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 deploy in a CDI environment. It enables all CDI functionality in that bean archive.

1.1.2. OSGi bundle

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

A bundle may be deploy in an OSGi environment. It enables all OSGi functionality 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 deploy in an OSGi environment and then be managed by the CDI-OSGI extension bundle. It enables both OSGi and CDI functionality for that bundle. CDI functionality is provided by the implementation bundle with which the extension bundle works.

It is the form taken by OSGi-CDI application component bundles.

1.1.4. APIs bundle

The APIs bundle is a bundle that only provides APIs used by CDI-OSGi. The two exposed APIS are extension API and integration API. They are needed by the extension, the implementation bundle and business bundles.

It is library bundle that is used by every bundles in a CDI-OSGi application.

1.1.5. Extension bundle

The extension bundle is a bundle that manage all the bean bundles of the current OSGi application. It manages all CDI relevant functionality and communications exposed in extension API. For that purpose it requires a implementation bundle conform to integration API.

It is the form that takes OSGi-CDI master bundle.

1.1.6. Implementation bundle

The implementation bundle is a bundle that is implementation of a CDI compliant container in a CDI-OSGi environment. It provides to the extension bundle the containers for managed bean bundles. It should be seen as a glu between CDI vendor-specific APIs and CDI-OSGi integration API contract.

Weld-OSGi is one implementation bundle.

1.1.7. 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 faster way to interact with the OSGi Framework.

It addresses the OSGi complexity about services management using CDI specification (JSR-299). Thus it provides a CDI OSGi extension with injection utilities for the OSGi environment.

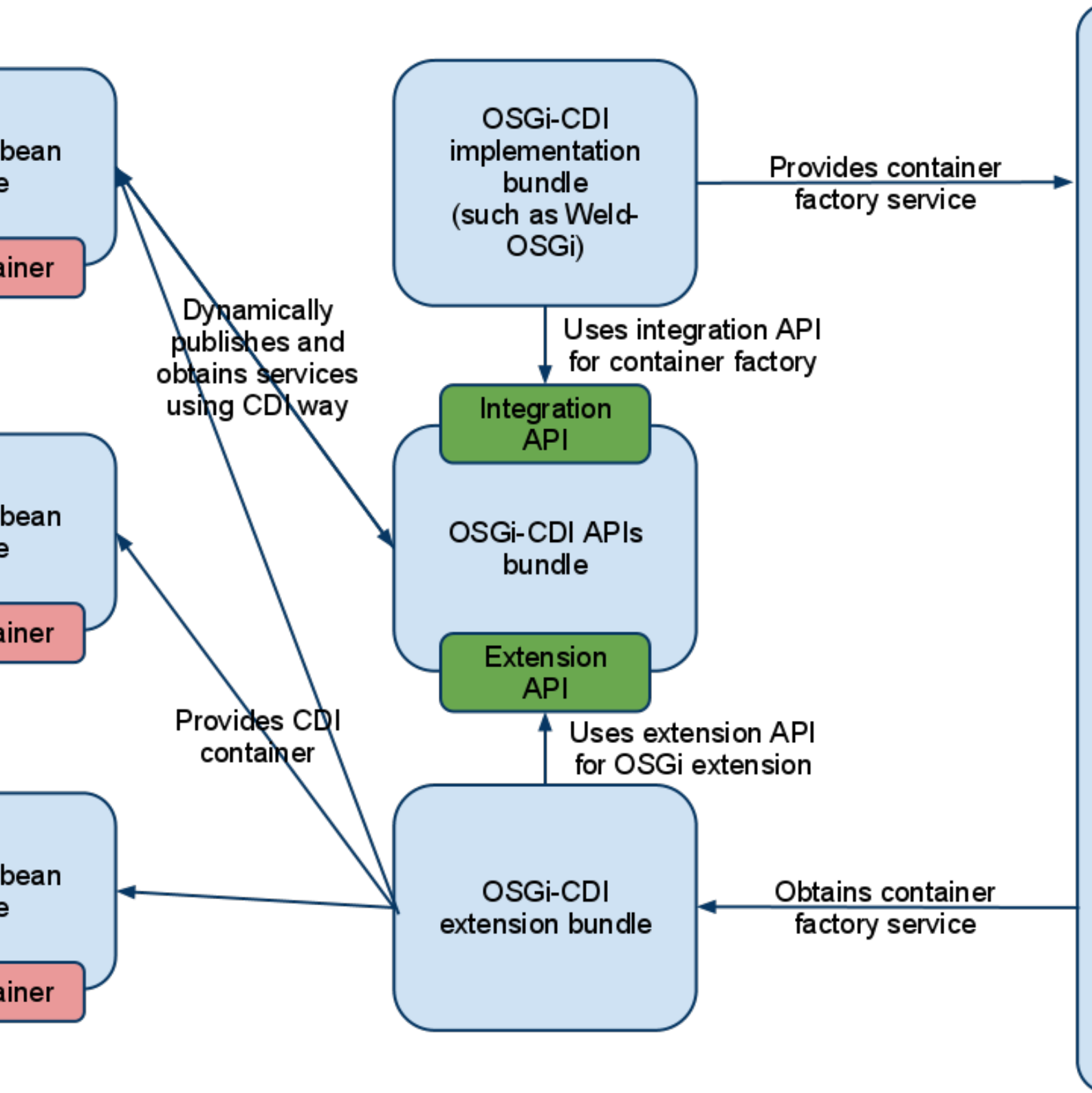
Moreover an integration with any CDI implementation, such as Weld, is possible through a well-defined bootstrapping API.

