CDI

Camel CDI

The Camel CDI component provides auto-configuration for Apache Camel using CDI as dependency injection framework based on *convention-over-configuration*. It auto-detects Camel routes available in the application and provides beans for common Camel primitives like **`Endpoint`**, **`ProducerTemplate`** or **`TypeConverter`**. It implements standard Camel bean integration so that Camel annotations like **`@Consume`**, **`@Produce`** and **`@PropertyInject`** can be used seamlessly in CDI beans. Besides, it bridges Camel events (e.g. **`RouteAddedEvent`**, **`CamelContextStartedEvent`**, **`ExchangeCompletedEvent`**, ...) as CDI events and provides a CDI events endpoint that can be used to consume / produce CDI events from / to Camel routes.

While the Camel CDI component is available as of **Camel 2.10**, it's been rewritten in **Camel 2.17** to better fit into the CDI programming model. Hence some of the features like the Camel events to CDI events bridge and the CDI events endpoint only apply starting Camel 2.17.
More details on how to test Camel CDI applications are available in Camel CDI testing.

Auto-Configured Camel Context

Camel CDI automatically deploys and configures a **`CamelContext`** bean. That **`CamelContext`** bean is automatically instantiated, configured and started (resp. stopped) when the CDI container initializes (resp. shuts down). It can be injected in the application, e.g.:

```
@Inject
CamelContext context;
```

That default **`CamelContext`** bean is qualified with the built-in **`@Default`** qualifier, is scoped **`@ApplicationScoped`** and is of type **`DefaultCamelContext`**.

Note that this bean can be customized programmatically and other Camel context beans can be deployed in the application as well.

Auto-Detecting Camel Routes

Camel CDI automatically collects all the **`RoutesBuilder`** beans in the application, instantiates and add them to the **`CamelContext`** bean instance when the CDI container initializes. For example, adding a Camel route is as simple as declaring a class, e.g.:

```
class MyRouteBean extends RouteBuilder {

        @Override
    public void configure() {
        from("jms:invoices").to("file:/invoices");
    }
}
```

Note that you can declare as many **`RoutesBuilder`** beans as you want. Besides, **`RouteContainer`** beans are also automatically collected, instantiated and added to the **`CamelContext`** bean instance managed by Camel CDI when the container initializes.

Available as of Camel 2.19

In some situations, it may be necessary to disable the auto-configuration of the **`RouteBuilder`** and **`RouteContainer`** beans. That can be achieved by observing for the **`CdiCamelConfiguration`** event, e.g.:

```
static void configuration(@Observes CdiCamelConfiguration configuration) {
    configuration.autoConfigureRoutes(false);
}
```

Similarly, it is possible to deactivate the automatic starting of the configured `CamelContext` beans, e.g.:

```
static void configuration(@Observes CdiCamelConfiguration configuration) {
    configuration.autoStartContexts(false);
}
```

Auto-Configured Camel Primitives

Camel CDI provides beans for common Camel primitives that can be injected in any CDI beans, e.g.:

```
@Inject
@Uri("direct:inbound")
ProducerTemplate producerTemplate;

@Inject
MockEndpoint outbound; // URI defaults to the member name, i.e. mock:outbound

@Inject
@Uri("direct:inbound")
Endpoint endpoint;

@Inject
TypeConverter converter;
```

Camel Context Configuration

If you just want to change the name of the default **`CamelContext`** bean, you can used the **`@ContextName`** qualifier provided by Camel CDI, e.g.:

```
@ContextName("camel-context")
class MyRouteBean extends RouteBuilder {

        @Override
    public void configure() {
        from("jms:invoices").to("file:/invoices");
    }
}
```

Else, if more customization is needed, any **`CamelContext`** class can be used to declare a custom Camel context bean. Then, the **`@PostConstruct`** and **`@PreDestroy`** lifecycle callbacks can be done to do the customization, e.g.:

```
@ApplicationScoped
class CustomCamelContext extends DefaultCamelContext {

    @PostConstruct
    void customize() {
        // Set the Camel context name
        setName("custom");
        // Disable JMX
        disableJMX();
    }

    @PreDestroy
    void cleanUp() {
        // ...
    }
}
```

