Going further with CDI 1.2

Antoine Sabot-Durand · Antonin Stefanutti

Antonin Stefanutti


Software Engineer


Murex

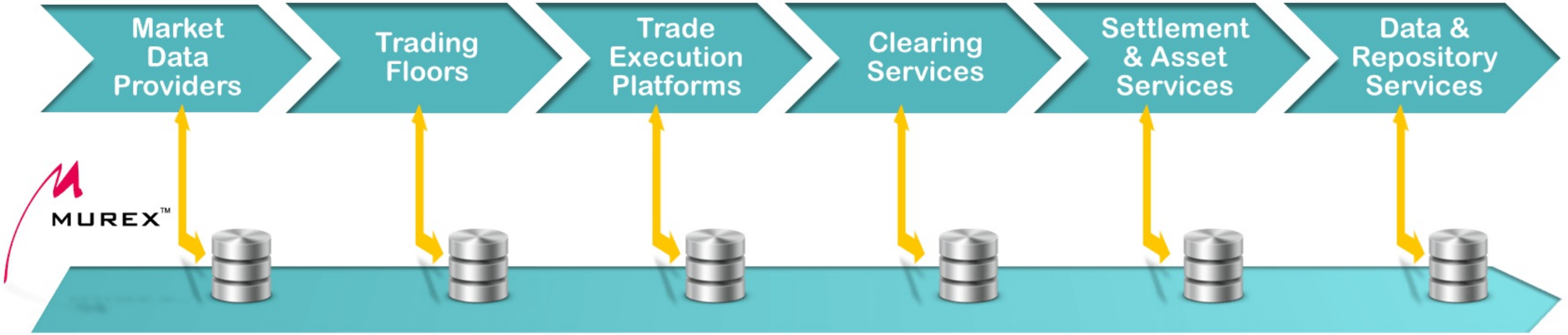

www.murex.com


@astefanut


github.com/astefanutti

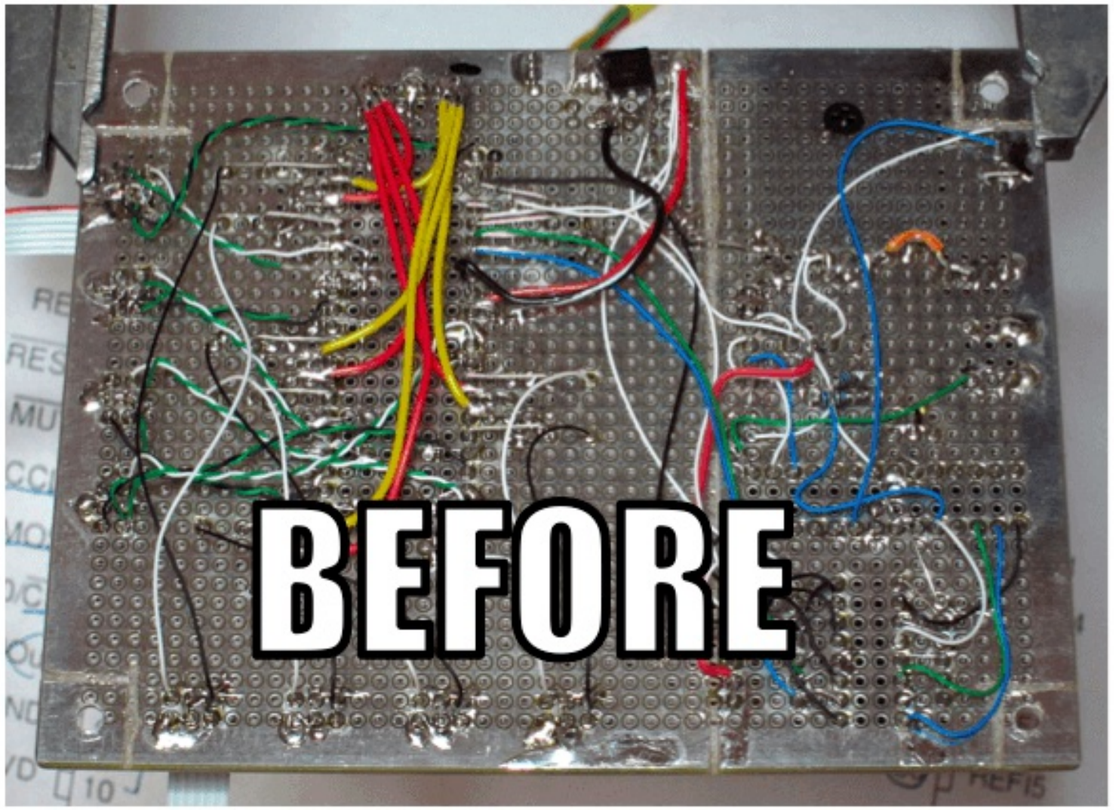

Slides available at github.com/astefanutti/further-cdi

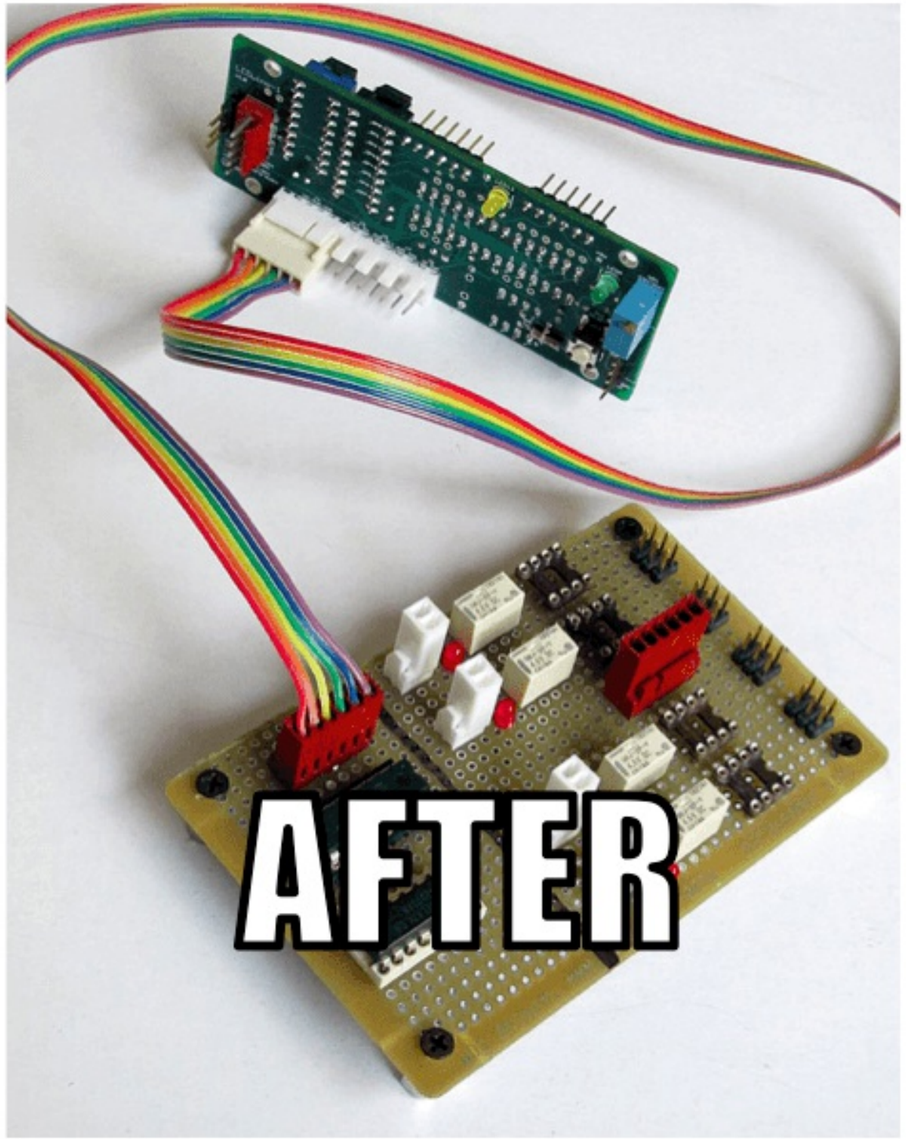
CDI @ Murex



CDI as the **productivity ecosystem** to build **connectivity interfaces**

CDI @ Murex





Antoine Sabot-Durand


Senior Software Engineer


CDI co-spec lead, Java EE 8 EG


Red Hat, Inc.


@antoine_sd


www.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 the most of these you can stay

