CDI OSGi integration

Design Specification

Mathieu Ancelin

`<mathieu.ancelin@serli.com>`

Matthieu Clochard

`<matthieu.clochard@serli.com>`

Kevin Pollet

`<kevin.pollet@serli.com>`

Preface

1.1. About naming and references

1.1.1. Bean archive

A bean archive is a java archive, such as a jar or a Java EE module, that contains a special marker file:`META-INF/bean.xml`.

A bean archive may be deployed in a CDI environment. It enables all CDI features in that bean archive.

1.1.2. OSGi bundle

A bundle is a Java archive, such as a jar or a folder, that contains some special OSGi marker headers in its `META-INF/Manifest.MF`.

A bundle may be deployed in an OSGi environment. It enables all OSGi features for that bundle.

1.1.3. Service, implementation, instance and registration

A service is mostly an interface. This interface defines the contract that describes what the service may do. It might be several way to actually providing the service, thus a service might have multiple implementations.

A service implementation is a class that implements this service. It is what is available to other components that use the service. To use the service the component obtain an instance of the implementation.

A service instance is an instance of one of the service implementations. It is what the user manipulates to perform the service.

A registration is the object that represents a service registered with a particular implementation. Then this implementation can be searched and its instances can be obtained. Every time a service implementation his register a corresponding registration object is created.

1.1.4. References

This document uses both CDI and OSGi specification documentations as technical references. You may refer to these documents for a better understanding of CDI and OSGi functionality, references and naming conventions.

CDI-OSGi comes with other documentations and you may refer to CDI-OSGi JavaDoc and CDI-OSGi user manual for a better understanding of CDI-OSGi.

1.2. What are CDI-OSGi and Weld-OSGi ?

1.2.1. CDI-OSGi

CDI-OSGi aims at simplifying application development in an OSGi environment by providing a more modern, more user-friendly and faster way to interact with the OSGi Framework.

It addresses the OSGi complexity about services management using CDI specification (JSR-299). Thus it provides a CDI OSGi extension with injection utilities for the OSGi environment. An integration of any CDI implementation, such as Weld, it is used. This integration is possible through a well-defined bootstrapping API.

CDI-OSGi is a framework that may be used in an OSGi environment and composed by five bundles.

1.2.2. Weld-OSGi

Weld-OSGi is an integration of Weld in the OSGi environment using CDI-OSGi. It is the exhibit implementation of features exposes by CDI-OSGi APIs.

Weld-OSGi is one of the five bundles composing CDI-OSGi. But it also names the framework CDI-OSGi using this particular CDI integration.

1.3. Third party dependencies and environment

CDI-OSGi may run into an OSGi environment, therefore it requires an OSGi implementation framework to run in (such as Apache Felix, Equinox, Knopflerfish ...).

CDI-OSGi also depends on a CDI implementation (such as Weld, Open Web Beans, CanDI ..).

CDI-OSGi logs its operations using the SLF4J logging facade and the LogBack logging implementation.

1.4. What about other frameworks

CDI-OSGi stays compliant with CDI specification and uses only standard OSGi mechanisms. Every things it does (or nothing from it) CDI and OGSi can do.

Thereby it is compatible with most of the current frameworks dealing with OSGi service management.

Weld-OSGi has the same compatibility since it is an implementation of CDI-OSGi. But as it provides some additional features it may be impossible to use all its possibilities coupling with other frameworks.

1.5. Organization of this document

Since this specification covers two different (but linked) pieces of software it is separated into two majors parts :

- The CDI-OSGi specifications for core functionality usages and CDI container integration.
- The Weld-OSGi specifications for Weld integration into CDI-OSGi.