1.2.2. Weld-OSGi

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

But Weld-OSGi is more than a simple implementation of CDI-OSGi. It provides other utilities for OSGi environment facilitation. It comes with a complete documentation and sample applications.

1.2.3. Organization and interactions between bundles



This diagram shows the organization of bundles and how they act with each other in a CDI-OSGi application.

Figure 1.1. Organization and interactions between bundles

1.3. What about other frameworks

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

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

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

1.4. 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 specification for core functionality usages and CDI container integration.
- The Weld-OSGi specification for special functionality usages and sample application review.

Organization of CDI-OSGi

2.1. APIs bundle, extension bundle and implementation bundle

CDI-OSGi is composed of three bundles:

- The APIs bundle that provides the APIs used to define both utilities provided and hooking up system with implementation bundle,
- The extension bundle that provides CDI-OSGi functionalities for bean bundles by managing them,
- An implementation bundle provides CDI functionalities usable by the extension bundle through an OSGi service.

2.1.1. Extention API versus integration API

The APIs bundle exposed all packages that could be needed by developers; both for client application using CDI-OSGi and for CDI compliant container integration. It may be use as a dependency for bean bundles, extension bundle and implementation bundle.

CDI-OSGi APIs bundle provides two distinct APIs:

- The extension API that describes all the new functionality provided by CDI-OSGi in the OSGi environment.
- The integration API that allows CDI compliant container to be used with CDI-OSGi.

These two APIs are more described below.

2.1.1.1. Extension API

The extension API defines all the utilities 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 realize its client bean bundle. It defines the programming model of CDI-OSGi client bundle. Mostly it is about publishing and consuming injectable services in a CDI way.

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

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

2.1.1.2. 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 implementation bundle.

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

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

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

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

2.1.2. 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 at the begining of a CDI-OSGi application. It requests the APIs bundle as a dependency.

The extension bundle is the heart of CDI-OSGi application. Once it is started, provided that it finds a started implementation 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, it just need to be started at the beginning of the application, then everything is transparent to the user. Client bean bundles do not have to do anything in order to use CDI-OSGi functionality.

