4

Using beans with Camel

This chapter covers

- Calling Java beans with Camel
- Understanding the Service Activator EIP
- How Camel looks up beans using registries
- How Camel selects bean methods to invoke
- Using bean parameter bindings
- Using Java beans as predicates or expressions in routes

If you've been developing software for many years, you've likely worked with various component models, such as CORBA, EJB, JBI, SCA, and lately OSGi. Some of these, especially the earlier ones, imposed a great deal on the programming model, dictating what you could and couldn't do, and they often required complex packaging and deployment procedures. This left the everyday programmer with a lot of concepts to learn and master. In some cases, much more time was spent working around the restrictive programming and deployment models than on the business application itself.

Because of this growing complexity and the resulting frustrations, a simpler, more pragmatic programming model arose from the open source community: the Plain Old Java Object (POJO) model. Many open source projects have proven that the POJO programming model and a lightweight container meet the expectations of today's businesses. In fact, the simple programming model and lightweight container concept proved superior to the heavyweight and overly complex enterprise application and integration servers that were used before. This trend continues to this day, and we're seeing the rise of microservices and containerless deployments. We cover microservices in chapter 7, but first you need to learn more about Camel basics, including this chapter's topic of using Java beans with Camel.

So what about Camel? Well, Camel doesn't mandate using a specific component or programming model. It doesn't mandate a heavy specification that you must learn and understand to be productive. Camel doesn't require you to repackage any of your existing libraries or require you to use the Camel API to fulfill your integration needs. Camel is on the same page as the Spring Framework; both are lightweight containers favoring the POJO programming model. Camel recognizes the power of the POJO programming model and goes to great lengths to work with your beans. By using beans, you fulfill an important goal in the software industry: reducing coupling. Camel not only offers reduced coupling with beans, but you get the same loose coupling with Camel routes. For example, three teams can work simultaneously on their own sets of routes, which can easily be combined into one system.

This chapter starts by showing you how *not* to use beans with Camel, which will make it clearer how you *should* use beans. After that, you'll learn the theory behind the Service Activator EIP and dive inside Camel to see how this pattern is implemented. We then cover the bean-binding process, which gives you fine-grained control over binding information to the parameters on the invoked method from within Camel and the currently routed message. The chapter ends by explaining how to use beans as predicates and expressions in your route.

4.1 Using beans the hard way and the easy way

In this section, you'll walk through an example that shows how *not* to use beans with Camel—the hard way to use beans. Then you'll look at how to use beans the easy way.

Suppose you have an existing bean that offers an operation (a service) you need to use in your integration application. For example, `HelloBean` offers the `hello` method as its service:

```
public class HelloBean {
  public String hello(String name) {
    return "Hello " + name;
  }
}
```

Let’s look at various ways to use this bean in your application.

4.1.1 Invoking a bean from pure Java

By using a Camel `Processor`, you can invoke a bean from Java code, as in the following listing.

Listing 4.1 Using a Processor to invoke the hello method on the HelloBean

```
public class InvokeWithProcessorRoute extends RouteBuilder {

  public void configure() throws Exception {
    from("direct:hello")
    .process(new Processor() {  1
```

1

Uses a Processor

```
        public void process(Exchange exchange) throws Exception {
      String name = exchange.getIn().getBody(String.class);  2
```

2

Extracts input from Camel message

```
          HelloBean hello = new HelloBean();
      String answer = hello.hello(name);  3
```

3

Invokes HelloBean

```
      exchange.getOut().setBody(answer);  4
```

4

Response from HelloBean is set on OUT message

```
            }
        });
    }
}
```

Listing 4.1 shows a `RouteBuilder`, which defines the route. You use an inlined Camel `Processor`, which gives you the `process` method, in which you can work on the message with Java code 1. First, you must extract the message body from the input message 2, which is the parameter you'll use when you invoke the bean later. Then you need to instantiate the bean and invoke it 3. Finally, you must set the output from the bean on the output message 4.

Now that you've done it the hard way using the Java DSL, let's take a look at using XML DSL.

4.1.2 Invoking a bean defined in XML DSL

When using Spring XML or OSGi Blueprint as a bean container, the beans are defined using its XML files. Listings 4.2 and 4.3 show how to revise listing 4.1 to work with a Spring bean this way.

Listing 4.2 Setting up Spring to use a Camel route that uses HelloBean

```
<bean id="helloBean" class="camelinaction.HelloBean"/>    1
```

1
Defines HelloBean

```
<bean id="route" class="camelinaction.InvokeWithProcessorSpringRoute"/>

<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <routeBuilder ref="route"/>
</camelContext>
```

First you define `HelloBean` in the Spring XML file with the ID `helloBean` 1. You still want to use the Java DSL to build the route, so you need to declare a bean that contains the route. Finally, you define `CamelContext`, which is the way you get Spring and Camel to work together.

Listing 4.3 takes a closer look at the route.

Listing 4.3 A Camel route using a Processor to invoke HelloBean

```
public class InvokeWithProcessorSpringRoute extends RouteBuilder {
  @Autowired  1
```

1

Injects HelloBean

```
    private HelloBean hello;

    public void configure() throws Exception {
      from("direct:hello")
        .process(new Processor() {
          public void process(Exchange exchange) throws Exception {
            String name = exchange.getIn().getBody(String.class);
            String answer = hello.hello(name);  2
```

2

Invokes HelloBean

```
            exchange.getOut().setBody(answer);
          }
        });
    }
}
```

The route in listing 4.3 is nearly identical to the route in listing 4.1. The difference is that now the bean is dependency injected using the Spring `@Autowired` annotation 1, and instead of instantiating the bean, you use the injected bean directly 2.

You can try these examples on your own; they're in the chapter4/bean directory of the book's source code. Run Maven with these goals to try the last two examples:

```
mvn test -Dtest=InvokeWithProcessorTest
mvn test -Dtest=InvokeWithProcessorSpringTest
```

So far, you've seen two examples of using beans with a Camel route, and a bit of plumbing is required to get it all to work. As you've seen, it's hard to work with beans for the following reasons:

- You must use Java code to invoke the bean.
- You must use the Camel `Processor`, which clutters the route, making it harder to understand what happens (route logic is mixed in with implementation logic).
- You must extract data from the Camel message and pass it to the bean, and you must move any response from the bean back into the Camel message.
- You must instantiate the bean yourself, or have it dependency injected.

Now let's look at the easy way of doing it.

4.1.3 Using beans the easy way

Calling a bean in Camel is easy. The previous example in listing 4.3 can be reduced to a few lines of code, as shown in the following listing.

Listing 4.4 Camel route using bean to invoke HelloBean

```
public class InvokeWithBeanSpringRoute extends RouteBuilder {
  @Autowired    1
```

1

Injects HelloBean

```
    private HelloBean helloBean;

    public void configure() throws Exception {
      from("direct:hello")
        .bean(helloBean, "hello");    2
```

2

Invokes HelloBean

```
}
```

Now the route is much shorter—only two lines of code. You still let Spring dependency inject `HelloBean` 1. And this time, the Camel route doesn't use a `Processor` to invoke the bean but instead uses Camel's bean method 2:

```
.bean(helloBean, "hello")
```

The first parameter is the bean to call, and hello is the name of the method to call, which means Camel will invoke the `hello` method on the bean instance named `helloBean`.

That's a staggering reduction from seven lines of `Processor` code to one line with bean. And on top of that, the one code line is much easier to understand. It's all high-level abstraction, containing no low-level code details, which were required when using inlined `Processor` s.

In XML DSL, this is just as easy. The example can be written as follows:

```
<bean id="helloBean" class="camelinaction.HelloBean"/>
```

Injects HelloBean

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
  <bean ref="helloBean" method="hello"/>
```

Invokes HelloBean

```
  </route>
</camelContext>
```

PREFER TO SPECIFY THE METHOD NAME

If the bean has only a single method, you can omit specifying the method name, and the example can be reduced to this:

```
    from("direct:hello").bean("helloBean");
```

In XML DSL, the example looks like this:

```
    <bean ref="helloBean"/>
```

Omitting the method name is recommended only when the beans have a single method. When a bean has several methods, Camel has to find the most suitable method to invoke based on a series of factors. You'll dive into how this works in section 4.4.2. As a rule of thumb, it's better to specify the method name, which also lets humans who maintain or later have to modify your code understand exactly which method is being invoked.

We've provided these examples with the source code located in the chapter4/beans directory. You can run the example by using the following Maven goals:

```
mvn test -Dtest=InvokeWithBeanTest
mvn test -Dtest=InvokeWithBeanSpringTest
```

TIP In the Java DSL, you don't have to preregister the bean in the registry. Instead, you can provide the class name of the bean, and Camel will instantiate the bean on startup. The previous example could be written like this: `from ("direct:hello").bean(HelloBean.class);`.

Camel also allows you to call beans by using the bean component as an endpoint.

Using to instead of bean

When you start learning Camel and build your first set of Camel routes, you often start with a few EIP patterns and can get far with just using `from` and `to`. Camel also makes it easy to call Java beans by using the `to` style. The previous examples could have been written as follows:

```
from("direct:hello")
 .to("bean:helloBean?method=hello");
```

And in XML DSL:

```
<route>
   <from uri="direct:start"/>
   <to uri="bean:helloBean?method=hello"/>
 </route>
```

You may ask, when should you use `bean` and when should you use `to`? We, the authors, have no preference. Both are equally good to use. Sometimes you're most comfortable building routes that are only using `from` and `to`. And other times you use a lot more EIP patterns, and `bean` fits better. The only caveat with using `to` is that the bean must be specified by using a String value as either the fully qualified class name or the bean name. Using the fully qualified class name is more verbose (especially if you have long package names):

```
from("direct:hello")
  .to("bean:camelinaction.HelloBean?method=hello");
```

And in XML DSL:

```
<route>
   <from uri="direct:start"/>
   <to uri="bean:camelinaction.HelloBean?method=hello"/>
 </route>
```

Now let's look at how to work with beans in Camel from the EIP perspective.

4.2 Understanding the Service Activator pattern

The Service Activator pattern is an enterprise pattern described in Hohpe and Woolf's *Enterprise Integration Patterns* book. It describes a service that can be invoked easily from both messaging and non-messaging services. Figure 4.1 illustrates this principle.

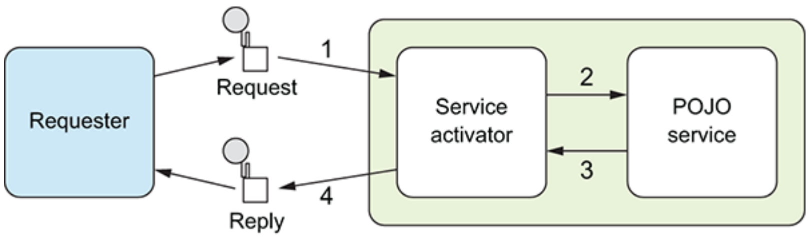

Figure 4.1 The service activator mediates between the requestor and the POJO service.

This service activator component invokes a service based on an incoming request and returns an outbound reply. The service activator acts as a mediator between the requester and the POJO service. The requester sends a request to the service activator 1, which is responsible for adapting the request to a format the POJO service understands (mediating) and passing the request on to the service 2. The POJO service then returns a reply to the service activator 3, which passes it back (requiring no translation on the way back) to the waiting requester 4.

As you can see in figure 4.1, the service is the POJO, and the service activator is something in Camel that can adapt the request and invoke the service. That something is the Camel `Bean` component, which eventually uses `org.apache.camel.component.bean.BeanProcessor` to do the work. You'll look at how this `BeanProcessor` works in section 4.4. You should regard the Camel `Bean` component as the Camel implementation of the Service Activator pattern.

Compare the Service Activator pattern in figure 4.1 to the Camel route example you looked at in section 4.1.3, as illustrated in figure 4.2.

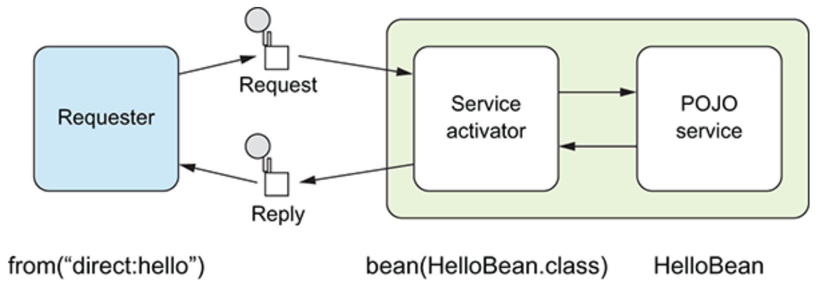

Figure 4.2 Relationship between a Camel route and the Service Activator EIP

Here you can see how the Camel route maps to the Service Activator EIP. The requester is the node that comes before the bean—it's the `from("direct:hello")` in our example. The service activator itself is the bean node, which is represented by the bean component in Camel. And the POJO service is the `HelloBean` bean itself.

You now know the theory behind how Camel works with beans—the Service Activator pattern. But before you can use a bean, you need to know where to look for it. This is where the registry comes into the picture. Let's look at how Camel works with various registries.

4.3 Using Camel's bean registries

When Camel works with beans, it looks them up in a registry to locate them. Camel's philosophy is to use the best of the available frameworks, so it uses a pluggable registry architecture to integrate them. Spring is one such framework, and figure 4.3 illustrates how the registry works.

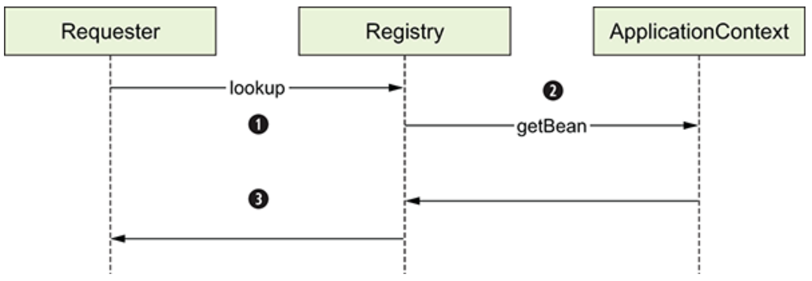

Figure 4.3 A requester looks up a bean using the Camel registry, which then uses the Spring `ApplicationContext` to determine where the bean resides.

The Camel registry is an abstraction that sits between the caller and the real registry. When a caller (requester) needs to look up a bean 1, it uses the Camel `Registry`. The Camel `Registry` then does the lookup via the real registry 2. The bean is then returned to the requester ❸ via the Camel `Registry`. This structure allows loose coupling but also a pluggable architecture that integrates with multiple registries. All the requester needs to know is how to interact with the Camel `Registry`.

The registry in Camel is merely a Service Provider Interface (SPI) defined in the `org.apache.camel.spi.Registry` interface, as follows:

```
Object lookupByName(String name);
<T> T lookupByNameAndType(String name, Class<T> type);
<T> Map<String, T> findByTypeWithName(Class<T> type);
<T> Set<T> findByType(Class<T> type);
```

You'll most often use one of the first two methods to look up a bean by its name. For example, to look up `HelloBean`, you'd do this:

```
HelloBean hello = (HelloBean) context.getRegistry()
                      .lookupByName("helloBean");
```

To get rid of that ugly typecast, you can use the second method instead:

```
HelloBean hello = context.getRegistry()
                      .lookupByNameAndType("helloBean", HelloBean.class)
```

NOTE The second method offers type-safe lookups because you provide the expected class as the second parameter. Under the hood, Camel uses its type-converter mechanism to convert the bean to the desired type, if necessary.

The last two methods, `findByTypeWithName` and `findByType`, are mostly used internally by Camel to support convention over configuration—they allow Camel to look up beans by type without knowing the bean name.

The registry itself is an abstraction and thus an interface. Table 4.1 lists the four implementations shipped with Camel.

Table 4.1 Registry implementations shipped in Camel

Registry	Description
`JndiRegistry`	An implementation that uses an existing Java Naming and Directory Interface (JNDI) registry to look up beans.
`SimpleRegistry`	An in-memory-only registry that uses `java.util.Map` to hold the entries.
`ApplicationContextRegistry`	An implementation that works with Spring to look up beans in the Spring `ApplicationContext`. This implementation is automatically used when you're using Camel in a Spring environment.
`OsgiServiceRegistry`	An implementation capable of looking up beans in the OSGi service reference registry. This implementation is automatically used when using Camel in an OSGi environment.
`BlueprintContainerRegistry`	An implementation that works with OSGi Blueprint to look up beans from the OSGi service registry as well as in the Blueprint container. This implementation is automatically used when you're using Camel in an OSGi Blueprint environment.
`CdiBeanRegistry`	An implementation that works with CDI to look up beans in the

Registry	Description
	CDI container. This implementation is used when using the camel-cdi component.

The following sections go over each of these six registries.

4.3.1 JndiRegistry

`JndiRegistry`, as its name indicates, integrates with a JNDI-based registry. It was the first registry that Camel integrated, so it's also the default registry if you create a Camel instance without supplying a specific registry, as this code shows:

```
CamelContext context = new DefaultCamelContext();
```

`JndiRegistry` (like `SimpleRegistry`) is often used for testing or when running Camel standalone. Many of the unit tests in Camel use `JndiRegistry` because they were created before `SimpleRegistry` was added to Camel.

NOTE `JndiRegistry` will be replaced with `SimpleRegistry` as the default registry in Camel 3.0 onward.

The source code for this book contains an example of using JndiRegistry that's identical to the next example using SimpleRegistry (shown in listing 4.5). We recommend that you read the next section and then compare the two examples.

You can try this test by going to the chapter4/bean directory and running this Maven goal:

```
mvn test -Dtest=JndiRegistryTest
```

Now let's look at the next registry: `SimpleRegistry`.

4.3.2 SimpleRegistry

`SimpleRegistry` is a `Map`-based registry that's used for testing or when running Camel standalone. For example, if you wanted to unit-test the `HelloBean` example, you could use `SimpleRegistry` to enlist `HelloBean` and refer to it from the route.

Listing 4.5 Using SimpleRegistry to unit-test a Camel route

```
public class SimpleRegistryTest extends TestCase {

  private CamelContext context;
  private ProducerTemplate template;

  protected void setUp() throws Exception {
    SimpleRegistry registry = new SimpleRegistry();
    registry.put("helloBean", new HelloBean()); ❶
```

❶ Registers HelloBean in SimpleRegistry

```
    context = new DefaultCamelContext(registry); ❷
```

❷ Uses SimpleRegistry with Camel

```
    template = context.createProducerTemplate();
    context.addRoutes(new RouteBuilder() {
      public void configure() throws Exception {
        from("direct:hello").bean("helloBean", "hello");
      }
    });
    context.start();
  }
```

```
    protected void tearDown() throws Exception {
    template.stop(); [3]
```

[3]

Cleans up resources after test

```
    context.stop(); [3]
    }

    public void testHello() throws Exception {
      Object reply = template.requestBody("direct:hello", "World");
      assertEquals("Hello World", reply);
    }
}
```

First you create an instance of `SimpleRegistry` and populate it with `HelloBean` under the `helloBean` name 1. Then, to use this registry with Camel, you pass the registry as a parameter to the `DefaultCamelContext` constructor 2. To aid when testing, you create `ProducerTemplate`, which makes it simple to send messages to Camel, as you can see in the test method. Finally, when the test is done, you clean up the resources by stopping `ProducerTemplate` and Camel 3. In the route, you use the `bean` method to invoke `HelloBean` by the `helloBean` name you gave it when it was enlisted in the registry 1.

You can try this test by going to the chapter4/bean directory and running this Maven goal:

```
mvn test -Dtest=SimpleRegistryTest
```

The next registry is for when you use Spring together with Camel.

4.3.3 ApplicationContextRegistry

`ApplicationContextRegistry` is the default registry when Camel is used in a Spring environment such as Spring Boot or from a Spring XML file. When we say *using Camel in Spring XML*, we mean the following, as this snippet illustrates:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <bean ref="helloBean" method="hello"/>
  </route>
</camelContext>
```

Defining Camel by using the `<camelContext>` tag automatically lets Camel use `ApplicationContextRegistry`. This registry allows you to define beans in Spring XML files as you would normally do when using Spring. For example, you could define the `helloBean` bean as follows:

```
<bean id="helloBean" class="camelinaction.HelloBean"/>
```

It can hardly be simpler than that. When you use Camel with Spring, you can keep on using Spring beans as you would normally, and Camel will use those beans seamlessly without any configuration.

The next two registries apply when you use Camel with OSGi.

4.3.4 OsgiServiceRegistry and BlueprintContainerRegistry

When Camel is used in an OSGi environment, Camel uses a two-step lookup process. First, it looks up whether a service with the name exists in the OSGi service registry. If not, Camel falls back and looks up the name in `BlueprintContainerRegistry` when using OSGi Blueprint.

POPULAR OSGI PLATFORMS WITH CAMEL

The most popular OSGi platform to use with Apache Camel is Apache Karaf or Apache ServiceMix. You can also find commercial platforms from vendors such as Red Hat and Talend.

