15

Running and deploying Camel

This chapter covers

- Starting and stopping Camel safely
- Adding and removing routes at runtime
- Deploying Camel
- Running standalone
- Running in web containers
- Running in Java EE servers
- Running with OSGi
- Running with CDI

In chapter 14, you learned all about securing Camel. We'll now shift focus to another topic that's important to master: running and deploying Camel applications.

We'll begin with starting Camel. You need to fully understand how to start, run, and shut down Camel reliably and safely, which is imperative in a production environment. We'll also review various options you can use to tweak the way Camel and routes are started. We'll continue on this path, looking at how to dynamically start and stop routes at runtime. Your applications won't run forever, so we'll spend time focusing on how to shut down Camel safely.

The other part of this chapter covers various strategies for deploying Camel. We'll take a look at five common runtime environments supported by Camel. Two other popular runtimes, Spring Boot and WildFly Swarm, are covered in chapter 7, where we discuss microservices.

As we discuss these topics, we'll work through an example involving Rider Auto Parts: you've been asked to help move a recently developed application safely into production. The application receives inventory updates from suppliers, provided via a web service or files. Figure 15.1 shows a high-level diagram of the application.

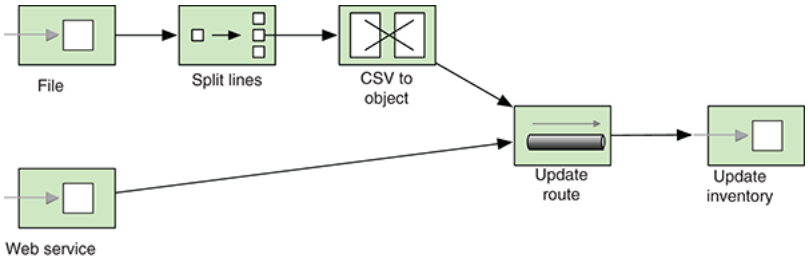


Figure 15.1 A Rider Auto Parts application accepting incoming inventory updates from either files or a web service

15.1 Starting Camel

In chapter 1, you learned how to download, install, and run Camel. That works well in development, but the game plan changes when you take an application into production.

Starting up a Camel application in production is harder than you might think because the route order may have to be arranged in a certain way to ensure a reliable startup. It's critical that the operations staff can safely manage the application in their production environment.

Let's look at how Camel starts.

15.1.1 How Camel starts

Camel doesn't start magically by itself. Often it's the server (container) that Camel is running inside that invokes the `start` method on `CamelContext`, starting up Camel. This is also what you saw in chapter 1, where you used Camel inside a standalone Java application. A standalone Java application isn't the only deployment choice; you can also run Camel inside a container such as Spring, CDI, or OSGi.

Regardless of which container you use, the same principle applies: the container must prepare and create an instance of `CamelContext` up front, before Camel can be started, as illustrated in figure 15.2.

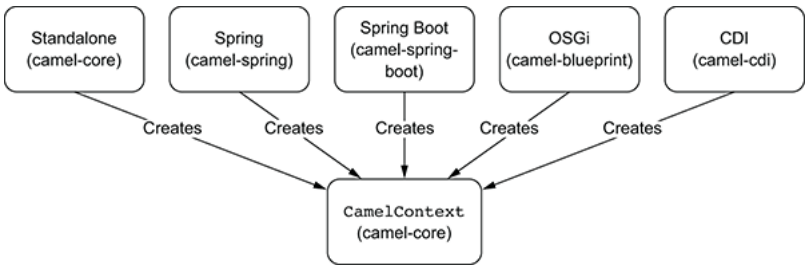


Figure 15.2 Using Camel with containers often requires the container in question to prepare and create `CamelContext` up front before it can be started.

Because Spring is a common container, we'll outline how Spring and Camel work together to prepare `CamelContext`.

Preparing CamelContext in a Spring container

Spring allows third-party frameworks to integrate seamlessly with Spring. To do this, the third-party frameworks must provide `org.springframework.beans.factory.xml. NamespaceHandler`, which is the extension point for using custom namespaces in Spring XML files. Camel provides `CamelNamespaceHandler`.

When using Camel in the Spring XML file, you define the `<camelContext>` tag as follows:

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

The `http://camel.apache.org/schema/spring` namespace is the Camel custom namespace. To let Spring know about this custom namespace, Camel provides a `META-INF/spring.handlers` file, where the namespace is mapped to the class implementation:

```
http\://camel.apache.org/schema/spring=org.apache.camel.spring.handler.C
```

`CamelNamespaceHandler` is then responsible for parsing the XML and delegating to other factories for further processing. One of these factories is `CamelContextFactoryBean`, which is responsible for creating the `CamelContext` that essentially is your Camel application.

PREPARING CAMELCONTEXT IN SPRING BOOT

When Camel is used with Spring Boot, then Camel is prepared using Spring Boot *auto configuration*, which is the standard way on Spring Boot. This is implemented in the `org.apache.camel.spring.boot.CamelAutoConfiguration` class in the camel-spring-boot JAR.

When Spring is finished initializing, it signals to third-party frameworks that they can start by broadcasting the `ContextRefreshedEvent` event.

Starting CamelContext

At this point, `CamelContext` is ready to be started. What happens next is the same regardless of which container or deployment option you're using with Camel. Figure 15.3 shows a flow diagram of the startup process.

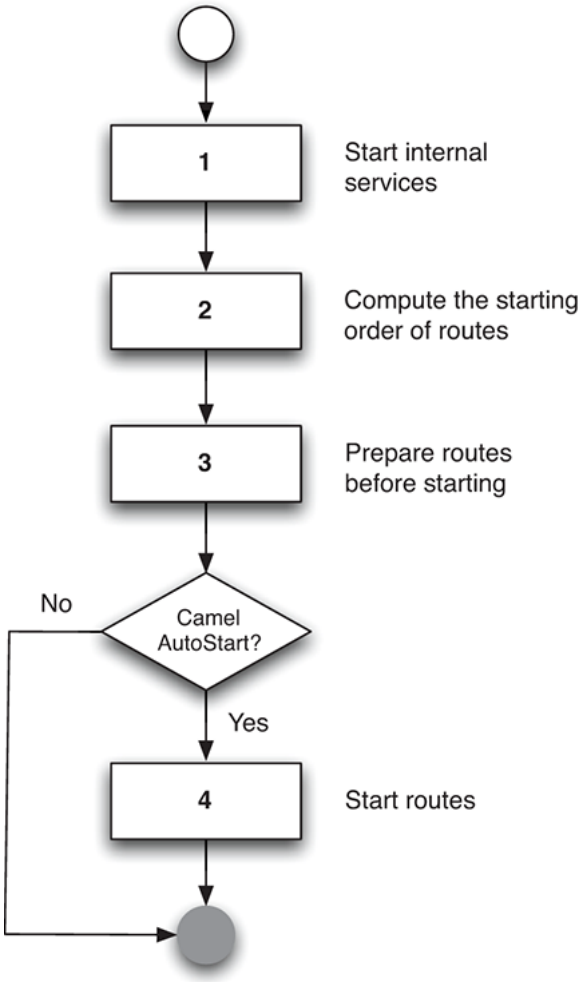


Figure 15.3 Flow diagram showing how Camel starts by starting internal services, computing the starting order of routes, and preparing and starting the routes

`CamelContext` is started by invoking its `start` method. Figure 15.3 shows the following four steps:

1. *Start internal services*—Prepares and starts internal services used by Camel, such as the type-converter mechanism.
2. *Compute starting order*—Computes the order in which the routes should be started. By default, Camel starts up all the routes in the order they're defined in the XML files or the Java `RouteBuilder` classes. Section 15.1.3 covers how to configure the order of routes.
3. *Prepare routes*—Prepares the routes before they're started.
4. *Start routes*—If `AutoStartup` is `true`, starts the routes by starting the consumers, which opens the gates to Camel and lets the messages start to flow in.

After step 4, Camel writes a message to the log indicating that it's been started and that the startup process is complete. In some cases, you may need to influence how Camel is started, so let's look at that now.

15.1.2 Camel startup options

Camel offers various startup options. For example, you may have a maintenance route that shouldn't be autostarted on startup. You may also want to enable tracing on startup to let Camel log traces of messages being routed.

Table 15.1 lists all the options that influence startup. These options can be divided into two kinds. The first four are related to startup and shutdown, and the remainder are miscellaneous options. We'll explain the miscellaneous options first and then turn our attention to the startup and shutdown options.

Table 15.1 Camel startup options

Option	Description
`AutoStartup`	Indicates whether the route should be started automatically when Camel starts. This option is enabled by default.
`StartupOrder`	Dictates the order in which the routes should be started when Camel starts. We cover this in section 15.1.3.
`ShutdownRoute`	Configures whether the route in question should stop immediately or defer while Camel is shutting down. We cover shutdown in section 15.3.
`ShutdownRunningTask`	Controls whether Camel should continue to complete pending running tasks at shutdown or stop immediately after the current task is complete. We cover shutdown in section 15.3.
`Tracing`	Traces how an exchange is being routed within that particular route. This option is disabled by default. Tracing is covered in chapter 16.
`Delayer`	Sets a delay in milliseconds that slows the processing of a message. You can use this during debugging to reduce how quickly Camel routes messages, which may help you track what happens when you watch the logs. This option is disabled by default.

Option	Description
`MessageHistory`	Stores the history of an exchange as it's being routed within a particular route. This option is enabled by default. Chapter 11, section 11.3.3 has sample output of the MessageHistory option in action.
`HandleFault`	Turns fault messages into exceptions. This isn't a typical thing to do in a pure Camel application, but when using SOAP faults, you'll need to set this option to let the Camel error handler react to faults. This option is disabled by default. We cover this in more detail shortly.
`StreamCaching`	Caches streams that otherwise couldn't be accessed multiple times. You may want to use this when you use redelivery during error handling, which requires being able to read the stream multiple times. This option is disabled by default. We cover this in more detail shortly.
`AllowUseOriginalMessage`	Tells Camel to make the original message available in error handlers. This option is enabled by default. If you don't need access to the original message in your error handlers, consider turning this option off; it may improve performance.

Option	Description
`LogExhaustedMessageBody`	Sets whether to log the full message body in the message history.
`LogMask`	Determines whether logging (log component, Log EIP, tracer, and so on) should mask sensitive information such as passwords. This option is disabled by default.

Configuring StreamCaching

The miscellaneous options, such as the `Tracing` option, are often used during development to turn on additional logging, which we cover in detail in the next chapter. Or you may need to turn on stream caching if you use Camel with other stream-centric systems. For example, to enable stream caching, you can do the following with the Java DSL:

```
public class MyRoute extends RouteBuilder {
    public void configure() throws Exception {
        context.setStreamCaching(true);
        from("cxf:bean:inventoryEndpoint?dataFormat=PAYLOAD")
        ...
    }
}
```

The same example using XML DSL looks like this:

```
<camelContext streamCache="true"
                xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="cxf:bean:inventoryEndpoint?dataFormat=PAYLOAD"/>
        ...
    </route>
</camelContext>
```

All the options from table 15.1 can be scoped at either the context or route level. The preceding stream cache example was scoped at the con-

text level. You could also configure it on a particular route:

```
public class MyRoute extends RouteBuilder {
    public void configure() throws Exception {
        from("cxf:bean:inventoryEndpoint?dataFormat=PAYLOAD")
            .streamCaching()
            ...
    }
}
```

You can configure route-scoped stream caching in XML as follows:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route streamCache="true">
        <from uri="cxf:bean:inventoryEndpoint?dataFormat=PAYLOAD"/>
        ...
    </route>
</camelContext>
```

NOTE Java DSL uses the syntax `no`*XXX* to disable an option, such as `noStreamCaching` or `noTracing`.

You need to know one last detail about the context and route scopes. The context scope is used as a fallback if a route doesn't have a specific configuration. The idea is that you can configure the default setting on the context scope and then override when needed at the route scope. For example, you could enable tracing on the context scope and then disable it on the routes you don't want traced.

Configuring HandleFault

In the preceding example, you routed messages using the CXF component. The `HandleFault` option is used to control whether Camel error handling should react to SOAP faults.

Suppose sending to the `cxf:bean:inventoryEndpoint?dataFormat=PAYLOAD` endpoint fails with a fault. Without `HandleFault` enabled, the fault would be propagated back to the consumer. By en-

abling `HandleFault`, you can let the Camel error handler react when faults occur.

The following code shows how to let the `DeadLetterChannel` error handler save failed messages to files in the error directory:

```
public class MyRoute extends RouteBuilder {
    public void configure() throws Exception {
        errorHandler(deadLetterChannel("file:errors"));
        from("direct:start")
            .streamCaching().handleFault()
            .to("cxf:bean:inventoryEndpoint?dataFormat=PAYLOAD");
    }
}
```

The equivalent example in XML is as follows:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <errorHandler id="EH" type="DeadLetterChannel"
                  deadLetterUri="file:errors"/>
    <route streamCache="true" errorHandlerRef="EH" handleFault="true">
        <from uri="direct:start"/>
        <to uri="cxf:bean:inventoryEndpoint?dataFormat=PAYLOAD"/>
    </route>
</camelContext>
```

We'll now look at how to control the ordering of routes.

