the injection point:

```
@Obtains @PayBy(CHEQUE) Instance<PaymentProcessor> paymentProcessor;
```

The `Instance` interface provides a method for obtaining instances of Web Beans of a specific type:

```
public interface Instance<T> {

    public T get(Annotation... bindings);

}
```

If two instances of the same binding type are passed to `get()`, a `DuplicateBindingTypeException` is thrown.

If an instance of an annotation that is not a binding type is passed to `get()`, an `IllegalArgumentException` is thrown.

The `@Obtains` annotation or `<Obtains>` element may be applied to any field of a Web Bean implementation class or to any parameter of a producer method, initializer method, disposal method or Web Bean constructor where the type of the field or parameter is `Instance`, and an actual type parameter is specified.

If the type of the injection point is not of type `Instance`, if no actual type parameter is specified, or if the type parameter contains a type variable or wildcard, a `DefinitionException` is thrown by the container at deployment time.

Whenever the `@Obtains` annotation appears at an injection point, an implicit Web Beans exists with:

- exactly the API type and binding annotations that appear at the injection point,
- deployment type `@Standard`,
- `@Dependent` scope,
- no Web Bean name, and
- an implementation provided automatically by the container.

The `get()` method of the provided implementation of `Instance` must call `Manager.getInstanceByType()`, passing the following parameters:

- all binding annotations declared at the injection point, except `@Obtains`
- all binding annotation instances passed to `Instance.get()`

Open issue: if no Web Bean satisfies the type and bindings, should an exception be thrown, or a null value returned.

The application may obtain a Web Bean instance by calling the `get()` method:

```
@Obtains @PayBy(CHEQUE) Instance<PaymentProcessor> paymentProcessor;
...
Annotation binding = processSynchronously ?
        new SynchronousBinding() {} : new AsynchronousBinding() {};
paymentProcessor.get(binding).process(payment);
```

In this example, the returned Web Bean has API type `PaymentProcessor` and binding `@PayBy(CHEQUE)` along with either `@Synchronous` or `@Asynchronous`.

When the application calls `Instance.get()` to obtain a Web Bean instance dynamically, it may need to pass instances of binding annotation types. The helper class `javax.webbeans.AnnotationLiteral` makes it easier to implement binding annotation types:

```
public class SynchronousBinding
          extends AnnotationLiteral<Synchronous>
          implements Synchronous {}
```

```
public abstract class PayByBinding
          extends AnnotationLiteral<PayBy>
          implements PayBy {}
```

Then the application may easily instantiate instances of the binding type:

```
PaymentProcessor pp = paymentProcessor.get( new SynchronousBinding(),
        new PayByBinding() { public PaymentMethod value() { return CHEQUE; } });
```

5.10. Instance resolution

The `Manager.getInstanceByType()` methods obtain a contextual instance of a Web Bean:

```
public interface Manager {

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

    ...

}
```

The first argument is a Web Bean API type, the remaining arguments are instances of binding annotation types.

For example:

```
PaymentProcessor pp = manager.getInstanceByType(PaymentProcessor.class,
        synchronousAnnotation,
        payByAnnotation);
```

If no binding annotations are passed to `getInstanceByType()`, the default binding type `@Current` is assumed.

If a parameterized type with a type parameter or wildcard is passed to `resolveByType()`, an `IllegalArgumentException` is thrown.

If two instances of the same binding type are passed to `getInstanceByType()`, a `DuplicateBindingTypeException` is thrown.

If an instance of an annotation that is not a binding type is passed to `getInstanceByType()`, an `IllegalArgumentException` is thrown.

The `getInstanceByType()` method must:

- Identify the Web Bean by calling `Manager.resolveByType()`, passing the type and binding annotations of the injection point.
- If `resolveByType()` did not return a Web Bean, throw an `UnsatisfiedDependencyException` or, if `resolveByType()` returned more than one Web Bean, throw an `AmbiguousDependencyException`.
- If the Web Bean has a normal scope type and the type cannot be proxied by the container, as defined in Section 5.4.1, “Unproxyable API types”, throw an `UnproxyableDependencyException`.
- Otherwise, obtain an instance of the Web Bean (or a client proxy) by calling `Manager.getInstance()`, passing the

`Bean` object representing the Web Bean, and return it.

The `getInstanceByType()` method is called whenever the container injects a Web Bean instance into another Web Bean.

5.10.1. Typesafe resolution algorithm

The process of matching a Web Bean to an injection point is called *typesafe resolution*. The container considers API type, binding annotations, and Web Bean precedences when resolving a Web Bean to be injected to an injection point.

Typesafe resolution usually occurs at container deployment time, allowing the container to warn the user if any enabled Web Beans have unsatisfied or ambiguous dependencies.

The `resolveByType()` method of the `Manager` interface returns the result of the typesafe resolution.

```
public interface Manager {

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

    ...

}
```

If no binding annotations are passed to `resolveByType()`, the default binding annotation `@Current` is assumed.

If a parameterized type with a type parameter or wildcard is passed to `resolveByType()`, an `IllegalArgumentException` is thrown.

If two instances of the same binding type are passed to `resolveByType()`, a `DuplicateBindingTypeException` is thrown.

If an instance of an annotation that is not a binding type is passed to `resolveByType()`, an `IllegalArgumentException` is thrown.

The following algorithm must be used by the container when resolving a Web Bean by type:

- First, the container identifies the set of *matching* enabled Web Beans which have the given API type. For this purpose, primitive types are considered to be identical to their corresponding wrapper types in `java.lang`, array types are considered identical only if their element types are identical and parameterized types are considered identical only if both the type and all type parameters are identical.
- Next, the container considers the given binding annotations. If no binding annotations were passed to `resolveByType()`, the container assumes the binding annotation `@Current`. The container narrows the set of matching Web Beans to just those where for each given binding annotation, the Web Bean declares a binding annotation with (a) the same type and (b) the same annotation member value for each member which is not annotated `@NonBinding` (see Section 5.10.1.1, “Binding annotations with members”).
- Next, the container examines the deployment types of the matching Web Beans, as defined in Section 2.5.7, “Deployment type precedence”, and returns the set of Web Beans with the highest precedence deployment type that occurs in the set. If there are no matching Web Beans, an empty set is returned.

If `resolveByType()` is called with the API type `java.lang.Object` and no binding annotations, it returns the set of all enabled Web Beans.

5.10.1.1. Binding annotations with members

According to the algorithm above, binding annotations with members are supported:

```
@PayBy(CHEQUE)
class ChequePaymentProcessor implements PaymentProcessor { ... }
```

```
@PayBy(CREDIT_CARD)
class CreditCardPaymentProcessor implements PaymentProcessor { ... }
```

Then only `ChequePaymentProcessor` is a candidate for injection to the following attribute:

```
@PayBy(CHEQUE) PaymentProcessor paymentProcessor;
```

On the other hand, only `CreditCardPaymentProcessor` is a candidate for injection to this attribute:

```
@PayBy(CREDIT_CARD) PaymentProcessor paymentProcessor;
```

The container calls the `equals()` method of the annotation member value to compare values.

An annotation member may be excluded from consideration using the `@NonBinding` annotation.

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

Array-valued or annotation-values members of a binding annotation must be annotated `@NonBinding`. If an array-valued or annotation-valued member of a binding annotation is not annotated `@NonBinding`, a `DefinitionException` is thrown by the container at deployment time.

5.10.1.2. Multiple binding annotations

According to the algorithm above, a Web Bean implementation class or producer method or field may declare multiple binding annotations:

```
@Synchronous @PayBy(CHEQUE)
class ChequePaymentProcessor implements PaymentProcessor { ... }
```

Then `ChequePaymentProcessor` would be considered a candidate for injection into any of the following attributes:

```
@PayBy(CHEQUE) PaymentProcessor paymentProcessor;
```

```
@Synchronous PaymentProcessor paymentProcessor;
```

```
@Synchronous @PayBy(CHEQUE) PaymentProcessor paymentProcessor;
```

A Web Bean must declare *all* of the binding annotations that are specified at the injection point to be considered a candidate for injection.

5.11. Injection point metadata

Occasionally, a component with scope `@Dependent` needs to access metadata relating to the object into which it is injected. For example, the following producer method creates injectable `Loggers`. The log category of a `Logger` depends upon the class of the object into which it is injected:

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

The interface `InjectionPoint` provides access to metadata about the injection point into which a dependent object is injected.

```
public interface InjectionPoint {
    public Type getType();
    public Set<Annotation> getBindingTypes();
    public Object getInstance();
    public Bean<?> getBean();
    public Member getMember():
    public <T extends Annotation> T getAnnotation(Class<T> annotationType);
    public Annotation[] getAnnotations();
    public boolean isAnnotationPresent(Class<? extends Annotation> annotationType);
}
```

- The `getInstance()` method returns the Web Bean instance into which the dependent object was injected. If `getInstance()` is invoked while the instance is being created or destroyed, an `IllegalStateException` is thrown.
- The `getBean()` method returns the `Bean` object representing the Web Bean that defines the injection point.
- The `getType()` and `getBindingTypes()` methods return the declared type and binding types of the injection point. If the injection point is declared in XML, the type and binding types are determined according to Section 10.8, “Specifying API types and binding types”.
- The `getMember()` method returns the `Field` object in the case of field injection, the `Method` object in the case of method parameter injection or the `Constructor` object in the case of constructor parameter injection.
- The `getAnnotation()` and `getAnnotations()` methods return annotations of the field in the case of field injection, or annotations of the parameter in the case of method parameter or constructor parameter injection. `getAnnotation()` returns a null value if no annotation of the given type exists at the injection point.

The container must provide a Web Bean with deployment type `@Standard`, scope `@Dependent`, API type `InjectionPoint` and binding type `@Current`.

- Whenever a `@Dependent` scoped object is instantiated by the container for injection into a second Web Bean, any injection point of type `InjectionPoint` and binding type `@Current` receives an instance of `InjectionPoint` that represents the injection point of the second Web Bean.
- Otherwise, when a `@Dependent` scoped object is instantiated by the container to receive a producer method, producer field, observer or disposal method invocation, during EL expression evaluation, or as a result of a direct call to the `Manager` API, any injection point of type `InjectionPoint` and binding type `@Current` receives a null value.

Open issue: should we say that an exception is thrown by the container, instead of just passing a null value?

If a Web Bean that declares any scope other than `@Dependent` has an injection point of type `InjectionPoint` and binding type `@Current`, a `DefinitionException` is thrown by the container at deployment time.

If an object that is not a Web Bean has an injection point of type `InjectionPoint` and binding type `@Current`, a `DefinitionException` is thrown by the container at deployment time.

5.12. EL name resolution

The container must provide a Unified EL `ELResolver` to the Servlet engine and JSF implementation that resolves Web Bean names. When this resolver is called with a null base object, it calls the method `Manager.getInstanceByName()` to obtain an instance of the Web Bean named in the EL expression:

```
public interface Manager {

    public Object getInstanceByName(String name);

    ...

}
```

For example:

```
Object pp = manager.getInstanceByName("paymentProcessor");
```

The `getInstanceByName()` method must:

- Identify the Web Bean by calling `Manager.resolveByName()`, passing the name.
- If `resolveByName()` returned an empty set, return a null value.
- Otherwise, if `resolveByName()` returned more than one Web Bean, throw an `AmbiguousDependencyException`.
- Otherwise, if exactly one Web Bean was returned, obtain an instance of the Web Bean by calling `Manager.getInstance()`, passing the `Bean` instance representing the Web Bean.

For each distinct name that appears in the EL expression, `getInstanceByName()` must be called at most once. Even if a name appears more than once in the same expression, the container may not call `getInstanceByName()` multiple times with that name. This restriction ensures that there is a unique instance of each Web Bean with scope `@Dependent` in any EL evaluation.

Open issue: Web Beans supports qualified names. The `ELResolver` implements support for qualified names in Unified EL. How exactly does this work?

5.12.1. Name resolution algorithm

The process of matching a Web Bean to a name used in EL is called *name resolution*. Since there is no typing information available in EL, the container may consider only Web Bean names.

The `resolveByName()` method of the `Manager` interface performs name resolution.

```
public interface Manager {

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

    ...

}
```

The following algorithm must be used by the container when resolving a Web Bean by name:

- The container identifies the set of *matching* enabled Web Beans which have the given name.
- Next, the container examines the deployment types of the matching Web Beans, as defined in Section 2.5.7, “Deployment type precedence”, and returns the set of Web Beans with the highest precedence deployment type that occurs in the set. If there are no matching Web Beans, an empty set is returned.

The name resolution algorithm usually occurs at runtime.

Chapter 6. Web Bean lifecycle

The lifecycle of a Web Bean instance is managed by the Web Beans context object for the Web Bean's scope. The context implementation collaborates with the container via the `Context` and `Bean` interfaces to create and destroy Web Bean instances.

The actual mechanics of Web Bean creation and destruction varies according to what kind of Web Bean it is:

- To create an enterprise Web Bean, the container creates an EJB local object reference
- To create a producer method Web Bean instance, the container calls the producer method
- To create a producer field Web Bean instance, the container retrieves the current value of the field
- To create a simple Web Bean, the container calls the Web Bean constructor
- To destroy a stateful session enterprise Web Bean, the container destroys the EJB bean
- To destroy a producer method Web Bean instance, the container calls the disposal method, if any

When the container injects a dependency or resolves an EL name, and there is no existing instance of the Web Bean cached by the context object for the Web Bean scope, the context object automatically creates a new instance of the Web Bean. When a Web Beans context is destroyed, the context object automatically destroys any instances associated with that context.

6.1. Creation

The `Bean.create()` method is responsible for creating new instances of a Web Bean.

```
public abstract class Bean<T> {
    public abstract T create();
    ...
}
```

The `create()` method performs the following tasks:

- obtains an instance of the Web Bean,
- creates the interceptor and decorator stacks and binds them to the instance,
- injects any dependencies,
- sets any initial field values defined in XML, and
- calls the `@PostConstruct` method, if necessary.

If any exception occurs while creating an instance, the exception is rethrown by the `create()` method. If the exception is a checked exception, it is wrapped and rethrown as an (unchecked) `CreationException`.

6.2. Destruction

The `Bean.destroy()` method is responsible for destroying instances of a Web Bean.

```
public abstract class Bean<T> {
    public abstract void destroy(T instance);
    ...
}
```

The `destroy()` method performs the following tasks:

- calls disposal method, if necessary,
- calls the `@PreDestroy` method, if necessary, and
- destroys all dependent objects of the instance, as defined in Section 9.4.2, “Dependent object destruction”.

If any exception occurs while destroying an instance, the exception is caught by the `destroy()` method.

If the application invokes a Web Bean instance after it has been destroyed, the behavior is undefined.

6.3. Lifecycle of simple Web Beans

When the `create()` method of the `Bean` object that represents a simple Web Bean is called:

- First, the container calls the Web Bean constructor to obtain an instance of the Web Bean. For each constructor parameter, the container passes the object returned by `Manager.getInstanceByType()`. The container is permitted to return an instance of a container-generated subclass of the Web Bean implementation class, allowing interceptor and decorator bindings.
- Next, the container initializes the values of any attributes annotated `@EJB`, `@PersistenceContext` or `@Resource`, as defined in the Common Annotations for the Java Platform and EJB 3.0 specifications.
- Next, the container initializes the values of all injected fields. For each injected field, the container sets the value to the object returned by `Manager.getInstanceByType()`.
- Next, the container initializes the values of any fields with initial values specified in XML, as defined in Section 10.3.5, “Field initial value declarations”.
- Next, the container calls all initializer methods. For each initializer method parameter, the container passes the object returned by `Manager.getInstanceByType()`.
- Next, the container builds the interceptor and decorator stacks for the instance as defined in Section 7.2.10, “Interceptor stack creation” and Section 7.3.8, “Decorator stack creation” and binds them to the instance.
- Finally, the container calls the `@PostConstruct` method, if any.

When the `destroy()` method is called:

- The container calls the `@PreDestroy` method, if any.
- Finally, the container destroys dependent objects.

6.4. Lifecycle of stateful session enterprise Web beans

When the `create()` method of a `Bean` object that represents a stateful session enterprise Web Bean that is called, the container creates and returns an enterprise bean proxy, as defined in Section 3.3.8, “Enterprise bean proxies”.