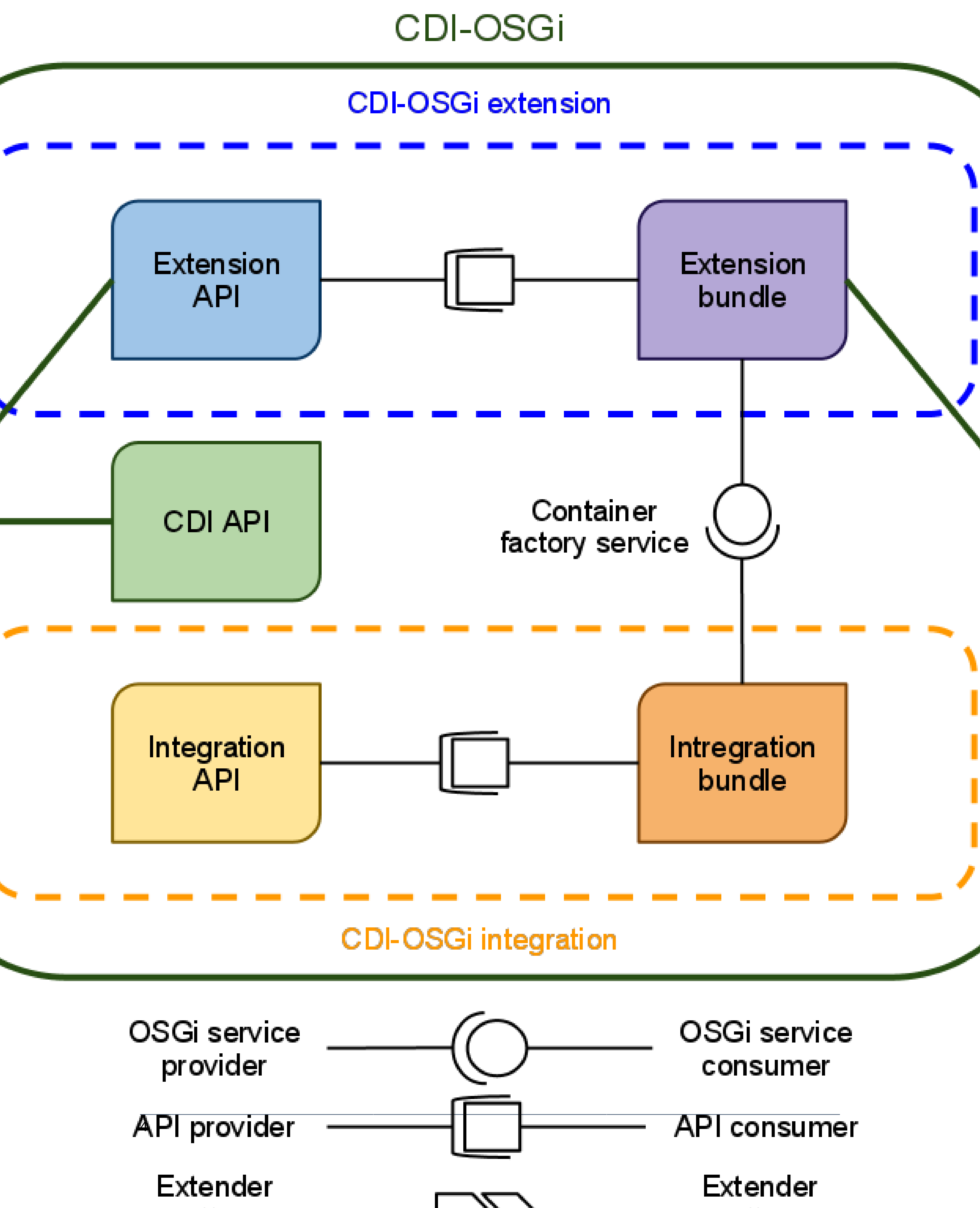
Organization of CDI-OSGi

2.1. Bundles and interactions

CDI-OSGi is composed of five bundles:

- The extension API that describes all the new features provided by CDI-OSGi in the OSGi environment.
- The integration API that allows CDI compliant container to be used with CDI-OSGi.
- The CDI API that describes all the regular CDI features provided by CDI-OSGi to bean bundles.
- The extension bundle that provides CDI-OSGi features for bean bundles by managing them,
- An integration bundle that provides CDI features usable by the extension bundle through an OSGi service.

Note that as CDI-OSGi runs in an OSGi environment it is implicit that there is an OSGi core bundle too. This one provide OSGi features for all other bundles, including CDI-OSGi managed bundles. But it is not an actual part of CDI-OSGi.


CDI-OSGi
CDI-OSGi extension
Extension API
Extension bundle
CDI API
Container factory service
Integration API
Intregration bundle
CDI-OSGi integration
OSGi service provider
OSGi service consumer
API provider
API consumer
Extender
Extender

This figure shows the five bundles of CDI-OSGi and the links between them.

Figure 2.1. The five bundles of CDI-OSGi

These bundles could regroup in three part (as shown in the figure above):

- CDI-OSGi extension: The blue part represents the CDI OSGi extension. It is composed of one API bundle and its implementation (the extension bundle). It is the core of CDI-OSGi that manages all bean bundles.Thus the extension API bundle exposes the CDI-OSGi features and the extension bundle enables these features. All interactions with client bundles go through the CDI-OSGi extension part.
- CDI-OSGi integration: The yellow part represents the CDI OSGi integration. It is composed of one API bundle and its implementation. It is how CDI features are provided to CDI-OSGi.Thus the integration API bundle exposes the requirements of CDI-OSGi in order to run CDI features in OSGi environment.The integration bundle is the implementation of these requirements using a vendor specific CDI implementation (such as Weld).Weld-OSGi is one of the possible extension bundle. So the extension bundle is commutable to support various CDI implementation.
- CDI API bundle: The fifth bundle is the CDI API. It exposes regular CDI features for all client bundles and exempts the user to load CDI API by himself. It is a third-party API provided for convenience to the user.

User client bean bundles should only know about the extension API bundle of CDI-OSGi and the CDI API bundle because they may import their packages in order to use CDI-OSGi features. They do not need to know the other three bundles.

The extension bundle manages bean bundles transparently. It also implements the extension API and uses the container factory service from the integration bundle.

Integration API bundle should only be known by users who want to provide an alternative integration bundle. This latter provides the CDI compliant containers used by the extension bundle. The CDI-OSGi integration part is only used internally.

2.2. Descriptions

2.2.1. Extension API

The extension API defines all the features provided to OSGi environment using CDI specification. It exposes all the new utilities and defines the comportment of the extension bundle.

It exposes all the interfaces, events and annotations usable by a developers in order to develop its client bean bundles. It defines the programming model of CDI-OSGi client bundle. Mostly it is about publishing and consuming injectable services in a CDI way.

It also describes the object the extension bundle needs to orchestrate bean bundles.

So this is where to search for new usages of OSGi.

2.2.2. Integration API

The integration API defines how a CDI container, such as Weld, should bootstrap with the CDI OSGi extension. So any CDI environment implementation could use the CDI OSGi extension transparently. The CDI compliant container may be provided using an implementation bundle.

This aims at providing the minimum integration in order to start a CDI compliant container with every managed bean bundle. Then the extension bundle can get a CDI container to provide to every one of its manages bean bundle.

Moreover the integration API allows to mix CDI compliant container in the same application by providing an embedded mode. In this mode a bean bundle is decoupled from the extension bundle and is managed on its own. Thus various implementations of CDI container can be used or the behavior of a particular bean bundle can be particularized.