15.1.3 Ordering routes

The order in which routes are started and stopped can become important when they're highly interdependent. For example, you may have reusable routes that must be started before being used by other routes. Also, routes that immediately consume messages that are bound for other routes may have to be started later to ensure that the other routes are ready in time.

To control the startup order of routes, Camel provides two options: `AutoStartup` and `StartupOrder`. The former dictates whether the routes should be started. The latter is a number that dictates the order in which the routes should be started.

Using StartupOrder to control ordering of routes

Let's return to our Rider Auto Parts example, outlined at the beginning of the chapter. Figure 15.4 shows the high-level diagram again, this time numbering the three routes in use: 1, 2, and 3.

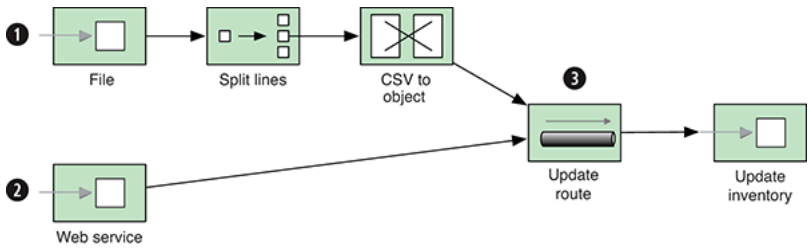


Figure 15.4 Camel application with two input routes, 1 and 2, that depend on a common route, 3

The file-based route ❶ polls incoming files and splits each line in the file. The lines are then converted to an internal `camelinaction.inventory.UpdateInventoryInput` object, which is sent to route 3.

The web service route ❷ is much simpler, because incoming messages are automatically converted to the `UpdateInventoryInput` object. The web service endpoint is configured to do this.

The update route ❸ is a common route that's reused by the first two routes.

You now have a dependency among the three routes. Routes 1 and 2 depend on route 3, and that's why you can use `StartupOrder` to ensure that the routes are started in correct order.

The following listing shows the Camel routes with the `StartupOrder` options in boldface.

Listing 15.1 Starting routes in a specific order

```
public class InventoryRoute extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        from("cxf:bean:inventoryEndpoint")
            .routeId("webservice").startupOrder(3)
            .to("direct:update")
            .transform().method("inventoryService", "replyOk");
        from("file://target/inventory/updates")
```

```
                .routeId("file").startupOrder(2)
                .split(body().tokenize("\n"))
                    .convertBodyTo(UpdateInventoryInput.class)
                    .to("direct:update")
                .end();
            from("direct:update")
                .routeId("update").startupOrder(1)
                .to("bean:inventoryService?method=updateInventory");
        }
    }
```

Listing 15.1 shows how easy it is in the Java DSL to configure the order of the routes using the `StartupOrder` method. Listing 15.2 shows the same example using the XML DSL.

NOTE In listing 15.1, `routeId` is used to assign each route a meaningful name, which will then show up in the management console or in the logs. If you don't assign an ID, Camel will auto-assign an ID using the pattern route1, route2, and so forth.

Listing 15.2 XML DSL version of listing 15.1

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route id="webservice" startupOrder="3">
        <from uri="cxf:bean:inventoryEndpoint"/>
        <to uri="direct:update"/>
        <transform>
            <method bean="inventoryService" method="replyOk"/>
        </transform>
    </route>
    <route id="file" startupOrder="2">
        <from uri="file://target/inventory/updates"/>
        <split>
            <tokenize token="\n"/>
            <convertBodyTo
                type="camelinaction.inventory.UpdateInventoryInput"/>
            <to uri="direct:update"/>
        </split>
    </route>
    <route id="update" startupOrder="1">
```

```
        <from uri="direct:update"/>
        <to uri="bean:inventoryService?method=updateInventory"/>
    </route>
</camelContext>
```

Notice that the numbers 1, 2, and 3 are used to dictate the order of the routes. Let's take a moment to see how this works in Camel.

How StartupOrder works

The `StartupOrder` option in Camel works much like the `load-on-startup` option for Java servlets. As with servlets, you can specify a positive number to indicate the order in which the routes should be started.

The numbers don't have to be consecutive. For example, you could use the numbers 5, 20, and 42 instead of 1, 2, and 3. All that matters is that the numbers must be unique.

You can also omit assigning `StartupOrder` to some of the routes. In that case, Camel will automatically assign these routes a unique number starting with 1,000 upward. The numbers from 1 to 999 are intended for Camel users, and the numbers from 1,000 upward are reserved by Camel.

TIP The routes are stopped in the reverse order in which they were started.

In practice, you may not need to use `StartupOrder` often. Camel is much more careful than it used to be when starting routes. In fact, in this example, if you let Camel determine the start order, it would likely be okay. The potential for issues comes most often if your endpoints are getting hammered with incoming messages while Camel is restarting. Then there's a chance a message could get in before all routes are operational. If that's the case, you may see an error like this:

```
org.apache.camel.component.direct.DirectConsumerNotAvailableException: N
```

The error indicates that the `update` route didn't start in time before someone sent a message to the `webservice` route.

NOTE The Camel team will improve on this for the upcoming Camel 2.21 release to let the direct component be smarter during startup and gracefully handle such a situation. Almost all the other components already handle interdependent startup ordering gracefully, so the need for explicit startup ordering will become less relevant in the future.

The book's source code contains this example in the chapter15/startup directory. You can try it using the following Maven goals:

```
mvn test -Dtest=InventoryJavaDSLTest
mvn test -Dtest=InventorySpringXMLTest
```

You've now learned to control the order in which routes are started. Let's move on and take a look at how to omit starting certain routes and start them on demand later, at runtime.

15.1.4 Disabling autostartup

Table 15.1 listed the `AutoStartup` option, which is used to specify whether a given route should be automatically started when Camel starts. Sometimes you may not want to start a route automatically. You may want to start it on demand at runtime to support business cases involving human intervention.

At Rider Auto Parts, there has been a demand to implement a manual process for updating the inventory based on files. As usual, you've been asked to implement this in the existing application depicted in figure 15.4.

You come up with the following solution: add a new route to the existing routes in listing 15.1. The new route listens for files being dropped in the manual directory, and uses these files to update the inventory:

```
from("file://target/inventory/manual")
    .routeId("manual")
    .log("Doing manual update with file ${file:name}")
    .split(body().tokenize("\n"))
        .convertBodyTo(UpdateInventoryInput.class)
```

```
            .to("direct:update")
        .end();
```

As you can see, the route is merely a copy of the file-based route in listing 15.1. Unfortunately, your boss isn't satisfied with the solution. The route is always active, so if someone accidentally drops a file into the manual folder, it would be picked up.

To solve this problem, you use the `AutoStartup` option to disable the route from being activated on startup:

```
from("file://target/inventory/manual")
    .routeId("manual").noAutoStartup()
    .log("Doing manual update with file ${file:name}")
    .split(body().tokenize("\n"))
        .convertBodyTo(UpdateInventoryInput.class)
        .to("direct:update")
    .end();
```

Notice that you use `noAutoStartup()` to disable route startup. You could have used `autoStartup(false)`, `autoStartup("false")`, or even `autoStartup("{{startupRouteProperty}}")`, where a property is referenced. In the XML DSL, there's no `noAutoStartup()`, but you can use `autoStartup="false"` or the other alternatives mentioned.

Now that the route isn't started by default, you can start the route when a manual file is meant to be picked up. This can be done using a management console, such as JConsole, to manually start the route, waiting until the file has been processed, and manually stopping the route again.

You've now learned how to configure Camel with various options that influence how it starts up. The next section presents various ways of programmatically controlling the lifecycle of routes at runtime.

15.2 Starting and stopping routes at runtime

In chapter 16, you'll learn how to use management tooling, designed for operations staff, to start and stop routes at runtime. Being able to programmatically control routes at runtime is also desirable. For example,

you might want business logic to automatically turn routes on or off at runtime. This section explains how to do this.

You can start and stop routes at runtime in several ways, including these:

- *Using* `CamelContext` —By invoking the `startRoute/startAllRoutes` and `stopRoute` methods.
- *Using* `RoutePolicy` —By applying a policy to routes that Camel enforces automatically at runtime.
- *Using JMX*—By obtaining the `ManagedRoute` MBean for the particular routes and invoking its `start` or `stop` methods. If you've enabled remote management, you can control the routes from another machine.
- *Using the ControlBus component*—The control bus allows you to manage routes at runtime using regular messaging instead of a special SPI. This is also discussed in chapter 16, in section 16.4.3.

The use of JMX is covered in chapter 16; we discuss using `CamelContext`, ControlBus, and `RoutePolicy` in this chapter.

15.2.1 Using CamelContext to start and stop routes at runtime

`CamelContext` provides methods to easily start and stop routes. To illustrate this, we'll continue with the Rider Auto Parts example from section 15.1.4. About a month into production with the new route, one of the operations staff forgets to manually stop the route after use, as he was supposed to. Not stopping the route leads to a potential risk because files accidentally dropped into the manual directory will be picked up by the route.

You're again summoned to remedy this problem, and you quickly improve the route with the two changes shown in bold in the following listing.

Listing 15.3 After a file has been processed, the route is stopped

```
from("file://target/inventory/manual?maxMessagesPerPoll=1")
    .routeId("manual").noAutoStartup()
    .log("Doing manual update with file ${file:name}")
    .split(body().tokenize("\n"))
        .convertBodyTo(UpdateInventoryInput.class)
        .to("direct:update")
```

```
        .end()
        .process(new Processor() {
            public void process(Exchange exchange) throws Exception {
                exchange.getContext().getInflightRepository()
                    .remove(exchange, "manual");

                ExecutorService executor =
                    getContext().getExecutorServiceManager()
                   .newSingleThreadExecutor(this, "StopRouteManually");
                executor.submit(new Callable<Object>() {
                    @Override
                    public Object call() throws Exception {
                        log.info("Stopping route manually");
                        getContext().stopRoute("manual");
                        return null;
                    }
                });
        }
    });
```

The first change uses the `maxMessagesPerPoll` option to tell the file consumer to pick up only one file at a time. The second change stops the route after that one file has been processed. This is done with the help of the inlined `Processor`, which can access `CamelContext` and tell it to stop the route by name. (`CamelContext` also provides a `startRoute` method for starting a route.) Before you stop the route, you must unregister the current exchange from the in-flight registry, which otherwise would prevent Camel from stopping the route, because it detects an exchange in progress. It's also important to call `stopRoute` from a thread separate from the current route. Older versions of Camel were less picky about this, but with the latest release, you need to use a separate thread.

The book's source code contains this example, which you can try from the chapter15/startup directory using the following Maven goal:

