RFC 193 - CDI Integration

Draft

28 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

The OSGi Alliance hereby grants you a limited copyright license to copy and display this document (the “Distribution”) in any medium without fee or royalty. This Distribution license is exclusively for the purpose of reviewing and providing feedback to the OSGi Alliance. You agree not to modify the Distribution in any way and further agree to not participate in any way in the making of derivative works thereof, other than as a necessary result of reviewing and providing feedback to the Distribution. You also agree to cause this notice, along with the accompanying consent, to be included on all copies (or portions thereof) of the Distribution. The OSGi Alliance also grants you a perpetual, non-exclusive, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights, to create and/or distribute an implementation of the Distribution that: (i) fully implements the Distribution including all its required interfaces and functionality; (ii) does not modify, subset, superset or otherwise extend the OSGi Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the OSGi Name Space other than those required and authorized by the Distribution. An implementation that does not satisfy limitations (i)-(ii) is not considered an implementation of the Distribution, does not receive the benefits of this license, and must not be described as an implementation of the Distribution. "OSGi Name Space" shall mean the public class or interface declarations whose names begin with "org.osgi" or any recognized successors or replacements thereof. The OSGi Alliance expressly reserves all rights not granted pursuant to these limited copyright licenses including termination of the license at will at any time.

EXCEPT FOR THE LIMITED COPYRIGHT LICENSES GRANTED ABOVE, THE OSGi ALLIANCE DOES NOT GRANT, EITHER EXPRESSLY OR IMPLIEDLY, A LICENSE TO ANY INTELLECTUAL PROPERTY IT, OR ANY THIRD PARTIES, OWN OR CONTROL. Title to the copyright in the Distribution will at all times remain with the OSGi Alliance. The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted therein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

THE DISTRIBUTION IS PROVIDED "AS IS," AND THE OSGi ALLIANCE (INCLUDING ANY THIRD PARTIES THAT HAVE CONTRIBUTED TO THE DISTRIBUTION) MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DISTRIBUTION ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.
NEITHER THE OSGi ALLIANCE NOR ANY THIRD PARTY WILL BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RELATING TO ANY USE OR DISTRIBUTION OF THE DISTRIBUTION.

Implementation of certain elements of this Distribution may be subject to third party intellectual property rights, including without limitation, patent rights (such a third party may or may not be a member of the OSGi Alliance). The OSGi Alliance is not responsible and shall not be held responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

The Distribution is a draft. As a result, the final product may change substantially by the time of final publication, and you are cautioned against relying on the content of this Distribution. You are encouraged to update any implementation of the Distribution if and when such Distribution becomes a final specification.

The OSGi Alliance is willing to receive input, suggestions and other feedback (“Feedback”) on the Distribution. By providing such Feedback to the OSGi Alliance, you grant to the OSGi Alliance and all its Members a non-exclusive, non-transferable,

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

0.5 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	10/16/12	Initial version of the draf. David Bosschaert <david@redhat.com>
2nd draft	10/26/12	First draft of chapter 5 Paul Bakker <paul.bakker@luminis.eu>
3rd draft	10/30/12	Life-cycle callbacks Paul Bakker <paul.bakker@luminis.eu>

Revision	Date	Comments
4th draft	01/27/13	Details on service dynamics Paul Bakker <paul.bakker@luminis.eu>
5th draft	02/27/13	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>

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. 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 specified by JSR 299. It defines a clean, mostly annotations-based injection model which has recently become very popular. CDI is part of JavaEE 6 but can also be used standalone in a JavaSE context.

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

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

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 Error: Reference source not found

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.

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.

~~CDI002 – The specification MUST make it possible to access all CDI managed beans from the OSGi Service Registry.~~

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 satisifed 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 ~~SHOULD~~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.

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 ~~SHOULD~~MUST provide support for @PostContruct and @PreDestroy activation and de-activation callbacks ~~similar to the BundleActivator methods to CDI beans in an OSGi Framework.~~.

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 (e.g. including beans.xml in WEB-INF/).

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.