Producer and disposer methods can also be used as well to customize the Camel context bean, e.g.:

```
class CamelContextFactory {

    @Produces
    @ApplicationScoped
    CamelContext customize() {
        DefaultCamelContext context = new DefaultCamelContext();
        context.setName("custom");
        return context;
    }

    void cleanUp(@Disposes CamelContext context) {
        // ...
    }
}
```

Similarly, producer fields can be used, e.g.:

```
@Produces
@ApplicationScoped
CamelContext context = new CustomCamelContext();

class CustomCamelContext extends DefaultCamelContext {

    CustomCamelContext() {
        setName("custom");
    }
}
```

This pattern can be used for example to avoid having the Camel context routes started automatically when the container initializes by calling the **`setAutoStartup`** method, e.g.:

```
@ApplicationScoped
class ManualStartupCamelContext extends DefaultCamelContext {

    @PostConstruct
    void manual() {
        setAutoStartup(false);
    }
}
```

Multiple Camel Contexts

Any number of **`CamelContext`** beans can actually be declared in the application as documented above. In that case, the CDI qualifiers declared on these **`CamelContext`** beans are used to bind the Camel routes and other Camel primitives to the corresponding Camel contexts. From example, if the following beans get declared:

```
@ApplicationScoped
@ContextName("foo")
class FooCamelContext extends DefaultCamelContext {
}

@ApplicationScoped
@BarContextQualifier
class BarCamelContext extends DefaultCamelContext {
}

@ContextName("foo")
class RouteAdddedToFooCamelContext extends RouteBuilder {

        @Override
    public void configure() {
        // ...
    }
}

@BarContextQualifier
class RouteAdddedToBarCamelContext extends RouteBuilder {

        @Override
    public void configure() {
        // ...
    }
}

@ContextName("baz")
class RouteAdddedToBazCamelContext extends RouteBuilder {

        @Override
    public void configure() {
        // ...
    }
}

@MyOtherQualifier
class RouteNotAddedToAnyCamelContext extends RouteBuilder {

        @Override
    public void configure() {
        // ...
    }
}
```

The **`RoutesBuilder`** beans qualified with **`@ContextName`** are automatically added to the corresponding **`CamelContext`** beans by Camel CDI. If no such **`CamelContext`** bean exists, it gets automatically created, as for the **`RouteAddedToBazCamelContext`** bean. Note this only happens for the **`@ContextName`** qualifier provided by Camel CDI. Hence the **`RouteNotAddedToAnyCamelContext`** bean qualified with the user-defined **`@MyOtherQualifier`** qualifier does not get added to any Camel contexts. That may be useful, for example, for Camel routes that may be required to be added later during the application execution.

Since Camel version 2.17.0, Camel CDI is capable of managing any kind of **`CamelContext`** beans. In previous versions, it is only capable of managing beans of type **`CdiCamelContext`** so it is required to extend it.

The CDI qualifiers declared on the **`CamelContext`** beans are also used to bind the corresponding Camel primitives, e.g.:

```
@Inject
@ContextName("foo")
@Uri("direct:inbound")
ProducerTemplate producerTemplate;

@Inject
@BarContextQualifier
MockEndpoint outbound; // URI defaults to the member name, i.e. mock:outbound

@Inject
@ContextName("baz")
@Uri("direct:inbound")
Endpoint endpoint;
```

Configuration Properties

To configure the sourcing of the configuration properties used by Camel to resolve properties placeholders, you can declare a **`PropertiesComponent`** be an qualified with **`@Named("properties")`**, e.g.:

```
@Produces
@ApplicationScoped
@Named("properties")
PropertiesComponent propertiesComponent() {
    Properties properties = new Properties();
    properties.put("property", "value");
    PropertiesComponent component = new PropertiesComponent();
    component.setInitialProperties(properties);
    component.setLocation("classpath:placeholder.properties");
    return component;
}
```

