8

Developing Camel projects

This chapter covers

- Managing Camel projects with Maven
- Developing Camel projects in the Eclipse IDE
- Debugging with Camel
- Creating custom components
- Using the API component framework
- Creating custom data formats

At this point, you should know a thing or two about how to develop Camel routes and how to take advantage of many Camel features. But do you know how to best start a Camel project from scratch? You could take an existing example and modify it to fit your use case, but that's not always ideal. And what if you need to integrate with a system that isn't supported out of the box by Camel?

This chapter shows you how to build your own Camel applications. We'll go over the Maven archetype tooling that allows you to skip the boring boilerplate project setup and create new Camel projects with a single command. We'll also show you how to start a Camel project from Eclipse when you need the extra power that an IDE provides. We'll even show you how to debug your new Camel application.

After that, we'll show you how to extend Camel by creating custom components. Creating custom components around large APIs isn't easy, so we'll also discuss Camel's solution to this problem: the API component framework. This framework can generate a near fully functional component just from scanning an arbitrary Java API. Finally, we'll wrap up by discussing custom data formats.

8.1 Managing projects with Maven

Camel was built using Apache Maven right from the start, so it makes sense that creating new Camel projects is easiest when using Maven. This section covers Camel's Maven *archetypes*, which are preconfigured templates for creating various types of Camel projects. After that, you'll learn about using Maven dependencies to load Camel modules and their third-party dependencies into your project.

Section 1.2 of chapter 1 has an overview of Apache Maven. If you need a Maven refresher, you might want to review that section before continuing here.

8.1.1 Using Camel Maven archetypes

Creating Maven-based projects is simple. You mainly have to worry about creating a POM file and the various standard directories that you'll use in your project. But if you're creating many projects, this can get repetitive because a lot of boilerplate setup is required for new projects.

Archetypes in Maven provide a means to define project templates and generate new projects based on those templates. They make creating new Maven-based projects easy because they create all the boilerplate POM elements, as well as key source and configuration files useful for particular situations.

NOTE For more information on Maven archetypes, see the guide: http://books.sonatype.com/mvnref-book/reference/archetypes.html.

As illustrated in figure 8.1, this is all coordinated by the Maven archetype plugin. This plugin accepts user input and replaces portions of the archetype to form a new project.

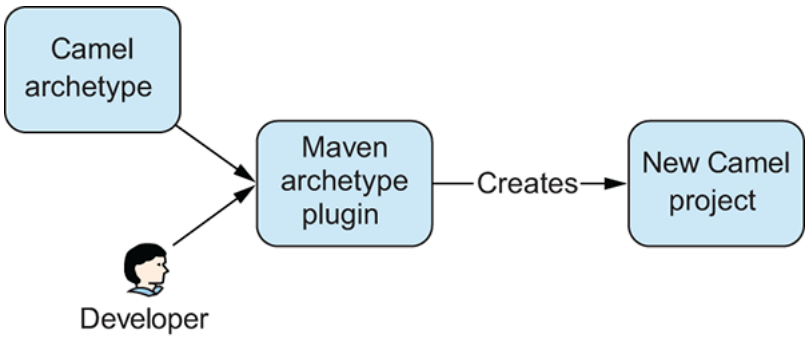

Figure 8.1 A Camel archetype and user input are processed by the Maven archetype plugin, which then creates a new Camel project.

To demonstrate how this works, let's look at the Maven quickstart archetype, which generates a plain Java application (no Camel dependencies). It's the default option when you run this command:

```
mvn archetype:generate
```

The archetype plugin asks you various questions, such as what `groupId` and `artifactId` to use for the generated project. When it's complete, you'll have a directory structure similar to this:

```
myApp
├── pom.xml
└── src
    ├── main
    │   └── java
    │       └── camelinaction
    │       └── App.java
    └── test
        └── java
            └── camelinaction
            └── AppTest.java
```

In this structure, `myApp` is the `artifactId`, and `camelinaction` is the `groupId`. The archetype plugin created a pom.xml file, a Java source file, and a unit test, all in the proper locations.

NOTE Maven follows the paradigm of convention over configuration, so locations are important. Wikipedia provides details about this paradigm: https://en.wikipedia.org/wiki/Convention_over_configuration.

Without any additional configuration, Maven knows that it should compile the Java source under the src/main/java directory and run all unit tests under the src/test/java directory. To kick off this process, you need to run the following Maven command:

```
mvn test
```

If you want to take it a step further, you could tell Maven to create a JAR file after compiling and testing by replacing the `test` goal with `package`.

You could start using Camel right from this example project, but it would involve adding Camel dependencies such as camel-core, starting up the `CamelContext`, and creating the routes. Although this wouldn't take that long, there's a much quicker solution: you can use one of the 11 archetypes provided by Camel to generate all this boilerplate Camel stuff for you. Table 8.1 lists these archetypes and their main use cases.

Table 8.1 Camel's Maven archetypes

Archetype name	Description
camel-archetype-blueprint	Creates a Camel project that uses OSGi Blueprint. Ready to be deployed in OSGi.
camel-archetype-component	Creates a new Camel component.
camel-archetype-api-component	Creates a new Camel component that's based on a third-party API.
camel-archetype-cdi	Creates a Camel project with Contexts and Dependency Injection (CDI) support.
camel-archetype-connector	Creates a new Camel connector. A *connector* is a simplified and preconfigured component (set up for a specific use-case).
camel-archetype-dataformat	Creates a new Camel data format.
camel-archetype-java	Creates a Camel project that defines a sample route in the Java DSL.
camel-archetype-java8	Creates a Camel project that defines a sample route in the Java DSL using Java 8 features such as lambda expressions and method references.
camel-archetype-	Creates a Camel project that loads up a `CamelContext` in Spring and defines a sample

Archetype name	Description
spring	route in the XML DSL.
camel-archetype-spring-boot	Creates a new Camel project by using Spring Boot. See the following note for an alternate way of creating Spring Boot–based Camel applications.
camel-archetype-web	Creates a Camel project that includes a few sample routes as a WAR file.

NOTE In addition to using the camel-archetype-spring-boot archetype to create Camel projects using Spring Boot, you can use the Spring Boot starter website at http://start.spring.io. This website allows you to customize the features/dependencies you want to use in your project (Camel, web application, security, transactions, and so forth), so it greatly reduces project setup time. We covered Spring Boot in chapter 7, section 7.2.4.

Out of these 11 archetypes, the most commonly used one is probably the camel-archetype-java archetype. You'll try this next.

Using the camel-archetype-java archetype

The camel-archetype-java archetype listed in table 8.1 boots up `CamelContext` and a Java DSL route. With this, we'll show you how to recreate the order-routing service for Rider Auto Parts as described in chapter 2. The project will be named `order-router`, and the package name in the source will be `camelinaction`.

To create the skeleton project for this service, run the following Maven command:

```
mvn archetype:generate \
  -B \
```

```
    -DarchetypeGroupId=org.apache.camel.archetypes \
    -DarchetypeArtifactId=camel-archetype-java \
    -DarchetypeVersion=2.20.1 \
    -DgroupId=camelinaction \
    -DartifactId=order-router
```

You specify the archetype to use by setting the `archetypeArtifactId` property to `camel-archetype-java`. You could replace this with any of the archetype names listed in table 8.1. The `archetypeVersion` property is set to the version of Camel that you want to use.

The `generate` goal of the Maven archetype plugin can also be used in an interactive fashion. Just omit the `-B` option, and the plugin will prompt you through the archetype you want to use. You can select the Camel archetypes through this interactive shell as well, so it's a useful option for developers new to Camel.

After a few seconds of activity, Maven will have created an order-router subdirectory in the current directory. The order-router directory layout is shown in the following listing.

Listing 8.1 Layout of the project created by camel-archetype-java

```
order-router
├── pom.xml
├── ReadMe.txt
└── src
    ├── data
    │   ├── message1.xml
```

Test data

```
    │   └── message2.xml
    ├── main
    │   ├── java
    │   │   └── camelinaction
    │   │       ├── MainApp.java
```

CamelContext setup

```
__|__|____└── MyRouteBuilder.java_____
```

Sample Java DSL route

```
    |   └── resources
__|____└── log4j.properties_____
```

Logging configuration

```
    └── test
        ├── java
        |   └── camelinaction
        └── resources
```

The archetype gives you a runnable Camel project, with a sample route and test data to drive it. The ReadMe.txt file tells you how to run this sample project: run `mvn compile exec:java`. Camel will continue to run until you press Ctrl-C, which causes Camel to stop.

