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 for 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. Bean bundle

A bean bundle is a java archive that contains both special marker file `META-INF/bean.xml` and special OSGi marker headers in its`META-INF/Manifest.MF`.

A bean bundle may be deployed in an OSGi environment with CDI-OSGi and then be managed by the CDI-OSGi. It enables both OSGi and CDI features for that bean bundle.

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.

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 more simple 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, is used. This integration is possible through a well-defined bootstrapping API.

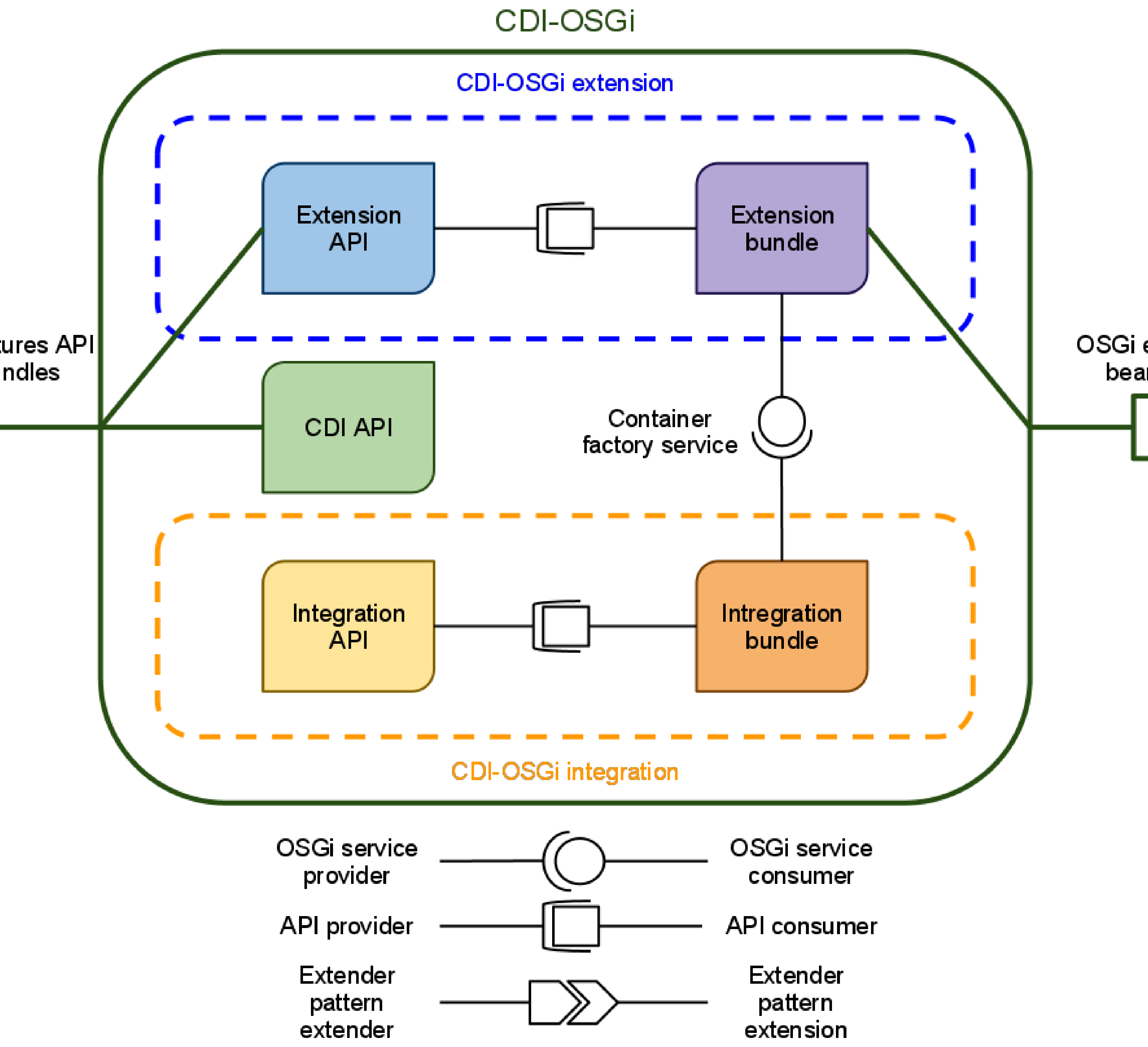
CDI-OSGi is a framework that may be used in an OSGi environment and composed by five bundles.

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.

1.2.3. Organization and interactions between CDI-OSGi bundles



This figure shows the five bundles of CDI-OSGi and the links between them. Each is described below.

Figure 1.1. The five bundles of CDI-OSGi

1.2.3.1. CDI-OSGi extension

The blue part represents the CDI OSGi extension. It is composed of one API bundle and its implementation. 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.

1.2.3.2. 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 and 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 bundles. So the extension bundle is commutable to support various CDI implementations.

1.2.3.3. 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.

1.2.3.4. Interaction between bundles

User client bean bundles should only know about the extension API bundle 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.

1.3. What about other frameworks

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

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

1.4. 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. API bundles, extension bundle and integration bundle

CDI-OSGi is composed of five bundles:

- The three API bundles that provide the API used to define both utilities provided and hooking up system with integration bundle,
- 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.

2.1.1. Extension API, integration API and CDI API

Three API bundles expose all packages that could be needed by developers; both for client applications using CDI-OSGi and for CDI compliant container integration. Their packages may be imported by bean bundles, extension bundle or implementation bundle.

These three distinct API are:

- The extension API that describes all the new features provided by CDI-OSGi in the OSGi environment.
- The integration API that describes how CDI compliant containers may be used by CDI-OSGi.
- The CDI API that describes all the regular CDI features provided by CDI-OSGi to bean bundles.

These three API are more described below.

2.1.1.1. Extension API

The extension API defines all the features provided to OSGi environment using CDI specifications. 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 bean bundles. Mostly it is about publishing and consuming injectable services in a CDI way.

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

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

2.1.1.2. 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 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 for each of its managed bean bundles.

Moreover the integration API allows to mix CDI implementations 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 and the behavior of a particular bean bundle can be particularized.

All this bootstrapping mechanism works using the service layer of OSGi. An 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 CDI-OSGi use a specific CDI compliant container.

2.1.1.3. 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.

2.1.2. 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 with OSGi environment. 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 publishings, service injections, CDI event notifications and bundle communications. It runs in background, 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 container factory service once it is started. Thus it can only work coupled with a bundle providing such a service: an integration bundle.

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

2.1.3. 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 requests 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).

2.2. An OSGi extension for CDI support: the extension API

2.2.1. 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.

We will see in the next sections these features in deep through the description of the extension API.

2.2.2. The interfaces

Extension API provides few interfaces that describe all specifics about OSGi service injection.