```
mvn test -Dtest=ManualRouteWithStopTest
```

Even though the fix to stop the route is simple, using the inlined processor at the end of the route isn't an optimal solution. It'd be better to keep

the business logic separated from the stopping logic. That can be done with a feature called `OnCompletion`.

Using OnCompletion

`OnCompletion` allows you to do *additional* routing after the original route is done. The classic example is to send an email alert if a route fails, but `OnCompletion` has a broad range of uses.

Instead of using the inlined processor to stop the route, you can use `OnCompletion` in `RouteBuilder` to process the `StopRouteProcessor` class containing the logic to stop the route. This is shown in bold in the following code:

```
public void configure() throws Exception {
    onCompletion().process(new StopRouteProcessor("manual"));
    from("file://target/inventory/manual?maxMessagesPerPoll=1")
        .routeId("manual").noAutoStartup()
        .log("Doing manual update with file ${file:name}")
        .split(body().tokenize("\n"))
            .convertBodyTo(UpdateInventoryInput.class)
            .to("direct:update")
        .end();
}
```

The implementation of `StopRouteProcessor` is simple, as shown here:

```
public class StopRouteProcessor implements Processor {
    private final String name;

    public StopRouteProcessor(String name) {
        this.name = name;
    }

    public void process(Exchange exchange) throws Exception {
        exchange.getContext().getInflightRepository().remove(exchange, n
        exchange.getContext().stopRoute(name);
    }
}
```

This improves the readability of the route, because it's shorter and doesn't mix high-level routing logic with low-level implementation logic. Using `OnCompletion`, the stopping logic has been separated from the original route.

You can use scopes to define `OnCompletion`s at different levels. Camel supports two scopes: context scope (high level) and route scope (low level). The preceding example uses context scope. If you want to use route scope, you have to define it within the route as follows:

```
from("file://target/inventory/manual?maxMessagesPerPoll=1")
    .onCompletion().process(new StopRouteProcessor("manual")).end()
    .routeId("manual").noAutoStartup()
    .log("Doing manual update with file ${file:name}")
    .split(body().tokenize("\n"))
        .convertBodyTo(UpdateInventoryInput.class)
        .to("direct:update")
    .end;
```

Notice the use of `.end()` to indicate where the `OnCompletion` route ends. You have to do this when using route scope so Camel knows which pieces belong to the *additional* route and which to the original route. This is the same principle as when you use `OnException` at route scope.

TIP `OnCompletion` also supports filtering using the `OnWhen` predicate so that you can trigger the additional route only if the predicate is `true`. In addition, `OnCompletion` can be configured to trigger only when the route completes successfully or when it fails using the `OnCompleteOnly` or `OnFailureOnly` options. For example, you can use `OnFailureOnly` to build a route that sends an email alert to support personnel when a route fails.

The book's source code contains this example in the chapter15/startup directory. You can try it using the following Maven goals:

```
mvn test -Dtest=ManualRouteWithOnCompletionTest
mvn test -Dtest=SpringManualRouteWithOnCompletionTest
```

We've covered how to stop a route at runtime using the `CamelContext` API. We'll now look at how to do this with the Control Bus EIP, which makes things simpler.

15.2.2 Using the Control Bus EIP to start and stop routes at runtime

Another way of controlling routes is using the Control Bus EIP. Yes, it's an enterprise integration pattern! Camel implements this via the ControlBus component, instead of using a method in the DSL. By sending a message to a ControlBus endpoint, you can start, stop, resume, or suspend a route.

TIP Control Bus is also discussed in chapter 16, in section 16.4.3.

Looking back to the previous example in listing 15.3, you can greatly simplify this with the Control Bus EIP:

```
from("file://target/inventory/manual?maxMessagesPerPoll=1")
    .routeId("manual").noAutoStartup()
    .log("Doing manual update with file ${file:name}")
    .split(body().tokenize("\n"))
        .convertBodyTo(UpdateInventoryInput.class)
        .to("direct:update")
    .end()
    .to("controlbus:route?routeId=manual&action=stop&async=true");
```

The ControlBus component has a few options that you're setting up here. The `routeId` refers to the route you'll be controlling; `action` is `stop` because you want to stop the route. Finally, you set the `async` flag to make this action nonblocking—you don't need a result set on the current exchange. Also notice that you don't need to set up any executors this time or even manage the inflight repository of exchanges—it's all handled by the Control Bus.

The book's source code contains this example in the chapter15/controlbus directory. You can try it using the following Maven goal:

```
mvn test -Dtest=ControlBusTest
```

We've now covered how to stop a route at runtime using the Control Bus EIP. Let's look at another feature called `RoutePolicy` that you can use to control the lifecycle of routes at runtime.

15.2.3 Using RoutePolicy to start and stop routes at runtime

`RoutePolicy` is a policy that can control routes at runtime. For example, `RoutePolicy` can control whether a route should be active. But you aren't limited to such scenarios; you can implement any kind of logic you want.

The `org.apache.camel.spi.RoutePolicy` interface defines several call-back methods that Camel will automatically invoke at runtime:

```
public interface RoutePolicy {
    void onInit(Route route);
    void onRemove(Route route);
    void onStart(Route route);
    void onStop(Route route);
    void onSuspend(Route route);
    void onResume(Route route);
    void onExchangeBegin(Route route, Exchange exchange);
    void onExchangeDone(Route route, Exchange exchange);
}
```

The idea is that you implement this interface, and Camel will invoke the callbacks when a new message arrives into a route, when a route has started or stopped, and so forth. You're free to implement whatever logic you want in these callbacks. For convenience, Camel provides the `org.apache.camel.support.RoutePolicySupport` class, which you can use as a base class to extend when implementing your custom policies.

Let's build a simple example using `RoutePolicy` to demonstrate how to flip between two routes, so only one route is active at any time. Figure 15.5 shows this principle.

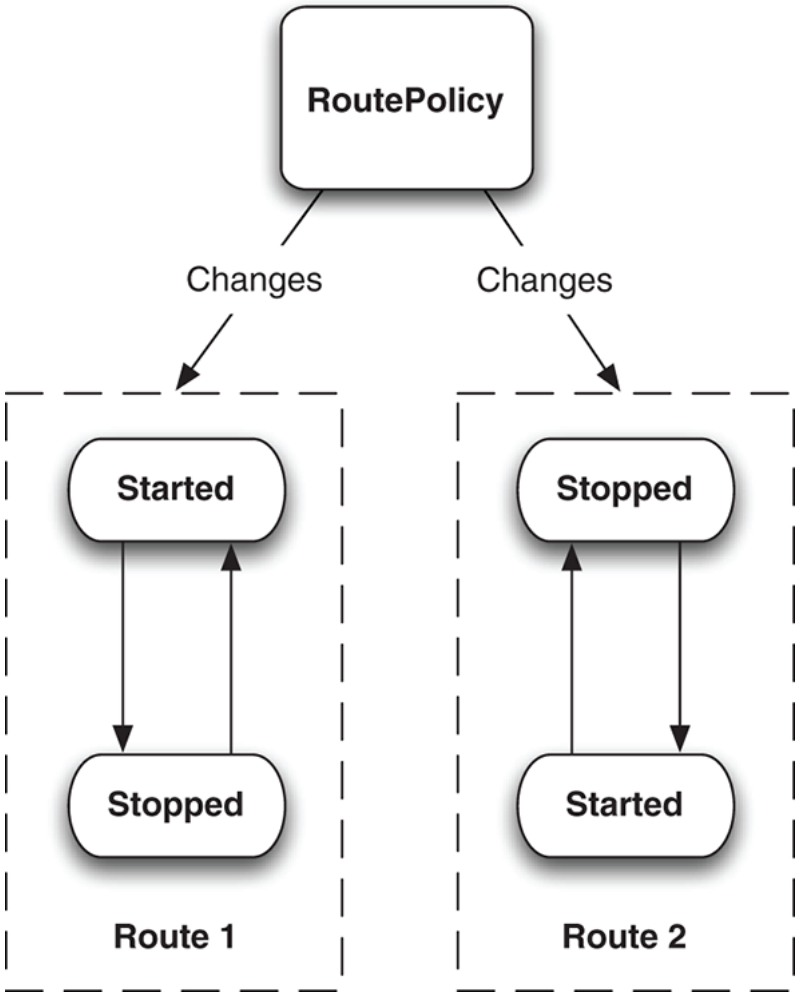


Figure 15.5 `RoutePolicy` changes the active state between the two routes so only one route is active at any time.

As you can see, `RoutePolicy` is being used to control the two routes, starting and stopping them so only one is active at a time. The following listing shows how this can be implemented.

Listing 15.4 A `RoutePolicy` that flips two routes being active at runtime

```
public class FlipRoutePolicy extends RoutePolicySupport {
    private final String name1;
    private final String name2;

    public FlipRoutePolicy(String name1, String name2) {    ❶
```

❶ Identifies routes to flip

```
        this.name1 = name1;
        this.name2 = name2;
```

```
    }

    @Override
    public void onExchangeDone(Route route, Exchange exchange) {
        String stop = route.getId().equals(name1) ? name1 : name2;
        String start = route.getId().equals(name1) ? name2 : name1;
        CamelContext context = exchange.getContext();
        try {
            exchange.getContext().getInflightRepository().remove(exchang
        context.stopRoute(stop);    ❷
```

---

❷

Flips the two routes

```
        context.startRoute(start);    ❷
        } catch (Exception e) {
            getExceptionHandler().handleException(e);
        }
    }
}
```

In the constructor, you identify the names of the two routes to flip ❶. As you extend the `RoutePolicySupport` class, you override only the `onExchangeDone` method, as the flipping logic should be invoked when the route is done. You then compute which of the two routes to stop and start with the help of the `route` parameter, which denotes the current active route. Having computed that, you use `CamelContext` to flip the routes ❷. If an exception is thrown, you let `ExceptionHandler` take care of it, which by default logs the exception.

To use `FlipRoutePolicy`, you must assign it to the two routes. In the Java DSL, this is done using the `RoutePolicy` method, as shown in the following `RouteBuilder`:

```
public void configure() throws Exception {
    RoutePolicy policy = new FlipRoutePolicy("foo", "bar");

    from("timer:foo")
        .routeId("foo").routePolicy(policy)
```

```
            .setBody().constant("Foo message")
            .to("log:foo").to("mock:foo");

        from("timer:bar")
            .routeId("bar").routePolicy(policy).noAutoStartup()
            .setBody().constant("Bar message")
            .to("log:bar").to("mock:bar");
    }
```

If you're using XML DSL, you can use `RoutePolicy`, as shown here:

```
<bean id="flipPolicy" class="camelinaction.FlipRoutePolicy">
    <constructor-arg index="0" value="foo"/>
    <constructor-arg index="1" value="bar"/>
</bean>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route id="foo" routePolicyRef="flipPolicy">
        <from uri="timer:foo"/>
        <setBody><constant>Foo message</constant></setBody>
        <to uri="log:foo"/>
        <to uri="mock:foo"/>
    </route>
    <route id="bar" routePolicyRef="flipPolicy" autoStartup="false">
        <from uri="timer:bar"/>
        <setBody><constant>Bar message</constant></setBody>
        <to uri="log:bar"/>
        <to uri="mock:bar"/>
    </route>
</camelContext>
```

As you can see, you use the `routePolicyRef` attribute on the `<route>` tag to reference the `flipPolicy` bean defined in the top of the XML file.

The book's source code contains this example in the chapter15/routepolicy directory. You can try it using the following Maven goals:

```
mvn test -Dtest=FlipRoutePolicyJavaDSLTest
mvn test -Dtest=FlipRoutePolicySpringXMLTest
```

When running either of the examples, you should see the two routes being logged interchangeably (`foo` and `bar`):

