RFC 193 - CDI Integration

Draft

30 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

0.2 Trademarks

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 10.1.

`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	12-10-16	Initial version of the draf. David Bosschaert <david@redhat.com>
2nd draft	12-10-26	First draft of chapter 5 Paul Bakker <paul.bakker@luminis.eu>
3rd draft	12-10-30	Life-cycle callbacks Paul Bakker <paul.bakker@luminis.eu>
4th draft	13-01-27	Details on service dynamics Paul Bakker <paul.bakker@luminis.eu>
5th draft	13-02-27	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	13-11-07	David Bosschaert, process input from call 6th November 2013.
10th draft	13-11-14	David Bosschaert, process input from the virtual F2F November 13, 2013.
11th draft	14-11-05	Emily Jiang, update to CDI 1.2
12th draft	14-11-20	Emily Jiang more update based on the comments from F2F in Los Angeles
13th draft	15-01-17	Emily Jiang. Add lifecycle and Event
14th draft	15-02-15	Emily Jiang, update based on the call 11th March 2015
15th draft	15-09-09	Emily Jiang, updated based on the call in June F2F
16th draft	15-12-18	Emily Jiang, updated based on F2F in November
17th draft	16-11-17	Raymond Augé, based on implementation experience
18th draft	17-03-24	Raymond Augé, based on feedback from Montpellier F2F

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://weld.cdi-spec.org/ 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 argument 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 [3].Error: Reference source not found and JSR 346 [4].

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 [5].

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 {
    @Override
    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 JavaEE, 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 JavaEE 6 and above. Furthermore, it introduces powerful extensibility into the JavaEE 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 JavaEE 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 when published as an OSGi service.

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 dependencies of the CDI container.

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 of the CDI container to support the dynamic life-cycle provided by OSGi services. For example, it MUST NOT be fatal to deploy a CDI container that does not have all its service dependencies initially satisfied and it MUST be possible to change service dependencies without requiring the CDI bundle to be redeployed or restarted.

CDI018 – The specification MUST provide a mechanism to consume multiple matching OSGi services of a given type in an injection point. For example via the @Inject Instance<T> mechanism. Service optionality should result from an empty number of matching services.

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 deactivation 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.

CDI034a – The specification MUST comply with CDI specification bean defining annotation for honoring a CDI bean not requiring the existence of beans.xml.

CDI034b – 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.

CDI036 – The specification SHOULD define a means of interacting with configuration admin.

CDI037 – The solution must perform bean discovery at tool time. Processing of beans is done at runtime.

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.

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

4.3 Requirements from RFP 98 (OSGi/JavaEE 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 – An OSGi bundle containing CDI metadata and beans.
- 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. This entity is implementation dependent and may or may not be separate from the CDI OSGi adapter.
- CDI Portable Extension – A portable extension as defined in CDI 1.2.
- CDI Portable Extension Bundle – A bundle providing one or more CDI portable extensions. An extension bundle may or may not be a CDI bundle at the same time.

5.2 Extender Capability and Opt-In

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. For these reasons, this specification defines an `osgi.extender` capability called `osgi.cdi`.

A CDI Bundle MUST require the OSGi CDI extender capability, e.g.

```
Require-Capability:\
  osgi.extender;\
    filter:="(&(osgi.extender=osgi.cdi)(version>=1.0)(!(version>=2.0)))"
```

A CDI Bundle must declare the location of any CDI bean descriptors to read from the bundle classpath using the attribute `beans` on the requirement, which is a `List<String>` of file paths, e.g.

```
Require-Capability:\
  osgi.extender;\
    filter:="(&(osgi.extender=osgi.cdi)(version>=1.0)(!(version>=2.0)))";\
    beans:List<String>="META-INF/beans.xml,WEB-INF/beans.xml"
```

If there are no beans files to declare the `beans` attribute is not required.

It may also declare an attribute `osgi.`~~-~~`beans` on the `osgi.cdi` extender requirement who's type is `List<String>` of file paths and who's values ~~is~~ are~~n~~ XML documents that contains ~~osgi~~OSGi~~-specific~~ ~~b~~Bean descriptor information, e.g.

```
Require-Capability:\
  osgi.extender;\
    filter:="(&(osgi.extender=osgi.cdi)(version>=1.0)(!(version>=2.0)))";\
    osgi.beans:List<String>="OSGI-INF/cdi/osgi-beans.xml"
```

If not specified, the location `OSGI-INF/cdi/`~~`osgi-beans`~~`*.xml` will be used.

