

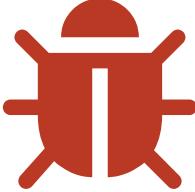
# Going further with CDI

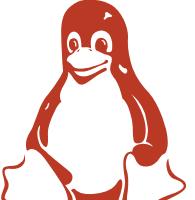
1.2

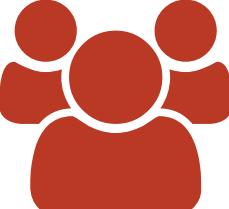
---

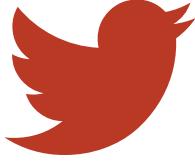
Antoine Sabot-Durand · Antonin Stefanutti

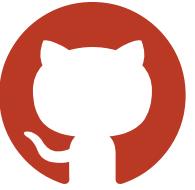
Antonin Stefanutti

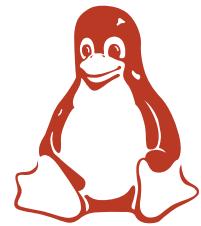
 Software Engineer

 Red Hat

 JBoss Fuse team

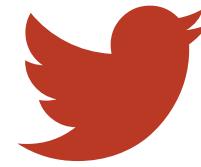
 @astefanut

 [github.com/astefanutti](https://github.com/astefanutti)



Red Hat

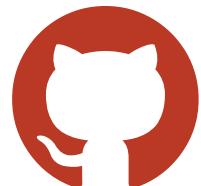
</> CDI spec lead



@antoine\_sd

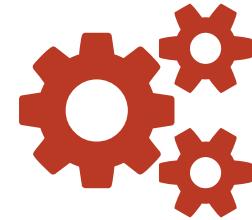


next-presso.com



github.com/antoinesd

Should I stay or should I go?



# A talk about advanced CDI

➡ Might be hard for beginners

➡ Don't need to be a CDI guru

**Should I stay or should I go?**

- 💡 If you know most of these you can stay

**@Inject**

**Event<T>**

**@Qualifier**

**@Produces**

**@Observes**

**InjectionPoint**

### 🔥 What's included:

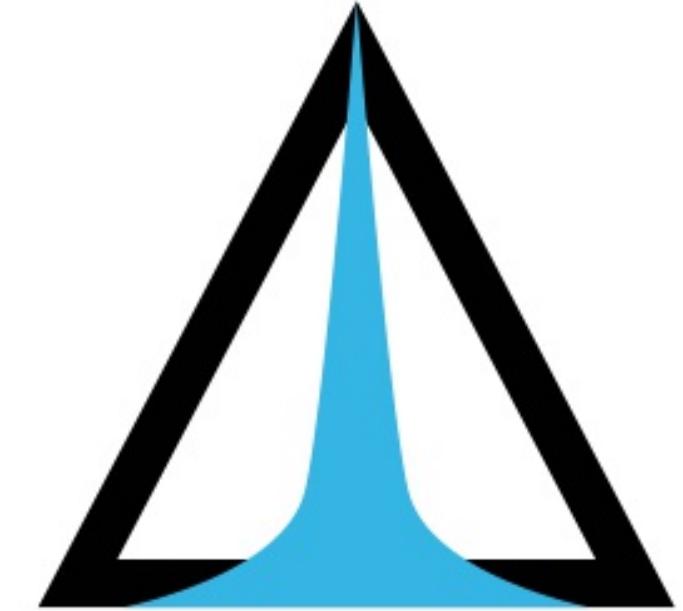
1. Real use cases from real life with real users
2. New approach to introduce portable extension concepts
3. Code in IDE with tests

### 🔥 What's not included:

1. Introduction to CDI
2. Old content on extension
3. Work with Context (need 2 more hours)

# Apache Deltaspike

1. Apache DeltaSpike is a great CDI toolbox
2. Provide helpers to develop extension
3. And a collection of modules like:
  1. Security
  2. Data
  3. Scheduler
4. More info on [deltaspike.apache.org](http://deltaspike.apache.org)



D E L T A S P I K E

# Arquillian

1. Arquillian is an integration test platform
2. It integrates with JUnit
3. Create your deployment in a dedicated method
4. And launch your tests against the container of your choice
5. We'll use the `weld-se-embedded` and `weld-ee-embedded` container
6. The right solution to test Java EE code
7. More info on [arquillian.org](http://arquillian.org)



# Agenda

- 💡 Slides available at [astefanutti.github.io/further-cdi](https://astefanutti.github.io/further-cdi)

**i** Meet CDI SPI

**i** Introducing CDI Extensions

**i** Metrics CDI

**i** Camel CDI

# Meet CDI SPI



# SPI can be split in 4 parts

# SPI can be split in 4 parts

-  Type meta-model

# SPI can be split in 4 parts

- Type meta-model
- CDI meta-model

# SPI can be split in 4 parts

- Type meta-model
- CDI meta-model
- CDI entry points

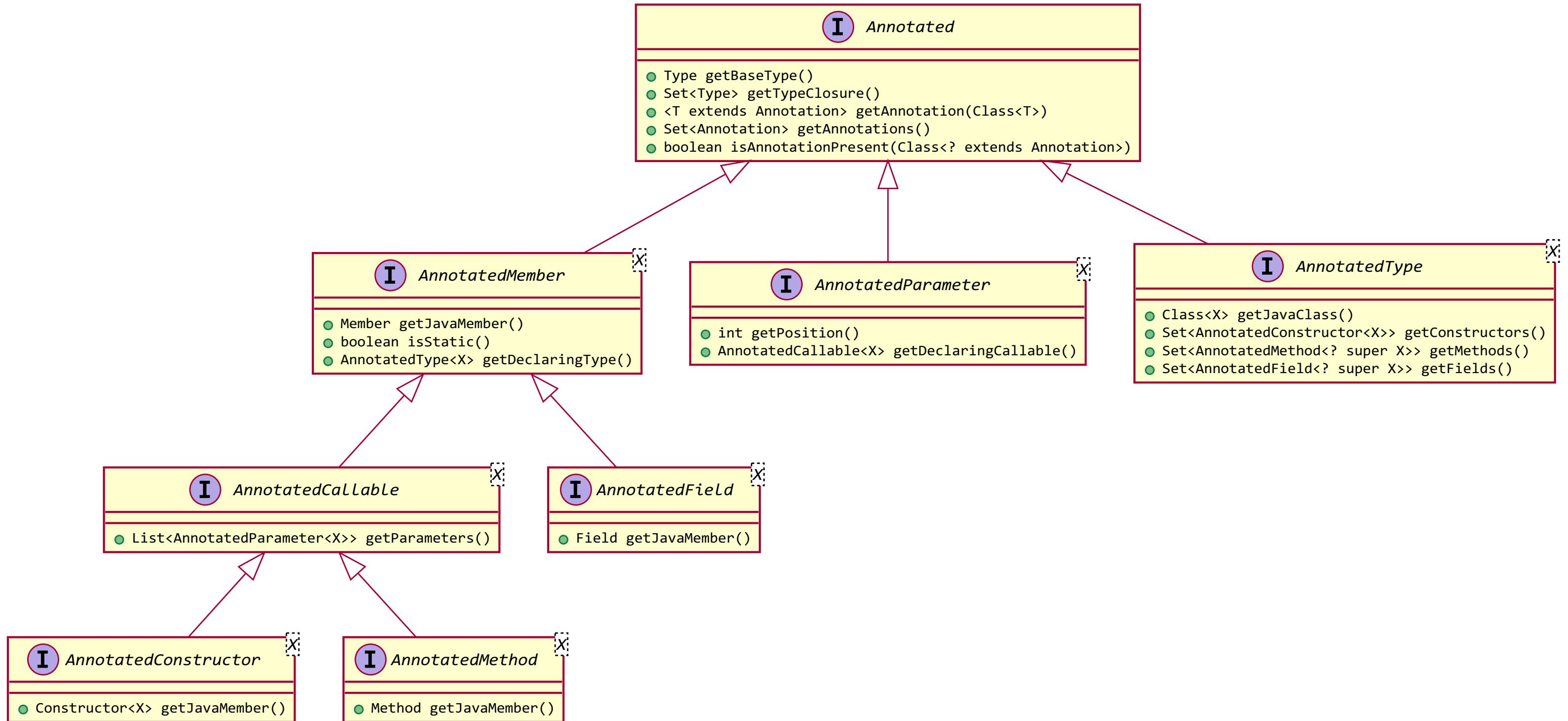
# SPI can be split in 4 parts

- Type meta-model
- CDI meta-model
- CDI entry points
- SPI dedicated to extensions

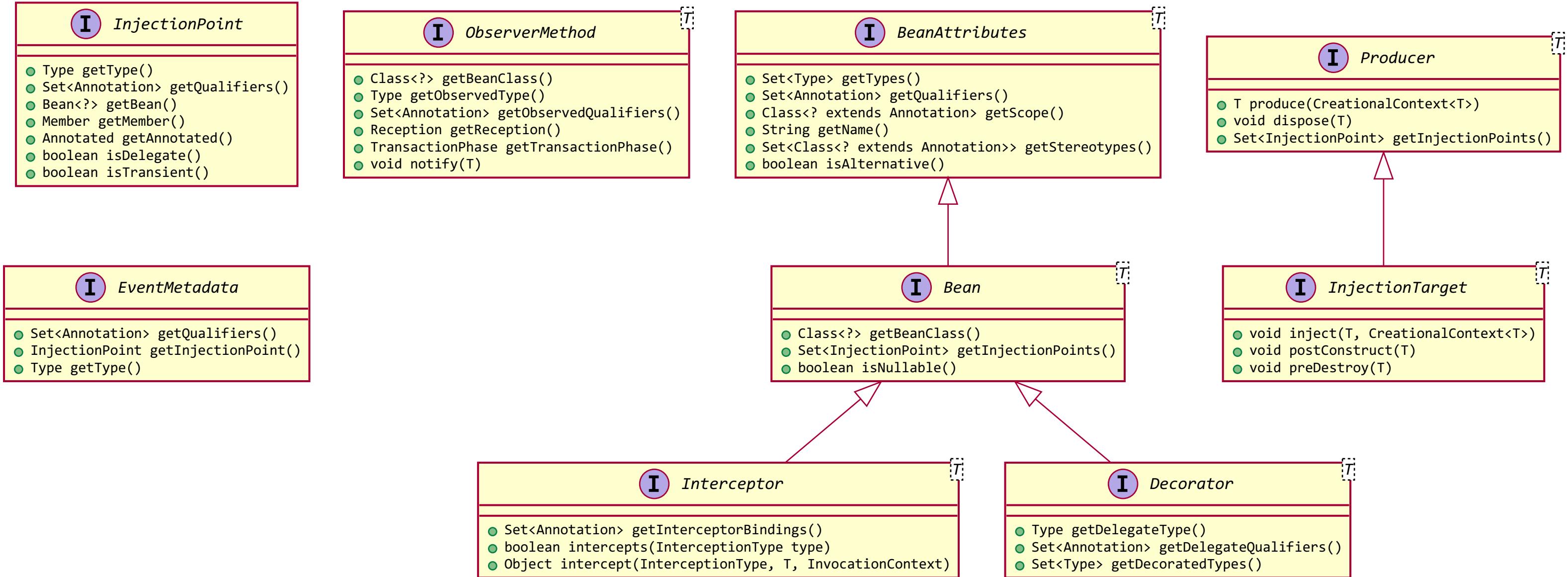
# Why having a type meta-model?