All this bootstrapping mechanism works using the service layer of OSGi. A CDI compliant implementation bundle may provide a service that allows the extension bundle to obtain a new container for every bean bundle.

So this is where to search to make CDI-OSGi use a specific CDI compliant container.

2.2.3. CDI API

The CDI API is described by the CDI specifications. It is provided with CDI-OSGi and defines all the CDI features usable in bean bundles.

This API will not be describe furthermore as it would be redundant with CDI specifications.

2.2.4. Extension bundle: the puppet master

The extension bundle is the orchestrator of CDI-OSGi. It may be use by any application that requires CDI-OSGi. It may be just started at the beginning of a CDI-OSGi application. It requests the extension API bundle as a dependency.

The extension bundle is the heart of CDI-OSGi application. Once it is started, provided that it finds a started integration bundle, it manages all the bean bundles. It is in charge of service automatic publishing, service injections, CDI event notifications and bundle communications.

It runs in background, it just need to be started with the OSGi environment, then everything is transparent to the user. Client bean bundles do not have to do anything in order to use CDI-OSGi functionality.

In order to perform injections the extension bundle search for a CDI compliant container service provider once it is started. Thus it can only work coupled with a bundle providing such a service: an implementation bundle.

The extension bundle provides an extension to OSGi as an extender pattern. The extension bundle, the extender, tracks for bean bundles, the extensions, to be started. Then CDI utilities are enabled for these bean bundles over OSGi environment.

The extension bundle works that way:

```
BEGIN
    start
    WHILE ! implementation_bundle.isStarted
        wait
    END_WHILE
    obtain_container_factory
    FOR bean_bundle : started_bundles
        manage_bean_bundle
        provide_container
    END_FOR
    WHILE implementation_bundle.isStarted
        wait_event
        OnBeanBundleStart
            manage_bean_bundle
            provide_container
        OnBeanBundleStop
            unmanage_bean_bundle
    END_WHILE
    stop
    FOR bean_bundle : namaged_bundles
        unmanage_bean_bundle
        stop_bean_bundle
    END_FOR
END
```

So this is where the magic happens and where OSGi applications become much more simple.

2.2.5. Integration bundle: choose a CDI compliant container

The integration bundle is responsible for providing CDI compliant containers to the extension bundle. It may be started with the extension bundle and publish a CDI container factory service. It request the integration API bundle as a dependency.

It is an implementation of the integration API but it can use any CDI implementation in order to fulfill it. So this bundle might not be unique but exist for each vendor specific CDI implementation (such as Weld).

A integration bundle may work that way:

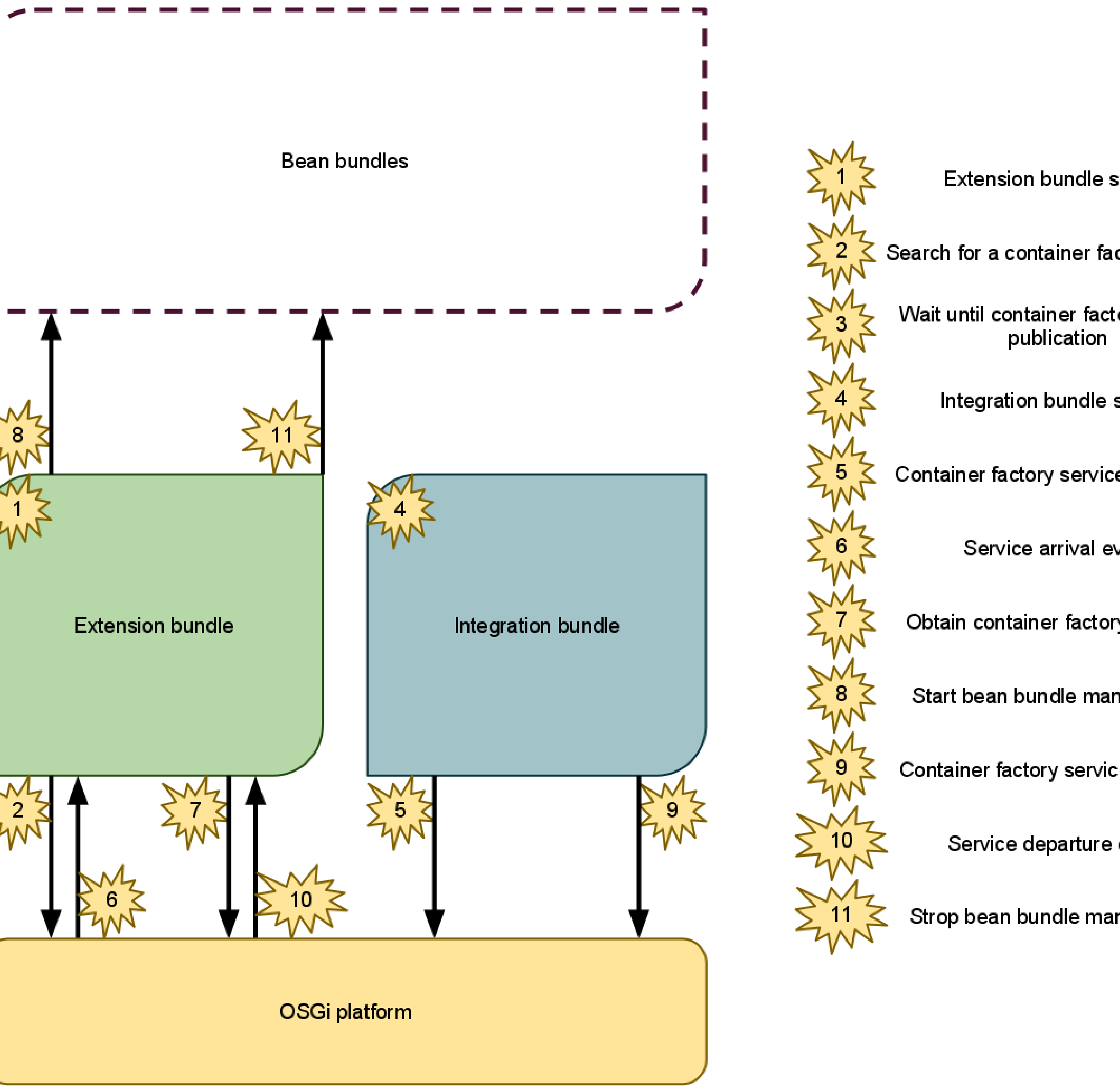
```
BEGIN
    start
    register_container_factory_service
    WHILE true
        wait
        OnContainerRequest
            provide_container
    END_WHILE
    unregister_container_factory_service
END
```