The elements in the osgi beans documents are defined in `cdi.xsd` in this RFC folder.

The bean classes may be filtered based on the declared `CDI beans descriptors`.

If `foo.A` is excluded by `beans.xml`, `foo.A` will not be considered. If there **are no CDI bean descriptors**, all the beans defined in the OSGi Beans Descriptor will be considered, e.g.

```
<?xml version="1.0" encoding="UTF-8"?>
<cdi:beans xmlns:cdi="http://www.osgi.org/xmlns/cdi/v1.0.0">
  <cdi:bean class="foo.FooImpl" />
  <cdi:bean class="foo.FumImpl" />
```



```
  <cdi:bean class="baz.Baz" />
</cdi:beans>
```

Note that the beans.xml's bean-discovery-mode mode is ignored in favor of populating the OSGi Beans Descriptor with the complete list of bean class names to be considered at runtime. This could be implemented in build tooling.

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:Version=1.0
```

Future versions of this specification will require higher version numbers.

5.3 CDI Portable Extensions

A CDI portable extension is a mechanism by which a third party may integrate with the CDI container by:

- Providing its own beans, interceptors and decorators to the container
- Injecting dependencies into its own objects using the dependency injection service
- Providing a context implementation for a custom scope
- Augmenting or overriding the annotation-based metadata with metadata from some other source

Use of the API associated with an extension may not be sufficient to express the requirement for the extension's implementation and perhaps an extension does not provide an API at all. Capabilities are useful to express such loose dependencies. Therefore a capability namespace osgi.cdi.extension is defined for use by CDI Portable Extensions and CDI Bundles. 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.

CDI Portable Extension Bundles MUST therefore provide an osgi.cdi.extension capability with an attribute osgi.cdi.extension of type String holding a distinctive name for the extension. It must also define a version attribute of type Version, e.g.

```
Provide-Capability:\
  osgi.cdi.extension; osgi.cdi.extension=frobnicator; version:Version=1.0
```

The extension name osgi.cdi is reserved by this specification in case a CDI OSGi adapter chooses to implement it's functionality using the extension model.

Due to the dynamic nature of the OSGi runtime, it's essential that a CDI Extender be able to react to the coming and going of CDI Portable Extension bundles. The CDI Extender must alter the state of CDI Containers who depend on these extensions. The most reliable way to address this is via the service registry. For this reason CDI Portable Extensions must be published into the OSGi service registry as services under the interface javax.enterprise.inject.spi.Extension. As well, the extension service must include the property osgi.cdi.extension containing the name of the extension. This name should be the same as that used in the capability, e.g.

```
@Component(property = {CdiExtenderConstants.CDI_EXTENSION + "=frobnicator"})
public class Frobnicator implements Extension {}
```

Extension implementations have been typically instantiated using Class.forName() per application classloader. As such, it's likely that Extensions would not support the multi-bundle nature of OSGi. In this case the extension should be published as a ServiceFactory. The simplest way to achieve this is using the DS @Component annotation with @Component.scope = ServiceScope.BUNDLE, e.g.

```
@Component(
```

```
    property = {CdiExtenderConstants.CDI_EXTENSION + "=frobnicator"},
    scope = ServiceScope.BUNDLE
)
public class Frobnicator implements Extension {}
```

A CDI Bundle may depend on CDI portable extensions. To express this requirement the CDI Bundle MUST declare the requirement using the **osgi.cdi.extension** namespace with a filter matching the CDI extension's Capability attributes, e.g.

