# LEARNING cdi

#cdi

# Table of Contents

About 1
Chapter 1: Getting started with cdi 2
Remarks 2
Examples 2
Installation or Setup 2
Quick setup in a Java SE environment 2
Step 1. Add dependencies to your POM. 2
Step 2. Add beans.xml 2
Step 3. Initialize CDI 3
Implementations 3
Chapter 2: Dependency Injection 4
Introduction 4
Examples 4
Constructor Injection 4
Field Injection 5
Chapter 3: Scopes 7
Remarks 7
Examples 7
The default scope 7
@RequestScoped 8
@ApplicationScoped 8
@SessionScoped 9
Credits 11

# About

# Chapter 1: Getting started with cdi

## Remarks

This section provides an overview of what cdi is, and why a developer might want to use it.

It should also mention any large subjects within cdi, and link out to the related topics. Since the Documentation for cdi is new, you may need to create initial versions of those related topics.

## Examples

### Installation or Setup

Detailed instructions on getting cdi set up or installed.

### Quick setup in a Java SE environment

If you are working with a Java EE 6+ application server, CDI is part of the container and you do not need to do anything to start using it. But CDI is not limited to Java EE application servers. It can be used in Java SE applications or simple servlet containers just as easily. Let's take a look at using CDI in a simple command-line application.

## Step 1. Add dependencies to your POM.

```
<dependency>
    <groupId>org.jboss.weld.se</groupId>
    <artifactId>weld-se-core</artifactId>
    <version>3.0.0.Alpha15</version>
</dependency>
```

## Step 2. Add beans.xml

CDI requires an empty beans.xml file so it can scan the JAR for classes. So create

```
src/main/resources/META-INF/beans.xml
```

with the following

```
<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_1_1.xsd"
       bean-discovery-mode="all">
    <scan>
        <exclude name="org.jboss.weld.**" />
    </scan>
```

```
</beans>
```

## Step 3. Initialize CDI

In this example, the main(String []) method initializes CDI and then CDI is used to get an instance of the class itself to start running the SE application.

```
import java.util.Arrays;
import java.util.List;
import javax.enterprise.inject.spi.CDI;
import javax.inject.Inject;

public class Main {
    public static void main(String[] args) {
        CDI<Object> cdi = CDI.getCDIProvider().initialize();
        Main main = cdi.select(Main.class).get();
        main.main(Arrays.asList(args));
    }

    @Inject
    protected MyService myService;

    protected void main(List<String> args) {
        System.out.println("Application starting");

        // MyService object injected by CDI
        myService.go();
    }
}
```

That's it, really simple.

## Implementations

CDI is a Java EE specification. It specifies how things should be done, and which features must be provided, but it isn't actually a specific library or set of code. In order to use CDI, you will need to use a CDI *implementation*.

The reference implementation of the CDI spec is a set of libraries known as Weld. An alternative implementation of the CDI spec exists as Apache OpenWebBeans. Either of these implementations will be able to give you the features of CDI. If you are not using a Java EE application server that ships with one of these implementations, it will be up to you to select and install one of these implementations into your application or runtime.

Read Getting started with cdi online: https://riptutorial.com/cdi/topic/5624/getting-started-with-cdi

# Chapter 2: Dependency Injection

## Introduction

CDI's flagship feature is a declarative API for dependency injection. Classes can have dependencies flagged with the `@Inject` annotation, which will indicate to the CDI manager that it needs to provide those dependencies when constructing an instance of the class.

## Examples

### Constructor Injection

The common case for injecting dependencies into a class is with constructor injection. This involves annotating a constructor on the class with @Inject. The CDI manager will look for a constructor with the @Inject annotation when creating an instance of the class. When it finds an @Inject-annotated constructor, it will use reflection to find which parameters are required by the constructor, construct or obtain instances of those dependencies, then call the constructor with those dependencies.

```
public class Spaceship {

    private final PropulsionSystem propulsionSystem;
    private final NavigationSystem navigationSystem;

    @Inject
    public Spaceship(PropulsionSystem propulsionSystem, NavigationSystem navigationSystem) {
        this.propulsionSystem = propulsionSystem;
        this.navigationSystem = navigationSystem;
    }

    public void launch() throws FlightUnavailableException {
        if (propulsionSystem.hasFuel()) {
            propulsionSystem.engageThrust();
        } else {
            throw new FlightUnavailableException("Launch requirements not met. Ship needs
fuel.");
        }
    }

}
```