2.2.2.1. The `Service` interface

```
public interface Service<T> extends Iterable<T> {

    T get();
    Service<T> select(Annotation... qualifiers);
    Service<T> select(String filter);
    boolean isUnsatisfied();
    boolean isAmbiguous();
    int size();
}
```

It represents a service instance producer parametrized by the service to inject. It has the same behavior than CDI `Instance<T>` except that it represents only OSGi service beans.

IT allows to:

- Wrap a list of potential service implementations as an `Iterable` java object,
- Select a subset of these service implementations filtered by `Qualifiers` or LDAP filters,
- Iterate through these service implementations,
- Obtain an instance of the first remaining service implementations,
- Obtain utility information about the contained service implementations.

OSGi services should not be subtyped.

2.2.2.2. The `Registration` interface

```
public interface Registration<T> extends Iterable<Registration<T>> {

    void unregister();
    <T> Service<T> getServiceReference();
    Registration<T> select(Annotation... qualifiers);
    Registration<T> select(String filter);
    int size();
}
```

This interface represents the registrations of an injectable service in the service registry. Its behavior is similar to `Service<T>`, thus it might represent the iterable set of all the registrations of a service.

It allows to:

- Wrap a list of service registration (i.e. the bindings between a service and its implementations) as an `Iterable` java object,
- Select a subset of these registration filtered by `Qualifiers` or LDAP filters,
- Iterate through these service registrations,
- Obtain the service implementations list as a `Service<T>`,
- Get the number of registrations (i.e the number of registered service implementations).

OSGi services should not be subtyped.

2.2.2.3. The `RegistrationHolder` interface

```
public interface RegistrationHolder {

    List<ServiceRegistration> getRegistrations();
    void addRegistration(ServiceRegistration reg);
    void removeRegistration(ServiceRegistration reg);
    void clear();
    int size();
}
```

This interface represents the bindings between a service and its registered implementations. It is used by `Registration` to maintain the list of registration bindings. It uses OSGi `ServiceRegistration`.

It allows to:

- Wrap a list of `ServiceRegistration` as binding between a service and its implementations as a `List`,
- Handle this list with addition, removal, clearing and size operations.

2.2.2.4. The `ServiceRegistry` interface

```
public interface ServiceRegistry {

    <T> Registration<T> registerService(Class<T> contract, Class<? extends T> implementation);
    <T, U extends T> Registration<T> registerService(Class<T> contract, U implementation);
    <T> Service<T> getServiceReferences(Class<T> contract);
}
```

This interface represents a service registry where all OSGi services may be handled.

It allows to:

- Register a service implementation with a service, getting back the corresponding `Registration`,

- Obtain the service implementations list as a `Service<T>`.

2.2.3. The events

Extension API provides numerous events that notify about life-cycle steps for CDI container management, service management, bundle management, bean bundle validation and bundle communication management. They all use CDI event system.

CDI container events, as well as service and bundle events, are regrouped using three classes. Thus each category of event has:

- An abstract class representing all the events of this category, defining the global comportment and allowing to handle all the lifecycle steps in one operation,
- An enumeration describing the list of possible event types, allowing to discriminate a particular lifecycle step of the category,
- A list of implementation class for each of the above type, allowing to handle a particular lifecycle step only. These implementation classes are regrouped as static inner classes of a global category event class.

2.2.3.1. The bundle events

The representing abstract class:

```
public abstract class AbstractBundleEvent {

    private final Bundle bundle;

    public AbstractBundleEvent(Bundle bundle) {...}

    public abstract BundleEventType getType();
    public long getBundleId() {...}
    public String getSymbolicName() {...}
    public Version getVersion() {...}
    public Bundle getBundle() {...}
}
```

This abstract class represents all the CDI-OSGi bundle events as a superclass.

It allows to:

- Represent all bundle events,
- Retrieve the current event type as a `BundleEventType`,
- Retrieve the firing bundle and its information.

It may be used in `Observes` method in order to listen all bundle events.

The possible event types:

```
public enum BundleEventType {
    INSTALLED,
    LAZY_ACTIVATION,
    RESOLVED,
    STARTED,
    STARTING,
    STOPPED,
    STOPPING,
    UNINSTALLED,
    UNRESOLVED,
    UPDATED,
}
```

This enumeration lists all possible states of a bundle and the corresponding event types.

The implementation classes (in the global event class):

```
public class BundleEvents {

    public static class BundleInstalled extends AbstractBundleEvent {

        public BundleInstalled(Bundle bundle) {...}

        @Override
        public BundleEventType getType() {...}
    }
    //next classes follow the same template as above
    public static class BundleLazyActivation extends AbstractBundleEvent {...}
    public static class BundleResolved extends AbstractBundleEvent {...}
    public static class BundleStarted extends AbstractBundleEvent {...}
    public static class BundleStarting extends AbstractBundleEvent {...}
    public static class BundleStopped extends AbstractBundleEvent {...}
    public static class BundleStopping extends AbstractBundleEvent {...}
    public static class BundleUninstalled extends AbstractBundleEvent {...}
    public static class BundleUnresolved extends AbstractBundleEvent {...}
    public static class BundleUpdated extends AbstractBundleEvent {...}
}
```

This class wraps all the bundle events as inner static classes. There is one event class by `BundleEventType`.

Each inner class allows to:

- Represent a specific bundle event,
- Retrieve the same information as `AbstractBundleEvent`.

They may be used in `Observes` method in order to listen a specific bundle event.

2.2.3.2. The service events

The representing abstract class:

```
public abstract class AbstractServiceEvent {

    private final ServiceReference reference;
    private final BundleContext context;
    private List<String> classesNames;
    private List<Class<?>> classes;
    private Map<Class, Boolean> assignable;

    public AbstractServiceEvent(ServiceReference reference, BundleContext context) {...}

    public abstract ServiceEventType eventType();
    public ServiceReference getReference() {...}
    public <T> TypedService<T> type(Class<T> type) {...}
    public Object getService() {...}
    public boolean ungetService() {...}
    public boolean isTyped(Class<?> type) {...}
    public Bundle getRegisteringBundle() {...}
    public List<String> getServiceClassNames() {...}
    public List<Class<?>> getServiceClasses() {...}

    public static class TypedService<T> {

        private final BundleContext context;
        private final ServiceReference ref;
        private final Class<T> type;

        TypedService(BundleContext context, ServiceReference ref, Class<T> type) {...}

       static <T> TypedService<T> create(Class<T> type, BundleContext context, ServiceReference
 ref) {...}
        public T getService() {...}
        public boolean ungetService() {...}
    }
}
```

This abstract class represents all the CDI-OSGi service events as a superclass.

It allows to:

- Represent all service events,
- Retrieve the current event type as a `ServiceEventType`,
- Retrieve the affected `ServiceReference`, the corresponding information and registering `Bundle`,
- Manipulate the service,
- Retrieve the firing `BundleContext`.

