

Jakarta EE WebProfile

Jakarta EE Platform Team, <https://projects.eclipse.org/projects/ee4j.jakartaee-platform>

11.0-M2, April 16, 2024: DRAFT

Table of Contents

Copyright	2
Eclipse Foundation Specification License	3
Disclaimers	3
1. Introduction	5
1.1. Target and Rationale for the Web Profile	5
1.2. Determining Applicable Requirements	6
1.3. Acknowledgements for Version 6	7
1.4. Acknowledgements for Version 7	7
1.5. Acknowledgements for Version 8	8
1.6. Acknowledgements for Jakarta EE 8	8
1.7. Acknowledgements for Jakarta EE 9	8
1.8. Acknowledgements for Jakarta EE 9.1	8
1.9. Acknowledgements for Jakarta EE 10.0	8
2. Component Specification Integration Requirements	9
2.1. CDI Extended Concepts for Jakarta EE	9
2.1.1. Functionality provided by the container to the bean in Jakarta EE	9
2.1.2. Bean types for Jakarta EE component	9
2.1.3. Scopes	10
2.1.4. Default bean discovery mode for Jakarta EE	10
2.1.5. Bean names in Jakarta EE	10
2.2. Addition to programming model for Jakarta EE	11
2.2.1. Managed beans in Jakarta EE	11
2.2.2. EJB Session beans	11
2.2.3. Producer methods on EJB session bean	13
2.2.4. Producer field on EJB session bean	14
2.2.5. Disposer methods on EJB session bean	14
2.2.6. Jakarta EE components	14
2.2.7. Resources	15
2.2.8. Additional built-in beans	16
2.2.9. Injected fields in Jakarta EE	17
2.2.10. Initializer methods in Jakarta EE	17
2.2.11. Inheritance of type-level metadata in Jakarta EE	17
2.2.12. Inheritance of member-level metadata in Jakarta EE	17
2.2.13. Specialization in Jakarta EE	17
2.3. Dependency injection, lookup and EL in Jakarta EE	17
2.3.1. Modularity in Jakarta EE	18
2.3.2. EL name resolution	18
2.3.3. Dependency injection in Jakarta EE	19

2.4. Scopes and contexts in Jakarta EE	20
2.4.1. Dependent pseudo-scope in Jakarta EE	20
2.4.2. Passivation and passivating scopes in Jakarta EE	21
2.4.3. Context management for built-in scopes in Jakarta EE	21
2.5. Lifecycle of contextual instances	25
2.5.1. Container invocations and interception in Jakarta EE	25
2.6. Decorators in Jakarta EE	27
2.6.1. Decorator beans in Jakarta EE	27
2.7. Interceptor bindings in Jakarta EE	27
2.7.1. Interceptor enablement and ordering in Jakarta EE	27
2.7.2. Interceptor resolution in Jakarta EE	27
2.8. Events in Jakarta EE	27
2.8.1. Observer methods in EJB session beans	27
2.9. Portable extensions in Jakarta EE	28
2.9.1. The Bean interface in Jakarta EE	28
2.9.2. Injecti onTarget interface in Jakarta EE	28
2.9.3. The BeanManager object in Jakarta EE	28
2.9.4. Alternative metadata sources and EJB	29
2.9.5. Addition to Container lifecycle events in Jakarta EE	29
2.10. Packaging and deployment in Jakarta EE	30
2.10.1. Bean archive with EJB Session Beans	30
2.10.2. Type and Bean discovery for EJB	31
2.11. Integration with Unified EL	31
2.11.1. Bean name resolution in EL expressions	31
2.11.2. Unified EL integration API	32
2.12. CDI Specification References	32
3. Web Profile Definition	36
3.1. Required Components	36
3.2. Optional Components	36
3.3. Additional Requirements	37
Appendix A: Revision History	38
A.1. Changes in Final Release for EE10	38
A.1.1. Editorial Changes for EE10	38
A.2. Changes in Final Release for EE9.1	38
A.2.1. Editorial Changes	38
Appendix B: Related Documents	39
4. Jakarta EE 10 Web Profile Dependencies	41

Specification: Jakarta EE WebProfile

Version: 11.0-M2

Status: DRAFT

Release: April 16, 2024

Copyright

Copyright (c) 2018, 2022 Eclipse Foundation

Eclipse Foundation Specification License

By using and/or copying this document, or the Eclipse Foundation document from which this statement is linked, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions:

Permission to copy, and distribute the contents of this document, or the Eclipse Foundation document from which this statement is linked, in any medium for any purpose and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the document, or portions thereof, that you use:

¥ link or URL to the original Eclipse Foundation document.

¥ All existing copyright notices, or if one does not exist, a notice (hypertext is preferred, but a textual representation is permitted) of the form: "Copyright (c) [\$date-of-document] Eclipse Foundation, Inc. <<url to this license>>"

Inclusion of the full text of this NOTICE must be provided. We request that authorship attribution be provided in any software, documents, or other items or products that you create pursuant to the implementation of the contents of this document, or any portion thereof.

No right to create modifications or derivatives of Eclipse Foundation documents is granted pursuant to this license, except anyone may prepare and distribute derivative works and portions of this document in software that implements the specification, in supporting materials accompanying such software, and in documentation of such software, PROVIDED that all such works include the notice below. HOWEVER, the publication of derivative works of this document for use as a technical specification is expressly prohibited.

The notice is:

"Copyright (c) [\$date-of-document] Eclipse Foundation. This software or document includes material copied from or derived from [title and URI of the Eclipse Foundation specification document]."

Disclaimers

THIS DOCUMENT IS PROVIDED "AS IS," AND THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DOCUMENT ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE DOCUMENT OR THE PERFORMANCE OR IMPLEMENTATION OF THE CONTENTS THEREOF.

The name and trademarks of the copyright holders or the Eclipse Foundation may NOT be used in advertising or publicity pertaining to this document or its contents without specific, written prior

permission. Title to copyright in this document will at all times remain with copyright holders.

Chapter 1. Introduction

This specification defines the Jakarta^a EE Web Profile (Web Profile), a profile of the Jakarta^a Platform, Enterprise Edition specifically targeted at web applications.

1.1. Target and Rationale for the Web Profile

The Web Profile is targeted at developers of modern web applications.

With the term “modern” we intend to highlight the fact that the world of web applications has made much progress since the introduction of the first Servlet specification. Inevitably, the number of technologies used to create even simple web applications had grown by leaps and bounds. In fact, few web applications today are written directly to the servlet API: most applications rely on standard or third-party frameworks and libraries, often developed as open source, which in turn use the services of the servlet container.

Besides managing HTTP interactions, most web applications have significant requirements in the areas of transaction management, security and persistence. Such requirements can be readily addressed by technologies that have been part of the Jakarta EE platform for quite some time, such as the Jakarta Enterprise Beans 3.x technology and the Jakarta Persistence, but that are rarely supported by “plain” servlet containers. By incorporating many of these APIs, the Web Profile aims at raising the bar for what should be considered a basic stack for the development of web applications using the Java platform.

Targeting “modern” web applications then implies offering a reasonably complete stack, composed of standard APIs, and capable out-of-the-box of addressing the needs of a large class of web applications. Furthermore, this stack should be easy to grow, so as to address any remaining developer needs.

Against this drive towards completeness, one wishes to balance a desire to limit the footprint of web containers, both in physical and in conceptual terms. From the point of view of developers learning the Web Profile, it is more valuable to have a small, focused profile, with as little overlap between technologies as possible, rather than a more powerful but overly complex one, with redundant APIs.

In defining the Web Profile we strove to find a middle ground between these two sets of requirements.

In terms of completeness, the Web Profile offers a complete stack, with technologies addressing presentation and state management (Jakarta Server Faces, Jakarta Server Pages), core web container functionality (Jakarta Servlet), business logic (Jakarta Enterprise Beans Lite), transactions (Jakarta Transactions), persistence (Jakarta Persistence) and more.

As for simplicity, it leaves out many of the enterprise backend APIs that are part of the Jakarta EE platform. It also relies on the pluggability features in the Servlet specification to allow applications to use libraries that extend the servlet container with minimal configuration overhead.

Finally, it is worth reminding that Web Profile products are allowed to ship with more technologies than the required ones. It is conceivable that products will offer a choice at installation time

between different configurations, some richer in extensions, or even allow for complete customization beyond the required core (à la carte installation).

1.2. Determining Applicable Requirements

!

Profile definitions can be quite terse, amounting to little more than a list of required technologies and a (possibly empty) set of additional requirements, beyond those entailed by all the referenced specifications. Being the first profile of the Java^a EE 6 Platform to be defined, we expect the Web Profile specification to be used as a model for future profiles. It will also be seen as a starting point for understanding how the requirements defined in the Jakarta EE Platform specification apply to a profile that subsets the platform itself, a significant innovation in this version of the platform. (The case of a profile that is a superset of the platform is much easier to picture.) To help with this process, this section attempts to shed light on how one should go from the definition of the Web Profile to figuring out the exact set of requirements that apply to it, and consequently to any product that implements it.