```
Require-Capability:\
  osgi.cdi.extension;\
    filter:="(&(osgi.cdi.extension=frobnicator)(version>=1.0)(!(version>=2.0)))"
```

5.4 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.

5.4.1 Initialization of a CDI Container

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

- The CDI Bundle state
- The CDI extender's bundle state

All activities on behalf of the CDI bundle must use the BundleContext of the CDI bundle. All dynamic class loads must use the CDI bundle's `Bundle.loadClass` method.

5.4.1.1 Initialization steps

The following is a description of the initialization steps. The CDI Container will update its state. State changes are broadcast as events [5.13.1]

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

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. The CDI container service is published with the service property `osgi.cdi.container.state = CREATING` and the `Creating` CDI event is fired
3. If the CDI bundle depends on any extensions, the CDI container's `osgi.cdi.container.state` property is updated to `WAITING_FOR_EXTENSIONS`, the `WaitingForExtensions` CDI event is fired and the CDI container waits for extensions to be resolved (this MUST not be a busy, blocking wait).
4. The CDI bundle's beans are processed and during processing `@org.osgi.service.cdi.annotations.Configuration`, `@org.osgi.service.cdi.annotations.Reference` and `@org.osgi.service.cdi.annotations.Service` annotations are collected.
5. If `@org.osgi.service.cdi.annotations.Configuration` annotations were found, or if the OSGi Beans Description contains any configuration elements the CDI container's `osgi.cdi.container.state` property is updated to `WAITING_FOR_CONFIGURATIONS`, the `WaitingForConfigurations` CDI event is fired and the CDI container waits for the referenced configurations to be resolved (this MUST not be a busy, blocking wait).

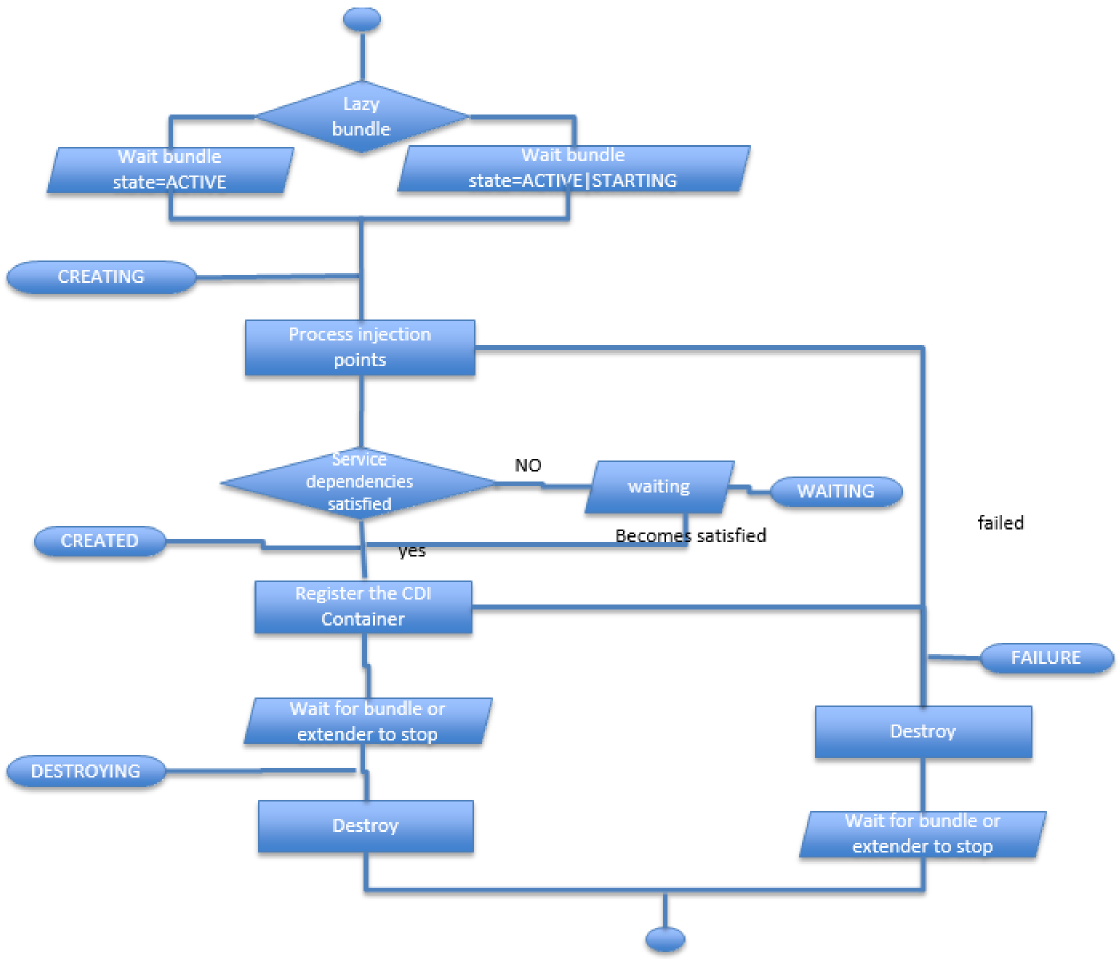
6. If `@org.osgi.service.cdi.annotations.Reference` annotations were found, or if the OSGi Beans Description contains any reference elements the CDI container's `osgi.cdi.container.state` property is updated to `WAITING_FOR_SERVICES`, the `WaitingForServices` CDI event is fired and the CDI container waits for referenced services to be resolved (this MUST not be a busy, blocking wait).
7. The CDI container's `osgi.cdi.container.state` property is updated to `SATISFIED` and the `Satisfied` CDI event is fired.
8. If `@org.osgi.service.cdi.annotations.Service` annotations were found the beans having the annotations are published into the OSGi service registry using whatever service properties accompanied the annotation and under the correct OSGi service scope depending on their CDI scope.
9. The CDI container's `javax.enterprise.inject.spi.BeanManager` service is published into the OSGi service registry. The CDI container's `osgi.cdi.container.state` property is updated to `CREATED` and the `Created` CDI event is fired.