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).
- CDI Provider – An implementation of the CDI 1.0 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.
- Bean Bundle – A CDI-enabled OSGi bundle.
- CDI Container – A container for managed beans in a bean bundle. Each bean bundle has its own CDI container.
- CDI Extender – An application of the extender pattern to discover bean bundles and to manage the CDI container life-cycle on behalf of bean bundles.
- CDI Extension – A portable extension as defined in CDI 1.0.
- Extension Bundle – A bundle providing one or more CDI extensions. An extension bundle may or may not be a bean 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 bean bundle, the CDI Extender creates a CDI Container for this bundle. When a tracked bean bundle is stopped, the CDI Extender stops the CDI Container for the given bundle.

Starting a CDI container requires scanning the bean 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 bean bundle, including any embedded archives or directories.
- All classes contained in packages imported by the bean bundle from other bundles wired to the given bundle.
- All classes exported by required bundles wired to the bean bundle.
- TODO: What about dynamic package imports?

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.

5.3 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 bean 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:

Bean 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.

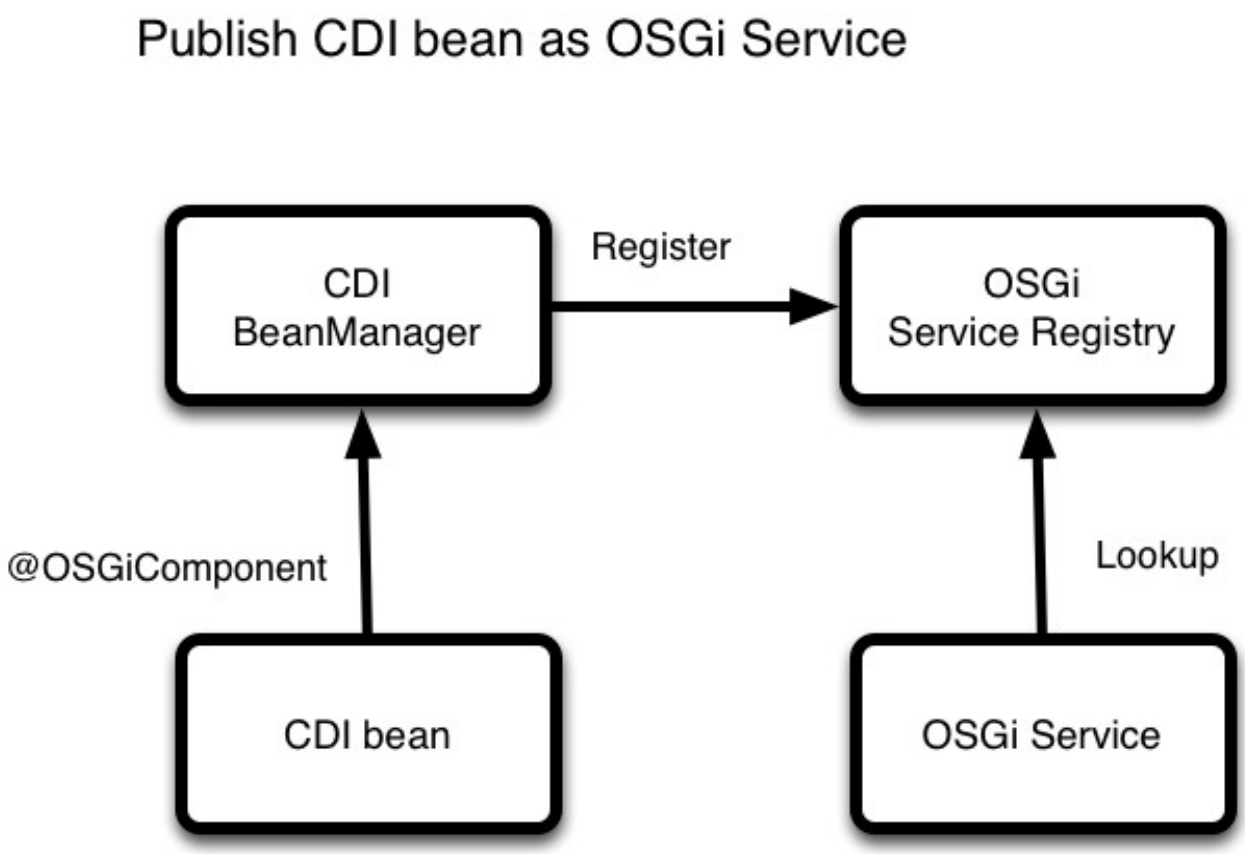
A bean bundle MUST require the OSGI extender capability named osgi.cdi, e.g.

