RFC 193 - CDI Integration

Draft

31 Pages

Abstract

While OSGi services are very powerful, some still find it challenging to use them effectively. This RFC looks at how CDI can be used to interact with the OSGi service layer. The intent is to bring the popular CDI programming model to OSGi as a way to interact with OSGi services. It will provide the convenience of CDI and allows developers familiar with the CDI technology to reuse their skills in an OSGi context.

0 Document Information

0.1 License

DISTRIBUTION AND FEEDBACK LICENSE, Version 2.0

worldwide, perpetual, irrevocable, royalty-free copyright license to copy, publish, license, modify, sublicense or otherwise distribute and exploit your Feedback for any purpose. Likewise, if incorporation of your Feedback would cause an implementation of the Distribution, including as it may be modified, amended, or published at any point in the future ("Future Specification"), to necessarily infringe a patent or patent application that you own or control, you hereby commit to grant to all implementers of such Distribution or Future Specification an irrevocable, worldwide, sublicenseable, royalty free license under such patent or patent application to make, have made, use, sell, offer for sale, import and export products or services that implement such Distribution or Future Specification. You warrant that (a) to the best of your knowledge you have the right to provide this Feedback, and if you are providing Feedback on behalf of a company, you have the rights to provide Feedback on behalf of your company; (b) the Feedback is not confidential to you and does not violate the copyright or trade secret interests of another; and (c) to the best of your knowledge, use of the Feedback would not cause an implementation of the Distribution or a Future Specification to necessarily infringe any third-party patent or patent application known to you. You also acknowledge that the OSGi Alliance is not required to incorporate your Feedback into any version of the Distribution or a Future Specification.

I HEREBY ACKNOWLEDGE AND AGREE TO THE TERMS AND CONDITIONS DELINEATED ABOVE.

0.2 Trademarks

OSGi™ is a trademark, registered trademark, or service mark of the OSGi Alliance in the US and other countries. Java is a trademark, registered trademark, or service mark of Oracle Corporation in the US and other countries. All other trademarks, registered trademarks, or service marks used in this document are the property of their respective owners and are hereby recognized.

0.3 Feedback

This document can be downloaded from the OSGi Alliance design repository at https://github.com/osgi/design The public can provide feedback about this document by opening a bug at https://www.osgi.org/bugzilla/.

0.4 Table of Contents

0 Document Information....2
0.1 License....2
0.2 Trademarks....3
0.3 Feedback....3
0.4 Table of Contents....3
0.5 Terminology and Document Conventions....4
0.6 Revision History....4

1 Introduction....6

2 Application Domain....6
2.1 CDI....7
2.1.1 Example....7
2.2 Weld-OSGi....8
2.2.1 Weld-OSGi example....8
2.3 Declarative Services, Blueprint and CDI....8
2.4 Terminology + Abbreviations....9

3 Problem Description....9

4 Requirements....9
4.1 Functional Requirements....9
4.2 Non-functional Requirements....11
4.3 Requirements from RFP 98 (OSGi/Java EE umbrella RFP)....12

5 Technical Solution....12
5.1 Entities....12
5.2 CDI Container Life-Cycle....13
5.2.1 Initialization of a CDI Container....13
5.3 Requirements and Capabilities....16
5.4 Managed Beans and OSGi Services....18
5.4.1 Publishing plain CDI providers in the Service Registry....19
5.5 Scopes for beans annotated as @Component@Service....19
5.6 Life-Cycle....20
5.6.1 Required dependencies (@Component@Service only)....21
5.6.2 Optional dependencies....22
5.6.3 Allowing Service Injections in vanilla CDI injection points....22
5.6.4 Service and bundle registration observers....23
5.6.5 Service Filters....23
5.7 BundleContext injection....25
5.8 BeanManager service....25
5.9 osgi.extender Capability....25
5.10 osgi.contract Capability....26
5.11 Events....26
5.11.1 CDI Event....26
5.11.2 Replay....27

6 Data Transfer Objects....27

7 Javadoc....27

8 Considered Alternatives....28
8.1 EventAdmin integration....28

9 Security Considerations....30

10 Document Support....30
10.1 References....30
10.2 Author’s Address....31
10.3 Acronyms and Abbreviations....31
10.4 End of Document....31

0.5 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 1.

`Source code is shown in this typeface.`

0.6 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial	16/10/12	Initial version of the draf. David Bosschaert <david@redhat.com>
2nd draft	26/10/12	First draft of chapter 5 Paul Bakker <paul.bakker@luminis.eu>
3rd draft	30/10/12	Life-cycle callbacks Paul Bakker <paul.bakker@luminis.eu>
4th draft	27/01/13	Details on service dynamics Paul Bakker <paul.bakker@luminis.eu>
5th draft	27/02/13	Comments after EEG call 2/20/2013
7th draft	09.04.13	New sections Entities, CDI container lifecycle, Requirements and Capabilities. A number of comments on existing sections. Harald Wellmann/ Rebaze <harald.wellmann@rebaze.com>
8th draft	October, 2013	David Bosschaert, some cleanup and additions: • Removed OSGi prefix from classnames • Replaced API described in the text with actual Javadoc generated from classes in OSGi git • Tentatively added sections about support for scopes and support for 'bare' @Inject injection points • Tentatively added section about support for declaring @Component via manifest headers • Added section on the CdiContainer service • Added section on osgi.capability portable Java contracts
9th draft	07/11/13	David Bosschaert, process input from call 6th November 2013.
10th draft	14/11/13	David Bosschaert, process input from the virtual F2F November 13, 2013.
11th draft	05/11/14	Emily Jiang, update to CDI 1.2
12th draft	20/11/14	Emily Jiang more update based on the comments from F2F in Los Angeles
13th draft	17/01/15	Emily Jiang. Add lifecycle and Event
14th draft	15/02/15	Emily Jiang, update based on the call 11th March 2015
15th draft	09/09/15	Emily Jiang, updated based on the call in June F2F
16th draft	18/12/15	Emily Jiang, updated based on F2F in November

1 Introduction

While OSGi services are very powerful, consuming them has been a challenge for many OSGi users. There have been a number of solutions to this problem both in OSGi specifications as well as in non-standardized technologies. OSGi Declarative Services and Blueprint are popular specifications in this area, however they provide new programming models that users need to learn. As of JavaEE 6, CDI (JSR 299) is included as a standard injection technology for JavaEE components. CDI becomes a core aspect of JavaEE platform. The JavaEE7 specification has further integrated CDI with many JavaEE components such as EJB, JSF, Bean Validation, JAX RS etc. It is enabled by default. The CDI programming model seems suitable for interaction with the OSGi service layer as well and has the benefit that developers who are familiar with CDI don't need to learn a new technology in order to interact with the OSGi service registry.

This document proposes that OSGi will support CDI with the goal of creating a specification that describes how the CDI programming model can be used to interact with OSGi services.

2 Application Domain

Software developers often need to build loosely coupled applications. The need for this stems from a number of factors:

- Developing reusable services for consumption outside of the team
- Allowing those services to be easily consumed
- Unit testing of applications and services
- Allowing larger teams to work effectively together by isolating areas of development

Software developers also wish to using a standardized programming model. This promotes:

- Transferability of skill sets
- Ease of sourcing new developers and low initial overhead

- Clear understanding of correct behavior when unexpected behavior is encountered
- Consistency of programming model across the technological strata to provide a uniformity of approach to aid understanding

Finally, software developers require an environment in which the focus can be on solving business issues rather than technological issues. This allows a more responsive development process.