If you want to use DeltaSpike configuration mechanism you can declare the following **`PropertiesComponent`** bean:

```
@Produces
@ApplicationScoped
@Named("properties")
PropertiesComponent properties(PropertiesParser parser) {
    PropertiesComponent component = new PropertiesComponent();
    component.setPropertiesParser(parser);
    return component;
}

// PropertiesParser bean that uses DeltaSpike to resolve properties
static class DeltaSpikeParser extends DefaultPropertiesParser {
    @Override
    public String parseProperty(String key, String value, Properties properties) {
        return ConfigResolver.getPropertyValue(key);
    }
}
```

You can see the **`camel-example-cdi-properties`** example for a working example of a Camel CDI application using DeltaSpike configuration mechanism.

Auto-Configured Type Converters

CDI beans annotated with the **`@Converter`** annotation are automatically registered into the deployed Camel contexts, e.g.:

```
@Converter
public class MyTypeConverter {

    @Converter
    public Output convert(Input input) {
        //...
    }
}
```

Note that CDI injection is supported within the type converters.

Camel Bean Integration

Camel Annotations

As part of the Camel bean integration, Camel comes with a set of annotations that are seamlessly supported by Camel CDI. So you can use any of these annotations in your CDI beans, e.g.:

	Camel annotation	CDI equivalent

Configuration property	`@PropertyInject("key")`
	`String value;`
		If using DeltaSpike configuration mechanism:
		`@Inject`
		`@ConfigProperty(name = "key")`
		`String value;`
		See configuration properties for more details.

Producer template injection (default Camel context)	`@Produce(uri = "mock:outbound")`
	`ProducerTemplate producer;`
		`@Inject`
		`@Uri("direct:outbound")`
		`ProducerTemplate producer;`

Endpoint injection (default Camel context)	`@EndpointInject(uri = "direct:inbound")`
	`Endpoint endpoint;`
		`@Inject`
		`@Uri("direct:inbound")`
		`Endpoint endpoint;`

Endpoint injection (Camel context by name)	`@EndpointInject(uri = "direct:inbound",`
	`context = "foo")`
	`Endpoint contextEndpoint;`
		`@Inject`
		`@ContextName("foo")`
		`@Uri("direct:inbound")`
		`Endpoint contextEndpoint;`

Bean injection (by type)	`@BeanInject` `MyBean bean;`	`@Inject` `MyBean bean;`

Bean injection (by name)	`@BeanInject("foo")`
	`MyBean bean;`
		`@Inject`
		`@Named("foo")`
		`MyBean bean;`

POJO consuming	`@Consume(uri = "seda:inbound")`
	`void consume(@Body String body) {`
	`  //...`
	`}`	

Bean Component

You can refer to CDI beans, either by type or name, from the Camel DSL, e.g., using the Java DSL:

```
class MyBean {
        //...
}

from("direct:inbound").bean(MyBean.class);
```

Or to lookup a CDI bean by name from the Java DSL:

```
@Named("foo")
class MyNamedBean {
        //...
}

from("direct:inbound")
  .bean("foo");
```

Referring Beans From Endpoint URIs

When configuring endpoints using the URI syntax you can refer to beans in the Registry using the # notation. If the URI parameter value starts with a # sign then Camel CDI will lookup for a bean of the given type by name, e.g.:

```
from("jms:queue:{{destination}}?transacted=true&transactionManager=#jtaTransactionManager")
  .to("...");
```

Having the following CDI bean qualified with **`@Named("jtaTransactionManager")`**:

```
@Produces
@Named("jtaTransactionManager")
PlatformTransactionManager createTransactionManager(TransactionManager transactionManager, UserTransaction
userTransaction) {
    JtaTransactionManager jtaTransactionManager = new JtaTransactionManager();
    jtaTransactionManager.setUserTransaction(userTransaction);
    jtaTransactionManager.setTransactionManager(transactionManager);
    jtaTransactionManager.afterPropertiesSet();
    return jtaTransactionManager;
}
```

