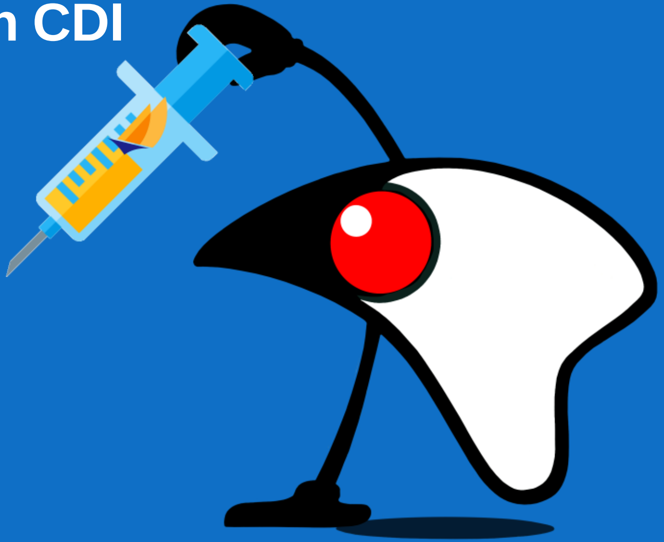
Dependency injection in JEE application with CDI



Matschieu, 2023

Introduction

What is dependency injection?

In software engineering, **dependency injection** is a design pattern in which an object or function receives other objects or functions that it depends on.

A form of inversion of control, dependency injection aims to **separate** the concerns of **constructing objects** and **using them**, leading to **loosely coupled programs**.

The pattern ensures that an object or function which wants to use a given service should not have to know how to construct those services. Instead, the receiving "client" (object or function) is provided with its dependencies by external code (an "injector"), which it is not aware of.

Introduction

Dependency injection in Java

The **EJB** specification was originally developed in 1997 by IBM and later adopted by Sun Microsystems in 1999 and enhanced under the Java Community Process.

In 2004, the first version of **Spring** was released and provides a container able to manage dependency injection.

In 2006, Java 6 has introduced a feature for discovering and loading implementations matching a given interface: **Service Provider Interface (SPI)**. This is a very basic dependency injection mechanism.

In 2009, Java EE 6 has introduced **CDI (Contexts and Dependency Injection)**, a standard dependency injection framework. It allows to manage the lifecycle of stateful components via domain-specific lifecycle contexts and inject components (services) into client objects in a type-safe way.

Introduction

History overview of dependency injection in JEE

Year	EJB	CDI	JEE	Content
1996	1.0/1.1		J2EE 1.2	
2001	2.0		J2EE 1.3	
2006	3.0		Java EE 5	Java EE 5 (EoD) 3.1 2009 Java EE 9
2009		1.0	Java EE6	Integration dans JSF
2013	3.2	1.1	Java EE7	Add interceptor, decorators, alternatives. Specially JTA and @AroundConstruct interceptors. SPI and portable extensions
2017		2.0	Java EE8	Typesafe injection, Lifecycle management; contexts; scopes; lifecycle callbacks, Event firing and delivery, Producers, Alernatives to configure available beans, Exstensible model and rich SPI
2019		2.0	Jakarta EE 8	dependencies name changed to jakarta
2020	4.0	3.0	Jakarta EE 9	Package updated form javax.enterprise.* to jakarta.enterprise.* Removal of methods relying on java.security.Identity, methods relying on JAX-RPC, deprecated EJBContext.getEnvironment() method, Support for Distributed Interoperability + Mark optional EJB 2.x API Group

Introduction

Jakarta EE vs Spring

JAKARTA™ EE

- Offers dependency injection mechanism (EJB/CDI)
- Jakarta EE are standard specifications, including EJB/CDI, with several implementations.
- Needs an application server to run (RedHat JBoss for example)

spring®

- Offers dependency injection mechanism (IoC container *ApplicationContext*)
- Spring is an open-source proprietary implementation made by Pivotal
- Spring is NOT a Jakarta EE implementation

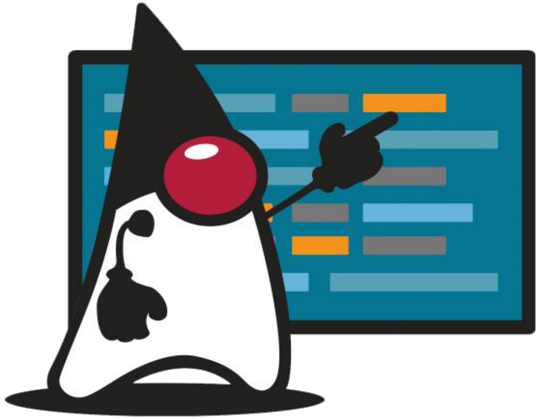
Technically both are similar, but they do not have the same model (Video explaining differences can be found [here](#))

Dependency injection in JEE

JSR 365: Contexts and Dependency Injection for Java 2.0

CDI: Context and Dependency Injection

Agenda



CDI: Context and Dependency Injection

Introduction

- CDI 1.1 is specified by JSR 346: Contexts and Dependency Injection for JavaTM EE 1.1 based on:
  - JSR 330, Dependency Injection for Java
  - JSR 342,The Managed Beans specification, an offshoot of the Java EE 7 platform specification
- **CDI 2.0 is specified by JSR 365: Contexts and Dependency Injection for Java 2.0**

CDI provides several packages containing classes and annotations to manage DI:

Packages < JEE9	Packages >= JEE9	Description
javax.inject	jakarta.inject	Core dependency injection API
javax.enterprise.inject	jakarta.enterprise.inject	Core CDI API
javax.enterprise.context	jakarta.enterprise.context	Scopes and contextual APIs
javax.enterprise.event	jakarta.enterprise.event	Events and observers APIs
javax.interceptor	jakarta.interceptor	Interceptor APIs
javax.decorator	jakarta.decorator	Decorator APIs
javax.enterprise.util	jakarta.enterprise.util	CDI utility package

CDI: Context and Dependency Injection

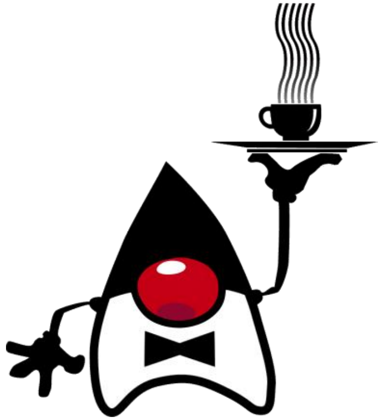
Introduction

- A **bean** is a source of contextual objects which define application state and/or logic.
- These objects are called *contextual instances of the bean*.
- The **container** creates and destroys these instances and associates them with the appropriate context.
- Contextual instances of a bean may be **injected** into other objects (including other bean instances) that execute in the same context.
- A bean may bear **metadata** defining its **lifecycle** and interactions with other beans.
- The **bean types and qualifiers of a bean** determine where its instances will be injected by the container

