# JSR 299: Web Beans

Web Beans Expert Group

Version: Early Draft Review

# Table of Contents

# Chapter 1. Architecture

Web Beans provides a unifying component model for Java EE by defining:

- A programming model for stateful, contextual components, where metadata may be defined using either annotations or XML deployment descriptors. This component model is compatible with both EJB 3.0 and JavaBeans.
- A sophisticated, typesafe dependency injection mechanism.
- Integration with the Unified Expression Language (EL).
- Support for method and component lifecycle interceptors.
- An event notification model.
- A facility for overriding API implementations at deployment time.
- A facility for configuring components via XML.
- An extensible context model.
- A web *conversation* context in addition to the three standard web contexts defined by the Java Servlet specification.
- Conversation-scoped JPA extended persistence context management.
- A deployment and packaging model compatible with existing Java EE standards.

Web Beans is compatible with Java EE 5 and above.

In particular, Web Beans allows EJB 3.0 components to be used as JSF managed beans, thus unifying the the component models of EJB and JSF and significantly simplifying the programming model when EJB and JSF are used together.

## 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 — the *Web Bean application* — 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 *Web Beans container*. Both the Web Bean application and the Web Beans container are written to comply with Java EE contracts and may take advantage of the functionality provided by Java EE.

The Web Beans container may be provided by the Java EE container vendor as *integrated* functionality of the Java EE container. Alternatively, the Web Beans container may be provided by some third-party as a *plugin* Web Beans container which may be integrated into other Java EE containers.

## 1.2. Supported environments

All plugin Web Beans containers are required to support any Java EE 6.0 compliant container. An integrated

Web Beans container is not required to support any environment other than that in which it is integrated.

A compliant standalone Web Beans container may optionally support Java EE 5.0. Certain functionality defined in this specification is optional when the Web Beans container executes in a Java EE 5.0 environment. This is the case only when explicitly noted in this specification. All other functionality defined by this specification must be supported by a compliant standalone Web Beans container that supports Java EE 5.0 when it executes in the Java EE 5.0 environment.

## 1.3. Relationship to other specifications

This specification defines a complete, standalone component model. However, Web Beans lacks certain useful functionality already defined by other specifications. In particular, Web Beans does not specify enterprise aspects such as declarative transaction management or declarative security. Nor does Web Beans specify any kind of presentation or orchestration technology.

Rather, Web Beans leverages existing specifications and integrates cleanly with the functionality provided by these specifications.

In addition, this specification defines an SPI that allows a Web Beans container to be integrated with alternative technologies, for example other web presentation technologies.

### 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 usually contextual. References to stateful component instances must be explicitly passed between clients and stateful instances must be explicitly destroyed by the application.

An EJB session bean may be declared as a Web Bean component. In this case, the session bean is a contextual object. It is bound to a context and available to all components that execute in that context. When the context ends it is automatically destroyed by the Web Beans container.

For an EJB session bean that is also a Web Bean, the EJB container provides the services defined by the EJB specification, and the Web Beans container provides the contextual lifecycle management defined by Web Beans. The Web Beans container integrates with the EJB container via standard EJB and Java EE APIs.

A Web Bean is not required to be an EJB.

### 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.

Web Beans components may be used in place of JSF managed beans in a JSF application. In this case, the application may take advantage of the more sophisticated context and component model provided by Web Beans. Even better, the Web Beans may be EJB session beans, allowing direct use of EJB components in JSF.

The Web Beans container integrates with JSF via standard JSF APIs.

An application that uses Web Beans need not use JSF.

### 1.3.3. Relationship to Java Servlets

Web Bean components may be called by a Servlet. The Web Beans container integrates with the Servlet engine via standard APIs defined by the Java Servlets specification.

### 1.3.4. Relationship to Common Annotations for the Java Platform

Web Beans supports some of the functionality defined by Common Annotations for the Java Platform. For Web Beans which are not EJB session beans, this functionality is implemented by the Web Beans container.

# Chapter 2. The Web Beans component model

A Web Bean *component* is a source of contextual objects which define application state and/or logic. These objects are called *component instances* or *instances of the Web Bean*. The lifecycle of a Web Bean instance is under the control of the Web Beans container, which determines when the instance is created and destroyed. 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 component comprises:

- A component type
- Either a bean implementation class or a producer method
- A set of API types
- A (possibly empty) set of binding annotation types
- A scope
- A component name

We will often write *Web Bean component* to explicitly distinguish the component from the implementation class and from instances of the component. However, when not explicit, the term *Web Bean* should be understood to mean a Web Bean component.

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

- lifecycle management and scoping to a particular Web Beans context
- scoped resolution by type and binding annotation type when injected into a Java-based client
- scoped resolution by name when used in a Unified EL expression
- automatic instantiation, when injected into a client, or used in an EL expression
- automatic injection of other Web Bean instances
- lifecycle callbacks
- method and lifecycle interception
- event notification

*Open issue: should Web Beans provide concurrency control?*

However, if the application directly instantiates an implementation class of a Web Bean component, instead of letting the container perform instantiation, these capabilities will not be available to that particular class instance.

If the application requires full control over instantiation of a Web Bean, the component may be defined by declaring a *producer method* that is invoked by the Web Beans container to instantiate the component. In this case, we describe the component as a *producer method component*.

A Web Bean component may be defined and deployed using any one of the following mechanisms:

- annotating the bean implementation class with any component type annotation and deploying it to an archive in the web application classpath that includes a `web-beans.xml` file
- explicitly declaring the component in a `web-beans.xml` file
- annotating a method of any other Web Bean with the `@Produces` annotation
- annotating an injection point of any other Web Bean with the `@New` annotation

See Section 8.1, “Web Bean component discovery” for further details about component discovery.

## 2.1. Component types

A component type allows the Web Beans container to identify which classes in the classpath are Web Bean component implementation classes, and which components should be *enabled* for use in a particular deployment of the system. The component type also determines the *precedence* of a component.

The set of component types is extensible.

### 2.1.1. Built-in component types

There are two standard component types defined by Web Beans: `@Component` and `@Standard`. All standard Web Beans components provided by the Web Beans container are defined using the `@Standard` component type. Application components may be defined using the `@Component` component type.

### 2.1.2. Defining new component types

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

Applications and third-party frameworks may define their own component types. For example, the following component type might be used for components which are used only in a particular deployment of the application:

```
@ComponentType
@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface MelbourneOffice {}
```

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

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

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

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

### 2.1.3. Declaring the component type of a component using annotations

A Web Beans component type annotation may only be applied to Web Beans implementation classes or producer methods. The annotation determines the component type of the component.

A Web Bean implementation class or producer method may specify only one component type. If an implementation class or producer method specifies multiple component type annotations, an exception is thrown by the Web Beans container at initialization time.

This component has the component type `@Component`:

```
@Component
public class Order {}
```

This component has the component type `@Mock`:

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

By default, if no component type annotation is explicitly specified, a producer method component inherits the component type of the Web Beans component upon which it is defined.