Camel Events to CDI Events

Available as of Camel 2.17

Camel provides a set of management events that can be subscribed to for listening to Camel context, service, route and exchange events. Camel CDI seamlessly translates these Camel events into CDI events that can be observed using CDI observer methods, e.g.:

```
void onContextStarting(@Observes CamelContextStartingEvent event) {
    // Called before the default Camel context is about to start
}
```

From Camel 2.18: it's possible to observe events for a particular route (**`RouteAddedEvent`**, **`RouteStartedEvent`**, **`RouteStoppedEvent`** and **`RouteRemovedEvent`**) should it have an explicit defined, e.g.:

```
from("...").routeId("foo").to("...");

void onRouteStarted(@Observes @Named("foo") RouteStartedEvent event) {
    // Called after the route "foo" has started
}
```

When multiple Camel contexts exist in the CDI container, the Camel context bean qualifiers, like **`@ContextName`**, can be used to refine the observer method resolution to a particular Camel context as specified in observer resolution, e.g.:

```
void onRouteStarted(@Observes @ContextName("foo") RouteStartedEvent event) {
    // Called after the route 'event.getRoute()' for the Camel context 'foo' has started
}

void onContextStarted(@Observes @Manual CamelContextStartedEvent event) {
    // Called after the the Camel context qualified with '@Manual' has started
}
```

Similarly, the **`@Default`** qualifier can be used to observe Camel events for the *default* Camel context if multiples contexts exist, e.g.:

```
void onExchangeCompleted(@Observes @Default ExchangeCompletedEvent event) {
    // Called after the exchange 'event.getExchange()' processing has completed
}
```

In that example, if no qualifier is specified, the **`@Any`** qualifier is implicitly assumed, so that corresponding events for all the Camel contexts get received.

Note that the support for Camel events translation into CDI events is only activated if observer methods listening for Camel events are detected in the deployment, and that per Camel context.

CDI Events Endpoint

Available as of Camel 2.17

The CDI event endpoint bridges the CDI events with the Camel routes so that CDI events can be seamlessly observed / consumed (resp. produced / fired) from Camel consumers (resp. by Camel producers).

The **`CdiEventEndpoint<T>`** bean provided by Camel CDI can be used to observe / consume CDI events whose *event type* is **`T`**, for example:

```
@Inject
CdiEventEndpoint<String> cdiEventEndpoint;

from(cdiEventEndpoint).log("CDI event received: ${body}");
```

This is equivalent to writing:

```
@Inject
@Uri("direct:event")
ProducerTemplate producer;

void observeCdiEvents(@Observes String event) {
    producer.sendBody(event);
}

from("direct:event")
  .log("CDI event received: ${body}");
```

Conversely, the **`CdiEventEndpoint<T>`** bean can be used to produce / fire CDI events whose *event type* is **`T`**, for example:

```
@Inject
CdiEventEndpoint<String> cdiEventEndpoint;

from("direct:event")
  .to(cdiEventEndpoint).log("CDI event sent: ${body}");
```

This is equivalent to writing:

```
@Inject
Event<String> event;

from("direct:event").process(new Processor() {
    @Override
    public void process(Exchange exchange) {
        event.fire(exchange.getBody(String.class));
    }
}).log("CDI event sent: ${body}");
```

Or using a Java 8 lambda expression:

```
@Inject
Event<String> event;

from("direct:event")
    .process(exchange -> event.fire(exchange.getIn().getBody(String.class)))
    .log("CDI event sent: ${body}");
```

The type variable **`T`** (resp. the qualifiers) of a particular **`CdiEventEndpoint<T>`** injection point are automatically translated into the parameterized *event type* (resp. into the *event qualifiers*) e.g.:

```
@Inject
@FooQualifier
CdiEventEndpoint<List<String>> cdiEventEndpoint;

from("direct:event").to(cdiEventEndpoint);

void observeCdiEvents(@Observes @FooQualifier List<String> event) {
    logger.info("CDI event: {}", event);
}
```