2.3. CDI-OSGi features

As an extension to OSGi, CDI-OSGi provides several features :

- Complete integration with OSGi world by the use of extender pattern and extension bundle. Thus complete compatibility with already existing tools.
- Non intruding, configurable and customizable behavior in new or upgraded application. Simple configuration and usage with annotations, completely xml free.
- Full internal CDI support for bean bundles: injection, producers, interceptors, decorators ...
- Lot of ease features for OSGi usages: injectable services, event notifications, inter-bundle communication ...
- OSGi and CDI compliance all along the way ensuring compatibility with all CDI compliant container and easy application realisation or portage.

2.4. Integration bundle discovery and CDI-OSGi start

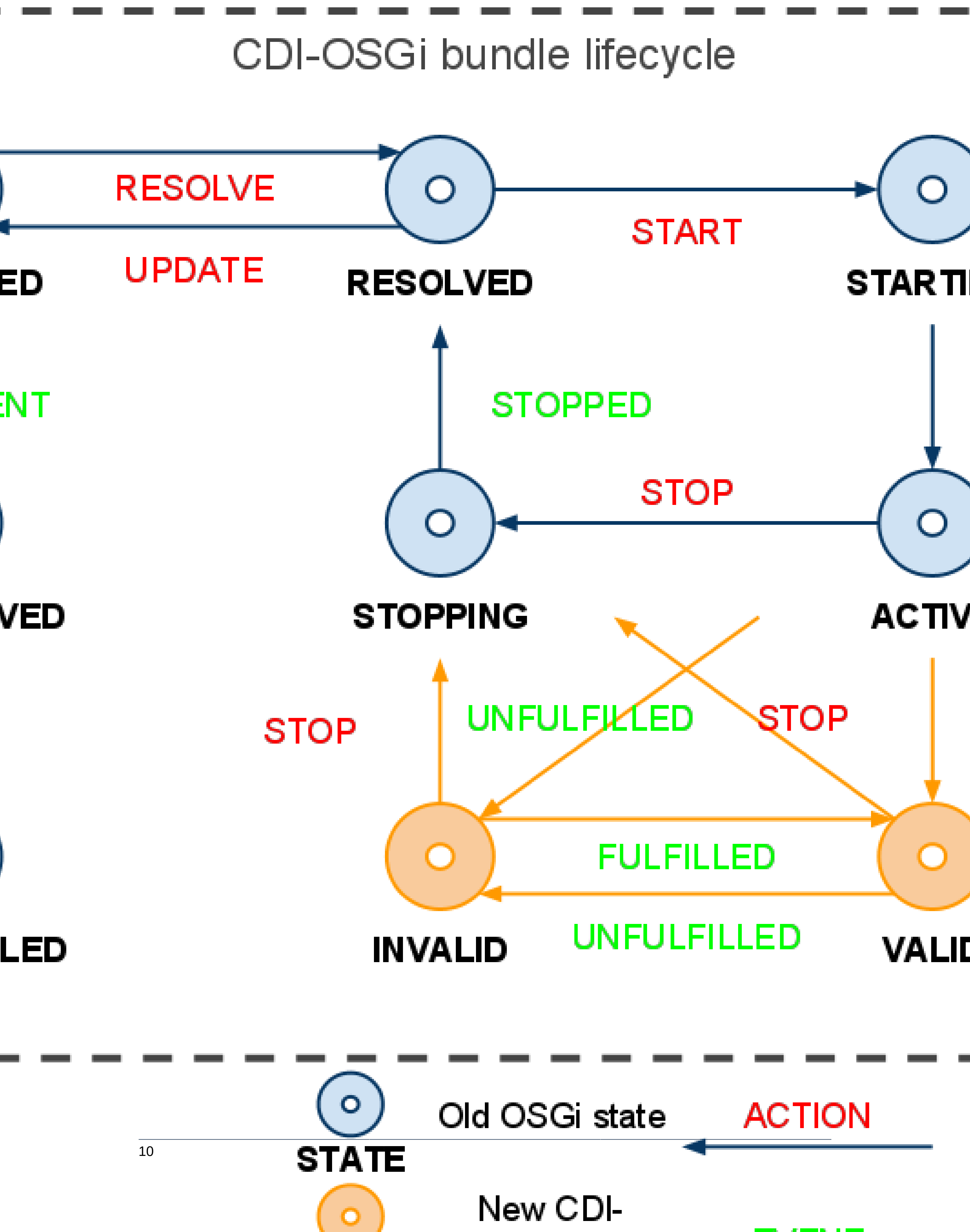


This figure shows the steps of the CDI-OSGi starting and stopping protocol. Between step 8 and step 11 the framework is in stable state and manages bean bundles.