While running, the sample route consumes files in the src/data directory and, based on the content, routes them to one of two directories. If you look in the target/messages directory, you should see something like this:

```
target/messages
├── others
|   └── message2.xml
└── uk
    └── message1.xml
```

Now you know that Camel is working on your system, so you can start editing MyRouteBuilder.java to look like the order-router application. You can begin by setting up FTP and web service endpoints that route to a JMS queue for incoming orders:

```
from("ftp://rider@localhost:21000/order?password=secret&delete=true")
  .to("jms:incomingOrders");

from("cxf:bean:orderEndpoint")
  .inOnly("jms:incomingOrders")
  .transform(constant("OK"));
```

At this point, if you try to run the application again using `mvn compile exec:java`, you'll get the following error message:

```
Failed to resolve endpoint: ftp://rider@localhost:21000/order?
delete=true&password=secret due to: No component found with scheme: ftp
```

Camel couldn't find the FTP component because it isn't on the classpath. You'd get the same error message for the CXF and JMS endpoints. There are other bits you have to add to your project to make this a runnable application: a test FTP server running on localhost, a CXF configuration, a JMS connection factory, and so on. A complete project is available in the book's source under chapter8/order-router-full.

For now, you'll focus on adding component dependencies using Maven.

8.1.2 Using Maven to add Camel dependencies

Technically, Camel is just a Java application. To use it, you add its JARs to your project's classpath. But using Maven to access these JARs will make your life a whole lot easier. Camel itself was developed using Maven for this very reason.

In the previous section, you saw that using an FTP endpoint with only the camel-core module as a dependency won't work. You need to add the camel-ftp module as a dependency to your project. In chapters 2 and 6, you saw that this was accomplished by adding the following to the dependencies section of the pom.xml file:

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

This dependency element tells Maven to download the camel-ftp JAR from Maven's central repository at http://repo1.maven.org/maven2/org/apache/camel/camel-ftp/2.20.1/camel-ftp-2.20.1.jar. This download URL is built up from Maven's central repository URL (http://repo1.maven.org/maven2) and Maven coordinates (`groupId`, `artifactId`, and so on) specified in the dependency element. After the download is complete, Maven will add the JAR to the project's classpath.

One detail that may not be obvious at first is that this dependency also has *transitive dependencies*. What are transitive dependencies? Well, in this case you have a project called order-router and you've added a dependency on camel-ftp. The camel-ftp module also has a dependency on commons-net, among others, so you can say that commons-net is a transitive dependency of order-router. Transitive dependencies are dependencies that a dependency has—the dependencies of the camel-ftp module, in this case.

When you add camel-ftp as a dependency, Maven will look up camel-ftp's POM file from the central Maven repository and look at the dependencies it has. Maven will then download and add those dependencies to this project's classpath.

The camel-ftp module adds a whopping 39 transitive dependencies to your project! Luckily, only 6 of them are needed at runtime; the other 33 are used during testing. The six transitive runtime dependencies can be viewed as a tree, as shown in figure 8.2.

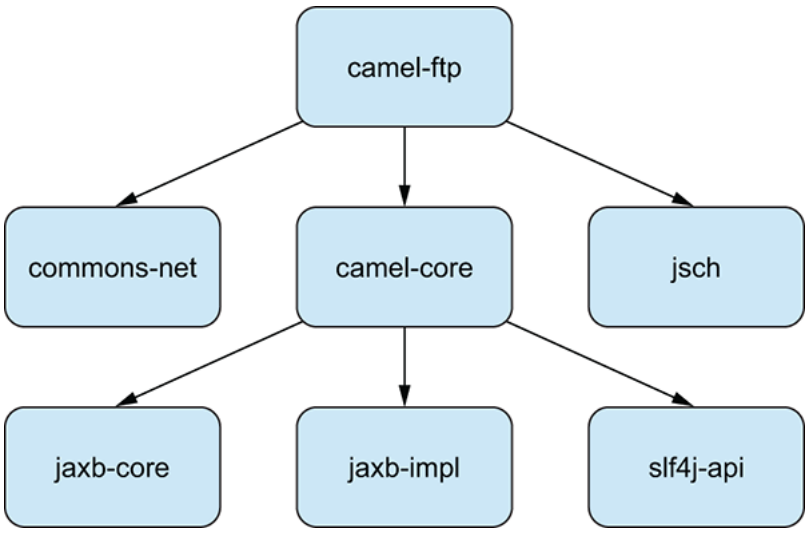

Figure 8.2 Transitive runtime dependencies of the camel-ftp module. When you add a dependency on camel-ftp to your project, you'll also get its transitive dependencies added to your classpath. In this case, commons-net, camel-core, and jsch are added. Additionally, camel-core has a dependency on jaxb-core, jaxb-impl, and slf4j-api, so these are added to the classpath as well.

You're already depending on camel-core in the order-router project, so only two dependencies—commons-net and jsch—are brought in by camel-ftp.

This is a view of only a small number of dependencies, but you can recognize that the dependency tree can get complex. Fortunately, Maven finds these dependencies for you and resolves any duplicate dependencies. The bottom line is that when you're using Maven, you can worry less about your project's dependencies.

If you want to know what your project's dependencies are (including transitive ones), Maven offers the `dependency:tree` command. To see the dependencies in your project, run the following command:

```
mvn dependency:tree -Dscope=runtime
```

After a few seconds of work, Maven will print out a listing like this:

```
[INFO] --- maven-dependency-plugin:2.10:tree (default-cli)
    @ chapter8-order-router ---
[INFO] camelinaction:chapter8-order-router:jar:2.0.0
[INFO] +- org.apache.camel:camel-core:jar:2.20.1:compile
[INFO] |  +- org.slf4j:slf4j-api:jar:1.7.21:compile
[INFO] |  +- com.sun.xml.bind:jaxb-core:jar:2.2.11:compile
[INFO] |  \- com.sun.xml.bind:jaxb-impl:jar:2.2.11:compile
[INFO] +- org.apache.camel:camel-spring:jar:2.20.1:compile
[INFO] |  +- org.springframework:spring-core:jar:4.3.12.RELEASE:compile
```

```
[INFO] |  |  \- commons-logging:commons-logging:jar:1.2:compile
[INFO] |  +- org.springframework:spring-aop:jar:4.3.12.RELEASE:compile
[INFO] |  +- org.springframework:spring-context:jar:4.3.12.RELEASE:compi
[INFO] |  +- org.springframework:spring-beans:jar:4.3.12.RELEASE:compile
[INFO] |  +- org.springframework:spring-expression:jar:4.3.12.RELEASE:co
[INFO] |  \- org.springframework:spring-tx:jar:4.3.12.RELEASE:compile
[INFO] +- org.apache.camel:camel-ftp:jar:2.20.1:compile
[INFO] |  +- com.jcraft:jsch:jar:0.1.54:compile
[INFO] |  \- commons-net:commons-net:jar:3.6:compile
[INFO] +- log4j:log4j:jar:1.2.17:compile
[INFO] \- org.slf4j:slf4j-log4j12:jar:1.7.21:compile
```

Here, you can see that Maven is adding 17 JARs to your project's runtime classpath, even though you added only camel-core, camel-spring, camel-ftp, log4j, and slf4j-log4j12. Some dependencies are coming from several levels deep in the dependency tree.

SURVIVING WITHOUT MAVEN

As you can imagine, adding all these dependencies to your project without the help of Maven would be tedious. If you absolutely must use an alternative build system, you can still use Maven to get the required dependencies for you by following these steps:

1. Download the POM file of the artifact you want. For camel-ftp, this would be http://repo1.maven.org/maven2/org/apache/camel/camel-ftp/2.20.1/camel-ftp-2.201.pom.
2. Run `mvn -f camel-ftp-2.20.1.pom dependency:copy-dependencies`.
3. The dependencies for camel-ftp are located in the target/dependency directory. You can now use these in whatever build system you're using.

If you absolutely can't use Maven but would still like to use Maven repos, Apache Ivy (http://ant.apache.org/ivy) is a great dependency management framework for Apache Ant that can download from Maven repos. Other than that, you'll have to download the JARs yourself from the Maven central repo.

You now know all you need to develop Camel projects by using Maven. To make you an even more productive Camel developer, let's now look at developing Camel applications inside an IDE, such as Eclipse.

8.2 Using Camel in Eclipse

We haven't mentioned IDEs much so far, mostly because you don't need an IDE to use Camel. Certainly, though, you can't match the power and ease of use an IDE gives you. From a Camel point of view, having the Java or XML DSLs autocomplete for you makes route development a whole lot easier. The common Java debugging facilities and other tools will further improve your experience.

8.2.1 Creating a new Camel project

Because Maven is used as the primary build tool for Camel projects, we'll show you how to use the Maven tooling in Eclipse to load up your Camel project. The standard Eclipse edition for Java developers has Maven tooling installed by default, but if you don't have that particular edition, you can easily search for and install the m2e plugin within the IDE. One thing that some developers like right away is that you don't have to leave the IDE to run Maven command-line tools during development. You can even access the Camel archetypes right from Eclipse. To demonstrate this feature, let's re-create the chapter8-order-router example you looked at previously.

Click File > New > Maven Project to start the New Maven Project wizard. Click Next on the first screen, and you'll be presented with a list of available archetypes, as shown in figure 8.3. After filtering down to include only org.apache.camel, you can easily spot the Camel archetypes.