This producer method component has the component type `@Component`:

```
@Component
public class Login {

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

}
```

This producer method component has the component type `@Mock`:

```
@Component
public class MockLogin implements Shop {

   @Produces @Mock
   public User getMockUser() { ... }

}
```

### 2.1.4. Declaring the component type of a component using XML

Alternatively, if the Web Beans component is declared in `web-beans.xml`, the component type is specified using `<type>`, and the implementation class need not have a component type annotation:

```
<component>
    <class>org.mydomain.melbourne.MelbourneOrder</class>
    <type>org.mydomain.melbourne.MelbourneOffice</type>
</component>
```

For Web Bean components declared in `web-beans.xml`, component type annotations appearing on the implementation class or producer method are ignored. If no component type is specified using using `<type>`, the component type is assumed to be `@Component`.

If a Web Bean component is declared in `web-beans.xml` and a component type annotation appears on the bean implementation class or producer method, the Web Beans container assumes that two different components exist!

### 2.1.5. Component type enablement and precedence

In a particular deployment, only some component types are *enabled*. Components declared with a component type that is not enabled are not available to the resolution algorithms defined in Chapter 3, *Injection and EL resolution*.

In a particular deployment, all enabled component types are strongly ordered in terms of *precedence*. The precedence of a component type is used by the resolution algorithms.

By default only the `@Standard` and `@Component` component types are enabled, and `@Standard` has a lower precedence.

See Section 8.1.1, “Enabled component types” for more information about component type enablement.

## 2.2. Web Bean implementation class

A Web Bean implementation class defines the state and behavior of the Web Bean component. The implementation class is a non-final, non-abstract Java class with no final methods. In particular, any EJB 3.0 session bean class may be a Web Bean implementation class.

An inner class may not be a Web Bean implementation class.

*Open issue: Web Beans should support message driven beans as Web Beans components.*

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

A subclass of a Web Bean implementation class is not, by default, the implementation class for any Web Bean component. It must be explicitly declared as a Web Bean component (either via component type annotation, `@New` or XML) if it is to be a Web Bean.

### 2.2.1. Declaring an implementation class using annotations

When a component is declared using annotations, the annotations are applied directly to the implementation class.

Here are some example Web Bean implementation classes, all of the standard `@Component` component type:

```
@Component
class Shop { .. }
```

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

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

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

```
public class LoginActionImpl implements LoginAction { ... }
```

This Web Bean is a "mock object" that overrides the implementation of `LoginAction` when running in an integration testing environment:

```
@Stateless
@Mock
@Named("loginAction")
public class LoginActionMock extends LoginActionImpl { ... }
```

## 2.2.2. Declaring an implementation class using XML

For Web Bean components declared in `web-beans.xml`, the implementation class is specified using the `<class>` element:

```
<component>
    <class>org.mydomain.Order</class>
</component>
```

## 2.2.3. Component constructors

When the Web Beans container instantiates a Web Bean that is not an EJB, it calls the *component constructor*.

- If the implementation class does not explicitly declare a constructor, the default constructor is the component constructor.
- If the implementation class declares exactly one constructor, that constructor is the component constructor.
- If the implementation class declares more than one constructor, exactly one of these constructors must be annotated `@In` or have a parameter annotated with a binding type. This constructor is the component constructor.

If the component constructor has parameters, the container uses the instance resolution procedure defined in Section 3.1, “Instance resolution” to determine a value for each of the parameters and calls the constructor with those parameter values.

```
@ConversationScoped @Component
public class Order {

   private Product product;
   private User customer;

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

}
```

If the implementation class declares multiple constructors and either:

- no constructor is annotated `@In` and no constructor is annotated with a binding type, or
- multiple constructors are annotated `@In` or have a parameter annotated with a binding type

then an exception is thrown by the Web Beans container at initialization time.

The application may call component constructors directly. However, in this case, no parameters will be passed to the constructor by the container; the returned object is not bound to any context; and the lifecycle of the new instance is not managed by the Web Beans container.

## 2.2.4. Component remove methods

When the Web Beans container destroys an EJB stateful session bean, it calls the *component remove method*.

- If the implementation class declares exactly one EJB `@Remove` method, that remove method is the component remove method.
- If the implementation class declares more than one EJB `@Remove` method, exactly one of these method must be annotated `@Destroys`. This remove method is the component remove method.

If the component remove method has parameters, the container will use the instance resolution procedure to determine a value for each of the parameters and call the method with those parameter values.

```
@ConversationScoped @Stateful @Component
public class Order {

   @Remove @Destroys
   public remove(Log log)
   {
       ...
   }

}
```

If the stateful session bean implementation class declares multiple EJB remove methods and either:

- the `@Destroys` annotation does not appear, or
- if multiple remove methods are annotated `@Destroys`

then an exception is thrown by the Web Beans container at initialization time. If the stateful session bean implementation class does not declare any EJB remove method, an exception is thrown by the Web Beans container at initialization time.

The application may call a component remove method, or any other EJB remove method directly, but in this case no parameters will be passed to the method by the container. However, whenever any remove method of a Web Bean component instance is called by the application, the Web Beans container *must* remove the instance from the context with which it is associated.

## 2.2.5. Injected fields

An *injected field* is a non-static, non-final field of a Web Bean implementation class that is annotated with any binding type or `@In`.

Injected fields are initialized by the container immediately after instantiation. The container uses the instance resolution procedure to determine a value for each injected field.

```
@ConversationScoped @Component
public class Order {
   @In Product product;
   @Current User customer;

}
```

### 2.2.6. Injector methods

An *injector method* is a non-static method of a Web Bean implementation class that:

- is annotated `@In` or has a parameter annotated by a binding type and
- is not annotated `@Produces` or `@Destroys` and
- does not have a parameter annotated `@Disposes` or `@Observes`.

Injector methods are called by the container immediately after injected fields have been initialized by the container.

If the implementation class is an EJB session bean, the injector method is *not* required to be a business method of the session bean.

If the injector method has parameters, the container will use the instance resolution procedure to determine a value for each of the parameters and call the producer method with those parameter values.

```
@ConversationScoped @Component
public class Order {

   private Product product;
   private User customer;

   @In void setOrder(Product product)
   {
       this.product = product;
   }

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

}
```

An injector method may have multiple (or zero) parameters.

```
@ConversationScoped @Component
public class Order {

   private Product product;
   private User customer;

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

}
```

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

The application may call injector methods directly, but in this case no parameters will be passed to the method by the container.

## 2.3. 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 components, or
- the concrete type of the objects to be injected may vary at runtime, or
- the objects to be injected may be null, or
- the objects require some custom initialization that is not performed by the component constructor.

A producer method must be a non-static method of a Web Bean. If the Web Bean implementation class is an EJB session bean, the producer method must be a business method of the session bean.