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

(Note that this is not a *sufficient* condition for a bundle to be a bean bundle. A bean bundle must also be a bean deployment archive as defined in CDI 1.0, i.e. it must have a beans.xml descriptor in one of the defined locations.)

~~A bean bundle SHOULD require the OSGI CDI extension, e.g.~~

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

~~This capability is not required if the given bean bundle does not publish or inject any OSGi services via CDI mechanisms.~~

A bean 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
```



~~using CDI annotations, but the following rules apply:~~

~~There is no scope defined. This concept is not supported by OSGi services. The container may throw an exception when a scope is defined.Any Java class can be published as an OSGi service~~

~~Note that there is a relation to RFC 195: Service Scopes. This RFC might make it possible to scope services in a similar way that CDI scopes work (e.g. session scoped). We are following this RFC and will try to use this to create tight integration with CDI scopes if possible.~~

A class can be published to the OSGi service registry using the @OSGiComponent annotation. Note that classes that are already CDI beans, are not automatically published to the service registry; the @OSGiComponent annotation 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 1.0 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.

The class doesn't have to have an interface defined. If no interface is defined the service will be registered with Object.class as the service interface which is sometimes useful for whiteboard style registrations. The bean's type itself might not be exported and is therefore not published. If an interface is defined, the service is registered using this interface. If multiple interfaces are defined the developer can optionally explicitly define the list of registered interfaces using the @OSGiComponent annotation. By default all interfaces will be published.~~:~~

```
@OSGiComponent(intefaces={A.class})

public class MyComponent implements A,B {}
```

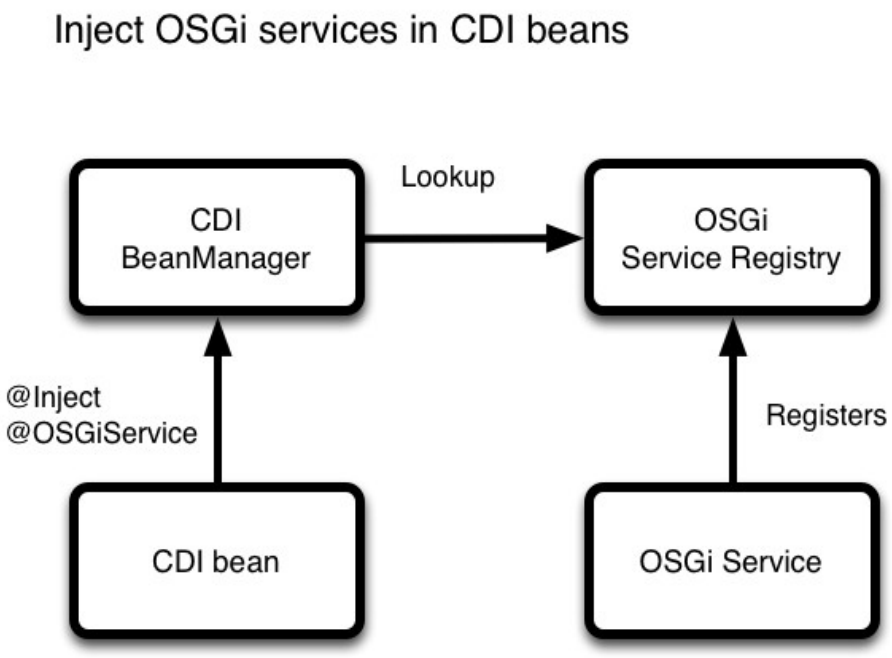
Service properties can be defined in the @OSGiComponent annotation using the “properties” argument. The properties argument takes an array of @ServiceProperty annotations. The @ServiceProperty annotations requires a key and value to be set, both of type String.

