Weld - OSGi integration

The power of modularity for CDI

Mathieu Ancelin

<mathieu.ancelin@serli.com>

Matthieu Clochard

<matthieu.clochard@serli.com>

Weld-OSGi user documentation

1.1. What is this documentation

This documentation allows new users to quickly setup and try Weld-OSGi. It covers most of Weld-OSGi features with lot of practical code examples allowing users to directly get their hands dirty.

The final chapter offers a complete walk-through of a sample application showing the power of Weld-OSGi.

1.2. What is Weld-OSGi

Weld-OSGi is a framework that allows CDI usages within an OSGi environment. It is an extension of Weld providing CDI for OSGi environment (like Weld-SE for Java SE environment) but it also brings numerous pieces of OSGi facilitation.

It is a set of five bundles that extends the behavior of client bundles once they are deployed in an OSGi framework.

Weld-OSGi provides:

- CDI usages in client bundles by obtaining a Weld container
- OSGi service layer support for CDI injection
- CDI event notification mechanism for OSGi event and inter bundle communication
- OSGi utilities facilitation using CDI injection

1.3. Naming convention

An OSGi bundle is a regular bundle that is not managed by Weld-OSGi.

A bean bundle is an OSGi bundle managed by Weld-OSGi. It has one or more valid `META-INF/bean.xml` marker files.

A bundle refers indifferently to an OSGi bundle or a bean bundle.

The extension bundle is the Weld-OSGi bundle that manages the bean bundles.

The implementation bundle is the Weld-OSGi bundle that provide Weld container to the extension bundle.

The API bundles are the Weld-OSGi bundles that provide all necessary API in order to use Weld-OSGi in bean bundles.

1.4. What tools are used in this documentation

All examples in this user documentation are made with these tools:

- Maven for managing dependencies at compile time and building the bundles
- Java 6 for the java sources
- Felix framework as the OSGi environment

All examples were tested on a Linux system with the latest version of each tools at the moment of redaction. Version may have evolved so little ajustement might be needed (like for bundle or package versions).

1.5. References and other documentations

This documentation assumes that you have at least an overview of OSGi and CDI. But for more advanced usages of Weld-OSGi information can be found in other documents.

The specification of OSGi can be found here: *http://www.osgi.org/Specifications/HomePage*.

The documentation of Weld can be found here: *http://docs.jboss.org/weld/reference/latest/en-US/html/*.

The complete specification of Weld-OSGi can be found here: *http://mathieuancelin.github.com/weld-osgi/html/index.html*.

Getting started with Weld-OSGi

In this chapter you will see:

- how to setup the environment in order to use Weld-OSGi
- how to build your first bean bundle using CDI in OSGi
- how to build a more complex example with advanced CDI usage

2.1. Setting up your environment

Weld-OSGi may run in an OSGi environment, you should setup one:

- Download the last version of Felix framework here: *http://felix.apache.org/site/downloads.cgi*
- Extract the files anywhere you want, this location will be called the Felix home
- Open a terminal in the Felix home and run the Felix framework with `java -jar bin/felix.jar`
- You should now have a running OSGi environment with the Felix framework prompt:

```
____________________________
Welcome to Apache Felix Gogo

g!
```

Try your Felix installation a bit with these three main command:

- The `lb` command, that lists all the bundles in the OSGi environment, with their id and state (currently you only have the Felix framework utility bundles)

```
g! lb
START LEVEL 1
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (3.2.2)
    1|Active     |    1|Apache Felix Bundle Repository (1.6.2)
    2|Active     |    1|Apache Felix Gogo Command (0.8.0)
    3|Active     |    1|Apache Felix Gogo Runtime (0.8.0)
    4|Active     |    1|Apache Felix Gogo Shell (0.8.0)
g!
```

- The `stop <bundle_id>` command, that stops the corresponding bundle

```
g! stop 1
g! lb
START LEVEL 1
```

```
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (3.2.2)
    1|Resolved   |    1|Apache Felix Bundle Repository (1.6.2)
    2|Active     |    1|Apache Felix Gogo Command (0.8.0)
    3|Active     |    1|Apache Felix Gogo Runtime (0.8.0)
    4|Active     |    1|Apache Felix Gogo Shell (0.8.0)
g!
```

- The `start <bundle_id>` command, that starts the corresponding bundle

```
g! start 1
g! lb
START LEVEL 1
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (3.2.2)
    1|Active     |    1|Apache Felix Bundle Repository (1.6.2)
    2|Active     |    1|Apache Felix Gogo Command (0.8.0)
    3|Active     |    1|Apache Felix Gogo Runtime (0.8.0)
    4|Active     |    1|Apache Felix Gogo Shell (0.8.0)
g!
```

Felix framework currently auto start all bundles, although it is a good thing for utility bundles it may be inconvenient with application bundles. You should configure the Felix framework by editing the Felix home `conf/config.properties` file:

- Unable the auto start option by replacing the line

```
felix.auto.deploy.action=install,start
```

  by

```
#felix.auto.deploy.action=install,start
```

- Ask for auto start of utility bundle by replacing the line

```
#felix.auto.start.1=
```

  by

```
felix.auto.start.1= file:bundle/org.apache.felix.bundlerepository-1.6.2.jar \
file:bundle/org.apache.felix.gogo.command-0.8.0.jar \
file:bundle/org.apache.felix.gogo.runtime-0.8.0.jar \
file:bundle/org.apache.felix.gogo.shell-0.8.0.jar
```

Install Weld-OSGi in the Felix framework:

- Add some bundle in the Felix framework by simply drop the corresponding `jar` files into the `bundle` directory of Felix home
- Download the last version of Weld-OSGi here: *TBA* []
- Extract the five bundles into the `bundle` directory of Felix home

> **Get Felix up to date**
>
> Felix framework keeps a cache of old bundles and actions you performed. You may remove the Felix home `felix-cache` directory to avoid older versions of bundle and configuration to be taken into account when:
>
> - You add/remove bundles from `bundle` directory
> - You modify the `conf/config.properties` file
> - You restart the Felix framework after running some commands

- Update the Felix framework configuration file in order to auto install the Weld-OSGi bundle by replacing the line

```
#felix.auto.install.1=
```

by

```
felix.auto.install.1= file:bundle/weld-osgi-core-api-1.0-SNAPSHOT.jar \
file:bundle/weld-osgi-core-extension-1.0-SNAPSHOT.jar \
file:bundle/weld-osgi-core-spi-1.0-SNAPSHOT.jar \
file:bundle/weld-osgi-core-mandatory-1.0-SNAPSHOT.jar \
file:bundle/weld-osgi-core-integration-1.0-SNAPSHOT.jar
```

- Check the bundle are in the OSGi environment by starting the Felix framework

```
____________________________
Welcome to Apache Felix Gogo

g! lb
START LEVEL 1
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (3.2.2)
    1|Active     |    1|Apache Felix Bundle Repository (1.6.2)
    2|Active     |    1|Apache Felix Gogo Command (0.8.0)
    3|Active     |    1|Apache Felix Gogo Runtime (0.8.0)
    4|Active     |    1|Apache Felix Gogo Shell (0.8.0)
    5|Installed  |    1|Weld-OSGi :: Core :: Extension API (1.0.0.SNAPSHOT)
    6|Installed  |    1|Weld-OSGi :: Core :: Extension Impl (1.0.0.SNAPSHOT)
    7|Installed  |    1|Weld-OSGi :: Core :: Integration API (1.0.0.SNAPSHOT)
    8|Installed  |    1|Weld-OSGi :: Core :: Mandatory (1.0.0.SNAPSHOT)
    9|Installed  |    1|Weld-OSGi :: Implementation :: Weld Integration (1.0.0.SNAPSHOT)
g!
```

- Run the bundles `Weld-OSGi :: Core :: Extension Impl` and `Weld-OSGi :: Implementation :: Weld Integration` in order to get the Weld-OSGi running in the Felix framework

```
g! start 6
g! start 9
g! lb
START LEVEL 1
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (3.2.2)
    1|Active     |    1|Apache Felix Bundle Repository (1.6.2)
    2|Active     |    1|Apache Felix Gogo Command (0.8.0)
    3|Active     |    1|Apache Felix Gogo Runtime (0.8.0)
    4|Active     |    1|Apache Felix Gogo Shell (0.8.0)
    5|Resolved   |    1|Weld-OSGi :: Core :: Extension API (1.0.0.SNAPSHOT)
    6|Active     |    1|Weld-OSGi :: Core :: Extension Impl (1.0.0.SNAPSHOT)
    7|Resolved   |    1|Weld-OSGi :: Core :: Integration API (1.0.0.SNAPSHOT)
    8|Resolved   |    1|Weld-OSGi :: Core :: Mandatory (1.0.0.SNAPSHOT)
    9|Active     |    1|Weld-OSGi :: Implementation :: Weld Integration (1.0.0.SNAPSHOT)
g!
```

It would be easier if these two bundles were auto started with Felix framework.