It may be used in `Observes` method in order to listen all service events.

The possible event types:

```
public enum ServiceEventType {

    SERVICE_ARRIVAL,
    SERVICE_DEPARTURE,
    SERVICE_CHANGED
}
```

This enumeration lists all possible states of a service and the corresponding event types.

The implementation classes (in the global event class):

```
public class ServiceEvents {

    public static class ServiceArrival extends AbstractServiceEvent {

        public ServiceArrival(
                ServiceReference ref, BundleContext context) {
            super(ref, context);
        }

        @Override
        public ServiceEventType eventType() {
            return ServiceEventType.SERVICE_ARRIVAL;
        }
    }
    //next classes follow the same template as above
    public static class ServiceChanged extends AbstractServiceEvent {...}
    public static class ServiceDeparture extends AbstractServiceEvent {...}

}
```

This class wraps all the service events as inner static classes. There is one event class by `ServiceEventType`.

Each inner class allows to:

- Represent a specific service event,
- Retrieve the same information as `AbstractServiceEvent`.

They may be used in `Observes` method in order to listen a specific service event.

2.2.3.3. The bundle container events

The representing abstract class:

```
public abstract class AbstractBundleContainerEvent {

    private BundleContext bundleContext;

    public AbstractBundleContainerEvent(final BundleContext context) {...}

    public abstract BundleContainerEventType getType();
    public BundleContext getBundleContext() {...}
}
```

This abstract class represents all the CDI-OSGi bundle container events as a superclass.

It allows to:

- Represent all bundle container events,

- Retrieve the current event type as a `BundleContainerEventType`,
- Retrieve the firing `BundleContext`.

It may be used in `Observes` method in order to listen all bundle container events.

The possible event types:

```
public enum BundleContainerEventType {

    INITIALIZED,
    SHUTDOWN
}
```

This enumeration list all possible states of a bundle container and the corresponding event types.

The implementation classes (in the global event class):

```
public class BundleContainerEvents {

    public static class BundleContainerInitialized extends AbstractBundleContainerEvent {

        public BundleContainerInitialized(BundleContext context) {
            super(context);
        }

        @Override
        public BundleContainerEventType getType() {
            return BundleContainerEventType.INITIALIZED;
        }
    }
    //next class follows the same template as above
    public static class BundleContainerShutdown extends AbstractBundleContainerEvent {...}
}
```

This class wraps all the bundle container events as inner static classes. There is one event class by `BundleContainerEventType`.

Each inner class allows to:

- Represent a specific bundle container event,
- Retrieve the same information as `AbstractBundleContainerEvent`.

They may be used in `Observes` method in order to listen a specific bundle container event.

2.2.3.4. The bean bundle validation events

Two events notify about the validation of bean bundle. After it starts a bean bundle should validate its required service dependencies. The `Valid` and `Invalid` events occurs when such dependencies are fulfilled or unfulfilled. These two new states are described in an enumeration: `BundleState`.

```
public class Valid {
    private final Bundle bundle;

    public Valid(Bundle bundle) {...}

    public long getBundleId() {...}
    public String getSymbolicName() {...}
    public Version getVersion() {...}
    public Bundle getBundle() {...}
}
```

This class represents all bean bundle validation event.

It allows to:

- Represent all bean bundle validation events,
- Retrieve the validated bean bundle and its information.

It may be used in `Observes` method in order to listen all bean bundle validation events.

```
public class Invalid {

    private final Bundle bundle;

    public Invalid(Bundle bundle) {...}

    public long getBundleId() {...}
    public String getSymbolicName() {...}
    public Version getVersion() {...}
    public Bundle getBundle() {...}
}
```

This class represents all bean bundle invalidation event.

It allows to:

- Represent all bean bundle invalidation events,
- Retrieve the invalidated bean bundle and its information.

It may be used in `Observes` method in order to listen all bean bundle invalidation events.

```
public enum BundleState {
    VALID, INVALID
}
```

This enumeration lists the two new states of a bean bundle.

A bean bundle is in `VALID` state if all its required service dependencies are validated otherwise is in `INVALID` state. Every time a bean bundle goes from one state to another a corresponding `Valid` or `Invalid` event may be fired.

2.2.3.5. The `InterBundleEvent` event

```
public class InterBundleEvent {

    private final Object event;
    private boolean sent = false;
    private Class<?> type;

    public InterBundleEvent(Object event) {...}
    public InterBundleEvent(Object event, Class<?> type) {...}

    public Class<?> type() {...}
    public boolean isTyped(Class<?> type) {...}
    public <T> Provider<T> typed(Class<T> type) {...}
    public Object get() {...}
    public boolean isSent() {...}
    public void sent() {...}
}
```

This class represents a communication event between bean bundles.

It allows to specify:

- The message as an `Object`,
- The type of the message as a `Class`,
- The origin of the message (within or outside the bundle).

It may be used in `Observes` method in order to listen inter-bundle communications.

2.2.4. The annotations

Extension API provides annotations in order to easily use CDI-OSGi features.

2.2.4.1. The `OSGiBundle`, `BundleName` and `BundleVersion` annotations

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
public @interface OSGiBundle {
    @Nonbinding String value();
    @Nonbinding String version() default "";
}
```

This annotation qualifies an injection point that represents a bundle or a bundle relative object.

It allows to specify:

- The symbolic name of the bundle, as a required value,
- The version of the bundle, as an optional value.

The symbolic name and version are **not** actually qualifying the injection point, thus this annotation is for global bundle injection point with additional data. In order to actually discriminate on the symbolic name or version see `BundleName` and `BundleVersion` annotations.

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
public @interface BundleName {
    String value();
}
```

This annotation qualifies an injection point that represents a bundle or a bundle relative object.

It allows to specify the symbolic name of the bundle, as a required value.

The symbolic name actually discriminate the injection point, thus this annotation is for specific bundle relative injection point. For global bundle relative injection point see `OSGiBundle` annotation. To discriminate the bundle version see `BundleVersion`.

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
public @interface BundleVersion {
    String value();
}
```

This annotation qualifies an injection point that represents a bundle or a bundle relative object.

It allows to specify the version of the bundle, as a required value.

The version actually discriminate the injection point, thus this annotation is for specific bundle relative injection point. For global bundle relative injection point see `OSGiBundle` annotation. To discriminate the bundle symbolic name see `BundleName`.

2.2.4.2. The `BundleDataFile`, `BundleHeader` and `BundleHeaders` annotations

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
public @interface BundleDataFile {
    @Nonbinding String value();
}
```

This annotation qualifies an injection point that represents a bundle data file.

It allows to specify the relative path of the file in the bundle, as a required value.

The file path is **not** actually qualifying the injection point, thus this annotation is for global bundle data file injection point with additional data. To discriminate the bundle use `OSGiBundle` or `BundleName` and `BundleVersion` annotations.