When multiple Camel contexts exist in the CDI container, the Camel context bean qualifiers, like **`@ContextName`**, can be used to qualify the **`CdiEventEndpoint<T>`** injection points, e.g.:

```
@Inject
@ContextName("foo")
CdiEventEndpoint<List<String>> cdiEventEndpoint;
// Only observes / consumes events having the @ContextName("foo") qualifier
from(cdiEventEndpoint).log("Camel context (foo) > CDI event received: ${body}");
// Produces / fires events with the @ContextName("foo") qualifier
from("...").to(cdiEventEndpoint);

void observeCdiEvents(@Observes @ContextName("foo") List<String> event) {
    logger.info("Camel context (foo) > CDI event: {}", event);
}
```

Note that the CDI event Camel endpoint dynamically adds an observer method for each unique combination of *event type* and *event qualifiers* and solely relies on the container typesafe observer resolution, which leads to an implementation as efficient as possible.

Besides, as the impedance between the *typesafe* nature of CDI and the *dynamic* nature of the Camel component model is quite high, it is not possible to create an instance of the CDI event Camel endpoint via URIs. Indeed, the URI format for the CDI event component is:

```
cdi-event://PayloadType<T1,...,Tn>[?qualifiers=QualifierType1[,...[,QualifierTypeN]...]]
```

With the authority **`PayloadType`** (resp. the **`QualifierType`**) being the URI escaped fully qualified name of the payload (resp. qualifier) raw type followed by the type parameters section delimited by angle brackets for payload parameterized type. Which leads to *unfriendly* URIs, e.g.:

```
cdi-event://org.apache.camel.cdi.example.EventPayload%3Cjava.lang.Integer%3E?qualifiers=org.apache.camel.cdi.
example.FooQualifier%2Corg.apache.camel.cdi.example.BarQualifier
```

But more fundamentally, that would prevent efficient binding between the endpoint instances and the observer methods as the CDI container doesn't have any ways of discovering the Camel context model during the deployment phase.

Camel XML Configuration Import

Available as of Camel 2.18

While CDI favors a typesafe dependency injection mechanism, it may be useful to reuse existing Camel XML configuration files into a Camel CDI application. In other use cases, it might be handy to rely on the Camel XML DSL to configure its Camel context(s).

You can use the **`@ImportResource`** annotation that's provided by Camel CDI on any CDI beans and Camel CDI will automatically load the Camel XML configuration at the specified locations, e.g.:

```
@ImportResource("camel-context.xml")
class MyBean {
}
```

Camel CDI will load the resources at the specified locations from the classpath (other protocols may be added in the future).

Every **`CamelContext`** elements and other Camel primitives from the imported resources are automatically deployed as CDI beans during the container bootstrap so that they benefit from the auto-configuration provided by Camel CDI and become available for injection at run-time. If such an element has an explicit `id` attribute set, the corresponding CDI bean is qualified with the `@Named` qualifier, e.g., given the following Camel XML configuration:

```
<camelContext id="foo">
    <endpoint id="bar" uri="seda:inbound">
        <property key="queue" value="#queue"/>
        <property key="concurrentConsumers" value="10"/>
    </endpoint>
<camelContext/>
```

The corresponding CDI beans are automatically deployed and can be injected, e.g.:

```
@Inject
@ContextName("foo")
CamelContext context;

@Inject
@Named("bar")
Endpoint endpoint;
```

Note that the **`CamelContext`** beans are automatically qualified with both the `Named` and **`ContextName`** qualifiers. If the imported **`CamelContext`** element doesn't have an **`id`** attribute, the corresponding bean is deployed with the built-in **`Default`** qualifier.

Conversely, CDI beans deployed in the application can be referred to from the Camel XML configuration, usually using the **`ref`** attribute, e.g., given the following bean declared:

```
@Produces
@Named("baz")
Processor processor = exchange -> exchange.getIn().setHeader("qux", "quux");
```

A reference to that bean can be declared in the imported Camel XML configuration, e.g.:

```
<camelContext id="foo">
    <route>
        <from uri="..."/>
        <process ref="baz"/>
    </route>
<camelContext/>
```

Transaction support

Available as of Camel 2.19

Camel CDI provides support for Camel transactional client using JTA.