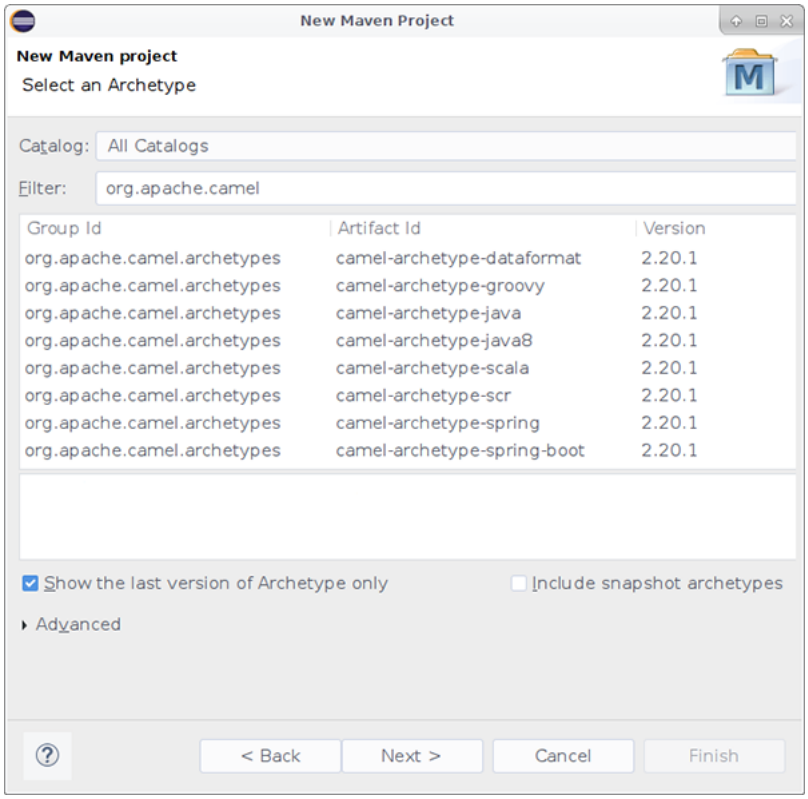

Figure 8.3 The New Maven Project wizard allows you to generate a new Camel project right in Eclipse.

Using Camel archetypes in this way is equivalent to the technique you used in section 8.1.1.

To run the order-router project with Eclipse, right-click the project in the Package Explorer and click Run As > Maven Build. This brings up an Edit Configuration dialog box where you can specify the Maven goals to use as well as any parameters. For the order-router project, use the `exec:java` goal, as shown in figure 8.4.

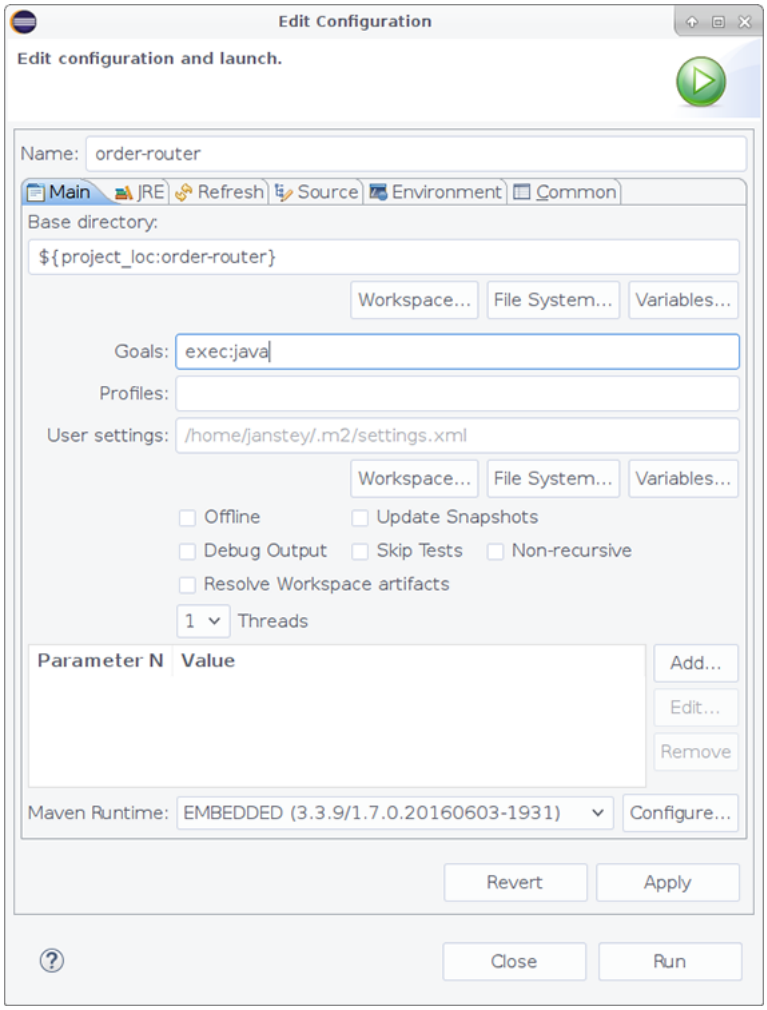

Figure 8.4 Right-clicking the order-router project in the Package Explorer and clicking Run As > Maven Build brings up this Edit Configuration dialog box. The `exec:java` Maven goal has been entered.

Clicking Run in this dialog box executes the `mvn exec:java` command in Eclipse, with console output showing in the Eclipse Console view, as shown in figure 8.5.

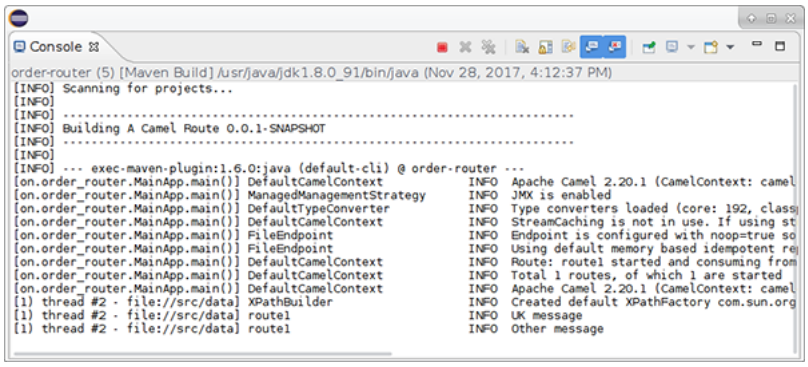

Figure 8.5 Console output from the order-router project when running the `exec:java` Maven goal

8.3 Debugging an issue with your new Camel project

You've just created a new Camel project in Eclipse by using an archetype, and now you want to know how to debug this thing if a problem arises.

Well, Camel is technically just a Java framework, so you can set breakpoints wherever you like. But you'll hit a couple of issues if you try to step through your route in the debugger. Take the `RouteBuilder` from the order-router project, for example:

```
public class MyRouteBuilder extends RouteBuilder {
    public void configure() {
       from("file:src/data?noop=true")
          .choice()
             .when(xpath("/person/city = 'London'"))
                .log("UK message")
                .to("file:target/messages/uk")
             .otherwise()
                .log("Other message")
                .to("file:target/messages/others");
    }
}
```

If you try to set a breakpoint on, say, the `log("UK message")` line, you won't hit that breakpoint for each message satisfying `when(xpath("/person/city = 'London'"));` the breakpoint will hit only once on startup. This is because `RouteBuilders` just forms the model from which Camel will create a graph of processors. The processor graph forms the runtime structure that you can debug on a per message basis. One trick that's quick to use in a pinch is an anonymous processor inside the route:

```
    .when(xpath("/person/city = 'London'"))
            .log("UK message")
            .process(new Processor() {
                  @Override
                  public void process(Exchange exchange) throws Excepti
                          // breakpoint goes here!
                  }
            })
            .to("file:target/messages/uk")
```

This isn't helpful to use in the XML DSL. Furthermore, having to modify your route just to debug isn't nice. What you can do is to not debug using

the Eclipse debugger at all, but to use Camel's built-in management and monitoring abilities. In chapter 16, we discuss these topics:

- *JMX*—You may overlook JMX in your debugging toolbox for most applications, but Camel has an extensive set of data and operations exposed over JMX. Section 16.2 covers this in detail.
- *Logs*—Logs are the first place you should go when investigating a problem in most applications, and Camel is no different. Section 16.3 covers this.
- *Tracing*—The tracer is probably Camel's most useful built-in tool to diagnose problems. Enabling tracing will log each message at every step of a route. In this way, you can see where a message goes through a route and how it changes at each step. Section 16.3.4 covers this in detail.

If you want to go beyond what Camel has to offer, several tooling projects are out there to assist you with debugging Camel. Chapter 19 covers the following:

- *JBoss Fuse Tooling*—The Fuse Tooling is an Eclipse plugin for Camel development that includes, among many other things, a visual debugger for Camel routes. Section 19.1.1 covers this.
- *hawtio*—The hawtio project is a highly extensible web console for managing Java applications. It has a nice Camel plugin that allows you to step through a route visually in your browser!

