Going Further with CDI 2.0

Antoine Sabot-Durand · Antonin Stefanutti

Antonin Stefanutti

- Software Engineer
- Red Hat
- JBoss Fuse team
- @astefanut
- github.com/astefanutti

Antoine Sabot-Durand

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`

More concretely

- What’s included:

1. Introduction to **portable extensions**
2. **Real** use cases from **real** projects
3. **Code** in IDE with **tests**

- What’s not included:

1. Introduction to CDI
2. Existing content on CDI extensions
3. Work with contexts (need 2 more hours)

Tools used in the code 1/2

Apache Deltaspike

1. Apache DeltaSpike is a great CDI toolbox
2. Provide helpers to develop extensions
3. And a collection of modules like:
   1. Security
   2. Data
   3. Scheduler
4. More info on deltaspike.apache.org

Tools used in the code 2/2

Arquillian

1. Arquillian is an integration testing platform
2. It integrates with JUnit
3. Create your SUT in a dedicated method
4. Run tests in the target containers of your choice
5. We’ll use the `arquillian-weld-embedded` container adapter
6. The proper solution to test Java EE code
7. More info on arquillian.org

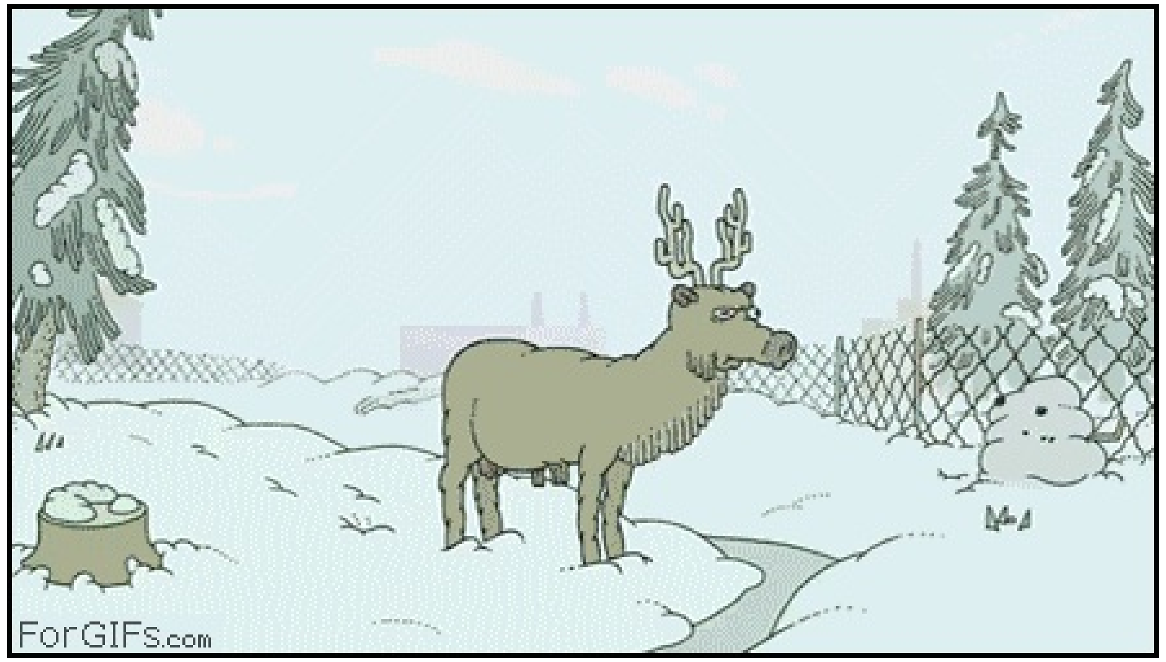
Agenda

💡 Slides available at astefanutti.github.io/further-cdi

- ℹ CDI Extensions
- ℹ Metrics CDI
- ℹ Camel CDI

CDI Extensions

Portable extensions



- One of the most powerful feature of the CDI specification
- Not really popularized, partly due to:

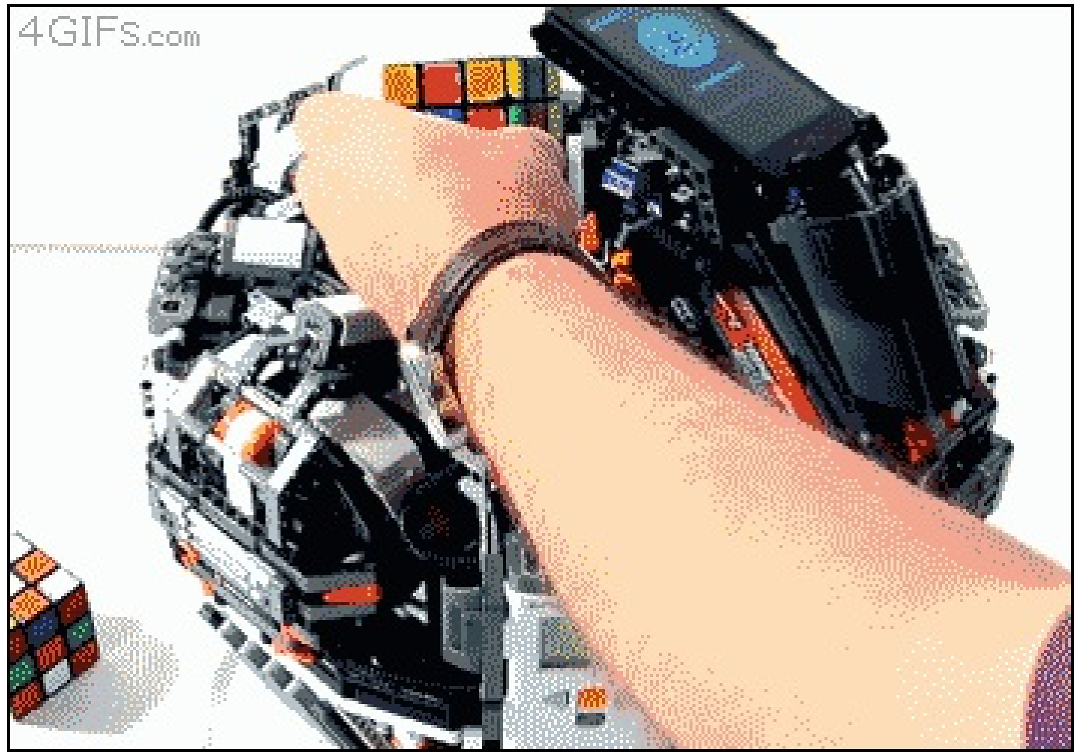
1. Their high level of abstraction
2. The pre-requisite knowledge about basic CDI and SPI
3. Lack of information (CDI is often perceived as a basic DI solution)