- 💡 Because `@Annotations` are configuration
- 💡 but they are also read-only
- 💡 So to configure we need a mutable meta-model...
- 💡 ... for annotated types

# SPI for type meta-model



# SPI dedicated to CDI meta-model



## This SPI can be used in your code (1/2)

 **InjectionPoint** can be used to get info about what's being injected

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

```
@Produces @HttpParam("")  
String getParamValue(InjectionPoint ip, HttpServletRequest req) {  
    return req.getParameter(ip.getAnnotated().getAnnotation(HttpParam.class).value());  
}
```

```
@Inject  
@HttpParam("productId")  
String productId;
```

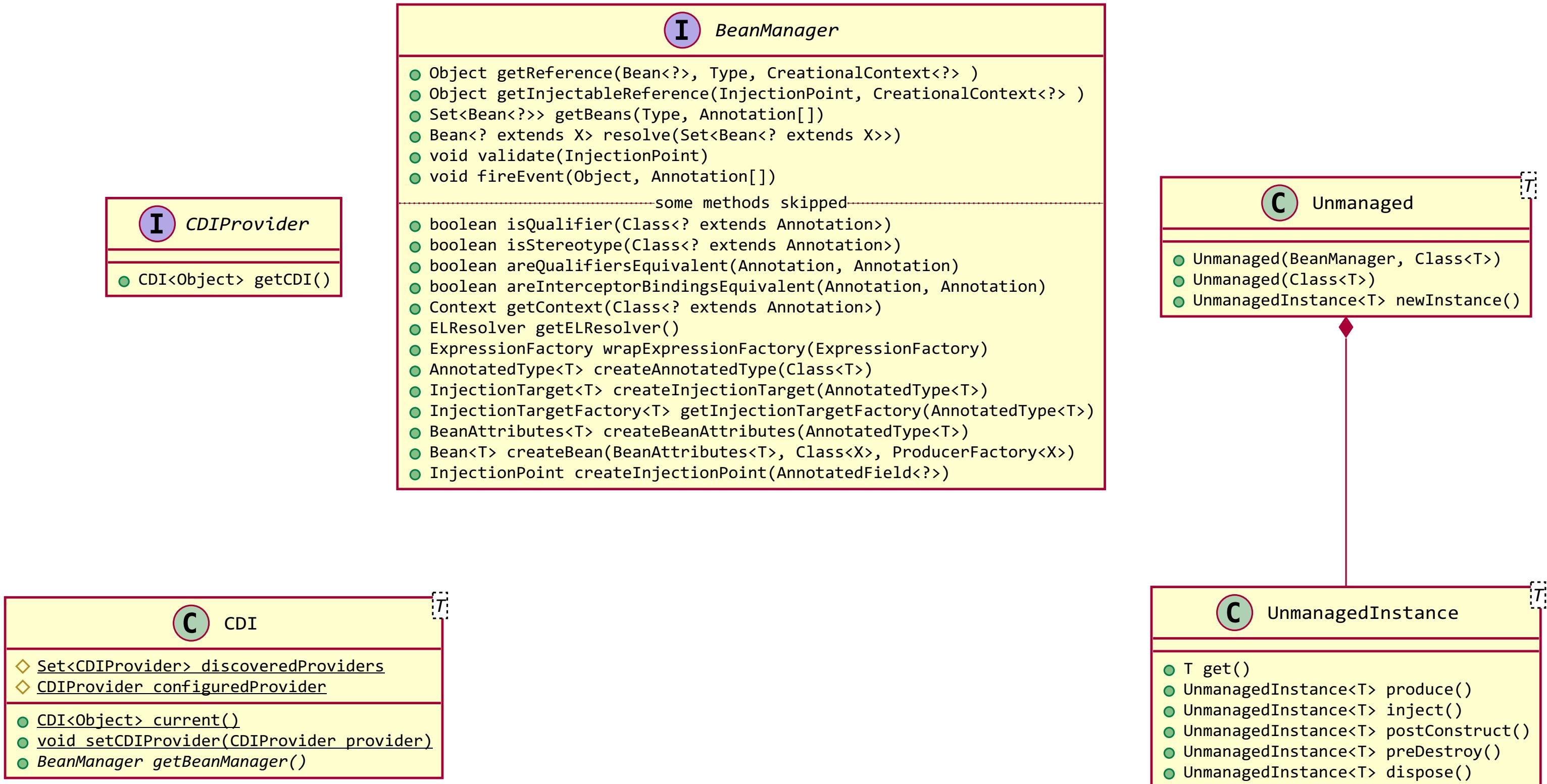
## This SPI can be used in your code (2/2)



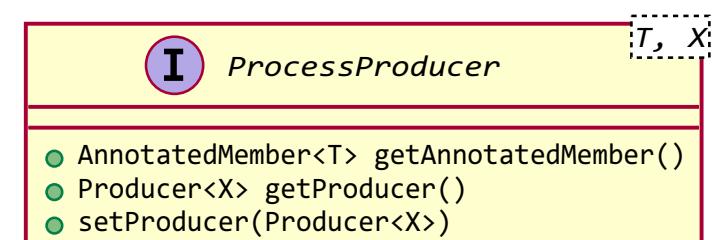
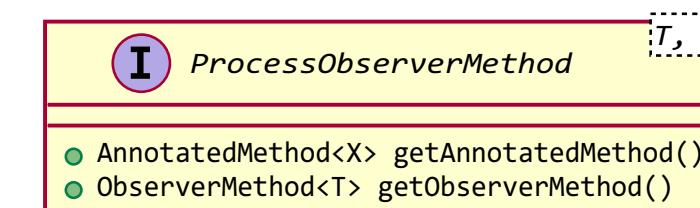
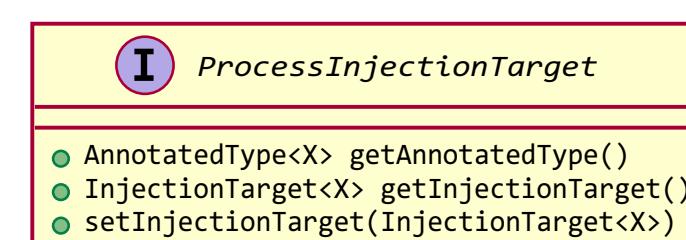
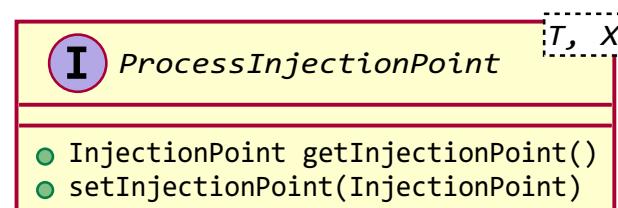
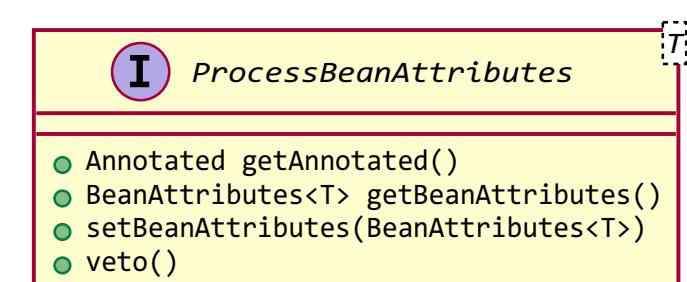
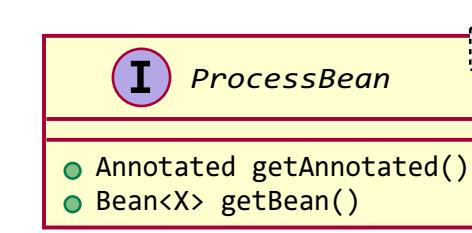
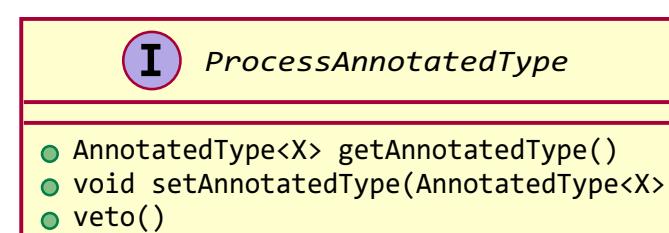
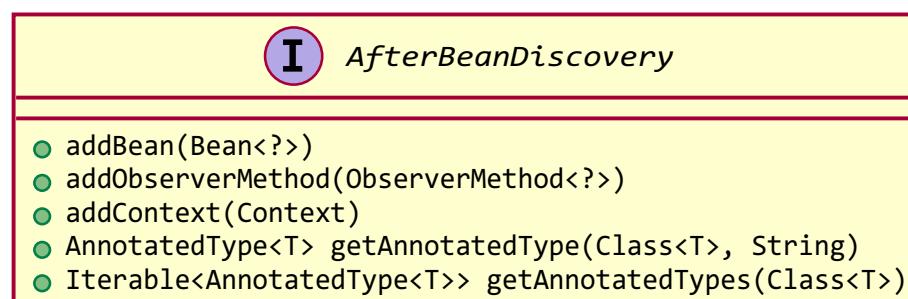
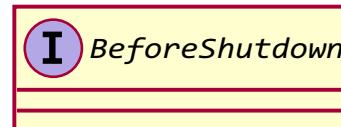
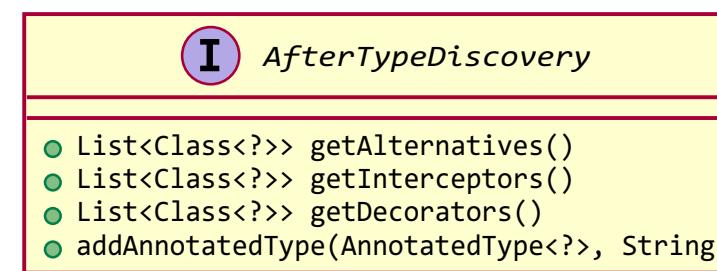
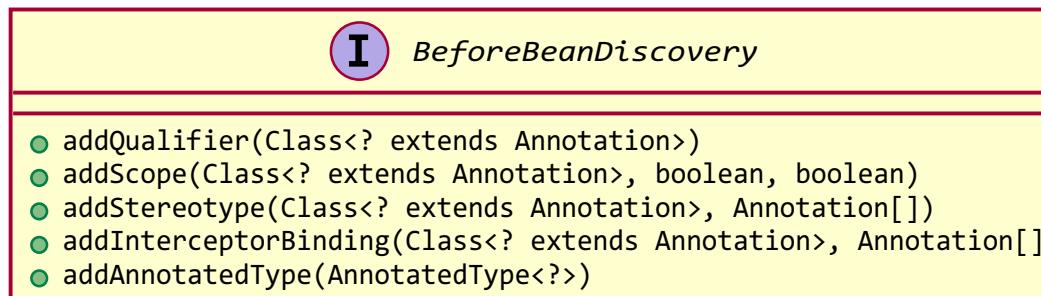
`InjectionPoint` contains info about requested type at `@Inject`

```
class MyMapProducer() {  
  
    @Produces  
    <K, V> Map<K, V> produceMap(InjectionPoint ip) {  
        if (valueIsNumber(((ParameterizedType) ip.getType())))  
            return new TreeMap<K, V>();  
        return new HashMap<K, V>();  
    }  
  
    boolean valueIsNumber(ParameterizedType type) {  
        Class<?> valueClass = (Class<?>) type.getActualTypeArguments()[1];  
        return Number.class.isAssignableFrom(valueClass)  
    }  
}
```