10. The CDI beans are now active and perform their function until the CDI bundle or the CDI extender bundle are stopped.
11. When the CDI bundle is stopped OR the CDI extender bundle is stopped, the CDI container's `osgi.cdi.container.state` property is updated to `DESTROYING` and the `Destroying` CDI event is fired.
12. The CDI container's published bean services are unpublished.
13. The CDI container's service dependencies are released.
14. The CDI container's extensions are released.
15. The CDI container's `osgi.cdi.container.state` property is updated to `DESTROYED` and the `Destroyed` CDI event is fired.
16. The CDI container is unpublished from the service registry.
17. At any time between the `CREATED` and `DESTROYING` state if any dependent service goes away, the container must return to the `WAITING_FOR_SERVICES` state making sure to unpublish it's services and destroy it's beans.
18. The moment service dependencies are again satisfied the CDI container may proceed through initialization as described above.
19. At any time between the `CREATED` and `DESTROYING` state if any dependent extension goes away, the container must return to the `WAITING_FOR_EXTENSIONS` state making sure to unpublish it's services and destroy it's beans.
20. The moment extension dependencies are again satisfied the CDI container may proceed through initialization as described above.

5.4.1.2 Failure

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

1. The CDI container's `osgi.cdi.container.state` property is updated to `FAILURE` and the `Failure` CDI event is fired.
2. Unregister the CDI container service
3. Destroy the CDI container
4. Wait for the CDI bundle to be stopped

5.4.1.3 Diagram



5.4.1.4 CDI container service

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

`osgi.cdi.container.state` – (`Cdi~~Container.State~~Event.Type` enum value) The state of the CDI container

5.4.1.5 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:

- The CDI container's published bean services are unpublished.
- The CDI container's service dependencies are released.
- The CDI container's extensions are released.
- Destroying all beans
- Unregistering the CDI Container service

A CDI Container must continue to follow the destruction even when 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 shutdown of many CDI containers, which the CDI extender should handle, MUST be safe to perform serially and without deadlocks as the cascade effect of having interdependent services should cause CDI containers not yet processed to safely degrade into one of the `WAITING_FOR_*` states making it safe to destroy when their turn arrives to be processed.

5.5 Publishing CDI Beans as OSGi Services

A bean can be published to the OSGi service registry using the `@org.osgi.service.cdi.annotations.Service` annotation, e.g.