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

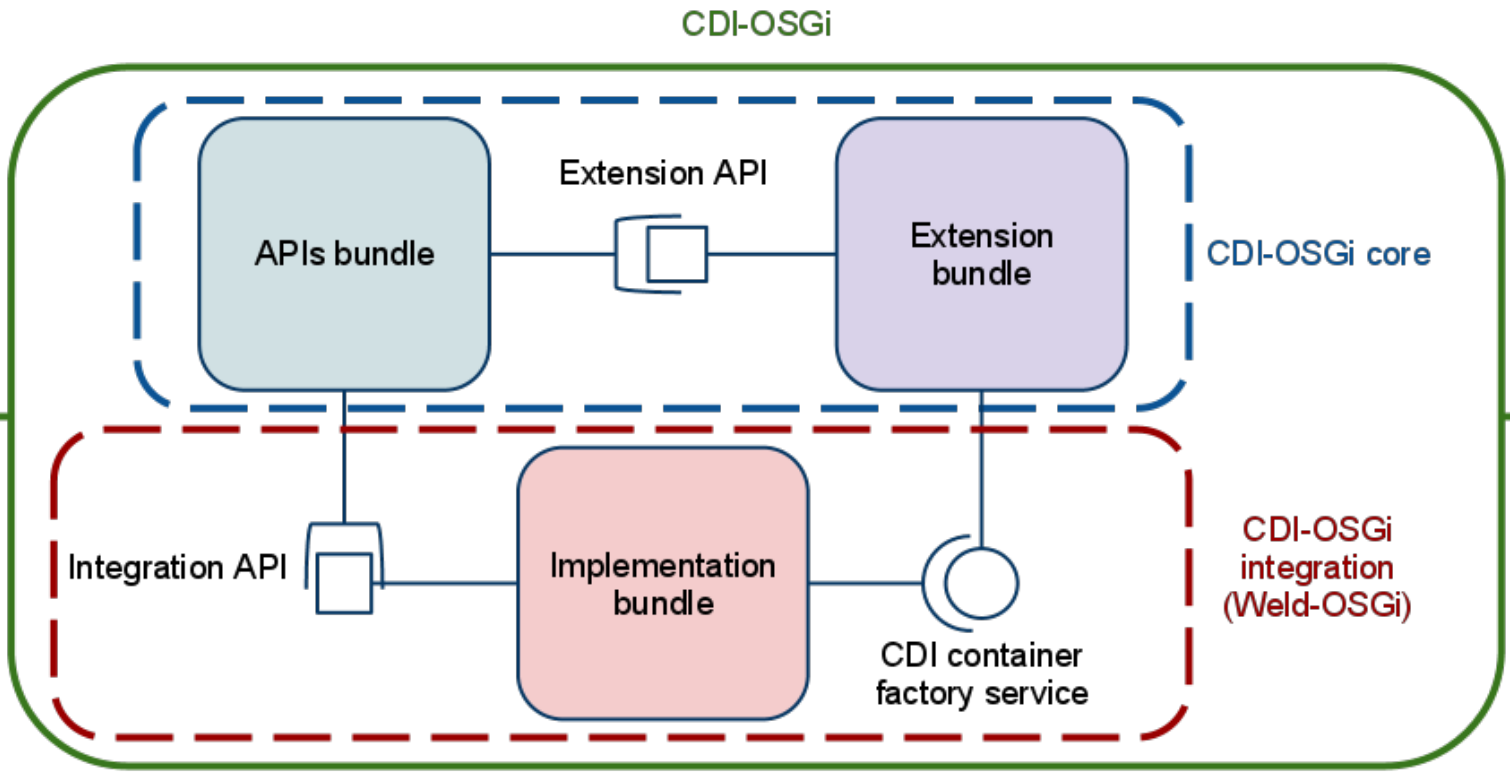
So this is where the magic happens and where OSGi applications become much more simple.

2.1.3. Implementation bundle: choose a CDI compliant container

The implementation bundle is responsible for providing CDI compliant containers to the extension bundle. It may be started with the extension bundle and publish the right service. It request the APIs bundle as a dependency.

It is an implementation of the integration API, thus it may provide a CDI container factory service.

2.1.4. CDI-OSGi bundles organization diagram



This diagram represents the internal organization of CDI-OSGi bundles. The coupling between the three bundles is represented as well as what is exposed to bean bundles.

Figure 2.1. CDI-OSGI bundles organization

The diagram above shows how CDI-OSGi bundles are linked together. The blue part is the core of OSGi and is provided. The red part is a required but interchangeable bundle.

All together these three bundle make CDI-OSGi. Then CDI-OSGi functionalities are exposed to bean bundles and an OSGi extension is provided in order to manage these bean bundles.

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

2.2.1. The extender pattern

CDI-OSGi provides an extension to OSGi as an extender OSGi 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.

2.2.2. CDI-OSGi features

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

- Complete integration in 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 using annotation, 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.

2.2.3. The interfaces

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

2.2.3.1. The `Service` interface

```
public interface Service<T> extends Instance<T> {
    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 informations about the contained service implementations.

OSGi services should not be subtyped.

2.2.3.2. 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 a injectable service in the service resgistry. Its fonctionnement 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 `Qualfiers` or LDAP filters,
- Iterate through these service registrations,
- Obtain the service implementations list as a `Service<T>`,
- Get the number of registration (i.e the number of registered service implementations).

OSGi services should not be subtyped.

2.2.3.3. 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.

2.2.3.4. 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);
    <T> Provider<T> newTypeInstance(Class<T> unmanagedType);
}
```

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>`,
- Obtain an instance provider for a specified type.