CDI: Context and Dependency Injection

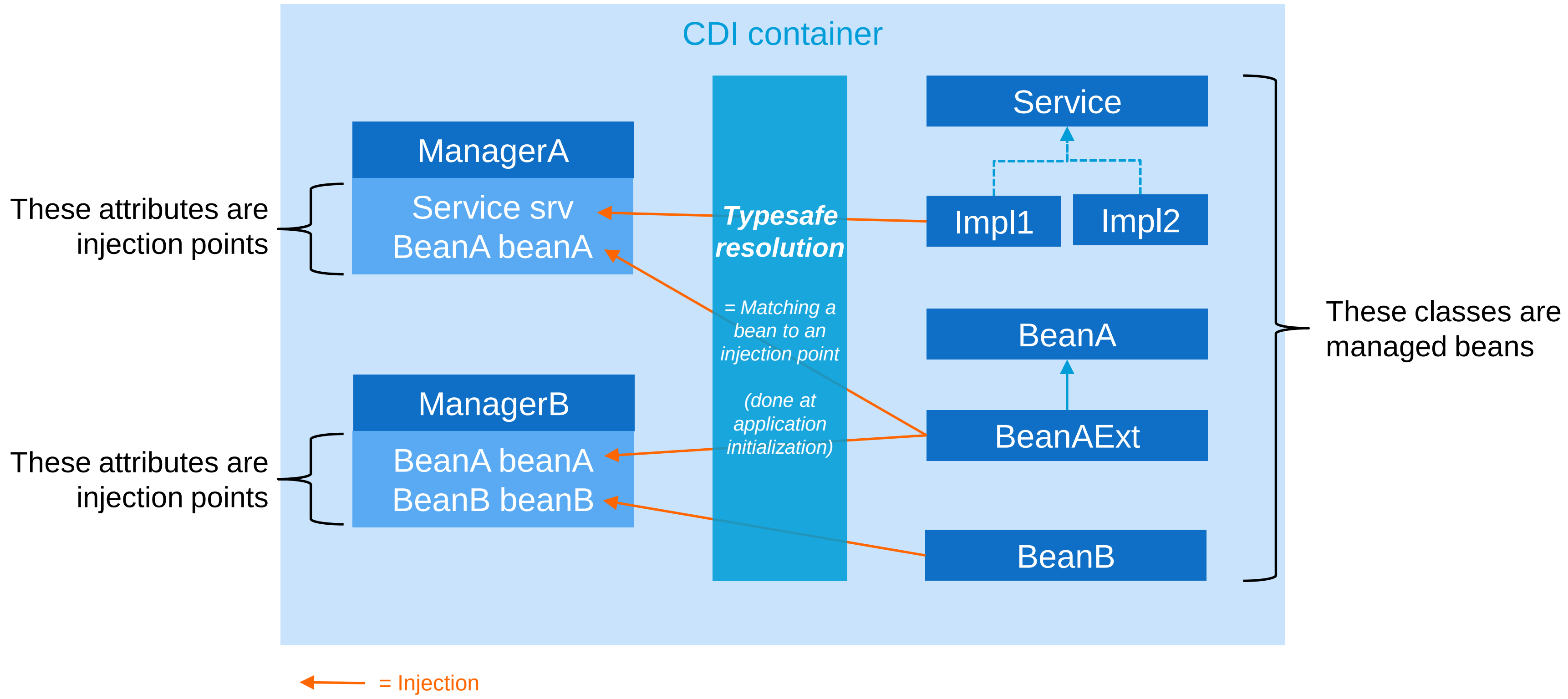
Introduction

- A bean is provided by the container with the following capabilities:
  - transparent **creation** and **destruction** and scoping to a particular context
  - **scoped resolution** by bean type and **qualifier annotation type** when injected into a Java-based client
  - **lifecycle callbacks** and **automatic injection** of other bean instances
  - **method interception**, callback interception, and decoration
  - **event notification**



CDI: Context and Dependency Injection

Introduction


CDI container
These attributes are injection points
ManagerA
Service srv
BeanA beanA
These attributes are injection points
ManagerB
BeanA beanA
BeanB beanB
Typesafe resolution
= Matching a bean to an injection point
(done at application initialization)
Service
Impl1
Impl2
BeanA
BeanAExt
BeanB
These classes are managed beans
= Injection

Managed beans

CDI: Context and Dependency Injection

Managed beans

A managed bean is a class with a constructor that takes no parameters. It does not require any special annotations.

The following classes are beans:

```
public class Card { ... }

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

A managed bean may extend another managed bean:

```
public class LoginAction { ... }
public class MockLoginAction extends LoginAction { ... }
```

If the managed bean does not have a constructor that takes no parameters, it must have a constructor annotated @Inject. No additional special annotations are required.

CDI: Context and Dependency Injection

Managed beans

A bean (or a package) can be excluded from the container management by using the annotation **@javax.enterprise.inject.Vetoed**.

```
@Vetoed
public class Card { ... }
```

Note: This should be used on a JPA entity to avoid injection which could lead to unexpected behaviour because a JPA entity is already managed by JPA and has its own lifecycle.

Bean types

CDI: Context and Dependency Injection

Bean types

A bean type defines a client-visible type of the bean. A bean may have multiple bean types.

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

This bean has 4 types, including implicit type **java.lang.Object**.

CDI: Context and Dependency Injection

Bean types

The bean types of a bean may be restricted by annotating the bean class or producer method or field with the annotation **@javax.enterprise.inject.Typed**.