When the `destroy()` method is called, the container destroys the stateful session bean.

If the underlying EJB was already destroyed by direct invocation of a remove method by the application, the container ignores the instance.

Note that the container performs additional work when the underlying EJB bean is created and destroyed, as defined in Section 6.9, “Lifecycle of EJB beans”

6.5. Lifecycle of stateless session and singleton enterprise Web Beans

When the `create()` method of a `Bean` object that represents a stateless session or singleton enterprise Web Bean is called, the container creates and returns an enterprise bean proxy, as defined in Section 3.3.8, “Enterprise bean proxies”.

When the `destroy()` method is called, the container simply discards the proxy and all underlying EJB local object references.

Note that the container performs additional work when the underlying EJB bean is created and destroyed, as defined in Section 6.9, “Lifecycle of EJB beans”

6.6. Lifecycle of producer methods

Any Java object may be returned by a producer method. It is not required that the returned object be an instance of another Web Bean. However, if the returned object is not an instance of another Web Bean, the container will provide none of the following capabilities:

- injection of other Web Beans
- lifecycle callbacks
- method and lifecycle interception

In the following example, the producer method returns instances of other Web Beans:

```
@SessionScoped
public class PaymentStrategyProducer {

   private PaymentStrategyType paymentStrategyType;

   public void setPaymentStrategyType(PaymentStrategyType type) {
      paymentStrategyType = type;
   }

   @Produces PaymentStrategy getPaymentStrategy(@CreditCard PaymentStrategy creditCard,
                                                @Cheque PaymentStrategy cheque,
                                                @Online PaymentStrategy online) {
      switch (paymentStrategyType) {
         case CREDIT_CARD: return creditCard;
         case CHEQUE: return cheque;
         case ONLINE: return online;
         default: throw new IllegalStateException();
      }
    }

}
```

In this case, the object returned by the producer method has already had its dependencies injected, receives lifecycle callbacks and has interception enabled.

But in this example, the returned objects are not Web Bean instances:

```
@SessionScoped
public class PaymentStrategyProducer {

   private PaymentStrategyType paymentStrategyType;

   public void setPaymentStrategyType(PaymentStrategyType type) {
      paymentStrategyType = type;
   }

   @Produces PaymentStrategy getPaymentStrategy() {
      switch (paymentStrategyType) {
         case CREDIT_CARD: return new CreditCardPaymentStrategy();
         case CHEQUE: return new ChequePaymentStrategy();
         case ONLINE: return new OnlinePaymentStrategy();
         default: throw new IllegalStateException();
      }
    }

}
```

In this case, the object returned by the producer method will not have any dependencies injected by the container, receives no lifecycle callbacks and does not have interception enabled.

When the `create()` method of a `Bean` object that represents a producer method is called, the container must invoke the producer method, passing the object returned by `Manager.getInstanceByType()` to each parameter.

- If the producer method is static, the container must invoke the method.
- Otherwise, if the producer method is non-static, the container must:
  - obtain the `Bean` object for the most specialized Web Bean that specializes the Web Bean which declares the producer method, and then
  - obtain an instance of the most specialized Web Bean, by calling `Manager.getInstance()`, passing the `Bean` object representing the Web Bean, and
  - invoke the producer method upon this instance.

The return value of the producer method, after method interception completes, is the new Web Bean instance to be returned by `Bean.create()`.

If the producer method returns a null value and the producer method Web Bean has the scope `@Dependent`, the `create()` method returns a null value.

Otherwise, if the producer method returns a null value, and the scope of the producer method is not `@Dependent`, the `create()` method throws an `IllegalProductException`.

When the `destroy()` method is called, and if there is a disposal method for this producer method, the container must invoke the disposal method, passing the instance given to `destroy()` to the disposed parameter, and the object returned by `Manager.getInstanceByType()` to each of the other parameters.

- If the disposal method is static, the container must invoke the method.
- Otherwise, if the disposal method is non-static, the container must:
  - obtain the `Bean` object for the most specialized Web Bean that specializes the Web Bean which declares the disposal method, and then
  - obtain an instance of the most specialized Web Bean, by calling `Manager.getInstance()`, passing the `Bean` object representing the Web Bean, and
  - invoke the disposal method upon this instance.

Finally, the container destroys dependent objects.

6.7. Lifecycle of producer fields

Any Java object may be the value of a producer field. It is not required that the returned object be an instance of another Web Bean. However, if the object is not an instance of another Web Bean, the container will provide none of the following capabilities:

- injection of other Web Beans
- lifecycle callbacks
- method and lifecycle interception

In the following example, the producer field contains an instance of another Web Bean:

```
@SessionScoped
public class PaymentStrategyProducer {

   @Produces PaymentStrategy paymentStrategy;

   @CreditCard PaymentStrategy creditCard;
   @Cheque PaymentStrategy cheque;
   @Online PaymentStrategy online;

   public void setPaymentStrategyType(PaymentStrategyType type) {
      switch (paymentStrategyType) {
         case CREDIT_CARD: paymentStrategy = creditCard;
         case CHEQUE: paymentStrategy = cheque;
```

```
         case ONLINE: paymentStrategy = online;
         default: throw new IllegalArgmentException();
      }
   }

}
```

In this case, the object contained by the producer field has already had its dependencies injected, received lifecycle callbacks and has interception enabled.

But in this example, the returned objects are not Web Bean instances:

```
@SessionScoped
public class PaymentStrategyProducer {

   @Produces PaymentStrategy paymentStrategy;

   public void setPaymentStrategyType(PaymentStrategyType type) {
      switch (paymentStrategyType) {
         case CREDIT_CARD: paymentStrategy = new CreditCardPaymentStrategy();
         case CHEQUE: paymentStrategy = new ChequePaymentStrategy();
         case ONLINE: paymentStrategy = new OnlinePaymentStrategy();
         default: throw new IllegalArgmentException();
      }
   }

}
```

In this case, the object contained by the producer field does not have any dependencies injected by the container, receives no lifecycle callbacks and does not have interception enabled.

When the `create()` method of a `Bean` object that represents a producer field is called, the container must access the producer field to obtain the current value of the field.

- If the producer method is static, the container must access the field value.
- Otherwise, if the producer method is non-static, the container must:
  - obtain the `Bean` object for the most specialized Web Bean that specializes the Web Bean which declares the producer field, and then
  - obtain an instance of the most specialized Web Bean, by calling `Manager.getInstance()`, passing the `Bean` object representing the Web Bean, and
  - access the field value of this instance.

The value of the producer field is the new Web Bean instance to be returned by `Bean.create()`.

If the producer field contains a null value and the producer field Web Bean has the scope `@Dependent`, the `create()` method returns a null value.

Otherwise, if the producer field contains a null value, and the scope of the producer method is not `@Dependent`, the `create()` method throws an `IllegalProductException`.

6.8. Lifecycle of JMS endpoints

An instance of a JMS endpoint is a *proxy object*, provided by the container, that implements all the API types defined in Section 3.6, “JMS endpoints”, delegating the actual implementation of these methods directly to the underlying JMS objects obtained via JNDI lookup and JMS APIs.

A JMS endpoint proxy object is a dependent object of the object it is injected into.

JMS endpoint proxy objects are serializable.

When the `create()` method of a `Bean` object that represents a JMS endpoint is called, the container creates and returns a special proxy object that implements all the API types of the JMS endpoint.

The methods of this proxy object delegate to JMS objects obtained as needed via JNDI lookup and JMS APIs.

- The `Destination` is obtained by JNDI lookup, using the JNDI name defined in `<destination>`.
- The `ConnectionFactory` is obtained by JNDI lookup, using the JNDI name defined in `<connectionFactory>`.
- The `Connection` is obtained by calling `QueueConnectionFactory.createQueueConnection()` or `TopicConnectionFactory.createTopicConnection()`. The container is permitted to share a connection between multiple proxy objects.
- The `Session` object is obtained by calling `QueueConnection.createQueueSession()` or `TopicConnection.createTopicSession()`.
- The `MessageProducer` object is obtained by calling `QueueSession.createSender()` or `TopicSession.createPublisher()`.

When the `destroy()` method is called, the container must ensure that all JMS objects created by the proxy object are destroyed by calling `close()` if necessary.

- The `Connection` is destroyed by calling `Connection.close()` if necessary. If the connection is being shared between multiple proxy objects, the container is not required to close the connection when the proxy is destroyed.
- The `Session` object is destroyed by calling `Session.close()`.
- The `MessageProducer` object is destroyed by calling `MessageProducer.close()`.

The `close()` method of a JMS endpoint proxy object always throws an `UnsupportedOperationException`.

6.9. Lifecycle of EJB beans

From time to time the EJB container creates EJB bean instances. The container must perform dependency injection upon any EJB session or message driven bean instance that executes in the context of a Web Beans application, regardless of whether it is a Web Bean instance.

When the EJB container creates a new instance of an EJB bean, the container must performs the following steps, after Java EE injection has been performed, and before the `@PostConstruct` callback occurs:

- First, the container initializes the values of all injected fields. For each injected field, the container sets the value to the object returned by `Manager.getInstanceByType()`.
- Next, if the EJB bean instance is an instance of a Web Bean, the container initializes the values of any fields with initial values specified in XML, as defined in Section 10.3.5, “Field initial value declarations”.
- Next, the container calls all initializer methods. For each initializer method parameter, the container passes the object returned by `Manager.getInstanceByType()`.
- Finally, the container builds the interceptor and decorator stacks for the instance as defined in Section 7.2.10, “Interceptor stack creation” and Section 7.3.8, “Decorator stack creation” and binds them to the instance.

When the EJB container destroys an instance of an EJB bean, the container destroys all dependent objects, after the the `@PreDestroy` callback completes.

6.10. Lifecycle of Servlets

The Servlet container creates instances of Servlets. The container must perform dependency injection upon any Servlet that executes in the context of a Web Beans application.

When the Servlet container creates a new instance of a Servlet, the container performs the following steps.

- First, the container initializes the values of all injected fields. For each injected field, the container sets the value to the object returned by `Manager.getInstanceByType()`.

- Next, the container calls all initializer methods. For each initializer method parameter, the container passes the object returned by `Manager.getInstanceByType()`.

When the Servlet container destroys a Servlet, the container destroys all dependent objects.

In a Java EE 5 environment, the container is not required to support injected fields or initializer methods of Servlets.

Chapter 7. Interceptors and decorators

Web Beans support interception as defined by the package `javax.interceptor`. Interceptors may be bound to a simple Web Bean, enterprise Web Bean or EJB session or message driven bean using the `javax.interceptor.Interceptors` annotation, or by using a Web Beans *interceptor binding*.

Interceptors are usually used to implement *cross-cutting* concerns, functionality that is orthogonal to the type system. In addition, Web Beans provides support for *decorators*. A decorator intercepts method invocations for a specific API type. Unlike interceptors, decorators are typesafe, and cannot be used to implement cross-cutting concerns.

Producers and JMS endpoints may not declare interceptors or decorators.

7.1. Business methods

Method interception by interceptors and decorators applies to *business method invocations* of a simple Web Bean, enterprise Web Bean or EJB bean.

For a simple Web Bean, a method invocation is considered a business method invocation if:

- the method was invoked upon an object obtained by calling `Manager.getInstance()`, passing the `Bean` object representing the simple Web Bean (this includes any instance of the Web Bean injected by the container), and
- the method is non-private and non-static.

Invocations of initializer methods by the container during Web Bean creation are not considered to be business method invocations.

Invocations of `@PreDestroy` and `@PostConstruct` callbacks by the container are not considered to be business method invocations.

All invocations of producer methods, disposal methods and observer methods *are* considered to be business method invocations.

Business method invocations of an enterprise Web Bean or EJB session or message driven bean are defined by the EJB specification.

Self-invocations of a simple Web Bean are considered to be business method invocations. However, self-invocations of an enterprise Web Bean or EJB session or message driven bean are not considered to be business method invocations.

7.2. Interceptors

An interceptor may be a method interceptor, a lifecycle callback interceptor, or both.

7.2.1. Business method interceptors

An interceptor method for business method invocations is a method of an interceptor with return type `Object` and a single parameter of type `javax.interceptor.InvocationContext`, annotated `@AroundInvoke`.

Interceptor methods for business method invocations are called by the container when a business method is invoked.

If an interceptor has an interceptor method for business method invocations, we describe it as a *business method interceptor*.

7.2.2. Lifecycle callback interceptors

An interceptor method for a lifecycle callback is a method of a Web Beans interceptor implementation class with return type `void` and a single parameter of type `javax.interceptor.InvocationContext`, annotated `@PostConstruct`, `@PreDestroy`, `@PrePassivate` or `@PostActivate`.

Interceptor methods for a lifecycle callbacks are called by the container when the corresponding `@PostConstruct`,

`@PreDestroy`, `@PrePassivate` or `@PostActivate` events occur.

If an interceptor has an interceptor method for a lifecycle callback, we describe it as a *lifecycle callback interceptor*.

7.2.3. Support for `@Interceptors`

Any Web Bean implementation class may declare interceptors using `@Interceptors`. The semantics are fully defined by the EJB specification.

7.2.4. Interceptor bindings

As an extension to the functionality defined by the `javax.interceptor` package, Web Beans provides an alternative method of binding interceptors to simple Web Beans, enterprise Web Beans and EJB session and message driven beans.

Even when interceptors are bound using this mechanism, the interception semantics are defined by the EJB specification.

An *interceptor binding type* is a Java annotation defined as `@Target({TYPE, METHOD})` or `@Target(TYPE)` and `@Retention(RUNTIME)`.

An interceptor binding type may be declared by specifying the `@InterceptorBindingType` meta-annotation.

```
@InterceptorBindingType
@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface Transactional {}
```

Alternatively, the `@InterceptorBindingType` meta-annotation may be omitted, and the interceptor binding type may be declared in `web-beans.xml`.

```
<myfwk:Transactional>
    <InterceptorBindingType/>
</myfwk:Transactional>
```

Multiple interceptors may be bound to the same interceptor binding type or types.

7.2.4.1. Interceptor binding types with additional interceptor bindings

An interceptor binding type may declare other interceptor bindings.

```
@InterceptorBindingType
@Target({TYPE, METHOD})
@Retention(RUNTIME)
@Transactional
public @interface DataAccess {}
```

```
<myfwk:DataAccess>
    <InterceptorBindingType/>
    <myfwk:Transactional/>
</myfwk:DataAccess>
```

Interceptor bindings are transitive—an interceptor binding declared by an interceptor binding type is inherited by all Web Beans and other interceptor binding types that declare that interceptor binding type.

Interceptor binding types declared `@Target(TYPE)` may not be applied to interceptor binding types declared `@Target({TYPE, METHOD})`.

7.2.4.2. Interceptor bindings for stereotypes

Interceptor bindings may be applied to a stereotype by annotating the stereotype annotation:

```
@Transactional
@Secure
@Production
@RequestScoped
@Stereotype
@Target(TYPE)
@Retention(RUNTIME)
public @interface Action {}
```

An interceptor binding declared by a stereotype are inherited by any Web Bean that declares that stereotype.

If a stereotype declares interceptor bindings, it must be defined as `@Target(TYPE)`.

7.2.5. Web Beans interceptors

A *Web Beans interceptor* is a simple Web Bean with an implementation class that is also an interceptor class as defined by the EJB specification. Web Beans interceptors must declare at least one interceptor binding type.

A Web Beans interceptor may be either a business method interceptor, a lifecycle callback interceptor or both.

Web Beans lifecycle callback interceptors may only declare interceptor binding types that are defined as `@Target(TYPE)`. If a lifecycle callback interceptor declares an interceptor binding type that is defined `@Target({TYPE, METHOD})`, a `DefinitionException` is thrown by the container at deployment time.

If a Web Beans interceptor does not declare any interceptor binding type, a `DefinitionException` is thrown by the container at deployment time.

Open issue: do we need to support defining interceptor methods in XML?

Open issue: should we support injection into interceptor methods?

7.2.5.1. Declaring a Web Beans interceptor using annotations

A Web Beans interceptor may be declared by annotating the interceptor implementation class with the `@Interceptor` stereotype, along with at least one interceptor binding type.

```
@Transactional @Interceptor
public class TransactionInterceptor {

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

}
```

7.2.5.2. Declaring a Web Beans interceptor using XML

Additional Web Beans interceptors may be declared in `web-beans.xml`, using the interceptor implementation class name and the `<Interceptor>` element:

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

When an interceptor is declared in XML, the container ignores any interceptor binding annotations applied to the interceptor class.