2.1 CDI

CDI, Contexts and Dependency Injection is first specified by JSR 299 and then CDI 1.1 by JSR346. CDI 1.2 is the maintenance release of JSR346. It defines a clean, mostly annotations-based injection model which has recently become very popular. CDI is part of JavaEE 6/7 but can also be used standalone in a JavaSE context.

Weld (http://seamframework.org/Weld) is the Reference Implementation of JSR 299 and JSR 346.

2.1.1 Example

Although many advanced features are available, the most basic annotation used in CDI is `javax.inject.Inject` which declares the injection points for CDI.

For example the following Servlet class uses CDI injection to obtain an implementation of the WeatherBean interface.

```
public class CDIServlet extends HttpServlet {
    @Inject
    WeatherBean weatherBean;

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        PrintWriter writer = resp.getWriter();
        writer.print("The Weather in Amsterdam: " +
            weatherBean.getDescription("Amsterdam"));

        writer.flush();
        writer.close();
    }
}
```

While for the most basic use a CDI provider does not need to be annotated, CDI will attempt to find an implementor class and instantiate it using a no-arg constructor. Other mechanisms to publish a bean into CDI can be defined by using the `javax.enterprise.inject.Produces` annotation. Additionally, a number of scopes are defined that can be used to the declare the lifecycle of a CDI bean.

For example, the WeatherBean above can be scoped to the application lifecycle by adding the `javax.enterprise.context.ApplicationScoped` annotation, as in this example:

```
@ApplicationScoped
public class WeatherBeanProducer {
    @Produces @ApplicationScoped
    public WeatherBean newWeatherBean() {
        WeatherBean wb = new WeatherBeanImpl();
        wb.initialize();
        return wb;
    }
```

```
    public void disposeWeatherBean(@Disposes WeatherBean wb) {
        wb.cleanup();
    }
}
```

For more information see the CDI specification at JSR 299 Error: Reference source not found and JSR 346 [3]..

2.2 Weld-OSGi

The Weld-OSGi project (http://mathieuancelin.github.com/weld-osgi/) has created an integration between CDI and OSGi. It allows CDI beans to be exposed as OSGi services and CDI injections to be satisfied by OSGi services. Weld-OSGi takes additional OSGi features into account such as service registration properties and the dynamic aspects of the Service Registry.

Furthermore, Weld-OSGi provides annotation based injection for the Bundle, BundleContext, Bundle Headers and the private bundle storage facility.

Additionally Weld-OSGi provides annotations-based integration with Service and Bundle events.

2.2.1 Weld-OSGi example

Many examples can be found in the weld-osgi documentation [4].

Weld-OSGi typically uses additional annotations to interact with the OSGi service Registry. For example, the `org.osgi.cdi.api.extension.annotation.Publish` annotation publishes the CDI bean in the OSGi Service Registry:

```
@Publish
@ApplicationScoped
public class MyServiceImpl implements MyService {
    @Overrides
    public void doSomething() { ... }
}
```

To have a CDI injection come from the OSGi Service Registry, use the `OSGiService` annotation:

```
    @Inject @OSGiService MyService service;
```

OSGi Services can also be selected by using LDAP filters:

```
    @Inject @OSGiService @Filter("&(lang=EN)(country=US)") MyService service;
```

For more examples, see the weld-osgi documentation.

2.3 Declarative Services, Blueprint and CDI

In Java EE, the EJB and CDI containers are able to collaborate such that EJB manages an EJB component's life-cycle, whilst CDI manages its runtime dependencies. For example, when a new EJB is created it can be handed over to the CDI container for it to process the injections (@Inject) before finally being made available for use. This relationship helps ensure a complementary positioning between the different component models and reduces runtime duplication (EJB is not required to handle @Inject processing itself).

OSGi has two existing component models in the form of Declarative Services and Blueprint. Each has its own mechanism for injection of services and Blueprint also supports bean injection within a bundle. Neither has standards support for runtime annotations for injection, although there is some Blueprint prototype work in Apache Aries. In addressing any requirements for runtime annotations support, serious consideration should be given to the use of existing annotations, such as @Inject. It also makes sense to consider creating similar complementary relationship between their containers and the CDI container for runtime injection processing, thus reducing duplication between various component model containers.

2.4 Terminology + Abbreviations

CDI – Context and Dependency Injection for JavaEE. Specified in JSR 299 and JSR 346.

3 Problem Description

CDI provides a standardized, type-safe, loosely coupled programming model for Java EE 6 and above. Furthermore, it introduces powerful extensibility into the Java EE programming model, and promotes an ecosystem of “portable extensions”.

CDI is declarative, with metadata provided via annotations. This allows developers to locate all logic and metadata in a single location, allowing easier comprehension of the application.

CDI does not specify any modularity or inter-application communication, relying instead on the Java EE platform to provide this.

OSGi provides the de facto standard within Java for modular, service orientated programming.

Use of CDI in the context of OSGi provides a very compelling programming model. However, today there is no standard way to achieve this. A standard for leveraging CDI in OSGi will provide a migration path between JavaEE and OSGi where developers familiar with CDI can reuse their skills in both contexts without being locked in to a particular implementation.

4 Requirements

4.1 Functional Requirements

CDI001 – The specification MUST make it possible to use the CDI annotations and XML descriptor in an OSGi bundle to expose and consume CDI beans.

CDI003 – The specification MUST make it possible to consume OSGi services in CDI @Inject injection points in an OSGi bundle.

CDI004 – The specification MUST make it possible to select OSGi services used in CDI beans based on OSGi filters.

CDI005 – The specification MUST make it possible to consider CDI qualifiers when looking up CDI beans in the OSGi Service Registry.

CDI014 – The specification MUST provide a mechanism to specify additional OSGi service registration properties for CDI beans.

CDI006 – The specification MUST make it possible to write a portable CDI jar that runs both in JavaEE as well as in OSGi.

CDI007 – The specification MUST consider the thread-safety issues that can arise when migrating CDI beans from JavaEE to OSGi.

CDI008 – The specification MUST consider the issues that can arise in relation to the dynamic bundle lifecycle in OSGi.

CDI015 – The specification MUST consider the issues that can arise with OSGi service dynamism when these services are injected into a CDI bean.

CDI009 – The specification MUST make it possible to take advantage of the dynamic service capabilities of OSGi.

CDI016 – The specification MUST extend the life-cycle dependency model as provided in CDI, to support the dynamic life-cycle provided by OSGi. For example, it MUST NOT be fatal to deploy a CDI bean that does not have all its dependencies initially satisfied and it MUST be possible to change bean dependencies without requiring the CDI application to be redeployed or restarted.

CDI031 – The specification MUST extend the life-cycle dependency model of CDI to include dynamic OSGi service dependencies.

CDI017 – The specification MUST make it possible to declare a CDI injection point to an OSGi service as optional.

CDI018 – The specification MUST provide a mechanism to consume multiple matching services/beans of a given type in an injection point. For example via the @Inject Instance<T> mechanism.