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



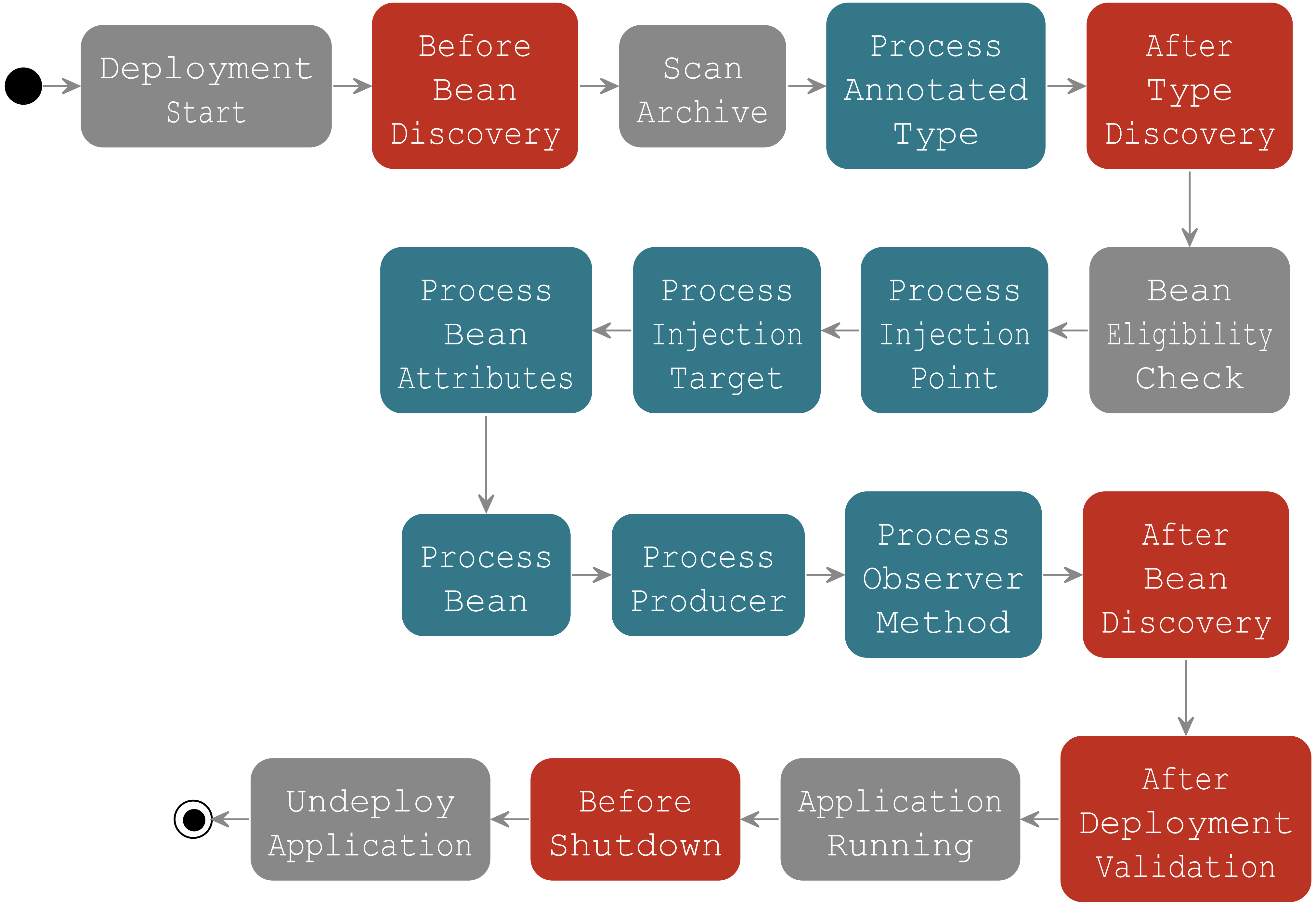
More concretely

ℹ Service provider of the service `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

This is a commonly admitted good practice

```
public class VetoEntity implements Extension {

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

Remember

⚠ 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

Metrics CDI

Dropwizard Metrics provides

- Different metric types: `Counter`, `Gauge`, `Meter`, `Timer`, …
- Different reporter: JMX, console, SLF4J, CSV, servlet, …
- `MetricRegistry` object which collects all your app metrics
- Annotations for AOP frameworks: `@Counted`, `@Timed`, …
- … but does not include integration with these frameworks
- More at 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"); ❶
        Timer.Context time = timer.time();
        try {
            /*...*/
        } finally {
            time.stop();
        }
    }
}
```

❶ 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
- Enable Metrics with the provided annotations
- Access same `Metric` instances through `@inject` or `MetricRegistry` API

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

What’s the problem with multiple Metrics of the same type?

⚠ This code throws a deployment exception (ambiguous dependency)