```
@Service
public class ExampleComponent {}
```

Note that classes that are already CDI beans, are not automatically published to the service registry; the `@org.osgi.service.cdi.annotations.Service` annotation or metadata as described in section 5.5.3 is always required for this. The requirement to explicitly declare 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 uses the classes declared in the OSGi Beans Description 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 fine 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.

OSGi services are published specifying the type under which they are available. In order to support this model the `@org.osgi.service.cdi.annotations.Service` annotation provides a `type` property of type `Class<?>[]`, e.g.

```
@Service(type=A.class)
public class MyBean implements A,B {}
```

The `type` property 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 directly implement any interfaces it will be registered under its implementation class.

5.5.1 Service Properties

Service properties can be defined on the `@org.osgi.service.cdi.annotations.Service` annotation using the `property` elements~~ies argument~~, **e.g.**

```
~~@Service(properties={~~
~~    @ServiceProperty(key = “foo”, value = “bar”),~~
~~    @ServiceProperty(~~
~~        key = “timestamp”, value = “1480706423”, type = PropertyType.Long)~~
~~})~~
@Service(
    property = {
        “key=bar”,
        “timestamp:Long=1480706423”
    }
)
public class MyBean {}
```

The `property` elements~~ies argument~~ **takes an array of** Strings. ~~`@org.osgi.service.cdi.annotations.ServiceProperty` annotations. The `@org.osgi.service.cdi.annotations.ServiceProperty` annotation 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 using the `PropertyType` enum.~~

5.5.2 CDI Qualifiers as Service Properties

All qualifiers will automatically be included as service properties. The mapping will follow the default behaviour of the converter spec in converting Annotations to Map<String, Object>. Collisions will be handled as last wins. Finally, properties set in the @Service will take highest precedence (be processed last).

5.5.3 Publishing plain CDI beans in the Service Registry

To support integration of existing CDI beans with OSGi, an alternative to the `@org.osgi.service.cdi.annotations.Service` annotation is available via the OSGi Beans Descriptor. This allows beans that act as providers in plain CDI environments to be exposed as services without having to change and recompile them. A bean can be declared as a service in the descriptor by specifying the service child element of the bean element, e.g.

```
<?xml version="1.0" encoding="UTF-8"?>
<~~cdi:~~beans xmlns:cdi="http://www.osgi.org/xmlns/cdi/v1.0.0">
  <cdi:bean class=”foo.FooImpl”>
    <service>
      <provide interface=”foo.Foo” />
    </service>
  </cdi:bean>
</~~cdi:~~beans>
```

It’s also possible to provide properties this way, e.g.

```
<?xml version="1.0" encoding="UTF-8"?>
<~~cdi:~~beans xmlns:cdi="http://www.osgi.org/xmlns/cdi/v1.0.0">
  <cdi:bean class=”foo.FooImpl”>
    <service>
      <property key=”somekey” value=”somevalue” type=”String” />
      <property key=”someotherkey” type=”Integer”>
         <!-- An array of integers -->
         23
         6578
```

```
      </property>
      <provide interface="foo.Foo" />
    </service>
  </cdi:bean>
</cdi:beans>
```

CDI qualifiers on the bean will automatically be added as service properties as described in 5.5.2.

5.6 Scopes for beans annotated as @Service

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

`@ApplicationScoped`	service.scope = bundle (ServiceFactory)
`@Singleton`	service.scope = singleton
`all others`	service.scope = prototype (PrototypeServiceFactory)

5.7 Injecting OSGi Services into beans

CDI beans using the standard `@Inject` annotation may declare that the intended injection target is an OSGi service. By default, a CDI bean has one of a variety of different scopes which means it's lifecycle is bound by the Context in which that scope is defined. In order to ensure that the OSGi services are adequately handled to suite the scope of the bean having the service reference the OSGi `ServiceObjects` API should be used.

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. Natural ordering of OSGi ServiceReferences will be used if multiple candidates are found that match the reference.

Beans must always use `@org.osgi.service.cdi.annotations.Reference` annotation 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 `@org.osgi.service.cdi.annotations.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, e.g..

```
class Fum {
  @Inject
  @Reference
  Foo foo;
}
```

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.

It's also possible using the `@Reference.scope` property to specify the scope of the service to be injected, e.g.

```
class Fum {
  @Inject
  @Reference(scope = ReferenceScope.SINGLETON)
  Foo foo;
}
```

When a ServiceReference or service properties are required, the `@Reference.service` can declare the type of service when it can't be discovered by the type of Injection point, e.g.

```
class Fum {
  @Inject
```

```
  @Reference(service = Foo.class)
  ServiceReference<?> sr;
}
```

5.7.1 Allowing Service Injections in vanilla CDI injection points

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.

Support for this ability is provided through the OSGi Beans Description using the reference element. References declare a target filter which allows matching against OSGi services. These references also define which specific class under which to make the bean available. The service should be cast-able to the specified type. The service will then be instantiated as a bean. These "reference" beans can then match any injection point as the CDI container sees fit, e.g.

~~Given the bean:~~

```
class Fum {
  @Inject
  Foo foo;
}
```

The following reference would define a reference bean that would match the injection point:

```
<?xml version="1.0" encoding="UTF-8"?>
<cdi:beans xmlns:cdi="http://www.osgi.org/xmlns/cdi/v1.0.0">
  <cdi:reference beanClass="foo.Foo" target="(objectClass=foo.Foo)" />
</cdi:beans>
```

5.7.2 Service Filters

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

```
@Reference(target = "(some.key=somevalue)")
```

As a convenience CDI Qualifiers will be converted to filters ~~alternatively the same could be done using CDI Qualifiers~~:

```
@Qualifier
@ReferenceFilter
public @interface SomeKey {
  String somekeyvalue();
}

@Inject
@Reference
@SomeKey(somekey = "somevalue")
MyService service;
```

Conversion of Qualifiers to filters will map to the behavior ~~of the~~specified by the converter specification in converting Annotations to Map<String, String> along with name mangling rules.

Note: Marker annotations (having no members) convert to an empty map and so those are effectively ignored.

Multiple qualifiers will be combined to create AND filters. In the following example both annotations are qualifiers, and the resulting filter is `"(&(foo>=10)(some.key=somevalue)(some.other.key=someothervalue))"`.