CDI019 – The specification MUST support CDI events as defined by the CDI specification.

CDI021 – The specification MAY provide a deep integration between CDI events and OSGi events or other OSGi mechanism.

CDI020 – The specification MUST support CDI extensions as defined by the CDI specification but the extension should not be required to contain the file of META-INF/services/`javax.enterprise.inject.spi.Extension`. It should discover the extension via the service registry service of `javax.enterprise.inject.spi.Extension`.

CDI022 – the specification MAY provide a deep integration between CDI extensions and OSGi services or other OSGi mechanism.

CDI010 – The specification MAY introduce additional annotations.

CDI011 – The specification MUST define the behavior in case of incorrect CDI metadata.

CDI012 – The specification MUST NOT prevent the use of @Inject (and other common java annotations) in other component models/technologies present in the OSGi Framework.

CDI013 – The specification MUST define an opt-in mechanism. Bundles not opting in MUST not be considered by the CDI-OSGi integration layer.

CDI023 – All the inter-bundle interaction between CDI beans MUST go through the OSGi Service Registry.

CDI024 – The specification MUST make it possible to access the BundleContext from inside a CDI bean in an OSGi Framework.

CDI025 – The specification MUST provide support for @PostContruct and @PreDestroy activation and de-activation callbacks.

CDI026 – The specification SHOULD consider defining behavior for relevant CDI scopes.

CDI027 – The solution MAY define new scopes for use with CDI inside an OSGi Framework.

CDI028 – The specification MUST define an opt-in mechanism for CDI extensions.

CDI029 – The specification MUST consider the issues that arise from dynamically adding CDI extensions to the system.

CDI030 – The specification MUST support the inclusion of CDI beans and descriptors in a Web Application Bundle in the same way they can be included in a WAR -.

CDI032 – The specification MUST support the OSGi Service Permission security model when publishing OSGi services from CDI beans and injecting services into CDI beans. It needs to take into account that the CDI extender acts on behalf of other bundles and uses the permissions associated with those.

CDI033 – The specification implementation MUST support CDI 1,2 but may support CDI 1,0.

CDI034 – The specification MUST comply with CDI specification bean defining annotation for honoring a CDI bean not requiring the existence of beans.xml.

CDI034 – The specification SHOULD choose a service when multiple services are available for a particular instead of throwing AmbiguousResolutionException specified by CDI specification

CDI035 – The specification MUST exclude the classes from considering to be CDI beans if the classes are listed in the beans.xml under excludes elements.

4.2 Non-functional Requirements

CDI050 – The specification MUST NOT prevent an implementation from injecting OSGi services into CDI beans which are not deployed as OSGi bundles.

CDI052 – The specification MUST NOT prevent and implementation from CDI050 – The specification MUST NOT prevent an implementation from injecting OSGi services into CDI beans which are not deployed as OSGi bundles.

CDI051 – The specification SHOULD adhere to the current CDI programming model as much as possible.

4.3 Requirements from RFP 98 (OSGi/Java EE umbrella RFP)

JEE001 – A Java EE/OSGi system SHOULD enable the standard Java EE application artifacts (e.g. web application) to remain installed when a supporting Java EE runtime element (e.g. web container) is dynamically replaced.

JEE002 – RFCs that refer to one or more Java EE technologies MUST NOT impede the ability of an OSGI-compliant implementation to also be compliant with the Java EE specification.

JEE003 – RFCs that refer to one or more Java EE technologies MAY define the additional aspects of the technology that are required for the technology to be properly integrated in an OSGi framework but MUST NOT make any syntactic changes to the Java interfaces defined by those Java EE specifications.

JEE004 – RFCs whose primary purpose is integration with Java EE technologies MUST NOT require an OSGi Execution Environment greater than that which satisfies only the signatures of those Java EE technologies.

5 Technical Solution

5.1 Entities

- CDI – Contexts and Dependency Injection 1.0 (JSR-299), Contexts and Dependency Injection 1.1/1.2 (JSR346).
- CDI Provider – An implementation of the CDI 1.2 specification.
- CDI OSGi adapter – Adapts a given CDI Provider to the OSGi environment. This entity is implementation dependent and may or may not be separate from the CDI provider.
- CDI Bundle – A CDI-enabled OSGi bundle.
- CDI Container – A container for managed beans in a CDI Bundle. Each CDI Bundle has its own CDI container.
- CDI Extender – An application of the extender pattern to discover CDI Bundles and to manage the CDI container life-cycle on behalf of CDI Bundles.
- CDI Extension – A portable extension as defined in CDI 1.2.
- Extension Bundle – A bundle providing one or more CDI extensions. An extension bundle may or may not be a CDI Bundle at the same time.

- OSGi CDI Extension – A specific CDI extension for publishing and consuming OSGi services to or from managed beans by means of annotations. This is a mandatory part of this specification.

5.2 CDI Container Life-Cycle

The CDI Extender tracks all bundles becoming ACTIVE. When a tracked bundle is identified as a CDI Bundle, the CDI Extender creates a CDI Container for this bundle. When a tracked CDI Bundle is stopped, the CDI Extender stops the CDI Container for the given bundle.

Starting a CDI container requires scanning the CDI Bundle for managed bean candidate classes. Class loading scenarios are far more complex in OSGi than in Java EE or Java SE, due to the modular and dynamic nature of the OSGi environment.

The bean scanner needs to consider all classes on the bundle classpath of the CDI Bundle, including any embedded archives or directories and fragments . This maps to the resources returned via the BundleWiring.listResources(“/”, “*.class”, 0) method call. The imported classes will not included as beans.

The set of candidate bean classes determined by the bean scanner is *not* equal to the set of managed beans in the bean container: The CDI Provider discards all candidate classes that do not satisfy the requirements for managed beans. The set of managed beans may be further extended or modified by CDI extensions.

5.2.1 Initialization of a CDI Container

A CDI extender manages the life cycle of a CDI bundle based on:

- The CDI bundle state
- The injection state
- The CDI extender's bundle statement

All activities on behalf of the CDI bundle must use the Bundle Context of the CDI bundle. All dynamic class loads must use the CDI bundle's BundleloadClass method.

In the following description of the initialization steps, The CDI Container will update its state. State changes are broadcast as events [5.11.1]

If any failures occurs during initialization, or the CDI bundle or CDI extender bundle is stopped, the CDI container must be destroyed.

5.2.1.1 Initialization steps

The initialization process of a CDI Container is defined in the following steps:

1. Wait until a CDI bundle is ready. A CDI bundle is ready when it is in the ACTIVE state, and for CDI bundles that have a lazy activation policy, also in the STARTING state.
2. State = CREATING and fire 'Creating' event
3. Processing injection points

4. If all mandatory service references are satisfied, the CDI Container is ready to provide component instances. Otherwise, it enters state=WAITING and fire 'Waiting' event.