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

@Produces
Timer timer = new Timer(new SlidingTimeWindowReservoir(1L, HOURS)); ❷

@Inject
Timer timer; ❸
```

❶ This timer that only keeps measurement of last minute is produced as a bean of type `Timer`

❷ This timer that only keeps measurement of last hour is produced as a bean of type `Timer`

❸ 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 = "my_timer")
Timer timer = new Timer(new SlidingTimeWindowReservoir(1L, MINUTES));

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

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

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

How to declare `@Metric` as a qualifier?

- By observing `BeforeBeanDiscovery` lifecycle event in an extension

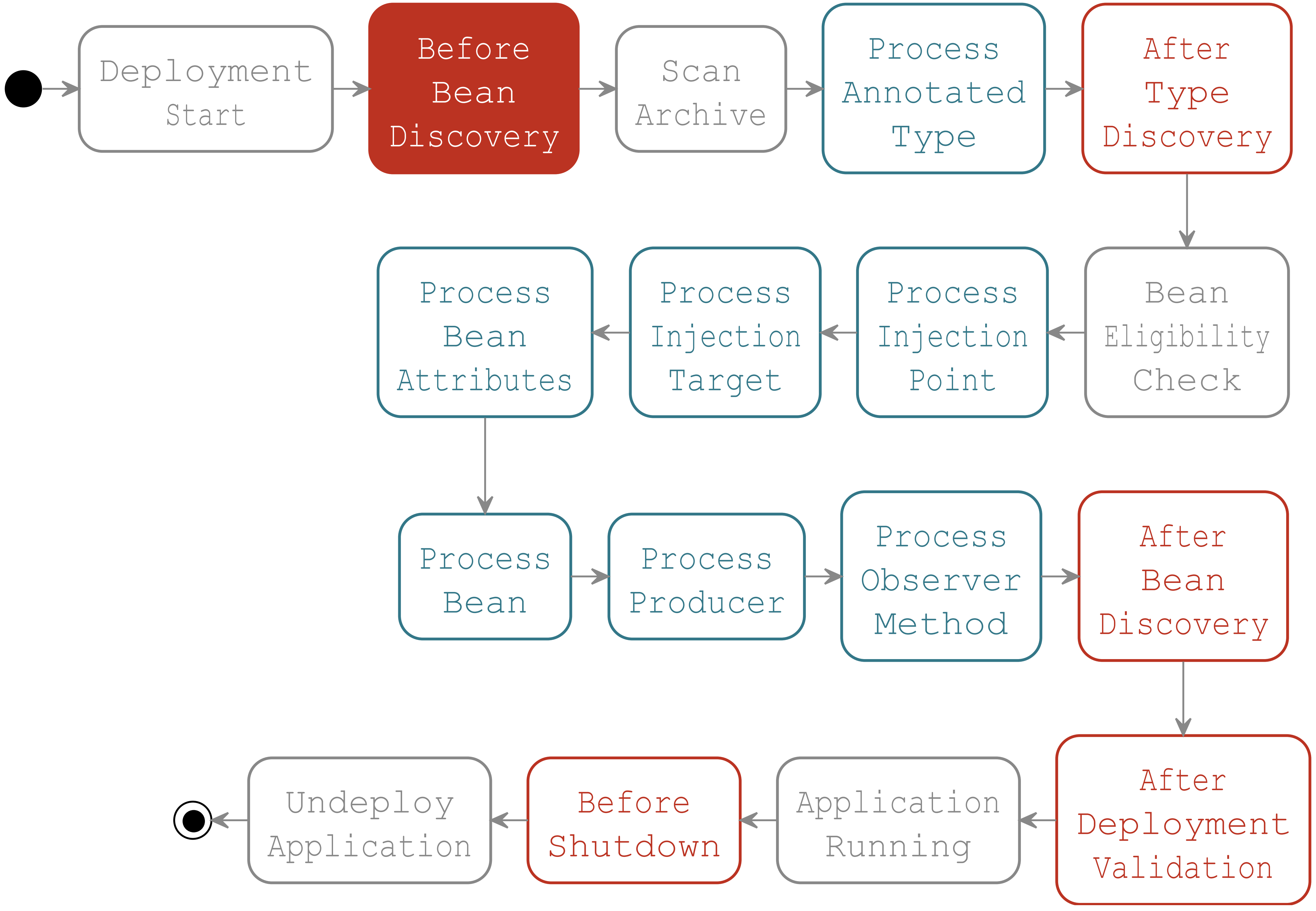
`javax.enterprise.inject.spi.BeforeBeanDiscovery`

```java
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 The method we need to declare the `@Metric` annotation as a CDI qualifier

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

`BeforeBeanDiscovery` is first in lifecycle



Deployment Start

Before Bean Discovery

Scan Archive

Process Annotated Type

After Type Discovery

Bean Eligibility Check

Process Injection Point

Process Injection Target

Process Bean Attributes

Process Bean

Process Producer

Process Observer Method

After Bean Discovery

After Deployment Validation

Application Running

Before Shutdown

Undeploy Application

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`

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

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

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

- And have the `Timer` injection points satisfied

GOAL 2 Apply Metrics with the provided annotations

Goal 2 in detail

- We want to be able to write:

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

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

- The solution is to declare an interceptor and bind it to `@Timed`

Goal 2 step by step

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

Preparing interceptor creation

- We should find the technical code that will wrap the business code

```java
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 code** of the application is called. All the code around is the **technical code**.

Activating the interceptor

```
@Interceptor
@Priority(Interceptor.Priority.LIBRARY_BEFORE)  ①
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` annotation 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 the `@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 achieve the required `@Timed` declaration?**

- **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>() {}; ①

    return new AnnotatedTypeBuilder().readFromType(Timed.class) ②
        .addToMethod(Timed.class.getMethod("name"), nonBinding) ③
        .addToMethod(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)
            .addToMethod(Timed.class.getMethod("name"), nonBinding)
            .addToMethod(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

1. Interceptor code should be enhanced to support `@Timed` on classes
2. Other interceptors should be developed for other metric types

Our goals

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

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

2. Register automatically produced custom metrics

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

① Annotations provided by Metrics

GOAL 3 Access same `Metric` instances through `@Inject` or `MetricRegistry` API

Goal 3 in detail

- When writing:

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

@Inject
MetricRegistry registry;
Timer timer2 = registry.timer("my_timer");
```

- … We want that `timer1 == timer2`

Goal 3 in detail

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

@Inject
@Metric(name = "my_timer")
Timer timer;

@Inject
MetricRegistry registry;
Timer timer = registry.timer("my_timer"); ❷
```

❶ Produced `Timer` should be added to the Metrics registry when produced

❷ When retrieved from the registry, a `Metric` should be identical to the produced instance and vice versa