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
```

```
public @interface BundleHeader {
    @Nonbinding String value();
}
```

This annotation qualifies an injection point that represents a specific bundle header.

It allows to specify the name of the bundle header, as a required value.

The header name is **not** actually qualifying the injection point, thus this annotation is for global specific bundle header injection point with additional data. To discriminate the bundle use `OSGiBundle` or `BundleName` and `BundleVersion` annotations. To obtain all the bundle headers see `BundleHeaders` annotations.

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
public @interface BundleHeaders {
}
```

This annotation qualifies an injection point that represents all headers of a bundle.

To discriminate the bundle use `OSGiBundle` or `BundleName` and `BundleVersion` annotations. To obtain a specific bundle header see `BundleHeader` annotation.

2.2.4.3. The `OSGiService` annotation

```
@Qualifier
@Target({ TYPE, METHOD, PARAMETER, FIELD })
@Retention(RetentionPolicy.RUNTIME)
public @interface OSGiService {
}
```

This annotation qualifies an injection point that represents a service from the OSGi service registry.

It may be use to obtain an injected OSGi service using `Service` interface or directly the service contract interface. The injected service might be filtered using regular `Qualifier` annotations or a LDAP filter with `Filter` annotation. It also might be mark as required for bundle running using `Required` annotation.

2.2.4.4. The `Publish` and `Property` annotations

```
@Target({ TYPE })
@Retention(RetentionPolicy.RUNTIME)
public @interface Publish {
    public Class[] contracts() default {};
    public Property[] properties() default {};
}
```

This annotation notices that this type is an OSGi service implementation and should be automatically published in the OSGi service registry.

It allows to specify:

- The contract interfaces of implemented service, as an optional array of `Classes`,
- The properties of the published service implementation, as an optional array of `Property`.

The published implementation might be discriminated using regular `Qualifier` annotations or a LDAP filter with `Filter` annotation.

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Property {
    String name();
    String value();
}
```

This annotation wraps an OSGi service property used for automatic OSGi service publishing.

It allows to specify:

- The name of the property, as a required `String`,
- The value of the property, as a required `String`.

It may be used within the `Publish` annotation to provide the published service implementation properties.

2.2.4.5. The `Required` annotation

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
public @interface Required {
}
```

This annotation qualifies an injection point that represents a required service.

It may be coupled with the `OSGiService` annotation in order to qualify a service that must be available for the application to start.

2.2.4.6. The `Filter` annotation

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
public @interface Filter {
    String value();
}
```

This annotation qualifies an injection point that represents a LDAP filtered service.

It allows to specify the LDAP filter, as a required `String`.

It may be coupled with a `OSGiService` or a `Publish` annotation in order to filter the injected or published service implementations. The LDAP filtering acts on `Qualifier` or `Property` annotations or regular OSGi properties used in service publishing.

2.2.4.7. The `Specification` annotation

```
@Qualifier
@Target({ PARAMETER })
@Retention(RetentionPolicy.RUNTIME)
public @interface Specification {
    Class value();
}
```

This annotation qualifies an injection point that represents events whose match one of the specifications.

It allows to specify the specification interfaces of the `Events` parametrizing types, as a required `Class`.

It may be used in an `Observes` method to restrict the listened events.

2.2.4.8. The `Sent` annotation

```
@Qualifier
@Target({ PARAMETER })
@Retention(RetentionPolicy.RUNTIME)
public @interface Sent {
}
```

This annotation qualifies an injection point that represents an `InterBundleEvent` from outside the current `Bundle`.

It may be used in an `Observes` method to restrict the listened `InterBundleEvent`. It allows to ignore the `InterBundleEvent` from within the current bundle.

2.3. A standard bootstrap for CDI container integration: the integration API

2.3.1. Why an integration API

CDI-OSGi could have been carried out with an internal CDI compliant container and work very well. Every one of the specifications above would have been fulfilled. But no other specific utilities or CDI evolution could have been added without a complete modification of CDI-OSGi.

Then there is the integration API. With that, developers are able to provide their own integration bundle. Thus it is possible to use any CDI compliant container with a CDI-OSGi application or mix up different CDI compliant containers for different bundles.

Integration API describes what needs the extension bundle to work (and therefore what may provide the integration bundle):

- A CDI container with standard comportment and functionality,
- A factory for such CDI containers initialized and ready to run.

2.3.1.1. The `CDIContainer` interface

```
public interface CDIContainer extends  Iterable<CDIContainer> {

    boolean initialize();
    boolean shutdown();
    boolean isStarted();
    void fire(InterBundleEvent event);
    Bundle getBundle();
    BeanManager getBeanManager();
    Event getEvent();
    Instance<Object> getInstance();
    Collection<String> getBeanClasses();
    Collection<ServiceRegistration> getRegistrations();
    void setRegistrations(Collection<ServiceRegistration> registrations);
    CDIContainer select(Bundle bundle);
    CDIContainer select(String name, String version);
    int size();
}
```

This interface represents an iterable list of CDI containers used by CDI-OSGi.

It allows to:

- Navigate through the list of CDI containers as an `Iterable`,
- Obtain the number of CDI containers,
- Select a specific container by its bundle,
- Start and stop the selected CDI container,
- Obtain the state of the selected CDI container,
- Obtain the corresponding `Bundle`, `BeanManager`, `Event`, managed bean `Class` and `Instance` and registred service as `ServiceRegistration`,
- Fire `InterBundleEvent`.

2.3.1.2. The `CDIContainerFactory` interface