```
@Typed(Shop.class)
public class BookShop
        extends Business
        implements Shop<Book> {
    ...
}
```

This bean has 2 types, including implicit type **java.lang.Object**.

CDI: Context and Dependency Injection

Bean types

Almost any Java type may be a bean type of a bean:

- A bean type may be an interface, a concrete class or an abstract class, and may be declared final or have final methods.
- A bean type may be a parameterized type with actual type parameters and type variables.
- A bean type may be an array type. Two array types are considered identical only if the element type is identical.
- A bean type may be a primitive type. Primitive types are considered to be identical to their corresponding wrapper types in java.lang.
- A bean type may be a raw type.

However, some Java types are <u>not legal bean types</u> :

- A type variable is not a legal bean type.
- A parameterized type that contains a wildcard type parameter is not a legal bean type.
- An array type whose component type is not a legal bean type.

CDI configuration

CDI: Context and Dependency Injection

CDI configuration

CDI beans can be **enabled** using only one file: **beans.xml** in :

- **WEB-INF/beans.xml** in web application
- **META-INF/beans.xml** in jar files

CDI can also be configured using this file or directly by using annotations in the code.

This file must contain at least the `<bean>` tag and it allows to configure the beans.

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/beans_2_0.xsd"
  bean-discovery-mode="all"
  version="2.0">
  <!-- optional beans config here -->
</beans>
```

CDI: Context and Dependency Injection

CDI configuration

- CDI API is provided by a **JakartaEE** application server
- CDI implementation is provided by **JBoss (Weld)**
- **beans.xml** must be present in the war or in a jar file.
- CDI container is started at the JBoss startup

- CDI API & implementation (**Weld**) must be added in the classpath (weld-core-impl)
- **beans.xml** must be present in the classpath
- CDI container must be started manually:

```
Weld weld = new Weld();
WeldContainer container = weld.initialize();
...
container.shutdown();
```

- CDI API & implementation (**Weld**) must be added in the classpath (weld-core-impl)
- **beans.xml** must be present in the classpath
- CDI integration (weld-junit5) can be used in test classes:

```
@EnableWeld
public class WeldTest {

  @WeldSetup
  private WeldInitiator weld =
         WeldInitiator.of(WeldInitiator
           .createWeld().enableDiscovery());
```

Bean injection

CDI: Context and Dependency Injection

Bean injection

Any **Java Beans** or **EJB** can be injected using **@javax.enterprise.inject.Inject** annotation.

```
public class MyService {

       @Inject
       private Data data;

}
```

← This is called *Injection point*

CDI is using the type to determine which object to instanciate and inject.

CDI: Context and Dependency Injection

Bean injection

Injected field in a bean:

```
public class Order {

    @Inject
    private Product product;

    @Inject
    private User customer;

}
```

Bean initializer methods or constructor:

```
public class Order {

    private Product product;
    private User customer;

    @Inject
    public Order(Product product) {
        this.product = product;
    }

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

}
```

CDI: Context and Dependency Injection

Bean injection

Injected field visibility:

```
public class Order {

✓   @Inject
    private Product product;

✗   @Inject
    private final User customer1;

✗   @Inject
    private static User customer2;

✓   @Inject
    public int number;

}
```

CDI: Context and Dependency Injection

Bean injection

It is possible to have a default constructeur with a bean constructor using injection.

```
public class Order { ✓

    @Inject
    public Order(Product product) {
        ...
    }

    public Order() {
        ...
    }

}
```

It is not possible to have more than one bean constructor using injection.

```
public class Order { ✗

    @Inject
    public Order(Product product) {
        ...
    }

    @Inject
    public Order(Product product, User customer) {
        ...
    }

}
```

CDI: Context and Dependency Injection

Bean injection

All injections in a class are done in a certain order:

```
public class Order {

    @Inject
    private Product product;

    @Inject
    public Order(Product product) {
        ...
    }

    @Inject
    public void setCustomer(User customer) {
        ...
    }

}
```

2 **Initializer fields**
Order of déclaration in the class

1 **Constructor**
Priority on the @Inject

3 Initializer methods
Order not guaranteed

CDI: Context and Dependency Injection

Bean injection

The container considers bean type and qualifiers when resolving a bean to be injected to an **injection point**.

An ***unsatisfied dependency*** exists at an injection point when no bean is eligible for injection to the injection point.

An ***ambiguous dependency*** exists at an injection point when multiple beans are eligible for injection to the injection point.

Both related exceptions are thrown at the startup of the container.

CDI: Context and Dependency Injection

Bean injection

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

- the bean type or qualifiers vary dynamically at runtime
- there may be no bean which satisfies the type and qualifiers, or
- we would like to iterate over all beans of a certain type.

In these situations, a **programmatic lookup** can be performed to inject an instance of the **javax.enterprise.inject.Instance** interface.

```
@Inject
Instance<PaymentProcessor> paymentProcessor;
...
if (paymentProcessor.isResolvable()) {
    PaymentProcessor pp = paymentProcessor.get();
}
```

Qualifiers

CDI: Context and Dependency Injection

Qualifiers

For a given bean type, there may be multiple beans which implement the type.

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

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

A qualifier type represents some client-visible semantic associated with a type that is satisfied by some implementations of the type (and not by others).

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

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

CDI: Context and Dependency Injection

Qualifiers

Qualifier types may be applied to injected fields to determine the bean that is injected

```
@Inject
@Synchronous
PaymentProcessor paymentProcessor;

@Inject
@Asynchronous
PaymentProcessor paymentProcessor;
```

An injection point may even specify multiple qualifiers.

CDI: Context and Dependency Injection

Qualifiers

A qualifier type may be declared by specifying the **@javax.inject.Qualifier** meta-annotation.

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

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

CDI: Context and Dependency Injection

Qualifiers

The qualifiers of a bean are declared by annotating the bean class or producer method or field with the qualifier types. Qualifier types may be applied to injected fields to determine the bean that is injected.

Any bean may declare multiple qualifier types.

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

A bean may only be injected to an injection point if it has all the qualifiers of the injection point.

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

CDI: Context and Dependency Injection

Qualifiers

Qualifiers can be applied when getting a contextual reference by programmatic lookup.

Any combination of qualifiers may be specified at the injection point:

```
@Inject @Synchronous Instance<PaymentProcessor> syncPaymentProcessor;

@Inject @Asynchronous Instance<PaymentProcessor> asyncPaymentProcessor;
```

@Any qualifier may be used, allowing the application to specify qualifiers dynamically:

```
@Inject @Any Instance<PaymentProcessor> anyPaymentProcessor;
...
Annotation qualifier = synchronously ?
       new SynchronousQualifier() : new AsynchronousQualifier();
PaymentProcessor pp = anyPaymentProcessor.select(qualifier).get();
```

CDI: Context and Dependency Injection

Qualifiers

CDI is providing some built-in qualifiers:

- **@javax.enterprise.inject.Any**
- **@javax.enterprise.inject.Default**
- **@javax.inject.Named**

Every bean has the built-in qualifier @Any, even if it does not explicitly declare this qualifier.