```
INFO  foo - Exchange[BodyType:String, Body:Foo message]
INFO  bar - Exchange[BodyType:String, Body:Bar message]
INFO  foo - Exchange[BodyType:String, Body:Foo message]
INFO  bar - Exchange[BodyType:String, Body:Bar message]
INFO  foo - Exchange[BodyType:String, Body:Foo message]
INFO  bar - Exchange[BodyType:String, Body:Bar message]
```

We've now covered both starting and controlling routes at runtime. It's time to learn about shutting down Camel, which is more complex than it sounds.

15.3 Shutting down Camel

The new inventory application at Rider Auto Parts is scheduled to be in production at the end of the month. You're on the team to ensure its success and help run the final tests before it's handed over to production. These tests also cover reliably shutting down the application.

Shutting down the Camel application is complex because numerous in-flight messages could be being processed. Shutting down while messages are in flight may harm your business, because those messages could be lost. The goal of shutting down a Camel application reliably is to shut it down when it's *quiet*—when there are no in-flight messages. All you have to do is find this quiet moment.

This is hard to do, because while you wait for the current messages to complete, the application may take in new messages. You have to stop taking in new messages while the current messages are given time to complete. This process is known as *graceful shutdown,* which means shutting down in a reliable and controlled manner.

15.3.1 Graceful shutdown

When `CamelContext` is being stopped, which happens when its `stop()` method is invoked, it uses a strategy to shut down. This strategy is defined in the `ShutdownStrategy` interface. The default implementation of this `ShutdownStrategy` interface uses the graceful shutdown technique.

For example, when you stop the Rider Auto Parts example, you'll see these log lines:

```
SpringCamelContext       - Apache Camel 2.20.1 (CamelContext: camel-1)
                           is shutting down
DefaultShutdownStrategy  - Starting to graceful shutdown 3 routes
                           (timeout 10 seconds)
DefaultShutdownStrategy  - Route: webservice shutdown complete, was
                           consuming from: cxf://bean:inventoryEndpoint
DefaultShutdownStrategy  - Route: file shutdown complete, was consuming
                           from: file://target/inventory/updates
DefaultShutdownStrategy  - Route: update shutdown complete, was consumin
                           from: direct://update
DefaultShutdownStrategy  - Graceful shutdown of 3 routes completed in
                           0 seconds
SpringCamelContext       - Apache Camel 2.20.1 (CamelContext: camel-1)
                           uptime 0.684 seconds
SpringCamelContext       - Apache Camel 2.20.1 (CamelContext: camel-1)
                           is shutdown in 0.025 seconds
```

This tells you a few things. You can see that the graceful shutdown is using a 10-second time-out. This is the maximum time Camel allows for shutting down gracefully before it starts to shut down more aggressively by forcing routes to stop immediately. The default value is 300 seconds, but the log was a unit test, and `CamelTestSupport` uses 10 seconds. You can configure this time-out yourself on `CamelContext`. For example, to use 20 seconds as the default time-out value, you can do the following:

```
camelContext.getShutdownStrategy().setTimeout(20);
```

Doing this in XML requires a bit more work, because you have to define a `<bean>` to set the time-out value:

```
<bean id="shutdown" class="org.apache.camel.impl.DefaultShutdownStrategy
    <property name="timeout" value="20"/>
</bean>
```

Notice that the time-out value is in seconds.

Then Camel logs the progress of the routes as they shut down, one by one, according to the reverse order in which they were started.

At the end, Camel logs the completion of the graceful shutdown, which in this case was fast and completed in less than 1 second. Camel also logs whether there were any in-flight messages just before it stops completely.

If any in-flight exchanges are holding up the graceful shutdown, Camel prints a status about these every second, as follows:

```
DefaultShutdownStrategy - Waiting as there are still 3 inflight and
pending exchanges to complete, timeout in 60 seconds
Inflights per route: [file = 2, update = 1]
```

This logging continues until the in-flight exchanges complete or the time-out is reached. A time-out indicates that Camel didn't complete the graceful shutdown, and it would log a warning showing the in-flight exchanges that didn't complete:

```
WARN  DefaultShutdownStrategy - Timeout occurred during graceful shutdow
                                Forcing the routes to be shutdown now.
                                Notice: some resources may still be runn
                                as graceful shutdown did not complete
                                successfully.
WARN  DefaultShutdownStrategy - Interrupted while waiting during gracefu
                                shutdown, will force shutdown now.
INFO  DefaultShutdownStrategy - Route: file shutdown complete, was
                                consuming from: file://target/inventory/
INFO  DefaultShutdownStrategy - Route: update shutdown complete, was
                                consuming from: direct://update
INFO  DefaultShutdownStrategy - There are 2 inflight exchanges:
       InflightExchange: [exchangeId=ID-ghost-33143-1479685285199-0-10,
       fromRouteId=file, routeId=update, nodeId=to3, elapsed=17, duratio
       InflightExchange: [exchangeId=ID-ghost-33143-1479685285199-0-4,
       fromRouteId=file, routeId=file, nodeId=split1, elapsed=3100, dura
```

All this additional logging may not be desirable for all applications. Maybe it's expected that on shutdown, messages are discarded. In such cases, you can disable the additional logging. To suppress the warning on time-out, you can do the following:

```
context.getShutdownStrategy().setSuppressLoggingOnTimeout(true);
```

To suppress the status messages about in-flight exchanges during graceful shutdown, you can do this:

```
context.getShutdownStrategy().setLogInflightExchangesOnTimeout(false);
```

Shutting down the Rider Auto Parts application

At Rider Auto Parts, you're in the final testing of the application before it's handed over to production. One of the tests is based on processing a big inventory file, and you want to test what happens if you shut down Camel while it's working on the big file. Let's look at how Camel handles this.

At first, you see the usual logging about the shutdown in progress:

```
SpringCamelContext      - Apache Camel 2.20.1 (CamelContext: camel-1) is
                          shutting down
DefaultShutdownStrategy - Starting to graceful shutdown 3 routes
                          (timeout 60 seconds)
DefaultShutdownStrategy - Route: webservice shutdown complete, was
                          consuming from: cxf://bean:inventoryEndpoint
```

Then there's a log line indicating that Camel has noticed in-flight exchanges, which are from the big file, and also exchanges generated from lines of the big file. This is expected behavior:

```
DefaultShutdownStrategy - Waiting as there are still 3 inflight and pend
                          exchanges to complete, timeout in 60 seconds.
                          Inflights per route: [file = 2, update = 1]
```

The application logs the progress of the inventory update, which should happen for each line in the big file:

```
Inventory 58004 updated
```

Finally, you see the last log lines, which report the end of the shutdown:

```
DefaultShutdownStrategy - Route: file shutdown complete, was consuming
                          from: file://target/inventory/updates
DefaultShutdownStrategy - Route: update shutdown complete, was consuming
                          from: direct://update
DefaultShutdownStrategy - Graceful shutdown of 3 routes completed in 14
                          seconds
SpringCamelContext      - Apache Camel 2.20.1 (CamelContext: camel-1)
                          uptime 17.747 seconds
SpringCamelContext      - Apache Camel 2.20.1 (CamelContext: camel-1)
                          is shutdown in 14.028 seconds
```

Notice that the routes are shut down in the reverse order in which they were started. This is handy because it maintains any order dependency they had during startup. Previous versions of Camel behaved differently than this, and you often had to defer shutdown of routes that were required longer than their startup order implied. For example, to defer shutdown of the `update` route, you could use the `ShutdownRoute` option, which was listed in table 15.1. All you have to do is add the option in the route, as shown in bold here:

```
from("direct:update")
    .routeId("update").startupOrder(1)
    .shutdownRoute(ShutdownRoute.Defer)
    .to("bean:inventoryService?method=updateInventory");
```

The same route in XML DSL is as follows:

```
<route id="update" startupOrder="1" shutdownRoute="Defer">
    <from uri="direct:update"/>
    <to uri="bean:inventoryService?method=updateInventory"/>
</route>
```

The book's source code contains this example in the chapter15/shutdown directory. You can try it using the following Maven goals:

```
mvn test -Dtest=GracefulShutdownBigFileTest
mvn test -Dtest=GracefulShutdownBigFileXmlTest
mvn test -Dtest=GracefulShutdownBigFileDeferTest
mvn test -Dtest=GracefulShutdownBigFileDeferXmlTest
```

As you've seen, Camel gives end users full control over how to configure their routes to ensure a proper shutdown. That said, most of the time you don't need to worry about manually ordering your routes. It's nice to have the flexibility, though, just in case you need to work around an issue.

About stopping and shutting down

Camel uses the graceful shutdown mechanism when it stops or shuts down routes, so listing 15.3 will stop the route in a graceful manner. As a result, you can reliably stop routes at runtime without the risk of losing in-flight messages.

The difference between using the `stopRoute` and `shutdownRoute` methods is that the latter also unregisters the route from management (JMX). Use `stopRoute` when you want to be able to start the route again; use `shutdownRoute` only if the route should be permanently removed.

That's all there is to shutting down Camel. It's now time to review some of the possible deployment strategies.

15.4 Deploying Camel

Camel is described as a lightweight and embeddable integration framework. It supports more deployment strategies and flexibility than traditional ESBs and application servers do. Camel can be used in a wide range of runtime environments, from standalone Java applications to web containers to the cloud.

This section presents five deployment strategies that are possible with Camel and their strengths and weaknesses:

- Embedding Camel in a Java application
- Running Camel in a web environment such as Apache Tomcat
- Running Camel inside WildFly
- Running Camel in an OSGi container such as Apache Karaf
- Running Camel in a container that supports CDI, such as Apache Karaf or WildFly

It's also possible to deploy Camel using Spring Boot and WildFly Swarm. We cover those two deployment options in chapter 7, where we discuss microservices.

15.4.1 Embedded in a Java application

Embedding Camel in a Java application is appealing if you need to communicate with the outside world. By bringing Camel into your application, you can benefit from all the transports, routes, EIPs, and so on that Camel offers. Figure 15.6 shows Camel embedded in a standard Java application.

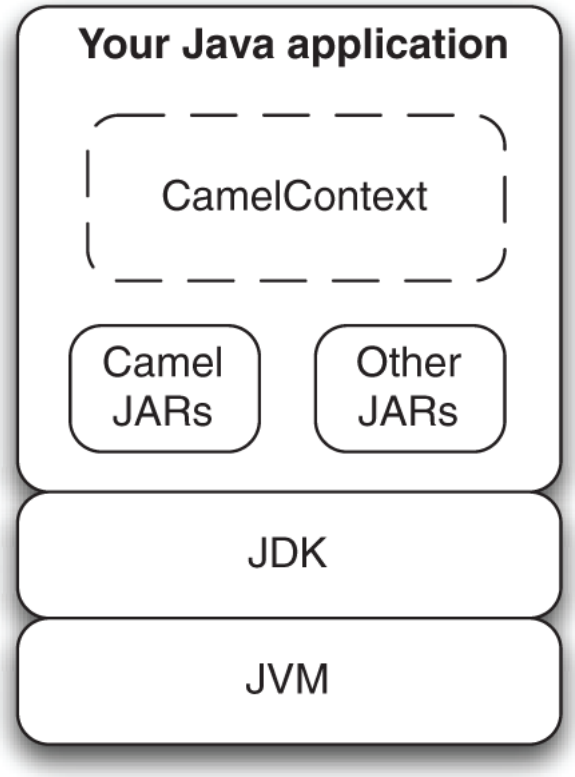


Figure 15.6 Camel embedded in a standalone Java application

In the embedded mode, you must add to the classpath all the necessary Camel and third-party JARs needed by the underlying transports. Because Camel is built with Maven, you can use Maven for your own project and benefit from its dependency-management system. We discuss building Camel projects with Maven in chapter 8.

Bootstrapping Camel for your code is easy. In fact, you did that in your first ride on the Camel in chapter 1. All you need to do is create `CamelContext` and start it, as shown in the following listing.

Listing 15.5 Bootstrapping Camel in your Java application