`@Inject`

`@Produces`

`Event<T>`

`@Observes`

`@Qualifier`

`InjectionPoint`

More concretely

- 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

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

Agenda

- Meet CDI SPI
- Introducing CDI Extensions
- Metrics CDI
- CDI Quizz
- Camel CDI

Slides available at astefanutti.github.io/further-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

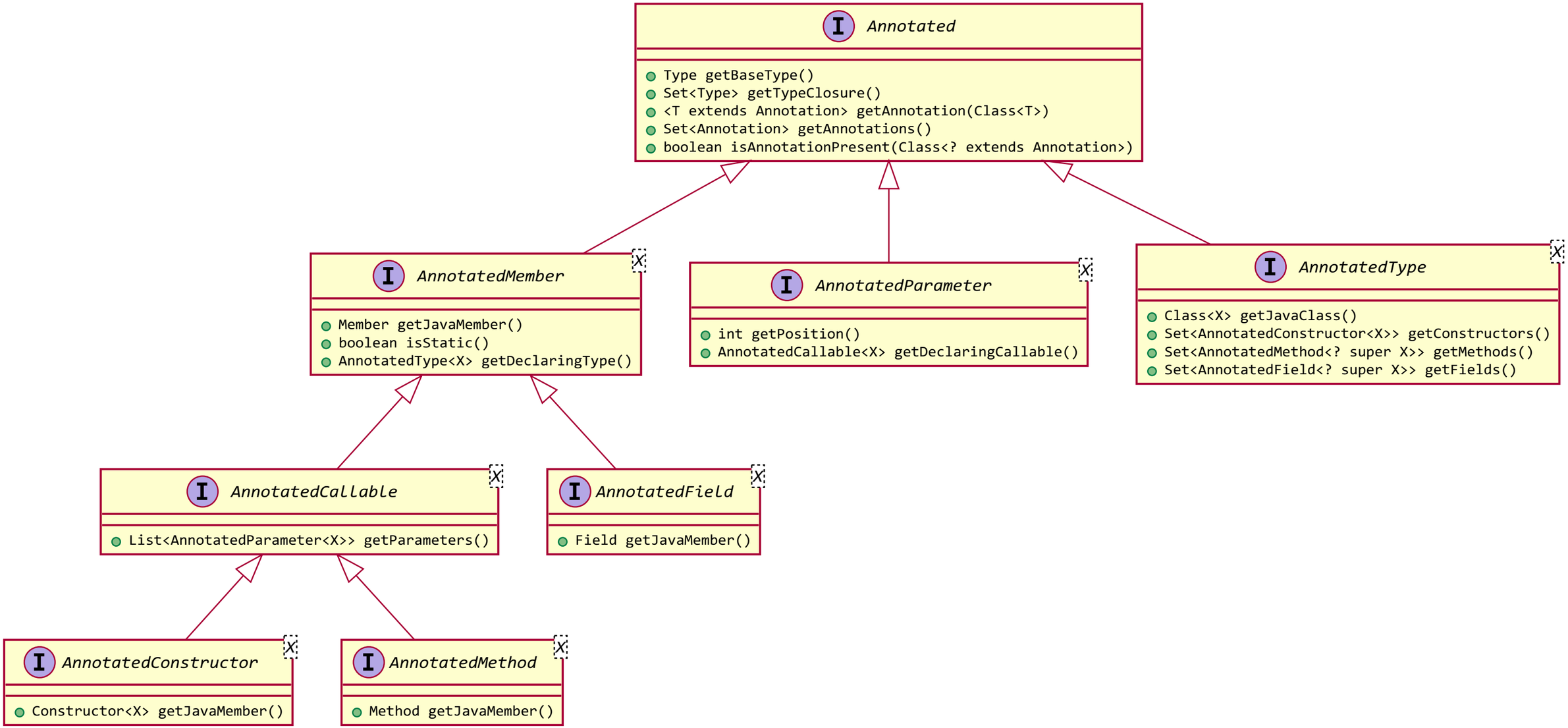
SPI can be split in 4 parts

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

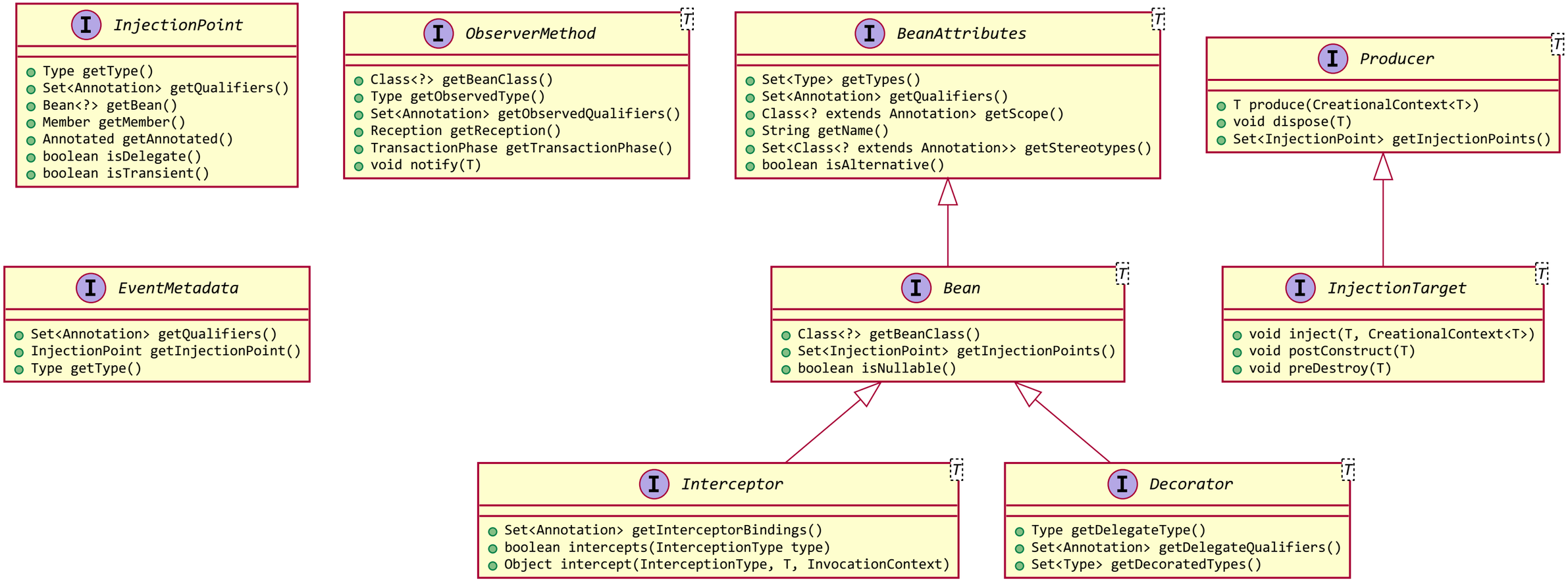
Why having a type meta-model?