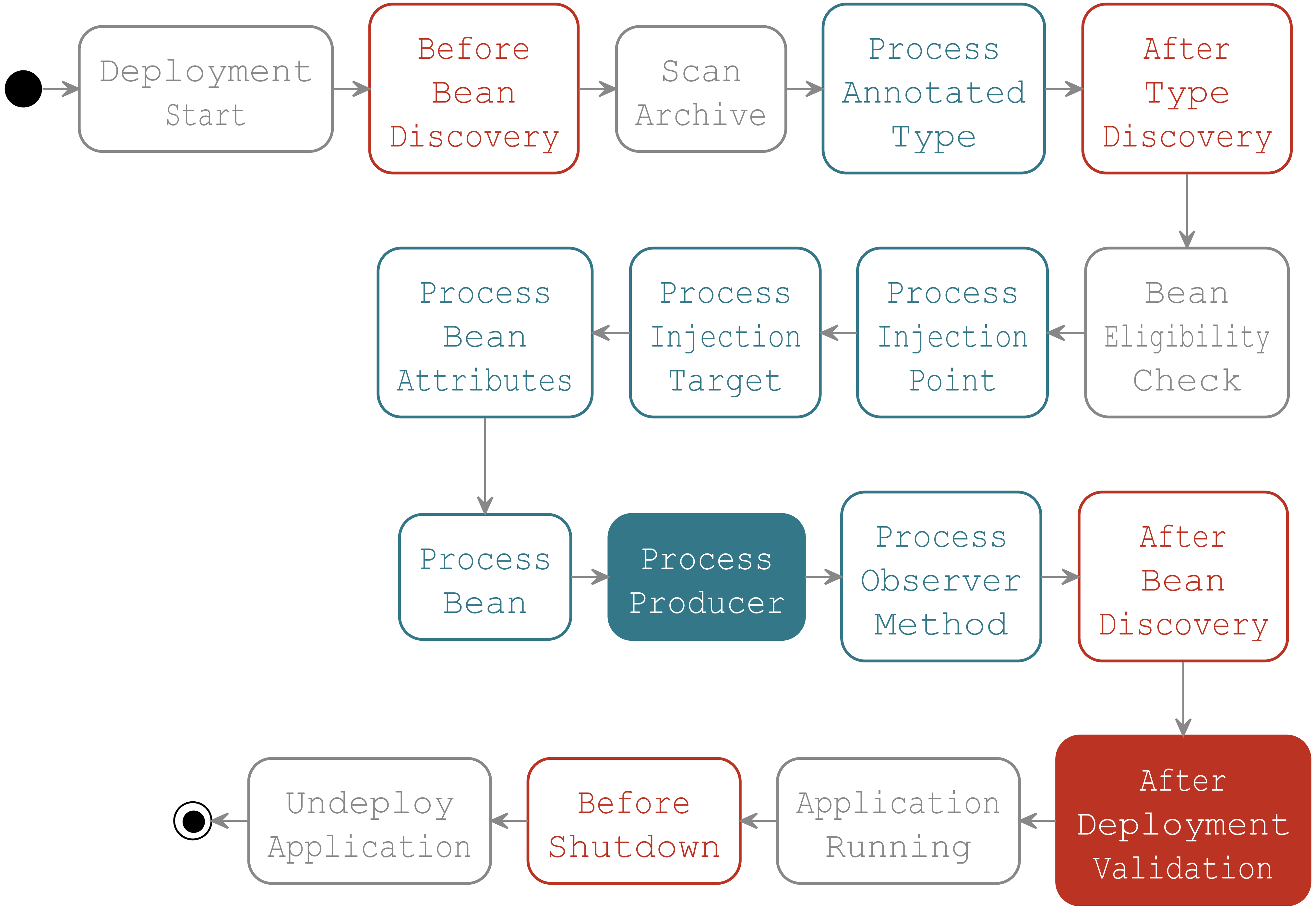
⚠ There are 2 `Metric` classes, the `com.codahale.metrics.Metric` interface and the `com.codahale.metrics.annotation.Metric` annotation

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 bootstrap 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



Deployment Start

Before Bean Discovery

Scan Archive

Process Annotated Type

After Type Discovery

Bean Eligibility Check

Process Injection Point

Process Injection Target

Process Bean Attributes

Process Bean

Process Producer

Process Observer Method

After Bean Discovery

After Deployment Validation

Application Running

Before Shutdown

Undeploy Application

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)

```java
    public X produce(CreationalContext<X> ctx) { ①
        MetricRegistry reg = 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 The `produce` method is used by the container at runtime to decorate declared producer with our logic

② `BeanProvider.getContextualReference` is helper class from **DeltaSpike** 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 the registry

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

ℹ This event allow us to substitute the original producer with ours

`javax.enterprise.inject.spi.ProcessProducer`

```
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(); ❶
        new pp.setProducer(new MetricProducer<>(pp.getProducer(), name)); ❷
    }
    // ...
}
```

❶ We retrieve metric’s name by calling the `name()` member from `@Metric`

❷ 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) {
        BeanProvider.getContextualReferences(com.codahale.metrics.Metric.class, true); ①
    }
    // ...
}
```

① `BeanProvider.getContextualReferences()` is a method from **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 = "my_timer")
Timer timer1 = new Timer(new SlidingTimeWindowReservoir(1L, MINUTES));

@Inject
@Metric(name = "my_timer")
Timer timer2;

@Inject
MetricRegistry registry;
Timer timer3 = registry.timer("my_timer");
```

- And make sure that `timer1 == timer2 == timer3`

Complete extension code

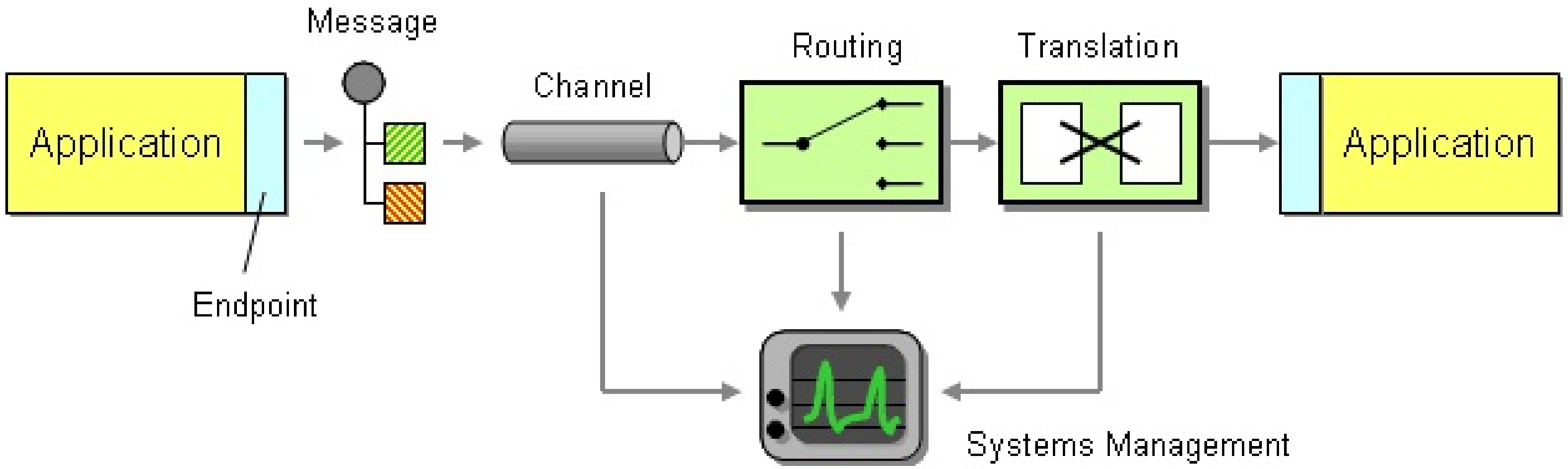
```
public class MetricsExtension implements Extension {