Suppose you want to allow other bundles in OSGi to reuse the following bean:

```
<bean id="helloBean" class="camelinaction.HelloBean"/>
```

You can do that by exporting the bean as an OSGi service that will enlist the bean into the OSGi service registry, as follows:

```
<osgi:service id="helloService" interface="camelinaction.HelloBean"
                ref="helloBean"/>
```

Now another bundle such as a Camel application can reuse the bean, by referring to the service:

```
<osgi:reference id="helloService" interface="camelinaction.HelloBean"/>
```

To call the bean from the Camel route is easily done by using the bean component as if the bean is a local `<bean>` element in the same OSGi Blueprint XML file:

```
<camelContext xmlns="http://camel.apache.org/schema/blueprint">
  <route>
    <from uri="direct:start"/>
    <bean ref="helloService" method="hello"/>
  </route>
</camelContext>
```

All you have to remember is the name with which the bean was exported. Camel will look it up in the OSGi service registry and the Blueprint bean container for you. This is convention over configuration.

The last registry is for using Camel with CDI.

4.3.5 CdiBeanRegistry

Contexts and Dependency Injection (*CDI*) is a Java specification that standardizes how Java developers can integrate Java beans in a loosely coupled way. For Camel developers, it means you can use the CDI annotations to inject Java beans, Camel endpoints, and other services.

CDI CONTAINERS

CDI is a Java specification that's part of Java EE, and therefore available in Java EE application servers such as Apache TomEE, WildFly, and in commercial offerings as well. But CDI is lightweight, and you can run in a standalone CDI container such as Apache OpenWebBeans or JBoss Weld. You'll encounter CDI again in chapters 7, 9, and 15 as you learn about CDI with microservices, testing, and running Camel.

`CdiBeanRegistry` is the default registry when using Camel with CDI. The registry is automatically created in the default constructor of the `CdiCamelContext`, as shown here:

```
public class CdiCamelContext extends DefaultCamelContext {

  public CdiCamelContext() {
    super(new CdiBeanRegistry());
```

Using CdiBeanRegistry as bean registry when using Camel with CDI—as a Camel end user, you don't need to configure this, because Camel with CDI is automatically configured

```
      setInjector(new CdiInjector(getInjector()));
    }
```

In CDI, beans can be defined by using CDI annotations. For example, to define a singleton bean with the name `helloBean`, you could do so as shown in the following listing.

Listing 4.6 A simple bean to print "Hello World" using CDI annotations

```
@Singleton  ❶
```

❶ The bean is singleton scoped

```
@Named("helloBean") ❷
```

❷ The bean is registered with the name helloBean

```
public class HelloBean {
  private int counter;

  public String hello() {
    return "Hello " + ++counter + " times";
  }
}
```

To use the singleton bean ❶ from a Camel route, you can use a bean reference to look up the bean by its name 2. The Camel route could also be coded with CDI, as shown in the following listing.

Listing 4.7 A Camel route using CDI to use the bean from listing 4.6

```
@Singleton ❶
```

❶ Defines the class as a CDI Singleton bean

```
public class HelloRoute extends RouteBuilder {

  @EndpointInject(uri = "timer:foo?period=5s")
 private Endpoint input; ❷
```

❷ Injects Camel endpoint that's used in the route

```

  @EndpointInject(uri = "log:output")
 private Endpoint output; ❸
```

3

Injects Camel endpoint that’s used in the route

```
  @Override
  public void configure() throws Exception {
    from(input)
      .bean("helloBean", "hello") 4
```

4

Invokes the bean by referring to the bean name

```
        .to(output);
    }
}
```

The route builder class has been annotated with `@Singleton` 1, which lets Camel automatically discover the route when starting. Then you inject endpoints by using `org.apache.camel.EndpointInject` ❷ 3. This isn’t needed, because you could have used the endpoint URIs directly in the Java DSL route, which could have been written as follows:

```
from("timer:foo?period=5s")
  .bean("helloBean", "hello")
  .to("log:output");
```

But we’ve chosen to show a practice often used with CDI, which is dependency injection, and why we inject the endpoints also. To call the bean, you can use `bean("helloBean", "hello")` ❹ to refer to the bean by its name and the name of the method to invoke. But you also could have injected the bean by using CDI. This can be done using `@Inject` and `@Named`, as shown in the following code snippet:

```
@Inject @Named("helloBean")
private HelloBean helloBean;

public void configure() throws Exception {
  from(input)
    .bean(helloBean)
    .to(output);
}
```

This example is provided in the book's source code in chapter4/cdi-beans. You can try the example with the following Maven goal:

```
mvn clean install camel:run
```

This concludes your tour of registries. Next we'll focus on how Camel selects which method to invoke on a given bean.

4.4 Selecting bean methods

You've seen how Camel works with beans from the route perspective. Now it's time to dig down and see the moving parts in action. You first need to understand the mechanism Camel uses to select the method to invoke.

Remember, Camel acts as a service activator using the bean component, which sits between the caller and the bean. At compile time, there are no direct bindings, and the JVM can't link the caller to the bean; Camel must resolve this at runtime.

Figure 4.4 illustrates how the bean component uses the registry to look up the bean to invoke.

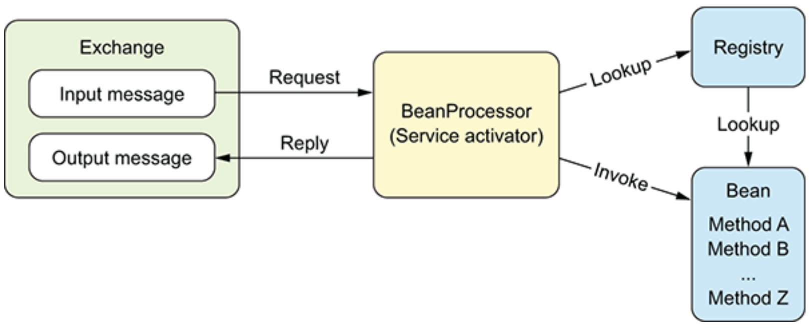

Figure 4.4 To invoke a bean in Camel, the bean component (`BeanProcessor`) looks it up in the registry, selects and adapts a method, invokes it, and passes the returned value as the reply to the Camel exchange.

At runtime, a Camel exchange is routed, and at a given point in the route, it reaches the bean component. The bean component (`BeanProcessor`) then processes the exchange, performing these general steps:

1. Looks up the bean in the registry
2. Selects the method to invoke on the bean
3. Binds to the parameters of the selected method (for example, using the body of the input message as a parameter; this is covered in detail in section 4.5)
4. Invokes the method
5. Handles any invocation errors that occur (any exceptions thrown from the bean will be set on the Camel exchange for further error handling)
6. Sets the method's reply (if there is one) as the body on the output message on the Camel exchange

Section 4.3 covered how registry lookups are done. The next two steps (steps 2 and 3 in the preceding list) are more complex, and we cover them in the remainder of this chapter. The reason this is more complex in Camel is that Camel has to compute which bean and method to invoke at runtime, whereas Java code is linked at compile time.

WHY DOES CAMEL NEED TO SELECT A METHOD?

Why is there more than one possible method name when you invoke a method? The answer is that beans can have overloaded methods, and in some cases the method name isn't specified either, which means Camel has to pick among all methods on the bean.

Suppose you have the following methods:

```
String echo(String s);
int echo(int number);
void doSomething(String something);
```

Camel has three methods to choose from. If you explicitly tell Camel to use the `echo` method, you're still left with two methods to choose from. We'll look at how Camel resolves this dilemma.

We'll first take a look at the algorithm Camel uses to select the method. Then we'll look at a couple of examples and see what could go wrong and how to avoid problems.

4.4.1 How Camel selects bean methods

Unlike at compile time, when the Java compiler can link method invocations together, the bean component has to select the method to invoke at runtime.

Suppose you have the following class:

```
public class EchoBean {
  String echo(String name) {
    return name + " " + name;
  }
}
```

At compile time, you can express your code to invoke the `echo` method like this:

```
EchoBean echo = new EchoBean();
String reply = echo.echo("Camel");
```

This ensures that the `echo` method is invoked at runtime. On the other hand, suppose you use the `EchoBean` in Camel in a route as follows:

```
from("direct:start")
  .bean(EchoBean.class, "echo")
  .to("log:reply");
```

When the compiler compiles this code, it can’t see that you want to invoke the `echo` method on the `EchoBean`. From the compiler’s point of view, `EchoBean.class` and `echo` are parameters to the bean method. All the compiler can check is that the `EchoBean` class exists; if you misspelled the method name, perhaps typing `ekko`, the compiler couldn’t catch this mistake. The mistake would end up being caught at runtime, when the bean component would throw a `MethodNotFoundException` stating that the method named `ekko` doesn’t exist.

Camel also allows you not to explicitly name a method. For example, you could write the previous route as follows:

```
from("direct:start")
  .bean(EchoBean.class)
  .to("log:reply");
```

Regardless of whether the method name is explicitly given, Camel has to compute which method to invoke. Let's look at how Camel chooses.

4.4.2 Camel's method-selection algorithm

The bean component uses a complex algorithm to select which method to invoke on a bean. You don't need to understand or remember every step in this algorithm; we simply want to outline what goes on inside Camel to make working with beans as simple as possible for you.

Figure 4.5 shows the first part of this algorithm, which is continued in figure 4.6.

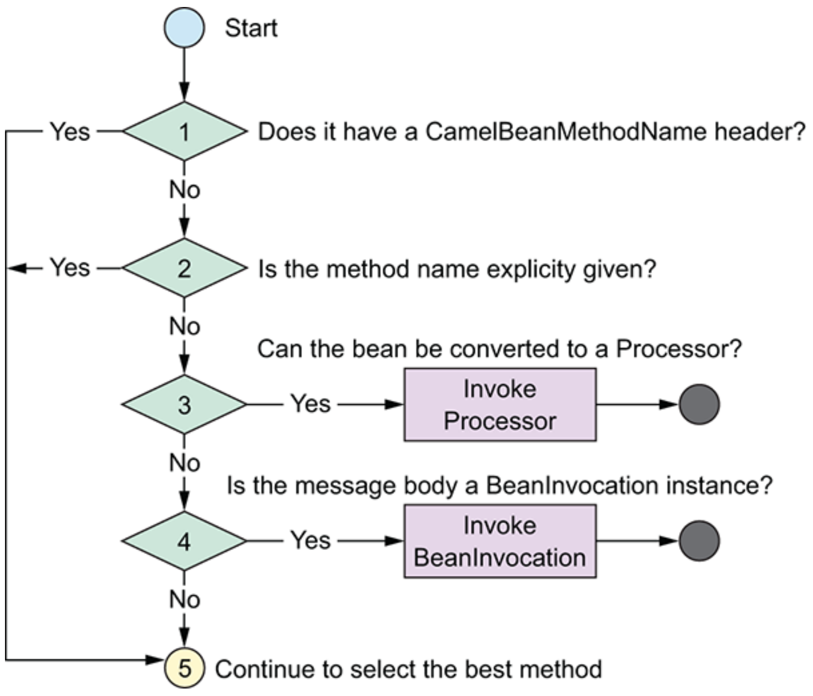
Start
Yes
1
Does it have a CamelBeanMethodName header?
No
Yes
2
Is the method name explicity given?
No
Can the bean be converted to a Processor?
3
Yes
Invoke Processor
No
Is the message body a BeanInvocation instance?
4
Yes
Invoke BeanInvocation
No
5
Continue to select the best method

Figure 4.5 How Camel selects which method to invoke (part 1, continued in figure 4.6)

Here's how the algorithm selects the method to invoke:

1. If the Camel message contains a header with the key `CamelBeanMethodName`, its value is used as the explicit method name. Go to step 5.
2. If a method is explicitly defined, Camel uses it, as we mentioned at the start of this section. Go to step 5.