- 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());
}
```

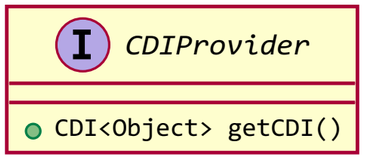
```
@Inject
@HttpParam("productId")
String productId;
```

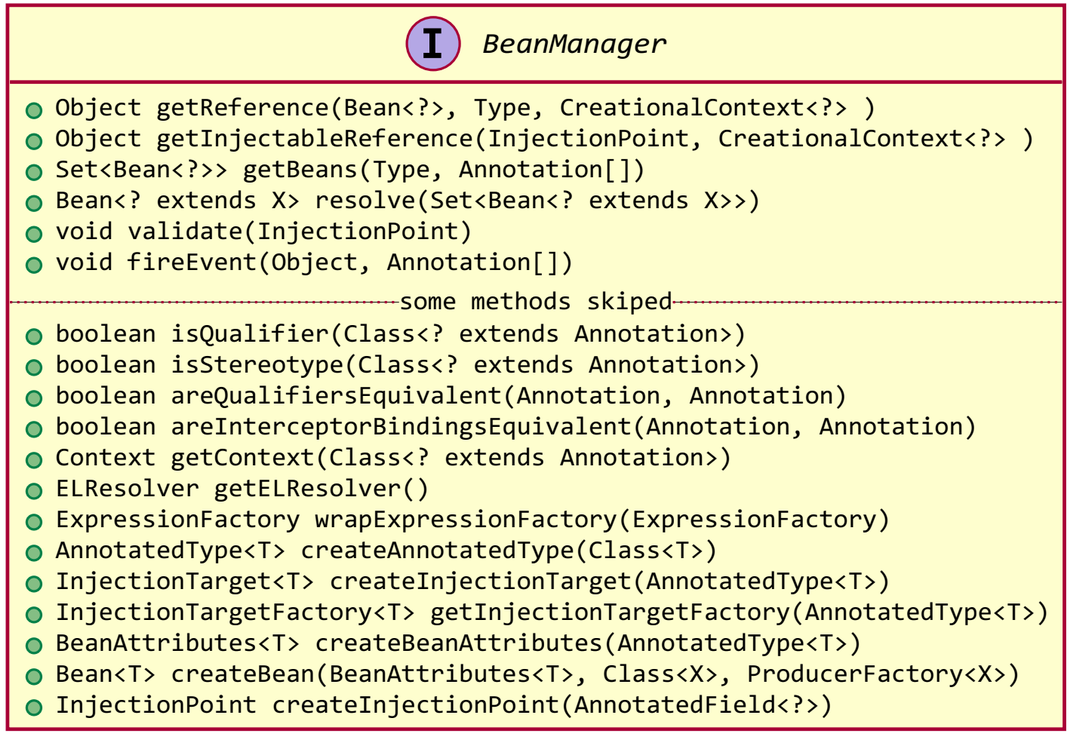
This SPI can be used in your code (2/2)

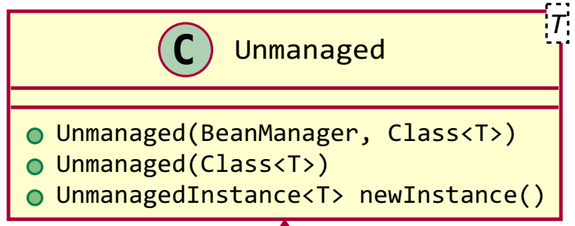
`InjectionPoint` contains info about requested type at `@Inject`

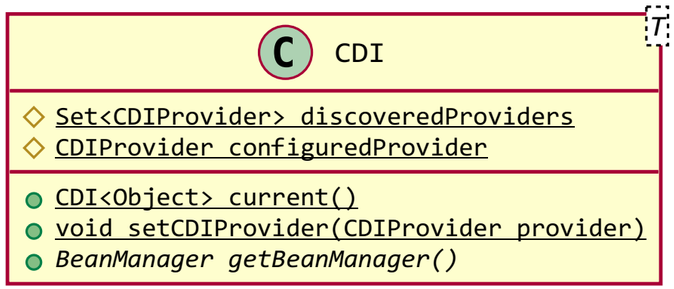
```
class MyMapProducer() {