# SPI providing CDI entry points

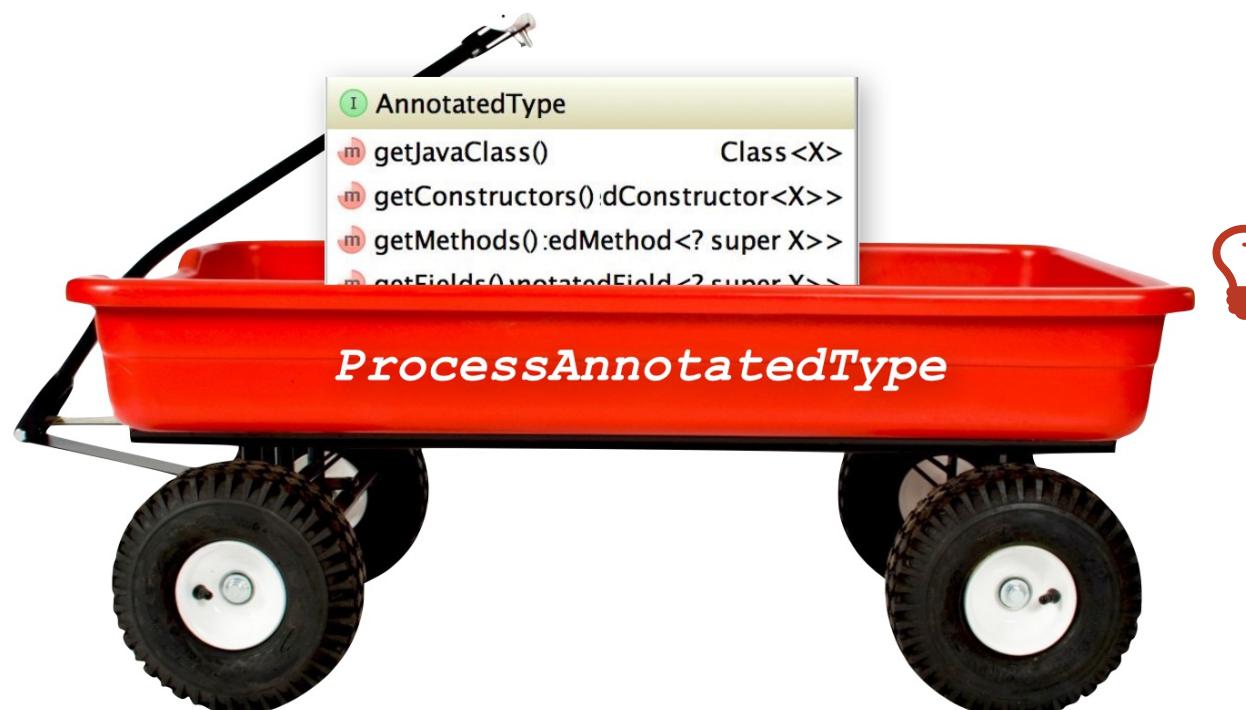


# SPI dedicated to extensions



# All these SPI interfaces are events containing meta-model SPI

- These events fired at boot time can only be observed in CDI extensions
- For instance:



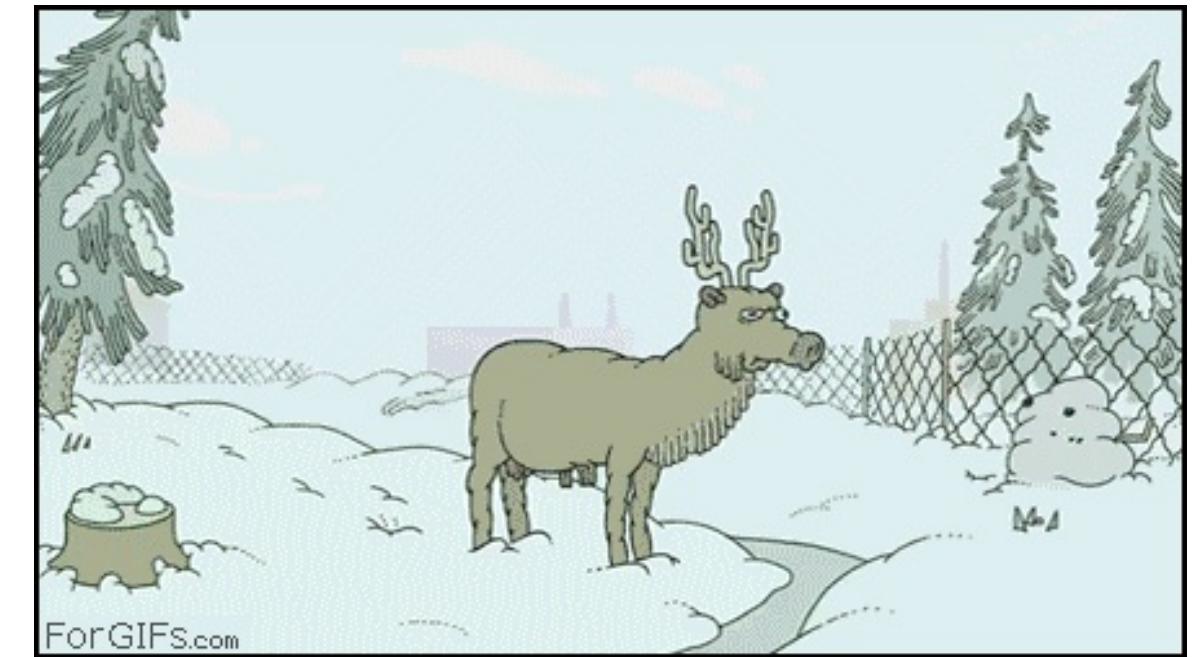
A `ProcessAnnotatedType<T>` event is fired for each type being discovered at boot time

Observing `ProcessAnnotatedType<Foo>` allows you to prevent `Foo` to be deployed as a bean by calling `ProcessAnnotatedType#veto()`

# Introducing CDI Portable Extensions

## Portable extensions

- i** One of the **most powerful feature** of the CDI specification
- i** Not really popularized, partly due to:
  1. Their **high level of abstraction**
  2. The good knowledge on Basic CDI and SPI
  3. Lack of information (CDI is often reduced to a basic DI solution)

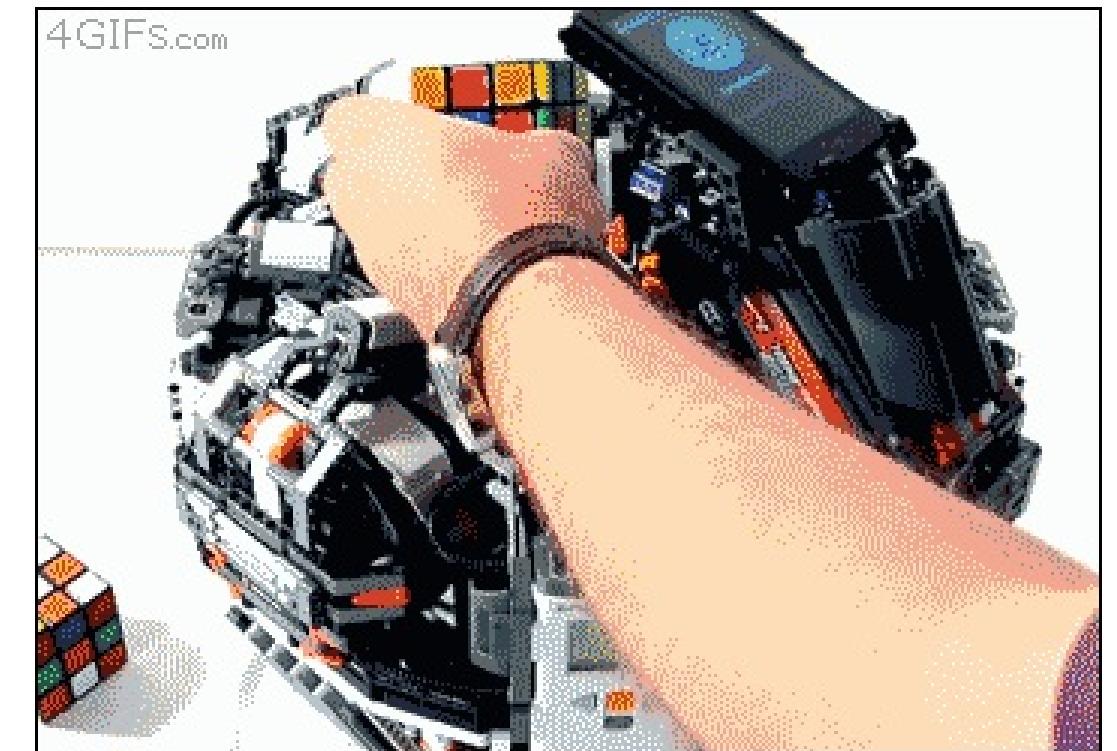


# Extensions, what for?

- 💡 To integrate 3rd party libraries, frameworks or legacy components
- 💡 To change existing configuration or behavior
- 💡 To extend CDI and Java EE
- 💡 Thanks to them, Java EE can evolve between major releases

# Extensions, how?

- 💡 Observing SPI events at boot time related to the bean manager lifecycle
- 💡 Checking what meta-data are being created
- 💡 Modifying these meta-data or creating new ones



## More concretely

### Service provider of the service

**i** `javax.enterprise.inject.spi.Extension` declared in  
`META-INF/services`

**💡** Just put the fully qualified name of your extension class in this file

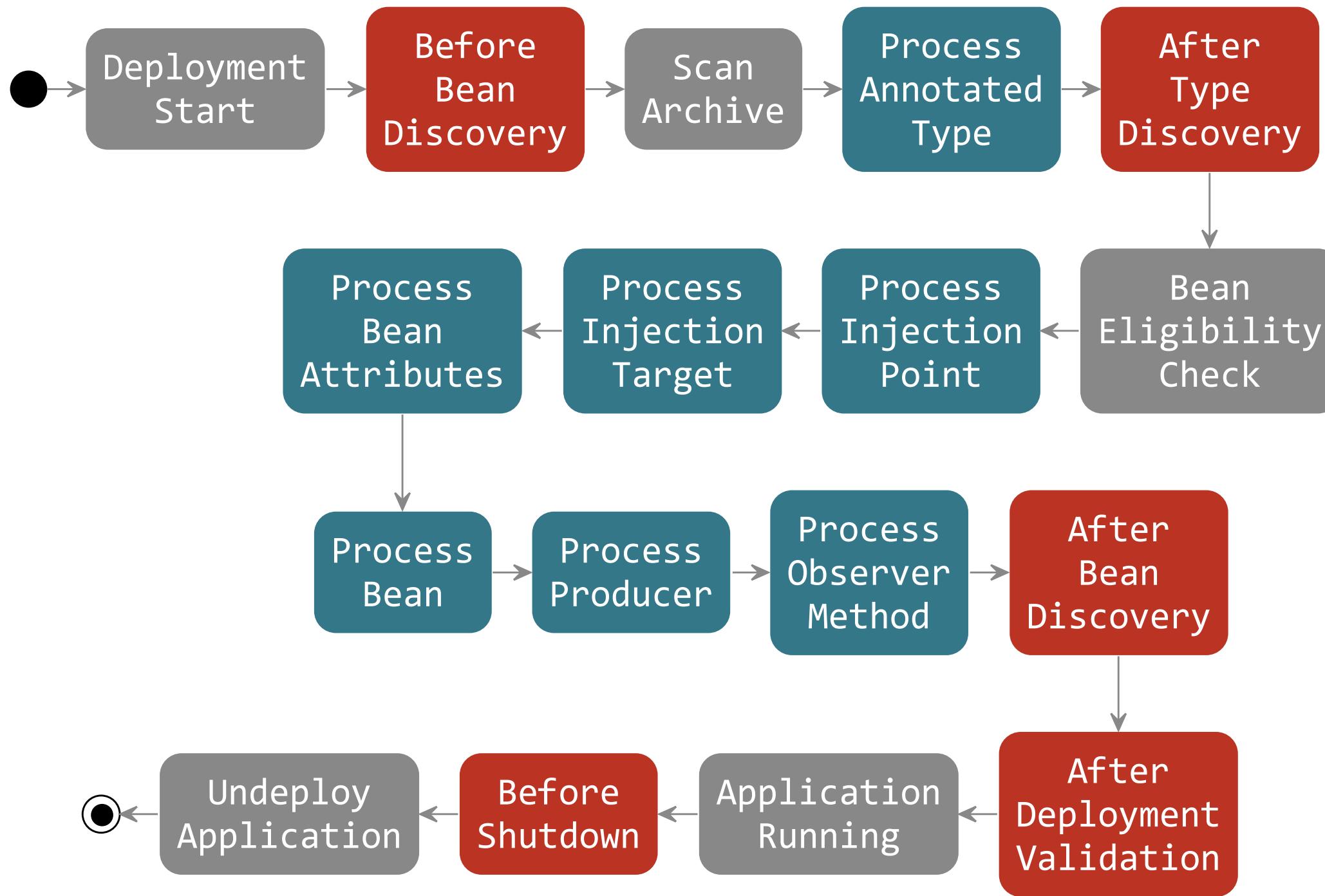
```
import javax.enterprise.event.Observes;
import javax.enterprise.inject.spi.Extension;

public class CdiExtension implements Extension {

    void beforeBeanDiscovery(@Observes BeforeBeanDiscovery bbd) {
    }
    //...

    void afterDeploymentValidation(@Observes AfterDeploymentValidation adv) {
    }
}
```