The application may call producer methods directly. In this case no parameters will be passed to the producer method by the container; the returned object is not bound to any context; and its lifecycle is not managed by the Web Beans container.

A producer method may return a null value.

A component may declare multiple producer methods.

### 2.3.1. Declaring a producer method using annotations

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

```
@Component
@Stateless
public class ShopBean implements Shop {

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

   @Produces
   public PaymentProcessor getPaymentProcessor() { ... }

}
```

### 2.3.2. Declaring a producer method using XML

Alternatively, a producer method may be defined in XML, by specifying a Unified EL method expression:

```
<web-beans>
    <component>
        <name>products</name>
        <producer>#{shopBean.getProducts}</producer>
    </component>
</web-beans>
```

Note that an XML-based component definition may not specify both an implementation class and a producer method expression. If the component definition specifies a producer method expression, the component is considered a producer method component.

### 2.3.3. Producer method parameters

If the producer method has parameters, the container will use the instance resolution procedure to determine a value for each of the parameters and call the producer method with those parameter values.

```
@Component
public class OrderFactory {

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

}
```

### 2.3.4. 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 non-static method of a Web Bean. If the Web Bean is a session bean, the producer method must be a business method of the session bean.

A disposal method must have a parameter annotated `@Disposes`. If this parameter resolves to a producer method component according to the typesafe resolution algorithm, the container must call this method when destroying an instance returned by that producer method.

```
@Component
public class CurrentEntityManager {

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

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

}
```

The container will use the instance resolution procedure to determine a value for each of the parameters of a disposal method and call the disposal method with those parameter values.

*Open issue: what happens when the application calls a disposal method directly? For consistency with the rule defined for component remove method, it should remove the object from the context.*

If there are multiple disposal methods that resolve to the same producer method component, an exception will be thrown by the container at initialization time.

*Open issue: this does not work well with overriding. There needs to be precedence-aware rules for this.*

A component may declare multiple disposal methods.

## 2.4. Web Bean API types

A Web Bean API type defines a client-visible type of the Web Bean component. A Web Bean component may have multiple API types. The set of API types depends upon whether the Web Bean component is a producer method component:

- If the Web Bean implementation class is not a session bean, the set of API types includes the implementation class, all superclasses and all interfaces implemented directly or indirectly.
- If the Web Bean implementation class is a session bean, the set of API types includes all local interfaces of the session bean and their superinterfaces. Remote interfaces are not included in the set of API types.
- If the Web Bean component is a producer method component, the API types include the method return type and all interfaces implemented directly or indirectly. If the method return type is a concrete class, the API types also include all superclasses of the method return type.

In the examples from Section 2.2.1, “Declaring an implementation class using annotations”, the first Web Bean has the API type `Shop`. The second Web Bean has the API types `ProductList` and `DataModel`. The third Web Bean has the API types `PaymentProcessorImpl` and `PaymentProcessor`. The fourth Web Bean has the API type `LoginAction` (the local interface of the stateless session bean).

An API type may be a parameterized type with an actual type parameter. For the purposes of the typesafe resolution algorithm, API types with are considered the identical by the Web Beans 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.

*Open issue: is this too restrictive? We should probably support API types with a type variable, considering them a match for any injection point which specifies an actual type for the variable.*

## 2.5. Binding annotations

For a given API type, there may be multiple Web Bean components which implement the type. In this case, the API type alone is not sufficient to uniquely identify the Web Bean component to inject into a Java-based client. Thus, the client must distinguish the particular component it requires using a *binding annotation*. The Web Beans container inspects the binding annotations and type of the injected attribute to determine the Web Bean instance to be injected.

### 2.5.1. Defining binding annotations

A binding annotation is a Java annotation defined as `@Target({METHOD, FIELD, PARAMETER, TYPE})` and `@Retention(RUNTIME)`. All binding annotations must specify the `@BindingType` meta-annotation.

For example:

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

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

A binding annotation may define annotation members.

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

### 2.5.2. Declaring the binding annotation types for a component using annotations

A binding annotation is declared by annotating the implementation class or producer method. For example, the following Web Beans both implement the API type `PaymentProcessor`, but specify different binding annotations:

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

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

A producer method may also declare binding annotations:

```
@Component
@Stateless
public class ShopBean implements Shop {

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

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

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

}
```

Any Web Bean component may declare multiple binding annotations.

### 2.5.3. Declaring the binding annotation types for a component using XML

If the Web Beans component is declared in `web-beans.xml`, binding types may be specified using `<binding>`:

```
<component>
    <class>org.mydomain.SynchronousPaymentProcessor</class>
    <type>javax.webbeans.Component</type>
    <binding>org.mydomain.Synchronous</binding>
</component>
```

If a producer method expression is specified, all binding types must be explicitly specified by `<binding>` elements.

Otherwise, if no `<binding>` element is specified, the binding annotations that appear on the implementation class are used. If a `<binding>` element does appear, the binding annotations appearing on the implementation class are ignored and all binding types must be explicitly specified by `<binding>` elements.

### 2.5.4. Using binding annotations on injected fields

Binding annotations are applied to injected fields to determine the component that is injected, according to the typesafe resolution algorithm defined in Section 3.1.1, “Typesafe resolution algorithm”.

For example, when the Web Beans 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;
```

For the case of producer methods, the producer method that is called depends upon the component identified by the binding annotations:

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

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

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

### 2.5.5. Using binding annotations on method parameters

Binding annotations are applied to method parameters to determine the component that is passed when the method is called by the container, according to the typesafe resolution algorithm.

For example, when the Web Beans container encounters the following producer method, an instance of `SynchronousPaymentProcessor` will be passed to the first parameter and an instance of `AsynchronousPaymentProcessor` will be passed to the second parameter:

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

### 2.5.6. The `@New` binding annotation

The built-in binding annotation `@New` may be applied to any injection point declared with a concrete Java type. It may not appear in conjunction with any other binding annotation. It may not be applied to an injection point of abstract or interface type. No component defined using annotations or XML may declare `@New` as a binding annotation.

When `@New` appears on an injection point, a component is implicitly defined with scope `@Dependent`, component type `@Component`, `@New` as the only binding annotation, no component name, and where the API type and implementation class are both the type of the injection point.

## 2.6. Component scopes

All Web Bean components have a *scope*. The scope of a Web Bean component determines the lifecycle of its instances, and which instances of the component are visible to instances of other components.

The set of scope types is extensible.

### 2.6.1. Built-in scope types

There are several standard scope types defined by Web Beans. The `@RequestScoped`, `@ApplicationScoped` and `@SessionScoped` annotations represent the standard scopes defined by the Java Servlets specification. The `@ConversationScoped` annotation represents the Web Beans conversation scope defined in Section 5.3.4, “Conversation context lifecycle”. In addition, there is the `@Dependent` pseudo-scope for dependent objects, as defined in Section 5.2, “Dependent pseudo-scope”.

### 2.6.2. Defining new scope types

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

For example, the following annotation declares a "method scope":

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

### 2.6.3. Declaring the component scope using annotations

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

A Web Bean implementation class or producer method may specify at most one scoping annotation. If an implementation class or producer method specifies multiple scoping annotations, an exception is thrown by the Web Beans container at startup time.

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