5. When the mandatory services references are satisfied, the CDI Container is now ready to provide component instances.

6. State= CREATED and fire 'Created' event

7. Register the CDI Container.

8. The components are now active and perform their function until the CDI bundle or the CDI extender bundle are stopped

9. When the CDI bundle is stopped, the CDI container will go to destroying state. State = DESTROYING and fire 'Destroying' event

10. perform the Destroy phase

11. State=DESTROYED and fire 'Destroyed' event

5.2.1.2 Failure

If at any time there is a failure, the CDI Container must perform the following steps:

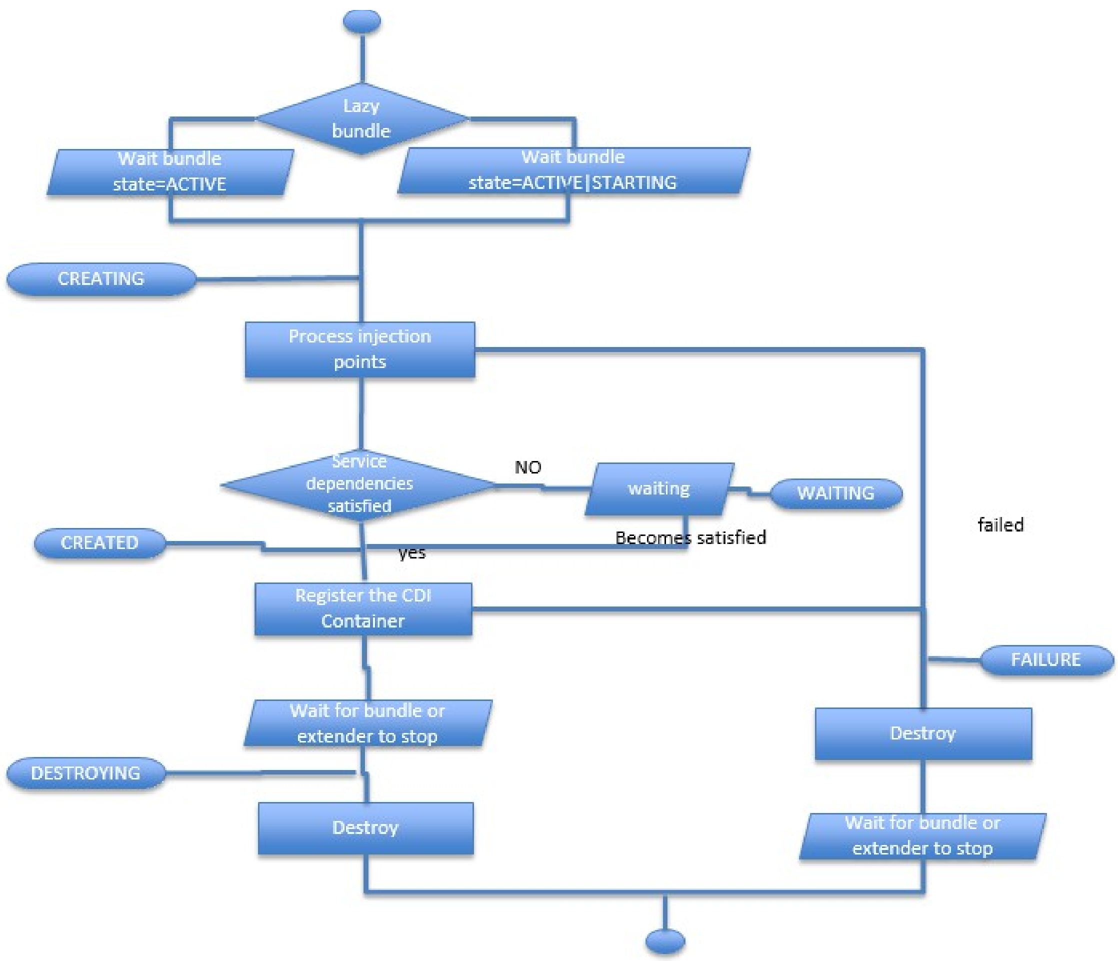
1. State= FAILURE

2. Unregister the CDI Container service

3. Destroy the CDI Container

4. Wait for the CDI bundle to be stopped

5.2.1.3 Diagram



5.2.1.4 Runtime Phase

The CDI Container must be registered as a service with the following service properties:

osgi.cdi.container.sybmolicname – The bundle symbolic name of the CDI bundle

osgi.cdi.container.version – The version of the CDI bundle

The CDI container service must only be available during the runtime phase when initialization has succeeded.

As long as the CDI extender and the CDI bundle is active, the application is in the runtime phase.

5.2.1.5 Destroy the CDI Container

The CDI Container must be destroyed when any of the following condition becomes true:

- The CDI bundle is stopped.
- The CDI extender is stopped
- One of the initialization phases failed.

Destroying the CDI Container means:

- Unregistering the CDI Container service
- Destroying all component instances in reverse dependency order

A CDI Container must continue to follow the destruction even when component instances throw exceptions or other problems occur. These errors should be logged.

If the CDI extender is stopped, then all its active CDI Containers must be destroyed in an orderly fashion, synchronously with the stopping of the CDI extender bundle. The CDI Containers must be destroyed, using the the following algorithm:

1. Destroy CDI Containers that do not have any services registered that are in use by other bundles.
2. The previous step can have released services, therefore, repeat step 1 until on more CDI Containers can be destroyed.
3. If there are still CDI Containers that are not destroyed, then destroy the CDI Containers with:
    - The registered service that is in user with the lowest ranking number.
    - The highest registered service id

    If there there still CDI Containers to be destroyed, retry step 1

During the shutting down of an OSGi framework, it is likely that many bundles are stopped near simultaneously. The CDI extender should be to handle this case, without deadlock, when the stop of a CDI Bundle overlaps with the stop of the CDI extender bundle.

5.2.1.6 Failure

If a failure occurs during the initialization of the CDI bundle, then first a FAILURE event must be posted. Then the CDI Container should be destroyed, ensuring that no uninitialized or half initialized objects are destroyed.

5.3 Requirements and Capabilities

Bean deployment archives according to CDI 1.0 are required to opt in to OSGi enrichment by the CDI Extender. However, opting in may have no effect at all if a would-be CDI Bundle is installed and started in a system where no CDI Extender is available.

This kind of dependency can be made explicit using capabilities (introduced in OSGi Core 4.3). Capabilities are also useful to express a loose dependency on a given CDI extension. This covers the following use case:

CDI Bundle A works with a CDI extension with an annotation API defined in bundle B and an extension implementation in bundle C. A has a package dependency on B, but not on C. There is only an implicit runtime dependency on C. C needs to be resolved when the CDI container for A is constructed, so that the extension can modify the set of manged beans, but C may not be available at all.

Extension capabilities are a means to declare a dependency on a given extension *by name* and not by implementation.

For these reasons, this specification defines the following capabilities.

- An OSGi extender capability named osgi.cdi.

- A capability osgi.cdi.extension with a mandatory attribute extension = <*name*>, where <*name*> is a logical name for the given extension. The name osgi.cdi is reserved for the OSGi CDI extension defined in this specification.

A CDI Bundle MUST require the OSGI extender capability named osgi.cdi, e.g.

```
Require-Capability: osgi.extender; filter:="(osgi.extender=osgi.cdi)"
```

(Note that this is not a *sufficient* condition for a bundle to be a CDI Bundle. A CDI Bundle must also be a bean deployment archive as defined in CDI 1.2, i.e. it contains a beans.xml descriptor in one of the defined locations with a version number of 1.1 or later and the bean-discoverty-mode of 'all' or with no version number or that is an empty file. Alternatively, it contains one or more bean classes with a bean defining annotation as defined by the JSR346,.)