Figure 2.2. CDI-OSGi framework start and stop protocol

2.5. The life of a bean bundle

This section presents the lifecycle of a bean bundle and how it impacts CDI and OSGi regular behaviors. Mostly bean bundles follow the same lifecycle than a regular bundle. There are only two new possible states and they do not modify the behavior from OSGi side.


CDI-OSGi bundle lifecycle
RESOLVE
UPDATE
ED
RESOLVED
START
STARTI
ENT
STOPPED
STOP
VED
STOPPING
ACTIV
STOP
UNFULFILLED
STOP
LED
FULFILLED
INVALID
UNFULFILLED
VALID
Old OSGi state
ACTION
STATE
New CDI-

This figure shows the two new states a bean bundle can be in. These states are triggered by two new events and address the CDI container dependency resolution (i.e. services annotated @Required).

Figure 2.3. The bean bundle lifecycle

The regular OSGi lifecycle is not modified by the new CDI-OSGi states as they have the same meaning than the ACTIVE state from an OSGi point of view. They only add information about the validation of required service availability.

2.6. How to make a bundle or a bean archive a bean bundle

There are very few things to do in order to obtain a bean bundle from a bean archive or a bundle. Mostly it is just adding the missing marker files and headers in the archive:

- Make a bean archive a bean bundle by adding special OSGi marker headers in its `META-INF/Manifest.MF` file.
- Or, in the other way, make a bundle a bean bundle by adding a `META-INF/bean.xml` file.

Thus a bean bundle has both `META-INF/bean.xml` file and OSGi marker headers in its `META-INF/Manifest.MF` file.

However there is a few other information that CDI-OSGi might need in order to perform a correct extension. In particular a bean bundle can not be manage by the extension bundle but by his own embedded CDI container. For that there is a new manifest header.

2.6.1. The `META-INF/bean.xml` file

The beans.xml file follows no particular rules and should be the same as in a native CDI environment. Thus it can be completely empty or declare interceptors, decorators or alternatives as a regular CDI beans.xml file.

There will be no different behavior with a classic bean archive except for CDI OSGi extension new utilities. But these don't need any modification on the `META-INF/bean.xml` file.

2.6.2. The Embedded-CDIContainer `META-INF/Manifest.MF` header

This header prevents the extension bundle to automatically manage the bean bundle that set this manifest header to true. So the bean bundle can be manage more finely by the user or use a different CDI container. If this header is set to false or is not present in the `META-INF/Manifest.MF` file then the bean bundle will be automatically manage by the extension bundle (if it is started).

How to make OSGi easy peasy

3.1. Injecting easiness in OSGi world

CDI-OSGi provides more functionality using CDI in a OSGi environment.

It mainly focuses on the OSGi service layer. It addresses the difficulties in publishing and consuming services. CDI-OSGi allows developers to:

- Automatically publish CDI beans as OSGi services
- Consume OSGi services as CDI beans

CDI-OSGi also provides utilities for event notification and communication in and between bundles as well as some general OSGi utilities. CDI-OSGi brings a complete support of CDI into bean bundles too.

3.2. CDI usage in bean bundles

Everything possible in CDI application is possible in bean bundle. They can take advantage of injection, producers, interceptors, decorators and alternative. But influence boundary of the CDI compliant container stay within the bean bundle for classic CDI usages. So external dependencies cannot be injected and interceptor, decorator or alternative of another bean bundle cannot be used (yet interceptors, decorators and alternatives still need to be declares in the bean bundle bean.xml file).

That is all we will say about classic CDI usages, please report to CDI documentation for more information.

3.3. Service bean and auto-published OSGi service

A CDI-OSGi auto-published service is described by these attributes (and their equivalents for a regular OSGi service):

- A (nonempty) set of service contracts (service class names)
- A set of qualifiers (service properties)
- A scope
- A `Publish` annotated CDI bean instance (service instance)

A CDI-OSGi service bean is described by these attributes (and their equivalents for OSGi service lookup):

- An `OSGiService` annotated or `Service<T>` typed injection point
- A type (lookup type)
- A `Filter` qualifier (lookup LDAP filter)
- A (possibly empty) set of reachable instance (lookup result)

3.4. OSGi service auto-publication with `Publish` annotation

Annotate a CDI bean class with a `Publish` annotation makes CDI-OSGi register this bean as a OSGi service. Then the service is accessible through CDI-OSGi service injection and OSGi classic mechanisms.

Automatically publish a new service implementation:

```
@Publish
```

```
public class MyServiceImpl implements MyService {
}
```

However, such an implementation also provides a regular CDI managed bean, so MyServiceImpl can also be injected using CDI within the bean bundle.

3.4.1. Service type resolution

CDI-OSGi auto-published service get their types from the following algorithm:

- If a (nonempty) contract list is provided (as an array of `Class`) with the `Publish` annotation the service is registered for all these types. This is how define a contract list:

```
@Publish(contracts = {
        MyService.class,
        AbstractClass.class
})
public class MyServiceImpl extends AbstractClass implements MyService, OtherInterface {
}
```

  The implementation class may be assignable for all of the contract types. If not, CDI-OSGi detects the problem and treats it as an error.

- Else if the implementation class possesses a (nonempty) list of non-blacklisted interfaces the service is registered for all these interface types.The blacklist is described below.
- Else if CDI-OSGi the implementation class possesses a non-blacklisted superclass the service is registered for this superclass type.
- Last if the implementation class has neither contract nor non-blacklisted interface or superclass, the service is register with is the implementation class type.

3.4.2. Service type blacklist

An CDI implementation bundle might provide a type blacklist in order to filter auto-published OSGi service allowed type. Please refer to the CDI implementation bundle documentation to see if such a blacklist is enable and how to configure it.