3. If the bean can be converted to a `Processor` using the Camel type-converter mechanism, the `Processor` is used to process the message. This may seem odd, but it allows Camel to turn any bean into a message-driven bean equivalent. For example, with this technique, Camel allows any `javax.jms.MessageListener` bean to be invoked directly by Camel without any integration glue. This method is rarely used by end users of Camel, but it can be a useful trick.
4. If the body of the Camel message can be converted into `org.apache.camel.component.bean.BeanInvocation`, that's used to invoke the method and pass the arguments to the bean. This is in use only when using the Camel bean proxy, which is rarely used and not covered in this book. (You can find coverage of this subject in the first edition of the book in chapter 14.)
5. Continue with the second part of the algorithm, shown in figure 4.6.

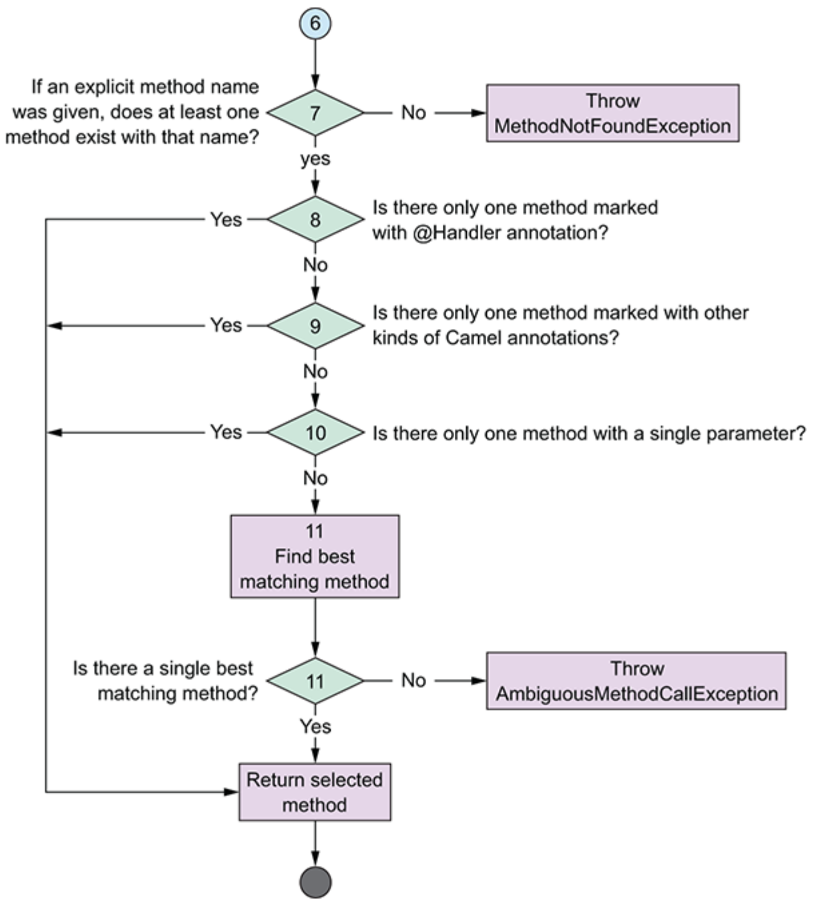
6
If an explicit method name was given, does at least one method exist with that name?
7
No
Throw MethodNotFoundException
yes
Yes
8
Is there only one method marked with @Handler annotation?
No
Yes
9
Is there only one method marked with other kinds of Camel annotations?
No
Yes
10
Is there only one method with a single parameter?
No
11 Find best matching method
Is there a single best matching method?
11
No
Throw AmbiguousMethodCallException
Yes
Return selected method

Figure 4.6 How Camel selects which method to invoke (part 2, continued from figure 4.5)

Figure 4.6 is a bit more complex, but its main goal is to narrow the number of possible methods and select a method if one stands out. Don't worry if you don't entirely understand the algorithm; you'll look at a couple of examples shortly that should make it much clearer.

Let’s continue with the algorithm and cover the last steps:

1. If a method name was given and no methods exist with that name, a `MethodNotFoundException` exception is thrown.
2. If only a single method has been marked with the `@Handler` annotation, it’s selected.
3. If only a single method uses any of the other Camel bean parameter-binding annotations, such as `@Body`, `@Header`, and so on, it’s selected. (You’ll look at how Camel binds to method parameters using annotations in section 4.4.3.)
4. If, among all the methods on the bean, there’s only one method with exactly one parameter, that method is selected. For example, this would be the situation for the `EchoBean` bean you looked at in section 4.4.1, which has only the `echo` method with exactly one parameter. Single-parameter methods are preferred because they map easily with the payload from the Camel exchange.
5. Now the computation gets a bit complex. There are multiple candidate methods, and Camel must determine whether a single method stands out as the best fit. The strategy is to go over the candidate methods and filter out methods that don’t fit. Camel does this by trying to match the first parameter of the candidate method; if the parameter isn’t the same type and it’s not possible to coerce the types, the method is filtered out. If a method name includes numerous parameter values, these values are also used during filtering. The values are matched with the pairing parameter type on the candidate methods. In the end, if only a single method is left, that method is selected. Because this logic is elaborate, we cover it in more detail in the following sections.
6. If Camel can’t select a method, an `AmbiguousMethodCallException` exception is thrown with a list of ambiguous methods.

Clearly, Camel goes through a lot to select the method to invoke on your bean. Over time you’ll learn to appreciate all this—it’s convention over configuration to the fullest.

NOTE The algorithm laid out in this book is based on Apache Camel version 2.20. This method-selection algorithm may change in the future to accommodate new features.

Now it's time to look at applying this algorithm in practice.

4.4.3 Some method-selection examples

To see how this algorithm works, you'll use the `EchoBean` from section 4.4.1 as an example. This time, you'll add another method to it—the `bar` method—to better understand what happens when you have multiple candidate methods:

```
public class EchoBean {

  public String echo(String echo) {
    return echo + " " + echo;
  }

  public String bar() {
    return "bar";
  }
}
```

And you'll start with this route:

```
from("direct:start")
  .bean(EchoBean.class)
  .to("log:reply");
```

If you send the `String` message `Camel` to the Camel route, the reply logger will surely output `Camel` `Camel` as expected, since the `EchoBean` will duplicate the input as its response. Although `EchoBean` has two methods, `echo` and `bar`, only the `echo` method has a single parameter. This is what step 9 in figure 4.6 ensures: Camel will pick the method with a single parameter if there's only one of them.

To make the example more challenging, let's change the `bar` method as follows:

```
public String bar(String name) {
  return "bar " + name;
}
```

What do you expect will happen now? You have two identical method signatures with a single method parameter. In this case, Camel can't pick one over the other, so it throws an `AmbiguousMethodCallException` exception, according to step 11 in figure 4.6.

How can you resolve this? One solution is to provide the method name in the route, such as specifying the `bar` method:

```
from("direct:start")
  .bean(EchoBean.class, "bar")
  .to("log:reply");
```

But there's another solution that doesn't involve specifying the method name in the route: you can use the `@Handler` annotation to select the method. This solution is implemented in step 7 of figure 4.6. `@Handler` is a Camel-specific annotation that you can add to a method. It simply tells Camel to use this method by default:

```
@Handler
public String bar(String name) {
  return "bar " + name;
}
```

Now `AmbiguousMethodCallException` won't be thrown because the `@Handler` annotation tells Camel to select the `bar` method.

TIP It's a good idea either to declare the method name in the route or to use the `@Handler` annotation. This ensures that Camel picks the method you want, and you won't be surprised if Camel chooses another method.

Suppose you change `EchoBean` to include two methods with different parameter types:

```
public class EchoBean {

  public String echo(String echo) {
    return echo + " " + echo;
  }

  public Integer doubleUp(Integer num) {
    return num.intValue() * num.intValue();
  }
}
```

The `echo` method works with a `String`, and the `doubleUp` method with an `Integer`. If you don't specify the method name, the bean component will have to choose between these two methods at runtime.

Step 10 in figure 4.6 allows Camel to be smart about deciding which method stands out. It does so by inspecting the message payloads of two or more candidate methods and comparing those with the message body type, checking whether there's an exact type match in any of the methods.

Suppose you send a message to the route that contains a `String` body with the word `Camel`. It's not hard to guess that Camel will pick the `echo` method, because it works with `String`. On the other hand, if you send in a message with the `Integer` value of `5`, Camel will select the `doubleUp` method, because it uses the `Integer` type.

Despite this, things can still go wrong, so let's go over a couple common situations.

4.4.4 Potential method-selection problems

A few things can go wrong when invoking beans at runtime:

- *Specified method not found*—If Camel can't find any method with the specified name, a `MethodNotFoundException` exception is thrown. This happens only when you've explicitly specified the method name.
- *Ambiguous method*—If Camel can't single out a method to call, an `AmbiguousMethodCallException` exception is thrown with a list of the ambiguous methods. This can happen even when an explicit method name is defined, because the method could potentially be overloaded, which means the bean would have multiple methods with the same name; only the number of parameters would vary.
- *Type conversion failure*—Before Camel invokes the selected method, it must convert the message payload to the parameter type required by

the method. If this fails, a `NoTypeConversionAvailableException` exception is thrown.