```
public class FileCopierWithCamel {
    public static void main(String args...) throws Exception {
        CamelContext context = new DefaultCamelContext();
```

```
            context.addRoutes(new RouteBuilder() {
                public void configure() {
                    from("file:data/inbox").to("file:data/outbox");
                }
            });
            context.start();
            Thread.sleep(10000);
            context.stop();
        }
    }
```

It's important to keep a reference to `CamelContext` for the lifetime of your application, because you'll need to perform a shutdown of Camel. As you may have noticed in listing 15.5, the code execution continues after starting `CamelContext`. To avoid shutting down your application immediately, the listing includes code to sleep for 10 seconds. In your application, you'll need to use a different means to let your application continue and shut down only when requested to do so.

Let's look at the Rider Auto Parts application illustrated in figure 15.1 and embed it in a Java application.

Embedding the Rider Auto Parts example in a Java application

The Rider Auto Parts Camel application can be run as a standalone Java application. Because the application is using Spring XML files to set up Camel, you need to start only Spring to start the application.

Starting Spring from a main class can be done as follows:

```
public class InventoryMain {
    public static void main(String[] args) throws Exception {
        String filename = "META-INF/spring/camel-context.xml";
        AbstractXmlApplicationContext spring =
               new ClassPathXmlApplicationContext(filename);
        spring.start();
        Thread.sleep(10000);
        spring.stop();
        spring.destroy();
    }
}
```

To start Spring, you create `ApplicationContext`, which in the preceding example means loading the Spring XML file from the classpath. This code also reveals the problem of having the main method wait until you terminate the application. The preceding code uses `Thread.sleep` to wait 10 seconds before terminating the application.

To remedy this, Camel provides a `Main` class that you can use instead of writing your own class. You can change the previous `InventoryMain` class to use this `Main` class as follows:

```
import org.apache.camel.spring.Main;

public class InventoryMain {
    public static void main(String[] args) throws Exception {
        Main main = new Main();
        main.setApplicationContextUri("META-INF/spring/camel-context.xml
        main.run();
    }
}
```

This approach also solves the issue of handling the lifecycle of the application. Camel will shut down gracefully when the JVM is being terminated, such as when the Ctrl-C key combination is pressed. You can also stop the Camel application by invoking the `stop` method on the `main` instance.

The book's source code contains this example in the chapter15/standalone directory. You can try it using the following Maven goal:

```
mvn compile exec:java
```

TIP You may not need to write your own main class. For example, the `org.apache.camel.main.Main`, `org.apache.camel.spring.Main`, and `org.apache.camel.cdi.Main` classes can be used directly. They have parameters to dictate which `RouteBuilder` class or Spring XML file should be loaded. By default, the `org.apache.camel.spring.Main` class loads all XML files from the classpath in the META-INF/spring location, so by dropping your Spring XML file in there, you don't even have to pass any arguments to the `Main` class. You can start it directly.

Table 15.2 summarizes the pros and cons of embedding Camel in a standalone Java application.

Table 15.2 Pros and cons of embedding Camel in a standalone Java application

Pros	Cons
• Gives flexibility to deploy just what's needed • Allows you to embed Camel in any standard Java application	• Requires deploying all needed JARs • Requires manually managing Camel's lifecycle (starting and stopping) on your own

You now know how to make your standalone application use Camel. Let's look at what you can do for your web applications.

15.4.2 Embedded in a web application

Embedding Camel in a web application brings the same benefits as those mentioned in section 15.4.1. Camel provides all you need to connect to your favorite web container. If you work in an organization, you may be able to use existing infrastructure for deploying your Camel applications. Deploying Camel in such a well-known environment gives you immediate support for installing, managing, and monitoring your applications.

When Camel is embedded in a web application, as shown in figure 15.7, you need to make sure all JARs are packaged in the WAR file. If you use Maven, this is done automatically.

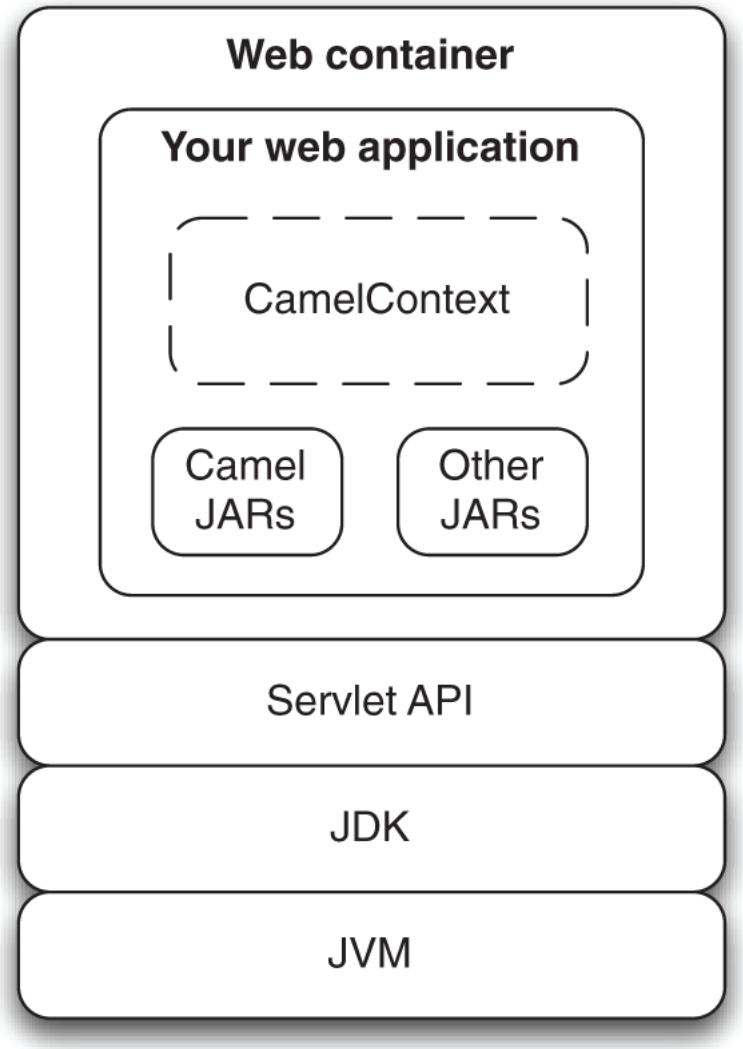


Figure 15.7 Camel embedded in a web application

The Camel instance embedded in your web application is bootstrapped by Spring. Using Spring, which is such a ubiquitous framework, you let end users use well-known approaches for deployment. This also conveniently ties Camel's lifecycle with Spring's lifecycle management and ensures that the Camel instance is properly started and stopped in the web container.

The following code demonstrates that you need only a standard Spring context listener in the web.xml file to bootstrap Spring and thereby Camel:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="
           http://java.sun.com/xml/ns/javaee
           http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
           version="3.0">
```

```
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
</web-app>
```

This context listener also takes care of properly shutting down Camel when the web application is stopped. Spring will, by default, load the Spring XML file from the WEB-INF folder using the name applicationContext.xml. In this file, you can embed Camel, as shown in listing 15.6.

SPECIFYING THE LOCATION OF YOUR SPRING XML FILE

If you want to use another name for your Spring XML file, you need to add a context parameter that specifies the filename as follows:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/camel-context.xml</param-value>
</context-param>
```

Listing 15.6 The Spring applicationContext.xml file with Camel embedded

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:camel="http://camel.apache.org/schema/spring"
       xsi:schemaLocation="
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd
         http://camel.apache.org/schema/spring
         http://camel.apache.org/schema/spring/camel-spring.xsd">
  <import resource="camel-cxf.xml"/>  (1)
```

(1) Imports CXF from another XML file

```
    <bean id="inventoryService" class="camelinaction.InventoryService"/>
    <bean id="inventoryRoute" class="camelinaction.InventoryRoute"/>
    <camelContext xmlns="http://camel.apache.org/schema/spring">
        <routeBuilder ref="inventoryRoute"/>
    </camelContext>
</beans>
```

Listing 15.6 is a regular Spring XML file in which you can use the `<import>` tag to import other XML files. For example, you do this by having CXF defined in the camel-cxf.xml file ❶. Camel itself is embedded using the `<camelContext>` tag.

The book's source code contains this example in the chapter15/war directory. You can try it using the following Maven goal:

```
mvn jetty:run
```

If you run this Maven goal, a Jetty plugin is used to quickly boot up a Jetty web container running the web application. To use the Jetty plugin in your projects, you must remember to add it to your pom.xml file in the `<build><plugins>` section:

```
<plugin>
    <groupId>org.eclipse.jetty</groupId>
    <artifactId>maven-jetty-plugin</artifactId>
    <version>9.2.21.v20170120</version>
</plugin>
```

If you run this goal, you should notice in the console that Jetty has been started:

```
2016-11-27 17:05:44,884 [main           ] INFO  SpringCamelContext
- Apache Camel 2.20.1 (CamelContext: camel-1) started in 0.788 seconds
...
[INFO] Started ServerConnector@2d2f09a4{HTTP/1.1}{0.0.0.0:8080}
[INFO] Started @34700ms
[INFO] Started Jetty Server
```

Let's look at running Camel as a web application in Apache Tomcat.

Deploying to Apache Tomcat

To package the application as a WAR file, you can run the `mvn package` command, which creates the WAR file in the target directory. Yes, it's that easy with Maven.

You want to use the hot deployment of Apache Tomcat, so you must first start it. Here's how to start it on a Linux system using the bin/startup.sh script:

```
[janstey@ghost apache-tomcat-8.5.23]$ bin/startup.sh
Using CATALINA_BASE:   /home/janstey/kits/apache-tomcat-8.5.23
Using CATALINA_HOME:   /home/janstey/kits/apache-tomcat-8.5.23
Using CATALINA_TMPDIR: /home/janstey/kits/apache-tomcat-8.5.23/temp
Using JRE_HOME:        /usr/java/jdk1.8.0_91/
Using CLASSPATH:       /home/janstey/kits/apache-tomcat-8.5.23/bin/boots
Tomcat started.
```

This starts Tomcat in the background, so you need to tail the log file to see what happens:

```
[janstey@ghost apache-tomcat-8.5.23]$ tail -f logs/catalina.out
27-Nov-2016 17:41:38.856 INFO [localhost-startStop-1]
org.apache.catalina.startup.HostConfig.deployDirectory Deployment of web
application directory /home/janstey/kits/apache-tomcat-8.5.8/webapps/exa
...
27-Nov-2016 17:41:38.898 INFO [main]
org.apache.coyote.AbstractProtocol.start Starting ProtocolHandler
[http-nio-8080]
27-Nov-2016 17:41:38.902 INFO [main]
org.apache.coyote.AbstractProtocol.start Starting ProtocolHandler
[ajp-nio-8009]
27-Nov-2016 17:41:38.902 INFO [main]
org.apache.catalina.startup.Catalina.start Server startup in 461 ms
```

To deploy the application, you need to copy the WAR file to the Apache Tomcat webapps directory:

```
cp target/riderautoparts-war-2.0.0.war ~/kits/apache-tomcat-8.5.23/webap
```

Apache Tomcat should show the application being started in the log file. You should see the familiar logging of Camel being started:

```
2016-11-27 17:45:22,507 [ost-startStop-2] INFO  SpringCamelContext
- Total 3 routes, of which 3 are started.
2016-11-27 17:45:22,509 [ost-startStop-2] INFO  SpringCamelContext
- Apache Camel 2.20.1 (CamelContext: camel-1) started in 1.037 seconds
2016-11-27 17:45:22,513 [ost-startStop-2] INFO  ContextLoader
- Root WebApplicationContext: initialization completed in 2604 ms
```

Now you need to test that the deployed application runs as expected by sending a web service request using SoapUI, as shown in figure 15.8. Doing this requires you to know the URL to the WSDL the web service runs at, which is http://localhost:9000/inventory?wsdl.