If a bean does not explicitly declare a qualifier other than @Named or @Any, the bean has exactly one additional qualifier, of type @Default. This is called the default qualifier. It is also assumed for any injection point that does not explicitly declare a qualifier

CDI: Context and Dependency Injection

Qualifiers

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

=

```
public class Order { ... }
```

```
@Any
@Default
public class Order { ... }
```

```
@Named("ord")
public class Order { ... }
```

```
@Any
@Default
@Named("ord")
public class Order { ... }
```

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

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

CDI: Context and Dependency Injection

Qualifiers

The use of **@Named** as an injection point qualifier is not recommended, except in the case of integration with legacy code that uses string-based names to identify beans.

The default name for a managed bean is the **unqualified class name** of the bean class, after converting the first character to lower case.
Example: bean class name = *ProductList* → default bean name = *productList*

If an injected field declares a @Named annotation that does not specify the value member, the name of the field is assumed.
For example, the following field has the qualifier @Named("paymentService"):

```
@Inject @Named PaymentService paymentService;
```

If any other injection point declares a @Named annotation that does not specify the value member, the container automatically detects the problem and treats it as a definition error.

Producers

CDI: Context and Dependency Injection

Producers

A *producer method* acts as a source of objects to be injected, where:

- the objects to be injected are not required to be instances of beans, or
- the concrete type of the objects to be injected may vary at runtime, or
- the objects require some custom initialization that is not performed by the bean constructor.

CDI: Context and Dependency Injection

Producers

A producer method may be declared by annotating a method with the **@javax.enterprise.inject.Produces** annotation.

```
public class Shop {

   @Produces
   PaymentProcessor getPaymentProcessor() { ... }

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

A producer method may also specify scope, bean name, stereotypes and/or qualifiers.

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

CDI: Context and Dependency Injection

Producers

A producer method may have any number of parameters. All producer method parameters are injection points.

```
public class OrderFactory {

   @Produces
   @ConversationScoped
   public Order createCurrentOrder(Shop shop, @Selected Product pdt) {
       Order order = new Order(pdt, shop);
       return order;
   }

}
```

CDI: Context and Dependency Injection

Producers

A *producer field* is a slightly simpler alternative to a producer method.
It may be declared by annotating a field with the @javax.enterprise.inject.Produces annotation.

```
public class Shop {
   @Produces PaymentProcessor paymentProcessor = ....;
   @Produces List<Product> products = ....;
}
```

A producer field may also specify scope, bean name, stereotypes and/or qualifiers.

```
public class Shop {
   @Produces @ApplicationScoped @Catalog @Named("catalog")
   List<Product> products = ....;
}
```

CDI: Context and Dependency Injection

Producers

Producer field visibility:

✓
```
@Produces @number
int value = 0;
```

✓
```
@Produces @number
public int value = 0;
```

✓
```
@Produces @number
protected int value = 0;
```

✓
```
@Produces @number
private int value = 0;
```

✓
```
@Produces @number
static int value = 0;
```

Producer method visibility:

✓
```
@Produces @number
int getInt()  { ... }
```

✓
```
@Produces @number
public int getInt()  { ... }
```

✓
```
@Produces @number
protected int getInt()  { ... }
```

✓
```
@Produces @number
private int getInt()  { ... }
```

✓
```
@Produces @number
static int getInt();
```

✗
```
@Produces @number
abstract int getInt();
```

CDI: Context and Dependency Injection

Producers

A producer field may also specify scope, bean name, stereotypes and/or qualifiers.

```
public class Shop {

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

}
```

Disposers

CDI: Context and Dependency Injection

Disposers

A disposer method allows the application to perform customized cleanup of an object returned by a producer method or producer field.
It is called when the container destroy any instance returned by that producer method or producer field.

CDI: Context and Dependency Injection

Disposers

A disposer method may be declared by annotating only one parameter @javax.enterprise.inject.Disposes. That parameter is the disposed parameter. Qualifiers may be declared by annotating the disposed parameter:

```
public class UserDatabaseEntityManager {

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

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

A bean may declare multiple disposer methods.

CDI: Context and Dependency Injection

Disposers

Disposer visibility:

✓
```
void dispose(@Disposes Service s) { ... }
```

✓
```
public void dispose(@Disposes Service s) { ... }
```

✓
```
protected void dispose(@Disposes Service s) { ... }
```

✓
```
private void dispose(@Disposes Service s) { ... }
```

✓
```
static void dispose(@Disposes Service s) { ... }
```

✗
```
abstract void dispose(@Disposes Service s);
```

Scopes

CDI: Context and Dependency Injection

Scopes

Scoped objects, exist in a well-defined lifecycle context:

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

All beans have a scope. The scope of a bean determines the lifecycle of its instances, and which instances of the bean are visible to instances of other beans.

CDI: Context and Dependency Injection

Scopes

A scope type is represented by an annotation type:

Scope	Annotation (javax.enterprise.context)	Duration
Dependent **(default)**	**@Dependent**	**The default scope if none is specified**; it means that an object exists to serve exactly one client (bean) and has the same lifecycle as that client (bean). When the container destroys an instance of a bean, the container destroys all dependent objects
Request	**@RequestScoped**	A user's interaction with a web application in a single HTTP request.
Session	**@SessionScoped**	A user's interaction with a web application across multiple HTTP requests.
Application	**@ApplicationScoped**	Shared state across all users' interactions with a web application.
Conversation	**@ConversationScoped**	A user's interaction with a JavaServer Faces application, within explicit developer-controlled boundaries that extend the scope across multiple invocations of the JavaServer Faces lifecycle. All long-running conversations are scoped to a particular HTTP servlet session and may not cross session boundaries.

CDI: Context and Dependency Injection

Scopes

When no scope is explicitly declared by annotating the bean class or producer method or field the scope of a bean is **defaulted**. The **default scope** for a bean which does not explicitly declare a scope is **@Dependent**.

Note:

- For all "normal scopes" (@ApplicationScoped, @SessionScoped, @ConversationScoped or @RequestScoped), CDI inject a generated client proxy class instead of a direct reference on the injected class. A such proxy allows CDI to access metada and perform transparent actions, for example serialization on @SessionScope to store data in HTTP session.
- For all "pseudo scopes“ (@Dependent, @Singleton), CDI inject a reference to the injected source class, no proxy are used.

CDI: Context and Dependency Injection

Scopes

The scope of a bean is defined by annotating the bean class or producer method or field with a scope type.

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


@ConversationScoped
public class Order { ... }


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

Alternatives

CDI: Context and Dependency Injection

Alternatives

An *alternative* is a bean that must be explicitly selected if it should be available for lookup, injection or name resolution.