As dictated by the general rules for Jakarta EE profiles in the Platform specification, products that implement the Web Profile must honor:

1. all requirements of the Jakarta EE Platform specification that apply to all profiles;
2. all requirements of this specification;
3. all requirements of the individual component specifications;
4. all requirements in the Jakarta EE Platform specification that are conditional on the presence of a specific technology or combinations of technologies.

Let's look at some examples of requirements from each grouping.

For the first one, the Jakarta EE Platform specification mandates support for the Java^a Platform, Standard Edition 11 API.

In the second category one can point out the requirement to support Jakarta EE web application modules (.war files) ([see Additional Requirements](#)).

The third category is hopefully self-explanatory. For example, Web Profile products must implement the Servlet API, which in turn means they need to satisfy all the requirements listed in the Jakarta Servlet specification.

The fourth category is the most complex. As a first example, since a Web Profile product is required to implement the Servlet technology, it must also follow all general requirements for Jakarta EE web containers in the Platform specification. Additionally, it must follow all security requirements in the Platform specification that pertain to Jakarta EE web containers, all interoperability requirements for such containers, etc. Furthermore, since a Web Profile product must implement the Jakarta Transactions API, it must also satisfy all the Platform specification's transaction management-related requirements for web components, which indeed are conditional on the presence of Jakarta Servlet and Jakarta Transactions.

As a negative example for the fourth category of requirements, consider the Jakarta Messaging technology. Since it is not a required component of the Web Profile, Web Profile products are not required to include an implementation of Jakarta Messaging, nor do they have to support other Jakarta Messaging-related requirements, like the ability to inject message destination references. On the other hand, a Web Profile product that included an implementation of Jakarta Messaging would be required to honor all the Jakarta Messaging-related requirements in the Jakarta EE Platform specification.

Particular care should be taken when determining applicable requirements based on the presence of Jakarta Enterprise Beans Lite in the Web Profile. As described in the Jakarta Enterprise Beans specification, Jakarta Enterprise Beans Lite is a subset of the Jakarta Enterprise Beans API. When examining an Jakarta Enterprise Beans-related requirement in the Jakarta EE Platform spec, one must first of all determine which API classes, component types and Jakarta Enterprise Beans container services are mentioned in the requirement itself. Only if all of them fall inside the Jakarta Enterprise Beans Lite subset that requirement is considered applicable to Web Profile products.

For example, since Jakarta Enterprise Beans Lite does not include any remote functionality, the *EJB* annotation may not be used to inject a remote reference, something that should be kept in mind when evaluating the requirements in the Platform specification section [Jakarta Enterprise Beans References](#).

1.3. Acknowledgements for Version 6

Version 6 of this specification was created under the Java Community Process as JSR-316. The spec leads for the JSR-316 Expert Group were Bill Shannon (Sun Microsystems, Inc.) and Roberto Chinnici (Sun Microsystems, Inc.). The expert group included the following members: Florent Benoit (Inria), Adam Bien (Individual), David Blevins (Individual), Bill Burke (Red Hat Middleware LLC), Larry Cable (BEA Systems), Bongjae Chan (Tmax Soft, Inc.), Rejeev Divakaran (Individual), Francois Exertier (Inria), Jeff Genender (Individual), Antonio Goncalves (Individual), Jason Greene (Red Hat Middleware LLC), Gang Huang (Peking University), Rod Johnson (SpringSource), Werner Keil (Individual), Michael Keith (Oracle), Wonseok Kim (Tmax Soft, Inc.), Jim Knutson (IBM), Erika S. Kohen (Individual), Peter Kristiansson (Ericsson AB), Changshin Lee (NCsoft Corporation), Felipe Leme (Individual), Ming Li (TongTech Ltd.), Vladimir Pavlov (SAP AG), Dhanji R. Prasanna (Google), Reza Rahman (Individual), Rajiv Shivane (Pramati Technologies), Hani Suleiman (Individual).

1.4. Acknowledgements for Version 7

Version 7 of this specification was created under the Java Community Process as JSR-342. The Expert Group work for this specification was conducted by means of the <http://javaee-spec.java.net> project in order to provide transparency to the Java community. The specification leads for the JSR-342 Expert Group were Bill Shannon (Oracle) and Linda DeMichiel (Oracle). The expert group included the following members: Deepak Anupalli (Pramati Technologies), Anton Arhipov (ZeroTurnaround), Florent Benoit (OW2), Adam Bien (Individual), David Blevins (Individual), Markus Eisele (Individual), Jeff Genender (Individual), Antonio Goncalves (Individual), Jason Greene (Red Hat, Inc.), Minehiko Iida (Fujitsu), Alex Heneveld (Individual), Jevgeni Kabanov (Individual), Ingyu Kang (Tmax Soft, Inc.), Werner Keil (Individual), Jim Knutson (IBM), Ming Li (TongTech Ltd.), Pete Muir (Red Hat, Inc.), Minoru Nitta (Fujitsu), Reza Rahman (Caucho Technology),

Inc), Kristoffer Sjogren (Ericsson AB), Kevin Sutter (IBM), Spike Washburn (Individual), Kyung Koo Yoon (Tmax Soft).

1.5. Acknowledgements for Version 8

Version 8 of this specification was created under the Java Community Process as JSR-366. The Expert Group work for this specification was conducted by means of the <http://javaee-spec.java.net> and <https://javaee.github.io/javaee-spec> projects in order to provide transparency to the Java community. The specification leads for the JSR-366 Expert Group were Bill Shannon (Oracle) and Linda DeMichiel (Oracle). The expert group included the following members: Florent Benoit (OW2), David Blevins (Tomitribe), Jeff Genender (Savoir Technologies), Antonio Goncalves (Individual), Jason Greene (Red Hat), Werner Keil (Individual), Moon Namkoong (TmaxSoft, Inc.) Antoine Sabot-Durand (Red Hat), Kevin Sutter (IBM), Ruslan Synytsky (Jelastic, Inc.), Markus Winkler (oparco - open architectures & consulting). Reza Rahman (Individual) participated as a contributor.

1.6. Acknowledgements for Jakarta EE 8

The Jakarta EE 8 specification was created by the Jakarta EE Platform Specification Project with guidance provided by the Jakarta EE Working Group (<https://jakarta.ee/>).

1.7. Acknowledgements for Jakarta EE 9

The Jakarta EE 9 specification was created by the Jakarta EE Platform Specification Project with guidance provided by the Jakarta EE Working Group (<https://jakarta.ee/>).

1.8. Acknowledgements for Jakarta EE 9.1

The Jakarta EE 9.1 specification was created by the Jakarta EE Platform Specification Project with guidance provided by the Jakarta EE Working Group (<https://jakarta.ee/>).

1.9. Acknowledgements for Jakarta EE 10.0

The Jakarta EE 10 specification was created by the Jakarta EE Platform Specification Project with guidance provided by the Jakarta EE Working Group (<https://jakarta.ee/>).

Chapter 2. Component Specification

Integration Requirements

This section defines the requirements between component specifications that are included in the Platform.

This part of the document specifies additional rules or features when using CDI in a Jakarta EE container. All content defined in [\[cdi-spec\]](#) applies to this part.

CDI implementations in Jakarta EE containers are required to support CDI Full.

2.1. CDI Extended Concepts for Jakarta EE

When running in Jakarta EE, the container must extend the concepts defined in [\[concepts\]](#) with:

- ¥ A Jakarta EE component is a *bean* if the lifecycle of its instances may be managed by the container according to the lifecycle context model defined in [\[contexts\]](#)
- ¥ Contextual instances of a bean may be used in EL expressions that are evaluated in the same context
- ¥ For some Jakarta EE components - like environment resources, defined in [Resources](#) - the developer provides only the annotations and the bean implementation is provided by the container.

2.1.1. Functionality provided by the container to the bean in Jakarta EE

When running in Jakarta EE, the container must extend the capabilities defined in [\[capabilities\]](#), by providing:

- ¥ scoped resolution by bean name when used in a Unified EL expression, as defined by [\[name_resolution\]](#).

2.1.2. Bean types for Jakarta EE component

As managed beans, EJB session beans may have multiple bean types depending on their client-visible types. For instance, this session bean has only the local interfaces [BookShop](#) and [Audi table](#), along with [Object](#), as bean types, since the bean class is not a client-visible type.

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

The rules for determining the (unrestricted) set of bean types for Jakarta EE components are defined in [Bean types of a session bean](#) and [Bean types of a resource](#).

2.1.3. Scopes

Jakarta EE components such as servlets, EJBs and JavaBeans do not have a well-defined *scope*. These components are either:

- ¥ *singletons*, such as EJB singleton session 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 explicitly created and destroyed by their client, such as JavaBeans and stateful session beans, whose state is shared by explicit reference passing between clients.

CDI scopes add to Jakarta EE these missing well-defined lifecycle context as defined in [\[scopes\]](#).