```
@OSGiComponent(properties = {@ServiceProperty(key = "key", value = "value"), @ServiceProperty(key = "key2", value = "value2")})

public class ExampleComponent {}
```

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

Scopes

In CDI there is the notion of scopes, such as @RequestScoped and @ConversationScoped. Currently this is not possible to implement for OSGi services because OSGi services are always singleton. However RFC 195: Service Scopes introduces a way to create multiple instances of a servivce.. This RFC might make it possible to scope services in a similar way that CDI scopes work (e.g. session scoped). We are following this RFC and will try to use this to create tight integration with CDI scopes if possible.



The life-cycle of an @OSGiComponent is different from a normal CDI bean, because it has to deal with service dynamics. An @OSGiComponent may have dependencies on other OSGi services which can influence the life-cycle of the @OSGiComponent 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. An OSGi service however has a single instance, it is effectively a singleton. 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 @OSGiService 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 @OSGiService 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.

~~OSGi services are dynamic, they can come and go at any time during runtime. Although CDI doesn't support this model, OSGi-CDI has to deal with this. There are two different types of components with different levels of support for service dynamics:~~

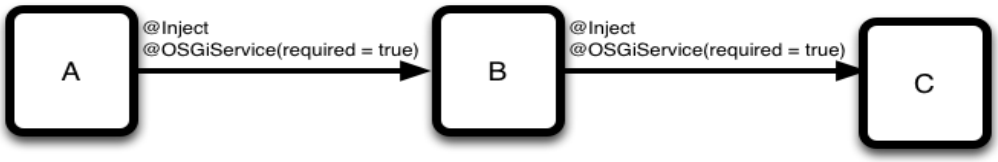
1. ~~Normal CDI beans~~
2. ~~@OSGiComponent beans~~

~~An @OSGiComponent bean is an OSGi service itself. Because of this we can support the notion of required dependencies. Required dependencies are not supported for normal CDI beans, because we can't influence their life-cycle. In the next section the different types of dependencies are described.~~

5.4.1 Required dependencies (@OSGiComponent only)

Require a dependency; the component will not be registered when a required dependency is not available. 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:



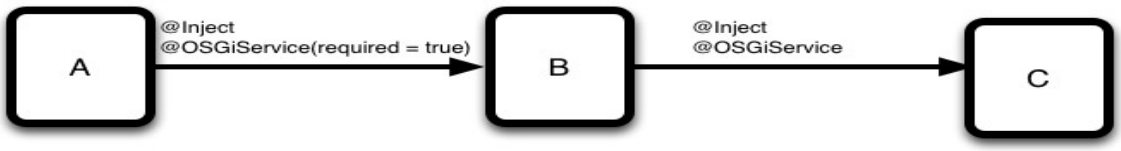
When C becomes unavailable, B will become unavailable as well, and so will A. When C becomes available again, so will be B, and so will A.

5.4.2 Optional dependencies

By default dependencies are optional. To deal with service dynamics a proxy is injected instead of the real reference to the service. The proxy is by default a null-~~proxy~~object, every method invocation should~~will~~ 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 @OSGiService 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 @OsgiService 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 @OSGiComponent annotation, required dependencies must be supported. An @OSGiComponent 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 @OSGiService qualifier. If the @OSGiServic qualifier is not used, the container should not query the service registry.

~~To prevent @OSGiComponents to be injected as normal CDI beans, they should not resolve injection points that don't specify the @OSGiService qualifier.~~ The @OSGi service

5.4.3 ~~Dependency injection~~Service and bundle registration ~~callbacks~~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 ~~callback~~observer method ~~can~~should be the type of the service ~~or a ServiceReference. Both parameters may be combined.~~

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 serviceAdded(@Observes @ServiceAdded ServiceReference<SomeService> ref) ;~~

~~void serviceAdded(@Observes @ServiceAdded SomeService, ServiceReference<SomeService> ref) ;~~

void serviceRemoved(@Observes @ServiceRemoved SomeService) ;

void serviceModified(@Observes @ServiceModified SomeService) ;

~~(@Observes @ServicRemoved ServiceReference<SomeService> ref) ;~~

~~Addedvoid service(@Observes @ServicRemoved SomeService, ServiceReference<SomeService> ref) ;~~

~~Addedvoid service~~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 @OSGiService described in the following section.

The definition of the @ServiceAdded and @ServiceRemoved annotations are below.