If the interceptor implementation class is already annotated `@Interceptor`, two different Web Beans interceptors exist, with different interceptor binding types.

7.2.6. Binding a Web Beans interceptor to a Web Bean or EJB bean

A Web Beans lifecycle callback interceptor may be bound to any simple Web Bean that is not an interceptor or decorator, any enterprise Web Bean or any EJB session or message-driven bean by declaring, at the class level, the same interceptor binding types that were declared by the interceptor.

A Web Beans business method interceptor may be bound to all non-static, non-private, non-final methods of a simple Web Bean that is not an interceptor or decorator or to all business methods of an enterprise Web Bean or EJB session or message-driven bean by declaring the same interceptor binding types, at the class level, that were declared by the interceptor.

A Web Beans business method interceptor may be bound to a non-static, non-private, non-final method of a simple Web Bean that is not an interceptor or decorator or to a business method of an enterprise Web Bean or EJB session or message-driven bean by declaring the same interceptor binding types, at the method level, that were declared by the interceptor.

If a simple Web Bean implementation class that is not an interceptor or decorator is declared final, or has any non-static, non-private, final methods, and also declares an interceptor binding type or a stereotype with interceptor bindings, a `DefinitionException` is thrown by the container at deployment time.

If a non-static, non-private method of a simple Web Bean implementation class is declared final and also declares an interceptor binding type, an `DefinitionException` is thrown by the container at deployment time.

7.2.6.1. Binding a Web Beans interceptor using annotations

Interceptor binding types may be declared by annotating the implementation class of a simple Web Bean, enterprise Web Bean, or by annotating the bean class of an EJB session or message-driven bean.

In the following example, the `TransactionInterceptor` will be applied at the class level, and therefore applies to all business methods of the class:

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

In this example, the `TransactionInterceptor` will be applied at the method level:

```
public class ShoppingCart {

   @Transactional
   public void placeOrder() { ... }

}
```

Web Beans interceptors may be enabled or disabled at deployment time. Disabled interceptors are never called at runtime.

7.2.6.2. Binding a Web Beans interceptor using XML

Class-level or method-level interceptor binding types may be applied to any Web Bean declared in `web-beans.xml`.

In the following example, the `TransactionInterceptor` will be applied at the class level:

```
<myapp:ShoppingCart>
   <myfwk:Transactional/>
</myapp:ShoppingCart>
```

In this example, the `TransactionInterceptor` will be applied at the method level:

```
<myapp:ShoppingCart>
   <myapp:placeOrder>
       <myfwk:Transactional/>
   </myapp:placeOrder>
</myapp:ShoppingCart>
```

7.2.7. Interceptor enablement and ordering

By default, interceptors bound via interceptor binding types are not enabled. An interceptor must be explicitly enabled by listing its implementation class under the `<Interceptors>` element in `web-beans.xml`.

```
<Interceptors>
    <myfwk:TransactionInterceptor/>
    <myfwk:LoggingInterceptor/>
</Interceptors>
```

The order of the interceptor declarations determines the interceptor ordering. Interceptors which occur earlier in the list are called first.

If a class listed under the `<Interceptors>` element is not the implementation class of at least one interceptor, a `DeploymentException` is thrown by the container at deployment time.

If the implementation class of an interceptor with a disabled deployment type is listed under the `<Interceptors>` element, a `DeploymentException` is thrown by the container at deployment time.

If the `<Interceptors>` element is specified in more than one `web-beans.xml` document, a `DeploymentException` is thrown by the container at deployment time.

Interceptors declared using `@Interceptors` or in `ejb-jar.xml` are called before interceptors declared using Web Beans interceptor bindings.

Interceptors are called before decorators.

7.2.8. The `Interceptor` object for an interceptor

The `Bean` object for an interceptor must extend the abstract class `Interceptor`.

```
public abstract class Interceptor extends Bean<Object> {

    protected Interceptor(Manager manager) {
        super(manager);
    }

    public abstract Set<Annotation> getInterceptorBindingTypes();

    public abstract Method getMethod(InterceptionType type);

}
```

An `InterceptionType` identifies the kind of lifecycle callback or business method.

```
public enum InterceptionType {
    AROUND_INVOKE, POST_CONSTRUCT, PRE_DESTROY, PRE_PASSIVATE, POST_ACTIVATE
}
```

The `getMethod()` method returns the interceptor method for the specified kind of lifecycle callback or business method. The `getMethod()` method must return a null value if the interceptor does not intercepts callbacks or business methods of the given type.

7.2.9. Interceptor resolution

The following method returns the ordered list of enabled interceptors for a set of interceptor binding types.

```
public interface Manager {

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

    ...

}
```

If two instances of the same interceptor binding type are passed to `resolveInterceptors()`, a `DuplicateBindingTypeException` is thrown.

If no interceptor binding type instance is passed to `resolveInterceptors()`, an `IllegalArgumentException` is thrown.

If an instance of an annotation that is not an interceptor binding type is passed to `resolveInterceptors()`, an `IllegalArgumentException` is thrown.

The following algorithm must be used by the container when resolving interceptors:

- First, the container identifies the set of *matching* enabled interceptors where for each declared interceptor binding annotation, there exists an interceptor binding annotation in the set of given binding annotations or, recursively, meta-annotations of those annotations, with (a) the same type and (b) the same annotation member value for each member which is not annotated `@NonBinding` (see Section 7.2.9.2, “Interceptor binding types with members”).
- Next, the container narrows the set of matching interceptors according to whether the interceptor intercepts the given kind of lifecycle callback or business method.
- Next, the container orders the matching interceptors according to the interceptor ordering specified in Section 7.2.7, “Interceptor enablement and ordering” and returns the resulting list of interceptors. If no matching interceptors exist in

the set, an empty list is returned.

7.2.9.1. Interceptors with multiple binding types

An interceptor class may specify multiple interceptor binding types, in which case the interceptor will be applied only to Web Beans and EJB beans with an implementation class that also declares all the binding types, and to methods of Web Beans and EJB beans where all the binding types appear on either the method or implementation class.

Consider the following interceptor:

```
@Transactional @Secure @Interceptor
public class TransactionalSecurityInterceptor {

   @AroundInvoke
   public void aroundInvoke() { ... }

}
```

This interceptor will be bound to all methods of this Web Bean:

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

The interceptor will also be bound to the `placeOrder()` method of this Web Bean:

```
@Transactional
public class ShoppingCart {

   @Secure
   public void placeOrder() { ... }

}
```

However, it will not be bound to the `placeOrder()` method of this Web Bean, since the `@Secure` interceptor binding type does not appear:

```
@Transactional
public class ShoppingCart {

   public void placeOrder() { ... }

}
```

7.2.9.2. Interceptor binding types with members

According to the interceptor resolution algorithm defined above, interceptor binding types may have annotation members.

This interceptor binding type declares a member:

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

Any interceptor with that interceptor binding type must select a member value:

```
@Transactional(requiresNew=true) @Interceptor
public class RequiresNewTransactionInterceptor {

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

}
```

The `RequiresNewTransactionInterceptor` applies to this Web Bean:

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

But not to this Web Bean:

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

Annotation member values are compared using `equals()`.

An annotation member may be excluded from consideration using the `@NonBinding` annotation.

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

Array-valued or annotation-valued members of an interceptor binding annotation must be annotated `@NonBinding`. If an array-valued or annotation-valued member of an interceptor binding annotation is not annotated `@NonBinding`, a `DefinitionException` is thrown by the container at deployment time.

7.2.10. Interceptor stack creation

When a simple or enterprise Web Bean is created, the container must:

- Identify the interceptors for each lifecycle callback and business method by calling `Manager.resolveInterceptors()` passing the interceptor bindings for the callback or business method, including all interceptor bindings defined at the class level, method level and by stereotypes.
- Identify the interceptors defined using the `@Interceptors` annotation for each lifecycle callback and business method.
- For each unique interceptor, call `Manager.getInstance()`, passing the `Interceptor` object, to obtain an instance of the interceptor. For a given interceptor and a given Web Bean instance, the container must call `Manager.getInstance()` exactly once.
- For each lifecycle callback and business method build an ordered list of returned interceptor instances.

The resulting ordered lists of interceptor instances are called *interceptor stacks*.

7.2.11. Interceptor invocation

Whenever a business method or lifecycle callback is invoked on an instance of a simple Web Bean, enterprise Web Bean or EJB session or message driven bean, the container intercepts the method invocation and invokes interceptors of the callback or business method.

The container identifies the first interceptor in the interceptor stack for the method. If no such interceptor exists, the container starts processing the decorator stack, as defined in Section 7.3.9, “Decorator invocation”. Otherwise, the container builds an instance of `javax.interceptor.InvocationContext` and calls the appropriate interceptor method of the interceptor.

When any interceptor is invoked by the container, it may in turn call `InvocationContext.proceed()`. The container then identifies the first interceptor in the interceptor stack for the method such that the interceptor has not previously been invoked during this business method or lifecycle callback invocation. If no such interceptor exists, the container starts processing the decorator stack. Otherwise, the container calls the appropriate interceptor method.

Eventually, by recursion, the interceptor stack is exhausted of uninvoked interceptors.

7.3. Decorators

A *decorator* implements one or more API types and intercepts business method invocations for methods defined by the implemented API types. These API types are called *decorated types*.

A decorator is a simple Web Bean. The set of decorated types of a decorator includes all interfaces implemented directly or indirectly by the implementation class, except for `java.io.Serializable`. The decorator implementation class and its

superclasses are not decorated types of the decorator. The decorator class may be abstract.

Alternative definition: the set of decorated types includes all interfaces implemented directly and indirectly by both the decorator implementation class and the declared type of the delegate attribute.

Decorators may be bound to any enterprise Web Bean, any simple Web Bean that implements an interface and is not an interceptor or decorator, or to any EJB bean. Decorators are called by the container according to the semantics defined in Section 7.3.9, “Decorator invocation”.

7.3.1. Declaring a decorator using annotations

A decorator is declared by annotating the implementation class with the `@Decorator` stereotype.

```
@Decorator
class TimestampLogger implements Logger { ... }
```

7.3.2. Declaring a decorator using XML

Additional decorators may be declared in `web-beans.xml`, using the decorator implementations class name and the `<Decorator>` element:

```
<myfwk:TimestampLogger>
    <Decorator/>
    ...
</myfwk:TimestampLogger>
```

If the decorator implementation class is already annotated `@Decorator`, two different decorators exist.

7.3.3. Decorator delegate attributes

All decorators have a *delegate attribute*.

A delegate attribute is a non-static, non-final field of a decorator implementation class.

The delegate attribute may be declared using the `@Decorates` annotation or `<Decorates>` element:

```
@Decorator
class TimestampLogger implements Logger {
    @Decorates Logger logger;
    ...
}
```

```
<myfwk:TimestampLogger>
    <Decorator/>
    <myfwk:logger>
        <Decorates>
            <myfwk:Logger/>
        </Decorates>
    </myfwk:logger>
</myfwk:TimestampLogger>
```

In this case, the decorator is bound to any Web Bean or EJB bean that has the type of the delegate attribute as an API type.

The declared type of the delegate attribute must be a Java interface type. If the declared type of a delegate attribute is not a Java interface type, a `DefinitionException` is thrown by the container at deployment time.

The delegate may optionally declare one or more binding types:

```
@Decorator
class TimestampLogger implements Logger {
    @Decorates @Debug Logger logger;
    ...
}
```

```
<myfwk:TimestampLogger>
    <Decorator/>
    <myfwk:logger>
        <Decorates>
            <myfwk:Logger>
```

```
                <myfwk:Debug/>
            </myfwk:Logger>
        </Decorates>
    </myfwk:logger>
</myfwk:TimestampLogger>
```

In this case, the decorator is bound to any Web Bean or EJB bean that has the type of the delegate attribute as an API type, and declares all the binding types specified by the delegate attribute.

All delegate binding types must be explicitly declared. If no binding type is explicitly declared by the delegate attribute, the set of binding types is empty.

A decorator must have exactly one delegate attribute. If a decorator has more than one delegate attribute, or does not have a delegate attribute, a `DefinitionException` is thrown by the container at deployment time.

If a decorator applies to a simple Web Bean, and the Web Bean implementation class is declared final, a `DefinitionException` is thrown by the container at deployment time.

If a decorator applies to a simple Web Bean with a non-static, non-private, final method, and the decorator also implements that method, a `DefinitionException` is thrown by the container at deployment time.

7.3.4. Decorated types of a decorator

A decorator is not required to implement all of the API types of its delegate attribute. If a decorator does not implement an API type of the delegate attribute, that API will not be intercepted by the decorator.

A decorator may be an abstract Java class, and is not required to implement all methods of its API types. If a decorator does not implement a method of one of its API types, that method will not be intercepted by the decorator.

The declared type of the decorator delegate attribute must implement or extend all of the decorated types of the decorator. If a decorator delegate attribute does not implement or extend a decorated type of the decorator, a `DefinitionException` is thrown by the container at deployment time.

7.3.5. Decorator enablement and ordering

By default, decorators are not enabled. A decorator must be explicitly enabled by listing its implementation class under the `<Decorators>` element in `web-beans.xml`.

```
<Decorators>
    <myfwk:TimestampLogger/>
    <myfwk:IdentityLogger/>
</Decorators>
```

The order of the decorator declarations determines the decorator ordering. Decorators which occur earlier in the list are called first.

If a class listed under the `<Decorators>` element is not the implementation class of at least one decorator, a `DeploymentException` is thrown by the container at deployment time.

If the implementation class of a decorator with a disabled deployment type is listed under the `<Decorators>` element, a `DeploymentException` is thrown by the container at deployment time.

If the `<Decorators>` element is specified in more than one `web-beans.xml` document, a `DeploymentException` is thrown by the container at deployment time.

Decorators are called after interceptors.

Would it be better to unify interceptors and decorators into a single stack, so that they can be interleaved?

7.3.6. The `Decorator` object for a decorator

The `Bean` object for an interceptor must extend the abstract class `Decorator`.

```
public abstract class Decorator extends Bean<Object> {
```

```
        protected Decorator(Manager manager) {
                super(manager);
        }

        public abstract Class<?> getDelegateType();

        public abstract Set<Annotation> getDelegateBindingTypes();

        public abstract void setDelegate(Object instance, Object delegate);

}
```

7.3.7. Decorator resolution

The following method returns the ordered list of enabled decorators for a set of API types and a set of binding types.

```
public interface Manager {

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

    ...

}
```

The first argument is the set of API types of the decorated Web Bean. The annotations are binding types declared by the decorated Web Bean.

If two instances of the same binding type are passed to `resolveDecorators()`, a `DuplicateBindingTypeException` is thrown.

If an instance of an annotation that is not a binding type is passed to `resolveDecorators()`, an `IllegalArgumentException` is thrown.

If the set of API types is empty, an `IllegalArgumentException` is thrown.

The following algorithm must be used by the container when resolving decorators:

- First, the container identifies the set of *matching* enabled decorators where the declared type of the delegate attribute is one of the given API types. For this purpose, primitive types are considered to be identical to their corresponding wrapper types in `java.lang`, array types are considered identical only if their element types are identical and parameterized types are considered identical only if both the type and all type parameters are identical.
- Next, the container considers the given binding annotations. If no binding annotations were passed to `resolveDecorators()`, the container assumes the binding annotation `@Current`. The container narrows the set of matching decorators to just those where for each binding annotation declared by the decorator delegate attribute, there is a given binding annotation with (a) the same type and (b) the same annotation member value for each member which is not annotated `@NonBinding` (see Section 5.10.1.1, “Binding annotations with members”).
- Next, the container orders the matching decorators according to the decorator ordering specified in Section 7.3.5, “Decorator enablement and ordering” and returns the resulting list of decorators. If no matching decorators exist in the set, an empty list is returned..

7.3.8. Decorator stack creation

When a simple or enterprise Web Bean is created, the container must:

- Identify the decorators for the Web Bean by calling `Manager.resolveDecorators()` passing the API types and binding types of the Web Bean.
- For each decorator, call `Manager.getInstance()`, passing the `Decorator` object, to obtain an instance of the decorator.
- For each returned decorator instance, call `Decorator.setDelegate()` to inject an object that implements the declared type of the delegate attribute to the delegate attribute of the decorator instance.
- Build an ordered list of the decorator instances.