2.1.3.1. Built-in scope types in Jakarta EE

When running in Jakarta EE, the implementations of the `@RequestScoped`, `@ApplicationScoped` and `@SessionScoped` annotations provided by the container, represent the standard scopes defined by the Java Servlets specification.

2.1.4. Default bean discovery mode for Jakarta EE

When running in Jakarta EE, If the *bean discovery mode* is `annotated`, the container must extend the rules defined in [\[default_bean_discovery\]](#) with:

- ¥ bean classes of EJB sessions beans, are discovered, and
- ¥ producer methods that are on an EJB session bean are discovered, and
- ¥ producer fields that are on an EJB session bean are discovered, and
- ¥ disposer methods that are on an EJB session bean are discovered, and
- ¥ observer methods that are on an EJB session bean are discovered.

2.1.5. Bean names in Jakarta EE

A bean with a name may be referred to by its name in Unified EL expressions.

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

Bean names allow the direct use of beans in JSP or JSF pages. For example, a bean with the name `products` could be used like this:

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

2.1.5.1. Default bean names for EJB session beans

In the circumstances listed in [\[default_name\]](#), the rule for determining default name for an EJB session bean are defined in [Default bean name for a session bean](#).

2.2. Addition to programming model for Jakarta EE

When running in Jakarta EE, the container must extend the rules defined in [\[implementation\]](#), and must also provide built-in support for injection and contextual lifecycle management of the following kinds of bean:

- ¥ Session beans

- ¥ Resources (Jakarta EE resources, persistence contexts, persistence units, remote EJBs and web services)

Jakarta EE and embeddable EJB containers are required by the Jakarta EE and EJB specifications to support EJB session beans and the Jakarta EE component environment. Other containers are not required to provide support for injection or lifecycle management of session beans or resources.

2.2.1. Managed beans in Jakarta EE

2.2.1.1. Which Java classes are managed beans in Jakarta EE?

When running in Jakarta EE, a top-level Java class is a managed bean if it meets requirements described in [\[what_classes_are_beans\]](#) or if it is defined to be a managed bean by any other Jakarta EE specification and if

- ¥ It is not annotated with an EJB component-defining annotation or declared as an EJB bean class in `ejb-jar.xml`.

2.2.2. EJB Session beans

A *session bean* is a bean that is implemented by a session bean with an EJB 3.x client view that is not annotated with `@Vetoed` or in a package annotated `@Vetoed`. The basic lifecycle and semantics of EJB session beans are defined by the EJB specification.

A stateless session bean must belong to the `@Dependent` pseudo-scope. A singleton session bean must belong to either the `@ApplicationScoped` scope or to the `@Dependent` pseudo-scope. If a session bean specifies an illegal scope, the container automatically detects the problem and treats it as a definition error. A stateful session bean may have any scope.

When a contextual instance of a session bean is obtained via the dependency injection service, the behavior of `SessionContext.getInvokedBusinessInterface()` is specific to the container implementation. Portable applications should not rely upon the value returned by this method.

If the bean class of a session bean is annotated `@Interceptor` or `@Decorator`, the container automatically detects the problem and treats it as a definition error.

If the session bean class is a generic type, it must have scope `@Dependent`. If a session bean with a parameterized bean class declares any scope other than `@Dependent`, the container automatically detects the problem and treats it as a definition error.

2.2.2.1. EJB remove methods of session beans

If a session bean is a stateful session bean:

- ¥ If the scope is `@Dependent`, the application *may* call any EJB remove method of a contextual instance of the session bean.
- ¥ Otherwise, the application *may not* directly call any EJB remove method of any contextual instance of the session bean.

The session bean is not required to have an EJB remove method in order for the container to destroy it.

If the application directly calls an EJB remove method of a contextual instance of a session 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 a contextual instance of a session 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 `Contextual.destroy()` is called, as defined in [Lifecycle of EJB stateful session beans](#).

2.2.2.2. Bean types of a session bean

The unrestricted set of bean types for a session bean contains all local interfaces of the bean and their superinterfaces. If the session bean has a no-interface view, the unrestricted set of bean types contains the bean class and all superclasses. In addition, `java.lang.Object` is a bean type of every session bean.

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

The resulting set of bean types for a session bean consists only of [legal bean types](#), all other types are removed from the set of bean types.

2.2.2.3. Declaring a session bean

A session bean does not require any special annotations apart from the component-defining annotation (or XML declaration) required by the EJB specification. The following EJBs are beans:

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

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

A bean class may also specify a scope, bean name, stereotypes and/or qualifiers:

```
@ConversationScoped @Stateful @Default @Model
```

```
public class ShoppingCart { ... }
```

A session bean class may extend another bean class:

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

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

2.2.2.4. Specializing a session bean

If a bean class of a session bean X is annotated `@Specializes`, then the bean class of X must directly extend the bean class of another session bean Y. Then X *directly specializes* Y, as defined in [\[specialization\]](#).

If the bean class of X does not directly extend the bean class of another session bean, the container automatically detects the problem and treats it as a definition error.

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

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

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

2.2.2.5. Default bean name for a session bean

The default name for a session bean is the unqualified class name of the session bean class, after converting the first character to lower case.

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

2.2.3. Producer methods on EJB session bean

A producer method defined in an EJB session bean follows the rules defined in [\[producer_method\]](#) with the following addition:

- ¥ A producer method defined in an EJB session bean must be either a business method exposed by a local business interface of the EJB or a static method of the bean class.

2.2.3.1. Declaring a producer method in an EJB session bean

A producer method declaration in an EJB session bean follows the rules defined in [\[declaring_producer_method\]](#) with the following addition:

- ¥ if a non-static method of a session bean class is annotated `@Produces`, and the method is not a business method exposed by a local business interface of the session bean, the container automatically detects the problem and treats it as a definition error.

2.2.4. Producer field on EJB session bean

A producer field defined in an EJB session bean follows the rules defined in [\[producer_field\]](#) with the following addition:

- ¥ A producer field defined in an EJB session bean must be a static field of the bean class.

2.2.4.1. Declaring a producer field in an EJB session bean

A producer field declaration in an EJB session bean follows the rules defined in [\[declaring_producer_field\]](#) with the following addition:

- ¥ If a non-static field of an EJB session bean class is annotated `@Produces`, the container automatically detects the problem and treats it as a definition error.

2.2.5. Disposer methods on EJB session bean

A disposer method defined in an EJB session bean follows the rules defined in [\[disposer_method\]](#) with the following addition:

- ¥ A disposer method defined in an EJB session bean must be either a business method exposed by a local business interface of the EJB or a static method of the bean class.

2.2.5.1. Declaring a disposer method on an EJB session bean

A disposer method declaration in an EJB session bean follows the rules defined in [\[declaring_disposer_method\]](#) with the following addition:

- ¥ If a non-static method of an EJB session bean class has a parameter annotated `@Disposes`, and the method is not a business method exposed by a local business interface of the session bean, the container automatically detects the problem and treats it as a definition error.

2.2.6. Jakarta EE components

Most Jakarta EE components support injection and interception, as defined in the Jakarta EE Platform, Specification, table EE.5-1, but are not considered beans (as defined by this specification). EJBs, as defined in [EJB Session beans](#) are the exception.

The instance used by the container to service an invocation of a Jakarta EE component will not be the same instance obtained when using `@Inject`, instantiated by the container to invoke a producer method, observer method or disposer method, or instantiated by the container to access the value

of a producer field. It is recommended that Jakarta EE components should not define observer methods, producer methods, producer fields or disposer methods. It is safe to annotate Jakarta EE components with `@Vetoed` to prevent them being considered beans.

2.2.7. Resources

A *resource* is a bean that represents a reference to a resource, persistence context, persistence unit, remote EJB or web service in the Jakarta EE component environment.

By declaring a resource, we enable an object from the Jakarta EE component environment to be injected by specifying only its type and qualifiers at the injection point. For example, if `@CustomerDatabase` is a qualifier:

```
@Inject @CustomerDatabase Datasource customerData;
```

```
@Inject @CustomerDatabase EntityManager customerDatabaseEntityManager;
```

```
@Inject @CustomerDatabase EntityManagerFactory customerDatabaseEntityManagerFactory;
```

```
@Inject PaymentService remotePaymentService;
```

The container is not required to support resources with scope other than `@Dependent`. Portable applications should not define resources with any scope other than `@Dependent`.

A resource may not have a bean name.

2.2.7.1. Declaring a resource

A resource may be declared by specifying a Jakarta EE component environment injection annotation as part of a producer field declaration. The producer field may be static.

¥ For a Jakarta EE resource, `@Resource` must be specified.

¥ For a persistence context, `@PersistenceContext` must be specified.

¥ For a persistence unit, `@PersistenceUnit` must be specified.

¥ For a remote EJB, `@EJB` must be specified.

¥ For a web service, `@WebServiceRef` must be specified.