```
@Target(ElementType.PARAMETER)

@Retention(RetentionPolicy.RUNTIME)

@Qualifier

public @interface ServiceAdded {

   String filter() default "";

}

@Target(ElementType.PARAMETER)

@Retention(RetentionPolicy.RUNTIME)

@Qualifier

public @interface ServiceRemoved {

   String filter() default "";

}
```

5.4.4 Service Filters

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

```
@OSGiService("(somekey=somevalue)")
```

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

```
@Qualifier

@OSGiServiceFilter

public @interface SomeKey {

 String value();

}

@Inject @OSGiService @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

@OSGiServiceFilter

public @interface SomeKey {

 String value() default “somevalue”;

}

@Inject @OSGiService @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 @OSGiService @SomeKey(“somevalue”) @SomeOtherKey(“someothervalue”)

MyService service;
```

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

~~**An alternative to camel casing is using underscores. Each underscore is translated to a dot. For example the qualifier @my_service_property translates to (“my.service.property=example).**~~

Requiring configuration

An @OSGiComponent 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:

```
@OSGiComponent(requireConfiguration=”PID.of.configuration”)
```

~~**Component life-cycle callbacks**~~

~~An @OSGiComponent can define @PostConstruct and @PreDestroy methods. They are invoked when the component is activated or deactivated. This is NOT the same as bundle activation and deactivation; e.g. a bundle~~

~~can still be active while the component is deregistered because of unavailable required dependencies. The @PostConstruct method must be invoked by the container immediately after the component became available even if there are no injection points that require the component.~~

```
~~@OSGiComponent~~

~~public MyServiceImpl implements MyService {~~

~~  @PostConstruct~~

~~  public void start() {~~

~~  }~~

~~  @PreDestroy~~

~~  public void stop() {~~

~~  }~~

~~}~~
```

5.5 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.

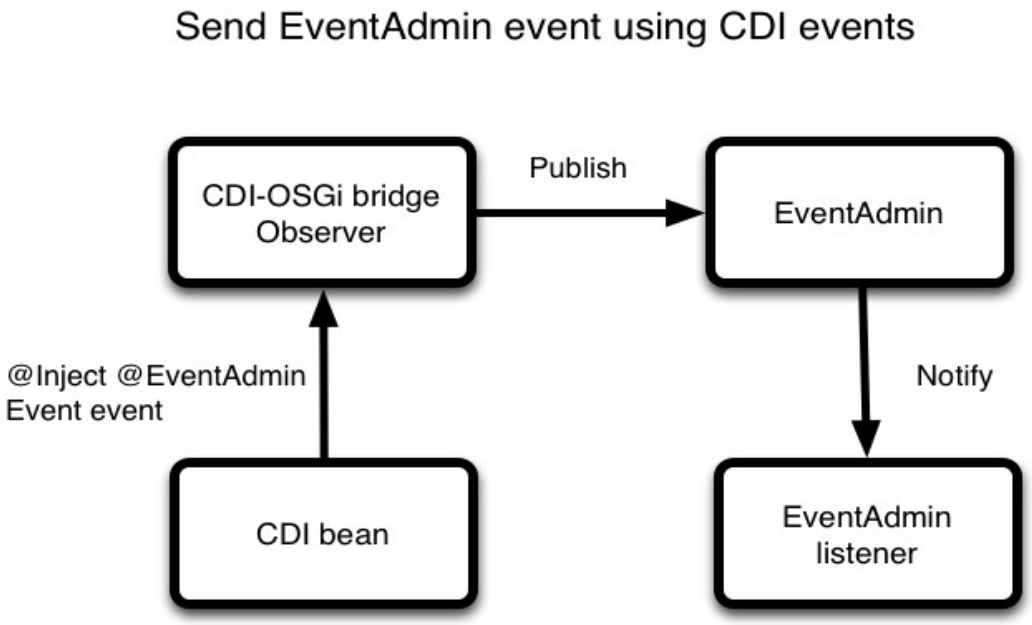
Bundle A

```
public abstract class MyBaseClass {

@Inject BundleContext bundleContext;