```
@ConversationScoped
@Component
public class ProductList implements DataModel { ... }
```

```
@Component
@Stateless
public class ShopBean implements Shop {

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

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

}
```

Likewise, a Web Bean with the custom method scope may be declared by annotating it with the `@MethodScoped` annotation:

```
@MethodScoped
@Component
public class Message {
    ...
}
```

### 2.6.4. Declaring the component scope using XML

If the Web Beans component is declared in `web-beans.xml`, the scope may be specified using `<scope>`:

```
<component>
    <class>org.mydomain.ProductList</class>
    <type>javax.webbeans.Component</type>
    <scope>javax.webbeans.ConversationScoped</scope>
</component>
```

If an implementation class with a scope type annotation is specified and if no `<scope>` element is explicitly specified, the scope defined by the scope type annotation is used.

### 2.6.5. Default component scopes

When no scope is explicitly declared by annotating the implementation class or by the `<scope>` element, the scope is defaulted.

*Open issue: should the default be `@Dependent`, or `@RequestScoped`? If we make `@RequestScoped` the default, we should rename `@RequestScoped` to `@DefaultScoped`.*

If a producer method does not explicitly declare a scope, the scope defaults to the `@Dependent` pseudo-scope.

Stateless session beans always belong to the `@Dependent` pseudo-scope, and may not specify any other scoping annotation.

## 2.7. Component names

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

In certain circumstances, multiple components may share the same name.

### 2.7.1. Default component names

By default, the name of a component defaults to the unqualified class name of the Web Beans implementation class, after converting the first character to lower case.

For example, the default component name of the `ProductList` component is `productList`.

A producer method component name defaults to the method name, unless the method follows the JavaBeans property getter naming convention, in which case the name defaults to the JavaBeans property name.

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

```
@Component
@Stateless
public class ShopBean implements Shop {

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

}
```

### 2.7.2. Declaring the component name using annotations

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

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

The `@Named` annotation is a binding annotation.

*Open issue: this allows typesafe injection by name, but only if the component specifies @Named explicitly. Should we allow injection by name for components with defaulted names?*

### 2.7.3. Declaring the component name using XML

If the Web Beans component is declared in `web-beans.xml`, the name declared by the implementation class may be specified using `<name>`:

```
<component>
    <class>org.mydomain.ProductList</class>
    <name>products</name>
</component>
```

If a producer method expression is specified, and if no `<name>` is explicitly specified, the component has no name.

If an implementation class with a `@Named` annotation is specified, and if no `<name>` element is explicitly specified, the name specified by the `@Named` annotation is used.

### 2.7.4. Using component names in EL

The name may be used in unified EL expressions, for example:

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

## 2.8. XML based configuration

*This functionality is yet to be specified.*

## 2.9. Additional examples

This example shows a full XML component declaration:

```
<component>
    <class>com.mydomain.myapp.AsynchronousCreditCardPaymentProcessor</class>
    <type>javax.webbeans.Component</type>
    <scope>javax.webbeans.SessionScoped</scope>
    <binding>com.mydomain.myapp.PayBy(com.mydomain.myapp.PaymentType.CREDIT_CARD)</binding>
    <binding>com.mydomain.myapp.Asynchronous</binding>
</component>
```

This example shows use of annotations defined by the Common Annotations and EJB specifications.

```
@SessionScoped @Component
@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;

    void setUser(@Current User customer) {
        this.customer = customer;
    }

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

    ...

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

    @Remove @Destroys void destroy() {}

}
```

# Chapter 3. Injection and EL resolution

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

The `Container` interface provides operations for resolving a component by type or name.

## 3.1. Instance resolution

When injecting a component instance, the Web Beans container uses the following algorithm. Each time an instance must be injected, the Web Beans container must:

- Identify the component by calling `Container.resolveByType()`, passing the type and binding annotations of the injection point.
- If necessary, instantiate a *scope adaptor* for the resulting component. In this case, the scope adaptor is the object that is injected.
- Otherwise, if no scope adaptor is required:
  - obtain the context object by calling `Container.getContext()`, passing the component scope.
  - Finally, obtain the current instance of the component by calling `Context.get()`, passing the `Component` object representing the component. The object returned by `get()` will be injected.

### 3.1.1. Typesafe resolution algorithm

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

Typesafe resolution usually occurs at container initialization time, allowing the container to warn the user if components have unsatisfied dependencies.

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

```
public interface Container {

    public <T> Component<T> resolveByType(Class<T> apiType, Annotation... bindingTypes);

    ...

}
```

For example:

```
Component<PaymentProcessor> component =
    container.resolveByType(PaymentProcessor.class, new Synchronous() {});
```

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

- First, the container inspects the type of the injection point and identifies the set of *matching* enabled components which have this type as an API type.
- Next, the container inspects the annotations which appear on the injection point, and identifies the binding annotations. The container narrows the set of matching components to just those which (a) declare all of the binding annotations specified at the injection point and (b) specify the same annotation member values for all binding annotations specified at the injection point where the member is not annotated `@NonBinding`.
- If at least one remaining component declares *only* the binding annotations specified at the injection point, the container narrows the set of components to those which declare only those binding annotations.
- Next, the container examines the component types of the matching components, and narrows the matching set to the components with the highest precedence component type that occurs in the set. If exactly one component remains, the resolution results in that component.
- Otherwise, an exception is thrown by the container.

*Open issue: these rules do not allow overriding of several components with the same API type but different binding annotations with a single component that implements the API and supports all the same bindings, due to the third rule. Is this an important feature that should be supported?*

#### 3.1.1.1. Binding annotations with members

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

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

```
@PayBy(CREDIT_CARD) @Component
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();
}
```

#### 3.1.1.2. Multiple binding annotations

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

```
@Synchronous @PayBy(CHEQUE) @Component
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 component must declare *all* of the binding annotations that are specified at the injection point to be considered a candidate for injection. If more than one component is a candidate, components which declare *exactly* the same binding annotations that are specified at the injection point are preferred over other candidates.

### 3.1.2. Scope adaptors

The Web Beans container must guarantee that when any component instance invokes an injected component instance, the invocation is always processed by the current instance (see Section 5.1, “Contexts”) of the injected component. In certain scenarios, for example if a request scoped component is injected into a session scoped component, this rule requires that the container must inject a *scope adaptor* object. A scope adaptor implements or extends the type of the injected attribute and delegates all method calls to the current instance of the injected component.

If a scope adaptor is not required in order to enforce the rule, the Web Beans container is not required to inject an adaptor, and may directly inject the current instance.