Debugging your unit tests

Tooling projects such as hawtio and JBoss Fuse Tooling both tie into the same Camel debugging support to achieve visual debugging. The main debugger API is `org.apache.camel.spi.Debugger`, which contains methods to set breakpoints in routes, step through a route, and so forth. The default implementation is `org.apache.camel.impl.DefaultDebugger`, which is in the camel-core module. How does this help you? Well, most of the Camel testing modules have support to use this default debugger implementation in your unit tests. You can see what's happening before and after each processor invocation in a Camel route's runtime processor graph. Table 8.2 lists the Maven modules as well as which test class to extend to gain access to the debugger.

Table 8.2 Test modules that support the Camel debugger

Maven module	Test support class
camel-test	`org.apache.camel.test.junit4.CamelTestSupport`
camel-test-spring	`org.apache.camel.test.spring.CamelSpringTestSupport`
camel-test-blueprint	`org.apache.camel.test.blueprint.CamelBlueprintTestSupport`
camel-test-karaf	`org.apache.camel.test.karaf.CamelKarafTestSupport`

After you've extended one of the compatible test support classes, you have to override `isUseDebugger` as follows:

```
@Override
public boolean isUseDebugger() {
    return true;
}
```

Next, you have to override one or both of the methods invoked around processor invocations:

```
@Override
protected void debugBefore(Exchange exchange, Processor processor,
        ProcessorDefinition<?> definition, String id, String shortName)
    log.info("MyDebugger: before " + definition + " with body "
```

```
        + exchange.getIn().getBody());
}

@Override
protected void debugAfter(Exchange exchange, Processor processor,
        ProcessorDefinition<?> definition, String id, String label,
        long timeTaken) {
    log.info("MyDebugger: after " + definition + " took " + timeTaken
        + " ms, with body " + exchange.getIn().getBody());
}
```

Now when you run your unit test, `debugBefore` will be invoked before each processor, and `debugAfter` will be invoked after. If you set a breakpoint inside each of these methods, you can inspect what's happening to `Exchange` precisely. You can also run the test case, and these debug methods will provide detailed tracing similar to Camel's tracer.

To try this for yourself, you can run the example in the chapter8/order-router-full directory:

```
mvn clean test -Dtest=OrderRouterDebuggerTest
```

Now you can say that you know how to debug a Camel application! Let's next discuss how to start extending Camel itself by adding your own components.

8.4 Developing custom components

For most integration scenarios, a Camel component is available to help. Sometimes, though, no Camel component is available, and you need to bridge Camel to another transport, API, data format, and so on. You can do this by creating your own custom component.

Creating a Camel component is relatively easy, which may be one of the reasons that custom Camel components frequently show up on other community sites, in addition to the official Camel distribution. In this section, you'll create your own custom component for Camel.

8.4.1 Setting up a new Camel component

One of the first things to do when developing a custom component is to decide what endpoint name to use. This name will be used to reference the custom component in an endpoint URI. You need to make sure this name doesn't conflict with a component that already exists by checking the online component list (http://camel.apache.org/components.html). As with a regular Camel project, you can start creating a new component by using a Maven archetype to generate a skeleton project. To create a new Camel component with `camelinaction.component` as the package name, `custom` as the `artifactId`, `MyComponent` as the component name, and `mycomponent` as the endpoint name, run the following Maven command:

```
mvn archetype:generate \
  -B \
  -DarchetypeGroupId=org.apache.camel.archetypes \
  -DarchetypeArtifactId=camel-archetype-component \
  -DarchetypeVersion=2.20.1 \
  -DgroupId=camelinaction.component \
  -DartifactId=custom \
  -Dname=My \
  -Dscheme=mycomponent \
  -Dversion=1.0-SNAPSHOT
```

This generates a project structure like that shown in the following listing.

Listing 8.2 Layout of a project created by camel-archetype-component

```
custom
├── pom.xml
├── ReadMe.txt
└── src
    ├── data
    ├── main
    │   ├── java
    │   │   └── camelinaction
    │   │       └── component
    │   │           ├── MyComponent.java  (1)
```

Component implementation

```
│  │  ├── MyConsumer.java  (1)
│  │  ├── MyEndpoint.java  (1)
│  │  └── MyProducer.java  (1)
│  └── resources
│      └── META-INF
│          └── services
│              └── org
│                  └── apache
│                      └── camel
│                          └── component
│  └── mycomponent  (2)
```

File that maps URI scheme to component class

```
└── test
    ├── java
    │   └── camelinaction
    │       └── component
    │  └── MyComponentTest.java  (3)
```

Test case for component

```
        └── resources
            └── log4j.properties
```

This is a fully functional Hello World demo component containing a simple consumer that generates dummy messages at regular intervals, and a producer that prints a message to the console. You can run the test case included with this sample component by running the following Maven command:

```
mvn test
```

This project is also available in the chapter8/custom directory of the book's source.

Your component can now be used in a Camel endpoint URI. But you shouldn't stop here. To understand how these classes make up a functioning component, you need to understand the implementation details of each.

8.4.2 Diving into the implementation

The four classes that make up a component in Camel have been mentioned several times before. To recap, it all starts with the `Component` class, which then creates an `Endpoint`. An `Endpoint`, in turn, can create producers and consumers. This is illustrated in figure 8.6.

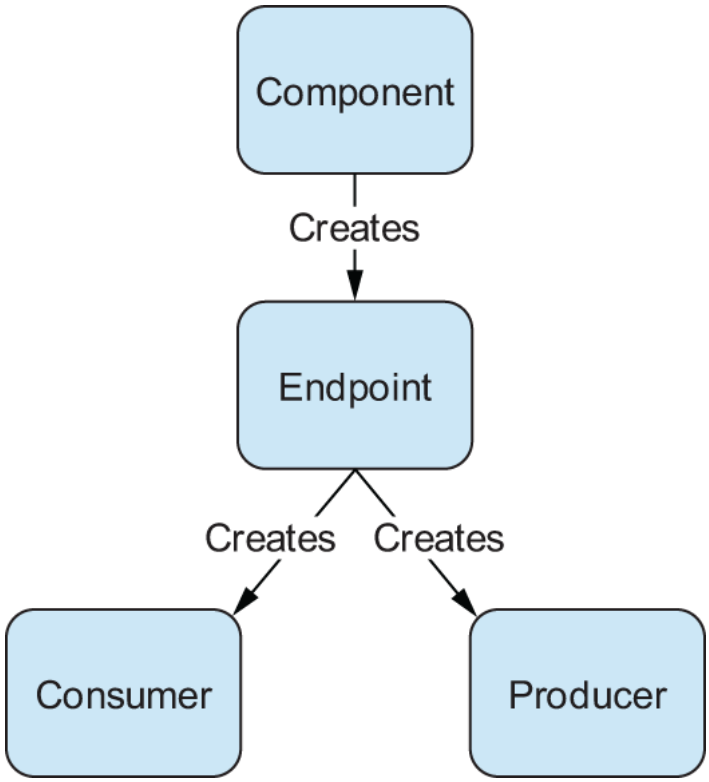

Figure 8.6 A `component` creates an `endpoint`, which then creates `producers` and `consumers`.

You'll first look into the `Component` and `Endpoint` implementations of the custom `MyComponent` component.

Component and Endpoint classes

The first entry point into a Camel component is the class implementing the `Component` interface. A component's main job is to be a factory of new endpoints. It does a bit more than this under the hood, but typically you don't have to worry about these details because they're contained in the `super` class.

The `DefaultComponent` allows you to define the configurable parameters of the endpoint URI via annotations, which we'll touch on when you look at the implementation of `MyEndpoint`. The `MyComponent` class generated by the camel-archetype-component archetype forms a simple and typical component class structure, as shown here:

```
package camelinaction.component;
import java.util.Map;
import org.apache.camel.CamelContext;
import org.apache.camel.Endpoint;
import org.apache.camel.impl.DefaultComponent;
public class MyComponent extends DefaultComponent {
    protected Endpoint createEndpoint(String uri, String remaining,
                                      Map<String, Object> parameters) throws
        Endpoint endpoint = new MyEndpoint(uri, this);
        setProperties(endpoint, parameters);
        return endpoint;
    }
}
```

This class is straightforward, except perhaps for the way in which properties are set with the `setProperties` method. This method takes in the properties set in the endpoint URI string, and for each will invoke a setter method on the endpoint through reflection. For instance, say you used the following endpoint URI:

```
mycomponent:endpointName?prop1=value1&prop2=value2
```

The `setProperties` method, in this case, would try to invoke `setProp1("value1")` and `setProp2("value2")` on the endpoint. Camel will take care of converting those values to the appropriate type.

The endpoint itself is also a relatively simple class, as shown in the following listing.