}
```

Bundle B

```
@OSGiComponent

public class MySubClass extends MyBaseClass {

}
```

5.6 EventAdmin integration



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.

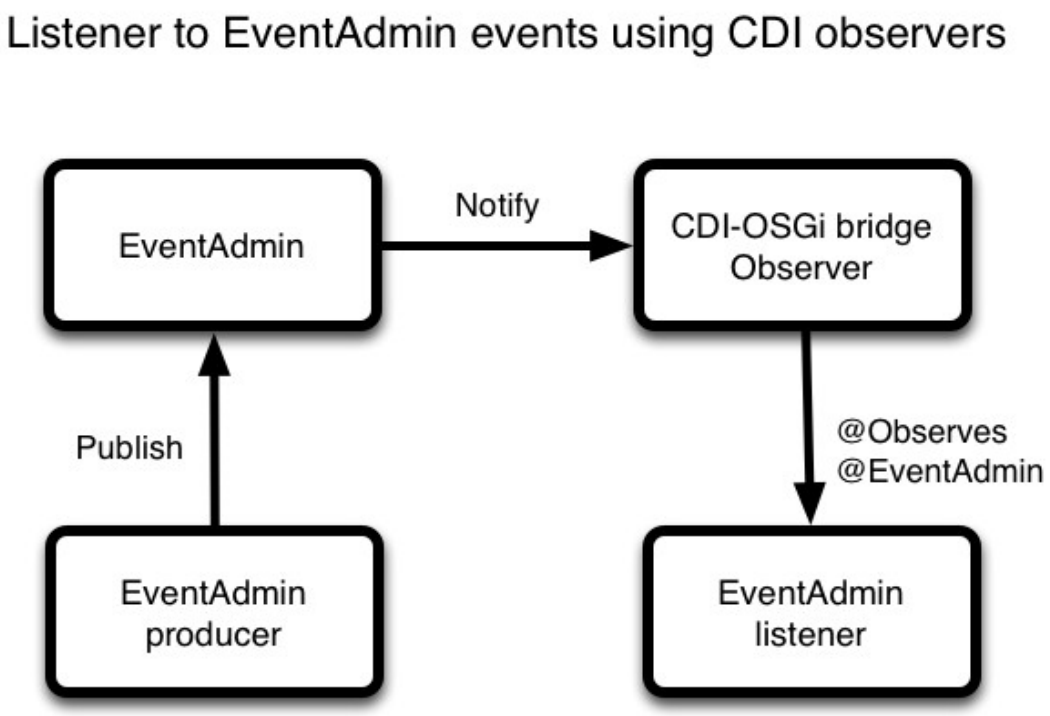
```
@Inject @EventAdmin("MyTopic") Event<MyEvent> event;

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) {}
}
```

~~Bootstrapping~~

~~In Java EE applications a JAR or WAR file should contain a bean.xml file in the meta-inf folder to enable CDI. Enabling CDI means that the archive is scanned for beans to add to the CDI container. An implementation of OSGI-CDI must enable CDI in the following two cases:~~

- ~~A bean.xml is present in the META-INF folder~~
- ~~The following manifest header is present: enable-cdi: 1.0~~

~~The version can optionally be used by the implementation to know which version of OSGi-CDI is expected by the bundle. This is useful to cover future backwards compatibility issues.~~

~~The container may optionally offer other bootstrapping mechanisms, but this is not required by the specification.~~

5.7 Annotations

In this section the annotations defined by this specification are listed. Besides these annotations no standardized types or constants are part of the specification.

```
package org.osgi.cdi;

import javax.inject.Qualifier;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;


@Retention(RetentionPolicy.RUNTIME)
```

```
@Target({ElementType.FIELD, ElementType.TYPE, ElementType.PARAMETER,
ElementType.CONSTRUCTOR})
@Qualifier
public @interface OSGiComponent {
    Class<?>[] interfaces() default {};
    ServiceProperty[] properties() default {};
}

package org.osgi.cdi;
import javax.inject.Qualifier;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.CONSTRUCTOR})
@Qualifier
public @interface OSGiService {
    boolean required() default false;
    String filter() default "";
}

package org.osgi.cdi;
import javax.inject.Qualifier;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
```

```
import java.lang.annotation.Target;



@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.TYPE, ElementType.PARAMETER,
ElementType.CONSTRUCTOR})
@Qualifier
public @interface OSGiServiceFilter {
}
package org.osgi.cdi;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
@Retention(RetentionPolicy.RUNTIME)
public @interface ServiceProperty {
   String key();
   String value();
}
```

6 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 Considered Alternatives

For posterity, record the design alternatives that were considered but rejected along with the reason for rejection. This is especially important for external/earlier solutions that were deemed not applicable.

8 Security Considerations

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

9 Document Support

9.1 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

9.2 Author's Address

Name	
Company	
Address	
Voice	
e-mail	

9.3 Acronyms and Abbreviations

9.4 End of Document