    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) {
        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



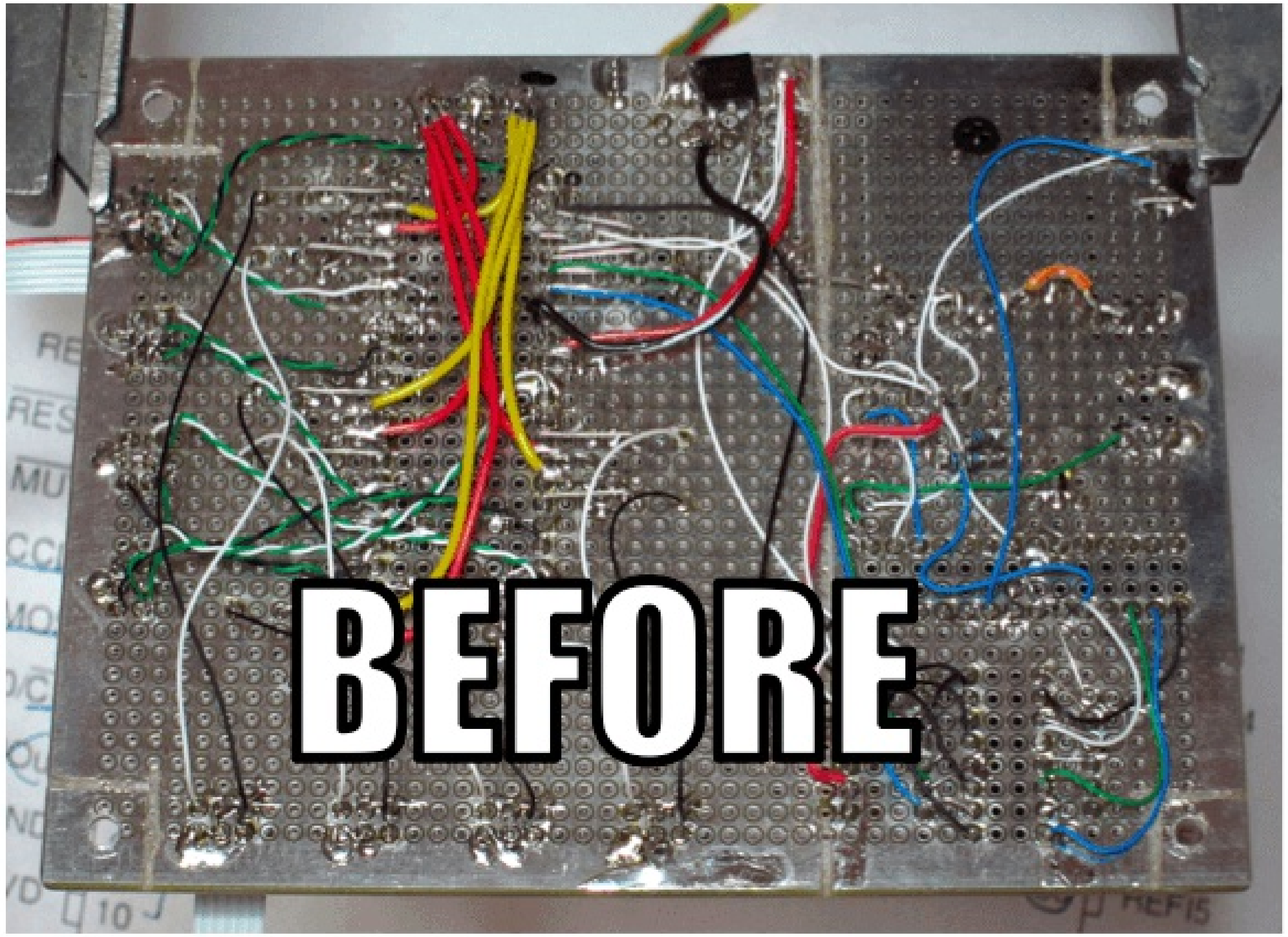
Discover how we created CDI integration module for Camel