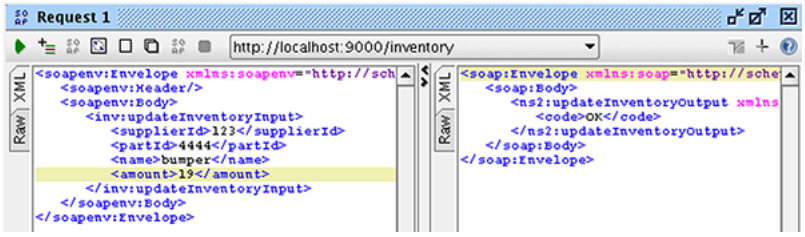


Figure 15.8 Using SoapUI to test the web service from the deployed application in Apache Tomcat

The web service returns `OK` as its reply, and you can also see from the log file that the application works as expected, outputting the inventory being updated:

```
Inventory 4444 updated
```

Another great benefit of this deployment model is that you can tap the servlet container directly for HTTP endpoints. In a standalone Java deployment scenario, you have to rely on the Jetty transport, but in the web deployment scenario, the container already has its socket management, thread pools, tuning, and monitoring facilities. Camel can use these container-provided features if you use the servlet transport for your inbound HTTP endpoints.

In the previously deployed application, you let Apache CXF rely on the Jetty transport. Let's change this to use the existing servlet transports provided by Apache Tomcat.

Using Apache Tomcat for HTTP inbound endpoints

When using Camel in an existing servlet container, such as Apache Tomcat, you may have to adjust Camel components in your application to tap into the servlet container. In the Rider Auto Parts application, it's the CXF component you must adjust. First, you have to add `CXFServlet` to the web.xml file, as shown in the following listing.

Listing 15.7 The web.xml file with `CXFServlet` to tap into Apache Tomcat

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="
           http://java.sun.com/xml/ns/javaee
           http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
           version="3.0">
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
    <servlet>
        <servlet-name>CXFServlet</servlet-name>
        <servlet-class>
            org.apache.cxf.transport.servlet.CXFServlet
        </servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>CXFServlet</servlet-name>
        <url-pattern>/services/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Maven users need to adjust the pom.xml file to depend on the HTTP transport instead of Jetty, as follows:

```
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-transports-http</artifactId>
</dependency>
```

Next, you must adjust the camel-cxf.xml file, as shown in the following listing.

Listing 15.8 Setting up the Camel CXF component to tap into Apache Tomcat

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:cxf="http://camel.apache.org/schema/cxf"
       xsi:schemaLocation="
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd
         http://camel.apache.org/schema/cxf
         http://camel.apache.org/schema/cxf/camel-cxf.xsd">
    <import resource="classpath:META-INF/cxf/cxf.xml"/>
    <import resource=
    "classpath:META-INF/cxf/cxf-servlet.xml"/>  (1)
```

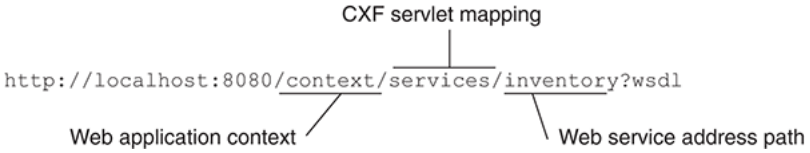
(1) Required import when using servlet

```
    <cxf:cxfEndpoint id="inventoryEndpoint"
    address="/inventory"  (2)
```

(2) Web service endpoint address

```
        serviceClass="camelinaction.inventory.InventoryEndpoint"/>
</beans>
```

To use Apache CXF in a servlet container, you have to import the cxf-servlet.xml resource ❶. This exposes the web service via the servlet container, which means the endpoint address has to be adjusted to a relative context path ❷.



In the previous example, the web service was available at http://localhost:9000/inventory?wsdl. Using Apache Tomcat, the web service is now exposed at a different address. Notice that the TCP port is 8080, which is the default Apache Tomcat setting.

The book's source code contains this example in the chapter15/war-servlet directory. You can package the application using mvn package and then copy the riderautoparts-war-servlet-2.0.0.war file to the webapps directory of Apache Tomcat to hot-deploy the application. Then the web service should be available at this address: http://localhost:8080/riderautoparts-war-servlet-2.0.0/services/inventory?wsdl.

NOTE Camel also provides a lightweight alternative to using CXF in the servlet container. The Servlet component allows you to consume HTTP requests coming into the servlet container in much the same way as you saw here with CXF. You can find more information on the Apache Camel website at http://camel.apache.org/servlet.html.

Table 15.3 lists the pros and cons of the web application deployment model of Camel.

Table 15.3 Pros and cons of embedding Camel in a web application

Pros	Cons
• Taps into the servlet container • Lets the container manage the Camel lifecycle • Benefits the management and monitoring capabilities of the servlet container • Provides familiar runtime platform for operations	• Can create annoying class-loading issues on some web containers

Embedding Camel in a web application is a popular, proven, and powerful way to deploy Camel. Another choice for running Camel applications is using an application server such as WildFly (which is the new name for JBoss Application Server).

15.4.3 Embedded in WildFly

A common way of deploying Camel applications in WildFly is using the web deployment model discussed in the previous section. But WildFly has a slightly different classloading mechanism, so you need to take care when loading things such as type converters. If you're using the camel-bindy component, you'll even have to use a special Camel JBoss component to adapt to this classloading. This component isn't provided out of the box with the Apache Camel distribution because of license implications with WildFly's LGPL license. This component is hosted at Camel Extra (https://github.com/camel-extra/camel-extra), which is a project site for additional Camel components that can't be shipped from Apache.

For most cases, you won't need this component, so we won't dwell on that here. You need to make just one modification to the war-servlet example in the previous section to get it to deploy on WildFly. For WildFly deployments, you can't rely on class resolution by package name. For our single type converter, you need to tell Camel the fully qualified name of the type-converter class to load. You make a change to

src/main/resources/META-INF/services/org/apache/camel/TypeConverter as follows:

```
-camelinaction
+camelinaction.InventoryConverter
```

That's all there is to it!

To deploy the application to WildFly, you start it and copy the WAR file into the standalone/deployments directory. For example, on our laptop, WildFly 11.0.0.Final is started as follows:

```
[janstey@ghost wildfly-11.0.0.Final]$ ./bin/standalone.sh
```

After a couple of seconds, WildFly is ready, and this is logged to the console:

```
00:57:47,693 INFO  [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: I
```

Then the WAR file is copied:

```
[janstey@ghost war-wildfly]$ cp target/riderautoparts-war-wildfly-2.0.0.w
```

You can then keep an eye on the WildFly console as it outputs the progress of the deployment. WildFly uses an embedded instance of Undertow as the servlet container, which is configured to have a similar location as before with Tomcat:

```
http://localhost:8080/riderautoparts-war-wildfly-2.0.0/services/inventor
```

Table 15.4 lists the pros and cons of deploying Camel as a WAR in WildFly.

Table 15.4 Pros and cons of embedding Camel as a WAR in WildFly

Pros	Cons
• Taps into the WildFly container. • Allows your application to use the facilities provided by the Java EE application server. • Lets the WildFly container manage Camel's lifecycle. • Benefits the management and monitoring capabilities of the application server. • Provides a familiar runtime platform for operations.	• Sometimes requires a special Camel component to remedy classloading issues. • WARs have to package many dependencies, which leads to a "fat" deployment artifact.

Deploying Camel as a web application in WildFly is a fairly easy solution. All you have to remember is to use the special Camel JBoss component to let the class loading work for the camel-bindy component and to use the fully qualified class name when loading type converters.

The WildFly-Camel Subsystem

There's another way to deploy Camel applications to WildFly that provides a much tighter integration between Camel and WildFly, and that is using a project called the WildFly-Camel Subsystem.[1] Why use this project? Well, you may not have noticed, but standard WAR-style deployment results in a pretty large WAR for many Camel applications. With WildFly-Camel, you don't have to include any of the Camel libraries in your WAR. Also, the versions of various third-party libraries that come with a Camel component may conflict with what's included in the WildFly container. WildFly-Camel takes care of this dependency management for you so you can focus on the Camel application.

[1] Check out the project source at https://github.com/wildfly-extras/wildfly-camel and docs at https://wildflyext.gitbooks.io/wildfly-camel/content.

You can even write Camel routes directly in WildFly XML configuration, like this:

```
<server xmlns="urn:jboss:domain:4.2">
  <profile>
    ...
    <subsystem xmlns="urn:jboss:domain:camel:1.0">
      <camelContext id="system-context-1">
        <![CDATA[
          <route id="file">
            <from uri="file://target/inventory/updates"/>
            <log message="Received order #{body}"/>
          </route>
        ]]>
      </camelContext>
    </subsystem>
    ...
  </profile>
</server>
```

To try it, you first need to install WildFly-Camel, which is a separate project from WildFly. You can find instructions on how to install WildFly-Camel in the project's documentation at http://wildfly-extras.github.io/wildfly-camel/. At the time of writing, the latest version was 5.0.0, so we used that.

After installing WildFly-Camel, you can boot up WildFly with Camel support by running this:

```
./bin/standalone.sh -c standalone-full-camel.xml
```

If standalone/configuration/standalone-full-camel.xml contained a `<camelContext>` element as shown previously, it'd be started with the container, and you'd see something like this in the logs:

```
17:35:49,523 INFO  [org.wildfly.extension.camel] (MSC service thread 1-8
 Camel context starting: system-context-1
17:35:49,885 INFO  [org.apache.camel.spring.SpringCamelContext] (MSC
 service thread 1-8) Route: file started and consuming from: file://targ
 inventory/updates
17:35:49,885 INFO  [org.apache.camel.spring.SpringCamelContext] (MSC
```

```
 service thread 1-8) Total 1 routes, of which 1 are started.
17:35:49,886 INFO  [org.apache.camel.spring.SpringCamelContext] (MSC
 service thread 1-8) Apache Camel 2.20.1 (CamelContext: system-context-1
 started in 0.465 seconds
```

Putting all your routes into the main WildFly configuration isn't exactly a scalable solution. You need to be able to deploy applications separately from this file. One other way is by defining your `CamelContext` in an XML file under META-INF and named *-camel-context.xml. For example, the previous WAR example had `CamelContext` defined in src/main/webapp/WEB-INF/applicationContext.xml. WildFly-Camel will search in META-INF, so you need to move and rename that file to src/main/resources/META-INF/inventory-camel-context.xml. The difference in the WAR size is apparent right away:

```
wildfly-war/target/riderautoparts-war-wildfly-2.0.0.war 15M
wildfly-camel-war/target/riderautoparts-wildfly-camel-war-2.0.0.war 16K
```

To ensure that you end up with this skinny WAR file, you must use the provided scope for all Camel dependencies in your POM. For our example, you have dependencies like this:

```
    <dependency>
      <groupId>org.apache.camel</groupId>
      <artifactId>camel-core</artifactId>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.apache.camel</groupId>
      <artifactId>camel-spring</artifactId>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.apache.camel</groupId>
      <artifactId>camel-cxf</artifactId>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-web</artifactId>
```

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

    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
      <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>org.apache.cxf</groupId>
      <artifactId>cxf-rt-transports-http</artifactId>
      <version>3.2.1</version>
      <scope>provided</scope>
    </dependency>
```

To deploy this example to a locally running WildFly instance, go to the chapter15/wildfly-camel-war directory of the book's source code and run this:

```
mvn clean install -Pdeploy
```

That builds a skinny WAR with our inventory-camel-context.xml and supporting Java files and then deploys to WildFly.

TIP Back in chapter 7, we also used the WildFly-Camel Subsystem with WildFly Swarm.

You can also avoid XML configuration altogether by annotating your Java Camel routes with CDI annotations. This is covered further in section 15.6.

Table 15.5 lists the pros and cons of deploying Camel using WildFly-Camel.