Let’s take a look at examples of each of these three situations using the following `EchoBean`:

```
public class EchoBean {

  public String echo(String name) {
    return name + name;
  }

  public String hello(String name) {
    return "Hello " + name;
  }
}
```

First, you could specify a method that doesn’t exist by doing this:

```
.bean("echoBean", "foo")
```

And in XML DSL:

```
<bean ref="echoBean" method="foo"/>
```

Here you try to invoke the `foo` method, but no such method exists, so Camel throws a `MethodNotFoundException` exception.

On the other hand, you could omit specifying the method name:

```
.bean("echoBean")
```

And in XML DSL:

```
<bean ref="echoBean"/>
```

In this case, Camel can’t single out a method to use because both the `echo` and `hello` methods are ambiguous. When this happens, Camel throws an `AmbiguousMethodCallException` exception containing a list of the ambiguous methods.

The last situation that could happen is when the message contains a body that can't be converted to the type required by the method. Suppose you have the following `OrderServiceBean` class:

```
public class OrderServiceBean {
  public String handleXML(Document xml) {
    ...
  }
}
```

And suppose you need to use that bean in this route:

```
from("jms:queue:orders")
  .bean("orderService", "handleXML")
  .to("jms:queue:handledOrders");
```

The `handleXML` method requires a parameter to be of type `org.w3c.dom.Document`, which is an XML type, but what if the JMS queue contains a `javax.jms.TextMessage` not containing any XML data, but just a plain-text message, such as `Camel rocks`? At runtime, you'll get the following stack trace:

```
org.apache.camel.TypeConversionException: Error during type conversionfr
type: java.lang.String to the required type: org.w3c.dom.Documentwith va
Camel rocks due org.xml.sax.SAXParseException; lineNumber: 1;columnNumbe
1; Content is not allowed in prolog.
     at org.apache.camel.impl.converter.BaseTypeConverterRegistry.create
ConversionException(BaseTypeConverterRegistry.java:571)
     at org.apache.camel.impl.converter.BaseTypeConverterRegistry.conver
(BaseTypeConverterRegistry.java:129)
     at org.apache.camel.impl.converter.BaseTypeConverterRegistry.conver
(BaseTypeConverterRegistry.java:100)
Caused by: org.xml.sax.SAXParseException; lineNumber: 1; columnNumber: 
1;Content is not allowed in prolog.
     at com.sun.org.apache.xerces.internal.parsers.DOMParser.parse
(DOMParser.java:257)
     at com.sun.org.apache.xerces.internal.jaxp.DocumentBuilderImpl.pars
(DocumentBuilderImpl.java:348)
     at org.apache.camel.converter.jaxp.XmlConverter.toDOMDocument
(XmlConverter.java:902)
```

Camel tries to convert the `javax.jms.TextMessage` to an `org.w3c.dom.Document` type, but it fails. In this situation, Camel wraps the error and throws it as a `TypeConversionException`

exception.
By looking further into this stack trace, you may notice that the cause of this problem is that the XML parser couldn't parse the data to XML. It reports `Content is not allowed in prolog`, which is a common error indicating that the XML declaration (`<?xml version="1.0"?>`) is missing, and therefore a strong indicator that the payload isn't XML based.

NOTE You may wonder what would happen if such a situation occurred. In this case, the Camel error-handling system would kick in and handle it. Error handling is covered thoroughly in chapter 11.

Before we wrap up this section, we'd like you to know about one more functionality. It covers use cases in which your beans have overloaded methods (methods using the same name, but with different parameter types, for example) and how Camel works in those situations.

4.4.5 Method selection using type matching

The previous algorithm in step 10 of figure 4.6 also allows users to filter methods based on parameter-type matching. For example, suppose the following class has multiple methods to handle the order:

```
public class OrderServiceBean {

  public String handleXML(Document xml) {
    ...
  }

  public String handleXML(String xml) {
    ....
  }
}
```

And you need to use that bean in this route:

```
from("jms:queue:orders")
  .bean("orderService", "handleXML")
  .to("jms:queue:handledOrders");
```

`OrderServiceBean` has two methods named `handleXML`, each one accepting a different parameter type as the input. What happens at runtime depends on whether the bean component is able to find a single suitable method to invoke, according to the algorithm listed in figures 4.5 and 4.6. To decide that Camel will be based on the message body class type, determine which one is the best candidate (if possible). If the message body is, for example, a `Document` or `String` type, there's a direct match with the types on those methods and the appropriate method is invoked. But if the Camel message body is of any other type, such as `java.io.File`, Camel will look up whether there are type converters that can convert from `java.io.File` and respectively to `org.w3c.Document` and `java.lang.String`. If only one type conversion is possible, the appropriate method is chosen. In any other case, `AmbiguousMethodCallException` is thrown.

In those situations, you can assist Camel by explicitly defining which of the two methods to call by specifying which type to use, as shown here:

```
from("jms:queue:orders")
  .bean("orderService", "handleXML(org.w3c.Document)")
  .to("jms:queue:handledOrders");
```

You'd need to specify the class type by using its fully qualified name. But for common types such as `Boolean`, `Integer`, and `String`, you can omit the package name, and use just `String` as the `java.lang.String` type:

```
from("jms:queue:orders")
  .bean("orderService", "handleXML(String)")
  .to("jms:queue:handledOrders");
```

Over the years with Camel, we haven't often seen the need for this. If possible, we suggest using unique method names instead, which makes it easier for both Camel and end users to know exactly which methods Camel will use.

That's all you need to know about how Camel selects methods at runtime. Now you need to look at the bean parameter-binding process, which happens after Camel has selected the method.

4.5 Performing bean parameter binding

The preceding section covered the process that selects which method to invoke on a bean. This section covers what happens next—how Camel adapts to the parameters on the method signature. Any bean method can have multiple parameters, and Camel must somehow pass in meaningful values. This process is known as *bean parameter binding*.

You've already seen parameter binding in action in the many examples so far in this chapter. What those examples have in common is using a single parameter to which Camel bound the input message body. Figure 4.7 illustrates this, using `EchoBean` as an example.

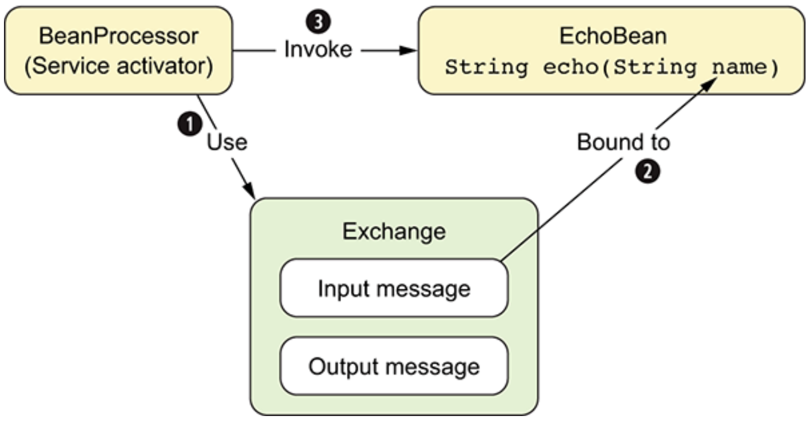

Figure 4.7 How a bean component binds the input message to the first parameter of the method being invoked

The bean component (`BeanProcessor)` uses the input message ❶ to bind its body to the first parameter of the method ❷, which happens to be the `String` `name` parameter. Camel does this by creating an expression that type-converts the input message body to the `String` type. This ensures that when Camel invokes the `echo` method ❸, the parameter matches the expected type.

This is important to understand, because most beans have methods with a single parameter. The first parameter is expected to be the input message body, and Camel will automatically convert the body to the same type as the parameter.

What happens when a method has multiple parameters? That's what we'll look at in the remainder of this section.

4.5.1 Binding with multiple parameters

Figure 4.8 illustrates the principle of bean parameter binding when multiple parameters are used.

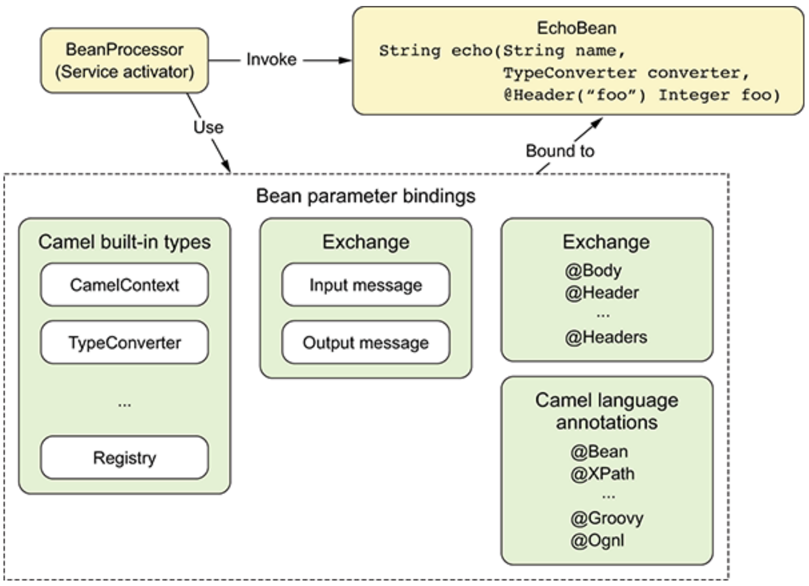

Figure 4.8 Parameter binding with multiple parameters involves a lot more options than with single parameters.

At first, figure 4.8 may seem overwhelming. Many new types come into play when you deal with multiple parameters. The big box entitled *Bean parameter bindings* contains the following four boxes:

- *Camel built-in types*—Camel provides special bindings for a series of Camel concepts. We cover them in section 4.5.2.
- *Exchange*—This is the Camel exchange, which allows binding to the input message, such as its body and headers. The Camel exchange is the source of the values that must be bound to the method parameters. It's covered in the sections to come.
- *Camel annotations*—When dealing with multiple parameters, you use annotations to distinguish them. This is covered in section 4.5.3.
- *Camel language annotations*—This is a less commonly used feature that allows you to bind parameters to languages. It's ideal when working with XML messages that allow you to bind parameters to XPath expressions. This is covered in section 4.5.4.

In addition, you can specify the parameter binding by using the method-name signature that resembles how you call methods in Java source code. We'll look at this in section 4.5.5.

WORKING WITH MULTIPLE PARAMETERS

Using multiple parameters is more complex than using single parameters. It's generally a good idea to follow these rules of thumb:

- Use the first parameter for the message body, which may or may not use the `@Body` annotation.
- Use either a built-in type or add Camel annotations for subsequent parameters.
- When having more than two parameters, consider specifying the binding in the method-name signature, which makes it clear for humans and Camel how each parameter should be mapped. We cover this in section 4.5.5.

