CDI utils

Rudy De Busscher

Version 1.1.0, 24/08/2022

Table of Contents

Release notes . . . 1
- 1.1.0 . . . 1
- 1.0.1 . . . 1
- 1.0.0 . . . 1

Setup . . . 2

Requirements . . . 3

Features . . . 4
- Programmatic CDI bean retrieval . . . 4
- CDI bean with generic Type created by Producer . . . 5
- Fake CDI provider . . . 6
- SLF4J LogProducer . . . 8

Known issues . . . 9

Exceptions . . . 10
- CDI-DEV-01 . . . 10
- CDI-DEV-02 . . . 10
- CDI-DEV-51 . . . 10
- CDI-DEV-52 . . . 10
- CDI-DEV-53 . . . 10

Release notes

1.1.0

- `CDICheck` class is available in 'utils-se' artifact but checks on the javax namespace. However, this class should be part of utils-cdi artifact and in this Jakarta version checks on the correct package.
- Compiled with JDK 11 since it is intended to be used with Jakarta EE 9.1.

Older versions should be avoided as they might give issues with Jakarta namespace (like CDICheck of utils-se which checks on javax namespace)

1.0.1

- Updated a dependency to a final version. (was snapshot in 1.0.0)

1.0.0

- Compiled against CDI 3.0 (Jakarta EE 9 namespace!)

Setup

Add the following artifact to your maven project file for using the programmatic CDI utilities.

```
<dependency>
    <groupId>be.atbash.jakarta.utils</groupId>
    <artifactId>utils-cdi</artifactId>
    <version>1.1.0</version>
</dependency>
```

When you want to use the fake CDI provider for testing purposes, you should add the following artifact.

```
<dependency>
    <groupId>be.atbash.jakarta.utils</groupId>
    <artifactId>utils-cdi</artifactId>
    <version>1.1.0</version>
    <classifier>tests</classifier>
    <scope>test</scope>
</dependency>
```

Requirements

These utilities work only with CDI 3.0 (thus Jakarta EE 9) running on JDK 8 or JDK 11.

Features

Programmatic CDI bean retrieval

Within those areas where you can't use CDI injection, or when you do not like to use the Instance type for injecting multiple or optional dependencies, you can use the *CDI.current()* construct.

The class **be.atbash.util.CDIUtils** has some handy utility methods using the *CDI.current()* method.

```
be.atbash.util.CDIUtils#retrieveInstances(Class<T>, Annotation... )
```

Retrieves all CDI beans which can be assigned to the class **T**. It can be typically used for retrieving all beans which implement a certain interface or abstract parent class. The second, optional, argument is/are the *Qualifiers* to restrict the list of beans.

The method returns an immutable list. When no CDI bean matches the criteria, an empty list is returned.

```
be.atbash.util.CDIUtils#retrieveInstance(Class<T>, Annotation... )
```

This version expects that there is exactly 1 CDI bean which matches the criteria (class type and optionally the qualifiers). It is used to retrieve the CDI bean at a place where no CDI injection is possible.

When there are no or multiple beans matching the parameters, the classic CDI exceptions *UnsatisfiedResolutionException* and *AmbiguousResolutionException* are thrown.

```
be.atbash.util.CDIUtils#retrieveOptionalInstance(Class<T>, Annotation... )
```

This variant of the *retrieveInstance* method doesn't throw an exception when no matching bean is found but returns **null** instead.

```
be.atbash.util.CDIUtils#retrieveInstanceByName(String, Class<T>)
```

This method returns the CDI bean with a certain name (typically assigned to it with *@Named*).

Important to know is that the CDI bean must be assignable to the class **T** through the *Class.isAssignableFrom()* method. This is done in order to prevent a *ClassCastException* when assigning the result to the generic type *T*.

When the CDI bean is not assignable to the class or the bean name is not found, an *UnsatisfiedResolutionException* is thrown.

```
be.atbash.util.CDIUtils#fireEvent(Object, Annotation... )
```

The above method will fire a CDI event programmatically, optionally give it the qualifiers along with it specified in the second parameter.

```
be.atbash.util.CDIUtils#getBeanManager()
```

Returns the **BeanManager** so that additional use-cases can be performed, other than the ones defined within the *CDIUtils* class.

CDI bean with generic Type created by Producer

CDI doesn't support beans with Generic types mainly because of Type erasure. But there exist use cases where it is desirable to have a CDI bean with a Generic type that is generated by a Producer method. CDI Utils has support for this use case without any additional code from the developer.

Imagine we are creating a framework which needs some kind of complex lookup object based on information in a certain interface/class. This lookup is optional so a fallback default behavior is available when not defined by the developer.

So imagine the following code.

```
public abstract class BaseInformation {

}
```

Which is the abstract parent class containing some general useful methods and developer needs to create his own version.

```
@Veto
public class ComplexLookup<T extends BaseInformation> {
    public ComplexLookup(T info) {}
}
```

This class contains all the complex lookup information based on the instance of *BaseInformation* we supply it. Since this information is mandatory and can't be a classic CDI bean, we create a constructor where we supply the information. We need to exclude this bean from the CDI management (hence the *Veto*) since it doesn't have a no-arg constructor or constructor with @Injected parameters.