A CDI bundle can define a list of bean classes, where the classes must be defined in this bundle not imported classes.

```
Provide-Capability:...;beans:List<String>="foo.A,foo.B".
```

The bean classes will be filtered based on the beans.xml. e.g. If foo.A is excluded by beans.xml, foo.A will not be included. If there is no beans.xml, the above defined list will be the bean classes of the bundle and a scan will not be performed. If no list is provided and no beans.xml exists, a scan will be performed to find bean classes.

A CDI Bundle MAY require additional osgi.cdi.extension capabilities with other filter attributes for other CDI extensions.

A CDI Extender implementation MUST provide the osgi.extender capability named osgi.cdi with version 1.0, e.g.

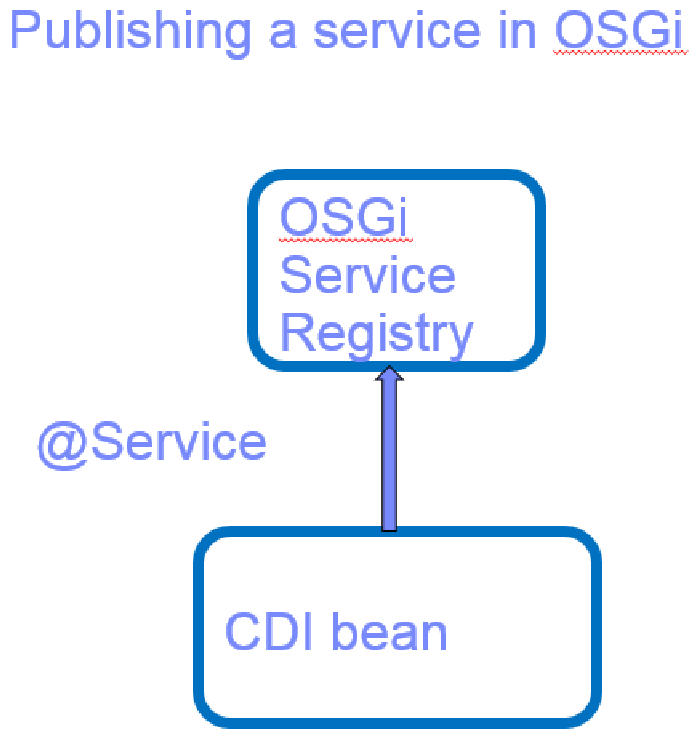
```
Provide-Capability = osgi.extender; osgi.extender=osgi.cdi; version=1.0
```

Future versions of this specification will require higher version numbers.

An OSGi-enabled CDI extension MUST provide an osgi.cdi.extension capability with a distinctive name, e.g.

```
Provide-Capability = osgi.cdi.extension; extension=frobnicator
```

5.4 Managed Beans and OSGi Services




```
@Service(properties = {@ServiceProperty(key = "key",
value = "value"), @ServiceProperty(key ="key2",
value = "42", type = "long")})
public class ExampleComponent {}
```

A class can be published to the OSGi service registry using the @org.osgi.service.cdi.~~Component~~.Service annotation. Note that classes that are already CDI beans, are not automatically published to the service registry; the ~~@Component~~@Service annotation or metadata as described in section 5.4.1 is always required for this. The requirement to explicitly annotate a bean to be an OSGi component is necessary for the following reasons:

1. In CDI 1.2 every class is a potential CDI bean, no annotations are required for this. The container scans injection points at container startup time to calculate which classes should be registered as beans. A CDI container in OSGi is scoped to a single bundle, and will only know about injection points in that bundle. OSGi services however can be used by other bundles as well. Although it is technically possible to find injection points in all bundles, this would be very hard to reason about for end users.

2. Not each CDI bean should be registered to the service registry. CDI is often used to inject tightly coupled classes into each other. This is find for internal bundle usage, but should not be reflected in the service registry.

3. Not all CDI beans might be exported. It's useless to publish services of types that other bundles can't have access to.

This behavior is different then the behavior described in the EJB integration specification, where all EJBs are published as OSGi services by default. Although technically EJBs and CDI beans are very similar, their usage in practice is often very different. EJBs tend to be used in a similar granularity as OSGi services, while CDI beans are not.

An example ~~@Component~~@Service annotation:

~~@Component~~@Service(interfaces={A.class})
public class MyComponent implements A,B {}

The `interfaces` annotation element is optional. If not specified the component is registered in the OSGi Service Registry under all the interfaces it directly implements. If the component does not implement any interfaces it will be registered under its implementation class.

Service properties can be defined in the ~~@Component~~@Service annotation using the “properties” argument. The properties argument takes an array of ~~@Component~~@ServiceProperty annotations. The ~~@Component~~@ServiceProperty annotations requires a key and value to be set, both of type String. An optional type element can be used to specify the data type of the property value. Supported data types are: service property values may only be one of the following types:

- Scalar – String, Integer, Long, Float, Double, Byte, Short, Character, Boolean.
- Array – An array of the allowable scalar types, declared using the following syntax: String[] or Float[].
- Collection – A list or set that contains scalar types declared using the following syntax: List<String> or Set<Short>.

~~@Component~~@Service(properties = {~~@Component~~@ServiceProperty(key = "key", value = "value"),
~~@Component~~@ServiceProperty(key = "key2", value = "42", type = “long”)})
public class ExampleComponent {}

5.4.1 Publishing plain CDI providers in the Service Registry

To support integration of existing CDI provider classes with OSGi an alternative to the `~~@Component~~@Service` annotation is available via the OSGi bundle manifest. This allows beans that act as providers in plain CDI environments to participate in the OSGi integration as services without having to change and recompile these components. A class can be declared as a component in the manifest through the `component` directive on the `osgi.extender=osgi.cdi` requirement:

```
Require-Capability: osgi.extender;
  filter:="(osgi.extender=osgi.cdi)";
  components:="org.acme.MyComponent,org.acme.ExampleComponent"
```

5.5 Scopes for beans annotated as ~~@Component~~@Service

The following table outlines the CDI scopes and their support by this specification:

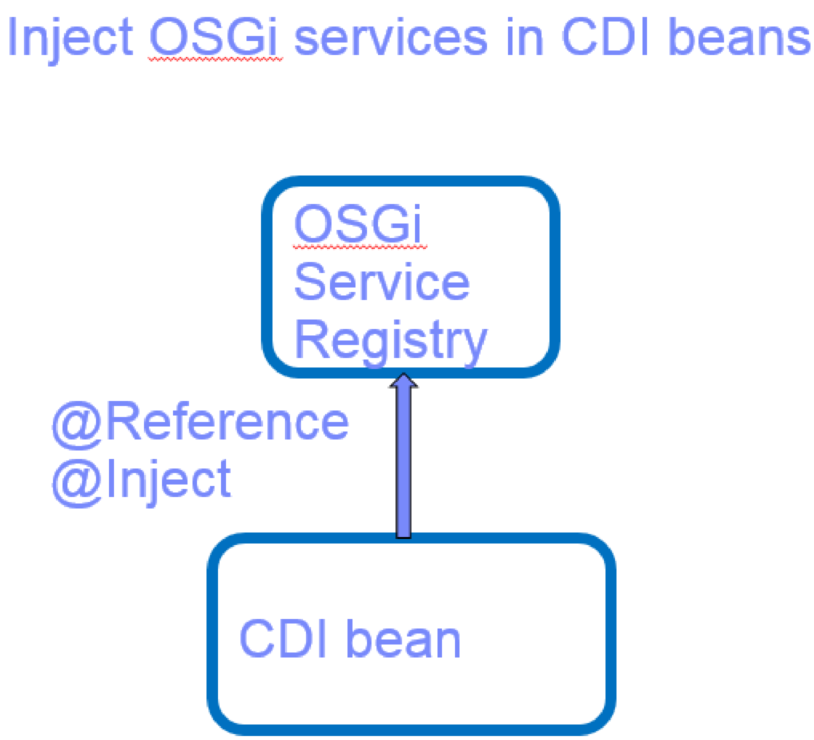
`@ApplicationScoped`	Same as @Singleton.
`@ConversationScoped`	-web request specific
`@Dependent`	Default, map to @Singleton
`@SessionScoped`	web request specific

`@Singleton`	Same as @SingletonScoped
`@RequestScoped`	web request specific

The following table lists OSGi-defined scopes and associated behavior. These scopes are only relevant for CDI beans registered in the OSGi Service Registry:

`@BundleScoped`	Maps to OSGi Service Factory
`@PrototypeScoped`	Maps to OSGi Prototype Service Factory
`@SingletonScoped`	Maps to normal OSGi singleton service

5.6 Life-Cycle



The life-cycle of an ~~@Component~~@Service is different from a normal CDI bean, because it has to deal with service dynamics. An ~~@Component~~@Service may have dependencies on other OSGi services which can influence the life-cycle of the ~~@Component~~@Service as described in the next section.

An OSGi service can be injected into another OSGi Service or a CDI bean using the standard @Inject annotation. The life-cycle of an OSGi service is different than a plain CDI bean however. By default, a CDI bean has the *Dependent* scope; it's life-cycle is bound to the life-cycle of the consuming bean, or alternatively it's scope is defined using annotations. For a mapping of CDI scope to OSGi behaviour, see section 5.5, however not all CDI scopes are supported in an OSGi environment. Another difference is that there may be multiple OSGi services publishing the same interface, a client may or may not choose a specific instance using service properties and ordering.

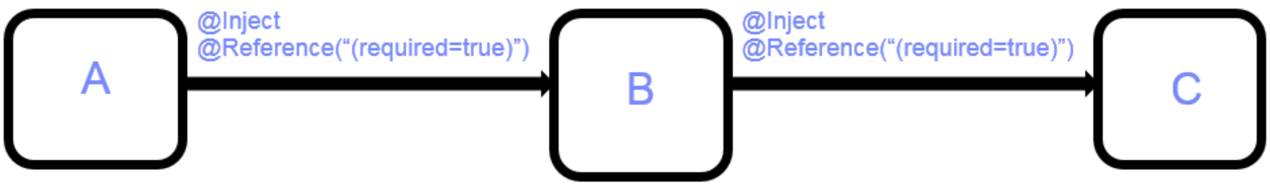
Because of the differences between the CDI BeanManager and OSGi service registry and to remove ambiguity we must always use @~~Service~~Reference CDI qualifier to instruct the CDI container (and the developer) that we are dealing with a specific type of bean, where different rules may apply. When the @~~Service~~Reference annotation is used, the CDI container will lookup the OSGi service in the service registry and inject it's instance into the CDI bean.

Any OSGi service registered in the service registry can be injected using CDI, even if the service was not registered using CDI but with the low level service API or DS for example.

5.6.1 Required dependencies (~~@Component~~@Service only)

Require a dependency; the component will not be registered when a required dependency is not available. This is the default behavior. When all dependencies becomes available, the component will be registered. If at some point during runtime a required dependency becomes unavailable, the component will be deregistered again. This model handles service dynamics correctly, while the code doesn't have to handle the case where dependencies are not available.

As an example we have the dependency structure:



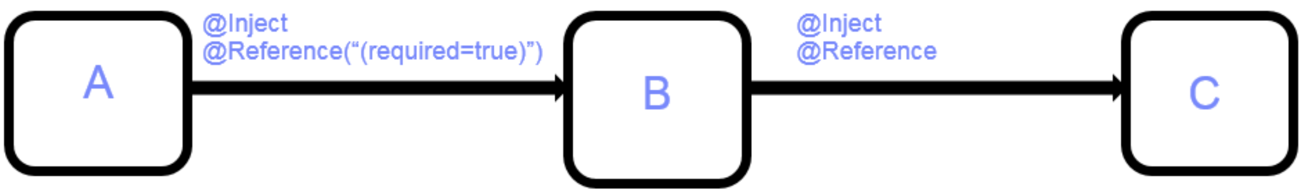
When C becomes unavailable, B will become unavailable as well, and so will A. When C becomes available again, so will be B, and so will A.

5.6.2 Optional dependencies

To deal with service dynamics a proxy is injected instead of the real reference to the service. The proxy is by default a null-object, every method invocation should return null. The code that uses an optional dependency must deal with the possibility that the service is not available and handle null values properly.

Alternatively the @~~Service~~Reference supports a configuration parameter "proxyType" that can be used to configure the proxy to throw a org.osgi.cdi.ServiceNotAvailableException when the proxy is invoked while the service is not available. This way clients can handle null values returned by the real service differently than the situation where the service is not available.

Optional and required dependencies can be mixed in a single component and in the dependency graph of a component. Take the following example again, note that the dependency on C is now optional.



When C becomes unavailable, B should still be available, and therefore A as well. When B becomes unavailable, A should still become unavailable as well. Because the dependency on C is now optional, the code of B should be handing the fact that method invocations on C might return null or throw exceptions.

CDI does not support dynamic dependencies. All beans must be registered and all dependencies must be resolved at container startup. For normal CDI beans the container should resolve @Inject @~~Service~~Reference injection points immediately with proxies as described for optional dependencies. Required dependencies are not required to be implemented by the container for normal CDI beans; the container may throw an exception to inform the developer that the required attribute is not supported. A CDI container may implement required dependencies for normal CDI beans as well too however.

For components registered with the ~~@Component~~@Service annotation, required dependencies must be supported. An ~~@Component~~@Service with unavailable dependencies must not be registered to the OSGi service registry, and it's @PostConstruct method must not be invoked. If the component was active before the dependency became unavailable, the @PreDestory method must be called and the service must be de-registered from the service registry.

Because OSGi services are dynamic, a developer should make the explicit choice to inject beans using the OSGi service registry by using the @~~Service~~Reference qualifier. If the @~~Service~~Reference qualifier is not used, the container should not query the service registry. @Reference can be used on fields or method parameters.

5.6.3 Allowing Service Injections in vanilla CDI injection points

As described in previous sections, the life-cycle and dynamics of OSGi makes this a fundamentally different environment to the JavaEE container for which CDI was originally designed. This is one of the key reasons for introducing the `@Reference~~Service~~` annotation to go with the `@Inject` annotation.