# Bean manager lifecycle



Internal Step

Happen Once

Loop on Elements

## Example: Ignoring JPA entities

- 💡 The following extension prevents CDI to manage entities
- ℹ️ This is a commonly admitted good practice

```
public class VetoEntity implements Extension {  
  
    void vetoEntity(@Observes @WithAnnotations(Entity.class)  
                    ProcessAnnotatedType<?> pat) {  
        pat.veto();  
    }  
}
```

**⚠ Extensions are launched during bootstrap and are based on CDI events**

**⚠ Once the application is bootstrapped, the Bean Manager is in **read-only mode** (no runtime bean registration)**

**⚠ You only have to `@Observes` built-in CDI events to create your extensions**

**Remember**

Integrating Dropwizard Metrics in CDI

Metrics CDI

## Dropwizard Metrics provides

- ➊ Different metric types: Counter, Gauge, Meter, Timer, ...
  - ➋ Different reporter: JMX, console, SLF4J, CSV, servlet, ...
  - ➌ MetricRegistry object which collects all your app metrics
  - ➍ Annotations for AOP frameworks: @Counted, @Timed, ...
  - ➎ ... but does not include integration with these frameworks
-  More at [dropwizard.github.io/metrics](https://dropwizard.github.io/metrics)

**Discover how we created CDI  
integration module for Metrics**

# Metrics out of the box (without CDI)

```
class MetricsHelper {  
    public static MetricRegistry registry = new MetricRegistry();  
}
```

```
class TimedMethodClass {  
  
    void timedMethod() {  
        Timer timer = MetricsHelper.registry.timer("timer"); ①  
        Timer.Context time = timer.time();  
        try {  
            /*...*/  
        } finally {  
            time.stop();  
        }  
    }  
}
```

- ① Note that if a `Timer` named `"timer"` doesn't exist, `MetricRegistry` will create a default one and register it

# Basic CDI integration

```
class MetricRegistryBean {  
    @Produces @ApplicationScoped  
    MetricRegistry registry = new MetricRegistry();  
}  
  
class TimedMethodBean {  
    @Inject MetricRegistry registry;  
  
    void timedMethod() {  
        Timer timer = registry.timer("timer");  
        Timer.Context time = timer.time();  
        try {  
            /*...*/  
        } finally {  
            time.stop();  
        }  
    }  
}
```



We could have a lot more with advanced **CDI** features

# Our goals to achieve full CDI integration

- 🔥 Produce and inject multiple **metrics** of the same type
- 🔥 Apply Metrics with provided annotations
- 🔥 Access same **Metric** through inject or **MetricRegistry** getter

**GOAL 1** Produce and inject  
**multiple** metrics of the same type

# What's the problem with multiple Metrics of the same type?

⚠ This code creates a deployment error (ambiguous dependency)

**@Produces**

```
Timer timer = new Timer(new SlidingTimeWindowReservoir(1L, MINUTES)); ①
```

**@Produces**

```
Timer timer = new Timer(new SlidingTimeWindowReservoir(1L, HOURS)); ②
```

**@Inject**

```
Timer timer; ③
```

- ① This timer that only keeps measurement of last minute is produced as a bean of type **Timer**
- ② This timer that only keeps measurement of last hour is produced as a bean of type **Timer**
- ③ This injection point is ambiguous since 2 eligible beans exist

## Solving the ambiguity

💡 We could use the provided `@Metric` annotation to qualify our beans

```
@Produces  
@Metric(name = "myTimer")  
Timer timer = new Timer(new SlidingTimeWindowReservoir(1L, MINUTES));
```

```
@Produces  
@Metric(name = "mySecondTimer")  
Timer timer = new Timer(new SlidingTimeWindowReservoir(1L, HOURS));
```

```
@Inject  
@Metric(name = "myTimer")  
Timer timer;
```

🔥 That won't work out of the box since `@Metric` is not a qualifier

# How to make `@Metric` a qualifier?

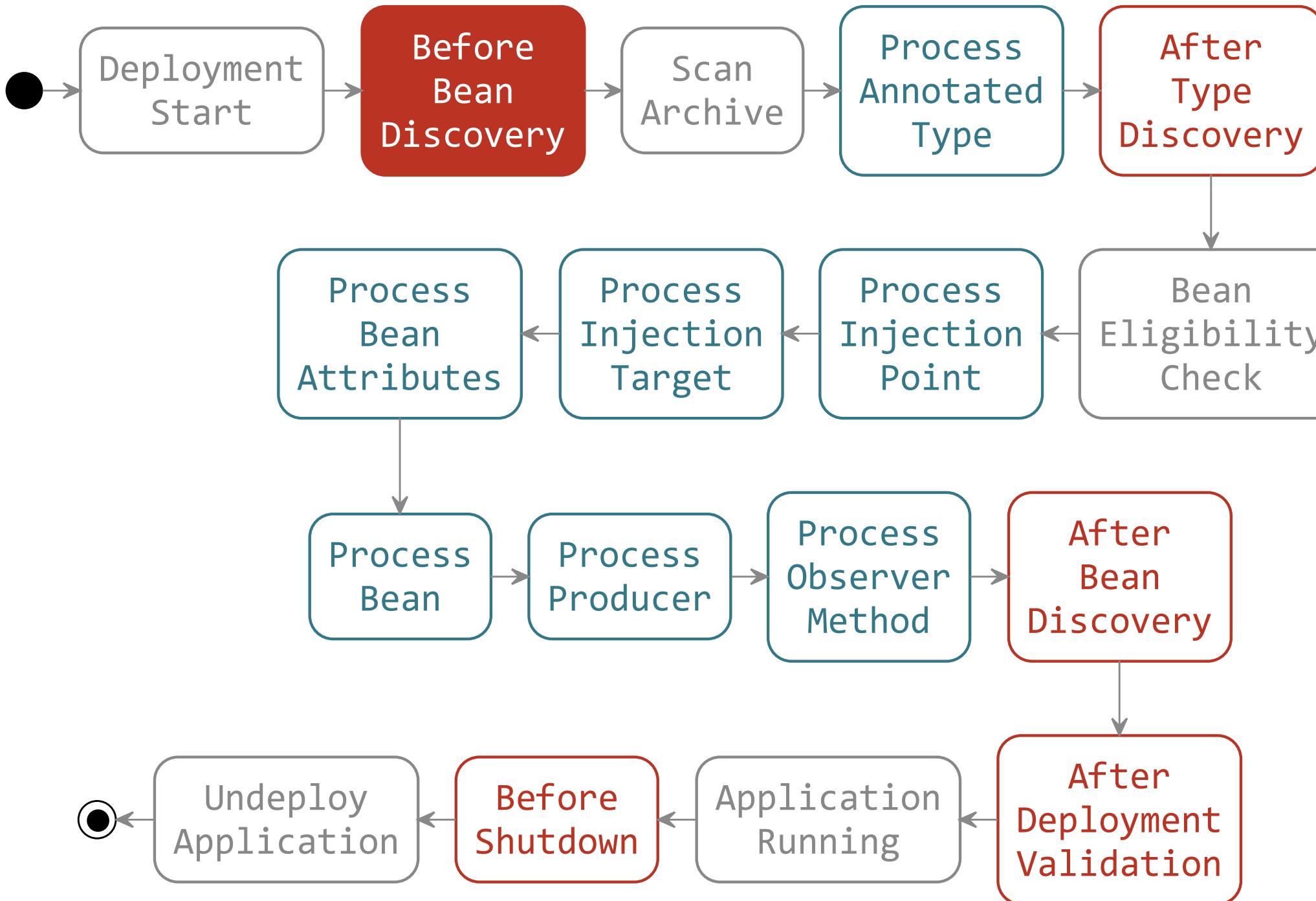
- 💡 By observing `BeforeBeanDiscovery` lifecycle event in an extension

```
public interface BeforeBeanDiscovery {  
  
    addQualifier(Class<? extends Annotation> qualifier); ①  
    addQualifier(AnnotatedType<? extends Annotation> qualifier);  
    addScope(Class<? extends Annotation> scopeType, boolean normal, boolean passivation);  
    addStereotype(Class<? extends Annotation> stereotype, Annotation... stereotypeDef);  
    addInterceptorBinding(AnnotatedType<? extends Annotation> bindingType);  
    addInterceptorBinding(Class<? extends Annotation> bindingType, Annotation... bindingTypeDef);  
    addAnnotatedType(AnnotatedType<?> type);  
    addAnnotatedType(AnnotatedType<?> type, String id);  
}
```

- ① This method is the one we need to declare `@Metric` as qualifier

- 💡 And use `addQualifier()` method in the event

# BeforeBeanDiscovery is first in lifecycle



Internal Step

Happen Once

Loop on Elements

# Our first extension

- 💡 A CDI extension is a class implementing the `Extension` tag interface

`org.cdi.further.metrics.MetricsExtension`

```
public class MetricsExtension implements Extension {  
  
    void addMetricAsQualifier(@Observes BeforeBeanDiscovery bdd) {  
        bdd.addQualifier(Metric.class);  
    }  
}
```

- 💡 Extension is activated by adding this file to `META-INF/services`

`javax.enterprise.inject.spi.Extension`

```
org.cdi.further.metrics.MetricsExtension
```

# Goal 1 achieved

💡 We can now write:

```
@Produces  
@Metric(name = "myTimer")  
Timer timer = new Timer(new SlidingTimeWindowReservoir(1L, MINUTES));
```

```
@Produces  
@Metric(name = "mySecondTimer")  
Timer timer = new Timer(new SlidingTimeWindowReservoir(1L, HOURS));
```

```
@Inject  
@Metric(name = "myTimer")  
Timer timer;
```

💡 And have the `Timer` injection points satisfied

## **GOAL 2 Apply Metrics with provided annotations**

# Goal 2 in detail

- 💡 We want to be able to write:

```
@Timed("timer") ①  
void timedMethod() {  
    //...  
}
```

- 💡 And have the timer "timer" activated during method invocation

- 🔥 The solution is to build an interceptor and bind it to @Timed

# Goal 2 step by step

- 💡 Create an interceptor for the timer technical code
- 💡 Make `@Timed` (provided by Metrics) a valid interceptor binding
- 💡 Programmatically add `@Timed` as an interceptor binding
- 💡 Use the magic

# Preparing interceptor creation

💡 We should find the **technical code** that will wrap the **business code**

```
class TimedMethodBean {  
  
    @Inject MetricRegistry registry;  
  
    void timedMethod() {  
        Timer timer = registry.timer("timer");  
        Timer.Context time = timer.time();  
        try {  
            // Business code  
        } finally {  
            time.stop();  
        }  
    }  
}
```

# Creating the interceptor

 Interceptor code is highlighted below

```
@Interceptor  
class TimedInterceptor {  
    @Inject MetricRegistry registry; ①  
    @AroundInvoke  
    Object timedMethod(InvocationContext context) throws Exception {  
        Timer timer = registry.timer(context.getMethod().getAnnotation(Timed.class).name());  
        Timer.Context time = timer.time();  
        try {  
            return context.proceed(); ②  
        } finally {  
            time.stop();  
        }  
    }  
}
```

① In CDI an interceptor is a bean, you can inject other beans in it

② Here the **business** of the application is called. All the code around is the **technical** one.