Any time a Spaceship instance defined in this manner is created by CDI, it will receive a PropulsionSystem and a NavigationSystem as arguments to its constructor. Because these dependencies are added via the constructor, we have an easy way to provide alternate dependencies in our test harness when unit testing:

```
public class SpaceshipTest {

    private Spaceship systemUnderTest;
    private TestPropulsionSystem testPropulsionSystem;
    private TestNavigationSystem testNavigationSystem;

    @Before
    public void setup() {
        setupCollaborators();
        systemUnderTest = new Spaceship(testPropulsionSystem, testNavigationSystem);
    }

    @Test
    public void launchSequenceEngagesThrustIfFueled() {
        //given
        testPropulsionSystem.simulateHavingFuel();

        //when
        systemUnderTest.launch();

        //then
        testPropulsionSystem.ensureThrustWasEngaged();
    }

}
```

## Field Injection

The same example from above can also be done using what is known as field injection. Instead of annotating the constructor with `@Inject`, we annotate the fields we wish to have injected

```
public class Spaceship {

    @Inject
    private PropulsionSystem propulsionSystem;
    @Inject
    private NavigationSystem navigationSystem;

    public void launch() throws FlightUnavailableException {
        if (propulsionSystem.hasFuel()) {
            propulsionSystem.engageThrust();
        } else {
            throw new FlightUnavailableException("Launch requirements not met. Ship needs
fuel.");
        }
    }

}
```

Note that getter/setter methods are not required for field injection to work. Also note that the field does not need to be public and can in fact be private, though it cannot be final. Making the field private will, however, make writing tests for the code more difficult, as the tests will have to use reflection to modify the field if it does not have a setter.

Also note that constructor injection and field injection can be used in tandem if desired, though it should be very carefully evaluated whether or not it makes sense to do so on a case by case basis. Perhaps it is necessary when working with legacy code to add a dependency to the class without modifying the constructor signature for some peculiar reason.

```
public class Spaceship {

    private PropulsionSystem propulsionSystem;
    @Inject
    private NavigationSystem navigationSystem;

    @Inject
    public Spaceship(PropulsionSystem propulsionSystem) {
        this.propulsionSystem = propulsionSystem;
    }

    public void launch() throws FlightUnavailableException {
        if (propulsionSystem.hasFuel()) {
            propulsionSystem.engageThrust();
        } else {
            throw new FlightUnavailableException("Launch requirements not met. Ship needs 
fuel.");
        }
    }

}
```

Read Dependency Injection online: https://riptutorial.com/cdi/topic/9637/dependency-injection

# Chapter 3: Scopes

## Remarks

**Why do I need these no-args constructors???**

*What happens if a session scoped bean gets injected into an application scoped bean? How does the application scoped bean get the correct session scoped bean instance for each request? Wouldn't the session scoped bean leak out into other requests? How does that work?* In order to facilitate scoping, CDI uses what is known as a proxy. When CDI injects a non-dependent scoped bean into another object, it does not inject the bean directly. Instead, it subclasses that bean to create what is known as a proxy. Whenever a method is called on the proxy, it asks the CDI runtime to look up the correct bean for that particular scope (if it's request scoped, get the bean for that request. If it's session scoped, get the bean for that session. etc.), and then forwards the call to the real object, returning any result for non-void methods. This means that the following is an okay thing to do:

```
@ApplicationScoped
public class ApplicationScopedClass {

    private final RequestScopedClass requestScopedClass;

    @Inject
    public ApplicationScopedClass(RequestScopedClass requestScopedClass) {
        this.requestScopedClass = requestScopedClass;
    }

    public ApplicationScopedClass() {

    }

    public doSomething() {
        requestScopedClass.doSomethingRequestSpecific(); //This works, because of the proxy
    }

}
```

However, the in order for a class to legally subclass another class in Java, it must have a valid constructor. And a valid constructor must call either another constructor on the class, or the parent class' constructor (e.g. `super()`). In order to simplify this subclassing, the CDI spec requires any non-dependent scoped bean to provide a public no-args constructor, so that the runtime doesn't have to try to guess at what to do with a class which needs to be proxied.