- Modify the Felix configuration file to do so

```
felix.auto.install.1= file:bundle/weld-osgi-core-api-1.0-SNAPSHOT.jar \
file:bundle/weld-osgi-core-spi-1.0-SNAPSHOT.jar \
file:bundle/weld-osgi-core-mandatory-1.0-SNAPSHOT.jar
...
felix.auto.start.1= file:bundle/org.apache.felix.bundlerepository-1.6.2.jar \
file:bundle/org.apache.felix.gogo.command-0.8.0.jar \
file:bundle/org.apache.felix.gogo.runtime-0.8.0.jar \
file:bundle/org.apache.felix.gogo.shell-0.8.0.jar \
file:bundle/weld-osgi-core-extension-1.0-SNAPSHOT.jar \
file:bundle/weld-osgi-core-integration-1.0-SNAPSHOT.jar
```

- Try this new configuration

```
____________________________
Welcome to Apache Felix Gogo

g! lb
START LEVEL 1
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (3.2.2)
    1|Active     |    1|Apache Felix Bundle Repository (1.6.2)
    2|Active     |    1|Apache Felix Gogo Command (0.8.0)
    3|Active     |    1|Apache Felix Gogo Runtime (0.8.0)
    4|Active     |    1|Apache Felix Gogo Shell (0.8.0)
    5|Active     |    1|Weld-OSGi :: Core :: Extension Impl (1.0.0.SNAPSHOT)
    6|Active     |    1|Weld-OSGi :: Implementation :: Weld Integration (1.0.0.SNAPSHOT)
    7|Resolved   |    1|Weld-OSGi :: Core :: Extension API (1.0.0.SNAPSHOT)
    8|Resolved   |    1|Weld-OSGi :: Core :: Integration API (1.0.0.SNAPSHOT)
    9|Resolved   |    1|Weld-OSGi :: Core :: Mandatory (1.0.0.SNAPSHOT)
```

```
g!
```

2.2. Say hello to the World: your first bean bundle

It is time to test your installation with your first application bean bundle. The goal here is to provide a "Hello World!" (and "Goodbye World!") service and use CDI to inject it into the main class of your bean bundle. It will be the `hello-world` bundle.

First you need to write down the "Hello World!" service:

- The `com.sample.api.HelloWorld.java` interface

```
package com.sample.api;

public interface HelloWorld {

    void sayHello();
    void sayGoodbye();
}
```

- The `com.sample.impl.HelloWorldImpl.java` implementation class

```
package com.sample.impl;

import com.sample.api.HelloWorld;

public class HelloWorldImpl implements HelloWorld {

    @Override
    public void sayHello() {
        System.out.println("Hello World!");
    }

    @Override
    public void sayGoodbye() {
        System.out.println("Goodbye World!");
    }
}
```

Nothing fancy here.

Next you need to write the entry point of your application (i.e the main class of the bean bundle). That is the `com.sample.App.java` main class

```
package com.sample;

import com.sample.api.HelloWorld; (1)
import org.osgi.cdi.api.extension.events.BundleContainerEvents; (2)
import javax.enterprise.event.Observes;
import javax.inject.Inject;

public class App {
    @Inject (3)
    HelloWorld helloWorld;

     public void onStartup(@Observes BundleContainerEvents.BundleContainerInitialized event)
 { (4)
        helloWorld.sayHello();
    }

   public void onShutdown(@Observes BundleContainerEvents.BundleContainerShutdown event) { (5)
        helloWorld.sayGoodbye();
    }
}
```

You import your service interface **(1)** and the CDI and Weld-OSGi dependencies **(2)**. You ask CDI for an injection of the `HelloWorld` service **(3)**. `onStartup` method**(4)** and `onShutdown` method **(5)** are called when CDI usage is enable for the bean bundle **(4)** (i.e the bean bundle application can start) or when CDI usage is disable for the bean bundle **(5)** (i.e the bean bundle application may stop).

Finally you should write the configuration files of your bean bundle:

- The `pom.xml` Maven configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.sample</groupId>
    <artifactId>hello-world</artifactId>
    <version>1.0</version>
    <packaging>bundle</packaging> (1)

    <dependencies> (2)
        <dependency>
            <groupId>javax.inject</groupId>
            <artifactId>javax.inject</artifactId>
            <version>1</version>
        </dependency>
        <dependency>
            <groupId>javax.enterprise</groupId>
            <artifactId>cdi-api</artifactId>
            <version>1.0-SP4</version>
        </dependency>
        <dependency>
            <groupId>org.osgi.cdi</groupId>
            <artifactId>weld-osgi-core-api</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId> (3)
                <extensions>true</extensions>
                <configuration>
                    <instructions>
                 <_include>-target/classes/META-INF/${project.artifactId}.bnd</_include> (4)
                    </instructions>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

You setup the dependencies for the `com.sample.App.java` main class **(2)**. You say that Maven may build an OSGi bundle **(1)** using the `maven-bundle-plugin` **(3)** with the `META-INF/hello-world.bnd` **(4)** configuration file.

- The `META-INF\beans.xml` CDI marker file

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  http://java.sun.com/xml/ns/
javaee/beans_1_0.xsd">
</beans>
```

This is an empty `beans.xml` file that only tells the bundle is a bean bundle (i.e Weld-OSGi may managed it).

- The `META-INF\hello-world.bnd` OSGi configuration file

```
# Let bnd handle the MANIFEST.MF generation
```

The bnd tool will generate the OSGi `MANIFEST.MF` configuration file just fine for this simple example.

Your project should now looks like that:

```
hello-world
      pom.xml
    - src
        - main
            - java
                - com.sample
                  App.java
                    - api
                      HelloWorld.java
                    - impl
                      HelloWorldImpl.java
            - resources
                - META-INF
                  beans.xml
                  hello-world.bnd
```

It is time to try out this first bean bundle:

- Build your project using Maven: `mvn clean install`
- Copy the generated `hello-world-1.0.jar` file to the Felix home `bundle` directory
- Update the Felix configuration file to auto install the bean bundle

```
felix.auto.install.1= file:bundle/weld-osgi-core-api-1.0-SNAPSHOT.jar \
file:bundle/weld-osgi-core-spi-1.0-SNAPSHOT.jar \
file:bundle/weld-osgi-core-mandatory-1.0-SNAPSHOT.jar \
file:bundle/hello-world-1.0.jar
```

- Start the Felix framework

```
____________________________
Welcome to Apache Felix Gogo

g! lb
START LEVEL 1
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (3.2.2)
    1|Active     |    1|Apache Felix Bundle Repository (1.6.2)
    2|Active     |    1|Apache Felix Gogo Command (0.8.0)
    3|Active     |    1|Apache Felix Gogo Runtime (0.8.0)
    4|Active     |    1|Apache Felix Gogo Shell (0.8.0)
    5|Active     |    1|Weld-OSGi :: Core :: Extension Impl (1.0.0.SNAPSHOT)
    6|Active     |    1|Weld-OSGi :: Implementation :: Weld Integration (1.0.0.SNAPSHOT)
    7|Resolved   |    1|Weld-OSGi :: Core :: Extension API (1.0.0.SNAPSHOT)
    8|Resolved   |    1|Weld-OSGi :: Core :: Integration API (1.0.0.SNAPSHOT)
    9|Resolved   |    1|Weld-OSGi :: Core :: Mandatory (1.0.0.SNAPSHOT)
   10|Installed  |    1|hello-world (1.0.0)
g!
```

and start your bean bundle

```
g! start 10
g! Hello World!
```

It is greeting the World ! Now stop the bean bundle

```
stop 10
g! Goodbye World!
```

and it says goodbye.

Congratulations, you just use CDI in an OSGi environment thank to Weld-OSGi. When you started your bean bundle Weld-OSGi started to manage it by providing it a Weld container. Once the container was completely started (and so the `HelloWorld` service injection completed) the `onStartup` method was called and the bean bundle application started.

2.3. CDI usage in bean bundle: a more complex example

This first bean bundle was very simple, is Weld-OSGi really enabling complex CDI usage in bean bundle? This section will pick up the `hello-world` example again and improve it by:

- Providing three implementations of the `HelloWorld` service to make your bean bundle multilingual (english, french and german)
- Force your bean bundle to present itself when is greeting

It will be the `hello-world-multilingual` bean bundle.

CDI usage limits in bean bundle

Every bean bundle gets its own Weld container from the extension bundle, so the CDI usage may stay within the bean bundle boundary:

- Any bean in the bean bundle can be injected only in that bean bundle (even exported package classes)
- Reciprocally, a bean from another bean bundle cannot be injected in the bean bundle (even imported package classes)
- Any decorator, interceptor or alternative declaration in the `beans.xml` file of a bean bundle applies only for this bean bundle

Furthermore, Weld-OSGi provides Weld container to a bean bundle when it becomes active in the OSGi environment, so any bundle in another state is not managed by Weld-OSGi. And an active bean bundle has CDI usage available only after the Weld container has initialized. In the same way CDI usage is unavailable once the Weld container has shutdown. That's the reason of `onStartup` and `onShutdown` methods from previous section:

- A `BundleContainerEvents.BundleContainerInitialized` event is fired when CDI usage gets available for the bean bundle
- A `BundleContainerEvents.BundleContainerShutdown` event is fired when CDI usage gets unavailable for the bean bundle

These two events are CDI events and may be observed with regular CDI mechanisms.