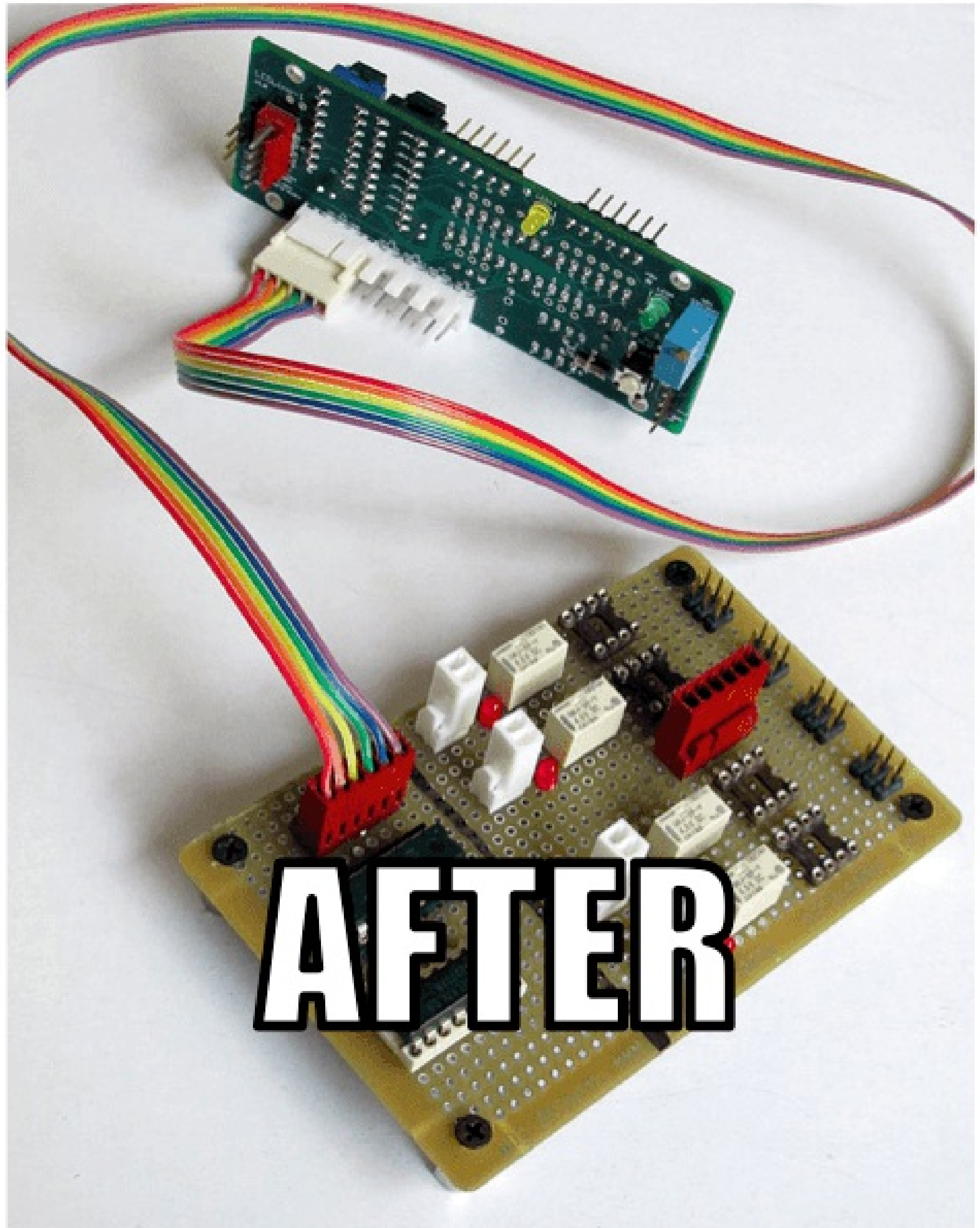
Camel out of the box (without CDI)

```
 public static void main(String[] args) {
    CamelContext context = new DefaultCamelContext();
    context.addRoutes(new RouteBuilder() {
      public void configure() {
        from("file:target/input?delay=1s")
          .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=1s")
            .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)

```java
@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

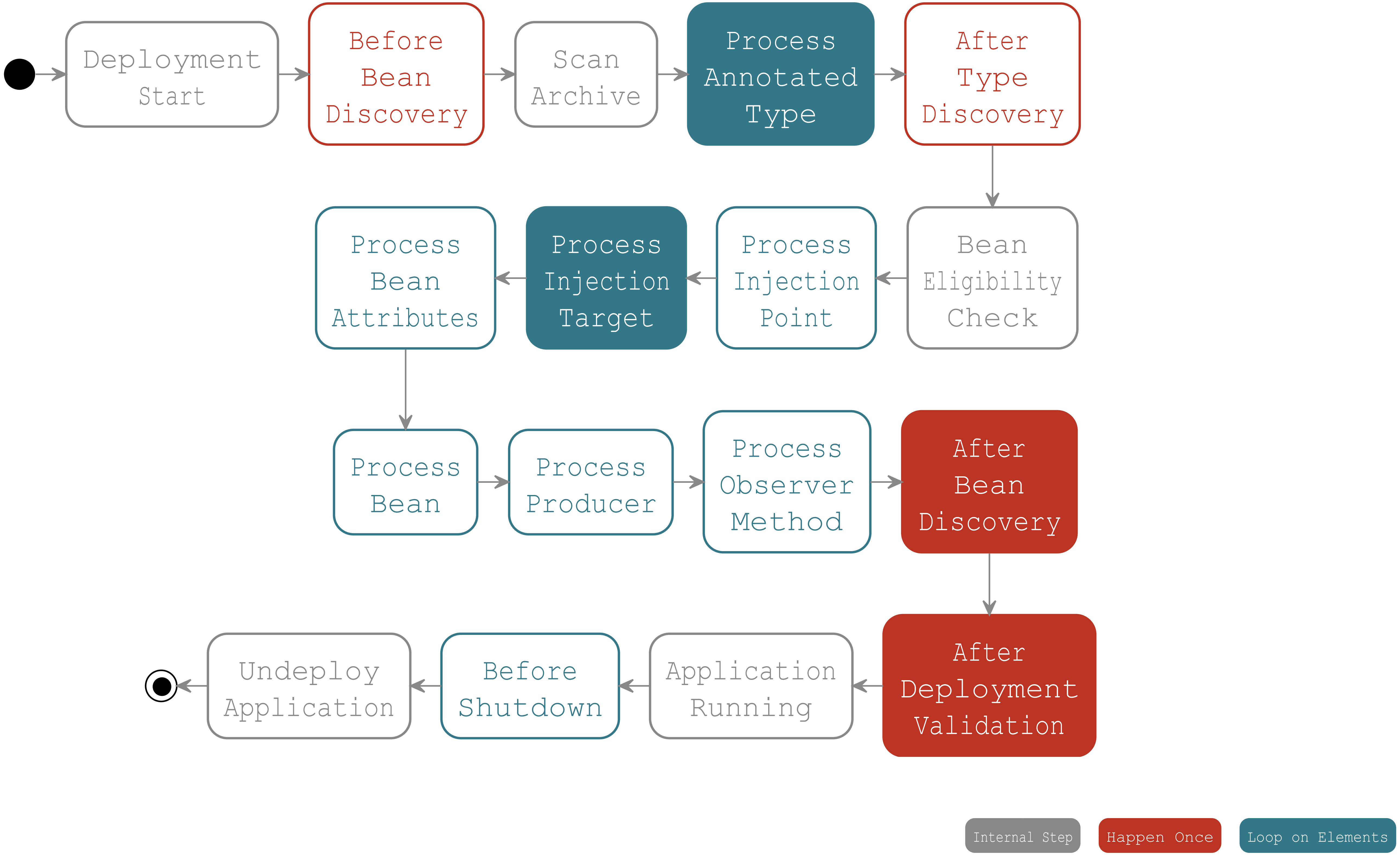
- Manage the creation and the configuration of the `CamelContext` bean
- Bind the `CamelContext` lifecycle to that of the CDI container
- Implement the Camel registry SPI to look up CDI bean references
- Use a custom `InjectionTarget` for CDI beans containing Camel annotations