    @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)
    }
}
```

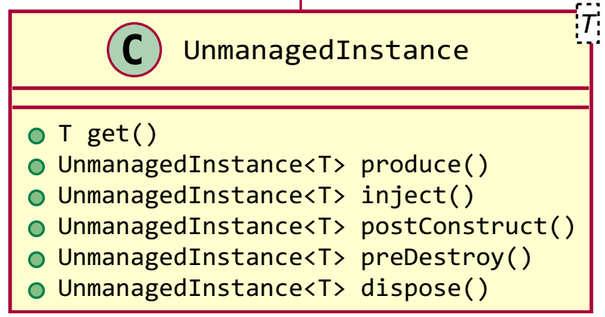
SPI providing CDI entry points



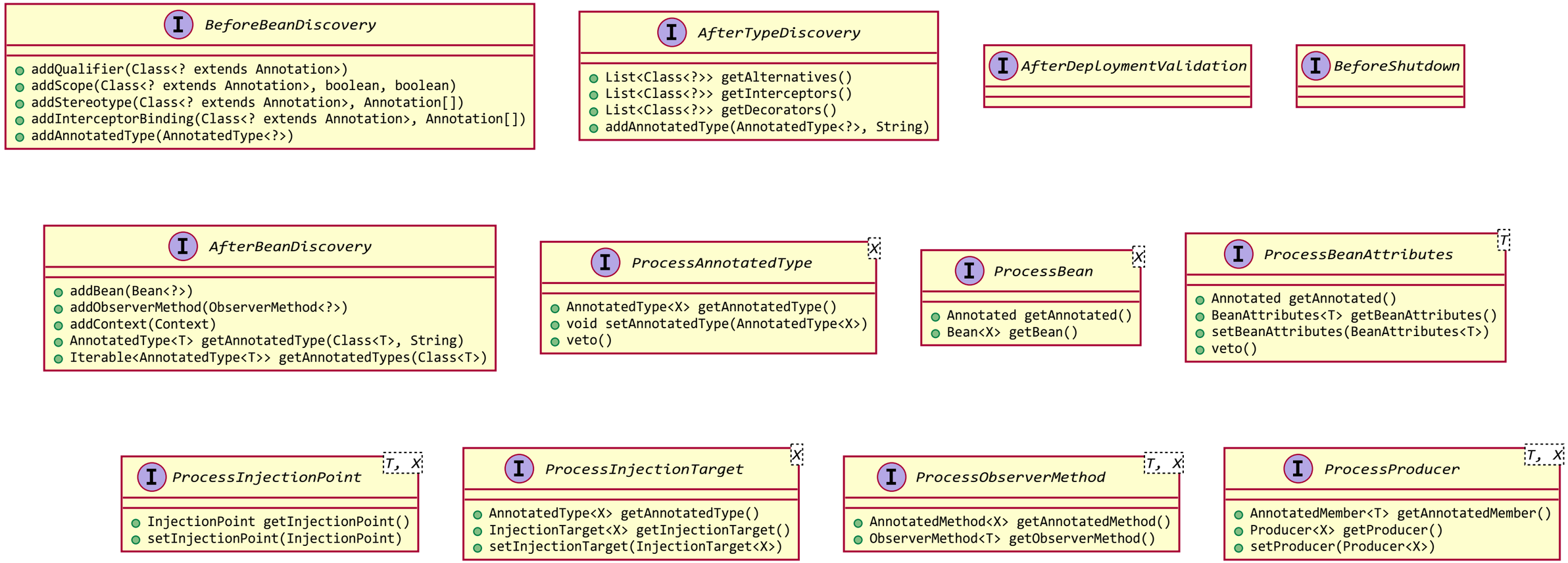








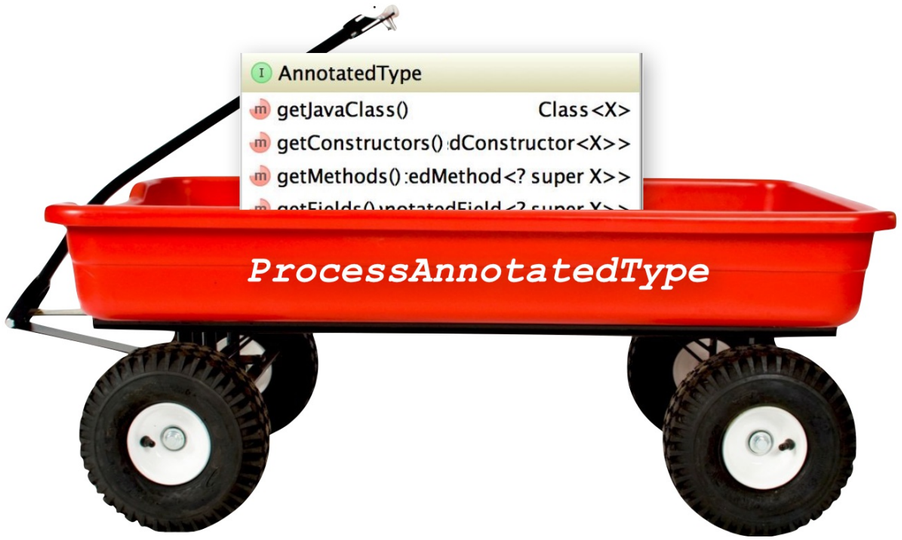
SPI dedicated to extensions



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

- One of the most powerful feature of the CDI specification

- 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

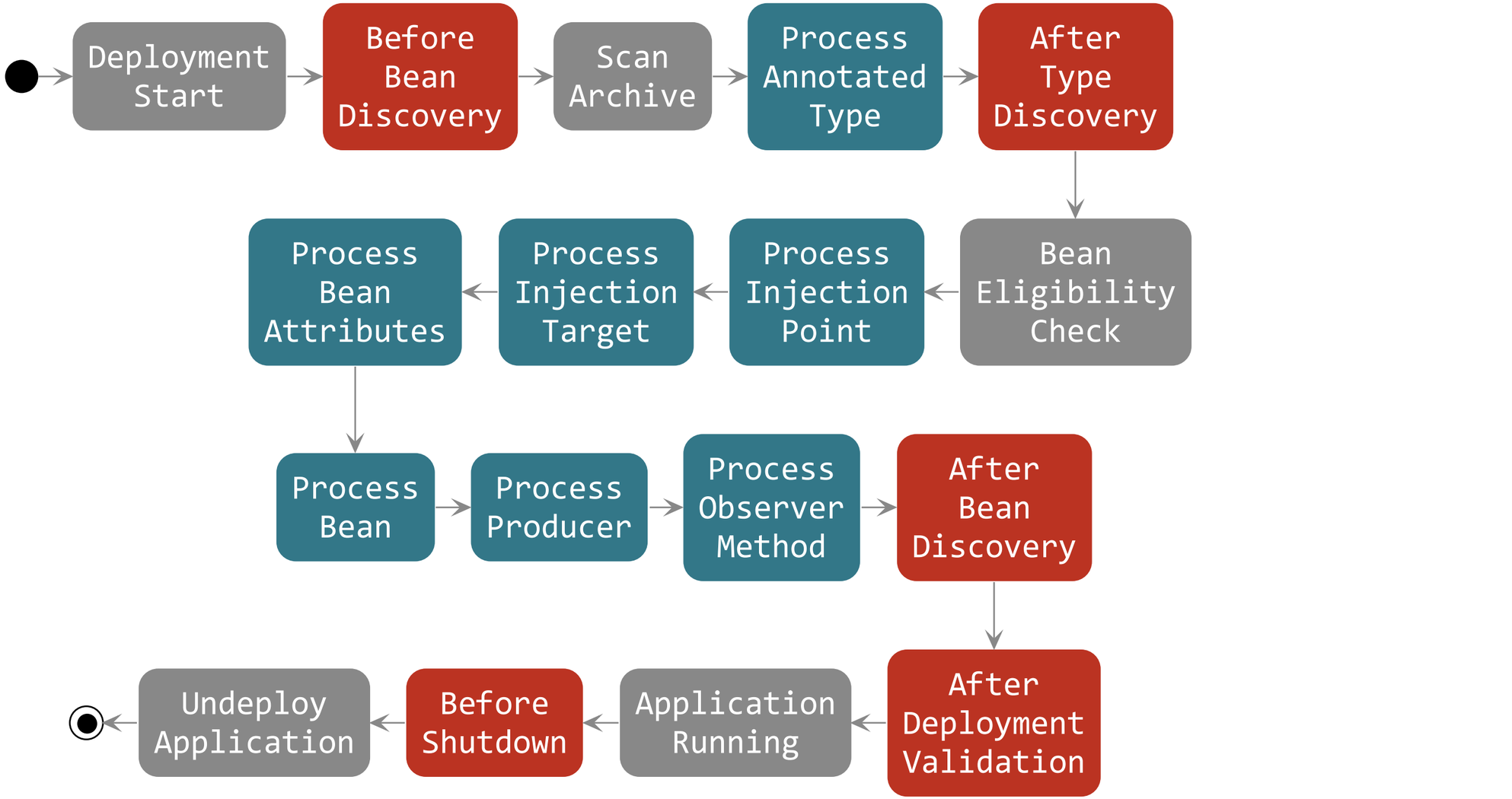
More concretely

Service provider of the service `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



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();
    }
}
```

