Guava EventBus

Guava EventBus Component

Available since Camel 2.10.0

The Google Guava EventBus allows publish-subscribe-style communication between components without requiring the components to explicitly register with one another (and thus be aware of each other). The **guava-eventbus:** component provides integration bridge between Camel and Google Guava EventBus infrastructure. With the latter component, messages exchanged with the Guava `EventBus` can be transparently forwarded to the Camel routes. EventBus component allows also to route body of Camel exchanges to the Guava `EventBus`.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-guava-eventbus</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI format

```
guava-eventbus:busName[?options]
```

Where **busName** represents the name of the `com.google.common.eventbus.EventBus` instance located in the Camel registry.

Options

Name	Default Value	Description
`eventClass`	`null`	**Camel 2.10:** If used on the consumer side of the route, will filter events received from the `EventBus` to the instances of the class and superclasses of `eventClass`. Null value of this option is equal to setting it to the `java.lang.Object` i.e. the consumer will capture all messages incoming to the event bus. This option cannot be used together with `listenerInterface` option.
`listenerInterface`	`null`	**Camel 2.11:** The interface with method(s) marked with the `@Subscribe` annotation. Dynamic proxy will be created over the interface so it could be registered as the `EventBus` listener. Particularly useful when creating multi-event listeners and for handling `DeadEvent` properly. This option cannot be used together with `eventClass` option.

Usage

Using `guava-eventbus` component on the consumer side of the route will capture messages sent to the Guava `EventBus` and forward them to the Camel route. Guava EventBus consumer processes incoming messages asynchronously.

```
SimpleRegistry registry = new SimpleRegistry();
EventBus eventBus = new EventBus();
registry.put("busName", eventBus);
CamelContext camel = new DefaultCamelContext(registry);

from("guava-eventbus:busName").to("seda:queue");

eventBus.post("Send me to the SEDA queue.");
```

Using `guava-eventbus` component on the producer side of the route will forward body of the Camel exchanges to the Guava `EventBus` instance.

```
SimpleRegistry registry = new SimpleRegistry();
EventBus eventBus = new EventBus();
registry.put("busName", eventBus);
CamelContext camel = new DefaultCamelContext(registry);

from("direct:start").to("guava-eventbus:busName");

ProducerTemplate producerTemplate = camel.createProducerTemplate();
producer.sendBody("direct:start", "Send me to the Guava EventBus.");

eventBus.register(new Object(){
  @Subscribe
  public void messageHander(String message) {
    System.out.println("Message received from the Camel: " + message);
  }
});
```

DeadEvent considerations

Keep in mind that due to the limitations caused by the design of the Guava EventBus, you cannot specify event class to be received by the listener without creating class annotated with `@Subscribe` method. This limitation implies that endpoint with `eventClass` option specified actually listens to all possible events (`java.lang.Object`) and filter appropriate messages programmatically at runtime. The snipped below demonstrates an appropriate excerpt from the Camel code base.

```
@Subscribe
public void eventReceived(Object event) {
  if (eventClass == null || eventClass.isAssignableFrom(event.getClass())) {
    doEventReceived(event);
...
```

This drawback of this approach is that `EventBus` instance used by Camel will never generate `com.google.common.eventbus.DeadEvent` notifications. If you want Camel to listen only to the precisely specified event (and therefore enable `DeadEvent` support), use `listenerInterface` endpoint option. Camel will create dynamic proxy over the interface you specify with the latter option and listen only to messages specified by the interface handler methods. The example of the listener interface with single method handling only `SpecificEvent` instances is demonstrated below.

```
package com.example;

public interface CustomListener {

  @Subscribe
  void eventReceived(SpecificEvent event);

}
```

The listener presented above could be used in the endpoint definition as follows.

```
from("guava-eventbus:busName?listenerInterface=com.example.CustomListener").to("seda:queue");
```

Consuming multiple type of events

In order to define multiple type of events to be consumed by Guava EventBus consumer use `listenerInterface` endpoint option, as listener interface could provide multiple methods marked with the `@Subscribe` annotation.

```
package com.example;

public interface MultipleEventsListener {

  @Subscribe
  void someEventReceived(SomeEvent event);

  @Subscribe
  void anotherEventReceived(AnotherEvent event);

}
```

The listener presented above could be used in the endpoint definition as follows.

```
from("guava-eventbus:busName?listenerInterface=com.example.MultipleEventsListener").to("seda:queue");
```