2.2.4. The events

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

2.2.4.1. The `BundleContainerInitialized` and `BundleContainerShutDown` events

```
public class BundleContainerInitialized {

    private BundleContext bundleContext;

    public BundleContainerInitialized(final BundleContext context) {
        this.bundleContext = context;
    }

    public BundleContext getBundleContext() {
        return bundleContext;
    }
}
```

```
public class BundleContainerShutdown {

    private BundleContext bundleContext;

    public BundleContainerShutdown(final BundleContext context) {
        this.bundleContext = context;
    }

    public BundleContext getBundleContext() {
        return bundleContext;
    }
}
```

These two events advise about the state of bean bundles CDI compliant container. They are fired respectively when a container has finished its initialization and when a container begin to shutdown. They expose the coresponding bean bundle `BundleContext`.

2.2.4.2. The `AbstractBundleEvent` and `AbstractServiceEvent` events

```
public abstract class AbstractBundleEvent {

    public static enum EventType {
        INSTALLED,LAZY_ACTIVATION,RESOLVED,STARTED,STARTING,
        STOPPED,STOPPING,UNINSTALLED,UNRESOLVED,UPDATED,
    }

    private final Bundle bundle;

    public AbstractBundleEvent(Bundle bundle) {
        this.bundle = bundle;
    }

    public abstract EventType getType();

    public long getBundleId() {
        return bundle.getBundleId();
    }

    public String getSymbolicName() {
        return bundle.getSymbolicName();
    }

    public Version getVersion() {
        return bundle.getVersion();
    }

    public Bundle getBundle() {
        return bundle;
    }
}
```

```
public abstract class AbstractServiceEvent {
```

```
public static enum EventType {
    SERVICE_ARRIVAL, SERVICE_DEPARTURE, SERVICE_CHANGED
}

private final ServiceReference ref;
private final BundleContext context;
private List<String> classesNames;
private List<Class<?>> classes;
private Map<Class, Boolean> assignable = new HashMap<Class, Boolean>();

public AbstractServiceEvent(
    ServiceReference ref, BundleContext context) {
    this.ref = ref;
    this.context = context;
}

public abstract EventType eventType();

public ServiceReference getRef() {
    return ref;
}

public <T> TypedService<T> type(Class<T> type) {
    if (isTyped(type)) {
        return TypedService.create(type, context, ref);
    } else {
        throw new RuntimeException("the type " + type
            + " isn't supported for the service. Supported types are "
            + getServiceClasses());
    }
}

public Object getService() {
    return context.getService(ref);
}

public boolean ungetService() {
    return context.ungetService(ref);
}

public boolean isTyped(Class<?> type) {
    boolean typed = false;
    if (!assignable.containsKey(type)) {
        for (Class clazz : getServiceClasses()) {
            if (type.isAssignableFrom(clazz)) {
                typed = true;
                break;
            }
        }
        assignable.put(type, typed);
    }
    return assignable.get(type);
}

public Bundle getRegisteringBundle() {
    return ref.getBundle();
}
```

```
    public List<String> getServiceClassNames() {
        if (classesNames == null) {
            classesNames = Arrays.asList((String[])
            ref.getProperty(Constants.OBJECTCLASS));
        }
        return classesNames;
    }

    public List<Class<?>> getServiceClasses() {
        if (classes == null) {
            classes = new ArrayList<Class<?>>();
            for (String className : getServiceClassNames()) {
                try {
                    classes.add(getClass().getClassLoader().loadClass(className));
                } catch (ClassNotFoundException ex) {
                return null;
                }
            }
        }
        return classes;
    }

    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) {
            this.context = context;
            this.ref = ref;
            this.type = type;
        }

       static <T> TypedService<T> create(Class<T> type, BundleContext context, ServiceReference
 ref) {
            return new TypedService<T>(context, ref, type);
        }

        public T getService() {
            return type.cast(context.getService(ref));
        }

        public boolean ungetService() {
            return context.ungetService(ref);
        }
    }
}
```

These two abstract classes represent the information that bundle and service events may carry when they are fired. They also allows to listen for all bundle or service events in one operation.