## 3.2. EL name resolution

The Web Beans container provides a Unified EL `ELResolver`. When this resolver is called with a null base object, the container:

- Identifies the component by calling `Container.resolveByName()`, passing the name.
- Obtains the context object by calling `Container.getContext()`, passing the component scope.
- Obtains an instance of the component by calling `Context.get()`, passing the `Component` instance representing the component.

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

### 3.2.1. Name resolution algorithm

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

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

```
public interface Container {

    public Component resolveByName(String name);

    ...

}
```

For example:

```
Component component = container.resolveByName("paymentProcessor");
```

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

- The container identifies the set of *matching* enabled components which have a component name matching the name used in the EL expression. If no enabled component has this name then the result of resolution is a null value.
- Next, the container examines the component types of the matching components and narrows the matching set to the components with the highest precedence component type that occurs in the set. If exactly one component remains, the resolution results in that component.
- Otherwise, an exception is thrown by the container.

The name resolution algorithm usually occurs at runtime.

### 3.2.2. Integration with Unified EL

In a Servlet or JSF application, the Web Beans container must register the `ELResolver` with the web container.

If `WEB-INF/web.xml` does not contain any `<servlet>` elements that reference `javax.faces.webapp.FacesServlet`, the Web Beans container will register the `ELResolver` by calling `JspApplicationContext.addELResolver()` at initialization time.

Otherwise, the Web Beans container uses standard JSF APIs to register the `ELResolver` with JSF at initialization time.

# Chapter 4. Component lifecycle

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

The actual mechanics of component creation and destruction varies according to what kind of component it is. To create a session bean, the container obtains an instance from the EJB container. To create a producer method component instance, the container calls the producer method. Otherwise, the container calls the component constructor.

*Open issue: we should also open up an SPI to allow frameworks to register a custom `Lifecycle` strategy with the Web Beans container, thereby supporting new kinds of components beyond Java classes, EJBs and producer methods.*

In addition to the capabilities defined by this specification, Web Bean components also support the following functionality defined by the Common Annotations for the Java Platform and Enterprise JavaBeans specifications:

- 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`

Of course, Web Beans which are also EJB session beans may take advantage of all additional functionality defined by the EJB specification.

## 4.1. Instances of producer method components

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 component. However, if the returned object is not an instance of another Web Bean component, the Web Beans 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 Bean components:

```
@Component @SessionScoped
public class PaymentStrategyProducer {

   private PaymentStrategyType paymentStrategyType;

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

   @Produces @Current
   public 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: return null;
      }
    }

}
```

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 component instances:

```
@Component @SessionScoped
public class PaymentStrategyProducer {

   private PaymentStrategyType paymentStrategyType;

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

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

}
```

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.

## 4.2. Component creation

When the Web Beans container injects dependencies or resolves EL names, and there is no existing instance of the component cached by the context object for the component scope, the context object automatically creates a new instance of the component by calling `Component.create()`.

```
public interface Component<T> {

    public T create();

    ...

}
```

The `create()` method performs the following tasks, in order:

- object instantiation
- additional injection

- `@PostConstruct` callback

In addition, any instances of `@Dependent` components (see Section 5.2, “Dependent pseudo-scope”) that were created and injected during these phases must be registered for later destruction.

### 4.2.1. Object instantiation

The mechanism used for object instantiation depends upon whether the Web Bean component is a producer method component, and upon whether the implementation class is a session bean.

#### 4.2.1.1. Instantiating a component by calling the component constructor

If the Web Bean implementation class is not an EJB, the container instantiates it by calling the component constructor.

For each constructor parameter, the container passes the object identified by the instance resolution algorithm.

#### 4.2.1.2. Instantiating an EJB session bean

If the Web Bean implementation class is an EJB session bean, the container instantiates it by looking up the bean in JNDI.

*Open issue: how does it know the JNDI name?*

#### 4.2.1.3. Instantiating a component by calling the producer method

If the Web Bean component is a producer method component, the container:

- obtains the current instance of the component which declares the producer method by calling `Context.get()`, passing the `Component` object representing that component, then
- invokes the producer method upon the current instance, passing the object identified by the instance resolution algorithm to each parameter.

The return value of the producer method is the new component instance to be returned by `Component.create()`.

If the producer method may return a null value, the `Component.create()` method returns null.

### 4.2.2. Additional injection

After instantiating the component, the container performs additional injection.

- First, 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. If the Web Bean is an EJB, the EJB container is responsible for injecting these attributes. Otherwise, the Web Beans container is responsible for injection of these attributes.
- Next, the container initializes the values of all injected fields. For each injected field, the container sets the value to the object identified by the instance resolution algorithm.
- Finally, the container calls all injector methods. For each injector method parameter, the container passes

the object identified by the instance resolution algorithm.

*Open issue: do we really need to support `@PersistenceContext` for non-EJB web beans?*

The container does not perform additional injection upon a component instance returned by a producer method. However, if the object returned by the producer method was an instance of some other component, the injection described above might have already have been performed.

### 4.2.3. `@PostConstruct` callback

After all injection has been performed upon a component instance that is not an EJB stateless session bean or a producer method return value, the `@PostConstruct` callback occurs. If the Web Bean is an EJB, the EJB container is responsible for this callback. Otherwise, the Web Beans container performs this callback, in accordance with the semantics defined by the Common Annotations for the Java Platform specification.

The Web Beans container does not perform the `@PostConstruct` callback upon a component instance returned by a producer method. However, if the object returned by the producer method was an instance of some other component, the `@PostConstruct` method might have already have been called.

### 4.2.4. Dependent instances

Any instances of components with `@Dependent` scope that were created and injected into the newly-created instance during the previous phases must later be destroyed. The container is responsible for registering these *dependent instances* for later destruction.

The container is permitted to destroy dependent instances at any time if these instances are no longer referenced by the application (excepting weak, soft and phantom references).

The container is required to destroy dependent instances after the instance upon which they are dependent is destroyed.

## 4.3. Component destruction

When a Web Beans context is destroyed, the context object automatically destroys any instances associated with that context by calling `Component.destroy()`.

```
public interface Component<T> {

    public void destroy(T instance);

    ...

}
```

The `destroy()` method performs the following tasks, in order:

- component remove method or disposal method call
- `@PreDestroy` callback
- destruction of dependent instances

### 4.3.1. Component remove method or disposal method call

If the Web Bean component is a producer method component with a disposal method, the disposal method must be called. Otherwise, if the implementation class is a stateful session bean, the component remove method must be called to remove the stateful bean.

*Open issue: disposal method and component remove method injection caveats.*

#### 4.3.1.1. Destroying a stateful session bean instance

If the component implementation class is an EJB stateful session bean, the Web Beans container is responsible for calling the component remove method. This causes the bean to be removed by the EJB container.

For each parameter of the component remove method, the container passes the object identified by the instance resolution algorithm.