# Activating the interceptor

```
@Interceptor  
① @Priority(Interceptor.Priority.LIBRARY_BEFORE)  
class TimedInterceptor {  
  
    @Inject  
    MetricRegistry registry;  
  
    @AroundInvoke  
    Object timedMethod(InvocationContext context) throws Exception {  
        Timer timer = registry.timer(context.getMethod().getAnnotation(Timed.class).name());  
        Timer.Context time = timer.time();  
        try {  
            return context.proceed();  
        } finally {  
            time.stop();  
        }  
    }  
}
```

- 1 Giving a `@Priority` to an interceptor activates and orders it

# Add a binding to the interceptor

```
@Timed ①
@Interceptor
@Priority(Interceptor.Priority.LIBRARY_BEFORE)
class TimedInterceptor {

    @Inject
    MetricRegistry registry;

    @AroundInvoke
    Object timedMethod(InvocationContext context) throws Exception {
        Timer timer = registry.timer(context.getMethod().getAnnotation(Timed.class).name());
        Timer.Context time = timer.time();
        try {
            return context.proceed();
        } finally {
            time.stop();
        }
    }
}
```

- ① We'll use Metrics `@Timed` annotation as interceptor binding

## Back on interceptor binding

- 💡 An **interceptor binding** is an annotation used in 2 places:
  1. On the **interceptor class** to bind it to this annotation
  2. On the **methods** or **classes** to be intercepted by this interceptor
- 💡 An interceptor binding should have the `@InterceptorBinding` annotation or should be declared programmatically
- 💡 If the interceptor binding annotation has members:
  1. Their values are **taken into account** to resolve interceptor
  2. Unless members are annotated with `@NonBinding`

## @Timed source code is not an interceptor binding

```
@Documented  
@Retention(RetentionPolicy.RUNTIME)  
@Target({ ElementType.TYPE, ElementType.CONSTRUCTOR, ElementType.METHOD,  
ElementType.ANNOTATION_TYPE })  
①  
public @interface Timed {  
  
    String name() default ""; ②  
  
    boolean absolute() default false; ②  
}
```

① Lack of `@InterceptorBinding` annotation

② None of the members have the `@NonBinding` annotation, so `@Timed(name = "timer1")` and  
`@Timed(name = "timer2")` will be 2 different interceptor bindings

# The required `@Timed` source code to make it an interceptor binding

```
@Documented  
@Retention(RetentionPolicy.RUNTIME)  
@Target({ ElementType.TYPE, ElementType.CONSTRUCTOR, ElementType.METHOD,  
ElementType.ANNOTATION_TYPE })  
@InterceptorBinding  
public @interface Timed {  
  
    @NonBinding String name() default "";  
  
    @NonBinding boolean absolute() default false;  
}
```

❓ How to obtain the required `@Timed`?

🚫 We cannot touch the component source / binary!

# Using the `AnnotatedType` SPI

💡 Thanks to DeltaSpike we can easily create the required `AnnotatedType`

```
AnnotatedType createTimedAnnotatedType() throws Exception {  
    Annotation nonBinding = new AnnotationLiteral<Nonbinding>() {};  
    return new AnnotatedTypeBuilder().readFromType(Timed.class) 1  
        .addToManyElements(Timed.class.getMethod("name"), nonBinding) 2  
        .addToManyElements(Timed.class.getMethod("absolute"), nonBinding) 3  
        .create();  
}
```

- 1 This creates an instance of `@NonBinding` annotation
- 2 It would have been possible but far more verbose to create this `AnnotatedType` without the help of DeltaSpike. The `AnnotatedTypeBuilder` is initialized from the Metrics `@Timed` annotation.
- 3 `@NonBinding` is added to both members of the `@Timed` annotation

## This extension will do the job

💡 We observe `BeforeBeanDiscovery` to add a new interceptor binding

```
public class MetricsExtension implements Extension {  
  
    void addTimedBinding(@Observes BeforeBeanDiscovery bdd) throws Exception {  
        Annotation nonBinding = new AnnotationLiteral<Nonbinding>() {};  
  
        bdd.addInterceptorBinding(new AnnotatedTypeBuilder<Timed>()  
            .readFromType(Timed.class)  
            .addToManyMethod(Timed.class.getMethod("name"), nonBinding)  
            .addToManyMethod(Timed.class.getMethod("absolute"), nonBinding)  
            .create());  
    }  
}
```

# Goal 2 achieved

💡 We can now write:

```
@Timed("timer")
void timedMethod() {
    // Business code
}
```

And have a Metrics **Timer** applied to the method

- 🔥 Interceptor should be enhanced to support **@Timed** on a class
- 🔥 Other interceptor should be developed for other metrics

# Our goals

## 1. Apply a metric with the provided annotation in AOP style

```
@Timed("timer") ①  
void timedMethod() {  
    //...  
}
```

## 2. Register automatically produced custom metrics

```
@Produces @Metric(name = "myTimer") ①  
Timer timer = new Timer(new SlidingTimeWindowReservoir(1L, MINUTES));  
//...  
@Timed("myTimer") ①  
void timedMethod() { /*...*/ }
```

① Annotations provided by Metrics

**GOAL 3 Access same Metric through `@Inject`  
or `MetricRegistry` getter**

# Goal 3 in detail

## 💡 When Writing:

```
@Inject  
@Metric(name = "myTimer")  
Timer timer1;
```

```
@Inject  
MetricRegistry registry;
```

```
Timer timer2 = registry.timer("myTimer");
```

💡 ... We want that `timer1 == timer2`

## Goal 3 in detail

```
@Produces @Metric(name = "myTimer") ①  
Timer timer = new Timer(new SlidingTimeWindowReservoir(1L, TimeUnit.MINUTES));  
  
@Inject  
@Metric(name = "myTimer")  
Timer timer;  
  
@Inject  
MetricRegistry registry;  
  
Timer timer = registry.timer("myTimer"); ②
```

- ① Produced `Timer` should be added to `MetricRegistry` when produced
- ② When retrieved from registry a `Metric` should be **identical** to the produced instance and vice versa

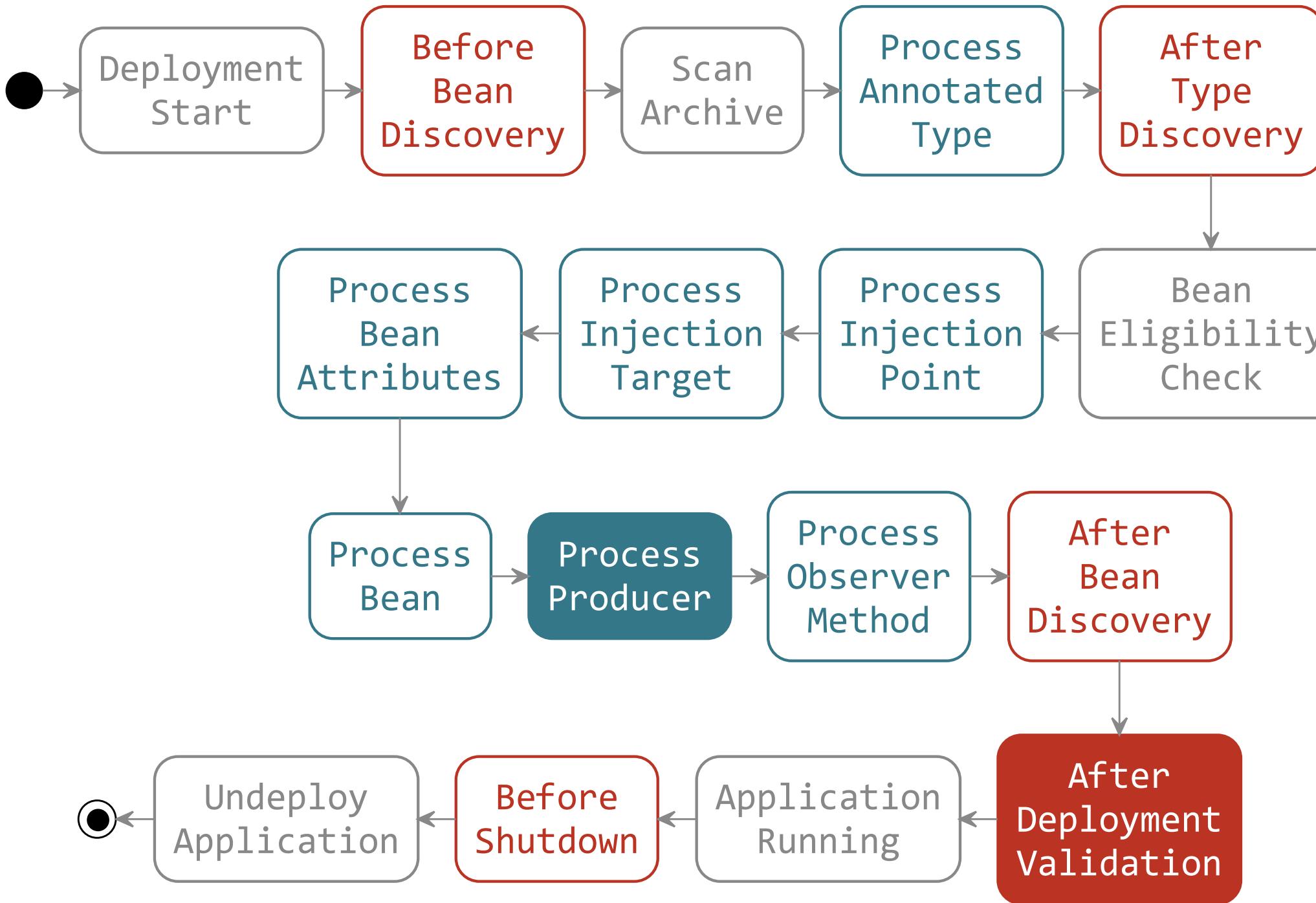
 There are 2 `Metric`: the `com.codahale.metrics.Metric` interface and the `com.codahale.metrics.annotation.Metric` annotation

## Goal 3 step by step

💡 We need to write an extension that will:

1. Change how a `Metric` instance is produced by looking it up in the registry first and producing (and registering) it only if it's not found.  
We'll do this by:
  1. observing the `ProcessProducer` lifecycle event
  2. decorating Metric `Producer` to add this new behavior
2. Produce all `Metric` instances at the end of boot time to have them in registry for runtime
  1. we'll do this by observing `AfterDeploymentValidation` event

# So we will `@Observes` these 2 events to add our features



Internal Step

Happen Once

Loop on Elements

# Customizing **Metric** producing process

- i We first need to create our implementation of the **Producer<X>** SPI

```
class MetricProducer<X extends Metric> implements Producer<X> {  
  
    final Producer<X> decorate;  
  
    final String metricName;  
  
    MetricProducer(Producer<X> decorate, String metricName) {  
        this.decorate = decorate;  
        this.metricName = metricName;  
    }  
  
    //...  
}
```

# Customizing Metric producing process (continued)

```
//...
public X produce(CreationalContext<X> ctx) { ①
    MetricRegistry reg = BeanProvider.getContextualReference(MetricRegistry.class, false); ②
    if (!reg.getMetrics().containsKey(metricName)) ③
        reg.register(metricName, decorate.produce(ctx));

    return (X) reg.getMetrics().get(metricName);
}

public void dispose(X instance) { }

public Set<InjectionPoint> getInjectionPoints() {
    return decorate.getInjectionPoints();
}
}
```

- ① Produce method will be used by the container at runtime to decorate declared producer with our logic
- ② `BeanProvider` is a DeltaSpike helper class to easily retrieve a bean or bean instance
- ③ If metric name is not in the registry, the original producer is called and its result is added to registry

## We'll use our `MetricProducer` in a `ProcessProducer` observer

- i This event allow us to substitute the original producer by ours

```
public interface ProcessProducer<T, X> {  
  
    AnnotatedMember<T> getAnnotatedMember(); ①  
  
    Producer<X> getProducer(); ②  
  
    void setProducer(Producer<X> producer); ③  
  
    void addDefinitionError(Throwable t);  
}
```

- ① Gets the `AnnotatedMember` associated to the `@Produces` field or method
- ② Gets the default producer (useful to decorate it)
- ③ Overrides the producer

# Customizing `Metric` producing process (end)

💡 Here's the extension code to do this producer decoration

```
public class MetricsExtension implements Extension {  
    //...  
    <X extends Metric> void decorateMetricProducer(@Observes ProcessProducer<?, X> pp) {  
        String name = pp.getAnnotatedMember().getAnnotation(Metric.class).name(); ①  
        pp.setProducer(new MetricProducer<>(pp.getProducer(), name)); ②  
    }  
    //...  
}
```

- ① We retrieve metric name from the `name()` member in `@Metric`
- ② We replace the original producer by our producer (which decorates the former)

# Producing all the `Metric` instances at the end of boot time

- i We do that by observing the `AfterDeploymentValidation` event

```
public class MetricsExtension implements Extension {  
    //...  
    void registerProducedMetrics(@Observes AfterDeploymentValidation adv) {  
        List<Metric> metrics = BeanProvider.getContextualReferences(Metric.class, true); ①  
    }  
    //...  
}
```

- 1 `getContextualReferences()` is a method in **DeltaSpike** `BeanProvider` helper class.  
It creates the list of bean instances for a given bean type (ignoring qualifiers)