That support is optional hence you need to have JTA in your application classpath, e.g., by explicitly add JTA as a dependency when using Maven:

```
<dependency>
    <groupId>javax.transaction</groupId>
    <artifactId>javax.transaction-api</artifactId>
    <scope>runtime</scope>
</dependency>
```

You'll have to have your application deployed in a JTA capable container or provide a standalone JTA implementation.

Note that, for the time being, the transaction manager is looked up as JNDI resource with the `java:/TransactionManager` key. More flexible strategies will be added in the future to support a wider range of deployment scenarios.

Transaction policies

Camel CDI provides implementation for the typically supported Camel `TransactedPolicy` as CDI beans. It is possible to have these policies looked up by name using the `transacted` EIP, e.g.:

```
class MyRouteBean extends RouteBuilder {

        @Override
    public void configure() {
        from("activemq:queue:foo")
            .transacted("PROPAGATION_REQUIRED")
            .bean("transformer")
            .to("jpa:my.application.entity.Bar")
            .log("${body.id} inserted");
    }
}
```

This would be equivalent to:

```
class MyRouteBean extends RouteBuilder {

    @Inject
    @Named("PROPAGATION_REQUIRED")
    Policy required;

        @Override
    public void configure() {
        from("activemq:queue:foo")
            .policy(required)
            .bean("transformer")
            .to("jpa:my.application.entity.Bar")
            .log("${body.id} inserted");
    }
}
```

The list of supported transaction policy names is: `PROPAGATION_NEVER`, `PROPAGATION_NOT_SUPPORTED`, `PROPAGATION_SUPPORTS`, `PROPAGATION_REQUIRED`, `PROPAGATION_REQUIRES_NEW`, `PROPAGATION_NESTED`, `PROPAGATION_MANDATORY`.

Transactional error handler

Camel CDI provides a transactional error handler that extends the redelivery error handler, forces a rollback whenever an exception occurs and creates a new transaction for each redelivery. Camel CDI provides the `CdiRouteBuilder` class that exposes the `transactionErrorHandler` helper method to enable quick access to the configuration, e.g.:

```
class MyRouteBean extends CdiRouteBuilder {

        @Override
    public void configure() {
        errorHandler(transactionErrorHandler()
            .setTransactionPolicy("PROPAGATION_SUPPORTS")
            .maximumRedeliveries(5)
            .maximumRedeliveryDelay(5000)
            .collisionAvoidancePercent(10)
            .backOffMultiplier(1.5));
    }
}
```

Auto-configured OSGi integration

Available as of Camel 2.17

The Camel context beans are automatically adapted by Camel CDI so that they are registered as OSGi services and the various resolvers (like **`ComponentResolver`** and **`DataFormatResolver`**) integrate with the OSGi registry. That means that the Karaf Camel commands can be used to operate the Camel contexts auto-configured by Camel CDI, e.g.:

```
karaf@root()> camel:context-list
 Context        Status              Total #       Failed #     Inflight #   Uptime
 -------        ------              -------       --------     ----------   ------
 camel-cdi      Started                   1              0              0   1 minute
```

See the **`camel-example-cdi-osgi`** example for a working example of the Camel CDI OSGi integration.

Lazy Injection / Programmatic Lookup

Available as of Camel 2.17

While the CDI programmatic model favors a type-safe resolution mechanism that occurs at application initialization time, it is possible to perform dynamic / lazy injection later during the application execution using the programmatic lookup mechanism.

Camel CDI provides for convenience the annotation literals corresponding to the CDI qualifiers that you can use for standard injection of Camel primitives. These annotation literals can be used in conjunction with the **`javax.enterprise.inject.Instance`** interface which is the CDI entry point to perform lazy injection / programmatic lookup.