3.5. `OSGiService` annotated or `Service<T>` typed injection points

A `OSGiService` annotated or a `Service<T>` typed injection point is managed by CDI-OSGi through the creation of a new service bean.`OSGiService` annotation and `Service<T>` type are exclusive on injection point. If an injection point has both, CDI-OSGi detects the problem and treats it as an error.

- Direct injection with `OSGiService` annotation and `OSGiServiceBean`:

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

  Such an injection point (an OSGi service injection point) will match an unique CDI-OSGi `OSGiServiceBean`.

  For every different OSGi service injection point an unique `OSGiServiceBean` is generated by CDI-OSGi.

- Injection using programmatic lookup with `Service<T>` type and `OSGiServiceProviderBean`:

```
@Inject Service<MyService> services;
```

  Such an injection point (an OSGi service provider injection point) will match an unique CDI-OSGi `OSGiServiceProviderBean`.

  For every different OSGi service provider injection point an unique `OSGiServiceProviderBean` is generated by CDI-OSGi.

`OSGiService` annotated or a `Service<T>` typed injection points are not eligible to regular CDI injection.

3.6. `OSGiServiceBean` and `OSGiServiceProviderBean`

`OSGiServiceBean` injects an instance of the first service implementation matching the injection point.

`OSGiServiceProviderBean` injects a service provider (as a `Service<T>`) for all the service implementations matching the injection point.

Service provider allows to over-specify the matching service implementation set with additional OSGi service properties.

Service provider does not allow to subtype the matching service implementation set.

Service provider allows to instantiate the first service implementation matching the (possibly) over-specified injection point.

3.7. Clearly specify a service implementation

`Qualifier` annotated annotations might be use for both specifying auto-published services and service injection points. Such qualifiers should be seen as OSGi service properties, thus every set of qualifiers corresponds to a set of OSGi service properties and so to a OSGi service LDAP filter.

However qualifiers keep a regular meaning for the CDI generated bean of an auto-published service class.

3.7.1. Link between qualifiers and OSGi LDAP properties

A qualifier will generate an OSGi service property for each of its valued element (an element with a default value is always considered valued) following these rules:

- A valued element generate a property with this template:

```
decapitalized_qualifier_name.decapitalized_element_name=element_value.toString()
```

```
@MyQualifier(lang="EN", country="US")
```

  will generate:

```
(myqualifier.lang=EN)
```

```
(myqualifier.country=US)
```

- A non valued element with a default value generate a property with this template:

```
decapitalized_qualifier_name.decapitalized_element_name=element_default_value.toString()
```

```
@MyQualifier(lang="EN")
```

  will generate:

```
(myqualifier.lang=EN)
(myqualifier.country=US) //admitting US is the default value for the element country
```

- A non valued element with no default value generate a property with this template:

```
decapitalized_qualifier_name.decapitalized_element_name=*
```

```
@MyQualifier(lang="EN")
```

  will generate:

```
(myqualifier.lang=EN)
(myqualifier.country=*) //admitting there is no default value for the element country
```

- A qualifier with no element generate a property with this template:

```
decapitalized_qualifier_name=*
```

```
@MyQualifier()
```

  will generate:

```
(myqualifier=*)
```

3.7.2. Special qualifiers

- `OSGiService` qualifier will not generate any service property

- `Required` qualifier will not generate any service property
- `Filter` qualifier undergoes a special processing:
  - Its value element value is reused as it is as a supposedly valid OSGi service LDAP filter
  - Its each of the string of its properties element (an array of string) is reused as it is as a supposedly valid OSGi service property

It is discourage to use the `Filter` qualifier on a bean that might be use as a regular CDI bean.

3.7.3. Final LDAP filter

CDI-OSGi processes all the OSGi LDAP properties and provided OSGi LDAP filter to generate a global OSGi LDAP filter as:

- With multiple OSGi LDAP properties and a provided OSGi LDAP filter

```
(& provided_ldap_filter (ldap_property_1) (ldap_property_2) ... (ldap_property_i) )
```

- With multiple OSGi LDAP properties and no provided OSGi LDAP filter

```
(& (ldap_property_1) (ldap_property_2) ... (ldap_property_i) )
```

- With one OSGi LDAP properties and a provided OSGi LDAP filter

```
(& provided_ldap_filter (ldap_property) )
```

- With one OSGi LDAP properties and no provided OSGi LDAP filter

```
(ldap_property)
```

- With no OSGi LDAP properties and a provided OSGi LDAP filter

```
provided_ldap_filter
```

- With no OSGi LDAP properties and no provided OSGi LDAP filter

```
null
```

CDI-OSGi never ensure that, neither the provided OSGi LDAP properties, neither the provided OSGi LDAP filter, neither the generated OSGi LDAP filter, are valid.

3.7.4. Using service filtering

- On an auto-published service class:

```
@Publish
@AnyQualifier
public class MyServiceQualifiedImpl implements MyService {
}
```

Will generate an `AnyQualifier` qualified regular CDI bean and register an OSGi service with the property (anyqualifier=*).

- On an OSGi service injection point:

```
@Inject @OSGiService @AnyQualifier MyService qualifiedService;
@Inject @AnyQualifier Service<MyService> qualifiedServices;
```

Will generate an `OSGiServiceBean` and an `OSGiServiceProducerBean` looking up for OSGi services with the property (anyqualifier=*).

- With an `OSGiServiceProducerBean`:

```
services.select(new AnnotationLiteral<AnyQualifier>() {}).get().deSomething();
```

Will over-specify the valid service implementation set to those matching the property (anyqualifier=*).