⚠ 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

How to integrate a 3rd party Library (Dropwizard Metrics) into the CDI Programming Model

3rd party Library

About Dropwizard Metrics

- Provides different metric types: `Counter`, `Gauge`, `Meter`, `Timer`, …
- Provides different reporter: JMX, console, SLF4J, CSV, servlet, …
- Provides a `MetricRegistry` which collects all your app metrics
- Provides 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"); (1)
        Timer.Context time = timer.time();
        try {
            /*...*/
        } finally {
            time.stop();
        }
    }
}
```

(1) Note that if `Timer` called `"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

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

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

2. Register automatically produced custom metrics

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

❶ Annotations provided by Metrics

Steps to apply a timer in AOP style

- Create an interceptor for the timer technical code
- Make the Metrics annotation `@Timed` a valid interceptor binding annotation
- Programmatically add `@Timed` as an interceptor binding
- Use the magic

Preparing interceptor creation

To create an interceptor we should start by detecting 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 is an independent specification (JSR 318).
Highlighted code below is part of it.

```
@Interceptor
class TimedInterceptor {
    @Inject MetricRegistry registry; (1)
    @AroundInvoke
    Object timeMethod(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
@Priority(Interceptor.Priority.LIBRARY_BEFORE)
class TimedInterceptor {

    @Inject
    MetricRegistry registry;

    @AroundInvoke
    Object timeMethod(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

1 Giving a `@Priority` to an interceptor activates it. This annotation is part of the Common Annotations specification (JSR 250). In CDI, interceptor activation can also be done in the `beans.xml` file.

Add a binding to the interceptor

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

    @Inject
    MetricRegistry registry;

    @AroundInvoke
    Object timeMethod(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 kind of places:

1. On the interceptor definitions to associate them to this annotation
2. On the methods / classes to be intercepted by this interceptor

- An interceptor binding should be annotated with the `@InterceptorBinding` meta annotation or should be declared as an interceptor binding programmatically

- If the interceptor binding annotation has members:

1. Their values are taken into account to distinguish two instances
2. Unless members are annotated with `@NonBinding`

`@Timed` source code tells us it's 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 and we have no code to add it programmatically

(2) None of the members have the `@NonBinding` annotation so they'll be used to distinguish two instances (i.e. `@Timed(name = "timer1")` and `@Timed(name = "timer2")` will be 2 different interceptor bindings)

The needed `@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!

Remember 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

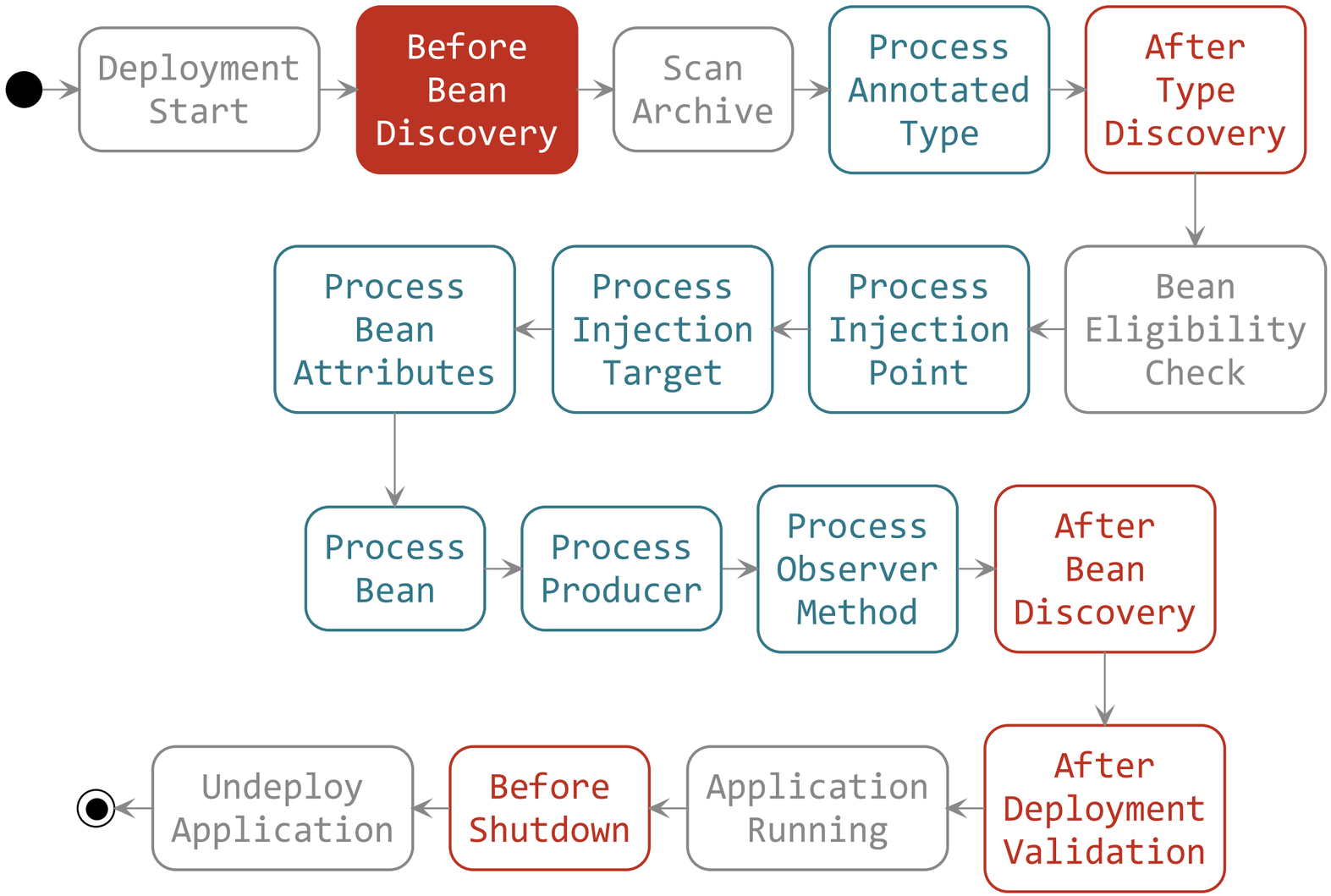
Add `@Timed` to the list of interceptor binding with an extension