In our experience, it becomes complicated when multiple parameters don't follow these guidelines, but Camel will make its best attempt to adapt the parameters to the method signature.

Let's start by looking at using the Camel built-in types.

4.5.2 Binding using built-in types

Camel provides a set of fixed types that are always bound. All you have to do is declare a parameter of one of the types listed in table 4.2.

Table 4.2 Parameter types that Camel automatically binds

Type	Description
`Exchange`	The Camel exchange. This contains the values that will be bound to the method parameters.
`Message`	The Camel input message. It contains the body that's often bound to the first method parameter.
`CamelContext`	The Camel context. This can be used in special circumstances when you need access to all of Camel's moving parts.
`TypeConverter`	The Camel type-converter mechanism. This can be used when you need to convert types. Chapter 3 covered the type-converter mechanism.
`Registry`	The bean registry. This allows you to look up beans in the registry.
`Exception`	An exception, if one was thrown. Camel will bind to this only if the exchange has failed and contains an exception. This allows you to use beans to handle errors. Chapter 11 covers error handling; you can find an example of using a custom bean to handle failures in section 11.4.4.

Let's look at a couple of examples using the types from table 4.2. First, suppose you add a second parameter that's one of the built-in types to the `echo` method:

```
public string echo(String echo, CamelContext context)
```

In this example, you bind `CamelContext`, which gives you access to all the moving parts of Camel.

Or you could bind the registry, in case you need to look up some beans:

```
public string echo(String echo, Registry registry) {
   OtherBean other = registry.lookup("other", OtherBean.class);
   ...
}
```

You aren't restricted to having only one additional parameter; you can have as many as you like. For example, you could bind both the `CamelContext` and the registry:

```
public string echo(String echo, CamelContext context, Registry registry)
```

So far, you've always bound to the message body. How would you bind to a message header? The next section explains that.

4.5.3 Binding using Camel annotations

Camel provides a range of annotations to help bind from the exchange to bean parameters. You should use these annotations when you want more control over the bindings. For example, without these annotations, Camel will always try to bind the method body to the first parameter, but with the `@Body` annotation, you can bind the body to any parameter in the method.

Suppose you have the following bean method:

```
public String orderStatus(Integer customerId, Integer orderId)
```

And you have a Camel message that contains the following data:

- Body, with the order ID, as a `String` type
- Header with the customer ID as an `Integer` type

With the help of Camel annotations, you can bind the exchange to the method signature as follows:

```
public String orderStatus(@Header("customerId") Integer customerId,
                          @Body Integer orderId)
```

Notice that you can use the `@Header` annotation to bind the message header to the first parameter and `@Body` to bind the message body to the second parameter.

Table 4.3 lists all the Camel parameter-binding annotations.

Table 4.3 Camel parameter-binding annotations

Annotation	Description
@Body	Binds the parameter to the message body.
@Header(name)	Binds the parameter to the message header with the given name.
@Headers	Binds the parameter to all the input headers. The parameter must be a `java.util.Map` type.
@ExchangeProperty(name)	Binds the parameter to the exchange property with the given name.
@ExchangeProperties	Binds the parameter to all the exchange properties. The parameter must be a `java.util.Map` type.
@ExchangeException	Binds the parameter to the exception set on the exchange.
@Attachments	Binds the parameter to the message attachments. The parameter must be a `java.util.Map` type.

You've already seen the first type in action, so let's try a couple examples with the other annotations. For example, you could use `@Header` to bind a named header to a parameter, and this can be done more than once, as shown here:

```
public String orderStatus(@Body Integer orderId,
                          @Header("customerId") Integer customerId,
                          @Header("customerType") Integer customerType)
   ...
}
```

If you have many headers, it may be easier to use `@Headers` to bind all the headers to a `Map` type:

```
public String orderStatus(@Body Integer orderId, @Headers Map headers) {
    Integer customerId = (Integer) headers.get("customerId");
    String customerType = (String) headers.get("customerType");
   ...
}
```

Finally, let's look at Camel's language annotations, which bind parameters to a language.

4.5.4 Binding using Camel language annotations

Camel provides additional annotations that allow you to use other languages as parameters. One of the most common languages to use is XPath, which allows you to evaluate XPath expressions on the message body as XML documents. For example, suppose the message contains the following XML document:

```
<order customerId="123">
  <status>in progress</status>
</order>
```

By using XPath expressions, you can extract parts of the document and bind them to parameters, like this:

```
public void updateStatus(@XPath("/order/@customerId") Integer customerId
                         @XPath("/order/status/text()") String sta
```

You can bind as many parameters as you like—the preceding example binds two parameters by using the `@XPath` annotations. You can also mix and match annotations, so you can use `@XPath` for one parameter and `@Header` for another.

Table 4.4 lists the most commonly used language annotations provided in Camel.

Table 4.4 Camel's language-based bean binding annotations

Annotation	Description	Maven dependency
`@Bean`	Invokes a method on a bean	`camel-core`
`@Constant`	Evaluates as a constant value	`camel-core`
`@Groovy`	Evaluates a Groovy script	`camel-script`
`@JavaScript`	Evaluates a JavaScript script	`camel-script`
`@JsonPath`	Evaluates a JsonPath expression	`camel-jsonpath`
`@MVEL`	Evaluates a MVEL script	`camel-mvel`
`@Simple`	Evaluates a Simple expression. (Simple is a built-in language provided with Camel; see appendix A for more details.)	`camel-core`
`@SpEL`	Evaluates a Spring expression	`camel-spring`
`@XPath`	Evaluates an XPath expression	`camel-core`
`@XQuery`	Evaluates an XQuery expression	`camel-saxon`

It may seem a bit magical that you can use an `@Bean` annotation when invoking a method, because the `@Bean` annotation itself also invokes a method. Let's try an example.

Suppose you already have a service that must be used to stamp unique order IDs on incoming orders. The service is implemented in the following listing.

Listing 4.8 A service that stamps an order ID on an XML document

```
public Document handleIncomingOrder(Document xml, int customerId,
                                              int orderId) {
  Attr attr = xml.createAttribute("orderId");    ❶
```

❶ Creates orderId attribute

```
    attr.setValue("" + orderId);
    Node node = xml.getElementsByTagName("order").item(0);
  node.getAttributes().setNamedItem(attr);    ❷
```

❷ Adds orderId attribute to order node

```
    return xml;
  }
```

As you can see, the service creates a new XML attribute with the value of the given order ID ❶. Then it inserts this attribute in the XML document ❷ using the rather clumsy XML API from Java ❷.

To generate the unique order ID, you have the following class:

```
public final class GuidGenerator {
  public static int generate() {
    Random ran = new Random();
    return ran.nextInt(10000000);
    }
  }
```

(In a real system, you'd generate unique order IDs based on another scheme.)

In Camel, you have the following route that listens for new order files and invokes the service before sending the orders to a JMS destination for further processing:

```
<bean id="xmlOrderService" class="camelinaction.XmlOrderService"/>

<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="file://riderautoparts/order/inbox"/>
    <bean ref="xmlOrderService"/>
    <to uri="jms:queue:order"/>
  </route>
</camelContext>
```

What's missing is the step that generates a unique ID and provides that ID in the `handleIncomingOrder` method (shown in listing 4.8). To do this, you need to declare a bean in the Spring XML file with the ID generator, as follows:

```
<bean id="guid" class="camelinaction.GuidGenerator"/>
```

Now you're ready to connect the last pieces of the puzzle. You need to tell Camel that it should invoke the `generate` method on the `guid` bean when it invokes the `handleIncomingOrder` method from listing 4.8. To do this, you use the `@Bean` annotation and change the method signature (highlighted in bold) to the following:

```
public Document handleIncomingOrder(@Body Document xml,
                    @XPath("/order/@customerId") int customerId,
                    @Bean(ref = "guid", method="generate") int orderId);
```

We've prepared a unit test you can use to run this example. Use the following Maven goal from the chapter4/bean directory:

```
mvn test -Dtest=XmlOrderTest
```

When it's running, you should see two log lines that output the XML order before and after the service has stamped the order ID. Here's an example:

```
2017-05-17 16:18:58,485 [: FileComponent] INFO  before
Exchange[BodyType:org.apache.camel.component.file.GenericFile,
Body:<order customerId="4444"><item>Camel in action</item></order>]
2017-05-17 16:18:58,564 [: FileComponent] INFO  after
Exchange[BodyType:com.sun.org.apache.xerces.internal.dom.
DeferredDocumentImpl, Body:<order customerId="4444"
orderId="7303381"><item>Camel in action</item></order>]
```

Here you can see that the second log line has an `orderId` attribute with the value of `7303381`, whereas the first doesn't. If you run it again, you'll see a different order ID because it's a random value. You can experiment with this example, perhaps changing how the order ID is generated.

Using namespaces with @XPath

In the preceding example, the XML order didn't include a namespace. When using namespaces, the bean parameter binding must include the namespace(s) in the method signature as highlighted:

```
public Document handleIncomingOrder(
  @Body Document xml,
  @XPath(
    value = "/c:order/@customerId",
    namespaces = @NamespacePrefix(
      prefix = "c",
      uri = "http://camelinaction.com/order")) int customerId,
  @Bean(ref = "guid", method = "generate") int orderId);
```

The namespace is defined by using the `@NamespacePrefix` annotation embedded in the `@XPath` annotation. Notice that the XPath expression value must use the prefix, which means the expression is changed from `/order/@customerId` to `/c:order/@customerId`.

In recent times, JSON has become increasing more popular to use as a data format when exchanging data. Now imagine that the previous example uses JSON instead of XML; let's see how the Camel `@JsonPath` binding annotation works in practice.