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 bean `destroy` method is called)
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 4 events to add our features


Deployment Start

Before Bean Discovery

Scan Archive

Process Annotated Type

After Type Discovery

Bean Eligibility Check

Process Injection Point

Process Injection Target

Process Bean Attributes

Process Bean

Process Producer

Process Observer Method

After Bean Discovery

After Deployment Validation

Application Running

Before Shutdown

Undeploy Application

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

Use the `BeanConfigurator<T>` API introduced in **CDI 2.0**

`javax.enterprise.inject.spi.builder.BeanConfigurator<T>`

```java
public interface BeanConfigurator<T> {
    BeanConfigurator<T> beanClass(Class<?> beanClass);
    <U extends T> BeanConfigurator<U> createWith(Function<CreationalContext<U>, U> callback);
    <U extends T> BeanConfigurator<U> produceWith(Supplier<U> callback);
    BeanConfigurator<T> destroyWith(BiConsumer<T, CreationalContext<T>> callback);
    BeanConfigurator<T> disposeWith(Consumer<T> callback);
    <U extends T> BeanConfigurator<U> read(AnnotatedType<U> type);
    BeanConfigurator<T> read(BeanAttributes<?> beanAttributes);
    BeanConfigurator<T> addType(Type type);
    BeanConfigurator<T> scope(Class<? extends Annotation> scope);
    BeanConfigurator<T> addQualifier(Annotation qualifier);
    BeanConfigurator<T> name(String name);
    // ...
}
```

Adding a programmatic bean to the deployment

- Access the `BeanConfigurator<T>` API by observing the `AfterBeanDiscovery` lifecycle event

```
public class CamelExtension implements Extension {

    void addCamelContextBean(@Observes AfterBeanDiscovery abd) {
        abd.addBean()
            .types(CamelContext.class)
            .scope(ApplicationScoped.class)
            .produceWith(() -> new DefaultCamelContext());
    }
}
```

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 manager) {
        CamelContext context = getReference(manager, CamelContext.class);
        for (Bean<?> bean : manager.getBeans(RoutesBuilder.class))
            context.addRoutes(getReference(manager, RouteBuilder.class, bean));
    }

    <T> T getReference(BeanManager manager, Class<T> type) {
        return getReference(manager, type, manager.resolve(manager.getBeans(type)));
    }

    <T> T getReference(BeanManager manager, Class<T> type, Bean<?> bean) {
        return (T) manager.getReference(bean, type, manager.createCreationalContext(bean));
    }
}
```

Managed the Camel context lifecycle (start)

- Start the context when the `AfterDeploymentValidation` event is fired

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

Managed the Camel context lifecycle (stop)

- Stop the context when the associated bean is destroyed

```
public class CamelExtension implements Extension {
    // ...
    void addCamelContextBean(@Observes AfterBeanDiscovery abd) {
        abd.addBean()
            .types(CamelContext.class)
            .scope(ApplicationScoped.class)
            .produceWith(() -> new DefaultCamelContext())
            .disposeWith(context -> 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();~~
    ~~}~~
}
```

❓ 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

```java
class CamelCdiRegistry implements Registry {

    private final BeanManager manager;

    CamelCdiRegistry(BeanManager manager) {
        this.manager = manager;
    }

    public Object lookupByName(String name) {
        return lookupByNameAndType(name, Object.class);
    }

    @Override
    public <T> T lookupByNameAndType(String name, Class<T> type) {
        return Optional.of(manager.getBeans(name))
            .map(manager::resolve)
            .map(bean -> manager.getReference(bean, type, manager.createCreationalContext(bean)))
            .map(type::cast)
            .orElse(null);
    }

    // ...
}
```

Add `CamelCdiRegistry` to the Camel context

```
public class CamelExtension implements Extension {

    void addCamelContextBean(@Observes AfterBeanDiscovery abd, BeanManager manager) {
        abd.addBean()
            .types(CamelContext.class)
            .scope(ApplicationScoped.class)
            .produceWith(() -> new DefaultCamelContext(new CamelCdiRegistry(manager)))
            .disposeWith(context -> context.stop());
    }
}
```

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

```java
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 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

```java
@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`

`javax.enterprise.inject.spi.InjectionTarget`

```java
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`

```java
class CamelInjectionTarget<T> implements InjectionTarget<T> {

    final InjectionTarget<T> delegate;

    final DefaultCamelBeanPostProcessor processor;

    CamelInjectionTarget(InjectionTarget<T> target, final BeanManager manager) {
        delegate = target;
        processor = new DefaultCamelBeanPostProcessor() {
            public CamelContext getOrLookupCamelContext() {
                return getReference(manager, 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` event and set the `InjectionTarget`

`javax.enterprise.inject.spi.ProcessInjectionTarget`

```java
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`

```java
public class CamelExtension implements Extension {

    <T> void camelBeansPostProcessor(@Observes ProcessInjectionTarget<T> pit, BeanManager manager) {
        pit.setInjectionTarget(new CamelInjectionTarget<>(pit.getInjectionTarget(), manager));
    }
}
```

But only for beans containing Camel annotations

```
public 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 manager) {
      if (camelBeans.contains(pit.getAnnotatedType())) ②
        pit.setInjectionTarget(new CamelInjectionTarget<>(pit.getInjectionTarget(), manager));
    }
}
```

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=1s")
    ~~.log("Sending message [${body}] to JMS...")~~
    .to("sjms:queue:output");
```

- With CDI observers

```
from("file:target/input?delay=1s").to("sjms:queue:output").id("join point");
```

```
void advice(@Observes @NodeId("join point") Exchange exchange) {
    logger.info("Sending message [{}] to JMS...", exchange.getIn().getBody(String.class));
}
```

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

`javax.enterprise.inject.spi.ProcessObserverMethod`

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

And collect the observer method metadata