The resulting ordered list of decorator instances is called the *decorator stack*.

7.3.9. Decorator invocation

Whenever a business method is invoked on an instance of a simple Web Bean, enterprise Web Bean or EJB session or message driven bean, the container intercepts the business method invocation and, after processing the interceptor stack, as defined in Section 7.2.11, “Interceptor invocation”, invokes decorators of the Web Bean.

The container searches for the first decorator in the decorator stack for the instance that implements the method that is being invoked as a business method. If no such decorator exists, the container invokes the business method of the Web Bean instance. Otherwise, the container calls the method of the decorator.

When any decorator is invoked by the container, it may in turn invoke a method of the delegate attribute. The container intercepts the delegate invocation and searches for the first decorator in the decorator stack for the instance such that:

- the decorator implements the method that is being invoked upon the delegate, and
- the decorator has not previously been invoked during this business method invocation.

If no such decorator exists, the container invokes the business method of the Web Bean instance. Otherwise, the container calls the method of the decorator.

Eventually, by recursion, the decorator stack is exhausted of uninvoked decorators.

Chapter 8. Events

Web Beans may produce and consume events. This facility allows Web Beans to interact in a completely decoupled fashion, with no compile-time dependency between the two Web Beans.

An event comprises:

- A Java object (the *event object*)
- A (possibly empty) set of instances of binding annotation types (the *event bindings*)

The event object acts as a payload, to propagate state from producer to consumer. The event bindings act as topic selectors, allowing the consumer to narrow to set of events it observes.

An event consumer observes events of a specific type, the *observed event type*, with a specific set of instances of event binding types, the *observed event bindings*.

8.1. Event types and binding types

An event object is an instance of a concrete Java class with no type variables or wildcards. The *event types* of the event include all superclasses and interfaces of the class of the event object.

An event binding type is just an ordinary binding type as specified in Section 2.3.2, “Defining new binding types” with the exception that it may be declared `@Target({FIELD, PARAMETER})`.

More formally, an event binding type is a Java annotation defined as `@Target({FIELD, PARAMETER})` or `@Target({METHOD, FIELD, PARAMETER, TYPE})` and `@Retention(RUNTIME)`. All event binding types must specify the `@BindingType` meta-annotation.

An event consumer will be notified of an event if the observed event type it specifies is one of the event types of the event, and if all the observed event bindings it specifies are event bindings of the event.

8.2. Firing an event via the `Manager` interface

The `Manager` interface provides a method for firing events:

```
public interface Manager {

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

    ...

}
```

The first argument is the event object:

```
public void login() {
    ...
    manager.fireEvent( new LoggedInEvent(user) );

}
```

If the type of the event object passed to `fireEvent()` contains type variables or wildcards, an `IllegalArgumentException` is thrown.

The remaining arguments are the event bindings, optional instances of event binding types:

```
public void login() {
    User user = ...;
    manager.fireEvent( user, new LoggedInBinding() {} );

}
```

where `LoggedInBinding` is an implementation of the event binding type `LoggedIn`:

```
public class LoggedInBinding
        extends AnnotationLiteral<LoggedIn>
        implements LoggedIn {}
```

8.3. Observing events via the `Observer` interface

An *observer* consumes events and allows the application to react to events that occur.

Observers of Web Beans events implement the `Observer` interface.

```
public interface Observer<T> {

    public void notify(T event);

}
```

An observer instance may be registered with the container by calling `Manager.addObserver()`:

```
public interface Manager {

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

    ...

}
```

The first parameter is the observer object. The second parameter is the observed event type. The remaining parameters are optional observed event binding types. The observer is notified when an event object that is assignable to the observed event type is raised with the observed event binding types.

An observer instance may be deregistered by calling `Manager.removeObserver()`:

```
public interface Manager {

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

    ...

}
```

If the observed event type passed to `addObserver()` or `removeObserver()` contains type variables or wildcards, an `IllegalArgumentException` is thrown.

If two instances of the same binding type are passed to `addObserver()` or `removeObserver()`, a `DuplicateBindingTypeException` is thrown.

If an instance of an annotation that is not a binding type is passed to `addObserver()` or `removeObserver()`, an `IllegalArgumentException` is thrown.

8.4. Observer invocation

When an event is fired by the application, the container must:

- determine the observers for that event by calling `Manager.resolveObservers()`, passing the event object and all event binding type instances, then,
- for each observer, call the `notify()` method of the `Observer` interface, passing the event object.

Observers may throw exceptions. If an observer throws an exception, the exception aborts processing of the event. No other observers of that event will be called. The `fireEvent()` method rethrows the exception.

Any observer called before completion of a transaction may call `setRollbackOnly()` to force a transaction rollback. An observer may not directly initiate, commit or rollback JTA transactions.

8.5. Observer methods

An *observer method* is an observer defined via annotations, instead of by explicitly implementing the `Observer` interface.

Unlike regular observers, observer methods are automatically discovered and registered by the container.

An observer method must be a method of a simple Web Bean implementation class or enterprise Web Bean implementation class. An observer method may be either static or non-static. If the Web Bean is an enterprise Web Bean, the observer method must be a business method of the EJB or a static method of the bean class.

There may be arbitrarily many observer methods with the same event parameter type and binding types.

A Web Bean may declare multiple observer methods.

8.5.1. Event parameter of an observer method

Each observer method must have exactly one *event parameter*, of the same type as the event type it observes. When searching for observer methods for an event, the container considers the type and binding types of the event parameter.

If the event parameter does not explicitly declare any binding type, the observer method observes events with no binding type.

If the type of the event parameter contains type variables or wildcards, a `DefinitionException` is thrown by the container at deployment time.

8.5.2. Declaring an observer method using annotations

A observer method may be declared using annotations by annotating a parameter `@Observes`. That parameter is the event parameter.

```
public void afterLogin(@Observes LoggedInEvent event) { ... }
```

If a method has more than one parameter annotated `@Observes`, a `DefinitionException` is thrown by the container at deployment time.

If an observer method is annotated `@Produces`, `@Initializer` or `@Destructor`, or has a parameter annotated `@Disposes`, a `DefinitionException` is thrown by the container at deployment time.

The event parameter may declare binding types:

```
public void afterLogin(@Observes @Admin LoggedInEvent event) { ... }
```

8.5.3. Declaring an observer method using XML

For a Web Beans defined in XML, an observer method may be declared using the method name, the `<Observes>` element, and the parameter types of the method:

```
<myapp:afterLogin>
    <Observes>
        <myapp:LoggedInEvent/>
    </Observes>
</myapp:afterLogin>
```

```
<myapp:afterLogin>
    <Observes>
        <myapp:LoggedInEvent>
            <myapp:Admin/>
        </myapp:LoggedInEvent>
    </Observes>
</myapp:afterLogin>
```

When an observer method is declared in XML, the container ignores binding annotations applied to the Java method parameters.

If the implementation class of a Web Bean declared in XML does not have a method with parameters that match those declared in XML, a `NonexistentMethodException` is thrown by the container at deployment time.

8.5.4. Observer method parameters

In addition to the event parameter, observer methods may declare additional parameters, which may declare binding types. The container calls the method `Manager.getInstanceByType()` defined in Section 5.10, “Instance resolution” to determine a value for each parameter of an observer method and calls the observer method with those parameter values.

```
public void afterLogin(@Observes LoggedInEvent event, @Manager User user, @Logger Log log) { ... }
```

```
public void afterAdminLogin(@Observes @Admin LoggedInEvent event, @Logger Log log) { ... }
```

```
<myapp:afterLogin>

    <Observes>
        <myapp:LoggedInEvent/>
    </Observes>

    <myapp:User>
        <myapp:Manager/>
    </myapp:User>

    <myfwk:Log>
        <myfwk:Logger/>
    </myfwk:Log>

</myapp:afterLogin>
```

```
<myapp:afterAdminLogin>

    <Observes>
        <myapp:LoggedInEvent>
            <myapp:Admin/>
        </myapp:LoggedInEvent>
    </Observes>

    <myfwk:Log>
        <myfwk:Logger/>
    </myfwk:Log>

</myapp:afterAdminLogin>
```

8.5.5. Conditional observers

Conditional observers are observer methods which are notified of an event only if an instance of the Web Bean that defines the observer method already exists in the current context.

A conditional observers may be declared by annotating the event parameter with the `@IfExists` annotation.

```
public void refreshOnDocumentUpdate(@IfExists @Observes @Updated Document doc) { ... }
```

Conditional observers may be declared in XML by adding a child `<IfExists>` element to the `<Observes>` element.

```
<myapp:refreshOnDocumentUpdate>
    <Observes>
        <IfExists/>
        <myapp:Document>
            <myapp:Updated/>
        </myapp:Document>
    </Observes>
</myapp:refreshOnDocumentUpdate>
```

8.5.6. Transactional observers

Transactional observers are observer methods which receive event notifications during the before or after completion

phase of the transaction in which the event was fired. If no transaction is in progress when the event is fired, they are notified at the same time as other observers.

Transactional observers may be declared by annotating the event parameter of the observer method.

- The `@AfterTransactionCompletion` annotation specifies that an observer method should be called during the after completion phase.
- The `@AfterTransactionSuccess` annotation specifies that an observer method should be called during the after completion phase, only when the transaction completes successfully.
- The `@AfterTransactionFailure` annotation specifies that an observer method should be called during the after completion phase, only when the transaction fails.
- The `@BeforeTransactionCompletion` annotation specifies that an observer method should be called during the before completion phase.

```
void onDocumentUpdate(@Observes @AfterTransactionSuccess @Updated Document doc) { ... }
```

Transactional observers may be declared in XML by a child element of the `<Observes>` element.

- The `<AfterTransactionCompletion>` element specifies that the observer method should be called during the after completion phase.
- The `<AfterTransactionSuccess>` element specifies that the observer method should be called during the after completion phase, only when the transaction completes successfully.
- The `<AfterTransactionFailure>` element specifies that the observer method should be called during the after completion phase, only when the transaction fails.
- The `<BeforeTransactionCompletion>` element specifies that the observer method should be called during the before completion phase.

```
<myapp:onDocumentUpdate>
    <Observes>
        <AfterTransactionSuccess/>
        <myapp:Document>
            <myapp:Updated/>
        </myapp:Document>
    </Observes>
</myapp:onDocumentUpdate>
```

8.5.7. Observer object for an observer method

For every observer method of an enabled Web Bean, the container is responsible for providing and registering an appropriate implementation of the `Observer` interface, that delegates event notifications to the observer method.

The `notify()` method of the `Observer` implementation for an observer method either invokes the observer method immediately, or registers the observer method for later invocation during the transaction completion phase, via a JTA `Synchronization` object.

- If the observer is a transactional observer and there is currently a JTA transaction in progress, the observer object calls the observer method during the appropriate transaction completion phase. At the appropriate point during the completion phase of the transaction, the container invokes the observer method. If the observer is a method of an EJB bean, the method is called with the same client invocation context as the call to the JTA `Synchronization` interface.
- Otherwise, the container calls the observer immediately. If the observer is a method of an EJB bean, the method is called in the client invocation context of the code that called `Event.fire()`.

To invoke an observer method, the container must pass the event object to the event parameter and the object returned by `Manager.getInstanceByType()` to each of the other parameters.

- If the observer method is static, the container must invoke the method.

- Otherwise, if the observer method is non-static, the container must:
  - obtain the `Bean` object for the most specialized Web Bean that specializes the Web Bean which declares the observer method, and then
  - obtain the context object by calling `Manager.getContext()`, passing the Web Bean scope, then
  - obtain an instance of the Web Bean by calling `Context.get()`, passing the `Bean` instance representing the Web Bean and `false` if this observer method is a conditional observer or `true` otherwise as the value of the `create` parameter, and then
  - if the `get()` method returned a non-null value, invoke the observer method on the returned instance

Observer methods may throw exceptions:

- If the observer is a transactional observer, any exception is caught and logged by the container.
- Otherwise, the exception is rethrown by the `notify()` method of the observer object. If the exception is a checked exception, it is wrapped and rethrown as an (unchecked) `ObserverException`.

The observer object is registered by calling `Manager.addObserver()`, passing the event parameter type as the observed event type, and the binding types of the event parameter as the observed event binding types.

8.6. The `Event` interface

Alternatively, an instance of the `Event` interface may be injected via use of the `@Fires` binding annotation:

```
@Fires Event<LoggedInEvent> loggedInEvent;
```

Additional binding annotations may be specified at the injection point:

```
@Fires @Admin Event<LoggedInEvent> loggedInEvent;
```

The `Event` interface provides a method for firing events of a specific type, and a method for registering observers for events of the same type:

```
public interface Event<T> {

    public void fire(T event, Annotation... bindings);

    public void observe(Observer<T> observer, Annotation... bindings);

}
```

The first parameter of `fire()` is the event object. The remaining parameters are event binding types.

The first parameter of `observe()` is the observer object. The remaining parameters are the observed event binding types.

If two instances of the same binding type are passed to `fire()` or `observes()`, a `DuplicateBindingTypeException` is thrown.

If an instance of an annotation that is not a binding type is passed to `fire()` or `observes()`, an `IllegalArgumentException` is thrown.

The `@Fires` annotation or `<Fires>` element may be applied to any field of a Web Bean implementation class or to any parameter of a producer method, initializer method, disposal method or Web Bean constructor where the type of the field or parameter is `Event`, and an actual type parameter is specified.

If the type of the injection point is not of type `Event`, if no actual type parameter is specified, or if the type parameter contains a type variable or wildcard, a `DefinitionException` is thrown by the container at deployment time.

Whenever the `@Fires` annotation appears at an injection point, an implicit Web Beans exists with:

- exactly the API type and binding annotations that appear at the injection point,

- deployment type `@Standard`,
- `@Dependent` scope,
- no Web Bean name, and
- an implementation provided automatically by the container.

The `fire()` method of the provided implementation of `Event` must call `Manager.fireEvent()`, passing the following parameters:

- the event object passed to `Event.fire()`
- all binding annotations declared at the injection point, except `@Fires`
- all binding annotation instances passed to `Event.fire()`

The application may fire events by calling the `fire()` method:

```
@Fires @LoggedIn Event<User> loggedInEvent;
...
if ( user.isAdmin() ) {
    loggedInEvent.fire( user, new AdminBinding() {} );
}
else {
    loggedInEvent.fire(user);
}
```

In this example, an event of type `User`, with binding types `@LoggedIn` and, sometimes, `@Admin` occurs.

The `observe()` method of the provided implementation of `Event` must call `Manager.addObserver()`, passing the following parameters:

- the observer object passed to `Event.observe()`
- all binding annotations declared at the injection point, except `@Fires`
- all binding annotation instances passed to `Event.observe()`

The application may register observers by calling the `observe()` method:

```
@Fires @LoggedIn Event<User> loggedInEvent;
...
loggedInEvent.observe( new Observer<User>() { public void notify(User user) { ... } } );
```

8.7. Observer resolution

The method `Manager.resolveObservers()` resolves observers for an event:

```
public interface Manager {

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

      ...

}
```

The first parameter of `resolveObservers()` is the event object. The remaining parameters are event binding types.

If the type of the event object passed to `resolveObservers()` contains type variables or wildcards, an `IllegalArgumentException` is thrown.

If two instances of the same binding type are passed to `resolveObservers()`, a `DuplicateBindingTypeException` is thrown.

If an instance of an annotation that is not a binding type is passed to `resolveObservers()`, an `IllegalArgumentException`

is thrown.

When searching for observers for an event, the container searches for observers which satisfy the following rules:

- the event object must be assignable to the observed event type, taking type parameters into consideration, and
- for each observed event binding type, (a) an instance of the binding type must have been passed to `fireEvent()` and (b) any member values of the binding type must match the member values of the instance passed to `fireEvent()`.