# Goal 3 achieved

💡 We can now write:

```
@Produces @Metric(name = "myTimer")
Timer timer = new Timer(new SlidingTimeWindowReservoir(1L, TimeUnit.MINUTES));

@Inject
MetricRegistry registry;

@Inject @Metric("myTimer")
Metric timer;
```

💡 And be sure that `registry.getMetrics().get("myTimer")` and `timer` are the same object (our custom `Timer`)

# Complete extension code

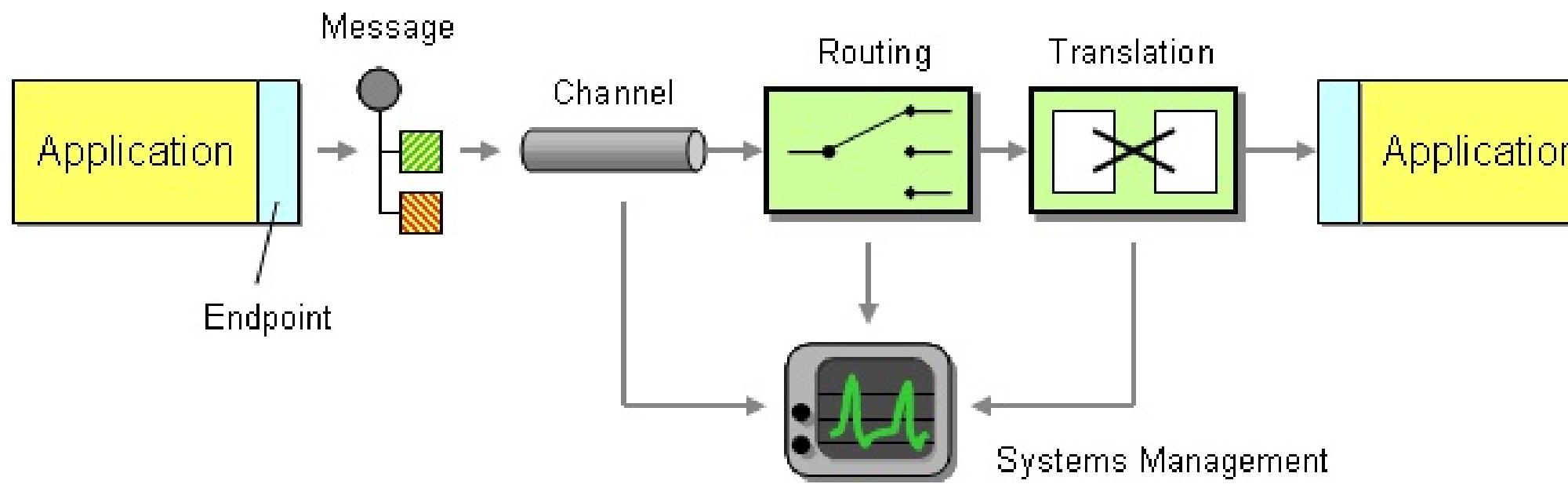
```
public class MetricsExtension implements Extension {  
  
    void addMetricAsQualifier(@Observes BeforeBeanDiscovery bdd) {bdd.addQualifier(Metric.class);}  
  
    void addTimedBinding(@Observes BeforeBeanDiscovery bdd) throws Exception {  
        Annotation nonBinding = new AnnotationLiteral<Nonbinding>() {};  
        bdd.addInterceptorBinding(new AnnotatedTypeBuilder<Timed>().readFromType(Timed.class)  
            .addToMethod(Timed.class.getMethod("name"), nonBinding)  
            .addToMethod(Timed.class.getMethod("absolute"),nonBinding).create());  
    }  
  
<T extends com.codahale.metrics.Metric> void decorateMetricProducer(@Observes ProcessProducer<?, T> pp) {  
    if (pp.getAnnotatedMember().isAnnotationPresent(Metric.class)) {  
        String name = pp.getAnnotatedMember().getAnnotation(Metric.class).name();  
        pp.setProducer(new MetricProducer(pp.getProducer(), name));  
    }  
}  
  
void registerProducedMetrics(@Observes AfterDeploymentValidation adv) {  
    List<com.codahale.metrics.Metric> metrics = BeanProvider  
        .getContextualReferences(com.codahale.metrics.Metric.class, true);  
}  
}
```

**How to use CDI as dependency injection container  
for an integration framework (Apache Camel)**

**Camel CDI**

# About Apache Camel

- Open-source **integration framework** based on known Enterprise Integration Patterns
- Provides a variety of DSLs to write routing and mediation rules
- Provides support for **bean binding** and seamless integration with DI frameworks



**Discover how we created CDI  
integration module for Camel**

# Camel out of the box (without CDI)

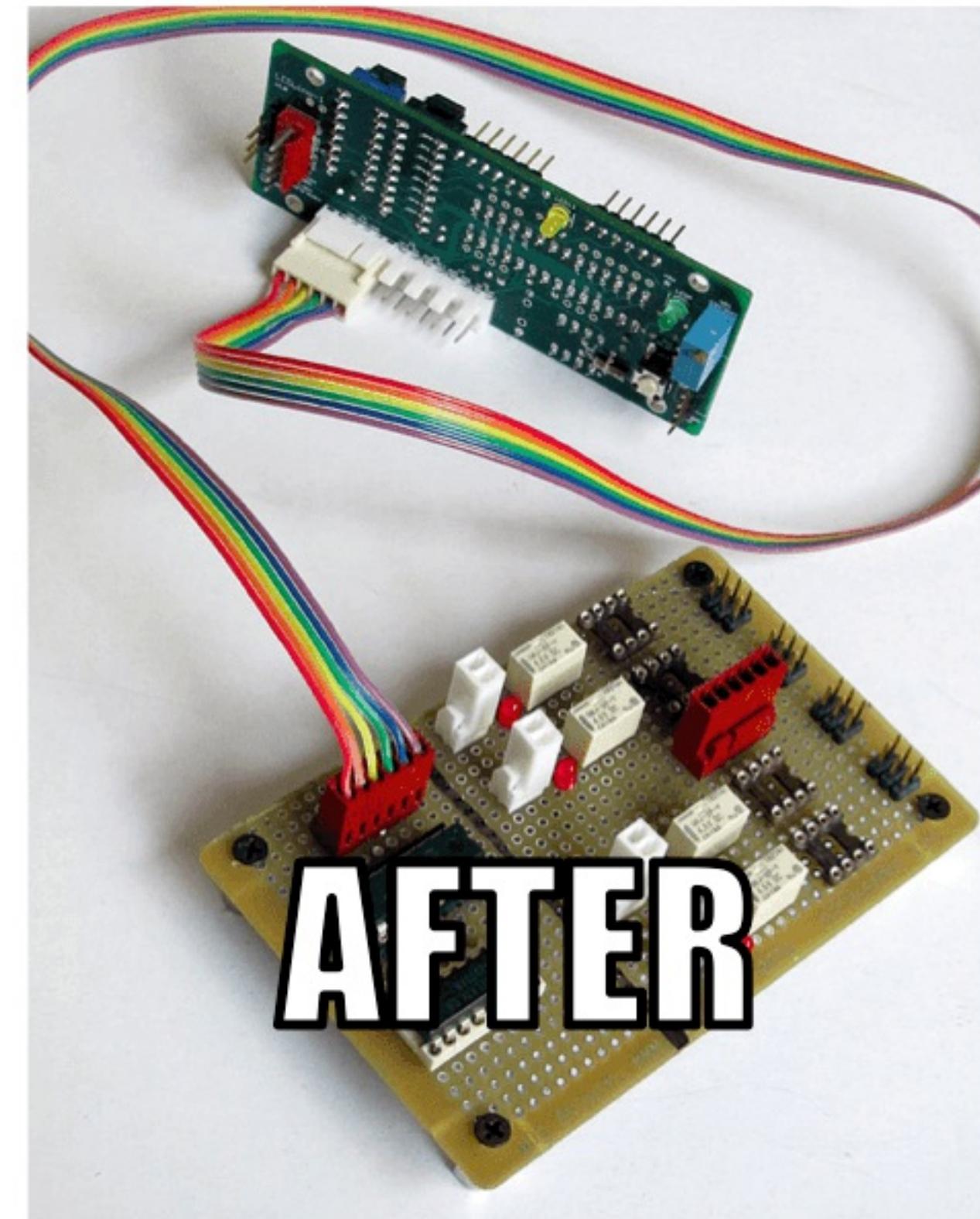
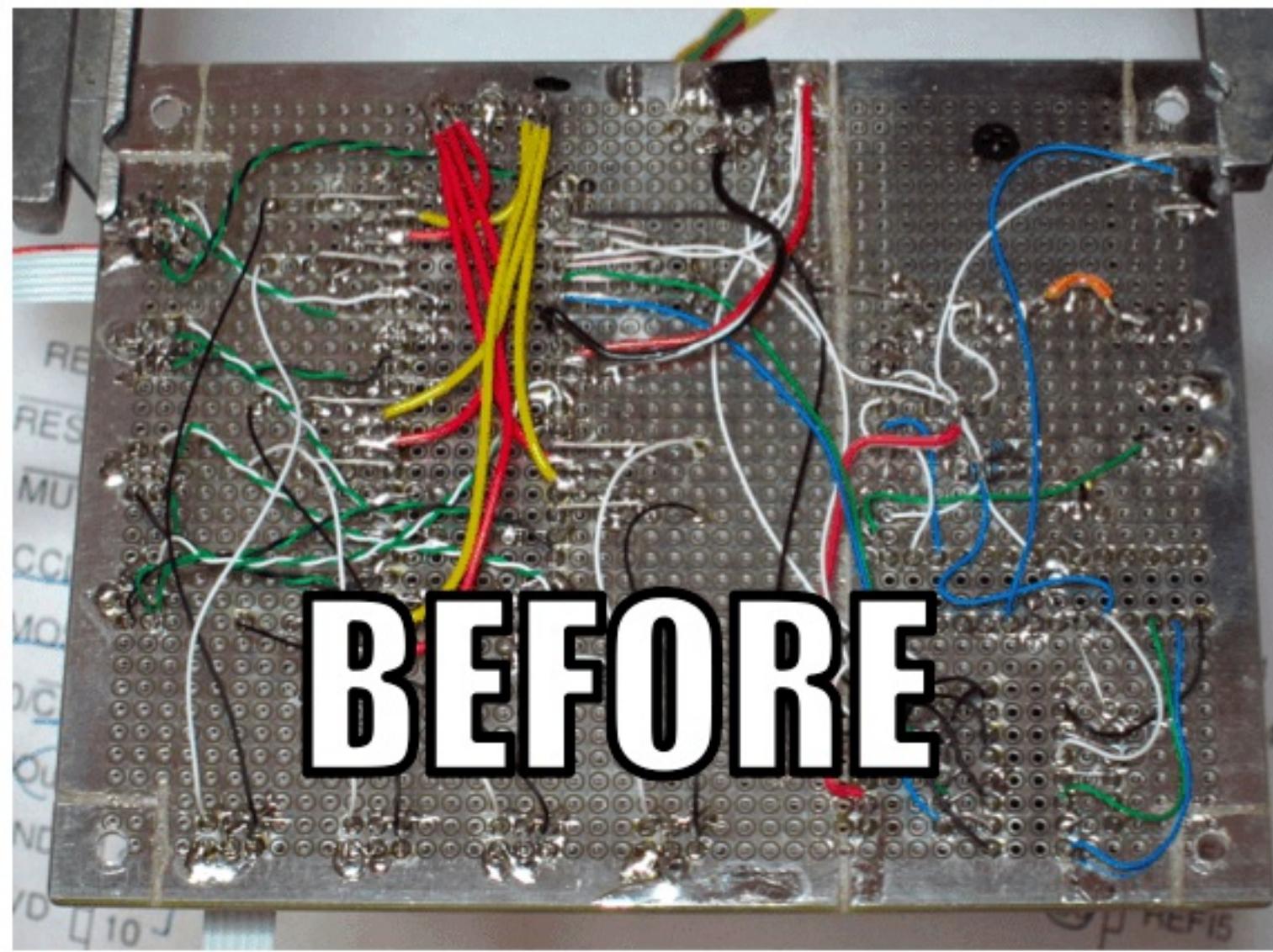
```
public static void main(String[] args) {
    CamelContext context = new DefaultCamelContext();
    context.addRoutes(new RouteBuilder() {
        public void configure() {
            from("file:target/input?delay=1000")
                .log("Sending message [${body}] to JMS ...")
                .to("sjms:queue:output"); ①
        }
    });
    PropertiesComponent properties = new PropertiesComponent();
    properties.setLocation("classpath:camel.properties");
    context.addComponent("properties", properties); // Registers the "properties" component

    SjmsComponent component = new SjmsComponent();
    component.setConnectionFactory(new ActiveMQConnectionFactory("vm://broker?broker.persistent=false"));
    jms.setConnectionCount(Integer.valueOf(context.resolvePropertyPlaceholders("{{jms.maxConnections}}")));
    context.addComponent("sjms", jms); // Registers the "sjms" component

    context.start();
}
```

① This route watches a directory every second and sends new files content to a JMS queue

Why CDI?