An alternative may be declared by annotating the bean class or producer method or field with the @Alternative annotation.

```
@Alternative
public class MockOrder extends Order { ... }
```

Alternatively, an alternative may be declared by annotating a bean, producer method or producer field with a stereotype that declares an @Alternative annotation.

An alternative is not available for injection, lookup or name resolution to classes in a module unless the module is a bean archive and the alternative is explicitly *selected:*

- for the application using **@javax.annotation.Priority** annotation.
- for the bean archive using the file **beans.xml**.

CDI: Context and Dependency Injection

Alternatives

Declaring selected alternatives for an application:

An alternative may be given a priority for the application:

- by placing the @Priority annotation on the bean class of a managed bean
- by placing the @Priority annotation on the bean class that declares the producer method, field or resource.

```
@Alternative
@Priority(javax.interceptor.Interceptor.Priority.APPLICATION + 100)
public class MockOrder extends Order { ... }
```

CDI: Context and Dependency Injection

Alternatives

Declaring selected alternatives for a bean archive:

An alternative may be explicitly declared using the `<alternatives>` element of the beans.xml file of the bean archive. The `<alternatives>` element contains a list of bean classes and stereotypes.

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
        http://xmlns.jcp.org/xml/ns/javaee/beans_2_0.xsd"
       bean-discovery-mode="all" version="2.0">
   <alternatives>
      <class>com.acme.myfwk.InMemoryDatabase</class>
      <stereotype>com.acme.myfwk.Mock</stereotype>
      <stereotype>com.acme.site.Australian</stereotype>
   </alternatives>
</beans>
```

CDI: Context and Dependency Injection

Alternatives

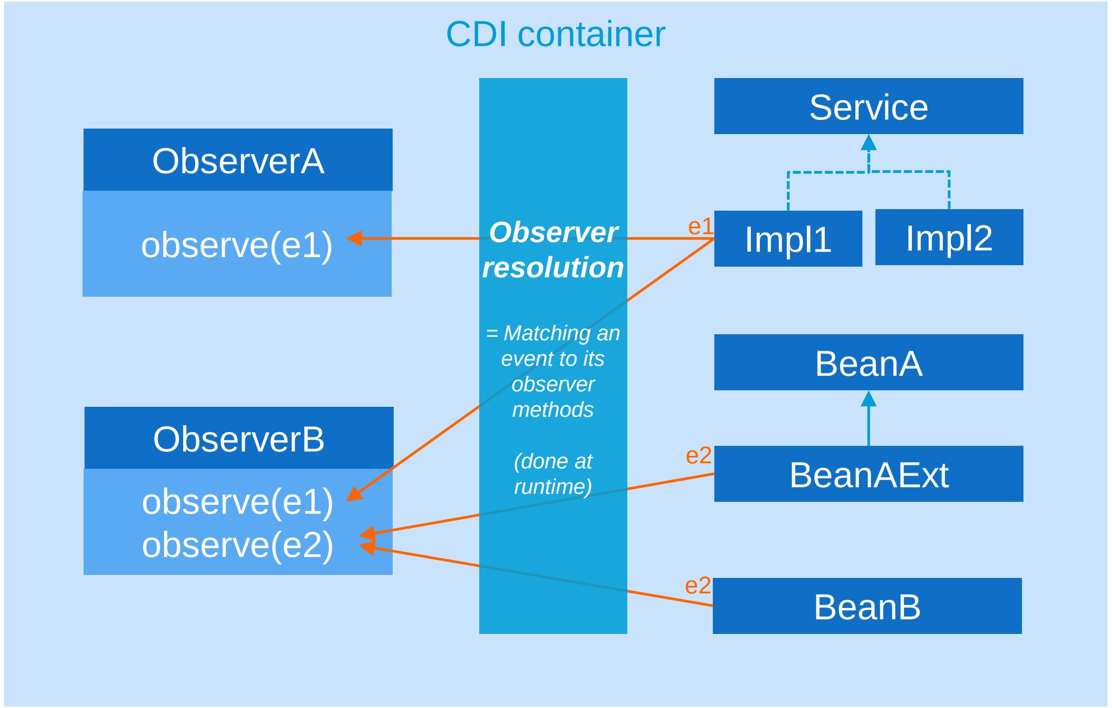
If two beans both support a certain bean type, and share at least one qualifier, then they are both eligible for injection to any injection point with that declared type and qualifier.

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

@Alternative
public class MockAsynchronousService extends AsynchronousService { ... }


@Inject Service service;
// ^^ AsynchronousService because @Asynchronous is missing

@Inject @Asynchronous Service service;
// ^^ MockAsynchronousService because @Asynchronous is present
```

CDI events

CDI: Context and Dependency Injection

CDI events



CDI: Context and Dependency Injection

CDI events

Beans may produce and consume events → allows beans to interact in a completely decoupled fashion, with no compile-time dependency between the interacting beans. Most importantly, it allows stateful beans in one architectural tier of the application to synchronize their internal state with state changes that occur in a different tier.

An event comprises:

- A Java object - the *event object*
- A set of instances of qualifier types - the *event qualifiers*

An *observer method* acts as event consumer, observing events of a specific type - the *observed event type* - with a specific set of qualifiers - the *observed event qualifiers*. An observer method will be notified of an event if the event object is assignable to the observed event type, and if the set of observed event qualifiers is a subset of all the event qualifiers of the event.

CDI: Context and Dependency Injection

CDI events

Beans fire events via an instance of the **@javax.enterprise.event.Event** interface, which may be injected:

```
@Inject Event<LoggedInEvent> loggedInEvent;

@Inject @Admin Event<LoggedInEvent> adminLoggedInEvent;
```

Or the application may specify qualifiers dynamically:

```
@Inject Event<LoggedInEvent> loggedInEvent;
...
LoggedInEvent event = new LoggedInEvent(user);
if (user.isAdmin()) {
    loggedInEvent.select(new AdminQualifier()).fire(event);
} else {
    loggedInEvent.fire(event);
}
```

CDI: Context and Dependency Injection

CDI events

Firing an event synchronously:

→ Method **fire()** which accepts an event object and blocks the calling thread until it completes.

```
@Inject Event<LoggedInEvent> loggedInEvent;

public void login() {
    ...
    loggedInEvent.fire(new LoggedInEvent(user));
}
```

All the resolved **synchronous observers** are called **in the thread in which fire() was called**.

Firing an event asynchronously:

→ Method **fireAsync()** which accepts an event object and returns immediately.

```
@Inject Event<LoggedInEvent> loggedInEvent;

public void login() {
    ...
    loggedInEvent.fireAsync(new LoggedInEvent(user));
}
```

All the resolved **asynchronous observers** are called in **one or more different threads**.