8.7.1. Event binding annotations with members

As usual, the binding type may have annotation members:

```
@EventBindingType
@Target(PARAMETER)
@Retention(RUNTIME)
public @interface Role {
    String value();
}
```

Consider the following event:

```
public void login() {
    final User user = ...;
    manager.fireEvent( new LoggedInEvent(user),
            new RoleBinding() { public String value() { return user.getRole(); } );
}
```

Where `RoleBinding` is an implementation of the binding type `Role`:

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

Then the following observer method will always be notified of the event:

```
public void afterLogin(@Observes LoggedInEvent event) { ... }
```

Whereas this observer method may or may not be notified, depending upon the value of `user.getRole()`:

```
public void afterAdminLogin(@Observes @Role("admin") LoggedInEvent event) { ... }
```

As usual, the container uses `equals()` to compare event binding type member values.

8.7.2. Multiple event binding annotations

An event parameter may have multiple binding annotations:

```
public void afterDocumentUpdatedByAdmin(@Observes @Updated @ByAdmin Document doc) { ... }
```

Then this observer method will only be notified if all the event binding types are specified when the event is fired:

```
manager.fireEvent( document, new UpdatedBinding() {}, new ByAdminBinding() {} );
```

Other, less specific, observers will also be notified of this event:

```
public void afterDocumentUpdated(@Observes @Updated Document doc) { ... }
```

```
public void afterDocumentEvent(@Observes Document doc) { ... }
```

Chapter 9. Scopes and contexts

Associated with every Web Beans scope type is a *context object*. The context object determines the lifecycle and visibility of instances of all Web Beans with that scope. In particular, the context object defines:

- When a new instance of any Web Bean with that scope is created
- When an existing instance of any Web Bean with that scope is destroyed
- Which injected references refer to any instance of a Web Bean with that scope

Each context object is represented by an instance of the `Context` interface. Objects that are associated with a context are created and destroyed by the context object via an implementation of the `Contextual` interface.

9.1. The `Contextual` interface

The `Contextual` interface defines operations to create and destroy contextual objects of a certain type:

```
public interface Contextual<T> {
    public abstract T create();
    public abstract void destroy(T instance);
}
```

Any implementation of `Contextual` is called a *contextual type*. In particular, the `Bean` abstract class defined in Section 3.11, “The Bean object for a Web Bean” implements `Contextual`.

The container and third party frameworks may define implementations of the `Contextual` interface that do not extend `Bean`, but it is not recommended that applications directly implement `Contextual`.

9.2. The `Context` interface

The `Context` interface provides an operation for obtaining *contextual instances* with a particular scope of any contextual type.

```
public interface Context {

    public Class<? extends Annotation> getScopeType();

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

    boolean isActive();

}
```

The `Context` SPI is called by the container and may be called by third party frameworks. It should not be called directly by the application.

```
User instance = context.get(userBean, true);
```

The `get()` method may either:

- return an existing instance of the given contextual type, or
- if the value of the `create` parameter is `false`, return a null value, or
- if the value of the `create` parameter is `true`, create a new instance of the given contextual type by calling `Bean.create()` and return the new instance.

The `get()` method may not return a null value unless the `create` parameter is `false` or `Contextual.create()` returns a null value.

The `get()` method may not create a new instance of the given contextual type unless the `create` parameter is `true`.

The `Context` implementation is responsible for destroying any contextual instance it creates by passing the instance to the `destroy()` method of the `Contextual` object representing the contextual type. A destroyed instance must not subsequently be returned by the `get()` method.

At a particular point in the execution of the program a scope may be *inactive* with respect to the current thread. When a scope is inactive, any invocation of the `get()` from the current thread upon the `Context` object for that scope results in a `ContextNotActiveException`.

Otherwise, we say that the scope is *active*.

The `isActive()` method returns `false` when the scope of the context object is inactive, and `true` when it is active.

9.3. Normal scopes and pseudo-scopes

Most scopes are *normal scopes*. The context object for a normal scope type is a mapping from each enabled contextual type with that scope type to an instance of that contextual type. There may be no more than one mapped instance per contextual type per thread. The set of all mapped instances of contextual types with a certain scope for a certain thread is called the *context* for that scope associated with that thread.

A context may be associated with one or more threads. A context with a certain scope is said to *propagate* from one point in the execution of the program to another when the set of mapped instances of contextual types with that scope is preserved.

The context associated with the current thread is called the *current context* for the scope. The mapped instance of a contextual type associated with a current context is called the *current instance* of the contextual type.

The `get()` operation of the `Context` object for an active normal scope returns the current instance of the given contextual type.

At certain points in the execution of the program a context may be *destroyed*. When a context is destroyed, all mapped instances of contextual types with that scope type are destroyed by passing them to the `Contextual.destroy()` method.

Contexts with normal scopes must obey the following rule:

Suppose Web Beans `A`, `B` and `Z` all have normal scopes. Suppose `A` has an injection point `x`, and `B` has an injection point `y`. Suppose further that both `x` and `y` resolve to Web Bean `Z` according to the typesafe resolution algorithm. If `a` is the current instance of `A`, and `b` is the current instance of `B`, then both `a.x` and `b.y` refer to the same instance of `Z`. This instance is the current instance of `Z`.

Any scope that is not a normal scope is called a *pseudo-scope*. The concept of a current instance is not well-defined in the case of a pseudo-scope.

All pseudo-scopes must be explicitly declared `@ScopeType(normal=false)`, to indicate to the container that no client proxy is required.

All scopes defined by the Web Beans specification, except for the `@Dependent` pseudo-scope, are normal scopes.

9.4. Dependent pseudo-scope

The `@Dependent` scope type is a pseudo-scope. Web Beans declared with scope type `@Dependent` behave differently to Web Beans with other built-in scope types.

When a Web Bean is declared to have `@Dependent` scope:

- No injected instance of the Web Bean is ever shared between multiple injection points.
- Any injected instance of the Web Bean is bound to the lifecycle of the Web Bean, Servlet or EJB bean into which it is injected.
- Any instance of the Web Bean that is used to evaluate a Unified EL expression exists to service that evaluation only.
- Any instance of the Web Bean that receives a producer method, producer field, disposal method or observer method

invocation exists to service that invocation only.

Every invocation of the `get()` operation of the `Context` object for the `@Dependent` scope with the value `true` for the `create` parameter returns a new instance of the given Web Bean.

Every invocation of the `get()` operation of the `Context` object for the `@Dependent` scope with the value `false` for the `create` parameter returns a null value.

The `@Dependent` scope is inactive except:

- when an instance of a Web Bean with scope `@Dependent` is created by the container to receive a producer method, producer field, disposal method or observer method invocation, or
- while a Unified EL expression is evaluated, or
- while an observer method is invoked, or
- when the container is creating or destroying a Web Bean instance or injecting its dependencies, or
- when the container is injecting dependencies of an EJB bean or Servlet or when an EJB bean `@PostConstruct` or `@PreDestroy` callback is invoked by the EJB container.

9.4.1. Dependent objects

A Web Bean may create an instance of a Web Bean with scope type `@Dependent` by calling `Manager.getInstance()` from the Web Bean constructor, from initializer methods, from producer methods, from disposal methods, from `@PostConstruct` and `@PreDestroy` callbacks and from Web Beans interceptors or decorators for any of these methods.

An EJB bean may create an instance of a Web Bean with scope type `@Dependent` by calling `Manager.getInstance()` from initializer methods, from `@PostConstruct` and `@PreDestroy` callbacks and from Web Beans interceptors for these methods.

A Servlet may create an instance of a Web Bean with scope type `@Dependent` by calling `Manager.getInstance()` from initializer methods.

Alternatively, a Web Bean, EJB bean, or Servlet may obtain an instance of a Web Bean with scope type `@Dependent` via dependency injection.

In any of these cases, the instance of the Web Bean with scope type `@Dependent` is called a *dependent object*.

9.4.1.1. Dependent objects of a simple or enterprise Web Bean

A `@Dependent` scoped contextual instance is said to be a dependent object of a simple or enterprise Web Bean instance if:

- it was injected into any field, the Web Bean constructor, any observer method or any initializer method of the simple or enterprise Web Bean instance, or
- it was created by a direct call to `Manager.getInstance()` or `Context.get()` during invocation of the Web Bean constructor, any observer method, any initializer method or any `@PostConstruct` or `@PreDestroy` callback of the simple or enterprise Web Bean instance.

9.4.1.2. Dependent objects of a producer method

A `@Dependent` scoped contextual instance is said to be a dependent object of a producer method Web Bean instance if:

- it was injected into the producer method or disposal method call that produced or disposed the instance, or
- it was created by a direct call to `Manager.getInstance()` or `Context.get()` during invocation of the producer method or disposal method that produced or disposed the instance.

9.4.1.3. Dependent objects of an EJB bean or Servlet

A `@Dependent` scoped contextual instance is said to be a dependent object of an EJB bean or Servlet if:

- it was injected into any field or initializer method of the EJB bean or Servlet, or
- it was created by a direct call to `Manager.getInstance()` or `Context.get()` during invocation of any initializer method of the EJB bean or Servlet or during invocation of any `@PostConstruct` or `@PreDestroy` callback of the EJB bean.

9.4.2. Dependent object destruction

The container is responsible for destroying `@Dependent` scoped contextual instances by passing them to the `Contextual.destroy()` method.

The container must:

- destroy all dependent objects of a Web Bean instance when the instance is destroyed,
- destroy all dependent objects of an EJB bean or Servlet when the EJB bean or Servlet is destroyed,
- destroy all `@Dependent` scoped contextual instances created during an EL expression evaluation when the evaluation completes, and
- destroy any `@Dependent` scoped contextual instance created to receive a producer method, producer field, disposal method or observer method invocation when the invocation completes.

Finally, the container is permitted to destroy any `@Dependent` scoped contextual instance at any time if the instance is no longer referenced by the application (excluding weak, soft and phantom references).

9.5. Passivating scopes and serialization

A *passivating scope* requires that instances of Web Beans with that scope be serializable, so that their state may be stored to disk when the scope becomes inactive. The process of storing the state of Web Bean instances belonging to a scope that is about to become inactive to disk is called *context passivation*. Passivating scopes must be explicitly declared `@ScopeType(passivating=true)`.

For example, the built-in session and conversation scopes defined in Section 9.6, “Context management for built-in scopes” are passivating scopes.

The container must validate that every Web Bean declared with a passivating scope truly is serializable:

- EJB local objects are serializable. Therefore, an enterprise Web Bean may declare any passivating scope.
- Simple Web Beans are not required to be serializable. If a simple Web Bean declares a passivating scope, and the implementation class is not serializable, a `DefinitionException` is thrown by the container at deployment time.
- If a producer method or field declares a passivating scope and returns a non-serializable object at runtime, an `IllegalProductException` is thrown by the container.

The built-in session and conversation scopes are passivating. No other built-in scope is passivating.

A Web Bean instance may be serialized under one of two circumstances:

- the Web Bean declares a passivating scope type, and context passivation occurs, or
- the Web Bean is an EJB stateful session bean, and it is passivated by the EJB container.

In either case, any non-transient field that holds a reference to another Web Bean must be serialized along with the Web Bean that is being serialized. Therefore, the reference must be to a serializable type.

Web Beans client proxies are serializable. Therefore, any reference to a Web Bean which declares a normal scope type is serializable. On the other hand, dependent objects (including interceptors and decorators with scope `@Dependent`) of a stateful session bean or of a Web Bean with a passivating scope must be serialized and deserialized along with their owner:

- EJB local objects are serializable. Therefore, any reference to an enterprise Web Bean of scope `@Dependent` is serializable.
- A simple Web Bean of scope `@Dependent` may or may not be serializable. If a simple Web Bean of scope `@Dependent` and a non-serializable implementation class is injected into a stateful session bean, into a non-transient field, Web Bean constructor parameter or initializer method parameter of a Web Bean which declares a passivating scope type, or into a parameter of a producer method which declares a passivating scope type, an `UnserializableDependencyException` must be thrown by the container at deployment time.
- If a producer method or field of scope `@Dependent` returns a non-serializable object for injection into a stateful session bean, into a non-transient field, Web Bean constructor parameter or initializer method parameter of a Web Bean which declares a passivating scope type, or into a parameter of a producer method which declares a passivating scope type, an `IllegalProductException` is thrown by the container.
- The container must guarantee that JMS endpoint proxy objects are serializable.

The method `Bean.isSerializable()` may be used to detect if a Web Bean is serializable.

9.6. Context management for built-in scopes

The container provides an implementation of the `Context` interface for each of the built-in scopes.

For each of the built-in normal scopes, contexts propagate across any Java method call, including invocation of EJB local business methods. The built-in contexts do not propagate across remote method invocations or to asynchronous processes such as JMS message listeners or EJB timer service timeouts.

9.6.1. Request context lifecycle

The Web Beans request context is provided by a built-in context object for the built-in scope type `javax.webbeans.RequestScoped`.

- The request scope is active during the `service()` method of any Servlet in the web application. The request context is destroyed at the end of the servlet request, after the Servlet `service()` method returns.
- The request scope is active during any Java EE web service invocation. The request context is destroyed after the web service invocation completes.
- The request scope is active during any remote method invocation of any EJB bean, during any call to an EJB timeout method and during message delivery to any EJB message driven bean. The request context is destroyed after the remote method invocation, timeout or message delivery completes.

Open issue: currently it is impossible to intercept timeout methods. This needs to be fixed in EJB 3.1.

Open issue: is the request context (and application context) active during servlet filter execution?

9.6.2. Session context lifecycle

The Web Beans session context is provided by a built-in context object for the built-in passivating scope type `javax.webbeans.SessionScoped`.

The session scope is active during the `service()` method of any servlet in the web application.

The session context is shared between all servlet requests that occur in the same HTTP servlet session. The session context is destroyed when the `HTTPSession` is invalidated or times out.

9.6.3. Application context lifecycle

The Web Beans application context is provided by a built-in context object for the built-in scope type `javax.webbeans.ApplicationScoped`.

- The application scope is active during the `service()` method of any servlet in the web application.

- The application scope is active during any Java EE web service invocation.
- The application scope is also active during any remote method invocation of any EJB bean, during any call to an EJB timeout method and during message delivery to any EJB message driven bean.

The application context is shared between all servlet requests, web service invocations, EJB remote method invocations, EJB timeouts and message deliveries to message driven beans that execute within the same application. The application context is destroyed when the application is undeployed.

9.6.4. Conversation context lifecycle

The Web Beans conversation context is provided by a built-in context object for the built-in passivating scope type `javax.webbeans.ConversationScoped`.

- For a JSF faces request, the context is active from the beginning of the apply request values phase, until the response is complete.
- For a JSF non-faces request, the context is active during the render response phase.

The conversation context provides access to state associated with a particular *conversation*. Every JSF request has an associated conversation. This association is managed automatically by the container according to the following rules:

- Any JSF request has exactly one associated conversation
- The conversation associated with a JSF request is determined at the end of the restore view phase and does not change during the request

Any conversation is in one of two states: *transient* or *long-running*.

- By default, a conversation is transient
- A transient conversation may be marked long-running by calling `Conversation.begin()`
- A long-running conversation may be marked transient by calling `Conversation.end()`

All long-running conversations have a string-valued unique identifier, which may be set by the application when the conversation is marked long-running, or generated by the container.

The container provides a built-in Web Bean with API type `javax.webbeans.Conversation`, scope `@RequestScoped`, deployment type `@Standard` and binding type `@Current`, named `javax.webbeans.conversation`.

```
public interface Conversation {
   public void begin();
   public void begin(String id);
   public void end();
   public boolean isLongRunning();
   public String getId();
   public long getTimeout();
   public void setTimeout(long milliseconds);
}
```

If the conversation associated with the current JSF request is in the *transient* state at the end of a JSF request, it is destroyed, and the conversation context is also destroyed.

If the conversation associated with the current JSF request is in the *long-running* state at the end of a JSF request, it is not destroyed. Instead, it may be propagated to other requests according to the following rules:

- The long-running conversation context associated with a request that renders a JSF view is automatically propagated to any faces request (JSF form submission) that originates from that rendered page.
- The long-running conversation context associated with a request that results in a JSF redirect (via a navigation rule) is automatically propagated to the resulting non-faces request, and to any other subsequent request to the same URL. This is accomplished via use of a GET request parameter named `cid` containing the unique identifier of the conversation.

- The long-running conversation associated with a request may be propagated to any non-faces request via use of a GET request parameter named `cid` containing the unique identifier of the conversation. In this case, the application must manage this request parameter.

When no conversation is propagated to a JSF request, the request is associated with a new transient conversation.

All long-running conversations are scoped to a particular HTTP servlet session and may not cross session boundaries.