2.2.4.3. The `BundleEvents` events

It regroups all the events about bundle life-cycle.

```
public class BundleEvents {

    public static class BundleInstalled extends AbstractBundleEvent {
        ...
    }

    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 {
        ...
    }
}
```

```
public static class BundleInstalled extends AbstractBundleEvent {

    public BundleInstalled(Bundle bundle) {
        super(bundle);
    }

    @Override
    public EventType getType() {
        return EventType.INSTALLED;
    }
}
```

```
public static class BundleLazyActivation extends AbstractBundleEvent {

    public BundleLazyActivation(Bundle bundle) {
        super(bundle);
    }

    @Override
    public EventType getType() {
        return EventType.LAZY_ACTIVATION;
    }
}
```

```
public static class BundleResolved extends AbstractBundleEvent {

    public BundleResolved(Bundle bundle) {
        super(bundle);
    }

    @Override
    public EventType getType() {
        return EventType.RESOLVED;
    }
}
```

```
public static class BundleStarted extends AbstractBundleEvent {

    public BundleStarted(Bundle bundle) {
        super(bundle);
    }

    @Override
    public EventType getType() {
        return EventType.STARTED;
    }
}
```

```
public static class BundleStarting extends AbstractBundleEvent {

    public BundleStarting(Bundle bundle) {
        super(bundle);
    }

    @Override
    public EventType getType() {
        return EventType.STARTING;
    }
}
```

```
public static class BundleStopped extends AbstractBundleEvent {

    public BundleStopped(Bundle bundle) {
        super(bundle);
    }

    @Override
    public EventType getType() {
        return EventType.STOPPED;
    }
}
```

```
public static class BundleStopping extends AbstractBundleEvent {

    public BundleStopping(Bundle bundle) {
        super(bundle);
    }

    @Override
    public EventType getType() {
        return EventType.STOPPING;
    }
}
```

```
public static class BundleUninstalled extends AbstractBundleEvent {

    public BundleUninstalled(Bundle bundle) {
        super(bundle);
    }

    @Override
    public EventType getType() {
        return EventType.UNINSTALLED;
    }
}
```

```
public static class BundleUnresolved extends AbstractBundleEvent {

    public BundleUnresolved(Bundle bundle) {
        super(bundle);
    }

    @Override
    public EventType getType() {
        return EventType.UNRESOLVED;
    }
}
```

```
public static class BundleUpdated extends AbstractBundleEvent {

    public BundleUpdated(Bundle bundle) {
        super(bundle);
    }

    @Override
    public EventType getType() {
        return EventType.UPDATED;
    }
}
```

These events are automatically fired by CDI-OSGi and transmitted to other bundles when the corresponding life-cycle step occurs in a bean bundle.

2.2.4.4. The `ServiceArrival`, `ServiceChanged` and `ServiceDeparture` events

```
public class ServiceArrival extends AbstractServiceEvent {

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

    @Override
    public EventType eventType() {
        return EventType.SERVICE_ARRIVAL;
    }
}
```

```
public class ServiceChanged extends AbstractServiceEvent {

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

    @Override
    public EventType eventType() {
        return EventType.SERVICE_CHANGED;
    }
}
```

```
public class ServiceDeparture extends AbstractServiceEvent {

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

    @Override
    public EventType eventType() {
        return EventType.SERVICE_DEPARTURE;
```

```
    }
}
```

These events are automatically fired by CDI-OSGi and transmitted to other bundles when the corresponding life-cycle step occurs in a OSGi service.

2.2.4.5. The `Valid` and `Invalid` events

```
public class Valid {
}
```

```
public class Invalid {
}
```

These two events notify about the state of bean bundles dependency resolution. They are automatically fired by CDI-OSGi when all required service are available or when at least one required service is unavailable.

2.2.5. The annotations

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

2.2.5.1. The `OSGiBundle` annotation

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

This annotation qualifies an injection point that represents a bundle.

2.2.5.2. The `BundleDataFile`, `BundleHeader`, `BundleHeaders`, `BundleName` and `BundleVersion` annotations

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

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

```
    @Nonbinding String value();
}
```

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

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

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

These annotations qualify an injection point that represents bundle relative information.

2.2.5.3. 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 in the OSGi service registry.