However in certain scenarios it may be desirable allow injections from the OSGi service registry into plain `@Inject` injection points. This is to allow migration paths from pure JavaEE to configurations where unmodified CDI beans are injected with services from the OSGi service registry. Clearly such a situations introduce risks and analysis of the system is necessary beforehand to ensure that no fatal situations can occur by the introduced dynamics.

Once satisfied that the system can cope in the OSGi environment, injection into pure `@Inject` points can be enabled by setting a directive on the extender requirement in the CDI Bundle:

```
Require-Capability: osgi.extender;
  filter:="(osgi.extender=osgi.cdi)";
  at-service:="optional"
```

5.6.4 Service and bundle registration observers

In most situations service dependencies are injected directly into a field. Sometimes some extra code needs to be executed however. This can be done using callback methods. There is a callback method for service registration and a callback for service deregistration. The parameters of the observer method should be the type of the service.

Only events fired while the bundle containing the observer methods are delivered. Events fired while the bundle was not active are ignored.

```
void serviceAdded(@Observes @ServiceAdded SomeService) ;
void serviceRemoved(@Observes @ServiceRemoved  SomeService) ;
void serviceModified(@Observes @ServiceModified  SomeService) ;
```

In some cases it's useful to also have access to the ServiceReference representing a service. For this case a special event type is introduced:

```
public class ServiceCdiEvent<T> {
  private ServiceReference<T> reference;
  private T service;
  // constructor and getters etc.
}

void serviceAdded(@Observes @ServiceAdded ServiceCdiEvent<SomeService> event)
void serviceRemoved(@Observes @ServiceRemoved ServiceCdiEvent<SomeService> event)
```

The @ServiceAdded and @ServiceRemoved annotations have the same service filtering semantics as @~~Service~~Reference described in the following section.

5.6.5 Service Filters

Service filtering can be done using the @Reference~~Service~~ value parameter using the standard OSGi service filter (LDAP like) syntax:

@~~Service~~Reference("(somekey=somevalue)")

As a convenience alternatively the same could be done using CDI Qualifiers:

@Qualifier

@~~Service~~ReferenceFilter


```
public @interface SomeKey {
 String value();
}
@Inject @~~Service~~Reference @SomeKey("somevalue")
MyService service;
```

The name of the qualifier is used as the name of the property, the value passed into the qualifier annotation the value. If the value is not a String (e.g. an enum value), the String representation of the value is used. If the qualifier has a default value, the annotation can be used without specifying a value. In the following example the used filter is "(somekey=somevalue)". This mechanism provides a slightly more type-safe approach to using service filters.

```
@Qualifier
@~~Service~~ReferenceFilter
public @interface SomeKey {
 String value() default "somevalue";
}
@Inject @~~Service~~Reference @SomeKey
MyService service;
```

It is possible to combine multiple qualifiers to create AND filters. In the following example both annotations are qualifiers, and the resulting filter is "(&(somekey=somevalue)(someotherkey=someothervalue))".

```
@Inject @~~Service~~Reference @SomeKey("somevalue") @SomeOtherKey("someothervalue")
MyService service;
```

More complex filters are not supported using qualifiers, the standard filter syntax should be used instead with the @~~Service~~Reference annotation.

When using OSGi service filters it is common to use dots in property names, e.g. 'service.ranking'. It is not possible however to use dots in a Java Annotation type name. To work around this issue the Annotation type name is parsed and certain markers in the name are interpreted as dots. Because service properties are case insensitive, camel casing can be used as a marker.

For example the qualifier @MyServiceProperty("example") translates to "(my.service.property=example)". The first character is lowercased. Each following capital is translated to a dot.

Requiring configuration

An ~~@Component~~@Service can require a Config Admin Configuration object with a specific PID. The component will only become available when a configuration object with the specified PID is found. This is useful when a component does not have usable default configuration values. If the configuration object is not available, the behaviour is the same as for a unavailable required service dependency.

A configuration dependency can be configured as follows:

~~@Component~~@Service(requireConfiguration="PID.of.configuration")

5.7 BundleContext injection

It's possible to inject the BundleContext using CDI annotations.

```
@Inject BundleContext context;
```

This will always inject the BundleContext of the bundle that registered the component, even if the component's class was imported from another bundle. For example we could have the following scenario. Although the @Inject BundleContext annotation is declared in a class exported by Bundle A, Bundle B does the actual component registration. This means that the BundleContext of bundle B will be injected.

Bundle A

public abstract class MyBaseClass {

@Inject BundleContext bundleContext;

}

Bundle B

~~@Component~~@Service

public class MySubClass extends MyBaseClass {

}

5.8 `BeanManager` service

The implementation will register a service registered under the `javax.enterprise.inject.spi.BeanManager` interface. The BeanManager provides a standard portable intropspective interfaces into the CDI container.

The BeanManager service is registered under the Bundle Context of the associated CDI bundle, as each CDI bundle has its own container. As a result many CdiContainer services will be present in the system.

5.9 `osgi.extender` Capability

The implementation must provide an osgi.extender capability as follows:

```
Provide-Capability: osgi.extender;
  osgi.extender="osgi.cdi";
  uses:="org.osgi.service.cdi,javax.enterprise.inject.spi";
  version:Version="1.0"
```

5.10 `osgi.contract` Capability

The OSGi Enterprise specification version 5 defines the `osgi.contract` capability namespace and RFC 180 defines mappings of JSR-defined technologies to these capabilities. Relevant technologies from RFC 180 to this specification are the `JavaCDI` and `JavaInject` API contracts.

An implementation of this specification is not required to export the associated `javax.**` packages, but if it does, it must also provide the `JavaCDI` and `JavaInject` capabilities in the `osgi.contract` namespace.

5.11 Events

The CDI Container must track all CDI Listener services and keep the listeners updated of the progress of all its managed bundles.

CDI Events must be sent to each registered CDI Listeners service. The service has the following method:

- cdiEvent(CdiEvent) – notify the listener of a new CDI Event synchronously.

5.11.1 CDI Event

The CDI Event supports the following event types:

- CREATING - The CDI extender has started creating a CDI Container for the bundle.
- CREATED - The CDI Container is ready. The application is now running.
- WAITING - A service reference is blocking because of unsatisfied mandatory dependencies. This event can happen multiple times in a row.
- DESTROYING - The CDI Container is being destroyed because the CDI bundle or CDI extender has stopped.
- DESTROYED - The CDI Container is completely destroyed.
- FAILURE - An error occurred during the creation of the CDI Container.

The CDI Event provides the following methods:

- getBundle() - The CDI bundle
- getCause() - Any occurred exception or null
- getDependencies() - A list of filters that specify the unsatisfied mandatory references.
- getExtenderBundle() - The CDI extender bundle.
- getTimestamp() - The time the event occurred
- getType() - The type of the event.

- isReplay() - Indicates if the event is a replay (true) or if it is a new event (false)

5.11.2 Replay

The CDI Extender must remember the last CDI Event for each ready bundle that it manages. The replay event must be delivered to the CDI Listener service as the first event, before any other event is delivered , during the registration of the CDI Listener service. The CDIEvent object for a replay event must return true for the isReplay() method is this situation, and false in all other situations.

