7 Microservices

This chapters covers

- Microservices overview and characteristics
- Developing microservices with Camel
- WildFly Swarm and Camel
- Spring Boot and Camel
- Designing for failures
- Netflix Hystrix circuit breakers

We wanted to bring you the most up-to-date material, so, because the IT industry is evolving constantly, we decided to wait and write chapters 7 and 18 last. This chapter covers using microservices with Camel, and chapter 18 is a natural continuation that goes even further and takes the microservices world into the cloud by using container technology such as Docker and Kubernetes. This chapter stays down-to-earth and is mainly focused on running locally on a single computer or laptop.

We initially wrote a lengthy chapter introduction to talk about how the software is eating the world, how businesses are being disrupted, and how new emerging businesses are doing business in a faster and more agile way, where microservices are instrumental. But we felt there are much better authors who can explain all that, and we mention two of them at the start of section 7.1.

The first section covers six microservices characteristics that you should have in mind when reading the remainder of this chapter.

Section 7.2 jumps into action by showing you how to build microservices with Camel by using various JVM frameworks, starting with plain, standalone Camel and moving all the way to using popular Java microservice runtimes such as Spring Boot and WildFly Swarm.

Section 7.3 covers how microservices can call each other. The section examines a use case at Rider Auto Parts as you build a prototype consisting of six microservices.

The first part of section 7.4 covers strategies you can use for building fault-tolerant microservices. The second part continues the Rider Auto Parts use case and walks you through improving those six microservices to become a fully functional fault-tolerant system. As part of the exercise, you'll learn how to use the popular Hystrix circuit breaker from the Netflix OSS stack and see firsthand how elegantly Camel integrates with it.

The chapter ends on a high note with some eye candy of a live graph that charts the state of the circuit breakers.

Let's start from the beginning and set the stage so you can understand the forces at play.

7.1 Microservices overview

The more abstract and fluffy a concept seems, the harder it is to explain and get everybody on board so they have a similar view of the landscape. Microservices is no exception. Don't mistake microservices as only a technology discussion. It's equal parts organizational structure, culture, and human forces.

Microservices aren't a new technical invention, like an operating system or programming language. Microservices are a style of software system that emerges when you stick to a set of design principles and methodologies. This section focuses on the technical characteristics of microservices. If you're interested in the nontechnical side, we recommend (among others) the following:

- *Building Microservices* by Sam Newman (O'Reilly, 2015)
- *Microservices for Java Developers* by Christian Posta, a free ebook published by O'Reilly (https://developers.redhat.com/promotions/microservices-for-java-developers/)

In this section, we cover six microservices characteristics that we consider most relevant to this book at the time of writing. In addition to learning about these characteristics and the common practices used for implementing microservices, you'll see the role Camel plays.

7.1.1 Small in size

A fundamental principle of a microservice is hinted at by its name: a *microservice* is small (micro) in size. The mantra of a microservice is said *to be small, nimble, do one thing only, and do that well*. This is reinforced by Robert C. Martin's definition of the *single responsibility principle*, which states: "Gather together those things that change for the same reason, and separate those things that change for different reasons." Microservices takes the same approach by focusing on service and business boundaries, making it obvious where the code lives for a given functionality. And by keeping the service focused on an explicit boundary, we avoid any temptation for it to grow too large, with all the complexities that can introduce. How to measure the size of a microservice is debatable—do you use the number of code lines, the number of classes, or some other measurement? A good measure of the size of a microservice is that a single person should be able to manage all of it in their head.

Camel shines in this area. Camel applications are inherently small. For example, a Camel application that receives events over HTTP or messaging, transforms and operates on the data, and sends a response back can be about 50 to 100 lines of code. That's small enough to fit into the head of one person who can build end-to-end testing and do code refactoring. And if the microservice is no longer needed, the code can be thrown away without losing or wasting hundreds of hours of implementing and coding.

7.1.2 Observable

Services should expose information that describes the state of the service and the way the service performs. Running microservices in production should allow easy management and monitoring. A good practice is to provide health checks in your microservices, accessible from known endpoints that the runtime platform uses for constant monitoring. This kind of information is provided out of the box by Camel, and you can make it available over JMX, HTTP, and Java microservice runtimes such as Spring Boot and WildFly Swarm.

