Weld-OSGi - Weld OSGi integration

Design specification

Mathieu Ancelin

`<mathieu.ancelin@serli.com>`

Matthieu Clochard

`<matthieu.clochard@serli.com>`

Preface

1. About naming and references

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

1.2. Bundle types

This document discners differents bundle types for OSGi environment using Weld-OSGi:

- *Bundle* reffers to any bundle deployable in the OSGi environment as a Java archive containing OSGi marker headers in its `META-INF/Manifest.MF` file.
- *Bean bundle* reffers to any bundle deployable in the OSGi environment and managable by Weld-OSGi as a *bundle* containing CDI marker file `META-INF/bean.xml`.
- *Regular bundle* reffers to any bundle deployable in the OSGi environment and not managable by Weld-OSGi as a *bundle* not containing CDI marker file.

2. What is this specification for ?

2.1. Contracts

This specification defines:

- The architecture, organization and workflow of the Weld-OSGi framework.
- The programming model for, and responsabilities of, application developper who uses the Weld-OSGi framework.
- The programming model for, and responsabilities of, vendor who provides the compatibility of another CDI implementation.

TODO table of content presentation

2.2. What is Weld-OSGi ?

Weld-OSGi is an extension to the Weld project as a framework for developping applications in OSGi environment. Weld-OSGi aims at:

- Providing a support for OSGi environments to the Weld project.
- Adressing the OSGi complexity using the CDI programming model.

Weld-OSGi is a group of four bundles deployable in any OSGi environment. It allows the usage of bean bundles in these OSGi environments.

2.3. Third party dependencies and environment

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

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

Weld-OSGi is a part of the Weld project and may use any of the Weld project dependencies.

Part I. Architecture of Weld-OSGi

Framework organisation

Weld-OSGi is composed of five bundles:

- The *API bundle* that describes the programming model and the OSGi extension mechanism.
- The *SPI bundle* that describes the CDI implementatio hooking mechanism.
- The *third party API bundle* that provides all the needed third party API.
- The *extension bundle* that extends the OSGi environment by managing bean bundles,
- The *integration bundle* that provides OSGi ready Weld containers to the extension bundle.

Note that as Weld-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 bean bundles. But it is not an actual part of Weld-OSGi.

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

Figure 1.1. The five bundles of Weld-OSGi

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

- Extension part: this part represents the actual OSGi extension. It detects and manages bean bundles and provides the programming model. This part is more described in chapter 2.
- Integration part: this part represents the Weld integration. It provides all the Weld containers needed by the extension part and defines how these containers should hookup with bean bundles. This part is more described in chapter 3.
- The third party API bundle : this bundle does not actually belong to a part.

Extension part

2.1. API bundle

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 bean 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. Extension bundle: the puppet master

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

The extension bundle is the heart of Weld-OSGi applications. 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 Weld-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: the integration 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 ! integration_bundle.isStarted
        wait
    END_WHILE
    obtain_container_factory
    FOR bean_bundle : started_bundles
        manage_bean_bundle
        provide_container
    END_FOR
    WHILE integration_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.

Integration part

3.1. SPI bundle

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

3.2. Integration bundle: provide Weld containers

The integration bundle is responsible for providing Weld 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.

The 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
```

Weld-OSGi features

As an extension to OSGi, Weld-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.

Chapter 5.

Weld-OSGi workflow

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

Figure 5.1. Weld-OSGi framework start and stop protocol

Chapter 6.

Bean bundles life cycle

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.

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 6.1. The bean bundle lifecycle

The regular OSGi lifecycle is not modified by the new Weld-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.

Bean bundle characteristics

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 Weld-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.

7.1. 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 Weld-OSGi extension new utilities. But these don't need any modification on the `META-INF/bean.xml` file.

7.2. 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).

Part II. Programming model of Weld-OSGi

CDI activation in bean bundles

Weld-OSGi detects a bundle as a bean bundle if:

- it possesses a META-INF/beans.xml file at its root path
- or/and it possesses one or more inner jar or zip files that possesses a META-INF/beans.xml file at their root paths

The managed set of bean classes of a bean class contains all the class file under all META-INF/beans.xml root paths.

CDI and Weld-OSGi features are enabled for all these manageable paths.

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 managed paths 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.

Service auto publication and injection

9.1. Service bean and auto-published OSGi service description

A Weld-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 Weld-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)

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

Annotate a CDI bean class with a `Publish` annotation makes Weld-OSGi register this bean as a OSGi service.

Such a service is accessible through Weld-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.

9.2.1. Service type resolution

Weld-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, Weld-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 Weld-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.

9.2.2. Service type blacklist

TODO

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

A `OSGiService` annotated or a `Service<T>` typed injection point is managed by Weld-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, Weld-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 Weld-OSGi `OSGiServiceBean`.

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

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

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

9.4. `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.

Service implementation are search into the OSGi service registry, it may be:

- An Weld-OSGi auto-published service
- An regular OSGi service

9.5. 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.

9.5.1. 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=*)
```