## Examples

### The default scope

```
public class DependentScopedClass {
    //This class has no scoping annotations, so a new instance gets created at every injection
point.

    @Inject
    public DependentScopedClass(SomeDependency someDependency) {
        doSomethingWith(someDependency);
    }

}
```

The default scope for most CDI beans is referred to as dependent scope. A class that does not contain any scope annotation will be treated as dependent scope, **unless** it is a JAX-RS resource or provider (resources default to request scoped, and providers default to application scoped). An instance of a dependent scope class lives as long as the object it is injected into does. Any time a class gets constructed and has a dependent scope class as a dependency, the dependent scoped class gets created and injected *directly* into the object which needs it. The significance of this will become apparent after later examples.

## @RequestScoped

```
@RequestScoped
public class RequestScopedClass {
    //This class gets constructed once per Servlet request, and is shared among all CDI-
managed classes within that request.

    @Inject
    public RequestScopedClass(SomeDependency someDependency) {
        doSomethingWith(someDependency);
    }

    public RequestScopedClass() {
        //Note that it is required that a request scoped class have a public no-args
constructor
    }

}
```

If a bean is annotated with @RequestScoped, it will be created once for any request. If two objects depend on a request scoped class, they will both get references to the same object.

**Note:** Any request scoped bean must have a public no-args constructor. The reason for this will be explained later on.

## @ApplicationScoped

```
@ApplicationScoped
public class ApplicationScopedClass {
    //This class gets constructed once for the entire life of the application, and is shared
among all CDI-managed classes throughout the life of the application.

    @Inject
    public ApplicationScopedClass(SomeDependency someDependency) {
        doSomethingWith(someDependency);
    }
    public ApplicationScopedClass() {
        //Note that it is required that an application scoped class have a public no-args
constructor
    }

}
```

Classes with @ApplicationScoped are created only once, and each object which depends on the class share the same instance. These classes are 'effectively' singletons, however it should be noted that there is nothing to prevent a coder from manually creating additional instances of the class. Thus, using @ApplicationScoped is useful for sharing context across the entire application, or as a performance optimization (if the class is expensive to construct instances of), but it should not be relied upon as an integrity measure for guaranteeing only one instance of a class exists.

Like request scoped beans, application scoped beans need to have a public no-args constructor.

### @SessionScoped

```
@SessionScoped
public class SessionScopedClass implements Serializable {
    //This class gets constructed once per session, and is shared among all CDI-managed
classes within that session. Notice that it implements Serializable, since the instance will
get put on the session.

    @Inject
    public SessionScopedClass(SomeDependency someDependency) {
        doSomethingWith(someDependency);
    }

    public SessionScopedClass() {
        //Note that it is required that a session scoped class have a public no-args
constructor
    }

}
```

Classes annotated with @SessionScoped will be created once per session, and two objects within the same session will share the same instance of the session scoped class.

It is important to notice that the session scoped class should implement Serializable. This requirement exists because the bean will be stored in the servlet container's session for that particular session scoped instance. In general, any object being put into the session needs to be serializable for two reasons: First, sessions may be persisted to disk after a certain amount of inactivity to save memory. This is know as session passivation, and it requires serialization. The second reason is that in high-availability clusters, session replication is often used in order to allow any server in the cluster to service a request for a given session. This also generally requires serialization.

Much like request scoped and application scoped, session scoped classes must have a public no-args constructor.

Read Scopes online: https://riptutorial.com/cdi/topic/9641/scopes

# Credits

| S. No | Chapters | Contributors |
|---|---|---|
| 1 | Getting started with cdi | Community, Dogs, Michael Remijan |
| 2 | Dependency Injection | Dogs |
| 3 | Scopes | Dogs |