CDI: Context and Dependency Injection

CDI events

Observer method visibility:

✓
```
void afterLogin(@Observes Event event) { ... }
```

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

✓
```
private void afterLogin(@Observes Event event) { ... }
```

✓
```
protected void afterLogin(@Observes Event event) { ... }
```

✓
```
static void afterLogin(@Observes Event event) { ... }
```

✗
```
abstract void afterLogin(@Observes Event event) { ... }
```

CDI: Context and Dependency Injection

CDI events

Observer method of a synchronous event:

→ An observer method may be declared by annotating a parameter @**javax.enterprise.event.Observes**

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

Observer method of an asynchronous event:

→ An observer method may be declared by annotating a parameter @**javax.enterprise.event.ObservesAsync**

```
public void asyncAfterLogin(@ObservesAsync
LoggedInEvent event) { ... }
```

CDI: Context and Dependency Injection

CDI events

Observers with **@Default** qualifier will only be notified for events having either no qualifiers or only @Default qualifier:

```
@Inject Event<Document> documentEvent;
@Inject @Default Event<Document> documentDefaultEvent;


documentEvent.fire(new Document());
documentDefaultEvent.fire(new Document());


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

CDI: Context and Dependency Injection

CDI events

- There may be arbitrarily many observer methods with the same event parameter type and qualifiers.
- A bean (or extension) may declare multiple observer methods.
- Modifications made to an event in an observer method are propagated to following observers. The container is not required to guarantee a consistent state for an event parameter modified by asynchronous observers.

CDI: Context and Dependency Injection

CDI events

Observed event qualifiers may be declared by annotating the event parameter:

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

CDI: Context and Dependency Injection

CDI events

An event parameter may have multiple qualifiers and will notify observer methods having their set of observer qualifiers a subset of the fired event's qualifiers or an empty set:

```
@Inject Event<User> userEvent;
...
userEvent.select(
    new UpdatedQualifier(),
    new ByAdminQualifier(),
    new DisabledQualifier()
).fire(user);
```

✓
```
public void afterUserUpdatedByAdmin(@Observes @Updated
@ByAdmin User user) { ... }
```

✓
```
public void afterUserUpdated(@Observes @Updated User
user) { ... }
```

✓
```
public void afterUserEvent(@Observes User user) { ... }
```

✗
```
public void afterUserEvent(@Observes @Default User
user) { ... }
```

CDI: Context and Dependency Injection

CDI events

```
@Inject Event<LoggedInEvent> loggedInEvent;
...
public void login() {
    final User user = ...;
    loggedInEvent.select(new RoleQualifier() {
        public String value() {
            return user.getRole();
        }
    }).fire(new LoggedInEvent(user));
}
```

```
@Qualifier @Target(PARAMETER) @Retention(RUNTIME)
public @interface Role {
    String value();
}

public abstract class RoleQualifier
            extends AnnotationLiteral<Role>
            implements Role { }
```

Then the following observer method will always be notified of the event:

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

Whereas this observer method may (or not) be notified, depending upon the value of user.getRole():

```
public void afterAdminLogin(
        @Observes @Role("admin") LoggedInEvent e) {...}
```

As usual, the container uses equals() to compare event qualifier type member values.

CDI: Context and Dependency Injection

CDI events

Before the actual observer notification, the container determines an order in which the observer methods for a certain event are invoked.
The priority of an observer method may be declared by annotating the event parameter with @**Priority** annotation.
Observers with smaller priority values are called first.

```
void afterLogin(@Observes @Priority(Priority.APPLICATION) LoggedInEvent e) {
    ...
}
```

Notes:

- If no @Priority is specified, the default priority javax.interceptor.Interceptor.Priority.APPLICATION + 500 is assumed.
- The order of more than one observer with the same priority is undefined and the observer methods are notified therefore in a non predictable order.

CDI: Context and Dependency Injection

CDI events

3 events are synchronously fired during the application lifecycle with a specific qualifier for each step.

```
public void init(@Observes @Initialized(ApplicationScoped.class) Object obj) {
    // Actions when The application context is initialized
}


public void stop(@Observes @BeforeDestroyed(ApplicationScoped.class) Object obj) {
    // Actions when the application context is about to be destroyed
}


public void destroy(@Observes @Destroyed(ApplicationScoped.class) Object obj) {
    // actions when the application context is destroyed
}
```

CDI: Context and Dependency Injection

CDI events

A **conditional observer method** is an observer method which is notified of an event only if an instance of the bean that defines the observer method already exists in the current context. It can be declared by specifying "**notifyObserver=IF_EXISTS**".

```
public void refreshOnDocumentUpdate(
        @Observes(notifyObserver=IF_EXISTS) @Updated Document doc) { ... }


public void asyncRefreshOnDocumentUpdate(
        @ObservesAsync(notifyObserver=IF_EXISTS) @Updated Document doc) { ... }
```

Beans with scope @Dependent may not have conditional observer methods.

CDI: Context and Dependency Injection

CDI events

Transactional observer methods receive event notifications during different phases of the transaction in which the event was fired. If no transaction is in progress when the event is fired, they are notified at the same time as other observers.

A transactional observer method may be declared by specifying any value from enumeration javax.enterprise.event.TransactionPhase other than IN_PROGRESS.

✓
```
void onDocUpdate(@Observes(during=AFTER_SUCCESS) @Updated Document doc) { ... }
```

✗
```
void onDocUpdate(@ObservesAsync(during=AFTER_SUCCESS) @Updated Document doc) {...}
```

TransactionPhase	Description
BEFORE_COMPLETION	Called during the before completion phase of the transaction.
AFTER_COMPLETION	Called during the after-completion phase of the transaction.
AFTER_FAILURE	Called during the after-completion phase of the transaction, only when the transaction fails (rollback).
AFTER_SUCCESS	Called during the after-completion phase of the transaction, only when the transaction completes successfully.

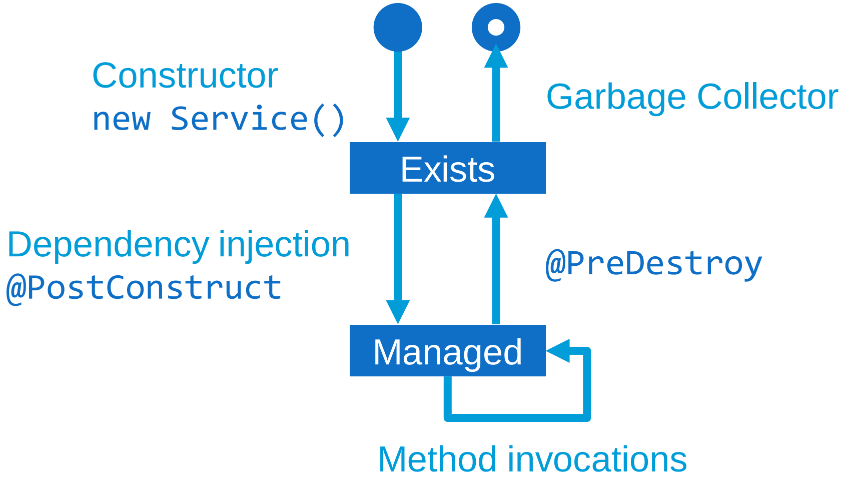
Lifecycle callback methods

CDI: Context and Dependency Injection

Lifecycle callback methods

Methods in a bean class may be declared as a lifecycle callback method by annotating the method with the following annotations:

- **@javax.annotation.PostConstruct** → method is invoked by the container on newly constructed bean instances after all dependency injection has completed and before the first business method is invoked on the enterprise bean.
- **@javax.annotation.PreDestroy** → method is invoked before the container destroys an instance of a bean.



CDI: Context and Dependency Injection

Lifecycle callback methods