- Some qualifiers follow a specific processing:
  - `OSGiService` qualifier will not generate any service property
  - `Required` qualifier will not generate any service property
  - `Default` qualifier will not generate any service property
  - Any qualifier will not generate any service property
  - `Filter` and `Properties` qualifiers processing is described below

9.5.2. `Filter` and `Properties` qualifiers

`Filter` qualifier allows to specify a OSGi LDAP filter for a `OSGiServiceBean` or a `OSGiServiceProducerBean` injection point.

A `Filter` qualifier generate a "as it is" OSGi LDAP filter.

`Properties` qualifier allows to specify OSGi LDAP properties for a auto-published service class or for a `OSGiServiceBean` or a `OSGiServiceProducerBean` injection point.

A `Properties` qualifier generate a property for every one of its `Property` annotation with this template:

```
Property.name()=Property.value()
```

```
@Properties({@Property(name = "lang", value = "EN")
             @Property(name = "country", value = "US"
})
```

will generate:

```
(lang=EN)
(country=US)
```

If a `Filter` qualifier is used on a bean class Weld-OSGi detects the problem and treats it as an error.

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

9.5.3. `Filter` and `Properties` stereotypes

An annotation annotated with a `Filter` qualifier is considered similar to the `Filter` qualifier alone.

An annotation annotated with a `Properties` qualifier is considered similar to the `Properties` qualifier alone.

Declaring a filter stereotype:

```
@Filter("name=1")
public @Interface Name1 {
}
```

```
@Inject @Name1 Service<MyService> named1Services;
```

is similar to:

```
@Inject @Filter("name=1") Service<MyService> named1Services;
```

Declaring a properties stereotype:

```
@Properties({@Property(name = "name", value = "1")})
```

```
public @Interface Name1 {
}
```

```
@Publish
@Name1
public class MyServiceImpl implements MyService {
}
```

is similar to:

```
@Publish
@Properties({@Property(name = "name", value = "1")})
public class MyServiceImpl implements MyService {
}
```

9.5.4. Final LDAP filter

Weld-OSGi processes all the OSGi LDAP properties (from regular qualifiers and `Properties` qualifier) and provided OSGi LDAP filter (from `Filter` qualifier) 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
```

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

9.5.5. 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 `Properties` qualifier:

```
@Publish
@Properties({@Property(name = "country", value = "US")
             @Property(name = "lang", value = "EN")
})
public class MyServiceQualifiedImpl implements MyService {
}
```

  Will generate an `Properties(...)` qualified regular CDI bean and register an OSGi service with the properties (lang=EN) and (country=US).

- Using the special `Filter` qualifier:

```
@Inject @OSGiService @Filter("(&(lang=EN)(country=US))") MyService qualifiedService;
@Inject @Filter("(&(lang=EN)(country=US))") Service<MyService> qualifiedServices;
```

Will generate an `OSGiServiceBean` and an `OSGiServiceProducerBean` looking up for OSGi services matching the OSGi LDAP filter *(&(lang=EN)(country=US))*.

9.6. Bean disambiguation and annotated type processing

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

Therefore, for every bean bundle Weld-OSGi:

- Processes annotated types
- Wraps every `OSGiService` annotated injection point

`OSGiService` annotated injection points are wrapped as:

```
@Inject @OSGiService @Filter(Calculated_filter) 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

9.6.1. 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 @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;
```

9.6.2. Justification

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

Figure 9.1. Annotated type processing justification

9.7. 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 Weld-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 Weld-OSGi application.

9.7.1. 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 Weld-OSGi

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

9.8. Required services

A `OSGiService` annotated or `Service<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

9.9. 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.

Weld-OSGi events

Weld-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));
```

10.1. CDI container lifecycle events

Weld-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

10.2. Bundle lifecycle events

Weld-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 Weld-OSGi detects the problem and treats it as an error.

10.3. Service lifecyle events

Weld-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.

10.4. Bean bundle required service dependency validation events

Weld-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

10.5. Intra and inter bundles communication events

Weld-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.

OSGi facilitation

11.1. Service registry

Weld-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

11.2. OSGi utilities

Weld-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;
```

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 Weld-OSGi detects the problem and treats it as an error.

11.3. The registration

Weld-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