The injection annotation specifies the metadata needed to obtain the resource, entity manager, entity manager factory, remote EJB instance or web service reference from the component environment.

```
@Produces @WebServiceRef(lookup="java:app/service/PaymentService")
```

```
PaymentService paymentService;
```

```
@Produces @EJB(ejbLink=".. /the ir. jar#PaymentService")  
PaymentService paymentService;
```

```
@Produces @Resource(lookup="java:global /env/jdbc/CustomDataSource")  
@CustomerDatabase DataSource customerDatabase;
```

```
@Produces @PersistenceContext(unitName="CustomerDatabase")  
@CustomerDatabase EntityManager customerDatabasePersistenceContext;
```

```
@Produces @PersistenceUnit(unitName="CustomerDatabase")  
@CustomerDatabase EntityManagerFactory customerDatabasePersistenceUnit;
```

The bean type and qualifiers of the resource are determined by the producer field declaration.

If the producer field declaration specifies a bean name, the container automatically detects the problem and treats it as a definition error.

If the matching object in the Jakarta EE component environment is not of the same type as the producer field declaration, the container automatically detects the problem and treats it as a definition error.

2.2.7.2. Bean types of a resource

The unrestricted set of bean types for a resource is determined by the declared type of the producer field, as specified by [\[producer_field_types\]](#).

The resulting set of bean types for a resource consists only of [legal bean types](#), all other types are removed from the set of bean types.

2.2.8. Additional built-in beans

A Jakarta EE or embeddable EJB container must provide the following built-in beans, all of which have qualifier `@Default`:

- ¥ a bean with bean type `jakarta.transaction.UserTransaction`, allowing injection of a reference to the JTA `UserTransaction`, and

A servlet container must provide the following built-in beans, all of which have qualifier `@Default`:

- ¥ a bean with bean type `jakarta.servlet.http.HttpServletRequest`, allowing injection of a reference to the `HttpServletRequest`

- ¥ a bean with bean type `jakarta.servlet.http.HttpSession`, allowing injection of a reference to the `HttpSession`,

¥ a bean with bean type `jakarta.servlet.ServletContext`, allowing injection of a reference to the `ServletContext`,

These beans are passivation capable dependencies, as defined in [\[passivation_capable_dependency\]](#).

If a Jakarta EE component class has an injection point of type `UserTransaction` and qualifier `@Default`, and may not validly make use of the JTA `UserTransaction` according to the Jakarta EE platform specification, the container automatically detects the problem and treats it as a definition error.

2.2.9. Injected fields in Jakarta EE

When running in Jakarta EE, the container must extend the rules defined for bean classes in [\[injected_fields\]](#) to Jakarta EE component classes supporting injection.

2.2.10. Initializer methods in Jakarta EE

When running in Jakarta EE, the container must extend the rules defined for bean classes in [\[initializer_methods\]](#) to Jakarta EE component classes supporting injection. The container must also ensure that:

¥ An initializer method defined in an EJB session bean is *not* required to be a business method of the session bean.

2.2.11. Inheritance of type-level metadata in Jakarta EE

When running in Jakarta EE, the container must extend the rules defined for managed beans in [\[type_level_inheritance\]](#) to EJB session beans.

2.2.12. Inheritance of member-level metadata in Jakarta EE

When running in Jakarta EE, the container must extend the rules defined for managed beans in [\[member_level_inheritance\]](#) to EJB session beans.

2.2.13. Specialization in Jakarta EE

2.2.13.1. Direct and indirect specialization in Jakarta EE

When running in Jakarta EE, the container must extend the rules defined in [\[member_level_inheritance\]](#) and is also required to support specialization for EJB session beans as defined in [Specializing a session bean](#).

2.3. Dependency injection, lookup and EL in Jakarta EE

When running in Jakarta EE, the container must extend the rules defined in [\[injection_and_resolution\]](#) and may also provide references to contextual instances by Unified EL expression evaluation.

When resolving a name in an EL expression, the container considers the bean name and selected alternatives.

2.3.1. Modularity in Jakarta EE

In the Jakarta EE module architecture, any Jakarta EE module or library is a module. The Jakarta EE module is a bean archive if it contains a `beans.xml` file, as defined in [\[bean_archive_full\]](#).

When running in Jakarta EE, the container must follow the same accessibility rules for beans and alternatives defined in [\[selection\]](#) for JSP/JSF pages using EL resolution and make sure that only beans available from injection in the module that defines the JSP/JSF pages are resolved.

In the Jakarta EE module architecture, a bean class is accessible in a module if and only if it is required to be accessible according to the class loading requirements defined by the Jakarta EE platform specification.

Note that, in some Jakarta EE implementations, a bean class might be accessible to some other class even when this is not required by the Jakarta EE platform specification. For the purposes of this specification, a class is not considered accessible to another class unless accessibility is explicitly required by the Jakarta EE platform specification.

An alternative is not available for injection, lookup or EL resolution to classes or JSP/JSF pages in a module unless the module is a bean archive and the alternative is explicitly *selected* for the bean archive or the application.

2.3.1.1. Declaring selected alternatives for an application in Jakarta EE

When running in Jakarta EE, the container must extend the rules defined for managed beans in [\[declaring_selected_alternatives_application\]](#) to EJB session beans.

2.3.1.2. Declaring selected alternatives for a bean archive in Jakarta EE

When running in Jakarta EE, the container must extend the rules defined for managed beans in [\[declaring_selected_alternatives_bean_archive\]](#) to EJB session beans.

2.3.1.3. Unsatisfied and ambiguous dependencies in Jakarta EE

When running in Jakarta EE, the container must extend the rules defined in [\[unsatisfied_and_ambig_dependencies\]](#) and must also validate all injection points of all Jakarta EE component classes supporting injection

2.3.2. EL name resolution

When running in Jakarta EE, the container must extend the rules defined in [\[name_resolution\]](#) and must also support name resolution for name used in Expression Language

An EL name resolves to a bean if:

- ¥ the name can be resolved to a bean according to rules in [\[name_resolution\]](#), and
- ¥ the bean is available for injection in the war containing the JSP or JSF page with the EL

expression.

2.3.2.1. Ambiguous EL names

When running in Jakarta EE, the container must extend the rules defined in [\[ambig_names\]](#) to names used in Expression Language.

2.3.3. Dependency injection in Jakarta EE

When running in Jakarta EE, the container must extend the rules defined in [\[injection\]](#) and is also required to perform dependency injection whenever it creates the following contextual objects:

- ¥ contextual instances of EJB session beans.

The container is also required to perform dependency injection whenever it instantiates any of the following non-contextual objects:

- ¥ non-contextual instances of EJB session beans (for example, session beans obtained by the application from JNDI or injected using `@EJB`), and
- ¥ instances of any other Jakarta EE component class supporting injection.

A Java EE 5 container is not required to support injection for non-contextual objects.

2.3.3.1. Injection using the bean constructor in Jakarta EE

When running in Jakarta EE, the container must extend the rules defined for managed beans in [\[instantiation\]](#) to EJB session beans.

2.3.3.2. Injection of fields and initializer methods in Jakarta EE

When running in Jakarta EE, the container must extend the rules defined for managed beans in [\[fields_initializer_methods\]](#) to EJB session beans and to any other Jakarta EE component class supporting injection.

The container is also required to ensure that:

- ¥ Initializer methods declared by a class X in the type hierarchy of the bean are called after all Jakarta EE component environment resource dependencies declared by X or by superclasses of X have been injected.
- ¥ Any `@PostConstruct` callback declared by a class X in the type hierarchy of the bean is called after all Jakarta EE component environment resource dependencies declared by X or by superclasses of X have been injected.
- ¥ Any servlet `init()` method is called after all initializer methods have been called, all injected fields have been initialized and all Jakarta EE component environment resource dependencies have been injected.

2.3.3.3. Destruction of dependent objects in Jakarta EE

When running in Jakarta EE, the container must extend the rules defined for managed beans in

[\[dependent_objects_destruction\]](#) to any other Jakarta EE component class supporting injection and perform destruction after the servlet `destroy()` method is called.

2.3.3.4. Bean metadata in Jakarta EE

Interceptor and decorator instances associated with Jakarta EE components that are not considered beans (as defined by this specification) cannot obtain information about the beans they intercept and decorate (as defined in [\[bean_metadata\]](#)) and thus `null` is injected into relevant injection points.

2.4. Scopes and contexts in Jakarta EE

2.4.1. Dependent pseudo-scope in Jakarta EE

When running in Jakarta EE, the container must extend the rules defined in [\[dependent_context\]](#) and must also ensure that if a bean is declared to have `@Dependent` scope:

- ¥ When a Unified EL expression in a JSF or JSP page that refers to the bean by its bean name is evaluated, at most one instance of the bean is instantiated. This instance exists to service just a single evaluation of the EL expression. It is reused if the bean name appears multiple times in the EL expression, but is never reused when the EL expression is evaluated again, or when another EL expression is evaluated.

2.4.1.1. Dependent objects in Jakarta EE

When running in Jakarta EE, the container must extend the rules defined for bean in [\[dependent_objects\]](#) to Jakarta EE component class instance.