```java
public class CamelExtension implements Extension {

    final Set<NodeId> joinPoints = new HashSet<>();

    void pointcuts(@Observes ProcessObserverMethod<Exchange, ?> pom) {
        pom.getObserverMethod().getObservedQualifiers().stream()
            .filter(q -> q instanceof NodeId)
            .map(NodeId.class::cast)
            .forEach(joinPoints::add);
    }
}
```

Instrument the Camel context

- Intercept matching nodes and fire a CDI event

```java
void configureCamelContext(@Observes AfterDeploymentValidation adv, final BeanManager manager) {
    context.addInterceptStrategy(new InterceptStrategy() {
        public Processor wrapProcessorInInterceptors(CamelContext context, ProcessorDefinition<?> definition,
                                                     Processor target, Processor nextTarget) {
            if (definition.hasCustomIdAssigned()) {
                for (final NodeId 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=1s").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 [{}] to [{}]...",
                exchange.getIn().getBody(String.class),
                history.get(history.size() - 1).getNode().getLabel());
}
```

Conclusion

References

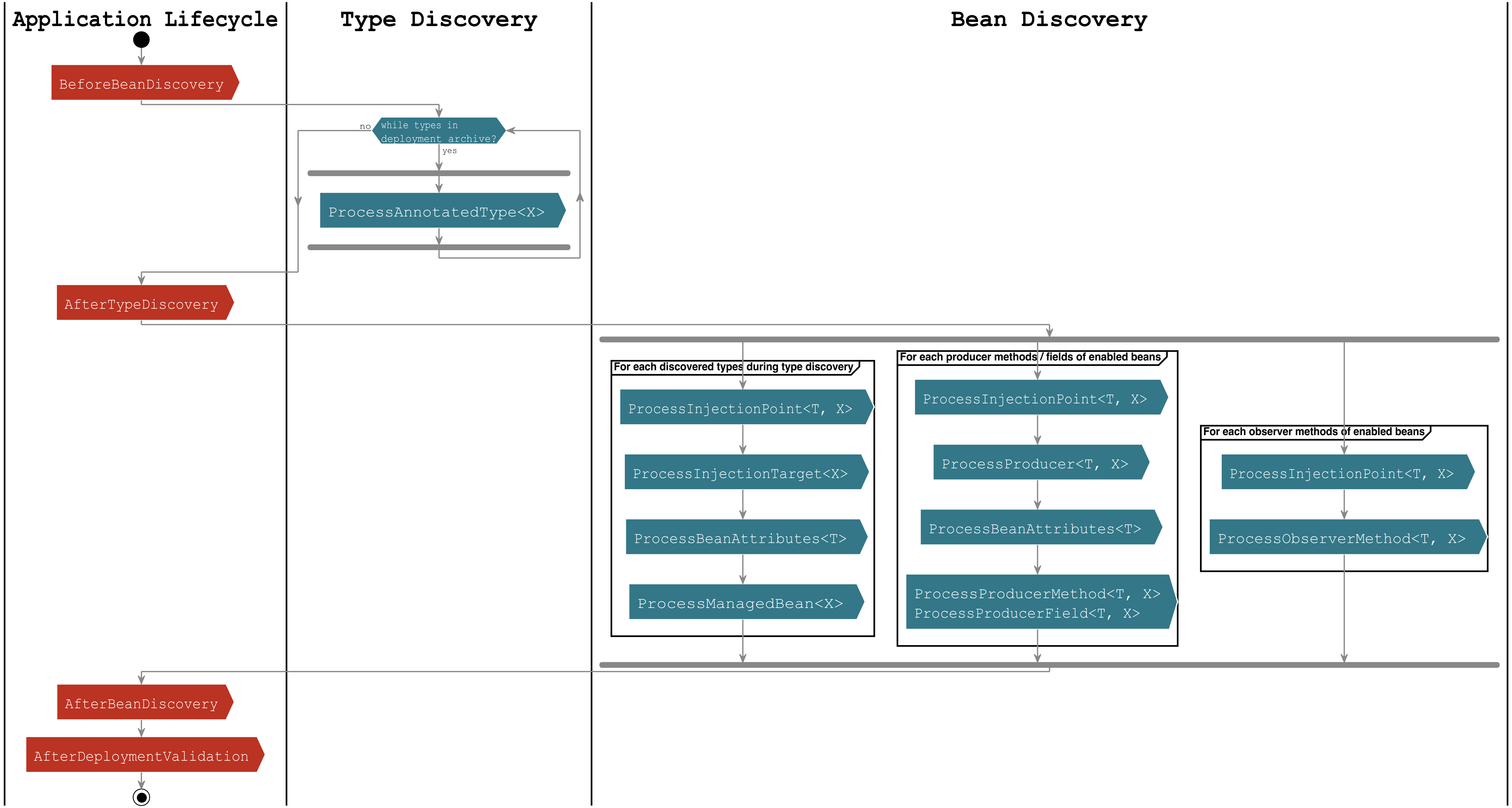
- CDI Specification - cdi-spec.org
- Slides sources - github.com/astefanutti/further-cdi
- Metrics CDI sources - github.com/astefanutti/metrics-cdi
- Camel CDI sources - github.com/astefanutti/camel-cdi
- Slides generated with Asciidoctor, PlantUML and DZSlides backend
- Original slide template - Dan Allen & Sarah White

Antoine Sabot-Durand
Antonin Stefanutti

@antoine_sd @astefanut

Annexes

Complete lifecycle events



Application Lifecycle

Type Discovery

Bean Discovery

BeforeBeanDiscovery

no

while types in deployment archive?

yes

ProcessAnnotatedType<X>

AfterTypeDiscovery

For each discovered types during type discovery

ProcessInjectionPoint<T, X>

ProcessInjectionTarget<X>

ProcessBeanAttributes<T>

ProcessManagedBean<X>

For each producer methods / fields of enabled beans

ProcessInjectionPoint<T, X>

ProcessProducer<T, X>

ProcessBeanAttributes<T>

ProcessProducerMethod<T, X>
ProcessProducerField<T, X>

For each observer methods of enabled beans

ProcessInjectionPoint<T, X>

ProcessObserverMethod<T, X>

AfterBeanDiscovery

AfterDeploymentValidation