#### 4.3.1.2. Disposing an instance returned by a producer method

If the Web Bean component is a producer method component, and if there is a disposal method for that component, the container:

- obtains the current instance of the component which declares the disposal method by calling `Context.get()`, passing the `Component` object representing that component, then
- invokes the disposal method upon the current instance, passing the object identified by the instance resolution algorithm to each parameter.

### 4.3.2. `@PreDestroy` callback

When a component instance that is not an EJB stateless session bean or a producer method return value is destroyed, the `@PreDestroy` callback occurs. If the Web Bean is an EJB, the EJB container is responsible for this callback. Otherwise, the Web Beans container performs this callback, as defined in the Common Annotations for the Java Platform specification.

The Web Beans container does not explicitly perform `@PreDestroy` callbacks upon a component instance returned by a producer method. However, if the object returned by the producer method was an instance of some other component, the `@PreDestroy` method might still be called.

### 4.3.3. Destruction of dependent instances

After an instance is destroyed, the container must destroy all *dependent instances* of that instance.

## 4.4. Interceptors

Web Beans components support interception as defined by the package `javax.interceptor`. Interceptors may be bound to a component using the `javax.interceptor.Interceptors` annotation, or by using a Web Beans *interceptor binding*.

### 4.4.1. Support for `@Interceptors`

Any Web Bean may declare interceptors using `@Interceptors`. If the Web Bean is an EJB session bean, the EJB container is responsible for calling interceptors declared using `@Interceptors`. Otherwise, the Web Beans container is responsible for calling the interceptors. In both cases, the semantics are fully defined by the EJB specification.

Interceptors declared using `@Interceptors` are called before interceptors declared using Web Beans interceptor bindings.

### 4.4.2. Interceptor bindings

As an extension to the functionality defined by the `javax.interceptor` package, Web Beans provides an alternative method of binding interceptors to components. Even when interceptors are bound to components using this mechanism, the interception semantics are defined by the EJB specification.

*Open issue: we need to feed this back to the EJB group, so that this functionality is available for all EJB session beans.*

*Open issue: can interceptors have Web Beans components injected? Can they have a scope?*

An *interceptor binding type* is a Java annotation defined as `@Target({TYPE, METHOD})` or `@Target(TYPE)` and `@Retention(RUNTIME)`. All interceptor binding types must also specify the `@InterceptorBindingType` meta-annotation.

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

A *Web Beans interceptor* is any interceptor that complies with the EJB specification and is also annotated `@Interceptor`. Web Beans interceptors must declare at least one interceptor binding type:

```
@Transactional @Interceptor
public class TransactionInterceptor {

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

}
```

A Web Beans interceptor for component lifecycle events may be bound to a component by annotating the implementation class with the interceptor binding types declared by the interceptor.

A Web Beans interceptor for method invocations may be bound to a component method by annotating the component method or implementation class with the interceptor binding types declared by the interceptor.

*Open issue: what happens when an interceptor for component lifecycle events is bound at the method level?*

In the following example, the `TransactionInterceptor` will be applied at the class level:

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

In this example, the `TransactionInterceptor` will be applied at the method level:

```
@Component
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.

Multiple interceptor classes may all be bound to the same interceptor binding type or types.

#### 4.4.2.1. Interceptors with multiple binding types

An interceptor class may specify multiple interceptor binding types, in which case the interceptor will be applied only to components with an implementation class that also declares all the binding types, and to component methods where all the binding types appear on either the method or implementation class.

Consider the following interceptor:

```
@Transactional @Action @Interceptor
public class TransactionalActionInterceptor {

   @AroundInvoke
   public void aroundInvoke() { ... }

}
```

This interceptor will be bound to all methods of this component:

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

The interceptor will also be bound to the `placeOrder()` method of this component:

```
@Transactional @Component
public class ShoppingCart {

   @Action
   public void placeOrder() { ... }

}
```

#### 4.4.2.2. Interceptor binding types with members

Interceptor binding types may have annotation members. The member value is used by the container to choose an interceptor.

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

```
@Transactional(requiresNew=true) @Interceptor
public class RequiresNewTransactionInterceptor {

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

}
```

```
@Transactional(requiresNew=true) @Component
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;
}
```

#### 4.4.2.3. Interceptor binding types with additional interceptor bindings

An interceptor binding type may be applied to another interceptor binding type. In this case, the interceptors bound to the first type will be also be bound to the second type.

```
@InterceptorBindingType
@Target({TYPE, METHOD})
@Retention(RUNTIME)
@Transactional(requiresNew=true)
public @interface Action {}
```

```
@Action @Component
public class ShoppingCart { ... }
```

#### 4.4.2.4. Declaring interceptor bindings using XML

Additional interceptor bindings may be declared in XML:

```
<interceptor>
   <class>com.mydomain.framework.TransactionInterceptor</class>
   <binding>com.mydomain.framework.Action(transactional=true)</binding>
</interceptor>
```

### 4.4.3. Interceptor enablement and ordering

By default, interceptors bound via interceptor binding types are not enabled. An interceptor class must be explicitly enabled by listing it in the `<interceptors>` element in `web-beans.xml`.

```
<interceptors>
   <interceptor>com.mydomain.framework.TransactionInterceptor</interceptor>
   <interceptor>com.mydomain.framework.LoggingInterceptor</interceptor>
</interceptors>
```

The order of the `<interceptor>` declarations determines the interceptor ordering. Interceptors which occur earlier in the list are called first.

If the `<interceptors>` element is specified in more than one `web-beans.xml` document, an exception is thrown by the Web Beans container at initialization time.

## 4.5. Events

Web Beans components may produce and consume events. This facility allows components to interact in a completely decoupled fashion, with no compile-time dependency between the two components.

### 4.5.1. Raising an event

The `Container` interface provides a method for raising events:

```
public interface Container {

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

}
```

The first argument is the *event object*:

```
public void login() {
    ...
    container.raiseEvent( new LoggedInEvent(user) );

}
```

The remaining arguments are optional instances of *event binding types*:

```
public void login() {
    User user = ...;
    container.raiseEvent( new LoggedInEvent(user), new Admin() {} );

}
```

An *event binding type* is a Java annotation defined as `@Target(PARAMETER)` and `@Retention(RUNTIME)`. All event binding types must also specify the `@EventBindingType` meta-annotation.

```
@EventBindingType
@Target(PARAMETER)
@Retention(RUNTIME)
public @interface Admin {}
```

### 4.5.2. Observer methods

An *observer method* is a non-static method of a Web Bean component with a parameter annotated `@Observes`. This parameter is called the *event parameter*. If the component implementation class is an EJB session bean, the observer method must be a business method of the session bean.