## Basic CDI integration (1/3)

1. Camel components and route builder as CDI beans
2. Bind the Camel context lifecycle to that of the CDI container

```
class FileToJmsRouteBean extends RouteBuilder {  
  
    @Override  
    public void configure() {  
        from("file:target/input?delay=1000")  
            .log("Sending message [${body}] to JMS...")  
            .to("sjms:queue:output");  
    }  
}
```

## Basic CDI integration (2/3)

```
class PropertiesComponentFactoryBean {
```

```
    @Produces  
    @ApplicationScoped  
    PropertiesComponent propertiesComponent() {  
        PropertiesComponent properties = new PropertiesComponent();  
        properties.setLocation("classpath:camel.properties");  
        return properties;  
    }  
}
```

```
class JmsComponentFactoryBean {
```

```
    @Produces  
    @ApplicationScoped  
    SjmsComponent sjmsComponent(PropertiesComponent properties) throws Exception {  
        SjmsComponent jms = new SjmsComponent();  
        jms.setConnectionFactory(new ActiveMQConnectionFactory("vm://broker?broker.persistent=false"));  
        jms.setConnectionCount(Integer.valueOf(properties.parseUri("{{jms.maxConnections}}")));  
        return component;  
    }  
}
```

## Basic CDI integration (3/3)

```
@ApplicationScoped
class CamelContextBean extends DefaultCamelContext {

    @Inject
    CamelContextBean(FileToJmsRouteBean route, SjmsComponent jms, PropertiesComponent properties) {
        addComponent("properties", properties);
        addComponent("sjms", jms);
        addRoutes(route);
    }
    @PostConstruct
    void startContext() {
        super.start();
    }
    @PreDestroy
    void preDestroy() {
        super.stop();
    }
}
```



We could have a lot more with advanced **CDI** features

## Our goals

1. Avoid assembling and configuring the `CamelContext` manually
2. Access CDI beans from the Camel DSL automatically

```
.to("sjms:queue:output"); // Lookup by name (sjms) and type (Component)
```

```
context.resolvePropertyPlaceholders("{{jms.maxConnections}}");  
// Lookup by name (properties) and type (Component)
```

3. Support Camel annotations in CDI beans

```
@PropertyInject(value = "jms.maxConnections", defaultValue = "10")  
int maxConnections;
```

# Steps to integrate Camel and CDI

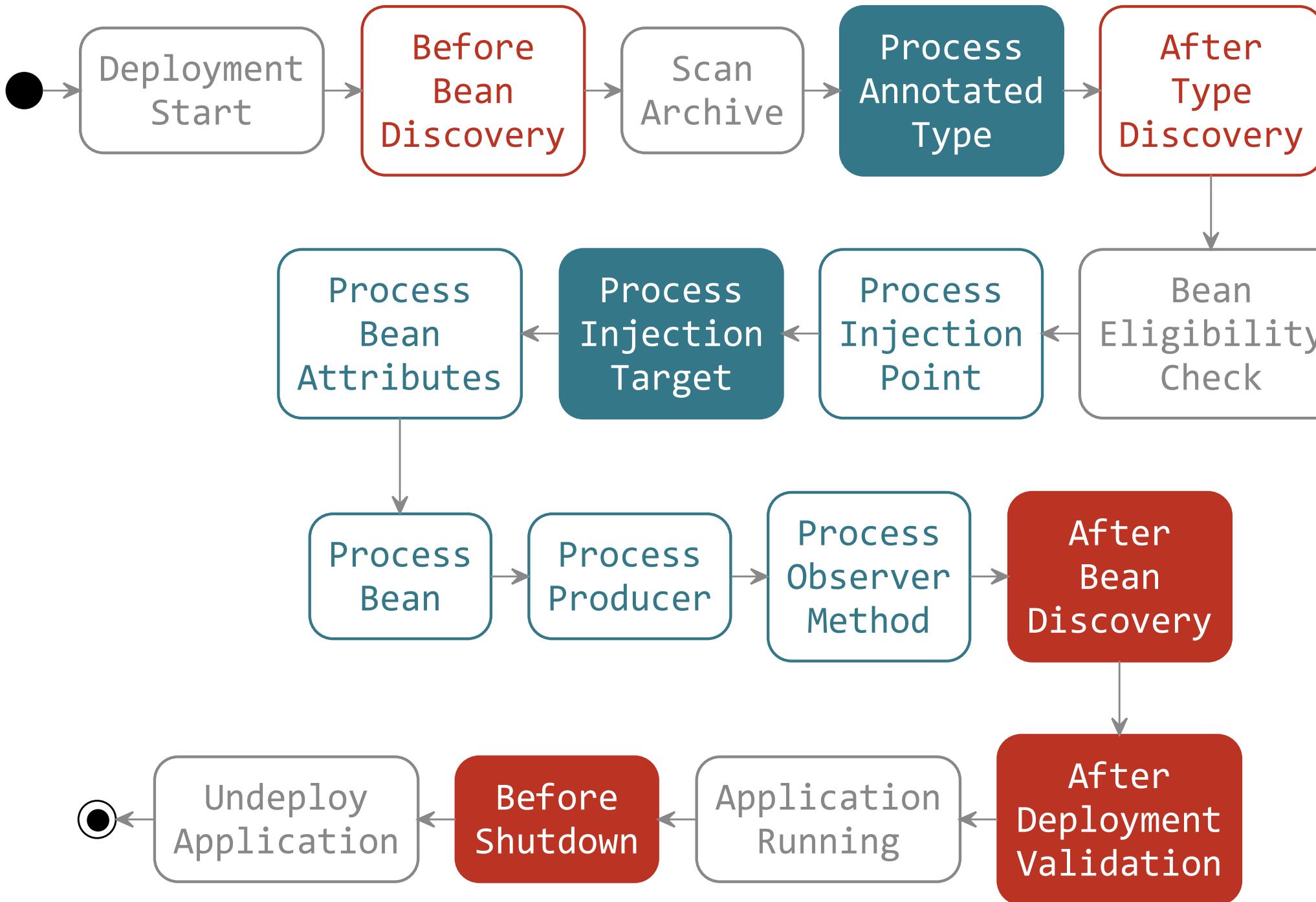
- 💡 Manage the creation and the configuration of the `CamelContext` bean
- 💡 Bind the `CamelContext` lifecycle to that of the CDI container
- 💡 Implement the Camel registry SPI to look up CDI bean references
- 💡 Use a custom `InjectionTarget` for CDI beans containing Camel annotations
- 💡 Use the magic

## How to achieve this?

💡 We need to write an extension that will:

1. Declare a `CamelContext` bean by observing the `AfterBeanDiscovery` lifecycle event
2. Instantiate the beans of type `RouteBuilder` and add them to the Camel context
3. Start (resp. stop) the Camel context when the `AfterDeploymentValidation` event is fired (resp. the `BeforeShutdown` event)
4. Customize the Camel context to query the `BeanManager` to lookup CDI beans by name and type
5. Detect CDI beans containing Camel annotations by observing the `ProcessAnnotatedType` event and modify how they get injected by observing the `ProcessInjectionTarget` lifecycle event

# So we will `@Observes` these 5 events to add our features



Internal Step

Happen Once

Loop on Elements

## Adding the `CamelContext` bean

- 💡 Automatically add a `CamelContext` bean in the deployment archive

❓ How to add a bean programmatically?

# Declaring a bean programmatically

💡 We need to implement the **Bean** SPI

```
public interface Bean<T> extends Contextual<T>, BeanAttributes<T> {  
  
    Class<?> getBeanClass();  
    Set<InjectionPoint> getInjectionPoints();  
    T create(CreationalContext<T> creationalContext); // Contextual<T>  
    void destroy(T instance, CreationalContext<T> creationalContext);  
    Set<Type> getTypes(); // BeanAttributes<T>  
    Set<Annotation> getQualifiers();  
    Class<? extends Annotation> getScope();  
    String getName();  
    Set<Class<? extends Annotation>> getStereotypes();  
    boolean isAlternative();  
}
```

# Implementing the Bean SPI

```
class CamelContextBean implements Bean<CamelContext> {

    public Class<? extends Annotation> getScope() { return ApplicationScoped.class; }

    public Set<Annotation> getQualifiers() {
        return Collections.<Annotation>singleton(new AnnotationLiteral<Default>() {});
    }
    public Set<Type> getTypes() { return Collections.<Type>singleton(CamelContext.class); }

    public CamelContext create(CreationalContext<CamelContext> creational) {
        return new DefaultCamelContext();
    }
    public void destroy(CamelContext instance, CreationalContext<CamelContext> creational) {}

    public Class<?> getBeanClass() { return DefaultCamelContext.class; }

    public Set<InjectionPoint> getInjectionPoints() { return Collections.emptySet(); }

    public Set<Class<? extends Annotation>> getStereotypes() { return Collections.emptySet(); }
    public String getName() { return null; } // Only called for @Named bean
    public boolean isAlternative() { return false; }
    public boolean isNullable() { return false; }
}
```

# Adding a programmatic bean to the deployment

- Then add the `CamelContextBean` bean programmatically by observing the `AfterBeanDiscovery` lifecycle event

```
public class CamelExtension implements Extension {  
  
    void addCamelContextBean(@Observes AfterBeanDiscovery abd) {  
        abd.addBean(new CamelContextBean());  
    }  
}
```

# Instantiate and assemble the Camel context

- Instantiate the `CamelContext` bean and the `RouteBuilder` beans in the `AfterDeploymentValidation` lifecycle event

```
public class CamelExtension implements Extension {  
    //...  
    void configureContext(@Observes AfterDeploymentValidation adv, BeanManager bm) {  
        CamelContext context = getReference(bm, CamelContext.class);  
        for (Bean<?> bean : bm.getBeans(RoutesBuilder.class))  
            context.addRoutes(getReference(bm, RouteBuilder.class, bean));  
    }  
    <T> T getReference(BeanManager bm, Class<T> type) {  
        return getReference(bm, type, bm.resolve(bm.getBeans(type)));  
    }  
    <T> T getReference(BeanManager bm, Class<T> type, Bean<?> bean) {  
        return (T) bm.getReference(bean, type, bm.createCreationalContext(bean));  
    }  
}
```

# Managed the Camel context lifecycle

- 💡 Start (resp. stop) the Camel context when the `AfterDeploymentValidation` event is fired (resp. the `BeforeShutdown`)

```
public class CamelExtension implements Extension {  
    //...  
    void configureContext(@Observes AfterDeploymentValidation adv, BeanManager bm) {  
        CamelContext context = getReference(bm, CamelContext.class);  
        for (Bean<?> bean : bm.getBeans(RoutesBuilder.class)  
            context.addRoutes(getReference(bm, RouteBuilder.class, bean));  
        context.start();  
    }  
    void stopCamelContext(@Observes BeforeShutdown bs, BeanManager bm) {  
        CamelContext context = getReference(bm, CamelContext.class);  
        context.stop();  
    }  
}
```

# First goal achieved

💡 We can get rid of the following code:

```
@ApplicationScoped
class CamelContextBean extends DefaultCamelContext {
    @Inject
    CamelContextBean(FileToJmsRouteBean route, SjmsComponent jms, PropertiesComponent properties) {
        addComponent("properties", propertiesComponent);
        addComponent("sjms", sjmsComponent);
        addRoutes(route);
    }
    @PostConstruct
    void startContext() {
        super.start();
    }
    @PreDestroy
    void stopContext() {
        super.stop();
    }
}
```