Table 15.5 Pros and cons of deploying Camel using WildFly-Camel

Pros	Cons
• Taps into the WildFly container • Allows your application to use the facilities provided by the Java EE application server • Lets the WildFly container manage Camel's lifecycle • Benefits the management and monitoring capabilities of the application server • Provides a familiar runtime platform for operations • Deployments are much smaller in size compared to regular WAR style deployment	• Not all Camel components available

The last strategy we'll cover is in a totally different ballpark: using OSGi. OSGi brings the promise of modularity to the extreme.

15.5 Camel and OSGi

OSGi is a layered module system for the Java platform that offers a complete dynamic component model. It's a truly dynamic environment in which components can come and go without requiring a reboot (hot deployment). Apache Camel is OSGi ready, in the sense that all the Camel JAR files are OSGi compliant and are deployable in OSGi containers.

This section shows you how to prepare and deploy the Rider Auto Parts application in the Apache Karaf OSGi runtime. Karaf provides functionality on top of the OSGi container, such as hot deployment, provisioning, local and remote shells, and many other goodies. You can choose between Apache Felix or Eclipse Equinox for the OSGi container. In addition, all of what we're about to discuss also applies to containers that build on top of Karaf, such as JBoss Fuse and Apache ServiceMix.

The example presented here is included with the book's source code in the chapter15/osgi directory.

NOTE This book doesn't go deep into the details of OSGi, which is a complex topic. The basics are covered on Wikipedia (http://en.wikipedia.org/wiki/OSGi), and if you're interested in more information, we highly recommend *OSGi in Action* by Richard S. Hall et al. (Manning, 2011). For more information on the Apache Karaf OSGi runtime, see the Karaf website: http://karaf.apache.org.

The first thing you need to do with the Rider Auto Parts application is make it OSGi compliant (packaged as an OSGi bundle). This involves setting up Maven to help prepare the packaged JAR file so it includes OSGi metadata in the `MANIFEST.MF` entry.

15.5.1 Setting up Maven to generate an OSGi bundle

In the pom.xml file, you have to set the packaging element to bundle, which means the JAR file will be packaged as an OSGi bundle:

```
<packaging>bundle</packaging>
```

To generate the `MANIFEST.MF` entry in the JAR file, you can use the Apache Felix Maven Bundle plugin, which is added to the pom.xml file under the `<build>` section:

```
<build>
  <plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <extensions>true</extensions>
    <configuration>
      <instructions>
        <Bundle-Name>${project.artifactId}</Bundle-Name>
        <Bundle-SymbolicName>riderautoparts-osgi</Bundle-SymbolicName>
        <Export-Package>
            camelinaction,
            camelinaction.inventory
        </Export-Package>
        <Import-Package>*</Import-Package>
        <Implementation-Title>Rider Auto Parts OSGi</Implementation-Titl
        <Implementation-Version>${project.version}</Implementation-Versi
```

```
        </instructions>
      </configuration>
    </plugin>
  </build>
```

The interesting part of `maven-bundle-plugin` is its ability to set the packages to be imported and exported. The plugin is set to export two packages: `camelinaction` and `camelinaction.inventory`. The `camelinaction` package contains the `InventoryRoute` Camel route, and it needs to be accessible by Camel so it can load the routes when the application is started. The `camelinaction.inventory` package contains the generated source files needed by Apache CXF when it exposes the web service.

In terms of imports, the preceding code uses an asterisk, which means the bundle plugin will figure it out by scanning the source for packages used. When needed, you can specify the imports by package name.

The book's source code contains this example in the chapter15/osgi directory. If you run the mvn package goal, you can see the MANIFEST.MF entry being generated in the target/classes/META-INF directory.

You've now set up Maven to build the JAR file as an OSGi bundle, which can be deployed to the container. The next step is to download and install Apache Karaf.

15.5.2 Installing and running Apache Karaf

For this example, you can download and install the latest version of Apache Karaf from http://karaf.apache.org. (At the time of writing, this was Apache Karaf 4.1.2.) Installing is just a matter of extracting the zip or tar.gz file.

To run Apache Karaf, start it from the command line using one of these two commands:

```
bin/karaf         (Linux/Unix)
bin\karaf.bat    (Windows)
```

That should start up Karaf and display a logo when it's ready, like this:

```
[janstey@ghost apache-karaf-4.1.2]$ ./bin/karaf
        __ __                  ____
       / //_/____ _____ _____ _/ __/
      / ,<  / __ `/ ___/ __ `/ /_
     / /| |/ /_/ / /  / /_/ / __/
    /_/ |_|\__,_/_/   \__,_/_/

  Apache Karaf (4.1.2)

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or type 'system:shutdown' or 'logout' to shutdown Karaf.

karaf@root()>
```

Running Karaf like this puts it in a shell mode, where you can enter commands to manage the container.

Now you need to install Camel and Apache CXF before you install the Rider Auto Parts application. Karaf makes installing easier using *features*, which are like *super bundles* that contain a set of bundles installed as a group. Installing features requires you to add the Camel feature descriptions to Karaf, which you do by typing the following command in the shell:

```
feature:repo-add camel 2.20.1
```

You can then type `feature:list` to see all the features available to be installed. Among these should be several Camel-related features. To narrow the list, you can use `grep` just as you would on the command line:

```
feature:list | grep camel
```

The example application requires the `http`, `camel`, and `camel-cxf` features. Type these commands in the shell:

```
feature:install http
feature:install camel
feature:install camel-cxf
```

TIP You can type `bundle:list` to see which bundles have already been installed and their status. The shell has autocompletion, so you can press Tab to see the possible choices. For example, type `bundle` and then press Tab to see the choices.

Let's look at how the Rider Auto Parts application is modified for OSGi deployment.

15.5.3 Using an OSGi Blueprint-based Camel route

The most popular way to deploy Camel routes to an OSGi container such as Apache Karaf is in a Blueprint XML file. *Blueprint* is a dependency-injection framework for OSGi and part of the OSGi specification. In Karaf, the blueprint implementation comes from the Apache Aries project.

Camel routes defined in Blueprint XML look pretty much identical to their equivalents in Spring XML. That's why in many parts of this book we refer to this as the *XML DSL*. But let's call out some differences in our Rider Auto Parts application so you get the full picture. First, the XML file is stored under the OSGI-INF/blueprint directory, which is different from Spring's META-INF/spring. Within the file, the XML looks close to the Spring equivalent:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:cxf="http://camel.apache.org/schema/blueprint/cxf"
           xsi:schemaLocation="
             http://www.osgi.org/xmlns/blueprint/v1.0.0
             http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
             http://camel.apache.org/schema/blueprint/cxf
             http://camel.apache.org/schema/blueprint/cxf/camel-cxf.xsd
             http://camel.apache.org/schema/blueprint
             http://camel.apache.org/schema/blueprint/camel-blueprint.xs

    <!-- CXF endpoint we expose -->
    <cxf:cxfEndpoint id="inventoryEndpoint"
                     address="http://localhost:9000/inventory"
                     serviceClass="camelinaction.inventory.InventoryEndp

    <!-- the order bean contain all the business logic -->
```

```
    <bean id="inventoryService" class="camelinaction.InventoryService"/>

    <!-- the route builder -->
    <bean id="inventoryRoute" class="camelinaction.InventoryRoute"/>

    <!-- Camel -->
    <camelContext id="myCamelContext" xmlns="http://camel.apache.org/sch
        <routeBuilder ref="inventoryRoute"/>
    </camelContext>

</blueprint>
```

All differences are highlighted in bold. As you can see, they're mainly just changes to XML namespaces and schemas.

You're now ready to deploy the Rider Auto Parts application.

15.5.4 Deploying the example

Karaf can install OSGi bundles from various sources, such as the filesystem or the local Maven repository.

To install using Maven, you first need to install the application in the local Maven repository, which you can easily do by running `mvn install` from the chapter15/osgi directory. After the JAR file has been copied to the local Maven repository, you can deploy it to Apache Karaf using the following command from the shell:

```
bundle:install mvn:com.camelinaction/riderautoparts-osgi/2.0.0
```

Upon installing any JAR to Karaf (a JAR file is known as a *bundle* in OSGi terms), Karaf will output on the console the bundle ID it has assigned to the installed bundle, as shown here:

```
Bundle ID: 133
```

You can then type `bundle:list` to see the application being installed:

```
karaf@root()> bundle:list
START LEVEL 100 , List Threshold: 50
 ID | State     | Lvl | Version | Name
```

```
---------------------------------------------------
 57 | Active    |  50 | 2.20.1  | camel-blueprint
 58 | Active    |  50 | 2.20.1  | camel-catalog
 59 | Active    |  50 | 2.20.1  | camel-commands-core
 60 | Active    |  50 | 2.20.1  | camel-core
 61 | Active    |  50 | 2.20.1  | camel-cxf
 62 | Active    |  50 | 2.20.1  | camel-cxf-transport
 63 | Active    |  50 | 2.20.1  | camel-spring
 64 | Active    |  80 | 2.20.1  | camel-karaf-commands
133 | Installed |  80 | 2.0.0   | riderautoparts-osgi
```

Notice that the application isn't started. You can start it by entering `bundle:start 133`, which changes the application's status when you do a `bundle:list` again:

```
133 | Active |  80 | 2.0.0   | riderautoparts-osgi
```

The application is now running in the OSGi container.

TIP You can install and start the bundle in a single command using the `-s` option on the `bundle:install` command, like this: `bundle:install -s mvn:com.camelinaction/riderautoparts-osgi/2.0.0`.

How can you test that it works? Start by checking the log with the `log:display` command. Among other things, it should indicate that Apache Camel has been started:

```
2016-12-20 20:02:07,174 | INFO  | nsole user karaf | BlueprintCamelConte
| 60 - org.apache.camel.camel-core - 2.20.1 | Apache Camel 2.20.1
(CamelContext: myCamelContext) started in 0.496 seconds
```

You can then use SoapUI to send a test request. The WSDL file is available at http://localhost:9000/inventory?wsdl.

When you're done testing the application, you may want to stop the OSGi container, which you can do by executing the `shutdown` command from the shell.

TIP You can tail the Apache Karaf log file using `tail -f log/karaf.log`. Note that this isn't done from within the Karaf shell but from the regular shell on your operating system. If you prefer to stay within Karaf, the Karaf shell also provides a `log:tail` command.

Taking our example a little further, let's look at using an OSGi-managed service factory to spin up multiple route instances solely by modifying configuration.

15.5.5 Using a managed service factory to spin up route instances

Managed service factories (MSFs) in OSGi are a useful abstraction in dealing with multiple services that vary by only a few parameters. Going back to our Rider Auto Parts example, let's say you need to spin up several instances of the file inventory route, with each instance consuming from a different directory. Because only the file URI would be changing, it wouldn't make sense to create a new `RouteBuilder` class for each variation. A much more efficient way to do this in OSGi is to have a template route that accepts a few parameters and then have an MSF spin up a new instance based on configuration updates. For example, if you extract the file consumer route into its own template class, you'd have a reusable route:

```
public class FileInventoryRoute extends RouteBuilder {

    private String inputPath;
    private String routeId;

    @Override
    public void configure() throws Exception {
        from(inputPath)
            .routeId(getRouteId())
            .split(body().tokenize("\n"))
                .convertBodyTo(UpdateInventoryInput.class)
                .to("direct:update");
    }
```

```
  // getter/setter omitted
```

Before this route would be useful, you'd have to set the `inputPath` and `routeId` fields. Looking at our Blueprint XML file from the previous section, you have to add only one extra element:

```
<bean id="camelServiceFactory" class="camelinaction.FileInventoryService
    <property name="bundleContext" ref="blueprintBundleContext"/>
    <property name="camelContext" ref="myCamelContext"/>
</bean>
```

This starts up your MSF `FileInventoryServiceFactory`, which can spin up additional instances of your `FileInventoryRoute` template route. The service factory is shown in the following listing.

Listing 15.9 A managed service factory for spinning up `FileInventoryRoute` instances