By observing `BeforeBeanDiscovery` lifecycle event

```
public interface BeforeBeanDiscovery {

    addQualifier(Class<? extends Annotation> qualifier);
    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);   (1)
    addInterceptorBinding(Class<? extends Annotation> bindingType, Annotation... bindingTypeDef);
    addAnnotatedType(AnnotatedType<?> type);
    addAnnotatedType(AnnotatedType<?> type, String id);
}
```

(1) This method is the one we need to add our modified `@Timed` `AnnotatedType`

`BeforeBeanDiscovery` is first in lifecycle



Internal Step | Happen Once | Loop on Elements

This extension will do the job

```
class MetricsExtension implements Extension {

    void addTimedBinding(@Observes BeforeBeanDiscovery bbd) throws Exception {
        bbd.addInterceptorBinding(createTimedAnnotatedType());
    }

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

First goal achieved

We can now write:

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

And have a Metrics `Timer` applied to the method

❓ Why would we want custom metrics?

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

(1) The registry provides a default `Timer` (if none was registered by the user). The default timer histogram is exponentially biased to the past 5 minutes of measurements. We may want to have an other behavior.

Automatically register custom metrics

We want to write this:

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

And have:

1. The possibility to retrieve this `Timer` from the registry when it's injected (instead of having a new instance created)
2. This `Timer` produced when needed (first use)
3. This `Timer` registered in the registry with its name (here `"myTimer"`)

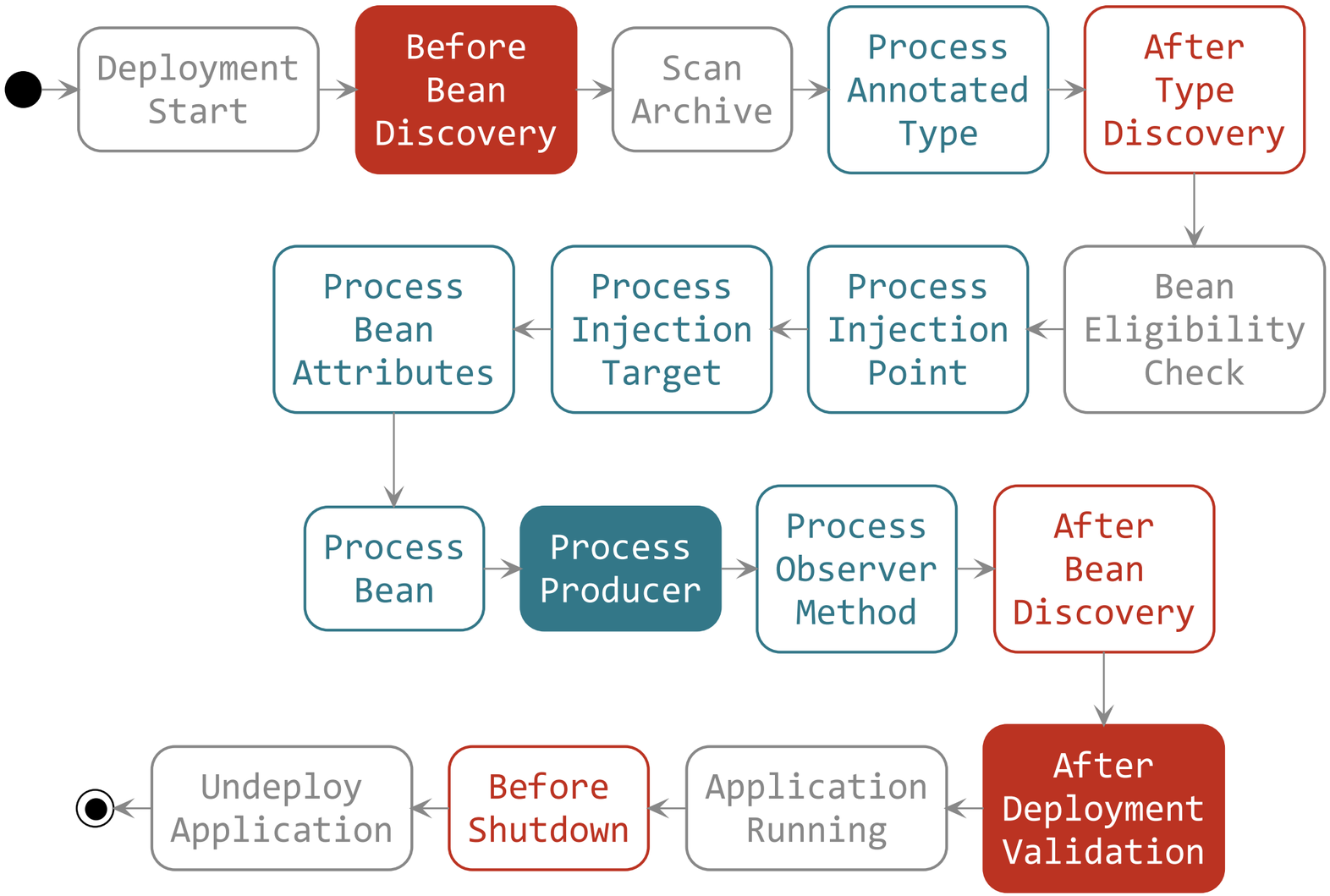
There are 2 `Metric`: the `com.codahale.metrics.Metric` interface and the `com.codahale.metrics.annotation.Metric` annotation

How to achieve this?

We need to write an extension that will:

1. Declare `@Metric` as a qualifier to ease injection and name resolution in a `BeforeBeanDiscovery` observer
2. Change how a `Metric` instance will be produced to look it up in the registry and produce (and register) 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
3. 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 3 events to add our features



Internal Step | Happen Once | Loop on Elements

Adding `@Metric` to the list of qualifiers

- This time we need annotation members to be "binding" (`@Metric("a")` and `@Metric("b")` should be distinguished)
- So we don’t have to add `@Nonbinding` annotation to them

```
public class MetricExtension implements Extension {

    void addMetricQualifier(@Observes BeforeBeanDiscovery bbd) {
        bbd.addQualifier(Metric.class);
    }
    //...
}
```

Customizing `Metric` producing process

We first need to create our implementation of the `Producer<X>` SPI

```
class MetricProducer<X extends Metric> implements Producer<X> {

    private final Producer<X> delegate;

    private final BeanManager bm;

    private final String name;

    MetricProducer(Producer<X> delegate, BeanManager bm, String name) {
        this.decorated = decorated;
        this.bm = bm;
        this.name = name;
    }
    //...
```

Customizing `Metric` producing process (continued)

```
    //...
    @Override
    public X produce(CreationalContext<X> ctx) {
        MetricRegistry registry = BeanProvider.getContextualReference(bm, MetricRegistry.class, false); (1)
        if (!registry.getMetrics().containsKey(name))
            registry.register(name, delegate.produce(ctx));
        return (X) registry.getMetrics().get(name);
    }

    @Override
    public void dispose(X instance) {
    } (2)