- Using the special `Filter` qualifier:

```
@Publish
@Filter(value="(lang=EN)",
        properties = {"country=US",
                      "currency=*")
        })
public class MyServiceQualifiedImpl implements MyService {
}
```

Will generate an `Filter(...)` qualified regular CDI bean and register an OSGi service with the properties (lang=EN) [as a complete OSGi service LDAP filter], (country=US) and (currency=*).

3.8. Bean disambiguation and annotated type processing

CDI-OSGi ensures that every `OSGiService` annotated or `Serive<T>` typed injection point matches an unique `OSGiServiceBean` or `OSGiServiceProviderBean`.

Therefore, for every bean bundle CDI-OSGi:

- Processes annotated types
- Wraps every `OSGiService` annotated or `Service<T>` typed injection point

`OSGiService` annotated injection points are wrapped as:

```
@Inject @OSGiService @Filter(Calculated_filter) Type var_name;
```

`Service<T>` typed injection points are wrapped as:

```
@Inject @Filter(Calculated_filter) Service<Type> var_name;
```

The global OSGi LDAP filter of the final `Filter` qualifier is calculated from:

- The original set of qualifiers (except `OSGiService` and `Filter`)
- The OSGi LDAP filter value of the original `Filter` qualifier
- The set of properties of the original `Filter` annotation

3.8.1. Examples

```
@Inject @OSGiService MyService qualifiedService;
```

will become:

```
@Inject @OSGiService @Filter("") MyService qualifiedService;
```

```
@Inject @OSGiService @AnyQualifier MyService qualifiedService;
```

will become:

```
@Inject @OSGiService @Filter("(anyqualifier=*)") MyService qualifiedService;
```

```
@Inject @AnyQualifier Service<MyService> qualifiedServices;
```

will become:

```
@Inject @Filter("(anyqualifier=*)") Service<MyService> qualifiedService;
```

```
@Inject @OSGiService @AnyQualifier Service<MyService> qualifiedServices;
```

will generate an error.

```
@Inject
 @OSGiService @AnyQualifier @Filter(value="(lang=EN)",properties={"country=US","currency=*"})
 MyService qualifiedService;
```

will become:

```
@Inject @OSGiService @Filter("(&(anyqualifier=*)(lang=EN)(country=US)(currency=*)) MyService
 qualifiedService;
```

3.8.2. Justification

This figure show the need for a annotated type processing in order to remove the ambiguous dependency between regular CDI and CDI-OSGi injection points.

Figure 3.1. Annotated type processing justification

3.9. Contextual services

An auto-published service instance is a CDI contextual instance, so:

- The instance injected through a `OSGiService` annotated or `Service<T>` typed injection point might be a CDI contextual instance
- The instance obtained through a regular OSGi service checkout might be a CDI contextual instance
- In either cases CDI-OSGi ensures that the injected or obtained instance is contextual if **no** similar service is published using regular OSGi mechanism

It is discourage to use regular OSGi service publication mechanisms in a CDI-OSGi application.

3.9.1. OSGi service scopes

A CDI scope might be precised for every auto-published service class:

- If no scope is provided `Dependent` is assumed, granting a capacity similar to regular OSGi service
- Only one scope may be precised for every auto-published service class
- The scope is shared by both generated regular CDI bean and OSGi service
- The available scopes are: `Dependent`, `Singleton`, `ApplicationScoped`, `SessionScoped`, `ConversationScoped` and `RequestScoped`
- Other scope or pseudo-scope may not be supported by CDI-OSGi

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

3.10. Required services

A `OSGiService` annotated or `Serive<T>` typed injection point might be annotated `Required`

- with no influence on this injection point
- with influence on the `Valid` and `Invalid` events management in the current bean bundle

3.11. Inaccessible service at runtime

`OSGiServiceBean` and `OSGiServiceProviderBeans` bean instances are dynamically obtained OSGi service instance.

No instance might be available at runtime due to OSGi dynamism, in such case a `OSGiServiceUnavailableException` is thrown with any `OSGiServiceBean` method call or the `OSGiServiceProviderBeans` `get` method call.

3.12. OSGi facilitation

3.12.1. Service registry

CDI-OSGi allows bean bundles to directly interact with the OSGi service registry by getting a `ServiceRegistry` bean:

```
@Inject ServiceRegistry registry;
```

This bean is injectable everywhere into a bean bundle.

It allows to:

- Register a service implementation
- Obtain a service provider as a `Service<T>`
- Obtain all existing registrations
- Obtain a specific set of registrations

3.12.2. OSGi utilities

CDI-OSGi allows to obtain, by injection into bean bundles, some of the useful objects of the OSGi environment:

- The current bundle

```
@Inject Bundle bundle;
```

- The current bundle context

```
@Inject BundleContext bundleContext;
```

- The current bundle headers

```
@Inject @BundleHeaders Map<String,String>metadata;
```

- A specific current bundle header

```
@Inject @BundleHeader("Bundle-SymbolicName") String symbolicName;
```

- A specific current bundle resource file

```
@Inject @BundleDataFile("test.txt") File file;
```

- The same object of a specified bundle by symbolic name and (optional) version

```
@Inject @BundleName("com.sample.gui") @BundleVersion("4.2.1") bundle;
@Inject @BundleName("com.sample.gui") bundle;
```

```
@Inject @BundleName("com.sample.gui") @BundleVersion("4.2.1") @BundleHeaders
 Map<String,String>metadata;
```

```
@Inject @BundleName("com.sample.gui") @BundleVersion("4.2.1") @BundleHeader("Bundle-
SymbolicName") String symbolicName;
```

It is possible to precise an external bundle by bundle symbolic name and (optional) version

```
@Inject @BundleName("com.sample.gui") @BundleVersion("4.2.1") bundle;
@Inject @BundleName("com.sample.gui") bundle;
```

```
@Inject @BundleName("com.sample.gui") @BundleVersion("4.2.1") BundleContext bundleContext;
```

```
@Inject @BundleName("com.sample.gui") @BundleVersion("4.2.1") @BundleHeaders
 Map<String,String>metadata;
```

```
@Inject @BundleName("com.sample.gui") @BundleVersion("4.2.1") @BundleHeader("Bundle-
SymbolicName") String symbolicName;
```

```
@Inject @BundleName("com.sample.gui") @BundleVersion("4.2.1") @BundleDataFile("test.txt") File
 file;
```

If a `BundleVersion` annotation is provided without a `BundleName` annotation CDI-OSGi detects the problem and treats it as an error.