2.4.1.2. Destruction of objects with scope `@Dependent` in Jakarta EE

When running in Jakarta EE, the container must extend the rules defined for bean in [\[dependent_destruction\]](#) to Jakarta EE component class instance, and must also ensure that :

- ¥ all `@Dependent` scoped contextual instances created during evaluation of a Unified EL expression in a JSP or JSF page are destroyed when the evaluation completes.

2.4.1.3. Dependent pseudo-scope and Unified EL

Suppose a Unified EL expression in a JSF or JSP page refers to a bean with scope `@Dependent` by its bean name. Each time the EL expression is evaluated:

- ¥ the bean is instantiated at most once, and
- ¥ the resulting instance is reused for every appearance of the bean name, and
- ¥ the resulting instance is destroyed when the evaluation completes.

Portable extensions that integrate with the container via Unified EL should also ensure that these rules are enforced.

2.4.2. Passivation and passivating scopes in Jakarta EE

2.4.2.1. Passivation capable beans in Jakarta EE

¥ As defined by the EJB specification, an EJB stateful session beans is passivation capable if:

- ! interceptors and decorators of the bean are passivation capable, and,
- ! the EJB stateful session bean does not have the `passivationCapable` flag set to `false`.

¥ As defined by the EJB specification, an EJB stateless session bean or an EJB singleton session bean is not passivation capable.

2.4.2.2. Passivation capable dependencies in Jakarta EE

When running in Jakarta EE, the container must extend the rules defined in [\[passivation_capable_dependency\]](#), and must also guarantee that:

- ¥ all EJB stateless session beans are passivation capable dependencies,
- ¥ all EJB singleton session beans are passivation capable dependencies,
- ¥ all passivation capable EJB stateful session beans are passivation capable dependencies, and
- ¥ all Jakarta EE resources are passivation capable dependencies.

2.4.2.3. Validation of passivation capable beans and dependencies in Jakarta EE

When running in Jakarta EE, the container must extend the rules defined for managed beans in [\[passivation_validation\]](#) to EJB session beans.

2.4.3. Context management for built-in scopes in Jakarta EE

When running in Jakarta EE, the container must extend the rules defined in [\[builtin_contexts\]](#) and is also required to ensure the following rules for built-in context implementation.

The built-in request and application context objects are active during servlet, web service and EJB invocations, and the built in session and request context objects are active during servlet and web service invocations.

2.4.3.1. Request context lifecycle in Jakarta EE

When running in Jakarta EE the container must extend the rules defined in [\[request_context\]](#) and is also required to implement request context with the following rules.

The request context is active:

- ¥ during the `service()` method of any servlet in the web application, during the `doFilter()` method of any servlet filter and when the container calls any `ServletRequestListener` or `AsyncListener`,
- ¥ during any Jakarta EE web service invocation,
- ¥ during any remote method invocation of any EJB, during any asynchronous method invocation of any EJB, during any call to an EJB timeout method and during message delivery to any EJB

message-driven bean.

The request context is destroyed:

- ¥ at the end of the servlet request, after the `service()` method, all `doFilter()` methods, and all `requestDestroyed()` and `onComplete()` notifications return,
- ¥ after the web service invocation completes,
- ¥ after the EJB remote method invocation, asynchronous method invocation, timeout or message delivery completes if it did not already exist when the invocation occurred.

The payload of the event fired when the request context is initialized or destroyed is:

- ¥ the `ServletRequest` if the context is initialized or destroyed due to a servlet request, or
- ¥ the `ServletRequest` if the context is initialized or destroyed due to a web service invocation, or
- ¥ any `java.lang.Object` for other types of request.

2.4.3.2. Session context lifecycle in Jakarta EE

When running in Jakarta EE the container is required to implement session context with the following rules.

The session scope is active:

- ¥ during the `service()` method of any servlet in the web application, during the `doFilter()` method of any servlet filter and when the container calls any `HttpSessionListener`, `AsyncListener` or `ServletRequestListener`.

The session context is shared between all servlet requests that occur in the same HTTP session. The session context is destroyed when the `HTTPSession` times out, after all `HttpSessionListener`s have been called, and at the very end of any request in which `invalidate()` was called, after all filters and `ServletRequestListener`s have been called.

An event with qualifier `@Initialized(SessionScoped.class)` is synchronously fired when the session context is initialized. An event with qualifier `@BeforeDestroyed(SessionScoped.class)` is synchronously fired when the session context is about to be destroyed, i.e. before the actual destruction. An event with qualifier `@Destroyed(SessionScoped.class)` is synchronously fired when the session context is destroyed, i.e. after the actual destruction. The event payload is `jakarta.servlet.http.HttpSession`.

2.4.3.3. Application context lifecycle in Jakarta EE

When running in Jakarta EE the container must extend the rules defined in [\[application_context\]](#) and is also required to implement application context with the following rules.

The application scope is active:

- ¥ during the `service()` method of any servlet in the web application, during the `doFilter()` method of any servlet filter and when the container calls any `ServletContextListener`, `HttpSessionListener`, `AsyncListener` or `ServletRequestListener`,

- ¥ during any Jakarta EE web service invocation,
- ¥ during any asynchronous invocation of an event observer,
- ¥ during any remote method invocation of any EJB, during any asynchronous method invocation of any EJB, during any call to an EJB timeout method and during message delivery to any EJB message-driven bean,
- ¥ when the disposer method or `@PreDestroy` callback of any bean with any normal scope other than `@ApplicationScoped` is called, and
- ¥ during `@PostConstruct` callback of any bean.

The application context is shared between all servlet requests, web service invocations, asynchronous invocation of an event observer, EJB remote method invocations, EJB asynchronous 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 shut down.

The payload of the event fired when the application context is initialized or destroyed is:

- ¥ the `ServletContext` if the application is a web application deployed to a Servlet container, or
- ¥ any `java.lang.Object` for other types of application.

2.4.3.4. Conversation context lifecycle in Jakarta EE

When running in Jakarta EE the container is required to implement conversation context with the following rules.

The conversation scope is active during all Servlet requests.

An event with qualifier `@Initialized(ConversationScoped.class)` is synchronously fired when the conversation context is initialized. An event with qualifier `@BeforeDestroyed(ConversationScoped.class)` is synchronously fired when the conversation is about to be destroyed, i.e. before the actual destruction. An event with qualifier `@Destroyed(ConversationScoped.class)` is synchronously fired when the conversation is destroyed, i.e. after the actual destruction. The event payload is:

- ¥ the conversation id if the conversation context is destroyed and is not associated with a current Servlet request, or
- ¥ the `ServletRequest` if the application is a web application deployed to a Servlet container, or
- ¥ any `java.lang.Object` for other types of application.

The conversation context provides access to state associated with a particular *conversation*. Every Servlet request has an associated conversation. This association is managed automatically by the container according to the following rules:

- ¥ Any Servlet request has exactly one associated conversation.
- ¥ The container provides a filter with the name "CDI Conversation Filter", which may be mapped in `web.xml`, allowing the user alter when the conversation is associated with the servlet request. If this filter is not mapped in any `web.xml` in the application, the conversation associated with a

Servlet request is determined at the beginning of the request before calling any `service()` method of any servlet in the web application, calling the `doFilter()` method of any servlet filter in the web application and before the container calls any `ServletRequestListener` or `AsyncListener` in the web application.

- ¥ The implementation should determine the conversation associated with the Servlet request in a way that does not prevent other filters or servlet from setting the request character encoding or parsing the request body themselves.

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.

If the conversation associated with the current Servlet request is in the *transient* state at the end of a Servlet request, it is destroyed, and the conversation context is also destroyed.

If the conversation associated with the current Servlet request is in the *long-running* state at the end of a Servlet request, it is not destroyed. The long-running conversation associated with a request may be propagated to any Servlet request via use of a request parameter named `cid` containing the unique identifier of the conversation. In this case, the application must manage this request parameter.

If the current Servlet request is a JSF request, and the conversation is in *long-running* state, it is propagated 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 (a redirect resulting from a navigation rule or JSF `NavigationHandler`) 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 request parameter named `cid` containing the unique identifier of the conversation.

When no conversation is propagated to a Servlet request, or if a request parameter named `conversationPropagation` has the value `none` 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, after the servlet `service()` method completes.

¥ The container is permitted to arbitrarily destroy any long-running conversation that is associated with no current Servlet request, in order to conserve resources.

The *conversation timeout*, which may be specified by calling `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.

If the propagated conversation cannot be restored, the container must associate the request with a new transient conversation and throw an exception of type `jakarta.enterprise.context.NonexistentConversationException`.

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. If the container rejects a request, it must associate the request with a new transient conversation and throw an exception of type `jakarta.enterprise.context.BusyConversationException`.

2.5. Lifecycle of contextual instances

2.5.1. Container invocations and interception in Jakarta EE

When the application invokes:

¥ a business method of a session bean via an EJB remote or local reference,

the invocation is treated as a *business method invocation*.