```
public void onStartup(@Observes BundleContainerEvents.BundleContainerInitialized
 event) {
    //CDI usage are available in the bean bundle
}

public  void  onShutdown(@Observes  BundleContainerEvents.BundleContainerShutdown
 event) {
    //CDI usage are unavailable in the bean bundle
}
```

Trying to use CDI mechanisms before the `BundleContainerEvents.BundleContainerInitialized` or after the `BundleContainerEvents.BundleContainerShutdown` event may result in errors.

First you need to upgrade your `HelloWorld` service and make it multilingual:

- The `com.sample.api.HelloWorld.java` interface

```
package com.sample.api;

public interface HelloWorld {

    void sayHello();
    void sayGoodbye();
}
```

  Nothing has changed here.

- The `com.sample.api.Language.java` qualifier annotation

```
package com.sample.api;

import javax.inject.Qualifier;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import static java.lang.annotation.ElementType.*;

@Qualifier (1)
@Target({ TYPE, METHOD, PARAMETER, FIELD })
@Retention(RetentionPolicy.RUNTIME)
public @interface Language {
    String value() default "ENGLISH"; (2)
}
```

  You create a regular CDI qualifier **(1)** in order to choose the language of your bean bundle. The language will be choose by providing a string (ENGLISH, FRENCH or GERMAN) to the qualifier **(2)**.

- The `com.sample.impl.HelloWorldImpl*.java` implementation classes

```
package com.sample.impl;

import com.sample.api.HelloWorld;
import com.sample.api.Language;

@Language("ENGLISH") (1)
public class HelloWorldEnglish implements HelloWorld {

    @Override
    public void sayHello() {
        System.out.println("Hello World!"); (2)
    }

    @Override
    public void sayGoodbye() {
        System.out.println("Goodbye World!"); (3)
    }
}
```

```
package com.sample.impl;

import com.sample.api.HelloWorld;
import com.sample.api.Language;

@Language("FRENCH") (1)
public class HelloWorldFrench implements HelloWorld {

    @Override
    public void sayHello() {
        System.out.println("Bonjour le Monde !"); (2)
    }

    @Override
    public void sayGoodbye() {
        System.out.println("Au revoir le Monde !"); (3)
    }
}
```

```
package com.sample.impl;

import com.sample.api.HelloWorld;
import com.sample.api.Language;

@Language("GERMAN") (1)
public class HelloWorldGerman implements HelloWorld {

    @Override
    public void sayHello() {
        System.out.println("Hallo Welt!"); (2)
    }

    @Override
    public void sayGoodbye() {
        System.out.println("Auf Wiedersehen Welt!"); (3)
    }
}
```

Here you just give the language of the implementation using the `Language` qualifier **(1)**, then you translate the outputs **(2)** **(3)**.

Now you will use CDI interceptor to force the bean bundle to present itself every time he greets the World:

- The `com.sample.api.Presentation.java` interceptor binding

```
package com.sample.api;

import javax.interceptor.InterceptorBinding;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import static java.lang.annotation.ElementType.*;

@InterceptorBinding (1)
@Target({ TYPE, METHOD })
@Retention(RetentionPolicy.RUNTIME)
public @interface Presentation {
}
```

You create a regular CDI interceptor binding **(1)** in order to add a bean bundle presentation everytime it greets the World.

- The `com.sample.impl.HelloWorldImpl*.java` implementation classes

```
...
public class HelloWorld* implements HelloWorld {

    @Override @Presentation (1)
    public void sayHello() {
        System.out.println("Hello World!");
    }
...
}
```

You add the interceptor binding on the `sayHello` method to force presentation only when the bean bundle is greeting and not saying goodbye.

- The `com.sample.impl.PresentationInterceptor.java` interceptor

```
package com.sample.impl;

import com.sample.api.Presentation;

import javax.interceptor.AroundInvoke;
import javax.interceptor.Interceptor;
import javax.interceptor.InvocationContext;

@Interceptor (1)
@Presentation
public class PresentationInterceptor {

    @AroundInvoke
    public Object present(InvocationContext ctx) throws Exception {
        ctx.proceed(); (2)
                                                            Language       language       =
 ctx.getMethod().getDeclaringClass().getAnnotation(Language.class); (3)
```

```
        if(language != null) {
            String lang = language.value();
            if(lang != null) {
                if(lang.equals("FRENCH")) {
                    System.out.println("Je suis le bundle hello-world-multilingual"); (4)
                    return null;
                } else if(lang.equals("GERMAN")) {
                    System.out.println("Ich bin das bundle hello-world-multilingual"); (5)
                    return null;
                }
            }
        }
        System.out.println("I am the bundle hello-world-multilingual"); (6)
        return null;
    }
}
```

The interceptor is declared so by its annotations **(1)**. It prints the normal sentence using the intercepted method **(2)**, then it detects the current language **(3)** and prints the corresponding description **(4)** **(5)** **(6)**.

Now the `HelloWorld` service is a bit more complex and you use several CDI features.

Next you need to update the `com.sample.App.java` main class

```
package com.sample;

import com.sample.api.HelloWorld;
import com.sample.api.Language;
import org.osgi.cdi.api.extension.events.BundleContainerEvents;

import javax.enterprise.event.Observes;
import javax.inject.Inject;

public class App {

    @Inject @Language("ENGLISH") (1)
    HelloWorld helloWorldEnglish;

    @Inject @Language("FRENCH") (2)
    HelloWorld helloWorldFrench;

    @Inject @Language("GERMAN") (3)
    HelloWorld helloWorldGerman;

    public void onStartup(@Observes BundleContainerEvents.BundleContainerInitialized event)
 { (4)
        helloWorldEnglish.sayHello();
        helloWorldFrench.sayHello();
        helloWorldGerman.sayHello();
    }

    public void onShutdown(@Observes BundleContainerEvents.BundleContainerShutdown event) { (5)
        helloWorldEnglish.sayGoodbye();
        helloWorldFrench.sayGoodbye();
        helloWorldGerman.sayGoodbye();
    }
}
```

You replace the initial `HelloWorld` service injection by three qualified injection, one for each language **(1) (2) (3)**. You also update `onStartup` and `onShutdown` methods to greet and say goodbye in the three languages **(4) (5)**.

Finally you should write the configuration files of your bean bundle:

- The `pom.xml` Maven configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.sample</groupId>
    <artifactId>hello-world-multilingual</artifactId>
    <version>1.0</version>
    <packaging>bundle</packaging>

    <dependencies>
        <dependency>
            <groupId>javax.inject</groupId>
            <artifactId>javax.inject</artifactId>
            <version>1</version>
        </dependency>
        <dependency>
            <groupId>javax.enterprise</groupId>
            <artifactId>cdi-api</artifactId>
            <version>1.0-SP4</version>
        </dependency>
        <dependency>
            <groupId>org.osgi.cdi</groupId>
            <artifactId>weld-osgi-core-api</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <extensions>true</extensions>
                <configuration>
                    <instructions>
                     <_include>-target/classes/META-INF/${project.artifactId}.bnd</_include>
                    </instructions>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

Nothing has changed here, but the `artifactId`.

- The `META-INF\beans.xml` CDI marker file

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  http://java.sun.com/xml/ns/
javaee/beans_1_0.xsd">
    <interceptors> (1)
        <class>com.sample.impl.PresentationInterceptor</class>
    </interceptors>
</beans>
```

You declare your interceptor in order to activate it **(1)**.

- The `META-INF\hello-world-multilingual.bnd` OSGi configuration file

```
# Let bnd handle the MANIFEST.MF generation
```

Nothing has changed here.

Update bnd files name

Every time you change the `artifactId` of one of your bean bundle you may also update the name of its bnd file. `artifactId` and bnd file name should be the same in order to generate a bundle.

Your project should now looks like that:

```
hello-world-multilingual
      pom.xml
    - src
        - main
            - java
                - com.sample
                  App.java
                    - api
                      HelloWorld.java
                      Language.java
                      Presentation.java
                    - impl
                      HelloWorldEnglish.java
                      HelloWorldFrench.java
                      HelloWorldGerman.java
                      PresentationInterceptor.java
            - resources
                - META-INF
                  beans.xml
                  hello-world-multilingual.bnd
```

It is time to try out these CDI features in an OSGi environment:

- Build your project using Maven: `mvn clean install`
- Copy the generated `hello-world-multilingual-1.0.jar` file to the Felix home `bundle` directory and the remove the old `hello-world-1.0.jar` file.
- Update the Felix configuration file to auto install the bean bundle

```
felix.auto.install.1= file:bundle/weld-osgi-core-api-1.0-SNAPSHOT.jar \
file:bundle/weld-osgi-core-spi-1.0-SNAPSHOT.jar \
file:bundle/weld-osgi-core-mandatory-1.0-SNAPSHOT.jar \
file:bundle/hello-world-multilingual-1.0.jar
```

- Start the Felix framework

```
____________________________
Welcome to Apache Felix Gogo

g! lb
START LEVEL 1
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (3.2.2)
    1|Active     |    1|Apache Felix Bundle Repository (1.6.2)
    2|Active     |    1|Apache Felix Gogo Command (0.8.0)
    3|Active     |    1|Apache Felix Gogo Runtime (0.8.0)
    4|Active     |    1|Apache Felix Gogo Shell (0.8.0)
    5|Active     |    1|Weld-OSGi :: Core :: Extension Impl (1.0.0.SNAPSHOT)
    6|Active     |    1|Weld-OSGi :: Implementation :: Weld Integration (1.0.0.SNAPSHOT)
    7|Resolved   |    1|Weld-OSGi :: Core :: Extension API (1.0.0.SNAPSHOT)
    8|Resolved   |    1|Weld-OSGi :: Core :: Integration API (1.0.0.SNAPSHOT)
    9|Resolved   |    1|Weld-OSGi :: Core :: Mandatory (1.0.0.SNAPSHOT)
   10|Installed  |    1|hello-world-multilingual (1.0.0)
g!
```

  and start your bean bundle

```
g! start 10
g! Hello World!
I am the bundle hello-world-multilingual
Bonjour le Monde !
Je suis le bundle hello-world-multilingual
Hallo Welt!
Ich bin das bundle hello-world-multilingual
```

  It is greeting the World in the three languages (and in the right order)! And it is also presenting itself. Now stop the bean bundle

```
stop 10
g! Goodbye World!
Au revoir le Monde !
Auf Wiedersehen Welt!
```

and it also says goodbye in the three languages.