```
public class Resources {

    private EntityManagerFactory emf;

    @PostConstruct
    public void setupEntityManagerFactory() {
        emf = Persistence.createEntityManagerFactory("...");
    }

    @PreDestroy
    public void setupEntityManagerFactory() {
        emf.close();
    }

}
```

Interceptors

CDI: Context and Dependency Injection

Interceptors

Interceptors (JSR 318) are used to allow developers to invoke interceptor methods on an associated target class. Common uses of interceptors are logging, auditing, and profiling.

An interceptor can be defined within a target class as an interceptor method, or in an associated class called an interceptor class. Interceptor classes contain methods that are invoked in conjunction with the methods or lifecycle events of the target class.

Interceptor classes and methods are defined using metadata annotations, or in the deployment descriptor of the application containing the interceptors and target classes.

Interceptor Metadata Annotation	Description
@javax.interceptor.AroundInvoke	Designates the method as an interceptor method.
@javax.interceptor.AroundTimeout	Designates the method as a timeout interceptor, for interposing on timeout methods for enterprise bean timers.
@javax.annotation.PostConstruct	Designates the method as an interceptor method for post-construct lifecycle events.
@javax.annotation.PreDestroy	Designates the method as an interceptor method for pre-destroy lifecycle events.

CDI: Context and Dependency Injection

Interceptors

A business method interceptor applies to invocations of methods of the bean by clients of the bean:

```
@Interceptor
public class TransactionInterceptor {

   @AroundInvoke
   public Object manageTransaction(InvocationContext ctx) throws Exception {
      ...
      Object o = ctx.proceed();
      ...
      return o;
   }

}
```

CDI: Context and Dependency Injection

Interceptors

The @Interceptors annotation defined by the interceptor specification (and used by the managed bean and EJB specifications) is still supported in CDI.

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

   public void checkout() { ... }

}
```

However, this approach suffers the following drawbacks:

- the interceptor implementation is hardcoded in business code,
- interceptors may not be easily disabled at deployment time, and
- the interceptor ordering is non-global — it is determined by the order in which interceptors are listed at the class level.

Therefore, it is recommended the use of CDI-style interceptor bindings.

CDI: Context and Dependency Injection

Interceptors

CDI enhances the interceptor functionality with a more sophisticated, semantic, annotation-based approach to binding interceptors to beans.
The interceptor binding type allows to specify exactly which beans we're interested in.

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

@Transactional
public class ShoppingCart { ... }

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

CDI: Context and Dependency Injection

Interceptors

The interceptor is a standard interceptor annotated with @Interceptor and @Transactional.

```
@Transactional @Interceptor
public class TransactionInterceptor {

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

}
```

Multiple interceptors may use the same interceptor binding type.

Note: Interceptors can take advantage of dependency injection.

CDI: Context and Dependency Injection

Interceptors

By default, all interceptors are disabled. The interceptor should be enabled in the beans.xml descriptor of a bean archive. This activation only applies to the beans in that archive. The order of declaration is the order of execution.

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans_2_0.xsd"
  bean-discovery-mode="all"
  version="2.0"

    <interceptors>
       <class>org.mycompany.myapp.SecurityInterceptor</class>
       <class>org.mycompany.myapp.TransactionInterceptor</class>
    </interceptors>
</beans>
```

Unit tests with CDI

CDI: Context and Dependency Injection

How to use JPA with CDI ?

- Add jakarta.persistence-api (JPA) dependency as provided to your classpath
- Then we need to produce EntityManager thanks to @PersistenceContext annotation with the persistence unit name defined in persistence.xml file:

```
@Produces
@PersistenceContext(unitName="my-pu")
@MyDatabase
private EntityManager em;
```

- You can also see the `@MyDatabase` annotation that is a qualifier. It will be usefull at injection point to know which EntityManager we want to inject when we have several persistence units:

```
@Inject
@MyDatabase
private EntityManager em;
```

CDI: Context and Dependency Injection

How to use JTA with CDI ?

- Add jboss-transaction-api_1.3_spec (JTA) dependency as provided to your classpath
- Then you can use @javax.transaction.Transactional on classes/methods
- This annotation takes a TxType parameter:
  - `REQUIRED (default)`
  - `REQUIRES_NEW`
  - `MANDATORY`
  - `SUPPORTS`
  - `NOT_SUPPORTED`
  - `NEVER`

CDI: Context and Dependency Injection

Qualifier

The qualifier `@MyDatabase` is a simple Java annotation annotated with @Qualifier annotation:

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER})
public @interface MyDatabase {}
```

Qualifiers are useful to specify what we want to produce and then inject (when you have several implementation for instance)

CDI: Context and Dependency Injection

Testing with Weld JUnit5

- Even if it’s possible to test CDI beans with JUnit4, we first recommend to migrate to JUnit5
- Add org.jboss.weld:weld-junit5:test to your dependencies
- Weld Junit5 allows to test CDI beans:

```
@EnableWeld
public class MyTest {

    @WeldSetup
    public WeldInitiator weld = WeldInitiator.from(WeldInitiator.createWeld()
                    .beanClasses(MyBean.class)
                    .addExtensions(TransactionExtension.class))
            .build();
```

- beanClasses() takes the list of bean classes to be tested (possibility to add a Scanner class to get all classes of a package)
- `TransactionExtension.class` is an extension to enable JTA (see JTA slide for details)