```
@Inject
@Reference(target = "(foo>=10)")
@SomeKey(~~somkey = ~~"somevalue")
@SomeOtherKey(~~someotherkey = ~~"someothervalue")
MyService service;
```

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

5.8 Multi-cardinality – Supporting N matching OSGi Services

In OSGi it's quite possible for several services to match some criteria. This is such a common case that several good low level mechanism exist to deal with the various use cases surrounding the possibility of having multiple matches.

CDI also provides a mechanism for accessing multiple beans with a few limitations. This is known as **contextual instance by programmatic lookup**. This is implemented by means of the interface `javax.enterprise.inject.Instance`. This interface allows an injection point to be created which does not bind a bean tightly to a specific injection but rather to be able to perform dynamic lookup or to iterate over matches, e.g.

```
@Inject
Instance<Foo> foos;

Foo foo = foos.get();
```

However, unlike the default behaviour of `Instance`'s inherited `javax.inject.Provider.get()` method, `Instance` implementations backed by services must not throw `AmbiguousResolutionException` when multiple matches exist and should return the best candidate determined by reverse natural ordering over the `ServiceReference` of the matching services. The method `isAmbiguous()` should still return `true` however. The iterator returned should also preserve reverse natural ordering over the `ServiceReference` of the matching services.

5.8.1 Explicit/Minimum Cardinality

In order to preserve the typical behavior of `Instance`, the default cardinality for the dependency created by it's use is ~~1 which will block the CDI container from resolving if no match is found~~0 indicating that the reference is optional. However, it may be convenient to support adjusting this cardinality in order to support ~~use cases like optional dependencies as well as~~ higher order minimum cardinality.

To support this the `@MinCardinality` annotation can be used to specify the lower bound, e.g.

An optional dependency could be declared as:

```
@Inject
~~)0@MinCardinality(~~
@Reference
Instance<Foo> foos;
```

A case where at least 3 are required could be expressed as:

```
@Inject
@MinCardinality(3)
@Reference
Instance<Foo> foos;
```

Negative values are ignored and treated as the default (which is 0~~1~~).

5.9 Requiring configuration

Beans can require Configuration Admin Configurations through @Inject with the qualifier `@Configuration`. The CDI container will only become available when matching configurations are found. This is useful when a bean does not have usable default configuration values. If the configuration object is not available, the behavior is the same as for a unavailable required service dependency. `@Configuration` is supported on all types which could be emitted by the converter specification given a `Configuration` object. The `@Configuration` annotation accepts an array of String values listing the expected PIDs. The `$` symbol is a placefolder for the default PID which is the name of the bean class by default. This is to prevent having to duplicate the name of the PID.

```
@Inject
@Configuration({"$", "other.pid", "some.other.pid"})
Map<String, Object> props;
```

Type safety is supported by using configuration types:

```
@interface MyConfig {
  int some_property() default 0;
  String some_other_property() default "foo";
}
@Inject
@Configuration({"$", "other.pid", "some.other.pid"})
MyConfig config;
```

5.9.1 Allowing Configuration Injections in vanilla CDI injection points

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

Support for this ability is provided through the OSGi Beans Description using the configuration element. Configurations declare a pid attribute which allows matching against the `service.pid` property of Configuration Admin configurations. These configurations also define which specific class under which to make the bean available. The class should be one which the CDI bundle can load and to which the converter specification describes a conversion from the configuration Dictionary. The configuration dictionary will then be instantiated as a bean of this type. These configuration beans can then match any injection point as the CDI container sees fit, e.g.