CDI seems to respond perfectly in an OSGi environment, thank to Weld-OSGi. But it is sad to use only one bundle for your application, in the next chapter we will see how Weld-OSGi allow to use OSGi powerfulness coupled with CDI easiness in multi bundles application.

Weld-OSGi addressing OSGi service layer complexity

You can now use CDI in an OSGi environment with Weld-OSGi and bean bundles. But Weld-OSGi also provide numerous solution helping you make your multi bundles OSGi application using CDI way.

In this chapter you will see:

- How to publish your CDI beans as OSGi services
- How to consume these new services in regular OSGi bundles
- How to match a auto published qualified CDI bean and a propertied OSGi service
- How to inject OSGi services in bean bundle

3.1. Publishing CDI beans as OSGi services

CDI beans can be seen like services, with an interface defining the service contract and one or many implementations performing the service. So Weld-OSGi allows to easily publish your CDI beans from bean bundles as OSGi services. To do so you just to put an annotation on your bean implementation classes, avoiding the whole OSGi publishing process.

Difference between regular OSGi services and auto published CDI beans

The main difference comes from that auto published services are proxied CDI bean instances so:

- Auto published services are contextual and then might be share between bundle (e.g with an auto published `ApplicationScope` annotated CDI bean)
- Furthermore the auto published service and the injected CDI instance may be the same in the same scope (i.e share their state)
- Auto published services might be decorated or intercepted by the providing bean bundle

Modify the `hello-world-multilingual` bean bundle to auto publish the `HelloWorld` services as OSGi services. It will be the `hello-world-provider` bean bundle.

Update the `com.sample.impl.HelloWorldImpl*.java` implementation classes

```
...
@Language("*")
@Publish (1)
public class HelloWorld* implements HelloWorld {
    ...
}
```

Simply put the `Publish` annotation on the implementation classes **(1)** and that is it ! Every time Weld-OSGi finds a CDI bean with the `Publish` annotation, it registers it as a new OSGi service.

Your project should now looks like that:

```
hello-world-provider
      pom.xml
    - src
        - main
            - java
                - com.sample
                  App.java
                    - api
                      HelloWorld.java
                      Language.java
                      Presentation.java
                    - impl
                      HelloWorldEnglish.java
                      HelloWorldFrench.java
                      HelloWorldGerman.java
                      PresentationInterceptor.java
            - resources
                - META-INF
                  beans.xml
                  hello-world-provider.bnd
```

Try your new `hello-world-provider` bean bundle in the OSGi environment:

- Build your project using Maven: `mvn clean install`
- Copy the generated `hello-world-provider-1.0.jar` file to the Felix home `bundle` directory and remove the old `hello-world-multilingual-1.0.jar` file.
- Update the Felix configuration file to auto install the bean bundle

```
felix.auto.install.1= file:bundle/weld-osgi-core-api-1.0-SNAPSHOT.jar \
file:bundle/weld-osgi-core-spi-1.0-SNAPSHOT.jar \
file:bundle/weld-osgi-core-mandatory-1.0-SNAPSHOT.jar \
file:bundle/hello-world-provider-1.0.jar
```

- Start the Felix framework and test your bean bundle

```
____________________________
Welcome to Apache Felix Gogo

g! lb
START LEVEL 1
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (3.2.2)
    1|Active     |    1|Apache Felix Bundle Repository (1.6.2)
    2|Active     |    1|Apache Felix Gogo Command (0.8.0)
    3|Active     |    1|Apache Felix Gogo Runtime (0.8.0)
    4|Active     |    1|Apache Felix Gogo Shell (0.8.0)
```

```
     5|Active     |    1|Weld-OSGi :: Core :: Extension Impl (1.0.0.SNAPSHOT)
     6|Active     |    1|Weld-OSGi :: Implementation :: Weld Integration (1.0.0.SNAPSHOT)
     7|Resolved   |    1|Weld-OSGi :: Core :: Extension API (1.0.0.SNAPSHOT)
     8|Resolved   |    1|Weld-OSGi :: Core :: Integration API (1.0.0.SNAPSHOT)
     9|Resolved   |    1|Weld-OSGi :: Core :: Mandatory (1.0.0.SNAPSHOT)
    10|Installed  |    1|hello-world-provider (1.0.0)
g! start 10
g! Hello World!
I am the bundle hello-world-provider
Bonjour le Monde !
Je suis le bundle hello-world-provider
Hallo Welt!
Ich bin das bundle hello-world-provider
stop 10
g! Goodbye World!
Au revoir le Monde !
Auf Wiedersehen Welt!
```

Nothing has visibly changed, but Weld-OSGi has published three new OSGi services. So the `Publish` annotation do not alter the regular CDI behavior in bean bundle.

In the next section you will see these new OSGi service in action by consuming them in a second bundle. But before that you will what options Weld-OSGi give when you auto publish OSGi service.

The `Publish` annotation allows to things:

- Modify the service rank of the auto published OSGi service

```
@Publish(rank = 1)
public class MyServiceImpl implements MyService {
}
```

- Provide the list of contracts that the service fulfills

```
@Publish(contracts = {ItfA.class, ItfB.class, AbsA.class})
public class MyServiceImpl extends AbsA implements MyService, ItfA, ItfB {
}
```

Every given class may be assignable to the service implementation type. It allows to publish a service with both its interface types, superclass type and own type.

What service types for your service implementation

Weld-OSGi auto-published services get their types from the following algorithm:

- If a (nonempty) contract list is provided with the `Publish` annotation the service is registered for all these types.
- Else if the implementation class possesses a (nonempty) list of non-blacklisted interfaces the service is registered for all these interface types.The blacklist is described below.

- Else if Weld-OSGi the implementation class possesses a non-blacklisted superclass the service is registered for this superclass type.
- Last if the implementation class has neither contract nor non-blacklisted interface or superclass, the service is register with is the implementation class type.

Weld-OSGi provides a type blacklist in order to filter auto-published OSGi service allowed type. TODO ?

3.2. Consuming Weld-OSGi auto published services

Create a new regular OSGi bundle that will consume the auto published services of the `hello-world-provider` bean bundle. It will be the `hello-world-consumer` bundle.

You need to write the entry point of your bundle (i.e the activator class of the bundle). That is the `com.sample.Activator.java` main class

```
package com.sample;

import com.sample.api.HelloWorld; (1)
import org.osgi.framework.BundleActivator; (2)
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceReference;

public class Activator implements BundleActivator {

    HelloWorld helloWorld;

    @Override
    public void start(BundleContext context) throws Exception { (3)
                                          ServiceReference     helloWorldReference      =
 context.getServiceReference(HelloWorld.class.getName()); (4)
        helloWorld = (HelloWorld)context.getService(helloWorldReference);
        helloWorld.sayHello(); (5)
    }

    @Override
    public void stop(BundleContext context) throws Exception { (6)
        helloWorld.sayGoodbye(); (7)
    }
}
```

You import your service interface **(1)** and the OSGi dependencies **(2)**. You ask the OSGi environment for the `HelloWorld`service **(4)**. Then you greet **(5)** and say goodbye **(7)** to the World at the start **(3)** and stop **(6)** of your bundle.

You should also do a quick update of the `com.sample.impl.PresentationInterceptor.java` interceptor

```
...
public class PresentationInterceptor {

    @AroundInvoke
    public Object present(InvocationContext ctx) throws Exception {
...
                if(lang.equals("FRENCH")) {
                    System.out.println("Je suis le bundle hello-world-provider");
                    return null;
                } else if(lang.equals("GERMAN")) {
                    System.out.println("Ich bin das bundle hello-world-provider");
                    return null;
                }
            }
        }
        System.out.println("I am the bundle hello-world-provider");
        return null;
    }
}
```

Your bean bundle may present itself right.