```
public void afterLogin(@Observes LoggedInEvent event) { ... }
```

If a method has more than one parameter annotated `@Observes`, an exception is thrown by the container.

When searching for observer methods for an event, the container considers the event parameter type, and event binding type annotations that appear on the event parameter:

```
public void afterAdminLogin(@Observes @Admin LoggedInEvent event) { ... }
```

There may be arbitrarily many observer methods with the same event parameter type and event binding type annotations.

An observer method may declare additional parameters, which may specify binding types. Any event binding types on these additional parameters will be ignored.

```
public void afterLogin(@Observes LoggedInEvent event, @Current User user) { ... }
```

When an event is raised by the application, the container determines the observer methods for that event.

- If the observer method is a transactional observer and there is currently a JTA transaction in progress, the container registers the observer for invocation during the transaction completion phase. At the appropriate point during the completion phase of the transaction, the container invokes the observer method.
- Otherwise, the container calls the observer method immediately.

To call an observer method, the container:

- obtains the current instance of the component which declares the observer method, by calling `Context.get()`, passing the `Component` object representing that component, then
- invokes the observer method on the current instance, passing the event object to the event parameter and using the instance resolution procedure to determine a value for each of the other parameters.

Observer methods may throw exceptions:

- If the observer method is a transactional observer, the exception is caught by the container.
- Otherwise, the exception aborts processing of the event. No other observers of that event will be called. The `raiseEvent()` method throws an `ObserverException` which wraps the exception.

*Open issue: declaring observer methods using XML - how do we specify the event parameter?*

### 4.5.3. Observer resolution

When searching for observers for an event, the container searches for observer methods which satisfy the following rules:

- The event object is assignable to the event parameter type.
- For each event binding type that appears on the event parameter declaration, (a) an instance of the type must have been passed to `raiseEvent()` and (b) any member values must match the member values of the instance passed to `raiseEvent()`.

#### 4.5.3.1. Event binding annotations with members

An event 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 = ...;
    container.raiseEvent( new LoggedInEvent(user),
            new Role() { public String value() { return user.getRole(); } );

}
```

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) { ... }
```

The container uses `equals()` to compare event binding type member values.

#### 4.5.3.2. Multiple event binding annotations

An observer method may have multiple event binding annotations:

```
public void afterDocumentUpdatedByAdmin(@Observes @Updated @ByAdmin Document doc) { ... }
```

Then this observer method will only notified if both the event binding types are specified when the event is raised:

```
container.raiseEvent( document, new Updated() {}, new ByAdmin() {} );
```

Other, less specific, observers will also be notified of this event:

```
public void afterDocumentUpdated(@Observes @Updated Document doc) { ... }
```

```
public void afterDocumentEvent(@Observes Document doc) { ... }
```

### 4.5.4. Conditional observers

*Conditional observers* are observer methods which are notified of an event only if an instance of the component that defines the observer method already exists in the current context. Conditional observers are specified by annotating the event parameter with the `@IfExists` annotation.

```
public void refreshOnDocumentUpdate(@IfExists @Observes @Updated Document doc) { ... }
```

### 4.5.5. 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 raised. If no transaction is in progress when the event is raised, they are notified at the same time as other observers.

*Note: this functionality is still under discussion by the expert group.*

Transactional observers are specified 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 completed successfully.
- The `@BeforeTransactionCompletion` annotation specifies that an observer method should be called during the before completion phase.

```
void onDocumentUpdate(@Observes @AfterTransactionSuccess @Updated Document doc) { ... }
```

# Chapter 5. The Web Beans context model

Web Bean component instances are bound to a context. The context implementation determines the instance lifecycle and visibility to other component instances.

## 5.1. Contexts

A *context* is a mapping from enabled Web Bean components to instances of those components. This mapping may be associated with a single thread or with a set of threads. The context associated with the current thread is called the *current context*.

The `Context` interface provides operations for accessing the entries in the current context.

The `get()` operation returns a component instance for the given Web Bean component.

```
public interface Context {

    public <T> T get(Component<T> component, boolean create);

    ...

}
```

For example:

```
Object instance = context.get(component, true);
```

If there is an instance already associated with the current context it is returned by `get()`. Otherwise, if the value of the `create` parameter is `false`, `get()` returns a null value. Or, if the value of the `create` parameter is `true`, the `Context` implementation creates a new instance of the given component by calling `Component.create()` and returns the new instance.

The instance returned by `get()` is called the *current instance* of the component for the calling thread.

The `get()` method may not return a null value, unless `Component.create()` returns null.

(Note that for a `@Dependent`-scoped component, the "current" instance depends not only upon the thread, but also upon the instance of the component that owns the dependent instance.)

The `remove()` operation must destroy the current instance of a component by passing the instance to the `destroy()` method of the `Component` object representing the component and removing the instance from the context:

```
public interface Context {

    public <T> void remove(Component<T> component);

    ...

}
```

For example:

```
context.remove(instance);
```

A destroyed instance must not subsequently be returned by the `get()` method.

If the given instance is not associated with the current context, when the `remove()` method is called, an exception should be thrown by the `Context` implementation.

### 5.1.1. Inactive contexts

At a particular point in the execution of the program, a scope may be *inactive* with respect to the current thread. In this case, any invocation of `get()` or `remove()` from the current thread upon the `Context` object for that scope should result in a `ContextNotActive` exception.

### 5.1.2. Context destruction

At certain points in the execution of the program, the context associated with the current thread is *destroyed*. When a context is destroyed, the `Context` implementation must destroy all component instances associated with the current context by passing the instance to the `destroy()` method of the `Component` object representing the component. A destroyed instance must not subsequently be returned by the `get()` method.

### 5.1.3. Pseudo-scopes

In general, `Context` implementations should obey the following rule:

*Suppose components X and Y both inject component Z. Then if x is the current instance of X, and y is the current instance of Y, then both x and y refer to the same injected instance of Z.*

However, this rule is not required by the Web Beans specification. When this rule is violated, we say that the scope type of Z is a *pseudo-scope*.

## 5.2. Dependent pseudo-scope

The `@Dependent` scope type is a pseudo-scope. Components declared with scope type `@Dependent` behave differently to components with other built-in scope types.

When a component is declared to have `@Dependent` scope:

- No injected instance of the component is ever shared between other component instances.
- Any injected instance of the component is bound to the lifecycle of the component into which it is injected.

Every Web Bean component instance has its own *dependent object context*. This context is bound to the component instance. The dependent object context must be destroyed by the Web Beans container when the associated component instance is destroyed.

The `@Dependent` scope is inactive except when the container is creating or destroying an instance or injecting its dependencies. A component instance may access its own dependent object context from the component constructor, the component remove method, injector methods, `@PostConstruct` methods, `@PreDestroy` methods and interceptors for any of these methods. A producer method component's dependent object context is accessible from the producer method, disposal method and interceptors for these methods.

## 5.3. Context management for built-in scopes

For each of the built-in scopes, contexts are automatically managed by the Web Beans container.

*Open issue: context management for RMI calls, EJB timer methods, message driven beans, etc, is yet to be defined.*

### 5.3.1. Request context lifecycle

The Web Beans request context is a built-in context for the built-in scope type `javax.webbeans.RequestScoped`. For any servlet request, this context is automatically managed by the Web Beans container. The request context is active during the `service()` method of any servlet in the web application.

The request context is destroyed at the end of the web request, after the servlet `service()` method returns.

### 5.3.2. Session context lifecycle

The Web Beans session context is a built-in context for the built-in scope type `javax.webbeans.SessionScoped`. For any servlet request, this context is automatically managed by the Web Beans container. The session context is active during the `service()` method of any servlet in the web application.

The session context is propagated between requests that occur in the same HTTP servlet session.

The session context is destroyed when the `HTTPSession` is invalidated or times out.

### 5.3.3. Application context lifecycle

The Web Beans application context is a built-in context for the built-in scope type `javax.webbeans.ApplicationScoped`. For any servlet request, this context is automatically managed by the Web Beans container. The application context is active during the `service()` method of any servlet in the web application.

The application context is shared between all requests to the same web application context.

The application context is destroyed when the web application context is destroyed.

### 5.3.4. Conversation context lifecycle

The Web Beans conversation context is a built-in context for the built-in scope type `javax.webbeans.ConversationScoped`. For any JSF request, this context is automatically managed by the Web Beans container, according to the following rules:

- For a JSF faces request, the context is active from the beginning of the apply request values phase, until the end of the render response phase
- For a JSF non-faces request, the context is active during the render response phase

A 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 Web Beans 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 transient conversation may be marked transient by calling `Conversation.end()`

`javax.webbeans.Conversation` is a built-in component and an instance may be obtained by injection.

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 associated conversation context object 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

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 Web Beans implementation is permitted to arbitrarily destroy any long-running conversation that is associated with no current JSF request.

If the propagated conversation cannot be restored, the request is associated with a new transient conversation.

*Open issue: allow the request to be blocked if the conversation cannot be restored.*

The Web Beans 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.*

## 5.4. 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 `Container.addContext()`.