6 Data Transfer Objects

RFC 185 defines Data Transfer Objects as a generic means for management solutions to interact with runtime entities in an OSGi Framework. DTOs provides a common, easily serializable representation of the technology.

For all new functionality added to the OSGi Framework the question should be asked: would this feature benefit from a DTO? The expectation is that in most cases it would.

The DTOs for the design in this RFC should be described here and if there are no DTOs being defined an explanation should be given explaining why this is not applicable in this case.

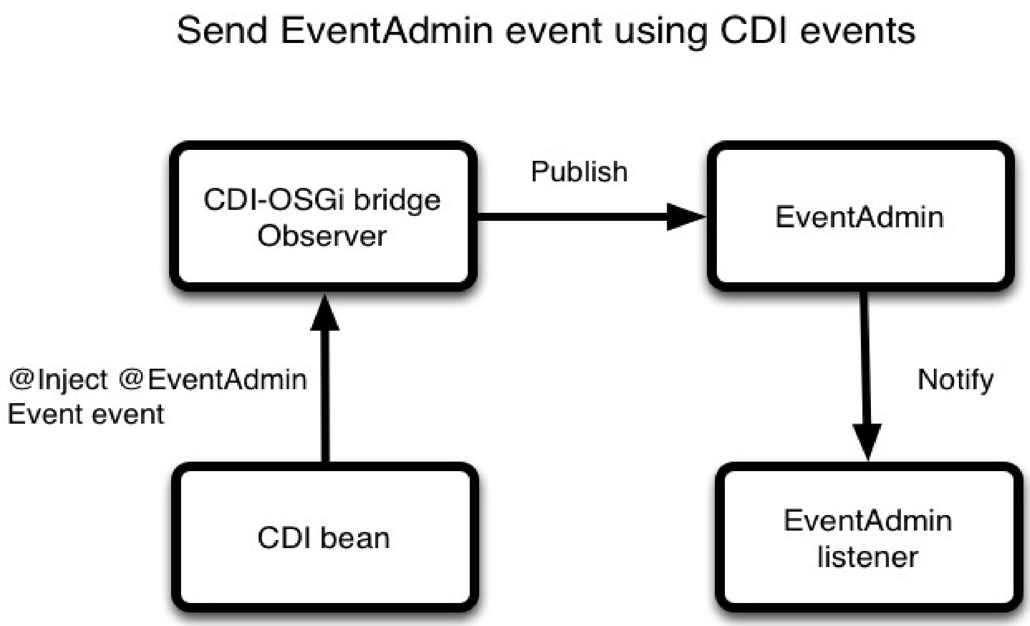
This section is optional and could also be provided in a separate RFC.

7 Javadoc

8 Considered Alternatives

8.1 EventAdmin integration

This section has moved into the Considered Alternatives chapter as it's postponed to a later release.



EventAdmin and CDI events are conceptually similar. The CDI programming model is much more user-friendly however. EventAdmin events can both be produced and observed using CDI annotations. Because not every event should be published to EventAdmin the developer has to use the @EventAdmin annotation. The value of this annotation should also contain the name of the *Topic,* or alternatively you can define your own qualifier that extends @EventAdmin to define the Topic.

The following is an example of using the @EventAdmin qualifier.

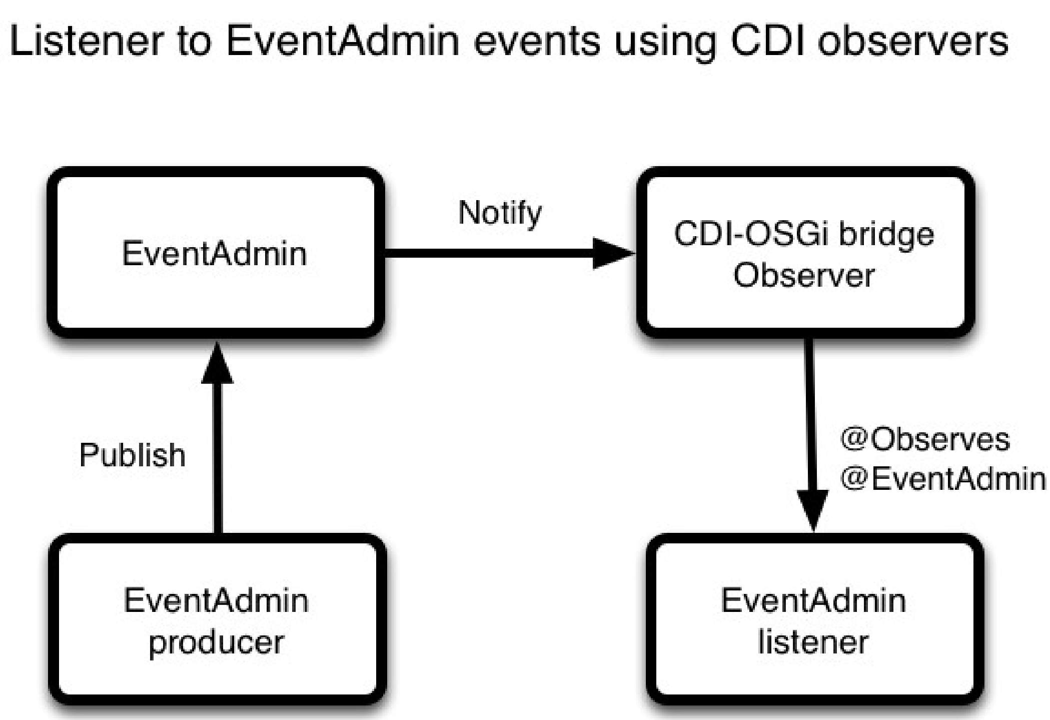
```
@Inject @EventAdmin("MyTopic") Event<MyEvent> event;

public void send() {

   event.fire(new MyEvent("example"));                              }

}
```

The following is an example of extending the EventAdmin qualifier.

```
@EventAdmin
```

```
public @interface Demo {
  String value();
}
@Inject @Demo Event<MyEvent> event;
public void send() {
   event.fire(new MyEvent("example"));                                  }
}
```

The CDI-OSGi bridge observes @EventAdmin events and republish them as EventAdmin events.



EventAdmin events can be observed using the CDI @Observes annotation. Similar to publishing EventAdmin events we need the @EventAdmin qualifier to specify the Topic name.

```
public class EventExample {
public void process(@Observes @EventAdmin("MyTopic") MyEvent event) {}
```

```
}
```

Alternatively, similar with publishing events, a qualifier can be used to define the topic name.

```
@EventAdmin
public @interface Demo {
   String value();
}


public class EventExample {
   public void process(@Observes @Demo MyEvent event) {}
}
```

9 Security Considerations

Description of all known vulnerabilities this may either introduce or address as well as scenarios of how the weaknesses could be circumvented.

10 Document Support

10.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0***JSR 299: Contexts and Dependency Injection for Java EE flatform***

[3]. JSR 346: Conexts and Dependency Injection for Java EE

[4]. Weld OSGi Reference: http://docs.jboss.org/weld/reference/1.2.0.Beta1/weld-osgi/user-guide/html/ch01.html

10.2 Author’s Address

Name	
Company	
Address	
Voice	
e-mail	

10.3 Acronyms and Abbreviations

10.4 End of Document