When running in Jakarta EE, the container must extend the rules defined in [\[biz_method\]](#), with:

¥ Invocation of EJB timer service timeouts by the container are not business method invocations, but are intercepted by interceptors for EJB timeouts.

¥ Only an invocation of business method on an EJB session bean is subject to EJB services such as declarative transaction management, concurrency, security and asynchronicity, as defined by the EJB specification.

¥ Additionally, invocations of message listener methods of message-driven beans during message delivery are passed through method interceptors.

2.5.1.1. Lifecycle of EJB stateful session beans

When the `create()` method of a `Bean` object that represents an EJB stateful session bean that is called, the container creates and returns a container-specific internal local reference to a new EJB session bean instance. The reference must be passivation capable. This reference is not directly exposed to the application. When the `create()` method of a `Bean` object that represents an EJB stateful session bean that is called, the container creates and returns a container-specific internal local reference to a new EJB session bean instance. The reference must be passivation capable. This reference is not directly exposed to the application.

Before injecting or returning a contextual instance to the application, the container transforms its

internal reference into an object that implements the bean types expected by the application and delegates method invocations to the underlying EJB stateful session bean instance. This object must be passivation capable.

When the `destroy()` method is called, and if the underlying EJB was not already removed by direct invocation of a remove method by the application, the container removes the EJB stateful session bean. The `@PreDestroy` callback must be invoked by the container.

Note that the container performs additional work when the underlying EJB is created and removed, as defined in [\[injection\]](#)

2.5.1.2. Lifecycle of EJB stateless and singleton session beans

When the `create()` method of a `Bean` object that represents an EJB stateless session or singleton session bean is called, the container creates and returns a container-specific internal local reference to the EJB session bean. This reference is not directly exposed to the application.

Before injecting or returning a contextual instance to the application, the container transforms its internal reference into an object that implements the bean types expected by the application and delegates method invocations to the underlying EJB session bean. This object must be passivation capable.

When the `destroy()` method is called, the container simply discards this internal reference.

Note that the container performs additional work when the underlying EJB is created and removed, as defined in [\[injection\]](#)

2.5.1.3. Lifecycle of resources

When the `create()` method of a `Bean` object that represents a resource is called, the container creates and returns a container-specific internal reference to the Jakarta EE component environment resource, entity manager, entity manager factory, remote EJB instance or web service reference. This reference is not directly exposed to the application.

Before injecting or returning a contextual instance to the application, the container transforms its internal reference into an object that implements the bean types expected by the application and delegates method invocations to the underlying resource, entity manager, entity manager factory, remote EJB instance or web service reference. This object must be passivation capable.

The container must perform ordinary Jakarta EE component environment injection upon any non-static field that functions as a resource declaration, as defined by the Jakarta EE platform and Common Annotations for the Java platform specifications. The container is not required to perform Jakarta EE component environment injection upon a static field. Portable applications should not rely upon the value of a static field that functions as a resource declaration.

References to EJBs and web services are always dependent scoped and a new instance must be obtained for every injection performed.

For an entity manager associated with a resource definition, it must behave as though it were injected directly using `@PersistenceContext`.

When the `destroy()` method of a bean which represents a remote stateful EJB reference is called, the container will *not* automatically destroy the EJB reference. The application must explicitly call the method annotated `@Remove`. This behavior differs to that specified in [Lifecycle of EJB stateful session beans](#) for beans which represent a local stateful EJB reference

2.6. Decorators in Jakarta EE

When running in Jakarta EE, the container must extend the rules defined for managed beans in [\[decorators\]](#) to EJB session beans.

2.6.1. Decorator beans in Jakarta EE

Decorators of an EJB session bean must comply with the bean provider programming restrictions defined by the EJB specification. Decorators of an EJB stateful session bean must comply with the rules for instance passivation and conversational state defined by the EJB specification.

2.7. Interceptor bindings in Jakarta EE

EJB session and message-driven beans support interception as defined in [\[interceptors\]](#).

2.7.1. Interceptor enablement and ordering in Jakarta EE

When running in Jakarta EE, the container must extend the rules defined in [\[enabled_interceptors\]](#) and also ensure that:

- ¥ Interceptors declared using interceptor bindings are called after interceptors declared using the `@Interceptor` annotation (or using the corresponding element of a deployment descriptor).
- ¥ Interceptors declared using interceptor bindings are called before any around-invoke, around-timeout, or lifecycle event callback methods declared on the target class or any superclass of the target class.

2.7.2. Interceptor resolution in Jakarta EE

For a custom implementation of the `Interceptor` interface defined in [\[interceptor\]](#), the container also calls `intercepts()` to determine if the interceptor intercepts an EJB timeout method invocation.

2.8. Events in Jakarta EE

2.8.1. Observer methods in EJB session beans

An observer method may also be a non-abstract method of an EJB session bean class. It must be either a business method exposed by a local business interface of the EJB or a static method of the bean class.

2.8.1.1. Declaring an observer method in an EJB

If a non-static method of a session bean class has a parameter annotated `@Observes` or

`@ObservesAsync`, and the method is not a business method exposed by a local business interface of the EJB, the container automatically detects the problem and treats it as a definition error.

2.8.1.2. Observer method invocation context in Jakarta EE

When running in Jakarta EE, the container must extend the rules defined in [\[observer_method_invocation_context\]](#) and must also ensure that all kinds of observers are called in the same client security context as the invocation of `Event.fire()` or `Event.fireAsync()`.

The transaction and security contexts for a business method exposed by a local business interface of an EJB session bean also depend upon the transaction attribute and `@RunAs` descriptor, if any.

2.9. Portable extensions in Jakarta EE

2.9.1. The `Bean` interface in Jakarta EE

When running in Jakarta EE, the container must extend the rules defined in [\[bean\]](#) for managed bean to EJB session bean.

2.9.1.1. The `Interceptor` interface in Jakarta EE

When running in Jakarta EE, the container must extend the rules defined in [\[interceptor\]](#) and must also ensure that

`PRE_PASSIVATE`, `POST_ACTIVATE` and `AROUND_TIMEOUT` `InterceptorType` values are linked to EJB lifecycle callback or timeout method.

2.9.2. `InjectionTarget` interface in Jakarta EE

When running in Jakarta EE, the container must extend the rules defined for `InjectionTarget` in [\[injectiontarget\]](#) and must also ensure that:

- ¥ when `inject()` is called, the container performs Jakarta EE component environment injection, according to the semantics required by the Jakarta EE platform specification, sets the value of all injected fields, and calls all initializer methods, as defined in [Injection of fields and initializer methods in Jakarta EE](#).
- ¥ `@PostConstruct` callback is called according to the semantics required by the Jakarta EE platform specification.
- ¥ `@PreDestroy` callback is called according to the semantics required by the Jakarta EE platform specification.

2.9.3. The `BeanManager` object in Jakarta EE

2.9.3.1. Obtaining a reference to the CDI container in Jakarta EE

A Jakarta EE container is required to provide a CDI provider that will allow access to the current container for any Jakarta EE application or Jakarta EE module which contains enabled beans.

Jakarta EE Components may obtain an instance of `BeanManager` from JNDI by looking up the name `java:comp/BeanManager`.

2.9.4. Alternative metadata sources and EJB

When running in Jakarta EE, the container must extend the rules defined in [\[alternative_metadata_sources\]](#) and ensure that:

- ¥ when an `AnnotatedType` represents an EJB session bean class, `Annotated.getTypeClosure()` must return the EJB session bean types as defined in [Bean types of a session bean](#).

2.9.5. Addition to Container lifecycle events in Jakarta EE

2.9.5.1. `ProcessAnnotatedType` event in Jakarta EE

When running in Jakarta EE, the container must extend the rules defined in [\[process_annotated_type\]](#) to Jakarta EE component and EJB session bean classes.

2.9.5.2. `ProcessInjectionPoint` event and EJB

When running in Jakarta EE, the container must also fire an event for every injection point of every Jakarta EE component class supporting injection that may be instantiated by the container at runtime, including every EJB session or message-driven bean.

2.9.5.3. `ProcessInjectionTarget` event and EJB

When running in Jakarta EE, the container must also fire an event for every Jakarta EE component class supporting injection that may be instantiated by the container at runtime, including every EJB session or message-driven bean.

The container must extend the rules defined in [\[process_injection_target\]](#) for managed bean to EJB session bean and other Jakarta EE component class supporting injection.

For example, this observer decorates the `InjectionTarget` for all servlets.

```
<T extends Servlet> void decorateServlet(@Observes ProcessInjectionTarget<T> pit) {  
    pit.setInjectionTarget( decorate( pit.getInjectionTarget() ) );  
}
```

2.9.5.4. `ProcessBeanAttributes` event and EJB

When running in Jakarta EE, the container must extend the rules defined in [\[process_bean_attributes\]](#) to EJB session bean.

2.9.5.5. `ProcessBean` event and EJB

In addition to definition given in [\[process_bean\]](#) the following apply:

- ¥ For a session bean with bean class `X`, the container must raise an event of type

`ProcessSessionBean<X>`.

