Using CamelProxy

Using CamelProxy

Camel allows you to proxy a producer sending to an Endpoint by a regular interface. Then when clients using this interface can work with it as if its regular java code but in reality its proxied and does a Request Reply to a given endpoint.

Proxy from Spring

You can define a proxy in the spring XML file as shown below{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/config/AnotherCamelProxyTest.xml}Now the client can grab this bean using regular spring bean coding and invoke it as if its just another bean.
The code is based on an unit test but proves the point:{snippet:id=e1|lang=java|url=camel/trunk/components/camel-spring/src/test/java/org/apache/camel/spring/config/AnotherCamelProxyTest.java}

Proxy from Java

You can also create a proxy from regular Java using a `org.apache.camel.component.bean.ProxyHelper` as shown below:

Endpoint endpoint = context.getEndpoint("direct:start"); MyProxySender sender = ProxyHelper.createProxy(endpoint, MyProxySender.class);

In **Camel 2.3** you can use `org.apache.camel.builder.ProxyBuilder` which may be easier to use than ProxyHelper:{snippet:id=e4|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/bean/BeanProxyTest.java}

Proxy with Annotation

Another way to configure the proxy from java is by using the @Produce annotation. Also see POJO Producing.

@Produce(uri="direct:start") MyProxySender sender;

This basically does the same as ProxyHelper.createProxy.

What is send on the Message

When using a proxy Camel will send the message payload as a `org.apache.camel.component.bean.BeanInvocation` object (*Camel 2.15 or older) which holds the details of which method was invoked and what the argument was. From **Camel 2.16** onwards Camel parameter binding is enabled by default, which will use binding information from the method signature parameters to bind to the Exchange/Message with the following annotations

Annotation	Parameter Type	Parameter binds to
@Body	Object	Binds the parameter to the message body
@Header(name)	Object	Binds the parameter to the message header with the given name
@Headers	Map	Binds the parameter to the message headers. The parameter is expected to be of `java.util.Map` type.
@ExchangeProperty (name)	Object	Binds the parameter to the exchange property with the given name

If a parameter does not have any annotation then the parameter is bound to the message body.

For example given the following interface

public interface MyAuditService { void auditMessage(@Header("uuid") String uuid, @Body String body); }

Then from Java DSL we can create a proxy and call the method

// must enable binding on proxy MyAuditService service = new ProxyBuilder(context).endpoint("jms:queue:foo").build(MyAuditService.class); service.auditMessage("1234", "Hello World");

Which will send the message to the JMS queue foo, with the header(uuid)=1234 and body=Hello World. The message is sent as InOnly as the method is void.

The old behavior can be enabled by setting binding off, such as:

// must enable binding on proxy MyAuditService service = new ProxyBuilder(context).endpoint("jms:queue:foo").binding(false).build(MyAuditService.class); service.auditMessage("1234", "Hello World");

Turning the BeanInvocation into a first class payload

Available as of Camel 2.1

If you proxied method signature only have one parameter such as:

String hello(String name);

Then it gives another advantage in Camel as it allows Camel to regard the value passed in to this single parameter as the *real* payload. In other words if you pass in `Camel` to the hello method, then Camel can *see* that as a `java.lang.String` payload with the value of `Camel`. This gives you a great advantage as you can use the proxy as first class services with Camel.

You can proxy Camel and let clients use the pure and clean interfaces as if Camel newer existed. Then Camel can proxy the invocation and receive the input passed into the single method parameter and regard that as if it was *just* the message payload.

From **Camel 2.16** onwards this is improved as binding is enabled out of the box, where Camel binds to the message parameters using the annotation listed in the table above. If a parameter has no annotation then the parameter is bound to the message body.

Okay lets try that with an example

Example with proxy using single parameter methods.

At first we have the interface we wish to proxy{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/bean/OrderService.java}Notice that all methods have single parameters. The return type is optional, as you can see one of them is void.
Also what you should know is that Camel uses its Type Converter mechanism to adapt to the types defined on the methods.

This allows us to easily use `org.w3c.dom.Document` and `String` types with no hazzle.

Okay then we have the following route where we route using a Content Based Router that is XML based. See that we use XPath in the choices to route the message depending on its a book order or not.{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/bean/BeanProxyTest.java}Now there is a couple of tests that shows using the Camel Proxy how we can easily invoke the proxy and do not know its actually Camel doing some routing underneath.{snippet:id=e2|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/bean/BeanProxyTest.java}And this one below shows using different types that Camel adapts to.{snippet:id=e3|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/bean/BeanProxyTest.java}Isn't this cool?

Asynchronous using Future

Available as of Camel 2.8

By default the Camel Proxy invocation is synchronous when invoked from the client. If you want this to be asynchronous you define the return type to be of `java.util.concurrent.Future` type. The `Future` is a handle to the task which the client can use to obtain the result.

For example given this client interface{snippet:id=e1|lang=java|title=Client Interface|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/bean/ProxyReturnFutureTest.java}The client can use this with a Camel Proxy as shown from the following snippet from an unit test:{snippet:id=e2|lang=java|title=Using Client|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/bean/ProxyReturnFutureTest.java}This allows you to fully define your client API without any Camel dependency at all, and decide whether the invocation should be synchronous or asynchronous.

If the Client is asynchronous (return type is Future) then Camel will continue processing the invocation using a thread pool which is being looked up using the key `CamelInvocationHandler`. Its a shared thread pool for all Camel Proxy in the CamelContext. You can define a thread pool profile with the id `CamelInvocationHandler` to configure settings such as min/max threads etc.

See also

- Bean
- User Guide
- Tutorial-JmsRemoting