2.2.5.4. The `Publish` annotation

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

This annotation notice that this type is an OSGi service that may be automatically published in the OSGi service registry.

2.2.5.5. The `Required` annotation

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

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

2.2.5.6. The `Filter` annotation

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

This annotation qualifies an injection point with an LDAP filter.

2.2.5.7. The `Specification` annotation

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

This annotation qualifies an injection point by specifying the expected specification class.

2.2.5.8. The `Sent` annotation

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

This annotation qualifies an injection point that represents a incoming inter bundle communication.

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 implementation 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 describe what needs the extension bundle to work (and therefore what may provide the implementation bundle) and what provides the extension bundle (and therefore what should use the implementation bundle).

2.3.1.1. The `BundleContainer` and `BundleContainers` interfaces

```
public interface BundleContainer {
    void fire(InterBundleEvent event);
}
```

```
public interface BundleContainers {
    Collection<BundleContainer> getContainers();
}
```

These interfaces represent the CDI compliant containers used by bean bundle. Such container may be accessible in the OSGi application to provide an event notification system.

Multiple containers can be manipulated by the `BundleContainers` interface.

2.3.1.2. The `BundleContainerFactory` interface

```
public interface BundleContainerFactory {

    Class<? extends BundleContainerFactory> delegateClass();
    String getID();
    Set<String> getContractBlacklist()
    BundleContainer container(Bundle bundle);
}
```

This the how the extension bundle require a CDI compliant container from the implementation bundle. This latter should publish an OSGi service that complies with this interface.

2.3.2. Embedded mode

TODO How does it work ? Should we provide something ? Should business bundles use the integration API ? A template activator ?

2.4. An orchestrator : the extension bundle

2.4.1. 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 implementation bundle

The implementation bundle is is necessary in a CDI-OSGi application. This section explains the part the implementation bundle plays but does not enter in specifics because it depends on the CDI vendor 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. How to make a bundle or a bean archive a bean bundle

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

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

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

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

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

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

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

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

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

How to make OSGi easy peasy

3.1. CDI usage in bean bundles

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

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

3.2. 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 registration.

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

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

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

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

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 implementation bundle.

3.2.2.2. Injection using programmatic lookup

Instead of obtain directly a service instance it is possible to choose between service implementations and instanciate one at runtime. The interface `Service<T>` works as a service instance producer: it retreives all the coresponding (to the service parametrized type) service implementations and allows to get an instances 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 a contextual instance of the service.

`Service<T>` extends CDI `Instance<T>` so the behavior is similar except that the injection process is managed by the extension bundle instead of implementation bundle. So the available implementations are searched dynanically 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 declaration, except that services are not register into the bean manager but into the service registry like regular service registry.

3.2.4. Clearly specify a service implementation

There might be multiple implementations of the same service. It is possible to qualify specifities of 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(Languages.EN)
@Country(Countries.US)
@ApplicationScoped
public class MyServiceImpl implements MyService {

    @Override
    public void doSomething() {
    }
}

@Inject Service<MyService> @Lang(Languages.EN) @Country(Countries.US) 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

TODO Is @AnyQualifier similar with @Property(name="anyqualifier", value="") ? And is @Property(name="anyqualifier", value="anyValue") similar with @AnyQualifier("anyValue") ? To any qualifier correspond a generated filter ?

Consider this code, will it work ? (it'll be cool!)

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

    @Override
    public void doSomething() {
    }
}

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

And this ?

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

    @Override
    public void doSomething() {
    }
}

@Inject @Lang(Languages.EN) @Country(Countries.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` is assumed, a new instance will be create for every injection.

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 coresponding `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 enregister 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 bundle as a regular service, using OSGi or CDI-OSGi ways. The service registry allows developers to dynamically register service implementation, to obtain services and registrations.

3.2.7.1. First get the service registry

TODO Is the service registry a bean or a service ? If it is a service it could use by regular bundle, couldn't it ?

First get the service registry:

```
@Inject @OSGiService ServiceRegistry registry;
```

Or in regular bundle:

```
ServiceReference                                reference                                =
 bundleContext.getServiceReference(ServiceRegistry.class.getName());
ServiceRegistry registry = (ServiceRegistry) bundleContext.getService(reference);
```

3.2.7.2. Register a service implementation

Register a service implementation:

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

or

```
@Inject MyServiceImpl implementation;
registry.registerService(MyService.class,implementation);
```

Here the scope of the service is ??? (Dependent ?)