Resources are considered to be producer fields.

When running in Jakarta EE, the interface `jakarta.enterprise.inject.spi.ProcessBean` is also a supertype of `jakarta.enterprise.inject.spi.ProcessSessionBean`:

```
public interface ProcessSessionBean<X>
    extends ProcessManagedBean<Object> {
    public String getEjbName();
    public SessionBeanType getSessionBeanType();
}
```

¥ `getEjbName()` returns the EJB name of the session bean.

¥ `getSessionBeanType()` returns a `jakarta.enterprise.inject.spi.SessionBeanType` representing the kind of session bean.

```
public enum SessionBeanType { STATELESS, STATEFUL, SINGLETON }
```

2.10. Packaging and deployment in Jakarta EE

2.10.1. Bean archive with EJB Session Beans

When running in Jakarta EE, the container must extend the rules defined in [\[bean_archive_full\]](#) with:

¥ An *implicit bean archive* may also contain EJB session beans, and

¥ EJB session bean should be considered as bean class with bean defining annotation when determining if an archive is an *implicit bean archive*.

When determining which archives are bean archives, the container must also consider:

¥ EJB jars or application client jars

¥ The `WEB-INF/classes` directory of a war

The container is not required to support application client jar bean archives.

A Jakarta EE container is required by the Jakarta EE specification to support Jakarta EE modules.

In a war, the `beans.xml` file must be named:

¥ `WEB-INF/beans.xml` or `WEB-INF/classes/META-INF/beans.xml`.

If a war has a file named `beans.xml` in both the `WEB-INF` directory and in the `WEB-INF/classes/META-INF` directory, then non-portable behavior results. Portable applications must have a `beans.xml` file in only one of the `WEB-INF` or the `WEB-INF/classes/META-INF` directories.

The following additional rules apply regarding container search for beans:

- ¥ In an application deployed as an ear, the container searches every bean archive bundled with or referenced by the ear, including bean archives bundled with or referenced by wars, EJB jars and rars contained in the ear. The bean archives might be library jars, EJB jars or war `WEB-INF/classes` directories.
- ¥ In an application deployed as a war, the container searches every bean archive bundled with or referenced by the war. The bean archives might be library jars or the `WEB-INF/classes` directory.
- ¥ In an application deployed as an EJB jar, the container searches the EJB jar, if it is a bean archive, and every bean archive referenced by the EJB jar.
- ¥ In an application deployed as a rar, the container searches every bean archive bundled with or referenced by the rar.
- ¥ An embeddable EJB container searches each bean archive in the JVM classpath that is listed in the value of the embeddable container initialization property `jakarta.ejb.embeddable.modules`, or every bean archive in the JVM classpath if the property is not specified. The bean archives might be directories, library jars or EJB jars.

2.10.2. Type and Bean discovery for EJB

In Jakarta EE, the container automatically discovers EJB session beans and other Jakarta EE component class supporting injection, in bean archives like it does for managed bean as defined in [\[type_bean_discovery_full\]](#).

2.10.2.1. Bean discovery in Jakarta EE

When running in Jakarta EE, the container must extend the rules defined in [\[bean_discovery_steps_full\]](#) and must also discover each EJB session bean.

2.10.2.2. Trimmed bean archive in Jakarta EE

When running in Jakarta EE, the container must extend the rules defined in [\[trimmed_bean_archive\]](#) and must ensure that EJB session beans are not removed from the set of discovered types.

2.11. Integration with Unified EL

2.11.1. Bean name resolution in EL expressions

The container must provide a Unified EL `ELResolver` to the servlet engine and JSF implementation that resolves bean names using the rules of name resolution defined in [\[name_resolution\]](#) and resolving ambiguities according to [\[ambig_names\]](#).

- ¥ If a name used in an EL expression does not resolve to any bean, the `ELResolver` must return a null value.
- ¥ Otherwise, if a name used in an EL expression resolves to exactly one bean, the `ELResolver` must return a contextual instance of the bean, as defined in [\[contextual_instance\]](#).

2.11.2. Unified EL integration API

Since CDI version 4.1, the Unified EL integration API, which is part of the `BeanManager` API, is deprecated. The relevant methods are placed in a new interface `jakarta.enterprise.inject.spi.el.ELAwareBeanManager`, which is present in a new supplemental CDI API artifact: `jakarta.enterprise:jakarta.enterprise.cdi-el-api`.

2.11.2.1. Obtaining `ELAwareBeanManager`

The `BeanManager` implementation in Jakarta EE must also implement `ELAwareBeanManager`. All rules that apply to the `BeanManager`, as specified in [\[beanmanager\]](#) and [The BeanManager object in Jakarta EE](#), also apply to `ELAwareBeanManager`.

It follows that the container provides a built-in bean with bean type `ELAwareBeanManager`, scope `@Dependent` and qualifier `@Default`, which is a passivation capable dependency as defined in [\[passivation_capable_dependency\]](#). It also follows that an `ELAwareBeanManager` may be obtained by using `CDI.current().getBeanManager()` and casting.

The EL-related methods of `ELAwareBeanManager` may be called at any time during the execution of the application.

2.11.2.2. Obtaining the `ELResolver`

The method `ELAwareBeanManager.getELResolver()` returns the `jakarta.el.ELResolver` specified in [Bean name resolution in EL expressions](#). This `ELResolver` is used to satisfy the rules defined in [Bean names in Jakarta EE](#).

```
public ELResolver getELResolver();
```

2.11.2.3. Wrapping a Unified EL `ExpressionFactory`

The method `ELAwareBeanManager.wrapExpressionFactory()` returns a wrapper `jakarta.el.ExpressionFactory` that delegates `MethodExpression` and `ValueExpression` creation to the given `ExpressionFactory`. When a Unified EL expression is evaluated using a `MethodExpression` or `ValueExpression` returned by the wrapper `ExpressionFactory`, the rules defined in [Dependent pseudo-scope and Unified EL](#) are enforced by the container.

```
public ExpressionFactory wrapExpressionFactory(ExpressionFactory expressionFactory);
```

2.12. CDI Specification References

References from the CDI EE Integration specification to the online CDI core specification are provided in the following sections.

¥ 2.1. Concepts

¥ 2.1.1. Functionality provided by the container to the bean

- ¥ 2.1.2.1. Legal bean types
- ¥ 2.1.4. Scopes
- ¥ 2.1.5. Default bean discovery mode
- ¥ 2.1.6.2. Default bean names
- ¥ 2.2. Programming model
 - ¥ 2.2.1.1. Which Java classes are managed beans?
 - ¥ 2.2.2. Producer methods
 - ¥ 2.2.2.2. Declaring a producer method
 - ¥ 2.2.3. Producer fields
 - ¥ 2.2.3.1. Bean types of a producer field
 - ¥ 2.2.3.2. Declaring a producer field
 - ¥ 2.2.4. Disposer methods
 - ¥ 2.2.4.2. Declaring a disposer method
 - ¥ 2.2.6. Injected fields
 - ¥ 2.2.7. Initializer methods
- ¥ 2.3.1. Inheritance of type-level metadata
- ¥ 2.3.2. Inheritance of member-level metadata
- ¥ 2.4. Dependency injection and lookup
 - ¥ 2.4.1. Modularity
 - ¥ 2.4.2.2. Unsatisfied and ambiguous dependencies
 - ¥ 2.4.3. Name resolution
 - ¥ 2.4.3.1. Ambiguous names
 - ¥ 2.4.5. Dependency injection
 - ¥ 2.4.5.1. Injection using the bean constructor
 - ¥ 2.4.5.2. Injection of fields and initializer methods
 - ¥ 2.4.5.3. Destruction of dependent objects
 - ¥ 2.4.5.8. Bean metadata

- ¥ 2.5. Scopes and contexts
 - ¥ 2.5.4. Dependent pseudo-scope
 - ¥ 2.5.4.1. Dependent objects
 - ¥ 2.5.4.2. Destruction of objects with scope `@Dependent`
 - ¥ 2.5.5.3. Contextual instance of a bean
- ¥ 2.5.6. Context management for built-in scopes
 - ¥ 2.5.6.1. Request context lifecycle
 - ¥ 2.5.6.2. Application context lifecycle
- ¥ 2.6.2. Container invocations and interception
- ¥ 2.7. Interceptor bindings
- ¥ 2.8.5.3. Observer method invocation context
- ¥ 3.10.1. Bean archives in CDI Full
- ¥ 3.10.4. Type and Bean discovery in CDI Full
 - ¥ 3.10.4.3. Trimmed bean archive
 - ¥ 3.10.4.4. Bean discovery in CDI Full
- ¥ 3.2.3. Specialization
- ¥ 3.4.5.3. Passivation capable dependencies
- ¥ 3.4.5.5. Validation of passivation capable beans and dependencies
- ¥ 3.6.2. Interceptor enablement and ordering in CDI Full
- ¥ 3.7. Decorators
- ¥ 3.9.1. The Bean interface
 - ¥ 3.9.1.2. The Interceptor interface
- ¥ 3.9.2. The Producer and InjectionTarget interfaces
- ¥ 3.9.3. The BeanManager object
- ¥ 3.9.4. Alternative metadata sources
- ¥ 3.9.5.10. ProcessBean event
- ¥ 3.9.5.6. ProcessAnnotatedType event

¥ 3.9.5.8. ProcessInjectionTarget event

¥ 3.9.5.9. ProcessBeanAttributes event

Chapter 3. Web Profile Definition

This chapter defines the contents of the Jakarta^a EE 10 Web Profile.