    @Override
    public Set<InjectionPoint> getInjectionPoints() {
        return decorated.getInjectionPoints();
    }
}
```

(1) `BeanProvider` is a DeltaSpike helper class to easily retrieve a bean or bean instance

(2) We don't want to have the produced `Metric` instance destroyed by the CDI container

We'll use our `Producer<Metric>` in a `ProcessProducer` observer

Through this event we can substitute the standard 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 MetricExtension implements Extension {
    //...
    <X extends com.codahale.metrics.Metric> void decorateMetricProducer(
        @Observes ProcessProducer<?, X> pp, BeanManager bm) {
        String name = pp.getAnnotatedMember().getAnnotation(Metric.class).name();
        pp.setProducer(new MetricProducer<>(pp.getProducer(), bm, name));
    }
    //...
}
```

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

We do that by observing the `AfterDeploymentValidation` event

```
public class MetricExtension implements Extension {
    //...
    void registerProduceMetrics(@Observes AfterDeploymentValidation adv, BeanManager bm) {
      for (Bean<?> bean : bm.getBeans(com.codahale.metrics.Metric.class, new AnyLiteral())) 1
        for (Annotation qualifier : bean.getQualifiers())
          if (qualifier instanceof Metric) 2
            BeanProvider.getContextualReference(bm, com.codahale.metrics.Metric.class, false, qualifier);
    }
    //...
}
```

1. Gets all the `Metric` beans
2. Retrieves an instance that will use our custom producer and thus will fill the registry

Second goal 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 MetricExtension implements Extension {

  void addMetricQualifier(@Observes BeforeBeanDiscovery bbd) {
      bbd.addQualifier(Metric.class);
  }

  void addTimedInterceptorBinding(@Observes BeforeBeanDiscovery bbd) throws NoSuchMethodException {
      Annotation nonBinding = new AnnotationLiteral<Nonbinding>() {};
      bbd.addInterceptorBinding(new AnnotatedTypeBuilder().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, BeanManager
bm) {
      String name = pp.getAnnotatedMember().getAnnotation(Metric.class).name();
      pp.setProducer(new MetricProducer<>(pp.getProducer(), bm, name);
  }

  void registerProduceMetrics(@Observes AfterDeploymentValidation adv, BeanManager bm) {
      for (Bean<?> bean : bm.getBeans(com.codahale.metrics.Metric.class, new AnyLiteral()))
          for (Annotation qualifier : bean.getQualifiers())
              if (qualifier instanceof Metric)
                  BeanProvider.getContextualReference(bm, com.codahale.metrics.Metric.class, false, qualifier);
    }
}
```

Test your CDI knowledge

Quizz time

Find the valid injections points

```
class MySuperBean {

    @Inject
    Bean<MySuperBean> myMeta;                                   // A [ ]

    @Inject
    Bean<MyService> serviceMeta;                                // B [ ]

    public MySuperBean(@Inject MyService service) {/*...*/}     // C [ ]

    @Inject
    private void myInitMethod(MyService service) {/*...*/}      // D [ ]

    @Inject
    @PostConstruct
    public void myInitMethod2(MyService service) {/*...*/}      // E [ ]
}
```

Solution

```
class MySuperBean {

    @Inject
    Bean<MySuperBean> myMeta;                                   // A [X]

    @Inject
    Bean<MyService> serviceMeta;                                // B [ ]

    public MySuperBean(@Inject MyService service) {/*...*/}     // C [ ]

    @Inject
    private void myInitMethod(MyService service) {/*...*/}      // D [X]

    @Inject
    @PostConstruct
    public void myInitMethod2(MyService service) {/*...*/}      // E [ ]
}
```

Find Beans candidates without `beans.xml` in jar (CDI 1.2)

```
@Decorator
public abstract class MyDecorator implements MyService {/*...*/}    // A [ ]

@Stateless
public class MyServiceImpl implements MyService {/*...*/}           // B [ ]

public class MyBean {/*...*/}                                       // C [ ]

@Model
public class MyBean {/*...*/}                                       // D [ ]

@Singleton
public class MyBean {/*...*/}                                       // E [ ]

@ConversationScoped
public class MyBean {/*...*/}                                       // F [ ]
```

Solution

```
@Decorator
public abstract class MyDecorator implements MyService {/*...*/}     // A [X]

@Stateless
public class MyServiceImpl implements MyService {/*...*/}            // B [X]

public class MyBean {/*...*/}                                        // C [ ]

@Model
public class MyBean {/*...*/}                                        // D [X]

@Singleton
public class MyBean {/*...*/}                                        // E [ ]

@ConversationScoped
public class MyBean {/*...*/}                                        // F [X]
```

Find the valid producers

```
@ApplicationScoped
public class MyBean {

  @Produces
  public Service produce1(InjectionPoint ip, Bean<Service> myMeta) {/*...*/} // A [ ]

  @Produces
  @SessionScoped
  public Service produce2(InjectionPoint ip) {/*...*/}                       // B [ ]

  @Produces
  public Map<K, V> produceMap(InjectionPoint ip) {/*...*/}                   // C [ ]

  @Produces
  public Map<String, ? extends Service> produceMap2() {/*...*/}              // D [ ]
}
```

Solution

```
@ApplicationScoped
public class MyBean {

  @Produces
  public Service produce1(InjectionPoint ip, Bean<Service> myMeta) {/*...*/} // A [X]

  @Produces
  @SessionScoped
  public Service produce2(InjectionPoint ip) {/*...*/}                       // B [ ]

  @Produces
  public Map<K, V> produceMap(InjectionPoint ip) {/*...*/}                   // C [X]

  @Produces
  public Map<String, ? extends Service> produceMap2() {/*...*/}              // D [ ]
}
```

Which observers will be triggered?

```
class FirstBean {

    @Inject
    Event<Post> postEvent;

    public void saveNewPost(Post myPost) {
        postEvent.select(new AnnotationLiteral() < French > {}).fire(myPost);
    }
}