```
public interface CDIContainerFactory {

    String getID();
    Set<String> getContractBlacklist();
    CDIContainer container(Bundle bundle);
}
```

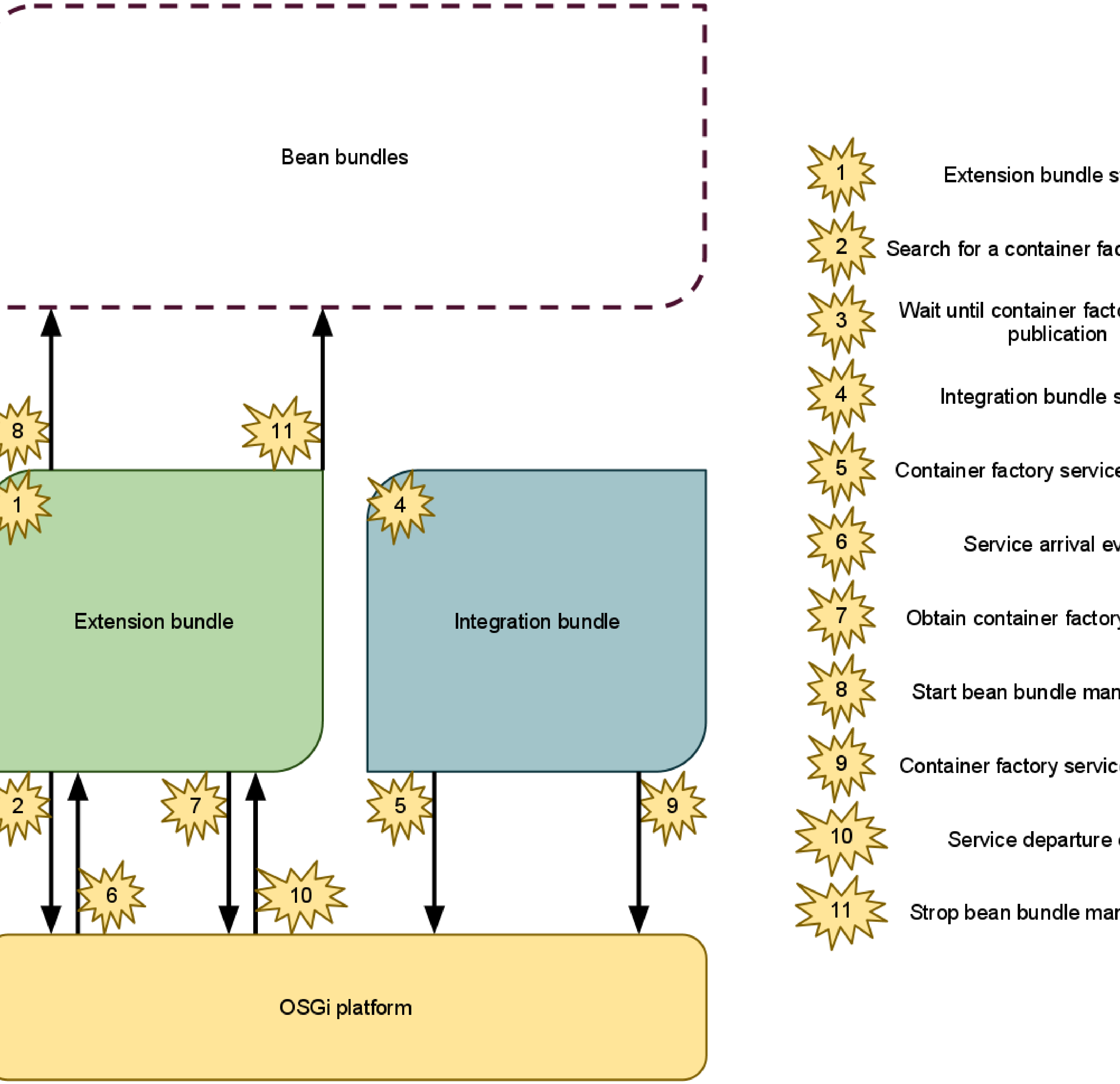
This interface represents a CDI container factory used by CDI-OSGi in order to obtain `CDIContainer`.

It allows to:

- Obtain the CDI container of a specific bean `Bundle` (singleton for each bean bundle),

- Provide a interface black list for service publishing,
- Obtain the ID of the used CDI implementation.

2.3.2. 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 2.1. CDI-OSGi framework start and stop protocol

2.3.3. Embedded mode

//TODO

2.4. An orchestrator: the extension bundle

2.4.1. The extender pattern

CDI-OSGi 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.

2.4.2. 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
```

2.5. A interchangeable CDI container factory: the integration bundle

The integration bundle is necessary in a CDI-OSGi application. This section explains the part the integration bundle plays but does not enter in specifics because it depends on the CDI implementation used.

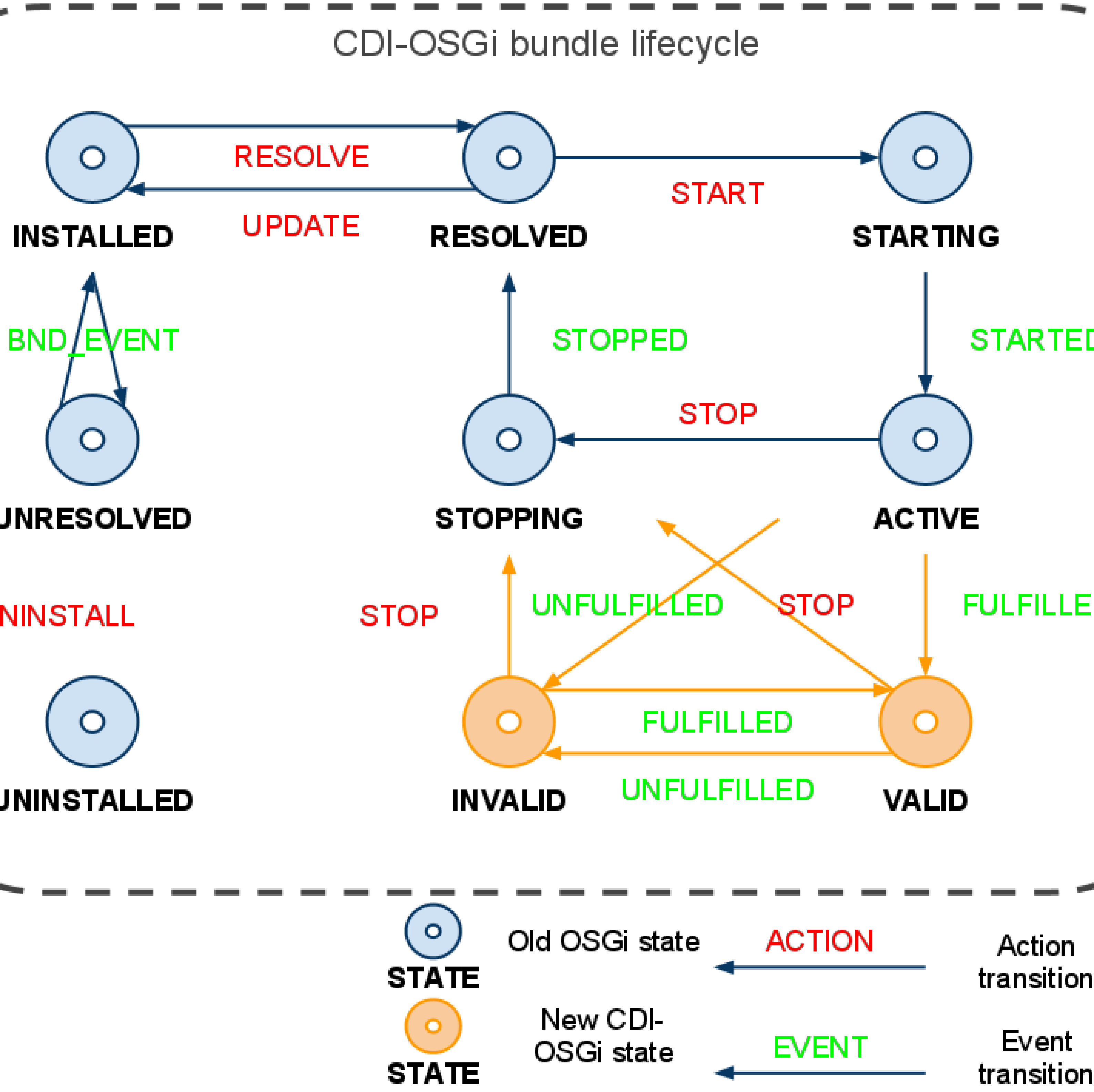
Weld-OSGi chapter presents how an implementation bundle can work.