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. Distinctions between CDI-OSGi service injection and CDI bean injection

All the way it is said that CDI-OSGi service injection is similar with classic CDI bean injection. That is true, but there some important distinctions to make. CDI users will quickly find their bearings in CDI-OSGi and that is the point because CDI way is much simplier that OSGi mechanisms (so CDI-OSGi is also a good thing for newcomers in both CDI and OSGi worlds!).

The main difference is about the dynamism of OSGi services. While a bean injection point should be satisfy at the start of the CDI container, service injection may be satisfy only at runtime.

So what ? TODO Ambigous and unsatisfied dependency managment and other stuffs that vary from classic CDI.

3.2.8.1. 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 {}
```

3.2.8.2. And everything else ?

If it is not indicated otherwise assume that the comportment is the same as CDI or OSGi without CDI-OSGi.

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 use by client bean bundles to perform some operations or overide 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. When a bundle is installed: `BundleInstalled` event

An event is fired every time a bundle is install in the OSGi environment.

Here the way to listen this event:

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

3.3.2.3. When a bundle is uninstalled: `BundleUninstalled` event

An event is fired every time a bundle is uninstall from the OSGi environment.

Here the way to listen this event:

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

3.3.2.4. When a bundle is lazy activated: `BundleLazyActivation` event

An event is fired every time a bundle is lazy activated in the OSGi environment.

Here the way to listen this event:

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

3.3.2.5. When a bundle is resolved: `BundleResolved` event

An event is fired every time a bundle is resolved in the OSGi environment.

Here the way to listen this event:

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

3.3.2.6. When a bundle is unresolved: `BundleUnresolved` event

An event is fired every time a bundle is unresolved in the OSGi environment.

Here the way to listen this event:

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

3.3.2.7. When a bundle is updated: `BundleUpdated` event

An event is fired every time a bundle is updated in the OSGi environment.

Here the way to listen this event:

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

3.3.2.8. When a bundle is started: `BundleStarted` event

An event is fired every time a bundle is started in the OSGi environment.

Here the way to listen this event:

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

3.3.2.9. When a bundle is starting: `BundleStrating` event

An event is fired every time a bundle is starting in the OSGi environment.

Here the way to listen this event:

```
public void startingBundle(@Observes BundleStarting event) {
    Bunde bundle = event.getBundle();
    long id = event.getBundleId();
    String symbolicName = event.getSymbolicName();
```

```
    Version version = event.getVersion();
}
```

3.3.2.10. When a bundle is stopped: `BundleStopped` event

An event is fired every time a bundle is stopped in the OSGi environment.

Here the way to listen this event:

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

3.3.2.11. When a bundle is stopping: `BundleStopping` event

An event is fired every time a bundle is stopping in the OSGi environment.

Here the way to listen this event:

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

3.3.2.12. 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 corespondign to the service so developers can retrieve useful information (such as the registering bundle) and acces useful actions (such as register or enregister 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

An event is fired every time a service is published in the OSGi environment.

Here the way to listen this event:

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

3.3.3.3. When a service is unpublished: `ServiceDeparture` event

An event is fired every time a service is unpublished in the OSGi environment.

Here the way to listen this event:

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

3.3.3.4. When a service is changed: `ServiceChange` event

An event is fired every time a service is changed in the OSGi environment.

Here the way to listen this event:

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

3.3.3.5. 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 specificaton 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 stoped.

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 transmited 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());
```

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);
```

3.5.2. Obtaining the current bundle context and bundle

How to obtain the current bundle context ?

In CDI-OSGi:

```
@Inject BundleContext bundleContext;
```

versus in classic OSGi:

```
//the bundle context mush have been saved in the activator class. Impossible to get for a
 bundle with no activator ???
```

How to obtain the current bundle ?

In CDI-OSGi:

```
@Inject Bundle currentBundle;
```

versus in classic OSGi:

```
//assuming that the bundleContext is known, impossible if not ???
ServiceReference reference = bundleContext.getServiceReference(PackageAdmin.class.getName());
PackageAdmin packageAdmin = bundleContext.getService(reference);
Bundle currentBundle = packageAdmin.getBundle(this);
```

3.5.3. TODO other example

TODO

Weld-OSGi implementation

Getting Started

Samples