Listing 8.3 Custom Camel endpoint— `MyEndpoint`

```
package camelinaction.component;

import org.apache.camel.Consumer;
```

```
import org.apache.camel.Processor;
import org.apache.camel.Producer;
import org.apache.camel.impl.DefaultEndpoint;
import org.apache.camel.spi.Metadata;
import org.apache.camel.spi.UriEndpoint;
import org.apache.camel.spi.UriParam;
import org.apache.camel.spi.UriPath;


@UriEndpoint(    1
```

1

Specifies that this endpoint is set up with annotations

```
    firstVersion = "1.0-SNAPSHOT",
    scheme = "mycomponent",
    title = "Custom",
    syntax="mycomponent:name",
    consumerClass = MyConsumer.class,
    label = "custom")
public class MyEndpoint extends DefaultEndpoint {    2
```

2

Extends from default endpoint class

```
    @UriPath @Metadata(required = "true")    3
```

3

Specifies the URI path content

```
    private String name;
    @UriParam(defaultValue = "10")    4
```

4

Specifies that this is a settable option on the endpoint URI

```
    private int option = 10;

    public MyEndpoint() {
    }

    public MyEndpoint(String uri, MyComponent component) {
        super(uri, component);
    }

    public MyEndpoint(String endpointUri) {
        super(endpointUri);
    }

    public Producer createProducer() throws Exception {
        return new MyProducer(this);
```

❺

Creates new producer

```
    }

    public Consumer createConsumer(Processor processor) throws Exception
        return new MyConsumer(this, processor);
```

❻

Creates new consumer

```
    }

    public boolean isSingleton() {
        return true;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
```

```
        return name;
    }

    public void setOption(int option) {
        this.option = option;
    }

    public int getOption() {
        return option;
    }
}
```

The first thing you'll notice is that a lot of setup is handled via annotations on the class and fields. What won't be obvious is that these are mainly used to generate documentation for the Camel components using `camel-package-maven-plugin`. This is also used by tooling (as discussed in chapter 19) to better determine configurable options, URI syntax, grouping of component by labels, and so forth. If you want your component to be widely reused, it's important to add these annotations. But they're not strictly required.

NOTE You can find more information on Camel's various endpoint annotations on the Camel website: http://camel.apache.org/endpoint-annotations.html.

The first annotation used, `@UriEndpoint`, tells Camel to expect that this endpoint is described via annotations ❶. Here you can specify things such as the scheme of the endpoint, its syntax, and any labels that you want to add. Labels are most often used to categorize things in Camel. For example, if this component integrated with a messaging system, you could add the label `messaging`.

As with the `MyComponent` class, you're deriving from a default implementation class from camel-core here too ❷. In this case, you're extending the `DefaultEndpoint` class. It's common when creating a new Camel component to have the `Component`, `Endpoint`, `Consumer`, and `Producer` all derive from default implementations in camel-core. This isn't necessary, but it makes new component development much easier, and you always ben-

efit from the latest improvements to the default implementations without having to code them yourself.

Moving on to the class fields, you have a few more uses of annotations. `@UriPath` specifies the part of the URI between the scheme and the options ❸. Metadata is used to add extra info about an endpoint option, such as whether it's required or has a label associated with it ❹. `@UriParam` specifies that this is a settable option on the endpoint URI. That means with the code in listing 8.3, you can have an option like this:

```
mycomponent:endpointName?option=value1
```

As mentioned in chapter 6, the `Endpoint` class acts as a factory for both consumers and producers. In this example, you're creating both producers ❺ and consumers ❻, which means that this endpoint can be used in a `to` or `from` Java DSL method. Sometimes you may need to create a component that has only a producer or consumer, not both. In that case, it's recommended that you throw an exception, so users know it isn't supported:

```
public Producer createProducer() throws Exception {
    throw new UnsupportedOperationException(
      "You cannot send messages to this endpoint: " + getEndpointUri());
}
```

The real bulk of most components is in the producer and consumer. The `Component` and `Endpoint` classes are mostly designed to fit the component into Camel. In the producers and consumers, which we'll look at next, you have to interface with the remote APIs or marshal data to a particular transport.

Producers and consumers

The producer and consumer are where you get to implement how messages will get on or off a particular transport—in effect, bridging Camel to something else. This is illustrated in figure 8.7.

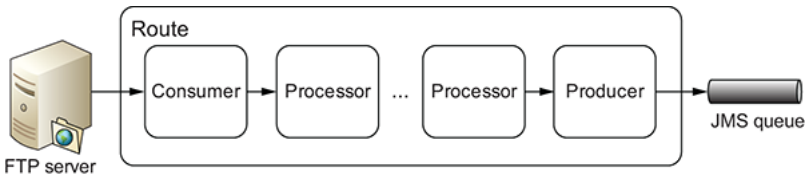

Figure 8.7 A simplified view of a route in which the consumer and producer handle interfacing with external systems. Consumers take messages from an external system into Camel, and producers send messages to external systems.

In your skeleton component project that was generated from an archetype, a producer and consumer are implemented and ready to go. These were instantiated by the `MyEndpoint` class in listing 8.3. The producer, named `MyProducer`, is shown in the following listing.

Listing 8.4 Custom Camel producer—`MyProducer`

```
package camelinaction.component;

import org.apache.camel.Exchange;
import org.apache.camel.impl.DefaultProducer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class MyProducer extends DefaultProducer {    ❶
```

❶ Extends from default producer class

```
    private static final Logger LOG = LoggerFactory.getLogger(MyProducer
    private MyEndpoint endpoint;

    public MyProducer(MyEndpoint endpoint) {
        super(endpoint);
        this.endpoint = endpoint;
    }

    public void process(Exchange exchange) throws Exception {    ❷
```

❷ Serves as entry point to producer

```
        System.out.println(exchange.getIn().getBody());    ❸
```

❸ Prints message body

```
    }

}
```

Like the `Component` and `Endpoint` classes, the producer also extends from a default implementation class from camel-core called `DefaultProducer` ❶. The `Producer` interface extends from the `Processor` interface, so you use a process method ❷. As you can probably guess, a producer is called in the same way as a processor, so the entry point into the producer is the `process` method. The sample component that was created automatically has a basic producer—it just prints the body of the incoming message to the screen ❸. If you were sending to an external system instead of the screen, you'd have to handle a lot more here, such as connecting to a remote system and marshaling data. In the case of data marshaling, it's often a good idea to implement this by using a custom `TypeConverter`, as described in chapter 3, which makes the converters available to other parts of your Camel application.

You can see how messages could be sent out of a route, but how do they get into a route? Consumers, like the `MyConsumer` class generated in your custom component project, get the messages into a route. The `MyConsumer` class is shown in the following listing.

Listing 8.5 Custom Camel consumer—`MyConsumer`

```
package camelinaction.component;

import java.util.Date;

import org.apache.camel.Exchange;
import org.apache.camel.Processor;
import org.apache.camel.impl.ScheduledPollConsumer;
```

```
public class MyConsumer extends ScheduledPollConsumer {    ❶
```

❶

Extends from built-in consumer

```
    private final MyEndpoint endpoint;

    public MyConsumer(MyEndpoint endpoint, Processor processor) {
        super(endpoint, processor);
        this.endpoint = endpoint;
    }

    @Override
    protected int poll() throws Exception {    ❷
```

❷

Is called every 500 ms

```
        Exchange exchange = endpoint.createExchange();

        // create a message body
        Date now = new Date();
        exchange.getIn().setBody("Hello World! The time is " + now);

        try {
            getProcessor().process(exchange);    ❶
```

❸

Sends to next processor in route

```
            return 1; // number of messages polled
        } finally {
            // log exception if an exception occurred and was not handle
            if (exchange.getException() != null) {
                getExceptionHandler().handleException("Error processing
            }
```

```
            }
        }
    }
```

The `Consumer` interface itself doesn't impose many restrictions or give any guidelines as to how a consumer should behave, but the `DefaultConsumer` class does, so it's helpful to extend from this class when implementing your own consumer. In listing 8.5, you extend from a subclass of `DefaultConsumer`, the `ScheduledPollConsumer` ❶. This consumer has a timer thread that will invoke the poll method every 500 milliseconds ❷.

TIP See the discussion of the Scheduler and Quartz components in chapter 6 for more information on creating routes that need to operate on a schedule.

Typically, a consumer will either poll a resource for a message or set up an event-driven structure for accepting messages from remote sources. In this example, you have no remote resource, so you can create an empty exchange and populate it with a Hello World message. A real consumer still would need to do this.

A common pattern for consumers is something like this:

```
Exchange exchange = endpoint.createExchange();
// populate exchange with data
getProcessor().process(exchange);
```

Here you create an empty exchange, populate it with data, and send it to the next processor in the route ❸.

At this point, you should have a good understanding of what's required to create a new Camel component. You may even have a few ideas about what you'd like to bridge Camel to next!

8.5 Generating components with the API component framework

Now you know how to write a component from a Camel API point of view. But what do you do when you start interacting with other transports or APIs? Often the first major decision you have to make is whether you'll have to code all these interactions from scratch or if there's already a suitable library out there that you can reuse. Components in Camel's core module take the first approach if you're looking for inspiration for your implementation. Most of the other component modules take the latter approach and are backed by a third-party library.

Using a third-party library significantly reduces the size and complexity of the component because most of the grunt work is handled in the third-party library. Component development in this case becomes a mundane task. You saw the boilerplate code from a typical component in the previous section; now your task becomes mapping elements of an endpoint URI to method calls in this third-party library.

For larger libraries, this process can be daunting. Take, for example, the camel-box component, which integrates with the box.com file-management service. This component has more than 50 operations accessible from the endpoint URI. That would have been a pain to implement. Dhiraj Bokde, a colleague of your authors, realized we could do so much better. Before drudging through 50-plus operations for the box component, he created tooling called the *API component framework* to generate much of this mapping code. Let's take a look at how this can be done.