In the following cases, a propagated long-running conversation cannot be restored and reassociated with the request:

- When the HTTP servlet session is invalidated, all long-running conversation contexts created during the current session are destroyed.
- The container is permitted to arbitrarily destroy any long-running conversation that is associated with no current JSF request, in order to conserve resources.

If the propagated conversation cannot be restored, the request is associated with a new transient conversation.

The method `Conversation.setTimeout()` is a hint to the container that a conversation should not be destroyed if it has been active within the last given interval in milliseconds.

Open issue: allow the request to be blocked if the conversation cannot be restored.

The container ensures that a long-running conversation may be associated with at most one request at a time, by blocking or rejecting concurrent requests.

Open issue: define a mechanism for "blocking" requests. For example, allow the request to be redirected.

9.7. Context management for custom scopes

A custom implementation of `Context` may be associated with any scope type at any point in the execution of a Web Beans application, by calling `Manager.addContext()`.

```
public interface Manager {

    public Manager addContext(Context context);

    ...

}
```

For example:

```
manager.addContext(new MethodContext());
```

Every time `Manager.getInstance()` is called, for example, during instance or EL name resolution, the container must call `Manager.getContext()` to retrieve an active context object associated with the Web Bean scope. The `getContext()` method searches for an active context object for the given scope type. If no active context object exists for the given scope type, `getContext()` must throw a `ContextNotActiveException`. If more than one active context object exists for the given scope type, `getContext()` must throw an `IllegalStateException`.

```
public interface Manager {

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

    ...

}
```

Chapter 10. XML based metadata

The `web-beans.xml` file provides an alternative to the use of Java annotations for Web Bean definition. For example, this XML declaration defines a simple Web Bean with an injected field and an initializer method:

```
<myapp:MockAsynchronousCreditCardPaymentProcessor>
    <myapp:Asynchronous/>
    <myapp:PayBy>CREDIT_CARD</myapp:PayBy>
    <SessionScoped/>
    <myfwk:Mock/>
    <myfwk:Service transactional="true"/>
    <Named>asyncCreditCardPaymentProcessor</Named>

    <myapp:synchronousProcessor>
        <myapp:PaymentProcessor>
            <myapp:Synchronous/>
            <myapp:PayBy>CREDIT_CARD</myapp:PayBy>
        </myapp:PaymentProcessor>
    <myapp:synchronousProcessor>

    <myapp:init>
        <Initializer/>
        <myfwk:SystemConfig/>
    </myapp:init>

 </myapp:MockAsynchronousCreditCardPaymentProcessor>
```

It is the equivalent to the following declaration using annotations:

```
@Asynchronous
@PayBy(CREDIT_CARD)
@SessionScoped
@Mock
@Service(transactional=true)
@Named("asyncCreditCardPaymentProcessor")
class MockAsynchronousCreditCardPaymentProcessor {

    @Synchronous @PayBy(CREDIT_CARD) PaymentProcessor synchronousProcessor;

    @Initializer void init(SystemConfig config) { ... }

    ...
}
```

XML-based Web Bean declarations define additional Web Beans—they do not redefine or disable any Web Bean that was declared via annotations.

The file format is typesafe and extensible:

- Multiple namespaces are accommodated, each representing a Java package.
- XML elements belonging to these namespaces represent Java types, fields and methods.
- Each namespace may declare an XML schema.

10.1. XML namespace for a Java package

Every Java package has a corresponding XML namespace. The namespace URN consists of the package name, with the prefix `urn:java:`. For example, the package `com.mydomain.myapp` has the XML namespace `urn:java:com.mydomain.myapp`.

```
<WebBeans xmlns="urn:java:javax.webbeans"
          xmlns:myapp="urn:java:com.mydomain.myapp">
    ...
</WebBeans>
```

Each namespace may, optionally, have a 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">
    ...
</WebBeans>
```

An XML element belonging to a namespace represents a Java type in the corresponding Java package, or a method or field of a type in that package. There are exactly ten exceptions to this rule: the root `<WebBeans>` element together with the `<Queue>`, `<Topic>`, `<destination>`, `<connectionFactory>`, `<value>`, `<Deploy>`, `<Interceptors>`, `<Decorators>` and `<Array>` elements in the namespace `urn:java:javax.webbeans` do not correspond to Java types or members of Java types.

A class, interface or annotation type is represented by an element with the same name as the type, in the namespace corresponding to the Java package. For example, the element `<List>` in the namespace `urn:java:java.util` represents `java.util.List`.

Type parameters may be specified by child elements of the element that represents the type. For example:

```
<util:List>
    <myapp:Product/>
</util:List>
```

Members of a type may be specified by child elements of the element that represents the type, in the same namespace as the element that represents the type. For example:

```
<myapp:ShoppingCart>
    <myapp:paymentProcessor>
        ...
    </myapp:paymentProcessor>
</myapp:ShoppingCart>
```

Primitive types may be represented by the XML element that represents the corresponding wrapper type in `java.lang`, since primitive and wrapper types are considered identical for the purposes of typesafe resolution, and assignable for the purposes of injection. For example, the element `<Integer>` in the namespace `urn:java:java.lang` represent both `int` and `java.lang.Integer`.

Java array types may be represented by an `<Array>` element in the namespace `urn:java:javax.webbeans`, with a child element representing the element type. For example:

```
<Array>
    <myapp:Product/>
</Array>
```

The namespace `urn:java:javax.webbeans` is called the Web Beans namespace.

If a `web-beans.xml` file contains any XML element without a declared namespace, a `DefinitionException` is thrown by the container at deployment time.

10.2. Stereotype, binding type and interceptor binding type declarations

An XML element that appears as a direct child of the root `<WebBeans>` element is interpreted as a binding type, interceptor binding type or stereotype declaration if it has a direct child `<BindingType>`, `<InterceptorBindingType>` or `<Stereotype>` element in the Web Beans namespace, as defined in Section 2.3.2, “Defining new binding types”, Section 7.2.4, “Interceptor bindings” and Section 2.7.1, “Defining new stereotypes”.

The name of the XML element is interpreted as a Java type name in the package corresponding to the child element namespace. If no such Java type exists in the classpath, a `NonexistentTypeException` is thrown by the container at deployment time. If the type is not an annotation type, a `DefinitionException` is thrown by the container at deployment time.

If the annotation type is already declared as a binding type, interceptor binding type or stereotype using annotations, the annotations are ignored by the container and the XML-based declaration is used.

If a certain annotation type is declared more than once as a binding type, interceptor binding type or stereotype using XML, a `DeploymentException` is thrown by the container at deployment time.

10.2.1. Child elements of a stereotype declaration

Every direct child element of a stereotype declaration is interpreted as a Java type name in the package corresponding to the child element namespace. If no such Java type exists in the classpath, a `NonexistentTypeException` is thrown by the container at deployment time. If the type is not an annotation type, a `DefinitionException` is thrown by the container at deployment time.

- If the annotation type is a Web Beans scope type, the default scope of the stereotype was declared.
- If the annotation type is a Web Beans deployment type, the default scope of the stereotype was declared.
- If the annotation type is a Web Beans interceptor binding type, an interceptor binding type of the stereotype was declared.
- If the annotation type is `javax.webbeans.Named`, a stereotype with name defaulting was declared.
- Otherwise, a `DefinitionException` is thrown by the container at deployment time.

10.2.2. Child elements of an interceptor binding type declaration

Every direct child element of an interceptor binding type declaration is interpreted as a Java type name in the package corresponding to the child element namespace. If no such Java type exists in the classpath, a `NonexistentTypeException` is thrown by the container at deployment time. If the type is not an annotation type, a `DefinitionException` is thrown by the container at deployment time.

- If the annotation type is a Web Beans interceptor binding type, an inherited interceptor binding type was declared, as defined in Section 7.2.4.1, “Interceptor binding types with additional interceptor bindings”.
- Otherwise, a `DefinitionException` is thrown by the container at deployment time.

10.3. Web Bean declarations

An XML element that appears as a direct child of the root `<WebBeans>` element is interpreted as a Web Bean declaration if it is not a `<Deploy>`, `<Interceptors>` or `<Decorators>` element in the Web Beans namespace, and does not have a direct child `<BindingType>`, `<InterceptorBindingType>` or `<Stereotype>` element in the Web Beans namespace.

If the XML element is a `<Queue>` or `<Topic>` element in the Web Beans namespace, it declares a JMS endpoint, as defined in Section 3.6.2, “Declaring a JMS endpoint using XML”.

Otherwise, the name of the XML element is interpreted as a Java type name in the package corresponding to the child element namespace. The container inspects the Java type and other metadata to determine what kind of Web Bean is being declared. If no such Java type exists in the classpath, a `NonexistentTypeException` is thrown by the container at deployment time.

- If the type is an EJB bean class, an enterprise Web Bean was declared, as defined in Section 3.3.5, “Declaring an enterprise Web Bean using XML”.
- If the type is a concrete class, is not an EJB bean class, and is not a parameterized type, a simple Web Bean was declared, as defined in Section 3.2.4, “Declaring a simple Web Bean using XML”.
- Otherwise, a `DefinitionException` is thrown by the container at deployment time.

For example, the following XML file declares a simple Web Bean with the implementation class `com.mydomain.myapp.PaymentProcessor`:

```
<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:PaymentProcessor>
        ...
    </myapp:PaymentProcessor>

</WebBeans>
```

In addition, inline Web Bean declarations may occur at injection points, as defined in Section 10.7, “Inline Web Bean declarations”. Inline Web Bean declarations always declare simple Web Beans.

10.3.1. Child elements of a Web Bean declaration

The container inspects the direct child elements of the Web Bean declaration. For each child element:

- If the name of the child element is the name of a Java annotation type in the package corresponding to the child element namespace, the container interprets the child element as declaring type-level metadata.
- If the name of the child element is the name of a Java class or interface in the package corresponding to the child element namespace, the container interprets the child element as declaring a parameter of the Web Bean constructor.
- Otherwise, if the child element namespace is the same as the namespace of the parent, the container interprets the element as declaring a method or field of the Web Bean.
    - If the name of the child element matches the name of both a method and a field of the Web Bean implementation class, a `DefinitionException` is thrown by the container at deployment time.
    - Otherwise, if the name of the child element matches the name of a method of the Web Bean implementation class, is it interpreted to represent that method.
    - Otherwise, if the name of the child element matches the name of a field of the Web Bean implementation class, is it interpreted to represent that field.
- Otherwise, a `DefinitionException` is thrown by the container at deployment time.

10.3.2. Type-level metadata for a Web Bean

Type-level metadata is specified via direct child elements of the Web Bean declaration that represent Java annotation types.

The name of the child element is interpreted as the name of a Java annotation type in the package corresponding to the child element namespace.

For each child element, the container inspects the annotation type:

- If the annotation type is a deployment type, the deployment type of the Web Bean was declared, as defined in Section 2.5.4, “Declaring the deployment type of a Web Bean using XML”.
- If the annotation type is a scope type, the scope of the Web Bean was declared, as defined in Section 2.4.4, “Declaring the Web Bean scope using XML”.
- If the annotation type is a binding type, a binding type of the Web Bean was declared, as defined in Section 2.3.4, “Declaring the binding types of a Web Bean using XML”.
- If the annotation type is an interceptor binding type, an interceptor binding type of the Web Bean was declared, as defined in Section 7.2.6.2, “Binding a Web Beans interceptor using XML”.
- If the annotation type is a stereotype, a stereotype of the Web Bean was declared, as defined in Section 2.7.3, “Declaring the stereotypes for a Web Bean using XML”.
- If the annotation type is `javax.webbeans.Name`, the name of the Web Bean was declared, as defined in Section 2.6.2, “Declaring the Web Bean name using XML”.
- If the annotation type is `javax.webbeans.Specializes`, the Web Bean was declared to directly specialize the Web Bean with the same implementation class that was defined using annotations, as specified in Section 3.2.6, “Specializing a simple Web Bean” and Section 3.3.6, “Specializing an enterprise Web Bean”.
- If the annotation type is `javax.webbeans.Interceptor`, or `javax.webbeans.Decorator` the Web Bean is an interceptor or decorator, as defined in Section 10.5, “Interceptor and decorator declarations”.

- Otherwise, a `DefinitionException` is thrown by the container at deployment time.

10.3.3. Web Bean constructor declarations

The Web Bean constructor for a simple Web Bean is declared by the list of direct child elements of the Web Bean declaration that represent Java class or interface types. The container interprets these elements as declaring parameters of the constructor.

```
<myapp:Order>
    <ConversationScoped/>
    <myapp:PaymentProcessor>
        <myapp:Asynchronous/>
    </myapp:PaymentProcessor>
    <myapp:User/>
</myapp:Order>
```

Each constructor parameter declaration is interpreted as an injection point declaration, as specified in Section 10.6, “Injection point declarations”.

If the simple Web Bean implementation class has exactly one constructor such that:

- the constructor has the same number of parameters as the Web Bean declaration has constructor parameter declarations, and
- the Java type represented by each constructor parameter declaration is assignable to the Java type of the corresponding constructor parameter

then the element is interpreted to represent that constructor, and that constructor is the Web Bean constructor.

If more than one constructor exists which satisfies these conditions, a `DefinitionException` is thrown by the container at deployment time.

If no constructor of the simple Web Bean implementation class satisfies these conditions, a `NonexistentConstructorException` is thrown by the container at deployment time.

For any constructor parameter, the API type declared in XML may be a subtype of the Java parameter type. In this case, the container will use the API type declared in XML when resolving the dependency.

10.3.4. Fields of a Web Bean

A field of a Web Bean is declared by a direct child element of the Web Bean declaration. The name of the field is the same as the name of the element.

If the Web Bean implementation class has exactly one field with the same name as the child element, then the child element is interpreted to represent that field.

Otherwise, if the Web Bean implementation class does not have exactly one field with the specified name, a `DefinitionException` is thrown by the container at deployment time.

If more than one child element of a Web Bean declaration represents the same field of the Web Bean implementation class, a `DefinitionException` is thrown by the container at deployment time.

A field declaration may contain child elements. If a field declaration has more than one direct child element, and at least one of these elements is something other than a `<value>` element in the Web Beans namespace, a `DefinitionException` is thrown by the container at deployment time.

An element that represents a field may declare an injected field, a producer field or a field with an initial value.

- If the element contains a child `<Produces>` element in the Web Beans namespace, a producer field was declared, as defined in Section 3.5.3, “Declaring a producer field using XML”.
- If the element contains a child `<value>` element in the Web Beans namespace, a field with an initial value of type `Set` or `List` was declared, as defined in Section 10.3.5, “Field initial value declarations”.

- Otherwise, if the element has exactly one child element, an injected field was declared, as defined in Section 3.7.2, “Declaring an injected field using XML”.
- If the element has a non-empty body, and no child elements, a field with an initial value was declared, as defined in Section 10.3.5, “Field initial value declarations”.
- Otherwise, a `DefinitionException` is thrown by the container at deployment time.

If a field declaration represents an injected field, the child element is interpreted as an injection point declaration, as specified in Section 10.6, “Injection point declarations”. If the declared type is not assignable to the Java type of the field, a `DefinitionException` is thrown by the container at deployment time.

The API type declared in XML may be a subtype of the Java field type. In this case, the container will use the API type declared in XML when resolving the dependency.

10.3.5. Field initial value declarations

The initial value of a field of a simple Web Bean or enterprise Web Bean with any one of the following types may be specified in XML:

- any primitive type, or `java.lang` wrapper type
- any enumerated type
- `java.lang.String`
- `java.util.Date`, `java.sql.Date`, `java.sql.Time` or `java.sql.Timestamp`
- `java.util.Calendar`
- `java.math.BigDecimal` or `java.math.BigInteger`
- `java.lang.Class`
- `java.util.List<java.lang.String>` or `java.util.Set<java.lang.String>`
- `java.util.List<java.lang.Class>` or `java.util.Set<java.lang.Class>`
- `java.util.List<X>` or `java.util.Set<X>` where `X` is an enumerated type.

The initial value of the field is specified in the body of an XML element representing the field.