3.1. Required Components

The following technologies are required components of the Web Profile:

- ¥ Jakarta Annotations 2.1*
- ¥ Jakarta Authentication 3.0*
- ¥ Jakarta Bean Validation 3.0
- ¥ Jakarta Concurrency 3.0*
- ¥ Jakarta Contexts and Dependency Injection 4.0*
- ¥ Jakarta Debugging Support for Other Languages 2.0
- ¥ Jakarta Dependency Injection 2.0
- ¥ Jakarta Enterprise Beans 4.0 Lite
- ¥ Jakarta Expression Language 5.0*
- ¥ Jakarta Interceptors 2.1*
- ¥ Jakarta JSON Binding 3.0*
- ¥ Jakarta JSON Processing 2.1*
- ¥ Jakarta Persistence 3.1*
- ¥ Jakarta RESTful Web Services 3.1*
- ¥ Jakarta Security 3.0*
- ¥ Jakarta Server Faces 4.0*
- ¥ Jakarta Server Pages 3.1*
- ¥ Jakarta Servlet 6.0*
- ¥ Jakarta Standard Tag Library 3.0*
- ¥ Jakarta Transactions 2.0
- ¥ Jakarta WebSocket 2.1*

Note: technologies with an asterik after them represent updated versions.

3.2. Optional Components

There are no optional components in the Web Profile.

Web Profile products may support some of the technologies present in the full Jakarta EE Platform and not already listed in [Required Components](#), consistently with their compatibility requirements.

3.3. Additional Requirements

Web Profile products must support the deployment of Jakarta EE web application modules (*.war* files). No other modules types are required to be supported.

The following functionality is required to be supported in Web Profile products:

- ¥ Resource annotations defined by the Common Annotations specification (*Resource*, *Resources*)
- ¥ JNDI `java:0` naming context as described in the JNDI section of the Platform specification
- ¥ Jakarta Transactions

Appendix A: Revision History

A.1. Changes in Final Release for EE10

¥ Major and minor updates to most specifications.

¥ Addition of Jakarta Concurrency to Web Profile.

A.1.1. Editorial Changes for EE10

¥ Updated [Related Documents](#) for the updated Specifications in Jakarta EE 10.

A.2. Changes in Final Release for EE9.1

A.2.1. Editorial Changes

¥ Updated [Related Documents](#) for the updated Specifications in Jakarta EE 9.1.

Appendix B: Related Documents

This specification refers to the following documents. The terms used to refer to the documents in this specification are included in parentheses.

Jakarta^a EE Platform Specification Version 10.0. Available at: <https://jakarta.ee/specifications/platform/10.0>

Java^a Platform, Standard Edition (Java SE specification), v11. Available at: <https://www.jcp.org/en/jsr/detail?id=384>

Java^a Platform, Standard Edition, v11 API Specification. Available at: <https://docs.oracle.com/en/java/javase/11/>

Jakarta^a Enterprise Beans Specification, Version 4.0. Available at: <https://jakarta.ee/specifications/enterprise-beans/4.0>

Jakarta^a Server Pages Specification, Version 3.1. Available at: <https://jakarta.ee/specifications/pages/3.1>

Jakarta^a Expression Language Specification, Version 5.0. Available at: <https://jakarta.ee/specifications/expression-language/5.0>

Jakarta^a Servlet Specification, Version 6.0. Available at: <https://jakarta.ee/specifications/servlet/6.0>

Jakarta^a Transaction Specification, Version 2.0. Available at: <https://jakarta.ee/specifications/transactions/2.0>

Jakarta^a RESTful Web Services Specification, Version 3.1. Available at: <https://jakarta.ee/specifications/restful-ws/3.1>

Jakarta^a Annotations Specification, Version 2.1. Available at: <https://jakarta.ee/specifications/annotations/2.1>

Jakarta^a Debugging Support for Other Languages Specification, Version 2.0. Available at: <https://jakarta.ee/specifications/debugging/2.0>

Jakarta^a Standard Tag Library Specification, Version 3.0. Available at: <https://jakarta.ee/specifications/tags/3.0>

Jakarta^a Server Faces Specification, Version 4.0. Available at: <https://jakarta.ee/specifications/faces/4.0>

Jakarta^a Persistence Specification, Version 3.1. Available at: <https://jakarta.ee/specifications/persistence/3.1>

Jakarta^a Bean Validation Specification, Version 3.0. Available at: <https://jakarta.ee/specifications/bean-validation/3.0>

Jakarta^a Interceptors Specification, Version 2.1. Available at: <https://jakarta.ee/specifications/interceptors/2.1>

Jakarta^a Contexts and Dependency Injection Specification, Version 4.0. Available at: <https://jakarta.ee/specifications/cdi/4.0>

Jakarta^a Dependency Injection Specification, Version 2.0. Available at: <https://jakarta.ee/specifications/dependency-injection/2.0>

Jakarta^a WebSocket Specification, Version 2.1. Available at: <https://jakarta.ee/specifications/websocket/2.1>

Jakarta^a JSON Processing Specification, Version 2.1. Available at: <https://jakarta.ee/specifications/jsonp/2.1>

Jakarta^a JSON Binding Specification, Version 3.0. Available at: <https://jakarta.ee/specifications/jsonb/3.0>

Jakarta^a Security Specification, Version 3.0. Available at: <https://jakarta.ee/specifications/security/3.0>

Jakarta^a Authentication Specification, Version 3.0. Available at: <https://jakarta.ee/specifications/authentication/3.0>

Jakarta^a Concurrency Specification, Version 3.0. Available at: <https://jakarta.ee/specifications/concurrency/3.0>

Chapter 4. Jakarta EE 10 Web Profile Dependencies

Failed to generate image: Could not find the 'dot' executable in PATH; add it to the PATH or specify its location using the 'graphvizdot' document attribute
digraph EE10WebProfile {

```
# Map a short node id to the label with version
wp [label="jakarta-web-api"];
servlet [label="servlet: 6.0.0"];
jsp [label="servlet.jsp: 3.1.0"];
el [label="el: 5.0.0"];
jstl [label="servlet.jsp.jstl: 3.0.0-RC1"];
jaxb [label="xml.bind: 4.0.0-RC3"];
faces [label="faces: 4.0.0-M2"];
jaxrs [label="jaxrsr: 3.1.0"];
websocket [label="websocket: 2.1.0"];
jsonp [label="jsonp: 2.1.0"];
jsonb [label="jsonb: 3.0.0-RC1"];
annotations [label="annotations: 2.1.0"];
ejb [label="ejb: 4.0.0"];
jta [label="jta: 2.0.1-RC1"];
jpa [label="jpa: 3.1.0-RC2"];
bv [label="bv: 3.0.1"];
interceptors [label="interceptors: 2.1.0"];
cdi [label="cdi: 4.0.0-RC5"];
di [label="di: 2.0.1"];
auth [label="auth: 2.1.0-RC1"];
jacc [label="jacc: 2.0.0"];
xml_bind [label="xml.bind: 4.0.0-RC3"];
activation [label="activation: 2.1.0"];
```

```
wp -> servlet;
wp -> jsp;
wp -> el;
wp -> jstl;
wp -> faces;
wp -> websocket;
wp -> jta;
wp -> jpa;
wp -> jsonb;
wp -> jsonp;
wp -> jaxrs;
wp -> annotations;
wp -> bv;
wp -> interceptors;
wp -> cdi;
wp -> auth;
```

```
wp -> jacc;

# JSP
jsp -> servlet;
jsp -> el;

# JSTL
jstl -> servlet ;
jstl -> jsp ;
jstl -> el ;
jstl -> jaxb ;

# Faces
faces -> servlet;
faces -> websocket;
faces -> el;
faces -> cdi;
faces -> bv;
faces -> jta;
faces -> jsp;
faces -> jstl;
faces -> jsonp;
faces -> ejb;
faces -> jpa;
faces -> annotations;
faces -> jaxb;

# jsonb
jsonb -> jsonp;

# jaxrs
jaxrs -> xml_bind;
jaxrs -> activation;

# interceptors
interceptors -> annotations;

# jacc
jacc -> servlet;

# ejb
ejb -> jta;

# jta
jta -> cdi ;
jta -> interceptors ;

# CDI
cdi -> el ;
cdi -> interceptors ;
cdi -> di ;
```

```
cdi -> jta [label="javadoc"];  
cdi -> ejb [label="javadoc"];  
}
```