8.5.1 Generating the skeleton API project

To generate a skeleton project using the Camel API component framework, you need to use the camel-archetype-api-component archetype. Try this with the following Maven command:

```
mvn archetype:generate \
  -B \
  -DarchetypeGroupId=org.apache.camel.archetypes \
  -DarchetypeArtifactId=camel-archetype-api-component \
  -DarchetypeVersion=2.20.1 \
  -DgroupId=camelinaction \
```

```
    -DartifactId=hi-world \
    -Dname=HiWorld \
    -Dscheme=hiworld
```

The `name` property is used in things such as class names (for example, `HiWorldComponent`), and `scheme` is used to form endpoint URIs (for example, `to("hiworld://...")`). The resultant hi-world directory layout is shown in the following listing.

Listing 8.6 Layout of the project created by camel-archetype-api-component

```
hi-world
├── hi-world-api     (1)
```

(1) Project with sample APIs

```
│   ├── pom.xml
│   └── src
│       └── main
│           └── java
│               └── camelinaction
│                   └── api
│                       ├── HiWorldFileHello.java     (2)
```

(2) Sample API with method signatures defined in file

```
│                       └── HiWorldJavadocHello.java
```

Sample API with method signatures defined in javadoc

```
├── hi-world-component
```

Project containing the Camel component

```
|   ├── pom.xml
|   ├── signatures
|   |   └── file-sig-api.txt ❷
```

Method signatures for API ❷

```
|   └── src
|       ├── main
|       |   ├── java
|       |   |   └── camelinaction
|       |   |       ├── HiWorldComponent.java
|       |   |       ├── HiWorldConfiguration.java
|       |   |       ├── HiWorldConsumer.java
|       |   |       ├── HiWorldEndpoint.java
|       |   |       ├── HiWorldProducer.java
|       |   |       └── internal
|       |   |           ├── HiWorldConstants.java
|       |   |           └── HiWorldPropertiesHelper.java
|       |   └── resources
|       |       └── META-INF
|       |           └── services
|       |               └── org
|       |                   └── apache
|       |                       └── camel
|       |                           └── component
|       |                               └── hiworld
|       └── test
|           ├── java
|           |   └── camelinaction
|           |       └── AbstractHiWorldTestSupport.java
|           └── resources
|               ├── log4j.properties
|               └── test-options.properties
└── pom.xml
```

The generated multimodule Maven project is a bit bigger than the custom component developed in the previous section. This is because it contains

a lot of sample code for you to play with. If you were creating a real API component, you'd most likely need to keep only a portion of the hi-world-component project ❷.

The hi-world-api project ❶ is the sample third-party API that your Camel component ❷ will be using. In a real Camel API component, instead of this sample API project, you'd be pointing the tooling at an already released library. For example, in the camel-box component (part of Apache Camel), a dependency is on a third-party library:

```
    <dependency>
      <groupId>net.box</groupId>
      <artifactId>boxjavalibv2</artifactId>
    </dependency>
```

Instead, your project will depend on the sample API project:

```
    <dependency>
      <groupId>camelinaction</groupId>
      <artifactId>hi-world-api</artifactId>
    </dependency>
```

Let's look at how to generate a working Camel component from this API.

8.5.2 Configuring the camel-api-component-maven-plugin

In listing 8.6, you saw that two projects were listed: a sample API and a Camel component using that sample API. Looking at the pom.xml for the Camel component, you first have a dependency on the sample API, as follows:

```
<dependency>
  <groupId>camelinaction</groupId>
  <artifactId>hi-world-api</artifactId>
</dependency>
<dependency>
  <groupId>camelinaction</groupId>
  <artifactId>hi-world-api</artifactId>
  <classifier>javadoc</classifier>
```

```
    <scope>provided</scope>
  </dependency>
```

Now, you may be wondering why we included a Javadoc JAR in addition to the main JAR. It comes down to a lack of information when interrogating the API JAR via reflection. Method signatures are maintained, but parameter names aren't. In order to get the full set of names (method and parameter names), you have to fill in the blanks for the tooling with either a signature file or Javadoc. Fortunately, Javadoc is a commonly available resource for Java APIs, so this will likely be the easiest option. The following listing shows the minimal camel-api-component-maven-plugin configuration for using both a signature file and Javadoc to supply parameter names.

Listing 8.7 Minimal configuration for the camel-api-component-maven-plugin

```
<plugin>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-api-component-maven-plugin</artifactId>
  <executions>
    <execution>
      <id>generate-test-component-classes</id>
      <goals>
        <goal>fromApis</goal>
      </goals>
      <configuration>
        <apis>
          <api>
      <apiName>hello-file</apiName>    (1)
```

(1) API name for file-based signature

```
      <proxyClass>camelinaction.api.HiWorldFileHello</proxyClass>
```

Class to interrogate using reflection

```
        <fromSignatureFile>signatures/file-sig-api.txt</fromSignatureFile>    ❸
```

❸ Location of text file with method signatures for class

```
          </api>
          <api>
        <apiName>hello-javadoc</apiName>    ❷
```

❷ API name for Javadoc-based signature

```
        <proxyClass>camelinaction.api.HiWorldJavadocHello</proxyClass>
```

Class to interrogate using reflection

```
        <fromJavadoc/>    ❹
```

❹ Specify to check for method signatures in javadoc for class defined in the preceding line

```
          </api>
        </apis>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Here you can see that the Maven plugin is configured to generate an end-point URI to the Java API mapping for two classes ❶ ❷. Let's first take a look at the file-supplemented API `camelinaction.api.HiWorldFileHello`, shown in the following listing.

Listing 8.8 Sample HiWorld API

```
package camelinaction.api;

/**
 * Sample API used by HiWorld Component whose method signatures are read
 */
public class HiWorldFileHello {

    public String sayHi() {
        return "Hello!";
    }

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

    public String greetUs(String name1, String name2) {
        return "Hello " + name1 + ", " + name2;
    }
}
```

When the Maven plugin interrogates `HiWorldFileHello`, it finds three methods and determines that three sub-URI schemes will be needed:

```
hiworld://hello-file/greetMe?inBody=name
hiworld://hello-file/greetUs
hiworld://hello-file/sayHi
```

The `hiworld` portion of the URI comes from the `-Dscheme=hiworld` argument passed into the Maven archetype plugin in section 8.4.1. The `hello-file` portion of the URI comes from ❶ in listing 8.7. The last three parts (`greetMe`, `greetUs`, and `sayHi`) come from the methods in the `HiWorldFileHello` class.

Input data is handled differently, depending on the number of parameters. For methods with just one parameter, the full message body is often used as the data. To set this, you use the `inBody` URI option and specify the parameter name you want the message body to map to. A URI like this

```
hiworld://hello-file/greetMe?inBody=name
```

will be calling the following:

```
HiWorldFileHello.greetMe(/* Camel message body */)
```

For methods with more than one parameter, each parameter is mapped to a message header. For the `greetUs` method, you'd set up headers like this:

```
final Map<String, Object> headers = new HashMap<String, Object>();
headers.put("CamelHiWorld.name1", /* Data for name1 parameter here */);
headers.put("CamelHiWorld.name2", /* Data for name2 parameter here */);
```

The header names are generated as follows:

- `Camel` is always used for the prefix.
- The `HiWorld` portion of the URI comes from the `-Dname=HiWorld` argument passed into the Maven archetype plugin in section 8.4.1.
- Finally, there's a dot followed by the parameter name.

Recall that for this API, you're using a signature file to provide the parameter names. We specified this in listing 8.7 ❸. The simple signature file signatures/file-sig-api.txt is shown here:

```
public String sayHi();
public String greetMe(String name);
public String greetUs(String name1, String name2);
```

Without this file, you'd have only parameter names such as `arg0`, `arg1`, and so forth.

This process is even easier when you have Javadoc available for the third-party API. You tried the Javadoc way of providing parameter names in listing 8.7 ❹. The proxy class that you're using, `camelinaction.api.HiWorldJavadocHello`, is essentially the same as `camelinaction.api.HiWorldFileHello` shown in listing 8.8. It produces three sub-URI schemes like this:

```
hiworld://hello-javadoc/greetMe?inBody=name
hiworld://hello-javadoc/greetUs
hiworld://hello-javadoc/sayHi
```

These endpoints are used in the same way as the file-supplemented ones from before.

For these simple handcrafted APIs, you don't need to change much in the conversion to Camel endpoints. For larger APIs, though, customization is a likely requirement. A ton of customization options are available to address this, so let's go over them.

8.5.3 Setting advanced configuration options

At times you may need to customize the mapping from a third-party API to a Camel endpoint URI. For instance, some API methods may not make sense to call on their own, or maybe you want to keep the scope of your component in check. Perhaps the naming used on the remote API also has conflicts with other names you want to use in your component. Or perhaps you think you can name things better.

Let's look at some of the advanced configuration options you have available for both file- and Javadoc-supplemented API components.

Global configuration options

You can apply six configuration options to both file- and javadoc-supplemented API components as well:

- `substitutions` —Substitutions are used to modify parameter names. This could be useful to avoid name clashes or if you feel you can provide a more descriptive name for your Camel component versus what was provided in the third-party API. For example, in our API the `greetMe` method has a single parameter called `name`. Perhaps it would be more useful in your endpoint URI to have this as `username` if it corresponds to a system username:

```
<plugin>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-api-component-maven-plugin</artifactId>
  <executions>
```

```
      <execution>
        <id>generate-test-component-classes</id>
        <goals>
          <goal>fromApis</goal>
        </goals>
        <configuration>
          <apis>
          ...
          </apis>
          <substitutions>
            <substitution>
              <method>^greetMe$</method>
              <argName>^(.+)$</argName>
              <replacement>user$1</replacement>
            </substitution>
          </substitutions>
        </configuration>
      </execution>
    </executions>
  </plugin>
```

The `method` and `argName` elements are regular expressions. You can refer to regular expression group matches in the `replacement` element as `$1`, `$2`, and so forth. You can also use a regular expression to grab text out of the parameter type. To do this, you set the `replaceWithType` element to `true` and specify a regular expression in the `argType` element. The following example changes the `greetMe` method `name` parameter to `usernameString`:

```
  <substitutions>
    <substitution>
      <method>^greetMe$</method>
      <argName>^name$</argName>
      <argType>^java.lang.(.+)$</argType>
      <replacement>username$1</replacement>
      <replaceWithType>true</replaceWithType>
    </substitution>
  </substitutions>
```

- `aliases`—Method aliases are useful for creating shorthand endpoint URI option names for long-winded method parameter names. For example, let's provide shorter names for your `greetMe`/`Us` methods:

```
<plugin>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-api-component-maven-plugin</artifactId>
  <executions>
    <execution>
      <id>generate-test-component-classes</id>
      <goals>
        <goal>fromApis</goal>
      </goals>
      <configuration>
        <apis>
        ...
        </apis>
        <aliases>
          <alias>
            <methodPattern>greet(.+)</methodPattern>
            <methodAlias>$1</methodAlias>
          </alias>
        </aliases>
      </configuration>
    </execution>
  </executions>
</plugin>
```

The `methodPattern` element is a regular expression that matches method names. Any groups captured from the method names are available in the element as `$1`, `$2`, and so on. In this case, you've made the following two URIs equivalent:

```
hiworld://hello-javadoc/greetUs
hiworld://hello-javadoc/us
```

And you've made the following two equivalent:

```
hiworld://hello-javadoc/greetMe?inBody=name
hiworld://hello-javadoc/me?inBody=name
```

- `nullableOptions`—By specifying parameters as nullable, if you neglect to set a corresponding header or message body, a null will be passed into the third-party API. It's useful to be able to do this because null can be a valid parameter value. Take the following example:

```
<plugin>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-api-component-maven-plugin</artifactId>
  <executions>
    <execution>
      <id>generate-test-component-classes</id>
      <goals>
        <goal>fromApis</goal>
      </goals>
      <configuration>
        <apis>
        ...
        </apis>
        <nullableOptions>
          <nullableOption>name</nullableOption>
        </nullableOptions>
      </configuration>
    </execution>
  </executions>
</plugin>
```

With this configuration, any `name` parameters will be allowed to be `null`.

- `excludeConfigNames`—Exclude any parameters with a name matching the specified regular expression.
- `excludeConfigTypes`—Exclude any parameters with a type matching the specified regular expression.
- `extraOptions`—Add extra options to your endpoint URI that weren't in the third-party API. For example, let's add a `language` option:

```
<plugin>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-api-component-maven-plugin</artifactId>
  <executions>
    <execution>
      <id>generate-test-component-classes</id>
      <goals>
        <goal>fromApis</goal>
      </goals>
      <configuration>
        <apis>
```

```
          ...
          </apis>
          <extraOptions>
            <extraOption>
              <type>java.lang.String</type>
              <name>language</name>
            </extraOption>
          </extraOptions>
        </configuration>
      </execution>
    </executions>
  </plugin>
```

Javadoc-only configuration options

Some additional options are available only within the `fromJavadoc` element. Let's take a look at them:

- `excludePackages`—Exclude methods from proxy classes with a specified package name. Useful for filtering out methods from unwanted superclasses. Methods from any classes in the `java.lang` package are excluded by default:

```
  <plugin>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-api-component-maven-plugin</artifactId>
    <executions>
      <execution>
        <id>generate-test-component-classes</id>
        <goals>
          <goal>fromApis</goal>
        </goals>
        <configuration>
          <apis>
            <api>
              <apiName>hello-javadoc</apiName>
              <proxyClass>camelinaction.api.HiWorldJavadocHello
              </proxyClass>
              <fromJavadoc>
                <excludePackages>package.name.to.exclude</excludePackage
              </fromJavadoc>
            </api>
```

```
          </apis>
        </configuration>
      </execution>
    </executions>
  </plugin>
```

- `excludeClasses` —Exclude classes (unwanted superclasses) matching a specified name:

```
  <fromJavadoc>
    <excludeClasses>SomeUnwantedAbstractSuperClass</excludeClasses>
  </fromJavadoc>
```

- `includeMethods` —Only methods matching this pattern will be included:

```
  <fromJavadoc>
    <includeMethods>greetMe</includeMethods>
  </fromJavadoc>
```

  Include only the `greetMe` method in processing.
- `excludeMethods` —Any methods matching this pattern won't be included:

```
  <fromJavadoc>
    <excludeMethods>greetMe</excludeMethods>
  </fromJavadoc>
```

  Don't include the `greetMe` method in processing.
- `includeStaticMethods` —Also include static methods. This is `false` by default:

```
  <fromJavadoc>
    <includeStaticMethods>true<includeStaticMethods>
  </fromJavadoc>
```

8.5.4 Implementing remaining functionality

With the sample component generated from camel-archetype-api-component, there isn't much left to do. But if you were running this against a

real third-party API, you'd certainly need to fill in some blanks.

Setting up unit tests

A good first step would be to copy the generated unit tests into the project test source directory. The unit tests are generated by `camel-api-component-maven-plugin` when you invoke `mvn install` and if there isn't already a test case in the src/test/java/packagename directory. In this case, you have a test case for the Javadoc- and file-supplemented APIs in the target/generated-test-sources/camel-component/camelinaction directory:

- HiWorldFileHelloIntegrationTest.java
- HiWorldJavadocHelloIntegrationTest.java

Both of these need to be copied into src/test/java/camelinaction so your edits won't be lost the next time the plugin is run.

These test cases don't do much as is. They mainly set up sample routes and show how parameter data is passed into the various APIs. Your job is to fill in the data with something useful.

For the file-supplemented API, your routes look like this:

```
@Override
protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        public void configure() {
            // test route for greetMe
            from("direct://GREETME")
                .to("hiworld://" + PATH_PREFIX + "/greetMe?inBody=name")

            // test route for greetUs
            from("direct://GREETUS")
                .to("hiworld://" + PATH_PREFIX + "/greetUs");

            // test route for sayHi
            from("direct://SAYHI")
                .to("hiworld://" + PATH_PREFIX + "/sayHi");

        }
    };
}
```

Notice that there isn't a route with a `hi-world` consumer—no `from("hiworld...)`—you'll have to fill that in yourself if the component will support it. A test case is created for each API method. For the `greetUs` method, the test case looks like this:

```
@Ignore
@Test
public void testGreetUs() throws Exception {
    final Map<String, Object> headers = new HashMap<String, Object>();
    // parameter type is String
    headers.put("CamelHiWorld.name1", null);
    // parameter type is String
    headers.put("CamelHiWorld.name2", null);

    final String result
        = requestBodyAndHeaders("direct://GREETUS", null, headers);

    assertNotNull("greetUs result", result);
    LOG.debug("greetUs: " + result);
}
```

There isn't much to this test case, but it's handy because it shows you how many parameters to set as well as what their names are. When you've filled in some data and additional asserts, you can remove the `@Ignore` annotation.

Filling in the other blanks

Now that you've seen how the tests are set up, let's look at the component. Recall that the sample `hi-world-component` contains several source files under src/main/java/camelinaction:

```
hi-world-component
├── pom.xml
└── src
    └── main
        └── java
            └── camelinaction
                ├── HiWorldComponent.java
                ├── HiWorldConfiguration.java
                ├── HiWorldConsumer.java
```

```
                    ├── HiWorldEndpoint.java
                    └── HiWorldProducer.java
```

Many of these classes have methods you can override or tweak for your own component. Most are optional, however. A good place to start is `HiWorldEndpoint`, and in particular the `afterConfigureProperties` method:

```
@Override
protected void afterConfigureProperties() {
    switch (apiName) {
        case HELLO_FILE:
            apiProxy = new HiWorldFileHello();
            break;
        case HELLO_JAVADOC:
            apiProxy = new HiWorldJavadocHello();
            break;
        default:
            throw new IllegalArgumentException("Invalid API name " + api
    }
}
```

Here you set `apiProxy` to the proxy object for the particular API being used in the current endpoint. This switch block matches up with the configuration for the `camel-api-component-maven-plugin`:

```
<api>
  <apiName>hello-file</apiName>
  <proxyClass>camelinaction.api.HiWorldFileHello</proxyClass>
  <fromSignatureFile>signatures/file-sig-api.txt</fromSignatureFile>
</api>
<api>
  <apiName>hello-javadoc</apiName>
  <proxyClass>camelinaction.api.HiWorldJavadocHello</proxyClass>
  <fromJavadoc/>
</api>
```

If this matches up, what's left to do? In many cases, the Java library containing this proxy class needs to be configured. This could include things like authentication for a remote server, opening a session, or setting con-

nection properties. These things often differ for each third-party library, so the API component framework can't figure this out for you.

If you still feel totally lost with how to proceed after the framework is done with its generation work, you may find inspiration from existing components supported by the API component framework. These are as follows:

- camel-box
- camel-braintree
- camel-google-calendar
- camel-google-drive
- camel-google-pubsub
- camel-google-mail
- camel-linkedin
- camel-olingo2
- camel-olingo4
- camel-twilio
- camel-zendesk

By now you should have a good understanding of how to write new components for Apache Camel. Hopefully, you'll also consider contributing them back to the project! (Appendix B covers contributing to Apache Camel.) Let's now consider another extension point in Camel: data formats.

8.6 Developing data formats

As you saw in chapter 3, *data formats* are pluggable transformers that can transform messages from one form to another, and vice versa. Each data format is represented in Camel as an interface in `org.apace.camel.spi.DataFormat` containing two methods:

- `marshal`—For marshaling a message into another form, such as marshaling Java objects to XML, CSV, EDI, HL7, JSON or other well-known data models
- `unmarshal`—For performing the reverse operation, which turns data from well-known formats back into a message

Camel has many data formats, but at times you might need to write your own. In this section, you'll look at how to develop a data format that can reverse strings. Let's first set up the initial boilerplate code using an archetype.

8.6.1 Generating the skeleton data format project

To generate a skeleton project by using the Camel API component framework, you need to use the camel-archetype-dataformat archetype. Try this with following Maven command:

```
mvn archetype:generate \
  -B \
  -DarchetypeGroupId=org.apache.camel.archetypes \
  -DarchetypeArtifactId=camel-archetype-dataformat \
  -DarchetypeVersion=2.20.1 \
  -DgroupId=camelinaction \
  -DartifactId=reverse-dataformat \
  -Dname=Reverse \
  -Dscheme=reverse
```

The resulting reverse-dataformat directory layout is shown in the following listing.

Listing 8.9 Layout of the project created by camel-archetype-api-component

```
reverse-dataformat
├── pom.xml
├── ReadMe.txt
└── src
    ├── main
    │   ├── java
    │   │   └── camelinaction
    │   │       └── ReverseDataFormat.java    (1)
```

(1) Skeleton data format class

```
    │   └── resources
    │       └── META-INF
    │           └── services
    │               └── org
    │                   └── apache
    │                       └── camel
    │                           └── dataformat
    │                               └── reverse ❷
```

❷ Ensure custom data format is registered with “reverse” name in Camel

```
    └── test
        ├── java
        │   └── camelinaction
        │       └── ReverseDataFormatTest.java ❸
```

❸ Test case for data format

```
        └── resources
            └── log4j.properties
```

This data format is usable right away and even has a working test case. But it doesn’t do much—it just returns the data unmodified. Let’s see how to make your data format change the data.

8.6.2 Writing the custom data format

Developing your own data format is fairly easy, because Camel provides a single API you must implement: `org.apache.camel.spi.DataFormat`. Let’s look at how to implement a string-reversing data format, shown in the following listing.

Listing 8.10 Developing a custom data format that can reverse strings

```
public class ReverseDataFormat
    extends ServiceSupport
```

```
    implements DataFormat, DataFormatName {

    public String getDataFormatName() {
        return "reverse";
    }

    public void marshal(Exchange exchange, Object graph,
        OutputStream stream) throws Exception {    (1)
```

(1) Marshals to reverse string

```
        byte[] bytes
            = exchange.getContext().getTypeConverter().mandatoryConvertT
            byte[].class, graph);
        String body = reverseBytes(bytes);
        stream.write(body.getBytes());
    }

    public Object unmarshal(Exchange exchange, InputStream stream)
        throws Exception {    (2)
```

(2) Unmarshals to unreverse string

```
        byte[] bytes = exchange.getContext().getTypeConverter().        m
        String body = reverseBytes(bytes);
        return body;
    }

    private String reverseBytes(byte[] data) {
        StringBuilder sb = new StringBuilder(data.length);
        for (int i = data.length - 1; i >= 0; i--) {
            char ch = (char) data[i];
            sb.append(ch);
        }
        return sb.toString();
    }
```

```
    @Override
    protected void doStart() throws Exception {
        // init logic here
    }

    @Override
    protected void doStop() throws Exception {
        // cleanup logic here
    }

}
```

The custom data format must implement the `DataFormat` interface, which forces you to develop two methods: `marshal` and `unmarshal`. That's no surprise, because they're the same methods you use in the route. The `marshal` method ❶ needs to output the result to `OutputStream`. To do that, you need to get the message payload as a `byte[]` and then reverse it with a helper method. Then you write that data to `OutputStream`. Note that you use the Camel type converters to return the message payload as a `byte[]`. This is powerful and saves you from doing a manual typecast in Java or trying to convert the payload yourself.

The `unmarshal` method ❷ is nearly the same. You use the Camel type-converter mechanism again to provide the message payload as a `byte[]`. `unmarshal` also reverses the bytes to get the data back in its original order. Note that in this method you return the data instead of writing it to a stream.

TIP As a best practice, use the Camel type converters instead of type-casting or converting between types yourself. Chapter 3 covers Camel's type converters.

To use this new data format in a route, all you have to do is define it as a bean and refer to it by using `<custom>` as follows:

```
<bean id="reverse" class="camelinaction.ReverseDataFormat"/>

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

```
    <route>
        <from uri="direct:marshal"/>
        <marshal>
            <custom ref="reverse"/>
        </marshal>
        <to uri="log:marshal"/>
    </route>

    <route>
        <from uri="direct:unmarshal"/>
        <unmarshal>
            <custom ref="reverse"/>
        </unmarshal>
        <to uri="log:unmarshal"/>
    </route>
</camelContext>
```

Using the Java DSL looks like this:

```
from("direct:marshal")
    .marshal().custom("reverse")
    .to("log:marshal");
from("direct:unmarshal")
    .unmarshal().custom("reverse")
    .to("log:unmarshal");
```

Alternatively, you can pass in an instance of the data format directly:

```
DataFormat format = new ReverseDataFormat();
from("direct:in").marshal(format);
from("direct:back").unmarshal(format).to("mock:reverse");
```

You'll find this example in the chapter8/reverse-dataformat directory, and you can try it by using the following Maven goal:

```
mvn test
```

At this point, you should have a good idea of what it takes to create a new Camel data format.

8.7 Summary and best practices

Knowing how to create Camel projects is important, and you may have wondered why we chose to discuss this so late in the book. We felt it was best to focus on the core concepts first and worry about project setup details later. Also, you should now have a better idea of the cool Camel applications you can create, having read about the features first. At this point, you should be well equipped to start your own Camel application and make it do useful work.

Before we move on, here are the key ideas to take away from this chapter:

- *The easiest way to create Camel applications is with Maven archetypes*—Nothing is worse than having to type out a bunch of boilerplate code for new projects. The Maven archetypes provided by Camel and the Spring Boot start website (http://start.spring.io) will get you started much faster.
- *The easiest way to manage Camel library dependencies is with Maven archetypes*—Camel is just a Java framework, so you can use whatever build system you like to develop your Camel projects. Using Maven will eliminate many of the hassles of tracking down JAR files from remote repos, letting you focus more on your business code than the library dependencies.
- *IDEs like Eclipse or IDEA make Camel development easier*—From autocompletion of DSL methods to great Maven integration, developing Camel projects is a lot easier within an IDE.
- *Camel has features that assist you when debugging*—You don't always have to dive deep into the Camel source in an IDE to solve bugs. Camel provides a great tracing facility, many JMX operations and attributes, and logging. If you're in a test case, Camel also provides a debugger facility that allows you to step through each node in the runtime processor graph. You can even take the easy road and use a tooling project such as hawtio or JBoss Fuse Tooling to visually debug your routes.
- *If you find no component in Camel for your use case, create your own*—Camel allows you to write and load up your own custom components easily. There's even a Maven archetype for starting a custom component project.

- *Consider using the API component framework when creating new components*—If you need to create a new component using an existing third-party library, the API component framework can generate much of the component for you.
- *If you find no data format in Camel for your use case, create your own*—As for components, Camel allows you to write and load your own custom data formats easily. There's also a Maven archetype for starting a custom data-format project.

In the next chapter, we'll look at a topic that can help make you a successful integration specialist: testing with Camel. Without it, you'll almost certainly be in trouble. We'll also look at simulating errors to test whether your error-handling strategies work as expected.