For example, you can use the provided annotation literal for the **`@Uri`** qualifier to lazily lookup for Camel primitives, e.g. for **`ProducerTemplate`** beans:

```
@Any
@Inject
Instance<ProducerTemplate> producers;

ProducerTemplate inbound = producers
    .select(Uri.Literal.of("direct:inbound"))
    .get();
```

Or for **`Endpoint`** beans, e.g.:

```
@Any
@Inject
Instance<Endpoint> endpoints;

MockEndpoint outbound = endpoints
    .select(MockEndpoint.class, Uri.Literal.of("mock:outbound"))
    .get();
```

Similarly, you can use the provided annotation literal for the **`@ContextName`** qualifier to lazily lookup for **`CamelContext`** beans, e.g.:

```
@Any
@Inject
Instance<CamelContext> contexts;

CamelContext context = contexts
    .select(ContextName.Literal.of("foo"))
    .get();
```

You can also refined the selection based on the Camel context type, e.g.:

```
@Any
@Inject
Instance<CamelContext> contexts;

// Refine the selection by type
Instance<DefaultCamelContext> context = contexts.select(DefaultCamelContext.class);

// Check if such a bean exists then retrieve a reference
if (!context.isUnsatisfied())
    context.get();
```

Or even iterate over a selection of Camel contexts, e.g.:

```
@Any
@Inject
Instance<CamelContext> contexts;

for (CamelContext context : contexts)
    context.setUseBreadcrumb(true);
```

Maven Archetype

Among the available Camel Maven archetypes, you can use the provided **`camel-archetype-cdi`** to generate a Camel CDI Maven project, e.g.:

```
mvn archetype:generate -DarchetypeGroupId=org.apache.camel.archetypes -DarchetypeArtifactId=camel-archetype-cdi
```

Supported Containers

The Camel CDI component is compatible with any CDI 1.0, CDI 1.1 and CDI 1.2 compliant runtime. It's been successfully tested against the following runtimes:

Container	Version	Runtime
Weld SE	`1.1.28.Final`	CDI 1.0 / Java SE 7
OpenWebBeans	`1.2.7`	CDI 1.0 / Java SE 7
Weld SE	`2.4.1.Final`	CDI 1.2 / Java SE 7
OpenWebBeans	`1.7.0`	CDI 1.2 / Java SE 7
WildFly	`8.2.1.Final`	CDI 1.2 / Java EE 7
WildFly	`9.0.1.Final`	CDI 1.2 / Java EE 7
WildFly	`10.0.0.Final`	CDI 1.2 / Java EE 7
Karaf	`2.4.4`	CDI 1.2 / OSGi 4 / PAX CDI
Karaf	`3.0.5`	CDI 1.2 / OSGi 5 / PAX CDI
Karaf	`4.0.4`	CDI 1.2 / OSGi 6 / PAX CDI

Examples

The following examples are available in the **`examples`** directory of the Camel project:

Example	Description
`camel-example-cdi`	Illustrates how to work with Camel using CDI to configure components, endpoints and beans
`camel-example-cdi-kubernetes`	Illustrates the integration between Camel, CDI and Kubernetes
`camel-example-cdi-metrics`	Illustrates the integration between Camel, Dropwizard Metrics and CDI
`camel-example-cdi-properties`	Illustrates the integration between Camel, DeltaSpike and CDI for configuration properties
`camel-example-cdi-osgi`	A CDI application using the SJMS component that can be executed inside an OSGi container using PAX CDI
`camel-example-cdi-test`	Demonstrates the testing features that are provided as part of the integration between Camel and CDI
`camel-example-cdi-rest-servlet`	Illustrates the Camel REST DSL being used in a Web application that uses CDI as dependency injection framework
`camel-example-cdi-xml`	Illustrates the use of Camel XML configuration files into a Camel CDI application
`camel-example-widget-gadget-cdi`	The Widget and Gadget use-case from the EIP book implemented in Java with CDI dependency injection
`camel-example-swagger-cdi`	An example using REST DSL and Swagger Java with CDI

See Also

- Camel CDI Testing
- CDI Web site
- CDI ecosystem
- Going further with CDI (See Camel CDI section)