Finally you write the configuration files of your bundle:

- The `pom.xml` Maven configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.sample</groupId>
    <artifactId>hello-world-consumer</artifactId>
    <version>1.0</version>
    <packaging>bundle</packaging> (1)

    <dependencies> (2)
        <dependency>
            <groupId>org.osgi</groupId>
            <artifactId>org.osgi.core</artifactId>
            <version>4.2.0</version>
        </dependency>
        <dependency>
            <groupId>com.sample</groupId>
            <artifactId>hello-world-provider</artifactId>
            <version>1.0</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId> (3)
                <extensions>true</extensions>
                <configuration>
                    <instructions>
                 <_include>-target/classes/META-INF/${project.artifactId}.bnd</_include> (4)
```

```
                      </instructions>
                  </configuration>
              </plugin>
          </plugins>
      </build>

  </project>
```

You setup the dependencies for the `com.sample.App.java` main class **(2)**. You say that Maven may build an OSGi bundle **(1)** using the `maven-bundle-plugin` **(3)** with the `META-INF/hello-world.bnd` **(4)** configuration file.

- The `META-INF\hello-world-consumer.bnd` OSGi configuration file

```
# Let bnd handle the MANIFEST.MF generation
# Just precise that this bundle as an activator
Bundle-Activator com.sample.Activator
```

The bnd tool will generate the OSGi `MANIFEST.MF` configuration file just fine but you need to precise that there is an activator class.

- You do not add the `META-INF\beans.xml` CDI marker file since you want a regular OSGi bundle (i.e a bundle not managed by Weld-OSGi).

Activator class versus `BundleContainerEvents` events

`BundleActivator` class `start` and `stop` methods are called when the bundle becomes active (`start`) or inactive (`stop`). It works for both regular bundles and bean bundles. But it do not ensure the availability of CDI usage in bean bundle.

`BundleContainerEvents` events listening method are called when the bean bundle Weld container has initialized (`BundleContainerEvents.BundleContainerInitialized`) or has shutdown (`BundleContainerEvents.BundleContainerShutdown`). It works only for bean bundles and ensure the availability of CDI usage.

`BundleContainerEvents.BundleContainerInitialized` event listening method always occurs after `BundleActivator` class `start` method and `BundleContainerEvents.BundleContainerShutdown` event listening method always occurs after `BundleActivator` class `stop` method in a bean bundle.

Your project should now looks like that:

```
hello-world-consumer
      pom.xml
    - src
        - main
            - java
                - com.sample
                  Activator.java
            - resources
                - META-INF
                hello-world-consumer.bnd
```

Try your new `hello-world-consumer` bean bundle in the OSGi environment:

- Build your project using Maven: `mvn clean install`
- Copy the generated `hello-world-consumer-1.0.jar` file to the Felix home `bundle` directory (keep the `hello-world-provider-1.0.jar` file).
- Update the Felix configuration file to auto install the bean bundle

```
felix.auto.install.1= file:bundle/weld-osgi-core-api-1.0-SNAPSHOT.jar \
file:bundle/weld-osgi-core-spi-1.0-SNAPSHOT.jar \
file:bundle/weld-osgi-core-mandatory-1.0-SNAPSHOT.jar \
file:bundle/hello-world-provider-1.0.jar \
file:bundle/hello-world-consumer-1.0.jar
```

- Start the Felix framework

```
____________________________
Welcome to Apache Felix Gogo

g! lb
START LEVEL 1
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (3.2.2)
    1|Active     |    1|Apache Felix Bundle Repository (1.6.2)
    2|Active     |    1|Apache Felix Gogo Command (0.8.0)
    3|Active     |    1|Apache Felix Gogo Runtime (0.8.0)
    4|Active     |    1|Apache Felix Gogo Shell (0.8.0)
    5|Active     |    1|Weld-OSGi :: Core :: Extension Impl (1.0.0.SNAPSHOT)
    6|Active     |    1|Weld-OSGi :: Implementation :: Weld Integration (1.0.0.SNAPSHOT)
    7|Resolved   |    1|Weld-OSGi :: Core :: Extension API (1.0.0.SNAPSHOT)
    8|Resolved   |    1|Weld-OSGi :: Core :: Integration API (1.0.0.SNAPSHOT)
    9|Resolved   |    1|Weld-OSGi :: Core :: Mandatory (1.0.0.SNAPSHOT)
   10|Installed  |    1|hello-world-provider (1.0.0)
   11|Installed  |    1|hello-world-consumer (1.0.0)
g!
```

Start the provider bundle

```
g! start 10
g! Hello World!
I am the bundle hello-world-provider
Bonjour le Monde !
Je suis le bundle hello-world-provider
Hallo Welt!
Ich bin das bundle hello-world-provider
```

It greets the World and auto publish the `HelloWorld` services. Now start the consumer bundle

```
start 11
Hallo Welt!
Ich bin das bundle hello-world-multilingual
g!
```

It greets the World too, but seems it bit confused ! Everything is explained below. Stop it

```
g! stop 11
Auf Wiedersehen Welt!
```

It says goodbye. Finally stop the provider bundle

```
g! stop 10
g! Goodbye World!
Au revoir le Monde !
Auf Wiedersehen Welt!
```

It keeps the same behavior, the consumer bundle has no impact on it.

Why your OSGi bundle seems to speak a random language and think it is the `hello-world-provider` bundle

Your bundle speaks a random language because you did not precise what language it should speak! You will see in the next section how to do it. OSGi service lookup mechanism just picks the first matching service implementation it finds (with its own internal magic), giving you a "random" language.

Your bundle presents itself as the the `hello-world-provider` bundle because the presentation occurs in the interceptor class of the `hello-world-provider` bundle. Indeed the obtained service implementation is a CDI bean from the `hello-world-provider` bundle and so it is intercepted.

You may be careful when you use CDI bean both for bean bundle internal injection and for OSGi service auto-publication because it will be the same instance. Interceptor and decorator will act on auto published services even if they are consumed in other bundles, and in the same scope every service instances and injected instances will share the same state.

3.3. Select the service instance

Now you need to decide what language your consumer bundle will speak. To do so you cannot use CDI qualifier like in provider bundle because you are using OSGi mechanisms to obtain the service instance. Fortunately Weld-OSGi provides a binding between CDI service qualification and OSGi service properties.

Conversion CDI qualifiers to OSGi service properties

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

- A valued element generate a property with this template:

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

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

will generate:

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

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

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

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

will generate:

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

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

```
decapitalized_qualifier_name.decapitalized_element_name=*
```

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

will generate:

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

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

```
decapitalized_qualifier_name=*
```

```
@MyQualifier()
```

will generate:

```
(myqualifier=*)
```

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

Using these rules, modify your `hello-world-consumer` bundle in order to make it speak the three languages like the `hello-world-provider` bean bundle. It will be the `hello-world-consumer-multilingual` bundle.

All the work happens in the `com.sample.Activator.java` main class

```
package com.sample;

import com.sample.api.HelloWorld; (1)
import org.osgi.framework.BundleActivator; (2)
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceReference;

public class Activator implements BundleActivator {

    HelloWorld helloWorldEnglish;
    HelloWorld helloWorldFrench;
    HelloWorld helloWorldGerman;

    @Override
    public void start(BundleContext context) throws Exception { (3)
                                         ServiceReference     helloWorldEnglishReference     =
 context.getServiceReferences(HelloWorld.class.getName(),"(language.value=ENGLISH)")[0]; (4)
```

```
                                        ServiceReference     helloWorldFrenchReference     =
context.getServiceReferences(HelloWorld.class.getName(),"(language.value=FRENCH)")[0];
                                        ServiceReference     helloWorldGermanReference     =
context.getServiceReferences(HelloWorld.class.getName(),"(language.value=GERMAN)")[0];

        helloWorldEnglish = (HelloWorld)context.getService(helloWorldEnglishReference);
        helloWorldFrench = (HelloWorld)context.getService(helloWorldFrenchReference);
        helloWorldGerman = (HelloWorld)context.getService(helloWorldGermanReference);

        helloWorldEnglish.sayHello(); (5)
        helloWorldFrench.sayHello();
        helloWorldGerman.sayHello();
    }

    @Override
    public void stop(BundleContext context) throws Exception { (6)
        helloWorldEnglish.sayGoodbye(); (7)
        helloWorldFrench.sayGoodbye();
        helloWorldGerman.sayGoodbye();
    }
}
```

You import your service interface **(1)** and the OSGi dependencies **(2)**. You ask the OSGi environment for the `HelloWorld`services specifying the correct filter **(4)**. Then you greet **(5)** and say goodbye **(7)** to the World at the start **(3)** and stop **(6)** of your bundle.

Your project should now looks like that:

```
hello-world-consumer-multilingual
      pom.xml
    - src
        - main
            - java
                - com.sample
                  Activator.java
            - resources
                - META-INF
                  hello-world-consumer-multilingual.bnd
```

Try your new `hello-world-consumer-multilingual` bean bundle in the OSGi environment:

- Build your project using Maven: `mvn clean install`
- Copy the generated `hello-world-consumer-multilingual-1.0.jar` file to the Felix home `bundle` directory and remove the old `hello-world-consumer-1.0.jar` file (keep the `hello-world-provider-1.0.jar` file).
- Update the Felix configuration file to auto install the bean bundle

```
felix.auto.install.1= file:bundle/weld-osgi-core-api-1.0-SNAPSHOT.jar \
file:bundle/weld-osgi-core-spi-1.0-SNAPSHOT.jar \
file:bundle/weld-osgi-core-mandatory-1.0-SNAPSHOT.jar \
file:bundle/hello-world-provider-1.0.jar \
file:bundle/hello-world-consumer-multilingual-1.0.jar
```

- Start the Felix framework