## Second goal: Access CDI beans from the Camel DSL

### ? How to retrieve CDI beans from the Camel DSL?

```
.to("sjms:queue:output"); // Lookup by name (sjms) and type (Component)
context.resolvePropertyPlaceholders("{{jms.maxConnections}}");
// Lookup by name (properties) and type (Component)

// And also...
.bean(MyBean.class); // Lookup by type and Default qualifier
.beanRef("beanName"); // Lookup by name
```

- 💡 Implement the Camel registry SPI and use the **BeanManager** to lookup for CDI bean contextual references by name and type

# Implement the Camel registry SPI

```
class CamelCdiRegistry implements Registry {  
    private final BeanManager bm;  
  
    CamelCdiRegistry(BeanManager bm) { this.bm = bm; }  
  
    public <T> T lookupByNameAndType(String name, Class<T> type) {  
        return getReference(bm, type, bm.resolve(bm.getBeans(name)));  
    }  
    public <T> Set<T> findByType(Class<T> type) {  
        return getReference(bm, type, bm.resolve(bm.getBeans(type)));  
    }  
    public Object lookupByName(String name) {  
        return lookupByNameAndType(name, Object.class);  
    }  
    <T> T getReference(BeanManager bm, Type type, Bean<?> bean) {  
        return (T) bm.getReference(bean, type, bm.createCreationalContext(bean));  
    }  
}
```

## Add the CamelCdiRegistry to the Camel context

```
class CamelContextBean implements Bean<CamelContext> {
    private final BeanManager bm;

    CamelContextBean(BeanManager bm) { this.bm = bm; }
    //...
    public CamelContext create(CreationalContext<CamelContext> creational) {
        return new DefaultCamelContext(new CamelCdiRegistry(bm));
    }
}

public class CamelExtension implements Extension {
    //...
    void addCamelContextBean(@Observes AfterBeanDiscovery abd, BeanManager bm) {
        abd.addBean(new CamelContextBean(bm));
    }
}
```

# Second goal achieved 1/3

💡 We can declare the `sjms` component with the `@Named` qualifier

```
class JmsComponentFactoryBean {

    @Produces
    @Named("sjms")
    @ApplicationScoped
    SjmsComponent sjmsComponent(PropertiesComponent properties) {
        SjmsComponent jms = new SjmsComponent();
        jms.setConnectionFactory(new ActiveMQConnectionFactory("vm://broker?..."));
        jms.setConnectionCount(Integer.valueOf(properties.parseUri("{{jms.maxConnections}}")));
        return component;
    }
}
```

...

# Second goal achieved 2/3

 Declare the **properties** component with the **@Named** qualifier

```
class PropertiesComponentFactoryBean {  
  
    @Produces  
    @Named("properties")  
    @ApplicationScoped  
    PropertiesComponent propertiesComponent() {  
        PropertiesComponent properties = new PropertiesComponent();  
        properties.setLocation("classpath:camel.properties");  
        return properties;  
    }  
}
```

...

# Second goal achieved 3/3

- 💡 And get rid of the code related to the `properties` and `sjms` components registration

```
@ApplicationScoped
class CamelContextBean extends DefaultCamelContext {
    @Inject
    CamelContextBean(FileToJmsRouteBean route, SjmsComponent jms, PropertiesComponent properties) {
        addComponent("properties", propertiesComponent);
        addComponent("sjms", sjmsComponent);
        addRoutes(route);
    }
    @PostConstruct
    void startContext() {
        super.start();
    }
    @PreDestroy
    void stopContext() {
        super.stop();
    }
}
```

## Third goal: Support Camel annotations in CDI beans

- 💡 Camel provides a set of DI framework agnostic annotations for resource injection

```
@PropertyInject(value = "jms.maxConnections", defaultValue = "10")
int maxConnections;

// But also...
@EndpointInject(uri = "jms:queue:foo")
Endpoint endpoint;

@BeanInject("foo")
FooBean foo;
```

❓ How to support custom annotations injection?

# How to support custom annotations injection?

- 💡 Create a custom `InjectionTarget` that uses the default Camel bean post processor `DefaultCamelBeanPostProcessor`

```
public interface InjectionTarget<T> extends Producer<T> {  
    void inject(T instance, CreationalContext<T> ctx);  
    void postConstruct(T instance);  
    void preDestroy(T instance);  
}
```

- 💡 Hook it into the CDI injection mechanism by observing the `ProcessInjectionTarget` lifecycle event

- 💡 Only for beans containing Camel annotations by observing the `ProcessAnnotatedType` lifecycle and using `@WithAnnotations`

# Create a custom `InjectionTarget`

```
class CamelInjectionTarget<T> implements InjectionTarget<T> {  
  
    final InjectionTarget<T> delegate;  
  
    final DefaultCamelBeanPostProcessor processor;  
  
    CamelInjectionTarget(InjectionTarget<T> target, final BeanManager bm) {  
        delegate = target;  
        processor = new DefaultCamelBeanPostProcessor() {  
            public CamelContext getOrLookupCamelContext() {  
                return getReference(bm, CamelContext.class);  
            }  
        };  
    }  
    public void inject(T instance, CreationalContext<T> ctx) {  
        processor.postProcessBeforeInitialization(instance, null); ①  
        delegate.inject(instance, ctx);  
    }  
    //...  
}
```

- ① Call the Camel default bean post-processor before CDI injection

## Register the custom `InjectionTarget`

- 💡 Observe the `ProcessInjectionTarget` lifecycle event and set the `InjectionTarget`

```
public interface ProcessInjectionTarget<X> {  
    AnnotatedType<X> getAnnotatedType();  
    InjectionTarget<X> getInjectionTarget();  
    void setInjectionTarget(InjectionTarget<X> injectionTarget);  
    void addDefinitionError(Throwable t);  
}
```

- 💡 To decorate it with the `CamelInjectionTarget`

```
class CamelExtension implements Extension {  
  
    <T> void camelBeansPostProcessor(@Observes ProcessInjectionTarget<T> pit, BeanManager bm) {  
        pit.setInjectionTarget(new CamelInjectionTarget<>(pit.getInjectionTarget(), bm));  
    }  
}
```

# But only for beans containing Camel annotations

```
class CamelExtension implements Extension {  
    final Set<AnnotatedType<?>> camelBeans = new HashSet<>();  
  
    void camelAnnotatedTypes(@Observes @WithAnnotations(PropertyInject.class)  
        ProcessAnnotatedType<?> pat) { ①  
        camelBeans.add(pat.getAnnotatedType());  
    }  
  
<T> void camelBeansPostProcessor(@Observes ProcessInjectionTarget<T> pit,  
    BeanManager bm) {  
    if (camelBeans.contains(pit.getAnnotatedType())) ②  
        pit.setInjectionTarget(  
            new CamelInjectionTarget<>(pit.getInjectionTarget(), bm));  
    }  
}
```

- ① Detect all the types containing Camel annotations with `@WithAnnotations`
- ② Decorate the `InjectionTarget` corresponding to these types

# Third goal achieved 1/2

- 💡 Instead of injecting the `PropertiesComponent` bean to resolve a configuration property

```
class JmsComponentFactoryBean {

    @Produces
    @Named("sjms")
    @ApplicationScoped
    SjmsComponent sjmsComponent(PropertiesComponent properties) {
        SjmsComponent jms = new SjmsComponent();
        jms.setConnectionFactory(new ActiveMQConnectionFactory("vm://broker?..."));
        jms.setConnectionCount(Integer.valueOf(properties.parseUri("{{jms.maxConnections}}")));
        return component;
    }
}
```

# Third goal achieved 2/2

- We can directly rely on the `@PropertyInject` Camel annotation in CDI beans

```
class JmsComponentFactoryBean {

    @PropertyInject("jms.maxConnections")
    int maxConnections;

    @Produces
    @Named("sjms")
    @ApplicationScoped
    SjmsComponent sjmsComponent() {
        SjmsComponent component = new SjmsComponent();
        jms.setConnectionFactory(new ActiveMQConnectionFactory("vm://broker?..."));
        component.setConnectionCount(maxConnections);
        return component;
    }
}
```

## Bonus goal: Camel DSL AOP

### 💡 AOP instrumentation of the Camel DSL

```
from("file:target/input?delay=1000")
    .log("Sending message [${body}] to JMS...")
    .to("sjms:queue:output");
```

### 💡 With CDI observers

```
from("file:target/input?delay=1000")
    .to("sjms:queue:output").id("join point");
}
void advice(@Observes @NodeId("join point") Exchange exchange) {
    logger.info("Sending message [{}] to JMS...", exchange.getIn().getBody());
}
```

## How to achieve this?

💡 We can create a CDI qualifier to hold the Camel node id metadata:

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

💡 And create an extension that will:

1. Detect the CDI beans containing observer methods with the `@NodeId` qualifier by observing the `ProcessObserverMethod` event and collect the Camel processor nodes to be instrumented
2. Customize the Camel context by providing an implementation of the Camel `InterceptStrategy` interface that will fire a CDI event each time an `Exchange` is processed by the instrumented nodes

# Detect the Camel DSL AOP observer methods

💡 Observe the **ProcessObserverMethod** lifecycle event

```
public interface ProcessObserverMethod<T, X> {  
    AnnotatedMethod<X> getAnnotatedMethod();  
    ObserverMethod<T> getObserverMethod();  
    void addDefinitionError(Throwable t);  
}
```

💡 And collect the observer method metadata

```
class CamelExtension implements Extension {  
    final Set<NodeId> joinPoints = new HashSet<>();  
  
    void pointcuts(@Observes ProcessObserverMethod<Exchange, ?> pom) {  
        for (Annotation qualifier : pom.getObserverMethod().getObservedQualifiers())  
            if (qualifier instanceof NodeId)  
                joinPoints.add(NodeId.class.cast(qualifier));  
    }  
}
```

# Instrument the Camel context

## 💡 Intercept matching nodes and fire a CDI event

```
void configureCamelContext(@Observes AfterDeploymentValidation adv, final BeanManager manager) {  
    context.addInterceptStrategy(new InterceptStrategy() {  
        public Processor wrapProcessorInInterceptors(CamelContext context, ProcessorDefinition<?> definition,  
            Processor target, Processor nextTarget) throws Exception {  
            if (definition.hasCustomIdAssigned()) {  
                for (final Node node : joinPoints) {  
                    if (node.value().equals(definition.getId())) {  
                        return new DelegateAsyncProcessor(target) {  
                            public boolean process(Exchange exchange, AsyncCallback callback) {  
                                manager.fireEvent(exchange, node);  
                                return super.process(exchange, callback);  
                            }  
                        };  
                    }  
                }  
            }  
            return target;  
        }  
    });  
}
```

# Bonus goal achieved

💡 We can define join points in the Camel DSL

```
from("file:target/input?delay=1000")
    .to("sjms:queue:output").id("join point");
}
```

💡 And advise them with CDI observers

```
void advice(@Observes @NodeId("join point") Exchange exchange) {
    List<MessageHistory> history = exchange.getProperty(Exchange.MESSAGE_HISTORY, List.class);
    logger.info("Sending message [{}]\tto [{}]\t...", exchange.getIn().getBody(String.class),
               history.get(history.size() - 1).getNode().getLabel());
}
```

# Conclusion

## References

- ❶ CDI Specification - [cdi-spec.org](http://cdi-spec.org)
- ❶ Slides sources - [github.com/astefanutti/further-cdi](https://github.com/astefanutti/further-cdi)
- ❶ Metrics CDI sources - [github.com/astefanutti/metrics-cdi](https://github.com/astefanutti/metrics-cdi)
- ❶ Camel CDI sources - [github.com/astefanutti/camel-cdi](https://github.com/astefanutti/camel-cdi)
- ❶ Slides generated with **Asciidoctor**, **PlantUML** and **DZSlides** backend
- ❶ Original slide template - **Dan Allen & Sarah White**

# Antoine Sabot-Durand

# Antonin Stefanutti



@antoine\_sd @astefanut

# Annexes

# Complete lifecycle events