Using @JsonPath binding annotation

To use `@JsonPath`, you first have to include the Camel component, which Maven users can do by adding the following dependency to their pom.xml file:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jsonpath</artifactId>
  <version>2.20.1</version>
</dependency>
```

Instead of dealing with incoming orders as XML documents, the orders are now in JSON format, as in the sample shown here:

```
{
    "order": {
        "customerId": 4444,
        "item": "Camel in Action"
    }
}
```

The following listing shows how to transform incoming orders in JSON format to a CSV representation ❷ by mapping the `customerId` and `item` fields ❶ to bean parameters by using the `@JsonPath` annotation.

Listing 4.9 A Service that transforms JSON to CSV by using a Java bean

```
import org.apache.camel.jsonpath.JsonPath;
import org.apache.camel.language.Bean;

public class JSonOrderService {

  public String handleIncomingOrder(
          @JsonPath("$.order.customerId") int customerId,     ❶
```

(1)

Bindings from JSON document to bean parameters using @JsonPath

```
                    @JsonPath("$.order.item") String item,
                    @Bean(ref = "guid", method = "generate") int orderId)

    return String.format("%d,%d,%s", orderId, customerId, item);   (2)
```

(2)

Returns a CSV representation of the incoming data

```
    }
}
```

The book's source code contains this example in the chapter4/json directory. Maven users can run the example by using the following Maven goal:

```
mvn test -Dtest=JsonOrderTest
```

When running the example, you should see a log after the message transformation, such as this:

```
INFO  after - Exchange[ExchangePattern: InOnly, BodyType: String, Body:5
```

TIP JsonPath allows you to work with JSON documents as XPath does for XML. As such, JsonPath offers a syntax that allows you to define expressions and predicates. For more information about the syntax, consult the JsonPath documentation at https://github.com/json-path/JsonPath.

What you've seen in this and the previous section is the need to use Camel annotations to declare the required binding information. Using Java annotations is common practice, but the caveat is that adding these annotations requires you to alter the source code of the bean. What if you could specify the binding without having to change the source code of the bean? The following section explains how this is possible.

4.5.5 Parameter binding using method name with signature

Camel also allows you to specify the parameter-binding information by using a syntax that's similar to calling methods in Java. This requires using the method-name header, including the binding information as the method signature.

PROS AND CONS

This is a powerful technique, as it completely decouples Camel from your Java bean. Your Java bean can stay *as is* without having to import any Camel code or dependencies at compile time nor runtime. The caveat is that the binding must be defined in a `String` value that prevents any compile-time checking. In addition, the binding information in the `String` is limited to what the Simple language provides. For example, the `@JsonPath` language annotation covered in the previous section isn't available in the Simple language.

It's easier to explain with an example.

Suppose you have this method used previously as an example in section 4.5.3:

```
public String orderStatus(@Body Integer orderId,
                          @Header("customerId") Integer customerId,
                          @Header("customerType") Integer customerType)
  ...
}
```

Instead of using the Camel annotations, the source code becomes this:

```
public String orderStatus(Integer orderId,
                          Integer customerId,
                          Integer customerType) {
  ...
}
```

Now the method is clean and has no Camel annotations, and the code has no Camel dependencies. The code can compile without having Camel JARs on the classpath. What you have to do is to specify the binding details in the Camel route instead:

```
from("direct:start")
  .bean("orderService",
    "orderStatus(${body}, ${header.customerId}, ${header.customerType}")
```

And in XML DSL:

```
<route>
  <from uri="direct:start"/>
  <bean ref="orderService" method="
    orderStatus(${body}, ${header.customerId}, ${header.customerType}"/>
</route>
```

What happened? If you take a closer look, you'll see that the method-name parameter almost resembles Java source code, as if calling a method with three parameters. If you were to write some Java code and use the Camel Exchange API to call the method from Java, the source code would be something like this:

```
OrderStatus bean = ...
Message msg = exchange.getIn();
String status = bean.orderStatus(msg.getBody(), msg.getHeader("customerI
                                 msg.getHeader("customerType"));
```

This code would be Java source code and therefore compiled by the Java compiler, so the parameter binding happens at compile time—whereas the preceding Camel routes are defined using a `String` value in Java code, or an XML attribute. Because it's not the Java compiler that performs the binding, Camel parses the `String` value, which happens upon starting up Camel. Figure 4.9 illustrates how each of the three Java parameters corresponds to a value.

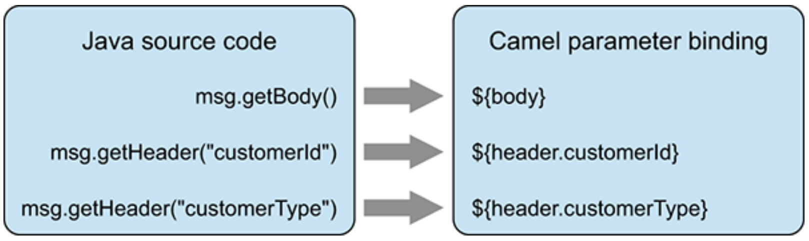

Figure 4.9 Bean parameter bindings in Camel resemble Java source code when calling a parameter with three values.

You may have guessed what syntax the Camel parameter binding is using, from the right-hand side in figure 4.9. Yes, it's the Simple language. Camel allows you to specify the bean parameter binding by using the method-name signature. Table 4.5 lists the rules that apply when using parameter binding and the method-name signature.

Table 4.5 Rules for bean parameter binding using method-name signature

Rule	Description
A Boolean value	The parameter can bind to a Boolean type by using a `true` or `false` value. For example: `method="goldCustomer(true)"`.
A numeric value	The parameter can bind to a number type by using an integer value. For example: `method="goldCustomer(true, 123)"`.
A null value	The parameter can bind as a `null` value. For example: `method="goldCustomer(true, 123, null)"`.
A literal value	The parameter can bind to a `String` type by using a literal value. For example: `method="goldCustomer(true, 123, 'James Strachan')"`.
A simple expression	The parameter can bind to any type as a Simple expression. For example: `method="goldCustomer(true, ${header.customerId}, ${body.customer.name})"`.

A value that can't apply to any of the preceding rules would cause Camel to throw an exception at runtime, stating a parameter-binding error.

BEAN-BINDING SUMMARY

Camel's rules for bean parameter binding can be summarized as follows:

- All parameters having a Camel annotation will be bound (tables 4.3 and 4.4).
- All parameters of a Camel built-in type will be bound (table 4.2).
- The first parameter is assumed to be the message in body (if not already bound).
- If a method name was given containing parameter-binding details, those will be used (table 4.5).
- All remaining parameters will be unbound, and Camel will pass in empty values.

You've seen all there is to bean binding. Camel has a flexible mechanism that adapts to your existing beans, and when you have multiple parameters, Camel provides annotations to bind the parameters properly.

Camel makes it easy to call Java beans from your routes, allowing you to invoke your business logic, or as you saw in chapter 3, to perform message translation using Java code. On top of that, Camel also makes it easy to use beans as decision makers during routing.

4.6 Using beans as predicates and expressions

Some enterprise integration patterns (EIPs), such as the Content-Based Router and Message Filter, use predicates to determine how they should process messages. Other EIPs, such as the Recipient List, Dynamic Router, and Idempotent Consumer, require using expressions. This section covers how to use Java beans as predicates or expressions with those EIPs.

First, let's briefly review Camel predicates and expressions. A *predicate* is an expression that evaluates as a Boolean; its return value is either `true` or `false`, as depicted in the Camel predicate API:

```
boolean matches(Exchange exchange);
```

An expression, on the other hand, evaluates to anything; its return value is a `java.lang.Object`, as the following Camel API defines:

```
Object evaluate(Exchange exchange);
```

4.6.1 Using beans as predicates in routes

As you learned in chapter 2, one of the most commonly used EIP patterns is the content-based router. This pattern uses one or more predicates to determine how messages are routed. If a predicate matches, the message is routed down the given path.

The example we'll use is a customer order system that routes orders from gold, silver, and regular customers (0–999 = gold, 1000–4999 = silver, 5000+ = regular). A bean is used to determine which kind of customer level the routed message is from. A simple implementation is shown in the following listing.

Listing 4.10 Using a bean with methods as a predicate in Camel

```
public class CustomerService {

 public boolean isGold(@JsonPath("$.order.loyaltyCode") int id) {   1
```

1 Method to determine whether it's a gold customer

```
    return id < 1000;
  }

 public boolean isSilver(@JsonPath("$.order.loyaltyCode") int id) {   2
```

2 Method to determine whether it's a silver customer

```
      return id >= 1000 && id < 5000;
    }
  }
```

The bean implements two methods, isGold ❶ and isSilver ❷, both returning a Boolean that allows Camel to use the methods as predicates. Each method uses bean parameter binding to map from the Camel message to the customer ID. This example continues from the previous JSON example and uses the `@JsonPath` annotation to extract the customer ID from the message body in JSON format.

TIP The bean in listing 4.10 uses the bean parameter binding covered in section 4.5. Therefore, you can use what you've learned, and instead of `@JsonPath`, the bean could have multiple parameters.

To use the bean as a predicate in the Content-Based Router pattern in Camel is easy, as shown in the following listing.

Listing 4.11 The content-based router uses the bean as a predicate in XML

```
<bean id="customerService" class="camelinaction.CustomerService"/>  ❶
```

❶ Declares the bean so you can refer to the bean in the upcoming route

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="file://target/order"/>
    <choice>
      <when>
    <method ref="customerService" method="isGold"/>  ❷
```

❷ The method call predicate that invokes the isGold method on the bean

```
          <to uri="mock:queue:gold"/>
        </when>
        <when>
      <method ref="customerService" method="isSilver"/>    ❸
```

❸

The method call predicate that invokes the isSilver method on the bean

```
          <to uri="mock:queue:silver"/>
        </when>
        <otherwise>
          <to uri="mock:queue:regular"/>
        </otherwise>
      </choice>
    </route>
</camelContext>
```

As you can see, using a bean as a predicate is easy, by using `<method>` ❷ ❸ to set up the bean as the predicate. Prior to that, the bean needs to be declared using a `<bean>` element 1.

The following listing shows how to implement the same route as in listing 4.11 but using Java DSL.

Listing 4.12 The content-based router using the bean as a predicate in Java DSL

```
public void configure() throws Exception {
  from("file://target/order")
    .choice()
      .when(method(CustomerService.class, "isGold"))    ❶
```

❶

The method call predicate that invokes the isGold method on the bean

```
        .to("mock:queue:gold")
      .when(method(CustomerService.class, "isSilver"))    ❷
```

❷
The method call predicate that invokes the isSilver method on the bean

```
          .to("mock:queue:silver")
        .otherwise()
          .to("mock:queue:regular");
  }
```

The route in listing 4.12 is almost identical to listing 4.11. But this time we show a slight variation that allows you to refer to the bean by its class name, `CustomerService.class` ❶ ❷. When doing this, Camel may be required to create a new instance of the bean while the route is being created, and therefore the bean is required to have a default no-argument constructor. But if the method `isGold` or `isSilver` is a static method, Camel won't create a new bean instance, but will invoke the static methods directly.

The Java DSL could also, as in listing 4.11, refer to the bean by an ID, which would require using the bean name instead of the class name, as shown in this snippet:

```
  .when(method("customerService", "isGold"))
    .to("mock:queue:gold")
```

In Java DSL, there's more, as you can combine multiple predicates into compound predicates.

Using compound predicates in Java

This is an exclusive feature in Java only that allows you to combine one or more predicates into a compound predicate. This can be used to logically combine (and, or, not) multiple predicates, even if they use different languages—for example, combing XPath, Simple, and Bean, as shown in listing 4.13.