```
____________________________
Welcome to Apache Felix Gogo

g! lb
START LEVEL 1
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (3.2.2)
    1|Active     |    1|Apache Felix Bundle Repository (1.6.2)
    2|Active     |    1|Apache Felix Gogo Command (0.8.0)
    3|Active     |    1|Apache Felix Gogo Runtime (0.8.0)
    4|Active     |    1|Apache Felix Gogo Shell (0.8.0)
    5|Active     |    1|Weld-OSGi :: Core :: Extension Impl (1.0.0.SNAPSHOT)
    6|Active     |    1|Weld-OSGi :: Implementation :: Weld Integration (1.0.0.SNAPSHOT)
    7|Resolved   |    1|Weld-OSGi :: Core :: Extension API (1.0.0.SNAPSHOT)
    8|Resolved   |    1|Weld-OSGi :: Core :: Integration API (1.0.0.SNAPSHOT)
    9|Resolved   |    1|Weld-OSGi :: Core :: Mandatory (1.0.0.SNAPSHOT)
   10|Installed  |    1|hello-world-provider (1.0.0)
   11|Installed  |    1|hello-world-consumer-multilingual (1.0.0)
g!
```

Start the provider bundle

```
g! start 10
g! Hello World!
I am the bundle hello-world-provider
Bonjour le Monde !
Je suis le bundle hello-world-provider
Hallo Welt!
Ich bin das bundle hello-world-provider
```

Everything works fine. Now start the consumer bundle

```
start 11
Hello World!
I am the bundle hello-world-provider
Bonjour le Monde !
Je suis le bundle hello-world-provider
Hallo Welt!
Ich bin das bundle hello-world-provider
g!
```

It is multilingual ! Stop everything

```
g! stop 11
Goodbye World!
```

```
Au revoir le Monde !
Auf Wiedersehen Welt!
g! stop 10
g! Goodbye World!
Au revoir le Monde !
Auf Wiedersehen Welt!
```

The goodbye part is good too.

You can now use auto published service from a bean bundle in any other bundle. It opens CDI features to regular OSGi bundles and ensure the compatibility of Weld-OSGi with old OSGi application.

But the OSGi looking process for services is still a complicated, in the next section you will see how you can use CDI programming in order to get your OSGi services with Weld-OSGi.

3.4. Injecting OSGi service in bean bundle

Create a new bean bundle that will use the auto published services by injection. It will be the `hello-world-consumer2-multilingual` bean bundle.

The entry point of your bean bundle, the `com.sample.App.java` main class

```
package com.sample;

import com.sample.api.HelloWorld;
import com.sample.api.Language;
import org.osgi.cdi.api.extension.Service;
import org.osgi.cdi.api.extension.annotation.OSGiService;
import org.osgi.cdi.api.extension.events.BundleContainerEvents;

import javax.enterprise.event.Observes;
import javax.enterprise.util.AnnotationLiteral;
import javax.inject.Inject;

public class App {

    @Inject (1)
    @OSGiService
    HelloWorld helloWorld;

    @Inject (2)
    Service<HelloWorld> helloWorldService;

    @Inject (3)
    @OSGiService
    @Language("ENGLISH")
    HelloWorld helloWorldEnglish;

    @Inject
    Service<HelloWorld> helloWorldServiceEnglish;

    @Inject
    @OSGiService
    @Language("FRENCH")
    HelloWorld helloWorldFrench;
    @Inject
    Service<HelloWorld> helloWorldServiceFrench;

    @Inject
    @OSGiService
    @Language("GERMAN")
    HelloWorld helloWorldGerman;

    @Inject
    Service<HelloWorld> helloWorldServiceGerman;

    HelloWorld helloWorld2;
    HelloWorld helloWorldEnglish2;
    HelloWorld helloWorldFrench2;
    HelloWorld helloWorldGerman2;

    public void onStartup(@Observes BundleContainerEvents.BundleContainerInitialized event)
{ (4)
        helloWorld2 = helloWorldService.get(); (5)
        helloWorldEnglish2 = helloWorldServiceEnglish.select(new
LanguageAnnotationEnglish()).get(); (6)
        helloWorldFrench2 = helloWorldServiceFrench.select("(language.value=FRENCH)").get(); (7)
        helloWorldGerman2 = helloWorldServiceGerman.select("(language.value=GERMAN)").get();

        helloWorld.sayHello(); (8)
        helloWorld2.sayHello();
        helloWorldEnglish.sayHello();
        helloWorldEnglish2.sayHello();
        helloWorldFrench.sayHello();
        helloWorldFrench2.sayHello();
        helloWorldGerman.sayHello();
        helloWorldGerman2.sayHello();

        for (HelloWorld service : helloWorldService) { (9)
            service.sayHello();
        }
    }

    public void onShutdown(@Observes BundleContainerEvents.BundleContainerShutdown event) { (10)
        helloWorld.sayGoodbye(); (11)
        helloWorld2.sayGoodbye();
        helloWorldEnglish.sayGoodbye();
        helloWorldEnglish2.sayGoodbye();
        helloWorldFrench.sayGoodbye();
        helloWorldFrench2.sayGoodbye();
        helloWorldGerman.sayGoodbye();
        helloWorldGerman2.sayGoodbye();

        for (HelloWorld service : helloWorldService) {
            service.sayGoodbye();
        }
    }

    private class LanguageAnnotationEnglish extends AnnotationLiteral<Language> implements
Language {
        @Override
        public String value() {
            return "ENGLISH";
        }
    }
}
```

A lot of things to discuss here. There is two ways to get an OSGi service injected:

- You put the `OSGiService` annotation on a regular injection point **(1)**
- You get a `Service<T>` injected with T the service type **(2)**, then you get the service **(5)**

If you want to choose the implementation:

- Qualify the `OSGiService` annotated injection with CDI qualifiers **(3)**
- Select the instance with the injected `Service<T> select` method and a set of CDI qualifier annotation **(6)** or an OSGi service filter **(7)**
- Iterate through the implementation of an injected `Service<T>` **(9)**

Finally you say hello **(8)** and goodbye **(11)** when its right **(4)** **(10)** in all languages with all technique.

Same old configuration files:

- The `pom.xml` Maven configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.sample</groupId>
    <artifactId>hello-world-consumer2-multilingual</artifactId>
    <version>1.0</version>
    <packaging>bundle</packaging>

    <dependencies>
        <dependency>
            <groupId>javax.inject</groupId>
            <artifactId>javax.inject</artifactId>
            <version>1</version>
        </dependency>
        <dependency>
            <groupId>javax.enterprise</groupId>
            <artifactId>cdi-api</artifactId>
            <version>1.0-SP4</version>
        </dependency>
        <dependency>
            <groupId>org.osgi.cdi</groupId>
            <artifactId>weld-osgi-core-api</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
        <dependency>
            <groupId>com.sample</groupId>
            <artifactId>hello-world-provider</artifactId>
            <version>1.0</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <extensions>true</extensions>
                <configuration>
                    <instructions>
                     <_include>-target/classes/META-INF/${project.artifactId}.bnd</_include>
                    </instructions>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

- The `META-INF\hello-world-consumer2-multilingual.bnd` OSGi configuration file

```
# Let bnd handle the MANIFEST.MF generation
```

- The `META-INF\beans.xml` CDI marker file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  http://java.sun.com/xml/ns/
javaee/beans_1_0.xsd">
</beans>
```

Your project should now looks like that:

```
hello-world-consumer2-multilingual
      pom.xml
    - src
        - main
            - java
                - com.sample
                  App.java
            - resources
                - META-INF
                  beans.xml
                  hello-world-consumer2-multilingual.bnd
```

Try your new `hello-world-consumer2-multilingual` bean bundle in the OSGi environment:

- Build your project using Maven: `mvn clean install`

- Copy the generated `hello-world-consumer2-multilingual.jar` file to the Felix home `bundle` directory (keep the `hello-world-provider-1.0.jar` and `hello-world-consumer-multilingual.jar` files).
- Update the Felix configuration file to auto install the bean bundle

```
felix.auto.install.1= file:bundle/weld-osgi-core-api-1.0-SNAPSHOT.jar \
file:bundle/weld-osgi-core-spi-1.0-SNAPSHOT.jar \
file:bundle/weld-osgi-core-mandatory-1.0-SNAPSHOT.jar \
file:bundle/hello-world-provider-1.0.jar \
file:bundle/hello-world-consumer-multilingual-1.0.jar \
file:bundle/hello-world-consumer2-multilingual-1.0.jar
```