2.5.1. A implementation bundle may work that way:

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

2.6. 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.



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 2.2. 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.

2.7. 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.

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

2.7.2. 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).

How to make OSGi easy peasy

3.1. CDI usage in bean bundles

Everything possible in CDI applications 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.

3.2. 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 publish and consume OSGi services as CDI beans. However, since OSGi services are dynamic there are some differences with classic bean injection. This section presents how OSGi services can be published and consumed using CDI-OSGi.

CDI-OSGi also provides utilities for event notification and communication in and between bundles as well as some general OSGi utilities.

Examples use this very sophisticated service interface:

```
public interface MyService {
    void doSomething();
}
```

3.2.1. Service, implementation, instance and registration

First it is important to be clear about what are a service, its implementations, its instances and its registrations.

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.

3.2.2. OSGi services injection

There are two ways to obtain a service instances using CDI-OSGi: direct injection and programmatic lookup.

3.2.2.1. Direct injection using `@OSGiService` annotation

The main way to perform an OSGi injection is to use the `@Inject @OSGiService` annotation combination. It acts like a common injection except that CDI-OSGi will search for injectable instances in the service registry.

That is how it looks like:

```
@Inject @OSGiService MyService service;
service.doSomething();
```

The behavior is similar with classic CDI injection. `@OSGiService` is just a special qualifier that allows extension bundle to manage the injection instead of CDI implementation.

3.2.2.2. Injection using programmatic lookup

Instead of obtain directly a service instance it is possible to choose between service implementations and instantiate one at runtime. The interface `Service<T>` works as a service instance producer: it retrieves all the corresponding (to the service parametrized type) service implementations and allows to get an instance for each.

Service implementations and a corresponding instance can be obtained like that:

```
@Inject Service<MyService> services;
services.get().doSomething();
```

All implementations can also be iterated like that:

```
@Inject Service<MyService> services;
for (MyService service : services) {
    service.get().doSomething();
}
```

The `get()` method returns an instance of the first available service implementation.

`Service<T>` acts like CDI `Instance<T>` except that the injection process is managed by the extension bundle instead of CDI implementation. So the available implementations are searched dynamically into the service registry.

3.2.3. OSGi service automatic publishing with `@Publish` annotation

CDI-OSGi allows developers to automatically publish service implementation. There is nothing to do, just put the annotation. OSGi framework is completely hidden. 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 {
    @Override
    public void doSomething() {
    }
}
```

It registers a service `MyService` with the implementation `MyServiceImpl`.

The behavior is similar with classic CDI bean registration, except that services are not register into the bean manager but into the service registry.

It is possible to specify the contracts interface that this specific implementation fulfill using the `Publish` annotation.

```
@Publish(contracts = {MyService.class, MyOtherService.class})
public class MyServiceImpl {

    public void doSomething() {
    }
}
```

If no contract list are provided, the implementation will match each of its interface types. Every CDI container of a bean bundle might have a blacklist that excludes some interface from being register as a service type. See the specific integration bundle doccumentation to obtain the used blacklist. This blacklist is only used when no interface array is provided within the `Publish` annotation.

3.2.4. Clearly specify a service implementation

There might be multiple implementations of the same service. It is possible to provide properties to an implementation in CDI-OSGi. This qualification is available both during publishing and consuming.

There are two ways for qualifying: a CDI like and a OSGi like, both are presented below.

3.2.4.1. Using `@Qualifier` annotations

Qualifiers are used like in classic CDI applications. An implementation can be qualified by as many qualifiers as needed. An injection point can be also qualified in order to restraint the potential injected implementations. It is finally possible to select the instance produced when using the `Service<T>` interface.

Qualified service publishing:

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

    @Override
    public void doSomething() {
    }
}
```

Qualified injection point:

```
@Inject @OSGiService @AnyQualifier MyService qualifiedService;
qualifiedService.doSomething();
```

or with `Service<T>`

```
@Inject @AnyQualifier Service<MyService> service;
service.get().doSomething();
```

The qualifiers should be seen as the service properties. Here another example:

```
@Publish
@Lang(EN)
@Country(US)
@ApplicationScoped
public class MyServiceImpl implements MyService {

    @Override
    public void doSomething() {
    }
}

@Inject @Lang(EN) @Country(US) Service<MyService> service;
```

The behavior is similar with classic CDI qualifiers. But there another possibility in order to qualify a service.

3.2.4.2. Filtering services

Since CDI-OSGi services stay OSGi services they can be filtered through LDAP filter. Properties might be added at publishing using the `Publish` annotation values. Then an LDAP filter can be use at injection point using the `Filter` annotation.

An example is worth a thousand words:

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

    @Override
    public void doSomething() {
    }
}
```

As many `@Property` annotation as wanted can be added, they are registered with the service implementation.

Then it is possible to filter an injection point with the `Filter` annotation like a regular LDAP filter:

```
@Inject @Filter("(&(lang=*)(country=US))") Service<MyService> service;
```

or

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

3.2.4.3. Links between qualifier annotations and LDAP filtering

A service implementation has properties. These properties can be added by both `Qualifier` annotations and `Property` annotations of a `Filter` annotation. Therefore there is a link between `Qualifier` and LDAP filter used in `Filter` annotation as `Property`.

A property is a couple of `String`, the first entry is the name of the property, the second the value. This scheme is clear in the `Property` annotation. For `Qualifier` the property name is the decapitalized `Qualifier` class name and the property value is the `value` of the `Qualifier` (empty `String` is assumed if `value` does not exist).