```
public class FileInventoryServiceFactory
  implements ManagedServiceFactory {
```

A service factory must implement org.osgi.service.cm.ManagedServiceFactory

```
    private static final Logger LOG = LoggerFactory.getLogger(FileInvent

    private CamelContext camelContext;
    private BundleContext bundleContext;
    private Map<String, FileInventoryRoute> routes
    = new HashMap<String, FileInventoryRoute>();
```

A map between persistent identifiers (PIDs) and routes

```
    private ServiceRegistration registration;
```

```
    @Override
    public String getName() {
        return "FileInventoryRouteCamelServiceFactory";
    }

    @SuppressWarnings("unchecked")
    public void init() {
        Dictionary properties = new Properties();
    properties.put( Constants.SERVICE_PID,
```

Register the service factory and provide the factory PID to watch for configuration

```
        "camelinaction.fileinventoryroutefactory");
    registration = bundleContext.registerService(
        ManagedServiceFactory.class.getName(),
        this, properties);

        LOG.info("FileInventoryRouteCamelServiceFactory ready to accept
        "new config with PID=camelinaction.fileinventoryroutefactory-xxx
    }

    @Override
  public void updated(String pid,
```

The updated method is called when configuration changes

```
    Dictionary<String, ?> properties)
    throws ConfigurationException {

    String path = (String) properties.get("path");
```

The sole configuration property you'll be using for route instances

```

        LOG.info("Updating route for PID=" + pid + " with new path=" + p
```

```
    deleted(pid);
```

Need to remove the old route before adding the updated one

```
        // now we create a new route with update path
        FileInventoryRoute newRoute = new FileInventoryRoute();
        newRoute.setInputPath(path);
        newRoute.setRouteId("file-" + pid);

        try {
    camelContext.addRoutes(newRoute);
```

Add the new route to the CamelContext

```
        } catch (Exception e) {
            LOG.error("Failed to add route", e);
        }
        routes.put(pid, newRoute);
    }

    @Override
public void deleted(String pid) {
```

The deleted method is called when the corresponding configuration is deleted

```
        LOG.info("Deleting route with PID=" + pid);

        try {
            FileInventoryRoute route = routes.get(pid);
            if (route != null) {
                camelContext.stopRoute(route.getRouteId());
                camelContext.removeRoute(route.getRouteId());
                routes.remove(pid);
```

```
            }
        } catch (Exception e) {
            LOG.error("Failed to remove route", e);
        }
    }

    ...
}
```

To explain what's going on here, you need to know about how configuration works in an OSGi container such as Karaf. Configuration is managed by the Configuration Admin service. Each group of configuration items tracked by Configuration Admin has a unique persistent identifier (PID). For our MSF, you assign a factory PID `camelinaction.fileinventoryroutefactory`, which means Configuration Admin will watch for configuration with PIDs such as `camelinaction.fileinventoryroutefactory-foo`, `camelinaction.fileinventoryroutefactory-bar`, and so forth. They just have to start with the factory PID for your MSF to get picked up. To put it more concretely, in the Karaf distribution, if you add a camelinaction.fileinventoryroutefactory-path1.cfg file to the etc directory with these contents

```
path=file://target/inventory/updates
```

that would get picked up by Configuration Admin, which would in turn call the updated method on our MSF. This would add a new route to `CamelContext`. In the logs, you'd see something like this:

```
2016-12-21 19:31:05,127 | INFO  | f7-466f8727ed59) | BlueprintCamelCont
```

You could add as many extra configuration files as you want—one for each distinct route. This also doesn't all have to be done though configuration files; Configuration Admin can be controlled via Karaf `config:*` commands, JMX, or even programmatically.

Check out the MSF example in the chapter15/osgi-msf directory of the book's source code.

You've now seen how to deploy a Camel application into an OSGi container. Table 15.6 lists the pros and cons of deploying Camel in an OSGi container.

Table 15.6 Pros and cons of using OSGi as a deployment strategy

Pros	Cons
• Uses OSGi for modularity • Provides classloader isolation and hot deployment	• Involves a learning curve for OSGi • Unsupported third-party frameworks; some frameworks have yet to become OSGi compliant • Requires extra effort to decide what package imports and exports to use for your module

We've only scratched the surface of OSGi in this chapter. If you go down that path, you'll need to pick up other books, because OSGi is a big concept to grasp and master. It's also powerful and allows you to create dynamic applications. On the other hand, the path of the web application is the beaten track, and plenty of materials and people can help you if you come up against any problems.

15.6 Camel and CDI

Contexts and Dependency Injection (CDI) is the part of the Java EE platform focused on the following:

- Dependency injection
- Lifecycle management of stateful objects bound to contexts

It's easy to get started with CDI. You typically need to add a single dependency to your project:

```
<dependency>
   <groupId>javax.enterprise</groupId>
   <artifactId>cdi-api</artifactId>
   <version>1.2</version>
```

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

Use CDI annotations and then deploy to a container that supports CDI such as WildFly, WebSphere, or Apache TomEE. Many containers support CDI using the reference implementation, the JBoss Weld project.

Instead of making you manually instantiate and wire `CamelContext`s and routes together, Camel (the camel-cdi component in particular) automatically deploys and configures `CamelContext` for you. It searches for any routes as well and adds them to the context. Common Camel services such as endpoints, `ProducerTemplate`s, `TypeConverter`s, and even `CamelContext` itself are injectable via CDI annotations.

We first covered CDI in chapter 7, so many of the basic concepts are discussed there. Testing CDI applications was discussed in chapter 9. If you're unfamiliar with CDI, you should review those chapters first. This section focuses on deployment to the WildFly container.

NOTE It's possible to deploy CDI-based routes in OSGi, but this support is deprecated in Camel as of version 2.19.0, so we don't cover it here. You may still be able to find an example in the distribution in the examples/camel-example-cdi-osgi directory.

Back in section 15.4.3 we discussed how to deploy Camel routes to the WildFly container. We also mentioned how to use an extension project called WildFly-Camel to make deployment easier. We'll be assuming a WildFly-Camel-enhanced container as well in this section.

First, let's go over what you have to add in your Maven pom.xml file. In the dependencies section, you have this:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-cdi</artifactId>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>javax.enterprise</groupId>
```

```
    <artifactId>cdi-api</artifactId>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.wildfly.camel</groupId>
    <artifactId>wildfly-camel-subsystem-core</artifactId>
    <scope>provided</scope>
  </dependency>
```

The first two are probably obvious if you've read the previous sections that covered CDI. The last one is for an annotation specifically for WildFly-Camel. Although not absolutely necessary, it's helpful to import the WildFly-Camel BOM so that you don't need to specify any dependency versions. You can do so as follows:

```
  <dependencyManagement>
    <dependencies>
      <!-- WildFly Camel -->
      <dependency>
        <groupId>org.wildfly.camel</groupId>
        <artifactId>wildfly-camel</artifactId>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
      <dependency>
        <groupId>org.wildfly.camel</groupId>
        <artifactId>wildfly-camel-patch</artifactId>
        <type>pom</type>
      </dependency>
    </dependencies>
  </dependencyManagement>
```

Moving on to our route, you don't have to add any CDI annotations to it for camel-cdi to pick it up. For WildFly-Camel to recognize it, though (as of version 5.0.0), you need either an `org.wildfly.extension.camel.CamelAware` or `org.apache.camel.cdi.ContextName` annotation. The route is shown here:

```
  import org.apache.camel.builder.RouteBuilder;
  import org.wildfly.extension.camel.CamelAware;
```

```
import camelinaction.inventory.UpdateInventoryInput;

@CamelAware
public class InventoryRoute extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        // this is the file route which is started 2nd last
        from("file://target/inventory/updates")
            .routeId("file").startupOrder(2)
            .split(body().tokenize("\n"))
                .convertBodyTo(UpdateInventoryInput.class)
                .to("direct:update");

        // this is the shared route which then must be started first
        from("direct:update")
            .routeId("update").startupOrder(1)
            .to("bean:inventoryService?method=updateInventory");
    }
}
```

As you can see, the route doesn't look that different from before except for the `@CamelAware` annotation. In previous examples, the inventory service was defined in Spring or Blueprint as follows:

```
<bean id="inventoryService" class="camelinaction.InventoryService"/>
```

But here, you have no such definition. With CDI, you can create named beans via annotations. Let's look at your `InventoryService` bean in action:

```
@Named("inventoryService")
public class InventoryService {

    private Random ran = new Random();

    public String xmlToCsv(UpdateInventoryInput input) {
        return input.getSupplierId() + "," + input.getPartId()
            + "," + input.getName() + "," + input.getAmount();
    }
```

```
    public UpdateInventoryOutput replyOk() {
        UpdateInventoryOutput ok = new UpdateInventoryOutput();
        ok.setCode("OK");
        return ok;
    }

    public void updateInventory(UpdateInventoryInput input) throws Excep
        int sleep = ran.nextInt(1000);
        Thread.sleep(sleep);

        System.out.println("Inventory " + input.getPartId() + " updated"
    }

}
```

Here you can see that you use the `javax.inject.Named` annotation to create a named bean instance you can refer to in your Camel route. The camel-cdi component starts up `org.apache.camel.cdi.CdiCamelRegistry` to hold such beans in `CamelContext`. To try this example for yourself, change to the chapter15/cdi directory of the book's source and run this command:

```
mvn clean install -Pdeploy
```

This deploys the example to a locally running instance of WildFly (if one is running). One handy tip for deploying during development is to use wildfly-maven-plugin as we have in this example. To use this plugin, you can add something like the following to your Maven pom.xml file:

```
  <build>
    <plugins>
      <plugin>
        <groupId>org.wildfly.plugins</groupId>
        <artifactId>wildfly-maven-plugin</artifactId>
        <configuration>
          <skip>${deploy.skip}</skip>
        </configuration>
        <executions>
          <execution>
             <id>wildfly-deploy</id>
             <phase>install</phase>
```

```
            <goals>
              <goal>deploy-only</goal>
            </goals>
          </execution>
          <execution>
            <id>wildfly-undeploy</id>
            <phase>clean</phase>
            <goals>
              <goal>undeploy</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>

  <!-- Profiles -->
  <profiles>
      <profile>
          <id>deploy</id>
          <properties>
              <deploy.skip>false</deploy.skip>
          </properties>
      </profile>
  </profiles>
```

Notice that it's turned off by default (because starting WildFly is a separate manual step), so you have to provide the `deploy` profile to enable it.

15.7 Summary and best practices

This chapter explored the internal details of the way Camel starts up. You learned which options you can control and whether routes should be autostarted. You also learned how to dictate the order in which routes should be started.

More importantly, you learned how to shut down a running application in a reliable way without compromising the business. You learned about Camel's graceful shutdown procedures and what you can do to reorder your routes to ensure a better shutdown process, when needed. You also learned how to stop and shut down routes at runtime. You can do this

programmatically, which allows you to fully control when routes are operating and when they aren't.

In the second part of this chapter, you explored the art of deploying Camel applications as standalone Java applications, as web applications, in Java EE containers, as CDI applications, and by running Camel in an OSGi container. Remember that the deployment strategies covered in this book aren't all of your options. For example, we covered several options popular with microservices in chapter 7 and will cover Docker containers in chapter 18.

Here are some pointers to help you with running and deployment:

- *Ensure reliable shutdown*—Take the time to configure and test that your application can be shut down in a reliable manner. Your application is bound to be shut down at some point, whether for planned maintenance, upgrades, or unforeseen problems. In those situations, you want the application to shut down in a controlled manner without negatively affecting your business.
- *Use an existing runtime environment*—Camel is agile, flexible, and can be embedded in whatever production setup you may want to use. Don't introduce a new production environment just for the sake of using Camel. Use what's already working for you, and test early in the project that your application can be deployed and run in the environment.

In the next chapter, you'll tour Camel's extensive monitoring and management facilities.