- Start the Felix framework and test everything out:

```
____________________________
Welcome to Apache Felix Gogo

g! lb
START LEVEL 1
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (3.2.2)
    1|Active     |    1|Apache Felix Bundle Repository (1.6.2)
    2|Active     |    1|Apache Felix Gogo Command (0.8.0)
    3|Active     |    1|Apache Felix Gogo Runtime (0.8.0)
    4|Active     |    1|Apache Felix Gogo Shell (0.8.0)
    5|Active     |    1|Weld-OSGi :: Core :: Extension Impl (1.0.0.SNAPSHOT)
    6|Active     |    1|Weld-OSGi :: Implementation :: Weld Integration (1.0.0.SNAPSHOT)
    7|Resolved   |    1|Weld-OSGi :: Core :: Extension API (1.0.0.SNAPSHOT)
    8|Resolved   |    1|Weld-OSGi :: Core :: Integration API (1.0.0.SNAPSHOT)
    9|Resolved   |    1|Weld-OSGi :: Core :: Mandatory (1.0.0.SNAPSHOT)
   10|Installed  |    1|hello-world-provider (1.0.0)
   11|Installed  |    1|hello-world-consumer-multilingual (1.0.0)
   12|Installed  |    1|hello-world-consumer2-multilingual (1.0.0)
g! start 10
Hello World!
I am the bundle hello-world-provider
Bonjour le Monde !
Je suis le bundle hello-world-provider
Hallo Welt!
Ich bin das bundle hello-world-provider
g! start 11
Hello World!
I am the bundle hello-world-provider
Bonjour le Monde !
Je suis le bundle hello-world-provider
Hallo Welt!
Ich bin das bundle hello-world-provider
g! start 12
Hallo Welt!
Ich bin das bundle hello-world-provider
Bonjour le Monde !
Je suis le bundle hello-world-provider
Hello World!
I am the bundle hello-world-provider
```

```
Hello World!
I am the bundle hello-world-provider
Bonjour le Monde !
Je suis le bundle hello-world-provider
Bonjour le Monde !
Je suis le bundle hello-world-provider
Hallo Welt!
Ich bin das bundle hello-world-provider
Hallo Welt!
Ich bin das bundle hello-world-provider
Bonjour le Monde !
Je suis le bundle hello-world-provider
Hello World!
I am the bundle hello-world-provider
Hallo Welt!
Ich bin das bundle hello-world-provider
g! stop 12
Auf Wiedersehen Welt!
Au revoir le Monde !
Goodbye World!
Goodbye World!
Au revoir le Monde !
Au revoir le Monde !
Auf Wiedersehen Welt!
Auf Wiedersehen Welt!
Au revoir le Monde !
Goodbye World!
Auf Wiedersehen Welt!
g! stop 11
Goodbye World!
Au revoir le Monde !
Auf Wiedersehen Welt!
g! stop 10
Goodbye World!
Au revoir le Monde !
Auf Wiedersehen Welt!
g! lb
START LEVEL 1
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (3.2.2)
    1|Active     |    1|Apache Felix Bundle Repository (1.6.2)
    2|Active     |    1|Apache Felix Gogo Command (0.8.0)
    3|Active     |    1|Apache Felix Gogo Runtime (0.8.0)
    4|Active     |    1|Apache Felix Gogo Shell (0.8.0)
    5|Active     |    1|Weld-OSGi :: Core :: Extension Impl (1.0.0.SNAPSHOT)
    6|Active     |    1|Weld-OSGi :: Implementation :: Weld Integration (1.0.0.SNAPSHOT)
    7|Resolved   |    1|Weld-OSGi :: Core :: Extension API (1.0.0.SNAPSHOT)
    8|Resolved   |    1|Weld-OSGi :: Core :: Integration API (1.0.0.SNAPSHOT)
    9|Resolved   |    1|Weld-OSGi :: Core :: Mandatory (1.0.0.SNAPSHOT)
   10|Resolved   |    1|hello-world-provider (1.0.0)
   11|Resolved   |    1|hello-world-consumer-multilingual (1.0.0)
   12|Resolved   |    1|hello-world-consumer2-multilingual (1.0.0)
g!
```

It becomes a bit cacaphonic but everything should be here.

You know now how to use the OSGi service layer using Weld-OSGi. Howover you may refer to the specification to specific usage and niceties. In the next chapters you will see what other things Weld-OSGi can do, helping you using OSGi framework for your application:

- Event management and inter-bundle communication
- OSGi framework utility

Weld-OSGi addressing event notification and comunication

Weld-OSGi is enabling CDI in bean bundles, so you can use CDI event notification in your bean bundles. But Weld-OSGi also provide the support of OSGi framework events through CDI event mechanisms. And following the same principle Weld-OSGi also allows a full uncouple communication between bean bundles based on CDI events.

In this chapter you will see:

- How to fire and observe OSGi framework and Weld-OSGi events in bean bundles
- How to send and receive inter-bundle communication between your bean bundles

4.1. Get OSGi event notifications in your bean bundles

From here you will create a new application based on bean bundles. So far your bundles were greeting and saying goodbye, let them be a bit more interactive. You will have three bean bundles that present themselves when they start and greet newcomers (and say goodbye to leavers) when they start (or stop).

Create your three bean bundles following this scheme. They will be the `welcoming-tom`, `welcoming-dick` and `welcoming-harry` bean bundles.

First write the `com.sample.App` main classes:

```
package com.sample;

import org.osgi.cdi.api.extension.events.BundleContainerEvents;
import org.osgi.cdi.api.extension.events.BundleEvents;

import javax.enterprise.event.Observes;

public class App {

     public void onStartup(@Observes BundleContainerEvents.BundleContainerInitialized event)
 { (1)
        System.out.println("Tom: Hi everyone!");
    }

   public void onShutdown(@Observes BundleContainerEvents.BundleContainerShutdown event) { (2)
        System.out.println("Tom: Bye everyone!");
    }

    public void greetNewcomer(@Observes BundleEvents.BundleStarted event) { (3)
      String name = event.getSymbolicName().substring(21, event.getSymbolicName().length()); (4)
        if (!name.equals("tom")) { (5)
            System.out.println("Tom: Welcome " + name +'!');
        }
    }

    public void sayGoodbyeToLeaver(@Observes BundleEvents.BundleStopped event) { ((6)
      String name = event.getSymbolicName().substring(21, event.getSymbolicName().length()); (7)
```

```
        if (!name.equals("tom")) { (8)
            System.out.println("Tom: Goodbye " + name +'!');
        }
    }
}
```

You monitor four different Weld-OSGi events, two events about the CDI life cycle (your entry point from the previous chapter) **(1)** **(2)** an two events about the bundle life cycle (from the OSGi framework) **(3)** **(6)**. It allows to perform actions when the bean bundle is started (both for OSGi and CDI point of view) **(1)** or stopped **(2)**. Here you just greet or say goodbye. When a bundle (every bundle) starts **(3)** or stops **(6)** you get its name **(4)** **(7)** and say the appropriate sentence to other bundle **(5)** **(8)**. Do not forget to change the name accordingly to the current bundle !

What are the events broadcast by Weld-OSGi

Weld-OSGi broadcast five types of events, two for OSGi framework events, two for CDI events and one for bean bundle communications.

OSGi events monitor:

- Bundle life cycle events. They all respect the `AbstractBundleEvent` abstract class:
  - `BundleEvents.BundleInstalled`
  - `BundleEvents.BundleLazyActivation`
  - `BundleEvents.BundleResolved`
  - `BundleEvents.BundleStarted`
  - `BundleEvents.BundleStarting`
  - `BundleEvents.BundleStopped`
  - `BundleEvents.BundleStopping`
  - `BundleEvents.BundleUninstalled`
  - `BundleEvents.BundleUnresolved`
  - `BundleEvents.BundleUpdated`
- Service life cycle events. They all respect the `AbstractServiceEvent` abstract class:
  - `ServiceEvents.ServiceArrival`
  - `ServiceEvents.ServiceChanged`
  - `ServiceEvents.ServiceDeparture`

CDI events monitor.

- Bean bundle CDI containers life cycle. They all respect the `AbstractBundleContainerEvent` abstract class:
  - `BundleContainerEvents.BundleContainerInitialized`
  - `BundleContainerEvents.BundleContainerShutdown`

- Bean bundle service dependency validation:
  - `Valid`
  - `Invalid`

Communication events broadcast message between bean bundle: `InterBundleEvent`

Each type of event carries its own set of information and has its own range, please refer to the Weld-OSGi specification for more information.

CDI events summon contextual beans

Be careful when you use Weld-OSGi in order to handle OSGi events, you are still using CDI mechanisms. Thus when an event is observed it generate a contextual instance of the bean that declares the observing method. Class state might be different between calls.

Now configure your three bean bundles:

- The `pom.xml` Maven configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.sample</groupId>
    <artifactId>welcoming-*</artifactId>
    <version>1.0</version>
    <packaging>bundle</packaging>

    <dependencies>
        <dependency>
            <groupId>javax.inject</groupId>
            <artifactId>javax.inject</artifactId>
            <version>1</version>
        </dependency>
        <dependency>
            <groupId>javax.enterprise</groupId>
            <artifactId>cdi-api</artifactId>
            <version>1.0-SP4</version>
        </dependency>
        <dependency>
            <groupId>org.osgi.cdi</groupId>
            <artifactId>weld-osgi-core-api</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
```

```
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <extensions>true</extensions>
                <configuration>
                    <instructions>
                     <_include>-target/classes/META-INF/${project.artifactId}.bnd</_include>
                    </instructions>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

Just put the right `artifactId` here.

- The `welcoming-*.bnd` OSGi configuration file

```
# Let bnd handle the MANIFEST.MF generation
```

Here it is the file name that may match the `artifactId`.