CDI: Context and Dependency Injection

Testing with Weld JUnit / JPA

- For JPA, we cannot rely on the default `EntityManager` producer as the `@PersistenceContext` is specific to JEE containers. So we need an alternative for unit testing:

```
@Singleton
@Alternative
@Priority(Interceptor.Priority.APPLICATION + 1)
public class EntityManagerProducerAlt {

        private static final EntityManagerFactory EMF = Persistence.createEntityManagerFactory("my-pu");

        @Produces
        @MyDatabase
        public EntityManager getEntityManger() {
                return EMF.createEntityManager();
        }

        public void disposeEntityManager(@Disposes @MyDatabase EntityManager em) {
                if(em.isOpen()) {
                        em.close();
                }
        }

}
```

CDI: Context and Dependency Injection

Testing with Weld JUnit / JTA

- Add narayana-jta (JTA implementation of JBoss EAP) and wildfly-naming (to enable JNDI) dependencies to test classpath
- Add jndi.properties file to src/test/resources with this line: java.naming.factory.initial=org.jboss.as.naming.InitialContextFactory
- Add the extension `TransactionExtension.class` to Weld JUnit context (see previous example)
- Bind JTA implementation by calling: `JNDIManager.bindJTAImplementation();`
- Bind datasource to JNDI:

```
final InitialContext initialContext = new InitialContext();
final JdbcDataSource dataSource = new JdbcDataSource();
dataSource.setURL("jdbc:h2:mem:test;MODE=MYSQL;DB_CLOSE_DELAY=-1");
dataSource.setUser("sa");
dataSource.setPassword("");
initialContext.bind("jdbc/myDS", dataSource);
```

- JNDI name must match the one defined in persistence.xml file:

```
<persistence-unit name="my-pu" transaction-type="JTA">
         <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
         <jta-data-source>jdbc/myDS</jta-data-source>
```

CDI: Context and Dependency Injection

Testing with Weld Standalone

- It is also possible to test CDI using weld as a standalone application (could be used for integration testing for instance):

```
final Weld weld = new Weld();
final WeldContainer weldContainer = weld.initialize();
...
weldContainer.shutdown();
```

CDI: Context and Dependency Injection

Testing with JBatch

- To unit test a specific step of a batch we can rely on Weld JUnit
- For integration tests (to test the whole batch), we can rely on JBeret implementation (used in Jboss) that will start the Weld container.
- You’ll need to have jberet-se dependency in your test classpath.
- Then you can execute your batch:

```
final JobOperator jobOperator = BatchRuntime.getJobOperator();
final long jobExecutionId = jobOperator.start("myBatch.xml", new Properties());
final JobExecutionImpl jobExecution = (JobExecutionImpl) jobOperator.getJobExecution(jobExecutionId);
jobExecution.awaitTermination(120, TimeUnit.SECONDS);
```

Annexes

Annexes

Inject an EJB inside a CDI bean

It is possible to use both EJB and CDI in a same application. Then an EJB can be injected inside a bean managed by CDI and vice versa.

```
public class PaymentService {

    @EJB
    private CardService cardService;

}
```

```
@Stateless
public class PaymentService {

    @Inject
    private CardService cardService;

}
```

Annexes

Inject an EntityManager or a Datasource in a CDI bean

```
@Singleton
public class EntityManagerProducer {

    @Produces
    @PersistenceContext(unitName="my-database")
    @MyDatabase
    private EntityManager em;

    @Produces
    @Resource(name = "jdbc/my-database")
    @MyDatabase
    private DataSource dataSource;

}
```

```
@Singleton
public class EntityManagerProducer {

    private static final EntityManagerFactory EMF =
        Persistence.createEntityManagerFactory(“my-database");

    @Produces @MyDatabase
    public EntityManager getEntityManager() {
        return EMF.createEntityManager();
    }

    @Produces @MyDatabase
    public DataSource getDataSource() throws NamingException {
        final InitialContext context = new InitialContext();
        return (DataSource) context.lookup("jdbc/my-database");
    }

    public void disposeEntityManager(@Disposes @MyDatabase EntityManager em) {
        if(em.isOpen()) { em.close(); }
    }
}
```

Introduction

EJV vs CDI

JAKARTA™ EE

EJB

- Designed to be processed by a JEE application server
- Long to deploy
- Based on a pool of instances managed by the app server
- No ability to manage the instantiation of an EJB
- Hard to override the behavior at runtime
- Hard to mock and test in unit test context
- Not embedded in serverless solutions (Microprofile, Quarkus)
- Only classes marked EJB are managed
- JNDI integration by default
- Integration of the JTA management by default
- Concurrency management on Singleton by default
- Asynchronous call integration by default

CDI

- Usable in JEE or JSE application
- Quick to deploy
- Powerfull DI tool: everything is managed and can be injected
- Ability to
  - manage instantiation/injection of a bean at runtime
  - perform actions during some steps in the lifecycle of the bean
  - react an event
- Easily to mock and test in unit test context
- No need JNDI anymore with CDI
- Need to manage manually the JTA transactions
- Embedded in recent serverless solutions (Microprofile, Quarkus)

Annexes

Asynchronous call on a CDI bean

EJB provides a way to execute a method asynchronously byt using **@Asynchronous** on a method wich returns a **Future**

```
@Asynchronous
public Future<Object> asynchronousCall() {
    Object response;
    ...
    return new AsyncResult<Object>(response);
}
```

CDI doesn’t provide such feature by default, but it is still possible to perform asynchronous call using the **ExecutorService**

```
public Future<Object> asynchronousCall() {
    return executorService.submit(() -> {
        Object response;
        ...
        return response;
    });
}
```

In a JEE application, the ExecutorService can be injected as a "ManagedExecutorService" using @Resource

```
@Resource
private ManagedExecutorService executorService;
```

Annexes

Documentation and useful links

JSR 365: Contexts and Dependency Injection for Java 2.0	https://docs.jboss.org/cdi/spec/2.0/cdi-spec.html
CDI introduction	• https://www.baeldung.com/java-ee-cdi • https://quarkus.io/guides/cdi
Interceptors	https://docs.jboss.org/weld/reference/1.0.0/en-US/html/interceptors.html
Contexts and Dependency Injection for Java EE tutorial	https://docs.oracle.com/javaee/7/tutorial/partcdi.htm#GJBNR
CDI event notification	https://www.baeldung.com/cdi-event-notification
Weld - CDI Reference Implementation	https://docs.jboss.org/weld/reference/5.1.0.Final/en-US/html_single