The above 2 classes are in our framework, and the developer can supply the CDI bean by using a CDI Producer method within the application.

```
@ApplicationScoped
@Produces
public ComplexLookup<ApplicationInformation> produceLookup() {
    ApplicationInformation info = .....
    return new ComplexLookup(info);
}
```

Where the class *ApplicationInformation* has the *BaseInformation* class as the parent.

When we have the following statement in our framework

```
   ... = CDIUtils.retrieveOptionalInstance(ComplexLookup.class);
```

It will return null, also in the presence of the Producer. This is because we can only lookup something within the CDI system based on a Class and not a Type (which can contain Generic Type information)

Of course, we could remove the Type information from ComplexLookup but there exists another solution for some use case.

Atbash utils define a CDI extension class which looks for all CDI producer methods. It keeps the method information.

Later when the developer calls *CDIUtils.retrieveOptionalInstance* and no match is found in the CDI container, it checks if there wasn't a Provider method found at startup. If it is the case, the method is called and the result is returned. The instance which is returned is also cached so that a second retrieval of the bean returns the same instance.

The good thing is that all this happens without the need to do something additional by the developer. But it has some limitations

- Only ApplicationScoped is supported so that no Context information is needed. The instance can also easily be cached. This requirement is checked and the error CDI-DEV-02 is thrown when the Producer method doesn't generate *applicationScoped* instance.
- No Proxy is generated, so interceptors and decorators are not possible.

Fake CDI provider

The Fake CDI provider is created so that you can run unit tests on classes where you are using the **CDIUtils#retrievexxx()** methods described above.

If you would run the unit test, it would try to access the CDI system which is of course not available within the test. The class **be.atbash.util.BeanManagerFake** can provide these, most of the time fake, instances with the help of the *Mockito framework*.

a typical usage scenario makes this much clearer.

```
@RunWith(MockitoJUnitRunner.class)
public class SomeUnitTest {

    @Mock
    private Dependency dependencyMock;

    private SomeUnit unitUnderTest;

    private BeanManagerFake beanManagerFake;

    @Before
    public void setup() {
        beanManagerFake = new BeanManagerFake();
        // Register dependencies for all tests
        beanManagerFake.registerBean(dependencyMock, Dependency.class);

        unitUnderTest = new SomeUnit();
    }

    @After
    public void tearDown() {
        beanManagerFake.deregistration();
    }

    @Test
    public void testSomething() {
        // Register dependency for this test case only
        //beanManagerFake.registerBean();

        // Finish preparation
        beanManagerFake.endRegistration();

        when(dependencyMock.method()).thenReturn();

        unitUnderTest.doSomething();
        // Test your assumptions
    }
}
```

The above example is using the *MockitoJUnitRunner* which is not needed in order to use the **BeanManagerFake** (but the class is using some Mockito methods under the hood). Here we use it to create a *Mock* class of a dependency which is used by our system under test.

```
beanManagerFake.registerBean(dependencyMock, Dependency.class);
```

With the *registerBean()* method, we can register a CDI bean instance (the *dependencyMock*) and define under which *Class* (here the Dependency Class) this instance will be registered.

Make sure that you register the instance with the correct Class (just as in a real system). The second parameter is there so that the developer can choose the class to which the instance is bound (the interface, the abstract class etc...). and just as in the real CDI system, an instance can be bound to multiple classes, just add these in the registration call.

```
beanManagerFake.registerBean(dependencyMock, Dependency.class, Object.class);
```

The registration of the beans is not enough to have a completely working system. Once all the beans are registered, you have to initialize the system by creating the required mocks for the CDI system. This is done by calling the method *endRegistration()*.

```
beanManagerFake.endRegistration();
```

In order to keep the different tests independently, that no CDI beans are left from the previous run, you need to reset the system by a call to *deregistration*. an ideal place to do this is the @After annotated method which runs after each test method.

```
    @After
    public void tearDown() {
        beanManagerFake.deregistration();
    }
```

SLF4J LogProducer

There is a CDI producer defined which creates a SLF4J Logger. It takes the class information from the injection for the creation information.

Usage of injectable logger.

```
@Inject
private Logger logger;

public void doSomething() {
   logger.info("Performed the doSomething");
}
```

The type of logger is `org.slf4j.Logger`.

Known issues

The **BeanManagerFake** can’t handle qualifiers for the moment.

Exceptions

CDI-DEV-01

When you ask for a named CDI bean (CDIUtils#retrieveInstanceByName), but you specified a null or empty parameter as bean name, this exception is thrown.

CDI-DEV-02

When you try to register a CDI bean producer method (CDIUtils.registerProducerMethod) which does not produce an ApplicationScoped bean, this method is thrown. This is because there is only support for singletons.

CDI-DEV-51

When you try to register a CDI bean with the Fake CDI system (BeanManagerFake#registerBean) but didn't specify any type to assign the instance to (the second parameter, actually a vararg forgotten)

CDI-DEV-52

When you try to register a CDI bean with the Fake CDI system (BeanManagerFake#registerBean) but did try to register a null instance (first method parameter is null)

CDI-DEV-53

When you try to register a producer (with CDIUtils#registerProducerMethod) method which doesn't produce an instance at *ApplicationScoped*. You should never try to register a producer method manually.