```
<myapp:Config>
    <myapp:version>1.2.5</myapp:version>
    <myapp:timeout>1000</myapp:timeout>
    <myapp:administrators>
        <value>juan</value>
        <value>antonio</value>
        <value>sonia</value>
        <value>sara</value>
    </myapp:administrators>
</myapp:Config>
```

- The initial value of a field of primitive type or `java.lang` wrapper type is specified using the Java literal syntax for that type.
- The initial value of a field of type `java.lang.String` is specified using the string value.
- The initial value of a field of enumerated type is specified using the unqualified name of the enumeration value.
- The initial value of a field of type `java.util.Date`, `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp` or `java.util.Calendar` is specified using a format that can be parsed by calling `java.text.DateFormat.getDateTimeInstance().parse()`.
- The initial value of a field of type `java.math.BigDecimal` or `java.math.BigInteger` is specified using a format that can be parsed by the constructor that accepts a string.

- The initial value of a field of type `java.lang.Class` is specified using the fully qualified Java class name.

The initial value of a field of type `java.util.List` or `java.util.Set` is specified by a list of `<value>` elements. The body of the value element is specified using the string value, fully qualified Java class name or unqualified name of the enumeration value.

If a field with an initial value specified in XML is not of one of the listed types, or if the initial value is not specified in the correct format for the type of the field, a `DefinitionException` is thrown by the container at deployment time.

If an element representing a field specifies both an initial value and a type declaration, a `DefinitionException` is thrown by the container at deployment time.

10.3.6. Methods of a Web Bean

A method of a Web Bean is declared by a direct child element of the Web Bean declaration. The name of the declared method is the same as the name of the child element.

A method declaration may have any number of direct child elements.

The container inspects the direct child elements of the method declaration. For each child element, the name of the element is interpreted as a Java type name in the package corresponding to the element's namespace. If no such Java type exists in the classpath, a `NonexistentTypeException` is thrown by the container at deployment time.

- If the type is `javax.webbeans.Disposes`, the container searches for a direct child element of the child element and interprets that element as declaring a disposed parameter of the disposal method.
- If the type is `javax.webbeans.Observes`, the container searches for a direct child element of the child element that is not an `<IfExists>`, `<AfterTransactionCompletion>`, `<AfterTransactionSuccess>`, `<AfterTransactionFailure>` or `<BeforeTransactionCompletion>` element in the Web Beans namespace, and interprets that element as declaring an event parameter of the observer method.
- If the type is some other Java annotation type, the container interprets the child element as declaring method-level metadata.
- If the type is a Java class or interface, the container interprets the child element as declaring a parameter of the method.
- Otherwise, a `DefinitionException` is thrown by the container at deployment time.

If a method declaration has more than one direct child element which is an `<Initializer>`, `<Destructor>`, `<Produces>`, `<Disposes>` or `<Observes>` element in the Web Beans namespace, a `DefinitionException` is thrown by the container at deployment time.

If a `<Disposes>` element does not contain exactly one direct child element, a `DefinitionException` is thrown by the container at deployment time.

If an `<Observes>` element does not contain exactly one direct child element that is not an `<IfExists>`, `<AfterTransactionCompletion>`, `<AfterTransactionSuccess>`, `<AfterTransactionFailure>` or `<BeforeTransactionCompletion>` element in the Web Beans namespace, a `DefinitionException` is thrown by the container at deployment time.

Each method parameter declaration and disposed parameter declaration is interpreted as an injection point declaration, as specified in Section 10.6, “Injection point declarations”. An event parameter declaration is interpreted as a type declaration, as defined in Section 10.8, “Specifying API types and binding types”.

If the Web Bean implementation class has exactly one method such that:

- the method name is the same as the name of the element that declares the method,
- the method has the same number of parameters as the element that declares the method has child elements, and
- the Java type represented by each method parameter declaration is assignable to the Java type of the corresponding method parameter

then the element is interpreted to represent that method.

If more than one method exists which satisfies these conditions, a `DefinitionException` is thrown by the container at deployment time.

If no method of the Web Bean implementation class satisfies these conditions, a `NonexistentMethodException` is thrown by the container at deployment time.

For any method parameter, the API type declared in XML may be a subtype of the Java parameter type. In this case, the container will use the API type declared in XML when resolving the dependency.

If more than one child element of a Web Bean declaration represents the same method of the Web Bean implementation class, a `DefinitionException` is thrown by the container at deployment time.

An element that represents a method may declare an initializer method, an observer method, a producer method or a disposal method. Alternatively, or additionally, it may declare method-level interceptor binding types.

- If the element contains a child `<Initializes>` element in the Web Beans namespace, an initializer method was declared, as defined in Section 3.8.2, “Declaring an initializer method using XML”.
- If the element contains a child `<Produces>` element in the Web Beans namespace, a producer method was declared, as defined in Section 3.4.3, “Declaring a producer method using XML”.
- If the element contains a child `<Disposes>` element in the Web Beans namespace, a disposal method was declared, as defined in Section 3.4.9, “Declaring a disposal method using XML”.
- If the element contains a child `<Observes>` element in the Web Beans namespace, an observer method was declared, as defined in Section 8.5.3, “Declaring an observer method using XML”.
- If the element contains a child element whose name is the name of a Web Beans interceptor binding type in the package corresponding to the child element namespace, method-level interceptor binding type was declared, as defined in Section 7.2.6.2, “Binding a Web Beans interceptor using XML”.

10.4. Producer method and field declarations

A producer method or field declaration is formed by adding a direct child `<Produces>` element to an element that represents the method or field, as defined in Section 3.4.3, “Declaring a producer method using XML” and Section 3.5.3, “Declaring a producer field using XML”.

```
<myapp:getPaymentProcessor>
    <Produces>
        <myapp:PaymentProcessor/>
    </Produces>
    ...
</myapp:getPaymentProcessor>
```

```
<myapp:paymentProcessor>
    <Produces>
        <myapp:PaymentProcessor/>
    </Produces>
</myapp:paymentProcessor>
```

10.4.1. Child elements of a producer field declaration

The container inspects the direct child elements of a producer field declaration.

If there is more than one direct child element, a `DefinitionException` is thrown by the container at deployment time.

Otherwise, the direct child element is a `<Produces>` element in the Web Beans namespace, and declares the return type, binding types and member-level metadata of the producer field.

The container inspects the direct child elements of the `<Produces>` element. For each child element, the name of the element is interpreted as a Java type name in the package corresponding to the child element namespace. If no such Java type exists in the classpath, a `NonexistentTypeException` is thrown by the container at deployment time.

- If the type is a Java class or interface type, the type of the producer field was declared.
- If the type is a Java annotation type, it declares member-level metadata of the producer field.
- Otherwise, a `DefinitionException` is thrown by the container at deployment time.

If more than one child element represents a Java class or interface type, or if no child element represents a Java class or interface type, a `DefinitionException` is thrown by the container at deployment time.

10.4.2. Child elements of a producer method declaration

The container inspects the direct child elements of a producer method declaration.

- If a child element is the `<Produces>` element in the Web Beans namespace, it declares the return type, binding types and member-level metadata of the producer method.
- If the child element name is the name of a Web Beans interceptor binding type in the package corresponding to the child element namespace, it declares a method-level interceptor binding type.
- Otherwise, the container interprets the child element as declaring a parameter of the producer method.

If there is more than one child `<Produces>` element in the Web Beans namespace, a `DefinitionException` is thrown by the container at deployment time.

The container inspects the direct child elements of the `<Produces>` element. For each child element, the name of the element is interpreted as a Java type name in the package corresponding to the child element namespace. If no such Java type exists in the classpath, a `NonexistentTypeException` is thrown by the container at deployment time.

- If the type is a Java class or interface type, the return type of the producer method was declared.
- If the type is a Java annotation type, it declares member-level metadata of the producer method.
- Otherwise, a `DefinitionException` is thrown by the container at deployment time.

If more than one child element represents a Java class or interface type, or if no child element represents a Java class or interface type, a `DefinitionException` is thrown by the container at deployment time.

10.4.3. Return type and binding types of a producer method or field

Every XML producer method or field declaration has a direct child `<Produces>` element. This element must, in turn, have a direct child element which declares the return type of the producer method or the type of the producer field and which is interpreted by the container as a type declaration, as defined in Section 10.8, “Specifying API types and binding types”.

This type declaration specifies the return type and binding types of the producer method Web Bean, or the type and binding types of the producer field Web Bean. The type is used to calculate the set of API types. The type declared in XML must be a supertype or subtype of the Java method or field type. If the declared type is not a supertype or subtype of the Java method or field type, a `DefinitionException` is thrown by the container at deployment time.

10.4.4. Member-level metadata for a producer method or field

Member-level metadata for a producer method or field declaration is specified via direct child elements of the `<Produces>` element that represent Java annotation types.

The name of each child element is interpreted as the name of a Java annotation type in the package corresponding to the child element namespace. If the declared type is not a Java annotation type, a `DefinitionException` is thrown by the container at deployment time.

The container inspects the annotation type:

- If the annotation type is a deployment type, the deployment type of the producer method or field was declared, as defined in Section 2.5.4, “Declaring the deployment type of a Web Bean using XML”.

- If the annotation type is a scope type, the scope of the producer method or field was declared, as defined in Section 2.4.4, “Declaring the Web Bean scope using XML”.
- If the annotation type is a stereotype, a stereotype of the producer method or field was declared, as defined in Section 2.7.3, “Declaring the stereotypes for a Web Bean using XML”.
- If the annotation type is `javax.webbeans.Name`, the name of the producer method or field was declared, as defined in Section 2.6.2, “Declaring the Web Bean name using XML”.
- Otherwise, a `DefinitionException` is thrown by the container at deployment time.

10.5. Interceptor and decorator declarations

A simple Web Bean declaration is interpreted as an interceptor or decorator declaration if it contains a direct child `<Interceptor>` or `<Decorator>` element in the Web Beans namespace.

For example, the following XML file declares an interceptor of class `RequiresTransactionInterceptor`, an interceptor of class `RequiresNewTransactionInterceptor` and a decorator of class `DataAccessAuthorizationDecorator`, all in the Java package `com.mydomain.myfwk`:

```
<WebBeans xmlns="urn:java:javax.webbeans"
          xmlns:myapp="urn:java:com.mydomain.myapp"
          xmlns:myfwk="urn:java:com.mydomain.myfwk"
          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.myfwk http://mydomain.com/xsd/myfwk-1.0.xsd
                              urn:java:com.mydomain.myapp http://mydomain.com/xsd/myapp-1.2.xsd">

    <myfwk:RequiresTransactionInterceptor>
        <Interceptor/>
        <myfwk:Transactional/>
    </myfwk:RequiresTransactionInterceptor>

    <myfwk:RequiresNewTransactionInterceptor>
        <Interceptor/>
        <myfwk:Transactional requiresNew="true"/>
    </myfwk:RequiresNewTransactionInterceptor>

    <myfwk:DataAccessAuthorizationDecorator>
        <Decorator/>
        <myfwk:dataAccess>
            <Decorates>
                <myfwk:DataAccess/>
            </Decorates>
        </myfwk:dataAccess>
    <myfwk:DataAccessAuthorizationDecorator>

</WebBeans>
```

If a Web Bean declaration that is not a simple Web Bean declaration contains a child `<Interceptor>` or `<Decorator>` element, or if an inline Web Bean declaration contains a child `<Interceptor>` or `<Decorator>` element, a `DefinitionException` is thrown by the container at deployment time.

If a simple Web Bean declaration contains more than one direct child `<Interceptor>` or `<Decorator>` element in the Web Beans namespace, a `DefinitionException` is thrown by the container at deployment time.

10.5.1. Decorator delegate attribute

Decorator declarations may declare the delegate attribute. A delegate declaration is a direct child element of the decorator declaration. The name of the delegate attribute is the same as the name of the element.

If a direct child element of a decorator declaration:

- exists in the same namespace as its parent, and
- has direct child `<Decorates>` element in the Web Beans namespace

then it is interpreted as a delegate declaration.

If the Web Bean implementation class has a field with the same name as the child element, then the child element is interpreted to represent that field.

If the Web Bean implementation class does not have have a field with the specified name, a `NonexistentFieldException` is thrown by the container at deployment time.

If a delegate declaration has more than one direct child element, a `DefinitionException` is thrown by the container at deployment time. This child element is a `<Decorates>` element in the Web Beans namespace. If the `<Decorates>` element does not, in turn, have exactly one direct child element, a `DefinitionException` is thrown by the container at deployment time.

The direct child element of the `<Decorates>` element is interpreted as a type declaration as specified by Section 10.8, “Specifying API types and binding types”. If the declared API type is not assignable to the type of the Java field, a `DefinitionException` is thrown by the container at deployment time.

The API type declared in XML may be a subtype of the Java field type. In this case, the container will use the API type declared in XML when resolving the dependency.

If simple Web Bean declaration that is not a decorator declaration contains a direct child element that in turn contains a direct child `<Decorates>` element, a `DefinitionException` is thrown by the container at deployment time.

10.6. Injection point declarations

An *injection point declaration* is either:

- a type declaration, as defined in Section 10.8, “Specifying API types and binding types”, or
- an inline Web Bean declaration, as defined in Section 10.7, “Inline Web Bean declarations”.

When the container encounters an injection point declaration, it interprets the name of the element as the name of a Java class or interface in the package corresponding to the element namespace. If no such Java type exists in the classpath, a `NonexistentTypeException` is thrown by the container at deployment time.

- If the Java type is a parameterized type, the injection point declaration is a type declaration, and the declared type of the injection point is the API type of the type declaration, including actual type parameters.
- Otherwise, the container inspects the direct child elements. If the name of any direct child element is the name of a Web Beans binding type in the package corresponding to the child element namespace, the injection point declaration is a type declaration, and the declared type of the injection point is the API type of the type declaration.
- Otherwise, if any direct child elements exist, the injection point declaration is an inline Web Bean declaration, and the declared type of the injection point is the implementation class of the Web Bean.
- Otherwise, the injection point declaration is a type declaration, and the declared type of the injection point is the API type of the type declaration.

10.7. Inline Web Bean declarations

An inline Web Bean declaration is a simple Web Bean declaration, as defined in Section 10.3, “Web Bean declarations” that occurs as an injection point declaration, instead of as a direct child of the `<WebBeans>` element.

```
<myapp:Admin>
    <ApplicationScoped/>

    <myapp:username>gavin</myapp:username>

    <myapp:name>
        <myapp:Name>
            <myapp:firstName>Gavin</myapp:firstName>
            <myapp:lastName>King</myapp:lastName>
        </myapp:Name>
    </myapp:name>

</myapp:Admin>
```

The name of the element is interpreted as the name of a Java class in the package corresponding to the element namespace. This Java class is the implementation class of the simple Web Bean.

Inline Web Bean declarations may not explicitly specify a binding type. If an inline Web Bean declaration explicitly specifies a binding type, a `DefinitionException` is thrown by the container at deployment time.

For every inline injection point, the container generates a unique value for an implementation-specific binding type. (For example, a particular container implementation might generate the value `com.vendor.webbeans.Inline(id=12345)` at some injection point.) This generated value is the binding type of the injection point, and the only binding type of the simple Web Bean. The API type of the injection point is the declared implementation class of the simple Web Bean.

Thus, an inline Web Bean declaration results in a simple Web Bean that is bound only to the injection point at which it was declared.

10.8. Specifying API types and binding types

Every injection point, event parameter and delegate attribute defined in XML must explicitly specify an API type and combination of binding types. XML-based producer method declarations must also explicitly specify the return type (which is used to calculate the set of API types) and binding types. A *type declaration* is:

- an element that represents a Java class or interface, or `<Array>`,
- if the type is a parameterized type, a set of child elements that represent Java classes and/or interfaces, and are interpreted as the actual type parameters, or, if the type is an array type, a single child element that represents the array element type,
- optionally, a set of child elements that represent Java annotation types, and are interpreted as binding types.

For example, the following XML fragment declares the type `List<Product>` with binding type `@All`.

```
<util:List>
    <myapp:All/>
    <myapp:Product/>
</util:List>
```

This XML fragment declares the type `Product[]` with binding type `@Available`.