```
class Foo {
  @Inject
  org.example.Config config;
}
```

The following configuration would define a configuration bean that would match the injection point:

```
<?xml version="1.0" encoding="UTF-8"?>
<cdi:configuration beanClass="org.example.Config" pid="some.pid"
  xmlns:cdi="http://www.osgi.org/xmlns/cdi/v1.0.0" />
```


5.10 BundleContext injection

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

```
@Inject BundleContext context;
```

This will always inject the BundleContext of the bundle that contains the CDI container.

5.11 `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.

5.12 `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.

5.13 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.

5.13.1 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_FOR_CONFIGURATIONS - The CDI container is waiting for dependent configurations.
- WAITING_FOR_EXTENSIONS – The CDI container is waiting for dependent extensions.
- WAITING_FOR_SERVICES – The CDI container is waiting for dependent services.
- SATISFIED – The CDI container has all it's dependencies satisfied and is completing initialization.
- 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 dependencies being waited for.
- getExtenderBundle() - The CDI extender bundle.
- getTimestamp() - The time the event occurred
- getType() - The type of the event.

It should be noted that while CDI Events are published onto the CDI container's event bus, not all events are visible to it since some are fired before and after the CDI container's beans are instantiated. However, CDI Listeners see all events (provided they are registered early enough).

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.

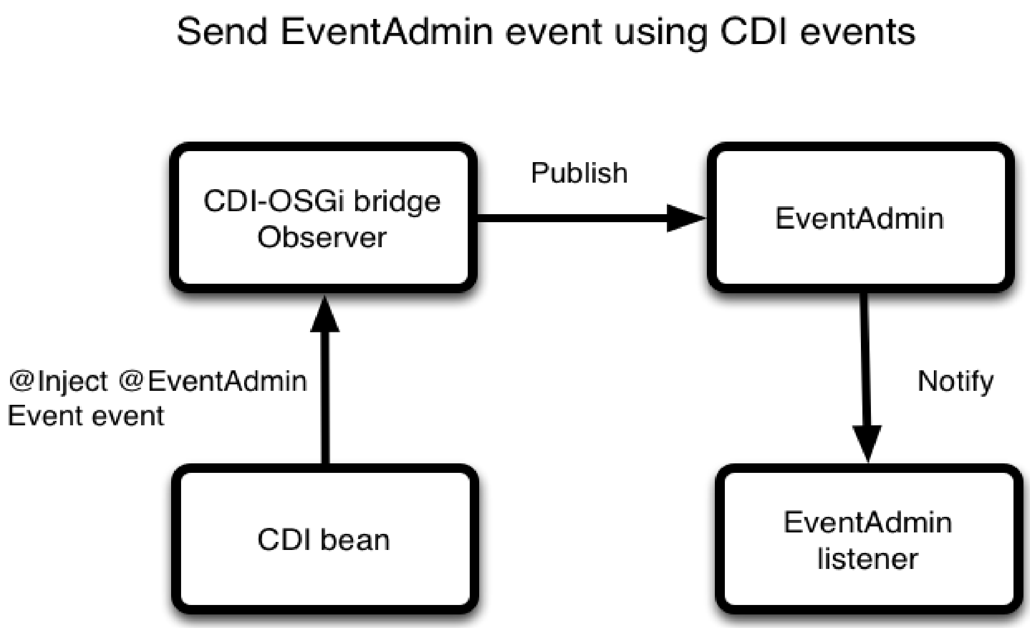
This section is optional and could also be provided in a separate RFC.

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.

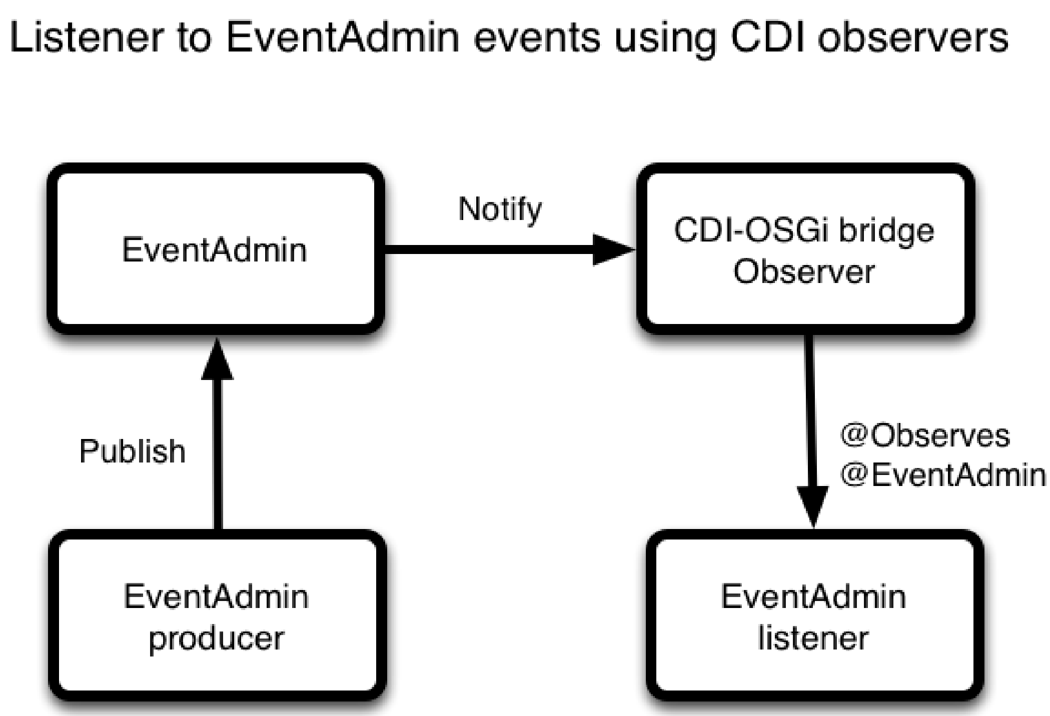
```
@Inject @EventAdmin("MyTopic") Event<MyEvent> event;