```
public interface Container {

    public void addContext(Class<Annotation> scopeType, Context context);

    ...

}
```

For example:

```
container.addContext(MethodScoped.class, new MethodContext());
```

During instance or EL name resolution, the Web Beans container must call `Container.getContext()` to retrieve the context object associated with the component scope. If no context object is associated with the scope, `getContext()` throws a `ContextNotActive` exception.

```
public interface Container {

    public Context getContext(Class<Annotation> scopeType);

    ...

}
```

# Chapter 6. Transactions and persistence

The Web Beans container provides transaction and persistence context management which is aware of the conversation and lifecycle of the web request:

- In a JSF environment, the Web Beans container manages the lifecyle of the *Web Beans transaction context*, a JTA transaction tied to the JSF lifecycle.
- Web Beans provides JPA persistence contexts scoped to the Web Beans conversation.

This model supports efficient optimistic transaction processing in a JSF environment.

## 6.1. Transaction management

*This functionality is yet to be specified.*

## 6.2. Persistence context management

*This functionality is yet to be specified.*

# Chapter 7. Validation and databinding

Web Beans integrates with JSF and the functionality defined by the Bean Validation specification, allowing Web Bean components to declare model-based validation constraints, and have those constraints validated by JSF.

*This functionality is yet to be specified.*

# Chapter 8. Packaging and configuration

The Web Beans container automatically discovers Web Bean components deployed in EAR or WAR archives.

## 8.1. Web Bean component discovery

Web Bean component discovery is the process of determining:

- what components *exist* in the deployment archive
- which components are *enabled* for this deployment
- the *precedence* of the enabled components

When the Web Beans container is initialized, it considers classes in the web application classpath which are deployed in any of the following locations in the EAR or WAR:

- The `WEB-INF/classes` directory
- Any EJB JAR listed in an `<ejb>` element of the EAR's `application.xml` file which has a `META-INF/web-beans.xml` file
- Any JAR in the `WEB-INF/lib` directory of the WAR which has a `META-INF/web-beans.xml` file
- Any JAR in the library directory of the EAR which has a `META-INF/web-beans.xml` file

A Web Beans component *exists* for every class marked with a component type annotation.

*Open issue: Define component discovery rules for other deployment scenarios.*

*Open issue: Alternatively, the container could discover components in any archive in the web application classpath (that has a `web-beans.xml` file). This is a much simpler definition.*

In addition, the Web Beans container considers components defined in the following XML documents:

- `WEB-INF/web-beans.xml` in the WAR
- `META-INF/web-beans.xml` in the EAR *(Open issue: is this needed and can it even be implemented?)*
- `META-INF/web-beans.xml` for any EJB JAR listed in an `<ejb>` element of the EAR's `application.xml` file
- `META-INF/web-beans.xml` for any JAR in the `WEB-INF/lib` directory of the WAR
- `META-INF/web-beans.xml` for any JAR in the library directory of the EAR

A Web Beans component *exists* for every `<component>` element in any of these `web-beans.xml` files.

In addition, for every concrete type that appears at an injection point or as a producer method parameter, annotated with the `@New` binding annotation, a component exists with scope `@Stateless`, component type `@Component`, `@New` as the only binding annotation, no component name, and where the API type and implementation class are both the given concrete type.

Finally, a Web Beans component exists for every producer method defined on any component implementation class.

### 8.1.1. Enabled component types

The Web Beans container inspects the component type of each existing component to determine whether the component is *enabled* for this deployment. A component which is enabled will be available for use at runtime. If the component is not enabled, an instance cannot be obtained by injection or EL resolution and is never instantiated by the container.

By default, only Web Bean components with the `@Standard` or `@Component` component types are enabled. To enable components with some other component type, a `<component-types>` element must be included in the `WEB-INF/web-beans.xml` file and the component type must be declared using `<component-type>`.

If a `<component-types>` element is specified, only the explicitly declared component types are enabled. The `@Standard` component type must be declared.

```
<web-beans>
    <component-types>
        <component-type>javax.webbeans.Standard</component-type>
        <component-type>javax.webbeans.Application</component-type>
        <component-type>org.nih.dao.DaoFramework</component-type>
        <component-type>au.com.makemoneymoney.backoffice.MelbourneOffice</component-type>
        <component-type>au.com.makemoneymoney.test.Mock</component-type>
    </component-types>
</web-beans>
```

If no `<component-types>` element is specified, only the `@Standard` and `@Component` component types are enabled.

If the `<component-types>` element is specified in more than one `web-beans.xml` document, an exception is thrown by the Web Beans container at initialization time.

*Open issue: conditional enablement of components should be supported, but what should the mechanism be: `isEnabled()` method? `@Enabled` annotation? Component dependencies?*

### 8.1.2. Component type precedence

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

If no `<component-types>` element is specified, the `@Component` component type has a higher precedence than the `@Standard` component type.

*Open issue: there are many web-beans.xml files, where can this precedence be specified?*