class SecondBean {

    void listenFrPost(@Observes @French Post post) {/*...*/}      // A [ ]
    void listenPost(@Observes Post post) {/*...*/}                // B [ ]
    void listenEnPost(@Observes @English Post post) {/*...*/}     // C [ ]
    void listenObject(@Observes Object obj) {/*...*/}             // D [ ]
}
```

Solution

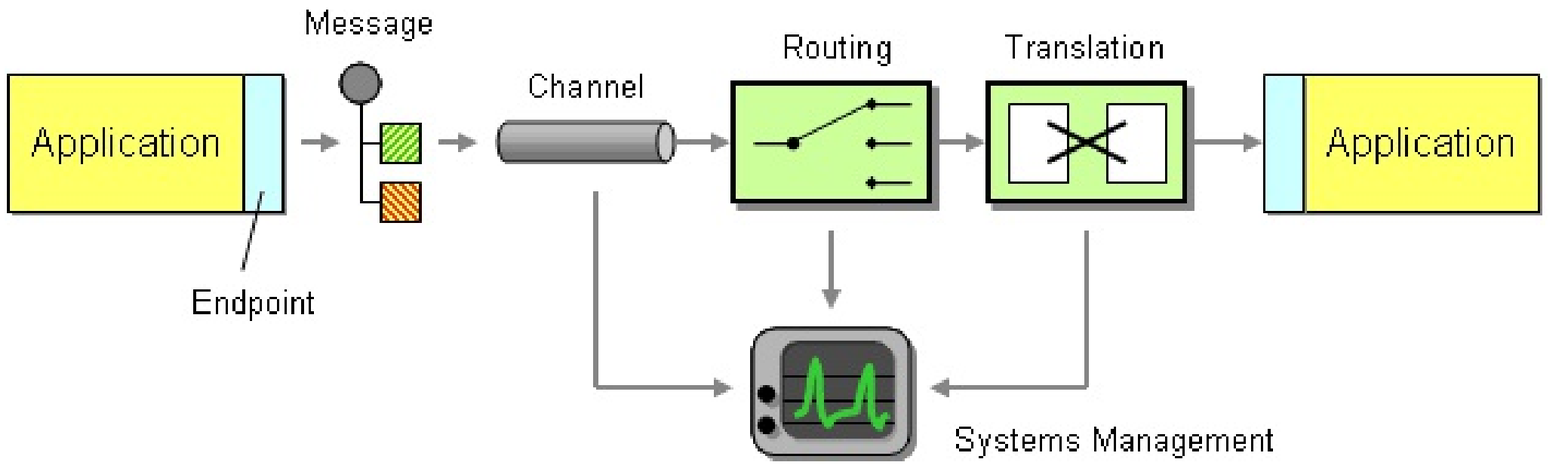
```
class FirstBean {

    @Inject
    Event<Post> postEvent;

    public void saveNewPost(Post myPost) {
        postEvent.select(new AnnotationLiteral() < French > {}).fire(myPost);
    }
}

class SecondBean {

    void listenFrPost(@Observes @French Post post) {/*...*/}      // A [X]
    void listenPost(@Observes Post post) {/*...*/}                // B [X]
    void listenEnPost(@Observes @English Post post) {/*...*/}     // C [ ]
    void listenObject(@Observes Object obj) {/*...*/}             // D [X]
}
```

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")
          .convertBodyTo(String.class)
          .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

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")
            .convertBodyTo(String.class)
            .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

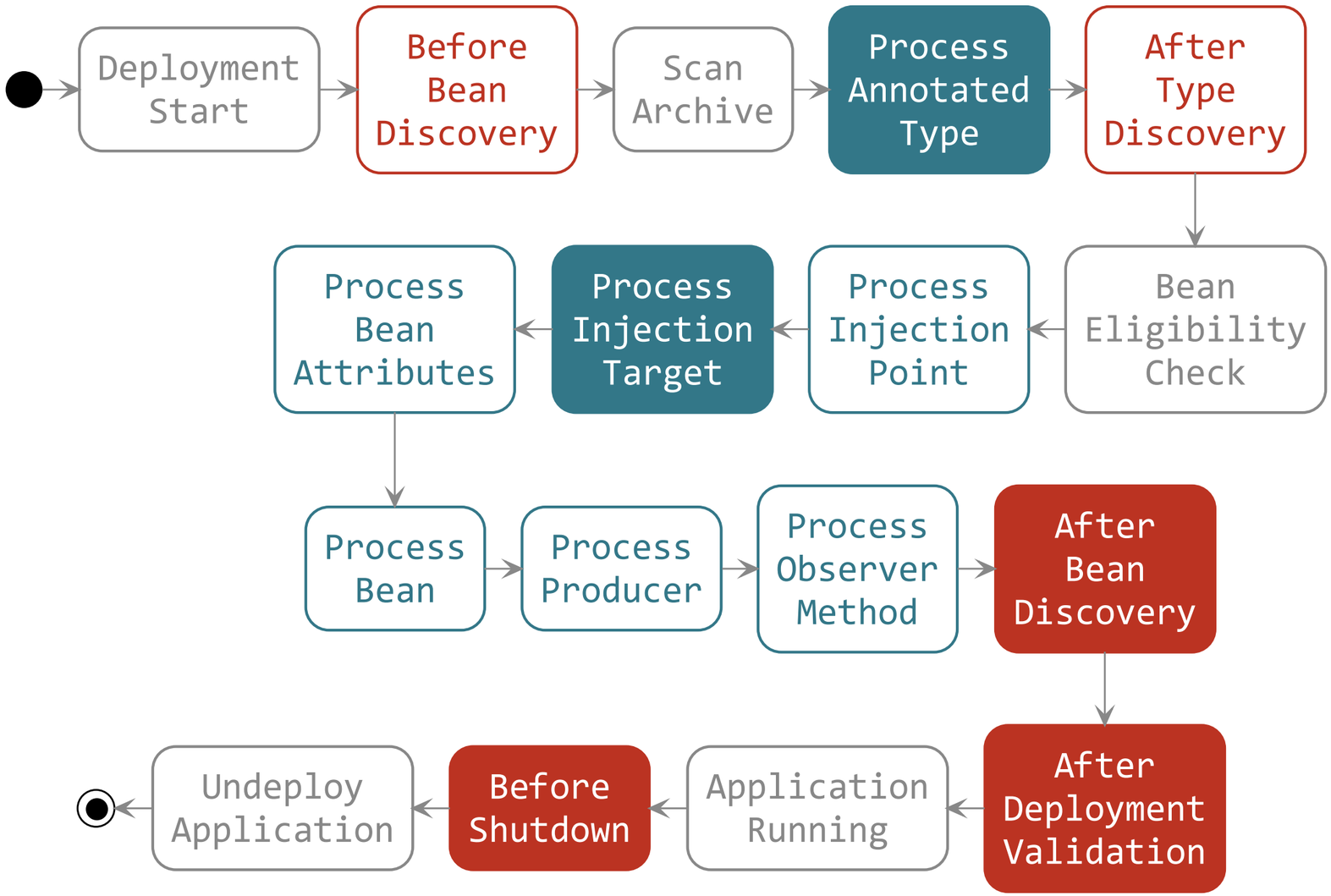
- Manage the creation and the configuration of the `CamelContext` bean
- Bind the `CamelContext` lifecycle that of the CDI container
- Implement the Camel SPI to look up CDI bean references
- Use a custom `InjectionTarget` for CDI beans containing Camel annotations
- Use the magic

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.singleton((Annotation) new AnnotationLiteral<Default>(){});
    }
    public Set<Type> getTypes() { return Collections.singleton((Type) 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 "camel-cdi"; }
    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` lifecyle 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();
    }
}
```

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` lifecyle 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(value = "jms.maxConnections", defaultValue = "10")
    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")
    .convertBodyTo(String.class)
    .log("Sending message [${body}] to JMS...")
    .to("sjms:queue:output");
```

with CDI observers

```
from("file:target/input?delay=1000")
    .convertBodyTo(String.class)
    .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` lifecyle 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")
    .convertBodyTo(String.class)
    .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(),
        history.get(history.size() - 1).getNode().getLabel());
}
```

Conclusion

References

- CDI Specification - cdi-spec.org
- Metrics CDI sources - github.com/astefanutti/metrics-cdi
- Camel CDI sources - github.com/astefanutti/camel-cdi
- Slides sources - github.com/astefanutti/further-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