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();
}
```

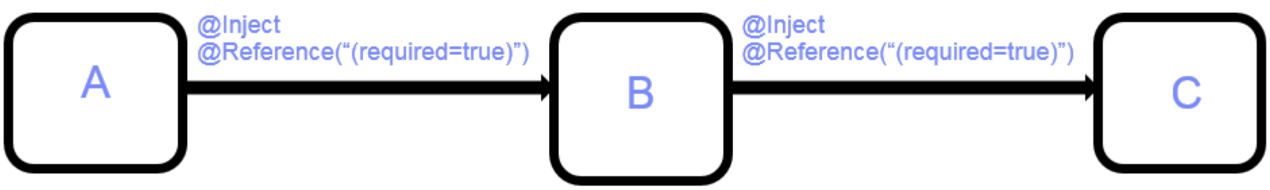
```
public class EventExample {
  public void process(@Observes @Demo MyEvent event) {}
}
```

8.2 Required dependencies (@Service only)

This was moved to considered alternatives because all References will be required by default.

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.

8.3 Additional OSGi CDI Scopes

The additional OSGi CDI scopes was moved to considered alternatives because it was deamed acceptable that the OSGi service scopes of singleton, bundle, and prototype could be mapped to @Singleton, @ApplicationScoped, and @Dependent.

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`	

8.4 Service and bundle registration observers

This was moved to considered alternatives due to the removal of the dynamic service update requirements.

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)
```

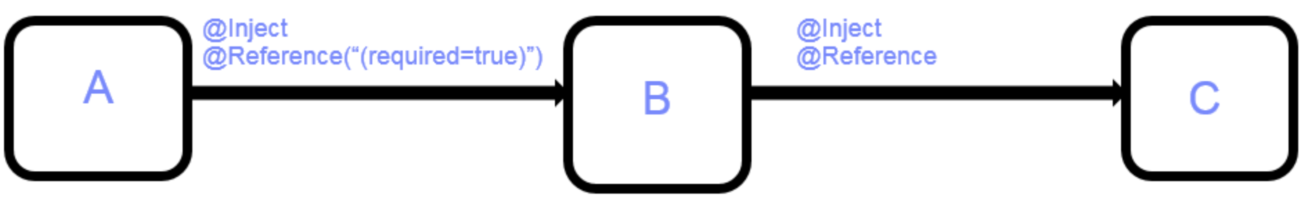
The @ServiceAdded and @ServiceRemoved annotations have the same service filtering semantics as @Reference described in the following section.

8.5 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 @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 @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 @Service annotation, required dependencies must be supported. An @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 @Reference qualifier. If the @Reference qualifier is not used, the container should not query the service registry. @Reference can be used on fields or method parameters.

8.5.1 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.

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

[3]. JSR 299: Contexts and Dependency Injection for Java EE flatform

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

[5]. 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	Emily Jiang
Company	IBM
Address	Hursley Lab, UK, IBM
Voice	
e-mail	emijiang@uk.ibm.com

10.3 Acronyms and Abbreviations

10.4 End of Document