Listing 4.13 Using PredicateBuilder to build a compound predicate using XPath, Simple, and method call together

```
return new RouteBuilder() {
  public void configure() throws Exception {
  Predicate valid = PredicateBuilder.and(   (1)
```

(1)

Uses PredicateBuilder to combine the predicates using and

```
       xpath("/book/title = 'Camel in Action'"),   (2)
```

(2)

The XPath predicate that tests whether the book title is Camel in Action

```
       simple("${header.source} == 'batch'"),   (3)
```

(3)

The Simple predicate that tests whether header.source has the value batch

```
       not(method(CompoundPredicateTest.class, "isAuthor")));   (4)
```

(4)

The method call predicate calls the isAuthor method that's negated using not, and therefore should return false

```
    from("direct:start")
   .validate(valid)   (5)
```

Uses the compound predicate in the Camel route using the Validate EIP pattern

```
            .to("mock:valid");
        }
    };
```

The substance is the `org.apache.camel.builder.PredicateBuilder` ❶ that has a number of builder methods to combine predicates. We use `and`, which means all three ❷ ❸ ❹ predicates must return `true` for the compound predicate to return `true`. If one of them returns `false`, the compound predicate response is also `false`.

The source code for the book includes this example in the chapter4/predicate directory, which can be executed using the following Maven goal:

```
mvn test -Dtest=CompoundPredicateTest
```

Beans can also be used as expressions in routes, which is the next topic.

4.6.2 Using beans as expressions in routes

This section covers a common use case with Camel: using a *dynamic to* route. You'll see how to route a message in Camel to a destination that's dynamically computed at runtime with information from the message itself.

In the *Enterprise Integration Patterns* book, the Recipient List pattern best describes the *dynamic to*, illustrated in figure 4.10.

The Recipient List EIP pattern in Camel is a versatile and flexible implementation that offers many features and functions. This EIP is covered in more detail in the following chapter.

To use a bean with the Recipient List pattern, you use a simple use case with a customer service system that routes orders depending on geographical region of the customer. A naive bean could be implemented in a few lines of code, as shown here:

```
public class CustomerService {

    public String region(@JsonPath("$.order.customerId") int customerId) {
      if (customerId < 1000) {
        return "US";
      } else if (customerId < 2000) {
        return "EMEA";
      } else {
        return "OTHER";
      }
    }
  }
```

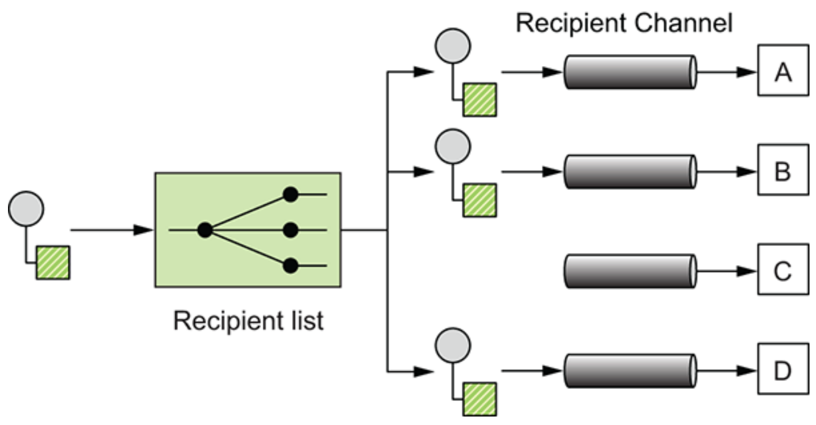

Figure 4.10 Recipient List EIP sends a copy of the same message to numerous dynamic computed destinations. The *dynamic to* is just a single dynamic computed destination.

To use the bean as an expression in the Recipient List pattern in Camel is also easy, as shown in the following listing.

Listing 4.14 Using a bean as an expression during routing with the recipient list to act as a dynamic to

```
<bean id="customerService" class="camelinaction.CustomerService"/>    1
```

1

Bean to be used as expression implementing logic to determine customer geographical location

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="file://target/order"/>
    <setHeader headerName="region">
      <method ref="customerService" method="region"/>    2
```

❷
Calling the bean to set a header with the region of the customer

```
    </setHeader>
    <recipientList>
      <simple>mock:queue:${header.region}</simple>    ❸
```

❸
Dynamic to using recipient list to route to a queue with the name of the region

```
    </recipientList>
  </route>
</camelContext>
```

First, you need to set up the bean as a `<bean>` ❶ so you can refer to the bean by using its ID, `customerService`. Then the bean is invoked to return the geographical region from which the customer order is placed ❷. The bean will return either `US`, `EMEA`, or `OTHER`, depending on the region. Then the message is routed using the recipient list ❸ to a single destination, hence it's also referred as *dynamic to*. If the customer is from `US`, the destination would be `mock:queue:US`, and for `EMEA` it would be `mock:queue:EMEA`.

Listing 4.14 could be implemented in Java DSL with fewer lines of code, as shown here:

```
from("file://target/order")
  .setHeader("region", method(CustomerService.class, "region"))
  .recipientList(simple("mock:queue:${header.region}"));
```

This book's source code includes this example in the chapter4/expression directory, which you can run using the following Maven goals:

```
mvn test -Dtest=JsonExpressionTest
mvn test -Dtest=SpringJsonExpressionTest
```

In the example, you didn't call the bean from the recipient list, but instead computed the region as a header. You could omit this and call the bean directly from the recipient list, as the following code shows:

```
from("file://target/order")
  .recipientList(simple("mock:queue:" +
     "${bean:camelinaction.CustomerService?method=region}"));
```

And using XML DSL:

```
<from uri="file://target/order"/>
<recipientList>
  <simple>
    mock:queue:${bean:camelinaction.CustomerService?method=region}
  </simple>
</recipientList>
```

A SIMPLER DYNAMIC TO

The Recipient List is the EIP pattern that allows you to send messages to one or more dynamic endpoints, and it has been our answer in Camel since Camel was created. But over the years, we've learned that some Camel users weren't familiar with the Recipient List pattern, and therefore couldn't find a way to easily send a message to a single Camel endpoint that was dynamically computed with information from the message. As a new user to Camel, you quickly get the hang of using `<from>` and `<to>` in your routes, which are key patterns. To make this easier in Camel, we introduced `<toD>` as an alternative to `<recipientList>`.

Using toD as dynamic to

If you need to send a message to a single dynamic computed endpoint, you should favor using `toD`. It's specially designed for the single-destination use case, whereas Recipient List is a much more elaborate EIP pattern that in those use cases can be overkill. `toD` has the following characteristics:

- Can send to only one dynamic destination

- Can use only the Simple language as the expression to compute the dynamic endpoint

Dynamic to has specifically been designed to work like to, but can send the message to a dynamic computed endpoint by using the Simple language. If you have any other needs, use the more powerful Recipient List EIP.

The previous example in listing 4.14 can be simplified to use `<toD>`, as shown in the following listing.

Listing 4.15 Using a bean as an expression during routing with dynamic to

```
<bean id="customerService" class="camelinaction.CustomerService"/>
```

Bean to be used as expression, implementing logic to determine customer geographical location

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="file://target/order"/>
    <setHeader headerName="region">
      <method ref="customerService" method="region"/>
```

Calling the bean to set a header with the region of the customer

```
    </setHeader>
    <toD uri="mock:queue:${header.region}"/>
```

Dynamic to route to a queue with the name of the region

```
  </route>
</camelContext>
```

The example is only a mere three lines of code when using Java DSL, as shown here:

```
from("file://target/order")
  .setHeader("region", method(CustomerService.class, "region"))
  .toD("mock:queue:${header.region}");
```

And if you want to take the example down to two lines of code, you can call the bean directly instead of setting the header:

```
from("file://target/order")
  .toD("mock:queue:${bean:customerService?method=region}");
```

This book's source code includes this example in the chapter4/expression directory, and you can run the example by using the following Maven goals:

```
mvn test -Dtest=JsonToDExpressionTest
mvn test -Dtest=SpringJsonToDExpressionTest
```

That's all we have to say about using beans as predicates and expressions. In fact, we've reached the end of this chapter.

4.7 Summary and best practices

We've now covered another cornerstone of using beans with Camel. It's important that end users of Camel can use the POJO programming model and have Camel easily use those beans (POJOs). Beans are just Java code, which is a language you're likely to feel comfortable using. If you hit a problem that you can't work around or figure out how to resolve using Camel and EIPs, you can always resort to using a bean and letting Camel invoke it.

We unlocked the algorithm used by Camel to select which method to invoke on a bean. You learned why this is needed: Camel must resolve method selection at runtime, whereas regular Java code can link method invocations at compile time.

We also covered what bean parameter binding is and how to bind a Camel exchange to any bean method and its parameters. You learned how to use annotations to provide fine-grained control over the bindings, and even how Camel can help bind XPath or JsonPath expressions to parameters, which is a great feature when working with XML or JSON messages.

Let's pull out some of the key practices you should take away from this chapter:

- *Use beans*—Beans are Java code, and they give you all the horsepower of Java.
- *Use loose coupling*—Prefer using beans that don't have a strong dependency on the Camel API. Camel is capable of adapting to existing bean-method signatures, so you can use any existing API you may have, even if it has no dependency on the Camel API. Unit testing is also easier because your beans don't depend on any Camel API. You can even have developers with no Camel experience develop the beans, and then have developers with Camel experience use those beans.
- *Use method facades*—If calling your existing beans in a loosely coupled fashion seems too difficult, or you have to specify too many bean parameter mappings or want to avoid introducing Camel annotations on your existing bean, you can use method facades. You can create a new bean as a facade, and use Java code to implement the mapping between Camel and your existing bean.
- *Prefer simple method signatures*—Camel bean binding is much simpler when method signatures have as few parameters as possible.
- *Specify method names*—Tell Camel which method you intend to invoke, so Camel doesn't have to figure it out. You can also use `@Handler` in the bean to tell Camel which method it should pick and use.
- *Favor parameter binding using method-signature syntax*—When calling methods on POJOs from Camel routes, it's often easier to specify the parameter binding in the method-name signature in the route that closely resembles Java code. This makes it easier for other users of Camel to understand the code.
- *Use the powers of Java as predicates or expressions*—When you need to define predicates or expressions when using a more powerful language, consider using plain-old Java code to implement this logic. The

Java code can be loosely coupled from Camel and allows for easier unit testing the code, isolated from Camel.

We've now covered three crucial features of integration kits: routing, transformations, and using beans. In chapter 2, you were exposed to some of Camel's routing capabilities by using standard EIPs. In the next chapter, you'll look at some of the more complex EIPs available in Camel.