- The `META-INF\beans.xml` CDI marker file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  http://java.sun.com/xml/ns/
javaee/beans_1_0.xsd">
</beans>
```

Nothing to say !

Your three projects should now looks like that:

```
welcoming-*
      pom.xml
    - src
        - main
            - java
                - com.sample
                  App.java
            - resources
                - META-INF
                  beans.xml
                  welcoming-*.bnd
```

Try everything out:

- Build your project,copy the generated files and update the Felix configuration ... you know the deal.

- Play with your three bean bundles

```
____________________________
Welcome to Apache Felix Gogo

g! lb
START LEVEL 1
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (3.2.2)
    1|Active     |    1|Apache Felix Bundle Repository (1.6.2)
    2|Active     |    1|Apache Felix Gogo Command (0.8.0)
    3|Active     |    1|Apache Felix Gogo Runtime (0.8.0)
    4|Active     |    1|Apache Felix Gogo Shell (0.8.0)
    5|Active     |    1|Weld-OSGi :: Core :: Extension Impl (1.0.0.SNAPSHOT)
    6|Active     |    1|Weld-OSGi :: Implementation :: Weld Integration (1.0.0.SNAPSHOT)
    7|Resolved   |    1|Weld-OSGi :: Core :: Extension API (1.0.0.SNAPSHOT)
    8|Resolved   |    1|Weld-OSGi :: Core :: Integration API (1.0.0.SNAPSHOT)
    9|Resolved   |    1|Weld-OSGi :: Core :: Mandatory (1.0.0.SNAPSHOT)
   10|Installed  |    1|welcoming-dick (1.0.0)
   11|Installed  |    1|welcoming-harry (1.0.0)
   12|Installed  |    1|welcoming-tom (1.0.0)
g! start 10
Dick: Hi everyone!
g! start 11
Harry: Hi everyone!
Dick: Welcome harry!
g! start 12
Tom: Hi everyone!
Harry: Welcome tom!
Dick: Welcome tom!
g! stop 10
Dick: Bye everyone!
Tom: Goodbye dick!
Harry: Goodbye dick!
g! start 10
Dick: Hi everyone!
Tom: Welcome dick!
Harry: Welcome dick!
g! stop 12
Tom: Bye everyone!
Harry: Goodbye tom!
Dick: Goodbye tom!
g! stop 11
Harry: Bye everyone!
Dick: Goodbye harry!
g! stop 10
Dick: Bye everyone!
g! start 10
Dick: Hi everyone!
g! start 11
Harry: Hi everyone!
Dick: Welcome harry!
g! start 12
Tom: Hi everyone!
Dick: Welcome tom!
Harry: Welcome tom!
```

You can now make interactive application using multiple bean bundles and CDI event mechanisms.

4.2. Use your own inter bundle notifications

You can now use inter bundle notification in order to make your bean bundles aware of the other bundles. But the choice of events seems a bit limited, more or less is about bundle (and bean bundle) and service life cycle. But what if you want to fire and listen custom events ?

Weld-OSGi allows it with its `InterBundleEvent` events. In this section you will update your three bean bundles to make them chat even when no movement is occurring. For example your bean bundles can ask if the others are still here on regular basis.

Update your three bean bundles following this scheme. They will be the `talkative-tom`, `talkative-dick` and `talkative-harry` bean bundles.

Update the `com.sample.App` main classes:

```
package com.sample;

import org.osgi.cdi.api.extension.annotation.Sent;
import org.osgi.cdi.api.extension.annotation.Specification;
import org.osgi.cdi.api.extension.events.BundleContainerEvents;
import org.osgi.cdi.api.extension.events.BundleEvents;
import org.osgi.cdi.api.extension.events.InterBundleEvent;
import org.osgi.framework.Bundle;

import javax.enterprise.event.Event;
import javax.enterprise.event.Observes;
import javax.inject.Inject;

public class App {

    @Inject
    private Event<InterBundleEvent> communication; (1)

     public void onStartup(@Observes BundleContainerEvents.BundleContainerInitialized event)
 { (2)
        System.out.println("Harry: Hi everyone!");
                                AskThread    askThread    =    new    AskThread(communication,
 event.getBundleContext().getBundle()); (3)
        askThread.start();
    }

   public void onShutdown(@Observes BundleContainerEvents.BundleContainerShutdown event) { (4)
        System.out.println("Harry: Bye everyone!");
    }

    public void greetNewcomer(@Observes BundleEvents.BundleStarted event) { (5)
        String name = event.getSymbolicName().substring(21, event.getSymbolicName().length());
        if (!name.equals("harry")) {
            System.out.println("Harry: Welcome " + name + '!');
        }
    }

    public void sayGoodbyeToLeaver(@Observes BundleEvents.BundleStopped event) { (6)
        String name = event.getSymbolicName().substring(21, event.getSymbolicName().length());
        if (!name.equals("harry")) {
            System.out.println("Harry: Goodbye " + name + '!');
        }
    }

      public  void  acknowledge(@Observes  @Sent  @Specification(String.class)  InterBundleEvent
 message) { (7)
        System.out.println("Harry: Hey " + message.get() + " i'm still here.");
    }

    private class AskThread extends Thread { (8)
        Event<InterBundleEvent> communication;
        Bundle bundle;

        AskThread(Event<InterBundleEvent> communication, Bundle bundle) {
            this.communication = communication;
            this.bundle = bundle;
        }

        public void run() {
            while(true) { (9)
                try {
                    sleep(5000);
                } catch (InterruptedException e) {
                }
                if(bundle.getState() == Bundle.ACTIVE) { (10)
                    System.out.println("Harry: is there still someone here ?");
                    communication.fire(new InterBundleEvent("harry")); (11)
                } else {
                    break;
                }
            }
        }
    }
}
```

There is no modification for the previous chat action **(2) (4) (5) (6)**; except that you start a thread when the bean bundle has started **(3)**. This thread **(8)** is an infinite loop **(9)** that ask the active bundle to acknowledge there presence by firing an `InterBundleEvent` **(11)**. This `InterBundleEvent` is listened by your three bean bundles in order to print the answer. The listening method catch only communication from outside the bean bundle (`@Sent`) and that has a `String` message (`@Specification(String.class)`) **(7)**.

There is nothing to update in the bean bundles configuration files.

Your three projects should now looks like that:

```
talkative-*
      pom.xml
    - src
        - main
            - java
                - com.sample
                  App.java
            - resources
                - META-INF
                  beans.xml
                  talkative-*.bnd
```

Install everything and let your bundle chat:

```
____________________________
Welcome to Apache Felix Gogo

g! lb
START LEVEL 1
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (3.2.2)
    1|Active     |    1|Apache Felix Bundle Repository (1.6.2)
    2|Active     |    1|Apache Felix Gogo Command (0.8.0)
    3|Active     |    1|Apache Felix Gogo Runtime (0.8.0)
    4|Active     |    1|Apache Felix Gogo Shell (0.8.0)
    5|Active     |    1|Weld-OSGi :: Core :: Extension Impl (1.0.0.SNAPSHOT)
    6|Active     |    1|Weld-OSGi :: Implementation :: Weld Integration (1.0.0.SNAPSHOT)
    7|Resolved   |    1|Weld-OSGi :: Core :: Extension API (1.0.0.SNAPSHOT)
    8|Resolved   |    1|Weld-OSGi :: Core :: Integration API (1.0.0.SNAPSHOT)
    9|Resolved   |    1|Weld-OSGi :: Core :: Mandatory (1.0.0.SNAPSHOT)
   10|Installed  |    1|talkative-dick (1.0.0)
   11|Installed  |    1|talkative-harry (1.0.0)
   12|Installed  |    1|talkative-tom (1.0.0)
g! start 10 11 12
Dick: Hi everyone!
Harry: Hi everyone!
Dick: Welcome harry!
Tom: Hi everyone!
Dick: Welcome tom!
Harry: Welcome tom!
g! Dick: is there still someone here ?
Harry: Hey dick i'm still here.
Tom: Hey dick i'm still here.
Harry: is there still someone here ?
Dick: Hey harry i'm still here.
Tom: Hey harry i'm still here.
Tom: is there still someone here ?
Dick: Hey tom i'm still here.
Harry: Hey tom i'm still here.
Dick: is there still someone here ?
Harry: Hey dick i'm still here.
Tom: Hey dick i'm still here.
Harry: is there still someone here ?
Dick: Hey harry i'm still here.
Tom: Hey harry i'm still here.
Tom: is there still someone here ?
Dick: Hey tom i'm still here.
Harry: Hey tom i'm still here.
stop 10
Dick: Bye everyone!
Tom: Goodbye dick!
Harry: Goodbye dick!
g! stop 11
Harry: Bye everyone!
Tom: Goodbye harry!
g! stop 12
```

```
Tom: Bye everyone!
g!
```

You can play with your bean bundles, looking them chat with each other. The `InterBundleEvent` allow you to fire any object you want, so it is a powerful communication system between your bean bundle. You may refer to the Weld-OSGi specifications for a complete review of `InterBundleEvent` usage.

OSGi is still alive

Weld-OSGi do not fordid you to use classic OSGi mechanisms. More it give you some way to easily get main OSGi object like Bundle or BundleContext. You just inject them, don't need to use OSGi verbosity.

There are :

- Current bundle injection:

  ```
  @Inject
  Bundle bundle;
  ```

  It provides the current bundle OSGi `Bundle` instance.

- Current bundle context injection:

  ```
  @Inject
  BundleContext context;
  ```

  It provides the current bundle OSGi `BundleContext` instance.

- Current bundle file injection:

  ```
  @Inject @DataFile("path/to/file")
  File dataFile;
  ```

  It provides a file within the current bundle path.

- Current bundle manifest headers injection:

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

  @Inject @BundleHeader("header-name")
  String header;
  ```

  It provides all the manifest headers or a specific header of the current bundle.

- Current bundle Weld-OSGi registration:

  ```
  @Inject
  Registration registration;
  ```

  It provides all the registrations for OSGi services managed by Weld-OSGi for the current bundle.