```
@Publish
@Lang(EN)
@Country(US)
@ApplicationScoped
public class MyServiceImpl implements MyService {
}

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

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

@Inject @Lang(EN) @Country(US) Service<MyService> service;
```

It is even possible to mix up `Qualifier` and `Property`:

```
@Publish({
    @Property(name="lang", value="EN"),
})
@Country(US)
@ApplicationScoped
public class MyServiceImpl implements MyService {
}

@Inject @Filter("lang=EN") @Country(US) Service<MyService> service;
```

3.2.4.4. Filtering after injection

It is possible to do a programmatic filtering after injection using `Service<T>` like with `Instance<T>`:

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

or

```
@Inject Service<MyService> services;
services.select("(&(lang=*)(country=US))").get().deSomething();
```

However OSGi service may not be subtyped, thus it is not possible to use `select()` methods with a specified subclass like with `Instance<T>`

3.2.5. Contextual services

Like for bean instances, service instances are contextual. Every implementation is bounded to a particular scope. Provided that an satisfactory implementation is available, a service injection will return a contextual instance of the implementation.

All CDI scopes are available for CDI-OSGi services and their use is the same:

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

A instance will be shared by the entire application.

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

A new instance is created for every request.

If no scope is provided `Dependent` scope is assumed, a new instance will be create for every injection. An instance obtained with regular OSGi mechanisms assumes a `ApplicationScoped` scope in order to maintain regular OSGi comportment.

3.2.6. The registration

A registration object represent all the bindings between a service contract class and its OSGi `ServiceRegistrations`. With this object it is possible to navigate through multiple implementations of the same service, obtain the corresponding `Service<T>` or unregister these implementations.

3.2.6.1. registration injection

A registration is obtained like that:

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

The injection point can be filtered using qualifiers:

```
@Inject @AnyQualifier Registration<MyService> qualifiedRegistrations;
```

or using LDAP filter:

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

3.2.6.2. Navigate into registrations and filter them

It is possible to iterate through registration:

```
if(registrations.size() > 0) {
    for(Registration<T> registration : registrations) {
    }
}
```

It is also possible to request a subset of the service implementations using qualifiers:

```
Registration<T>                           filteredRegistrations                          =
 registrations.select(new AnnotationLiteral<AnyQualifier>() {});
```

or a LDAP filter:

```
Registration<T> filteredRegistrations = registrations.select("(&(lang=EN)(country=US))");
```

or both:

```
Registration<T>                           filteredRegistrations                           =
      registrations.select(new      AnnotationLiteral<AnyQualifier>()      {}).select("(&(lang=EN)
(country=US))");
```

Link between qualifier and LDAP filter as well as subtyped service selection are explain above in service injection section.

3.2.6.3. Registration usages

A registration allows to obtain service implementations:

```
Service<T> myServiceImplementations = registrations.getServiceReference();
```

And it allows to unregister service implementations:

```
registrations.unregister();
```

3.2.7. Service registry

CDI-OSGi offers another way to deal with services: the service registry. It can be obtained in any bean bundle as a injected bean. The service registry allows developers to dynamically register service implementation, to obtain services and registrations.

3.2.7.1. First get the service registry

First get the service registry:

```
@Inject ServiceRegistry registry;
```

3.2.7.2. Register a service implementation

Register a service implementation:

```
registry.registerService(MyService.class,MyServiceImpl.class);
```

or

```
MyServiceImpl implementation = new MyServiceImpl();
registry.registerService(MyService.class,implementation);
```

It is possible to collect the corresponding registration:

```
Registration<MyService>                          registeredService                                =
 registry.registerService(MyService.class,MyServiceImpl.class);
```

3.2.7.3. Obtain a service implementations

Obtain a service implementations:

```
Service<MyService> services = registry.getServiceReference(MyService.class);
for (MyService service : services) {
    service.doSomething();
}
```

3.2.7.4. Obtain registrations

Obtain all registrations of a filtered or not filtered specified service:

```
Registration<?> registrations = registry.getRegistrations();
Registration<MyService> myRegistrations = registry.getRegistrations(MyService.class);
Registration<MyService>                        myFrenchRegistrations                             =
 registry.getRegistrations(MyService.class,"(lang=FR)");
```

3.2.8. The `OSGiServiceUnavailableException` exception

Because OSGi service are dynamic they might be unavailable at the time they should be used. On a service call if the targeted service isn't available a specific runtime exception is raised:

```
public class OSGiServiceUnavailableException extends RuntimeException {}
```

3.3. CDI-OSGi events

CDI-OSGi makes heavy usage of CDI events. These events allow CDI-OSGi to do its work. Events are important for the running of CDI-OSGi itself but they can also be used by client bean bundles to perform some operations or override some normal CDI-OSGi behavior.

This section shows the numerous events provided by CDI-OSGi and the ways to use them.

3.3.1. CDI container lifecycle events

Two events inform about the state of the CDI container of every bean bundle: `BundleContainerInitialized` and `BundleContainerShutdown` events.

The `BundleContext` from where the event comes is sent with these events so developers can retrieve useful information (such as the bundle owning the CDI container).

3.3.1.1. When the container is started: `BundleContainerInitialized` event

The `BundleContainerInitialized` event is fired every time a CDI container is initialized in a bean bundle. It point out that the CDI container is ready to manage the bean bundle with the CDI-OSGi features.

Here the way to listen this event:

```
public void onStartup(@Observes BundleContainerInitialized event) {
    BundleContext bundleContext = event.getBundleContext();
    Bundle firingBundle = bundleContext.getBundle();
}
```

3.3.1.2. When the container is stopped: `BundleContainerShutdown` event

The `BundleContainerShutdown` event is fired every time a CDI container is stopped in a bean bundle. It point out that the CDI container wil not manage the bean bundle with CDI-OSGi anymore.

Here the way to listen this event:

```
public void onShutdown(@Observes BundleContainerShutdown event) {
    BundleContext bundleContext = event.getBundleContext();
    Bundle firingBundle = bundleContext.getBundle();
}
```

3.3.2. Bundle lifecycle events

Ten events might be fired during a bundle lifecycle. They represent the possible states of a bundle and monitor the changes occurred in bundle lifecycles. These events carry the `Bundle` object from where the event comes so developers can retrieve useful information (such as the bundle id).

3.3.2.1. Listen all bundle events: `AbstractBundleEvent` event

CDI-OSGi provides a way to listen all bundle events in a single method: the `AbstractBundleEvent` abstract class. Every bundle lifecycle event is an `AbstractBundleEvent` and a method allows to retrieve the actual fired bundle event. All bundle states are listed in the `BundleEventType` enumeration. Then this event can be used as the corresponding bundle event.

Here the way to listen all bundle events and to retrieve the corresponding bundle state:

```
public void bundleChanged(@Observes AbstractBundleEvent event) {
    BundleEventType event.getType();
}
```

3.3.2.2. Listen a specific bundle event

Rather than listen all bundle events and then filter the intended, it is possible to choose the listened bundle event.

Here the way to listen a specific bundle event:

```
public void bindBundle(@Observes BundleInstalled event) {
    Bundle bundle = event.getBundle();
    long id = event.getBundleId();
    String symbolicName = event.getSymbolicName();
    Version version = event.getVersion();
}
```

All other bundle events could be listened the same way.

3.3.2.3. Filtering bundle events: `BundleName` and `BundleVersion` annotations

It is possible to filter the listened events depending on the bundles that are concerned. This filter might cover the name or the version of the bundles.

Here the way to filter bundle events on the bundle names:

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

or on the bundle versions:

```
public void bindBundle(@Observes @BundleVersion("4.2.1") BundleInstalled event) {
}
```

or on both bundle names and versions:

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

3.3.3. Service lifecyle events

Three events might be fired during a service lifecycle. They represent the possible states of a service and monitor the changes occurred in service lifecycles. These events carry the `BundleContext` object from where the event comes and the `ServiceReference` object corresponding to the service so developers can retrieve useful information (such as the registering bundle) and acces useful actions (such as register or unregister services).

3.3.3.1. Listen all service events: `AbstractServiceEvent` event

CDI-OSGi provides a way to listen all service events in a single method: the `AbstractServiceEvent` abstract class. Every service lifecycle event is an `AbstractServiceEvent` and a method allows to retrieve the actual fired service event. All service states are listed in the `ServiceEventType` enumeration. Then this event can be used as the corresponding service event.

Here the way to listen all service events and to retrieve the corresponding service state:

```
public void serviceEvent(@Observes AbstractServiceEvent event) {
    ServiceEventType event.getType();
}
```

3.3.3.2. When a service is published: `ServiceArrival` event

Rather than listen all service events and then filter the intended, it is possible to choose the listened service event.

Here the way to listen a specific service event:

```
public void bindService(@Observes ServiceArrival event) {
    ServiceReference serviceReference = event.getRef();
    Bundle registeringBundle = event.getRegisteringBundle();
}
```

All other bundle events could be listened the same way.

3.3.3.3. Filtering service events: `Specification` and `Filter` annotations

It is possible to filter the listened events depending on the services that are concerned. This filter might cover the service specification class or it might be a LDAP filter.

Here the way to filter service events on the specification class:

```
public void bindService(@Observes @Specification(MyService.class) ServiceArrival event) {
}
```

or using a LDAP filter:

```
public void bindService(@Observes @Specification(MyService.class) ServiceArrival event) {
}
```

or on both specification class and LDAP filter:

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

3.3.4. Application dependency validation events

Some bean bundle might declare required dependencies on services (using the `Required` annotation). CDI-OSGi fired events when these required dependencies are all validated or when at least one is invalidated. It allows application to be automatically started or stopped.

3.3.4.1. When all dependencies are validated: `Valid` event

An event is fired every time all the required dependencies are validated.

Here the way to listen this event:

```
public void validate(@Observes Valid event) {
}
```

3.3.4.2. When at least one dependency is invalidated: `Invalid` event

An event is fired every time at least one of the required dependencies is invalidated.

Here the way to listen this event:

```
public void invalidate(@Observes Invalid event) {
}
```

3.3.5. Intra and inter bundles communication events

CDI-OSGi provides a way to communicate within and between bean bundles. This communication occurs in a totally decoupled manner using CDI events.

3.3.5.1. Firing a bundle communication event: `InterBundleEvent` event

An `InterBundleEvent` is a message containing a object and transmitted using CDI event.

Here the way to fire such an event:

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

3.3.5.2. When a bundle communication is received: `InterBundleEvent` event

When an `InterBundleEvent` is fired it might be catch either within the bundle or in other bundles or both.

Here the way to receive a communication event from within the bundle:

```
public void listenAllEvents(@Observes InterBundleEvent event) {
}
```

Here the way to receive a communication event from another bundles:

```
public void listenAllEventsFromOtherBundles(@Observes @Sent InterBundleEvent event) {
}
```

3.3.5.3. Filtering communication events: `Specification` annotation

It is possible to filter the listened events depending on the message type. This filter may specify the type of message that is listened.

Here the way to filter communication events specifying a message type:

```
public void listenStringEventsFromOtherBundles(@Observes @Sent @Specification(MyMessage.class)
 InterBundleEventevent) {
}
```

Only the message with type `MyMessage` will be received.

3.4. OSGi utilities

CDI-OSGi provide some facilities for OSGi usage. It allows to obtain, by injection, some of the useful objects of the OSGi environment.

3.4.1. From the current bundle

Here a way to obtain the current bundle:

```
@Inject Bundle bundle;
```

Here a way to obtain the current bundle context:

```
@Inject BundleContext bundleContext;
```

Here a way to obtain all the current bundle headers:

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

or a particular header

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

Here a way to obtain a resource file from the current bundle:

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

3.4.2. From external bundle

Here a way to obtain a specified bundle from its name and version:

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

or just from its name:

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

Here a way to obtain all the specified bundle headers:

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

or a particular header:

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

3.5. CDI-OSGi, what else ?

Here there are some examples that concretely show what CDI-OSGi is avoiding to the developer:

3.5.1. Getting service references and instances

How to obtain a service available implementations list ?

In CDI-OSGI:

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

versus in classic OSGi:

```
ServiceReference references = bundleContext.getServiceReferences(MyService.class.getName());
//Current BundleContext must be known
```

How to obtain a service instance ?

In CDI-OSGi:

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

versus in classic OSGi:

```
ServiceReference reference = bundleContext.getServiceReference(MyService.class.getName());
MyService service = (MyService) bundleContext.getService(reference);
//Current BundleContext must be known
```

3.5.2. Publishing a service implementation

How to publish a new service implementation ?

In CDI-OSGi:

```
@Publish
public class MyServiceImpl implements MyService {

    @Override
    void doSomething() {...}
}
```

versus in classic OSGi:

```
public class MyServiceImpl implements MyService {

    @Override
    void doSomething() {...}
}

{ //in another class
      ServiceRegistration  registration  =  bundleContext.registerService(MyService.class,  new
 MyServiceImpl(), null);
}
//Current BundleContext must be known
```

3.5.3. Obtaining the `Bundle`s and `BundleContext`s

How to obtain the current `Bundle` ?

In CDI-OSGi:

```
@Inject Bundle currentBundle;
```

versus in classic OSGi:

```
Bundle currentBundle = FrameworkUtil.getBundle(this.getClass());
```

How to obtain the current `BundleContext` ?

In CDI-OSGi:

```
@Inject BundleContext bundleContext;
```

versus in classic OSGi:

```
BundleContext bundleContext = FrameworkUtil.getBundle(this.getClass()).getBundleContext();
```

How to obtain a specified `Bundle` and `BundleContext` ?

In CDI-OSGi:

```
@Inject @BundleNamer("symbolicName") @BundleVersion("version") Bundle specifiedBundle;
@Inject @BundleNamer("symbolicName") @BundleVersion("version") BundleContext
 specifiedBundleContext;
```

versus in classic OSGi:

```
ServiceReference reference = bundleContext.getServiceReference(PackageAdmin.class.getName());
PackageAdmin packageAdmin = bundleContext.getService(reference);
Bundle specifiedBundle = packageAdmin.getBundles("symbolicName","version")[0];
BundleContext specifiedBundleContext = specifiedBundle.getBundleContext();
//Current BundleContext must be known
```

Chapter 4.

Weld-OSGi implementation