3.12.3. The registration

CDI-OSGi allows to obtain, by injection into bean bundles, `Registration<T>` of a specific type. A registration object represent all the bindings between a service contract class and its OSGi `ServiceRegistration`.

```
@Inject Registration<MyService> registrations;
```

It is possible to filter the obtained bindings by specifying OSGi LDAP properties and filter.

```
@Inject @AnyQualifier Registration<MyService> qualifiedRegistrations;
@Inject @Filter("(&(lang=EN)(country=US))") Registration<MyService> qualifiedRegistrations;
```

A `Registration<T>` allows to:

- Iterate over the contained bindings
- Select a subset of the bindings using OSGi LDAP properties and filter
- Obtain a service provider, as a `Service<T>` for the current bindings
- Unregister all the services for the current bindings

3.13. CDI-OSGi events

CDI-OSGi provides numerous events about OSGi events and bean bundle lifecycle events. It also allows decoupled bean bundle communication.

All these features uses CDI events mechanisms:

- These events may be listened with a `Observes` annotated parameter method

```
public void bindBundle(@Observes AbstractBundleEvent event) {
}
```

- These events may be fires with the regular CDI mechanisms

```
BeanManager beanManager;
...
beanManager.fireEvent(new
 BundleContainerEvents.BundleContainerInitialized(bundle.getBundleContext()));
```

```
Event<Object> event;
...
event.select(AbstractBundleEvent.class).fire(new BundleInstalled(bundle));
```

3.13.1. CDI container lifecycle events

CDI-OSGi provides a CDI event notification for bean bundle about bean bundle CDI container lifecycle events:

- A `BundleContainerInitialized` event is fired every time a bean bundle CDI container is initialized
- A `BundleContainerShutdown` event is fired every time a bean bundle CDI container is shutdown

3.13.2. Bundle lifecycle events

CDI-OSGi provides a CDI event notification for bean bundle about bundle lifecycle events:

- Such an event is fired every time the correspondent OSGi bundle event is fired
- All bundle lifecycle events may be listen using the `AbstractBundleEvent` event
- Specific bundle lifecycle events are: `BundleInstalled`, `BundleUninstalled`, `BundleLazyActivation`, `BundleResolved`, `BundleUnresolved`, `BundleUpdated`, `BundleStarted`, `BundleStarting`, `BundleStopped` and `BundleStopping`

It is possible to filter the listened source bundle by bundle symbolic name and (optional) version

```
public void bindBundle(@Observes @BundleName("com.sample.gui") @BundleVersion("4.2.1")
 AbstractBundleEvent event) {
}
public void bindBundle(@Observes @BundleName("com.sample.gui") BundleInstalled event) {
}
```

Only the events from the corresponding bundle are listened.

If a `BundleVersion` annotation is provided without a `BundleName` annotation CDI-OSGi detects the problem and treats it as an error.

3.13.3. Service lifecyle events

CDI-OSGi provides a CDI event notification for bean bundle about service lifecycle events:

- Such an event is fired every time the correspondent OSGi service event is fired
- All service lifecycle events may be listen using the `AbstractServiceEvent` event
- Specific bundle lifecycle events are: `ServiceArrival`, `ServiceDeparture` and `ServiceChanged`

It is possible to filter the listened source service by specification and or OSGi LDAP properties and filter

```
public void bindService(@Observes @Specification(MyService.class) AbstractServiceEvent event) {
}
public void bindService(@Observes @AnyQualifier ServiceArrival event) {
}
public void bindService(@Observes @Specification(MyService.class) @Filter("(&(lang=EN)
(country=US))") ServiceChanged event) {
}
```

Only the corresponding service events are listened.

3.13.4. Bean bundle required service dependency validation events

CDI-OSGi provides a CDI event notification for bean bundle about bean bundle required service dependency validation:

- A `Valid` event is fired every time a bean bundle got all its required service dependency validated
- A `Invalid` event is fired every time a bean bundle got one of its required service dependency invalidated

3.13.5. Intra and inter bundles communication events

CDI-OSGi provides a way to communicate within and between bean bundles:

- A `InterBundleEvent` is fired by a bean bundle

```
@Inject Event<InterBundleEvent> event;
MyMessage myMessage = new MyMessage();
event.fire(new InterBundleEvent(myMessage));
```

- A `InterBundleEvent` may be listened by every active bean bundle

It is possible to filter the listened source message by message type and ignoring the events from the current bundle

```
public void listenAllEventsFromOtherBundles(@Observes @Sent InterBundleEvent event) {
}
public  void  listenMyMessageEvents(@Observes  @Specification(MyMessage.class)  InterBundleEvent
 event) {
}
public           void           listenMyMessageEventsFromOtherBundles(@Observes           @Sent
 @Specification(MyMessage.class) InterBundleEvent event) {
}
```

Only the corresponding events are listened.

Chapter 4.

Weld-OSGi implementation