```
<Array>
    <myapp:Available/>
    <myapp:Product/>
</Array>
```

When the container encounters a type declaration it interprets the element as a Java type:

- If the element is an `<Array>` element in the Web Beans namespace, an array type was declared.
- Otherwise, the name of the element is interpreted as the name of a Java class or interface in the package corresponding to the element namespace. If no such Java type exists in the classpath, a `NonexistentTypeException` is thrown by the container at deployment time. If the Java type is not a class or interface type, a `DefinitionException` is thrown by the container at deployment time.

Next, the container inspects every direct child element of the type declaration. The name of each child element is interpreted as the name of a Java type in the package corresponding to the child element namespace. If no such Java type exists in the classpath, a `NonexistentTypeException` is thrown by the container at deployment time.

- If the type is a Java annotation type, a binding type was declared.
- If the type is a Java class or interface type, an actual type parameter or array element type was declared.
- Otherwise, a `DefinitionException` is thrown by the container at deployment time.

If multiple array element types are declared, a `DefinitionException` is thrown by the container at deployment time.

If the number of declared actual type parameters is not the same as the number of parameters of the Java type, a `DefinitionException` is thrown by the container at deployment time.

If a type parameter of the Java type is bounded, and the corresponding declared actual type parameter does not satisfy the upper or lower bound, a `DefinitionException` is thrown by the container at deployment time.

If a binding type declaration declares a Java annotation type that is not a Web Beans binding type, a `DefinitionException` is thrown by the container at deployment time.

If no binding type is declared, the default binding type `@Current` is assumed.

If the same binding type occurs more than once, a `DuplicateBindingTypeException` is thrown by the container at deployment time.

For fields, type declarations are specified as direct child elements of the field declaration:

```
<myapp:Order>

    <myapp:paymentProcessor>

        <myapp:PaymentProcessor>
            <myapp:PayBy>CHEQUE</myapp:PayBy>
        </myapp:PaymentProcessor>

    </myapp:paymentProcessor>

</myapp:Order>
```

```
<myapp:ShoppingCart>

    <myapp:catalog>

        <util:List>
            <myapp:All/>
            <myapp:Product/>
        </util:List>

    </myapp:catalog>

</myapp:ShoppingCart>
```

For methods, the method parameter declarations are type declarations:

```
<myapp:Order>

    <myapp:setPaymentProcessor>
        <Initializer/>

        <myapp:PaymentProcessor>
            <myapp:PayBy>CHEQUE</myapp:PayBy>
        </myapp:PaymentProcessor>

        <myfwk:Logger/>

    </myapp:setPaymentProcessor>

</myapp:Order>
```

For producer methods, the return type must also be specified:

```
<app:Shop>

    <app:getAvailableProducts>

        <Produces>
            <ApplicationScoped/>
            <Array>
                <app:Available/>
                <app:Product/>
            </Array>
        </Produces>

        <util:List>
            <app:All/>
            <app:Product/>
        </util:List>
```

```
    </app:getAvailableProducts>

</app:Shop>
```

For constructors, the constructor parameter declarations are type declarations:

```
<myapp:Order>
    <ConversationScoped/>

    <myapp:PaymentProcessor>
        <myapp:PayBy>CHEQUE</myapp:PayBy>
    </myapp:PaymentProcessor>

    <myfwk:Logger/>

</myapp:Order>
```

10.9. Annotation members

Any binding type or interceptor binding type declaration must define the value of any annotation member without a default value, and may additionally define the value of any annotation member with a default value. Annotation member values are defined by attributes of the XML element which represents the Java annotation.

All attributes of any XML element which corresponds to a Java annotation are interpreted as members of the annotation. The name of the attribute is interpreted as the name of the corresponding annotation member. The value of the attribute is interpreted as the value of the annotation member. If there is no annotation member with the same name as the attribute, a `NonexistentMemberException` is thrown by the container at deployment time.

```
<myfwk:DataAccess transactional="true"/>
```

Alternatively, the value of an annotation member named `value` may be specified in the body of the XML element which corresponds to the Java annotation. If the XML element has a non-empty body and also specifies an attribute named `value`, a `DefinitionException` is thrown by the container at deployment time. If the XML element has a non-empty body, and there is no annotation member named `value`, a `NonexistentMemberException` is thrown by the container at deployment time.

```
<myapp:PayBy>CHEQUE</myapp:PayBy>
```

- The value of a member of primitive type is specified using the Java literal syntax for that type.
- The value of a member of type `java.lang.String` is specified using the string value.
- The value of a member of enumerated type is specified using the unqualified name of the enumeration value.
- The value of a member of type `java.lang.Class` is specified using the fully qualified Java class name.

If the member value is not specified in the correct format for the type of the member, a `DefinitionException` is thrown by the container at deployment time.

If an XML element that refers to a Java annotation with a member with no default value does not declare a value for that member, a `DefinitionException` is thrown by the container at deployment time.

10.10. Deployment declarations

The `<Deploy>`, `<Interceptors>` and `<Decorators>` elements in the Web Beans namespace determine which Web Beans, interceptors and decorators are enabled in a particular deployment.

10.10.1. The `<Deploy>` declaration

Each direct child element of a `<Deploy>` element is interpreted as the declaring an enabled deployment type, as specified in Section 2.5.6, “Enabled deployment types”.

For each child element, the name of the child element is interpreted as the name of a Java annotation type in the package

corresponding to the child element namespace. If no such Java type exists in the classpath, a `DefinitionException` is thrown by the container at deployment time. If the type is not a Web Beans deployment type, a `DefinitionException` is thrown by the container at deployment time.

If the same deployment type is declared more than once, a `DefinitionException` is thrown by the container at deployment time.

10.10.2. The `<Interceptors>` declaration

Each direct child element of an `<Interceptors>` element is interpreted as the declaring an enabled interceptor, as specified in Section 7.2.7, “Interceptor enablement and ordering”.

For each child element, the name of the child element is interpreted as the name of a Java class in the package corresponding to the child element namespace. If no such Java class exists in the classpath, a `DefinitionException` is thrown by the container at deployment time.

If the same interceptor is declared more than once, a `DefinitionException` is thrown by the container at deployment time.

10.10.3. The `<Decorators>` declaration

Each direct child element of a `<Decorators>` element is interpreted as the declaring an enabled decorator, as specified in Section 7.3.5, “Decorator enablement and ordering”.

For each child element, the name of the child element is interpreted as the name of a Java class in the package corresponding to the child element namespace. If no such Java class exists in the classpath, a `DefinitionException` is thrown by the container at deployment time.

If the same decorator is declared more than once, a `DefinitionException` is thrown by the container at deployment time.

Chapter 11. Packaging and deployment

Web Bean discovery is the process of determining:

- What Web Beans, interceptors and decorators *exist* in the deployment archive
- Which Web Beans, interceptors and decorators are *enabled* for this deployment
- The *precedence* of the enabled Web Beans, and the *ordering* of enabled interceptors and decorators

The container automatically discovers Web Beans when the EAR or WAR is deployed, or when the embeddable EJB Lite container is initialized.

Web Bean classes must be deployed in an EAR, WAR, EJB-JAR or JAR archive or directory in the application classpath that has a file named `web-beans.xml` in the metadata directory (`META-INF`, or `WEB-INF` in the case of a WAR). If a Web Bean is deployed to a location that is not in the application classpath, or does not contain a file named `web-beans.xml` in the metadata directory, it will not be discovered by the container.

Additional Web Beans may be registered programatically with the container by the application or third-party framework after the automatic Web Bean discovery completes. Third-party frameworks may even provide the ability to register certain Web Bean definitions with a *child container*, thereby limiting their visibility to certain contexts.

11.1. Web Bean discovery

When Web Bean discovery occurs, the container considers:

- any `web-beans.xml` file in any metadata directory of the application classpath,
- any `ejb-jar.xml` file in any metadata directory of the application classpath that also contains a `web-beans.xml` file, and
- any Java class in any archive or directory in the classpath that has a `web-beans.xml` file in the metadata directory.

First, the container discovers all binding types, stereotypes and interceptor binding types declared in XML, according to the rules of Section 10.2, “Stereotype, binding type and interceptor binding type declarations”.

The container automatically discovers simple Web Beans (according to the rules of Section 3.2.1, “Which Java classes are simple Web Beans?”) and enterprise Web Beans (according to the rules of Section 3.3.2, “Which EJBs are enterprise Web Beans?”) deployed and/or declared in these locations and searches the implementation classes for producer methods, producer fields, disposal methods and observer methods declared using annotations.

The container discovers Web Beans, disposal methods and observer methods defined using XML by parsing the `web-beans.xml` files according to the rules of Chapter 10, *XML based metadata*.

The container validates the Web Bean classes and metadata and aborts deployment if any definition errors exist, as defined in Section 12.1, “Definition errors”.

Next, the container determines which Web Beans, interceptors and decorators are enabled, according to the rules defined in Section 2.5.6, “Enabled deployment types”, Section 7.2.7, “Interceptor enablement and ordering” and Section 7.3.5, “Decorator enablement and ordering”, taking into account any `<Deploy>`, `<Interceptors>` and `<Decorators>` declarations in the `web-beans.xml` files.

Finally, the container creates and registers `Bean` objects (that implement the rules of Chapter 6, *Web Bean lifecycle*) and `Observer` objects.

- For each enabled Web Bean that is not an interceptor or decorator, the container creates an instance of `Bean`, and registers it by calling `Manager.addBean()`.
- For each enabled interceptor, the container creates an instance of `Interceptor` and registers it by calling `Manager.addInterceptor()`.
- For each enabled decorator, the container creates an instance of `Decorator` and registers it by calling `Man-`

ager.addDecorator().

- For each observer method of an enabled Web Bean, the container creates an instance of `Observer` that implements the rules of Section 8.5.7, “Observer object for an observer method” and registers it by calling `Manager.addObserver()`.

The container validates the Web Bean dependencies and specialization and aborts deployment if any deployment problems exist, as defined in Section 12.2, “Deployment problems”.

11.2. Web Bean registration

The `Manager` API provides methods for registering a new Web Bean with the container.

```
public interface Manager {

    public Manager addBean(Bean<?> bean);

    public Manager addInterceptor(Interceptor interceptor);

    public Manager addDecorator(Decorator decorator);

    ...

}
```

These methods may be called at any time by the application or third-party framework.

The `Manager` API also provides a method for dynamically validating all constraints that are defined by this specification to cause `DeploymentExceptions`:

```
public interface Manager {

    public Manager validate();

    ...

}
```

It is recommended that any application or third-party framework calls `validate()` after dynamically registering Web Beans.

11.3. Providing additional XML based metadata

The `Manager` API provides a method that allows the application or third-party framework to provide additional XML based metadata specified in a file other than `web-beans.xml`.

```
public interface Manager {

    public Manager parse(InputStream xmlStream);

    ...

}
```

The container parses the XML stream according to the rules of Chapter 10, *XML based metadata*.

This method may be called at any time by the application or third-party framework.

11.4. Initialization event

Third party frameworks and application components may require notification that the deployment is complete. The container must fire an event when it has fully completed the Web Bean discovery process and all other initialization. The container must not allow any request to be processed by the deployment until all observers of this initialization event return.

The event object must be the `Manager` object, and the event must have the following binding type:

```
@BindingType
@Retention(RUNTIME)
```

```
@Target( { FIELD, PARAMETER })
public @interface Initialized {}
```

Any Web Bean may observe this event.

```
public void managerInitialized(@Observes @Initialized Manager manager) { ... }
```

A third party framework might take advantage of this event to register Web Beans and interceptors with the container.

The request and application contexts are active when the initialization event is fired.

11.5. Child containers

Web Bean definitions may be scoped to a *child container*. The Web Beans specification only provides a programmatic API for defining child containers, since this feature is intended for use with third-party orchestration frameworks that integrate with Web Beans.

The `Manager` API provides a method for creating a child container:

```
public interface Manager {

    public Manager createChildManager();

    ...

}
```

A child container inherits all Web Beans, interceptors, decorators, observers, and contexts defined by its direct and indirect parent containers:

- every Web Bean belonging to a parent container also belongs to the child container, is eligible for injection into other Web Beans belonging to the child container and may be obtained by dynamic lookup via the child container,
- every interceptor and decorator belonging to a parent container also belongs to the child container and may be applied to any Web Bean belonging to the child container,
- every observer belonging to a parent container also belongs to the child container and receives events fired via the child container, and
- every context object belonging to the parent container also belongs to the child container.

Web Beans and observers may be registered with a child container by calling `addBean()` or `addObserver()` on the `Manager` object that represents a child container.

Web Beans and observers registered with a container are visible only to that container and its children—they are never visible to direct or indirect parent containers, or to other children of the parent container:

- a Web Bean registered with the child container is not available for injection into any Web Bean registered with a parent container,
- a Web Bean registered with a child container is not available for injection into a Servlet or EJB bean,
- a Web Bean registered with a child container may not be obtained by dynamic lookup via the parent container, and
- an observer registered with the child container does not receive events fired via a parent container.

If a Web Bean registered with a child container has the API type and all binding types of some injection point of some Web Bean registered with a direct or indirect parent container, a `DeploymentException` is throw by the container when `validate()` is called upon the `Manager` object that represents the child container.

Interceptors and decorators may not be registered with a child container. The `addInterceptor()` and `addDecorator()` methods throw `UnsupportedOperationException` which called on a `Manager` object that represents a child container.

11.5.1. Current container

A child container may be associated with the current context for a normal scope by calling `setCurrent()`, passing the normal scope type:

```
public interface Manager {

    public Manager setCurrent(Class<? extends Annotation> scopeType);

    ...

}
```

If the given scope is inactive when `setCurrent()` is called, a `ContextNotActiveException` is thrown. If the given scope is not a normal scope type, an `IllegalArgumentException` is thrown.

All EL evaluations (as defined ???), all calls to any injected `Manager` object or `Manager` object obtained via JNDI lookup (as defined by Section 5.8, “The Manager object”), all calls to any injected `Event` object (as defined in Section 8.6, “The Event interface”) and all calls to any injected `Instance` object (as defined by Section 5.9, “Dynamic lookup”) are processed by the *current container*:

- If the root container has no active normal scope such that the current context for that scope has an associated child container, the root container is the current container.
- If the root container has exactly one active normal scope such that the current context for that scope has an associated child container, that child container is the current container.
- Otherwise, there is no well-defined current container, and the behavior is undefined. Portable frameworks and applications should not depend upon the behavior of the container when two different current contexts have an associated child container

A Web Bean registered with a child container is only available to Unified EL expressions that are evaluated when that container or one of its children is the current container.

Chapter 12. Exceptions

Exceptions thrown by the container fall into three groups:

- Definition errors—occur when a single Web Bean definition violates the rules of this specification
- Deployment problems—occur when there are problems resolving dependencies, or inconsistent specialization, in a particular deployment
- Execution errors—occur at runtime

Definition errors are *developer errors*. They may be detected by tooling at development time, and are also detected by the container at deployment time. If a definition error exists in a deployment, the deployment will be aborted by the container.

Deployment problems are detected by the container at deployment time. If a deployment problem exists in a deployment, the deployment will be aborted by the container.

Execution errors may not be detected until they actually occur at runtime.

All exceptions defined by this specification are runtime exceptions.

12.1. Definition errors

Definition errors are represented by instances of `DefinitionException` and its subclasses.

```
public class DefinitionException extends RuntimeException {

    public DefinitionException(String message) { ... }

}
```

This specification defines the following subclasses:

- `NonexistentTypeException`
- `NonexistentMemberException`
- `NonexistentFieldException`
- `NonexistentMethodException`
- `NonexistentConstructorException`

container implementations may define their own subclasses of `DefinitionException`, and throw an instance of a subclass anywhere that this specification requires a `DefinitionException` to be thrown.

12.2. Deployment problems

Deployment problems are represented by instances of `DeploymentException` and its subclasses.

```
public class DeploymentException extends RuntimeException {

    public DeploymentException(String message) { ... }

}
```

This specification defines the following subclasses:

- `UnsatisfiedDependencyException`
- `AmbiguousDependencyException`
- `UnserializableDependencyException`

- `NullableDependencyException`
- `UnproxyableDependencyException`
- `InconsistentSpecializationException`

container implementations may define their own subclasses of `DeploymentException`, and throw an instance of a subclass anywhere that this specification requires a `DeploymentException` to be thrown.

12.3. Execution errors

Execution errors are represented by instances of `ExecutionException` and its subclasses.

```
public class ExecutionException extends RuntimeException {

    public ExecutionException(String message) { ... }

}
```

This specification defines the following subclasses:

- `CreationException`
- `IllegalProductException`
- `ObserverException`
- `DuplicateBindingTypeException`
- `ContextNotActiveException`