TIP Chapter 16 covers details of management and monitoring with Camel.

7.1.3 Designed for failure

A consequence of building distributed systems by using microservices is that applications need to be designed so they can tolerate failures of services. Any upstream service can fail for any number of reasons, which imposes upon the client to respond to the failure as gracefully as possible. Because a service can come and go at any time, it's important to be able to detect the failures quickly and, if possible, automatically restore the service on the runtime platform—hence be self-monitoring.

Camel has many EIP patterns to help design with failures. The Dead Letter Channel EIP can ensure that messages aren't lost in the event of service failures. The error handler can perform redelivery attempts to attempt to mitigate a temporary service failure. The load balancer can balance the service call between multiple machines hosting the same service. The Service Call EIP (covered in chapter 17) can, based on service availability, call a healthy service on a set of machines hosting the same service. The Circuit Breaker EIP can be used to protect unresponsive upstream services from receiving further calls during a period of time while the circuit breaker is open. We cover designing for failures and using the Circuit Breaker EIP in section 7.4.

7.1.4 Highly configurable

A sound practice is to never hardcode or commit your environment-specific configuration, such as usernames and passwords, in your source code. Instead, keep the configuration outside your application in external configuration files or environment variables. A good mantra to help you remember this practice goes like this: *Separate config from code*.

By keeping configuration outside your application, you can deploy the same application to different environments by changing only the configuration and not the source code. With the rise of container platforms such as Docker Swarm, Kubernetes, Apache Mesos, and Cloud Foundry, this has become common practice: you can safely move a Docker image (with your application) between environments (test, staging, pre-production, and production) by applying environment-specific configuration to the Docker image.

Camel applications are highly configurable; you can easily externalize configuration of all your Camel routes, endpoints, and so on. This configuration can then easily be configured per environment without having to recompile your source code or set up credentials, encryption, threading models, and whatnot.

7.1.5 Smart endpoints and dumb pipes

Over the last couple of decades, we've seen different integration products based on the principles of Enterprise Application Integration (EAI), enterprise service bus (ESB), and service-oriented architecture (SOA). A common denominator for these products is that they include sophisticated facilities for message routing, mediation, transformation, business rules, and much more. The most complex of these products rely on overly complex protocols such as Web Service Choreography (WS-Choreography), Business Process Execution Language (BPEL), or Business Process Model and Notation (BPMN).

Microservices favor RESTful and lightweight protocols rather than complex protocols such as SOAP. The style used by microservices is *smart endpoints and dumb pipes*. Applications built from microservices aim to be as decoupled and as cohesive as possible. They own their own domain logic and act more like the UNIX shell commands (pipes and filters): receive an event, apply some logic, and return a response.

Camel supports both worlds. You can find components supporting SOAP Web Services, BPMN, and JMS. For microservices, you can find a lot of components supporting REST, as well as a Rest DSL, which makes defining RESTful services much easier (we cover this in chapter 10). There are plenty of components for lightweight messaging, such as Advanced Message Queuing Protocol (AMQP), Amazon Simple Queue Service (SQS), Amazon Simple Notification Service (SNS), Kafka, Message Queue

Telemetry Transport (MQTT), gRPC, and Protocol Buffers. And each new Camel release brings more components.

7.1.6 Testable

Let's step back and ask ourselves, why are we using microservices? There may be many reasons for doing so, but a key goal should be delivering services faster into production. By having many more microservices deployed into production, developers and operations teams can have confidence that what's being deployed is working as intended, and can help ensure that testing microservices is faster and easier—and as automated as possible.

Camel has extensive support for testing at different levels, including unit and integration tests. With Camel, you can test your application in isolation by mocking external endpoints, simulating events, and verifying that all is as expected. We devote all of chapter 9 to this.

We'll touch on these microservices characteristics throughout the rest of the chapter.

It's time to leave the background theory, get down to action, and show some code. The next section teaches you how to get Camel riding on the hype of microservice waves.

7.2 Running Camel microservices

Which microservice runtimes does Camel support? The answer is *all of them*. Camel is just a library you include in the JVM runtime, and it runs anywhere. This section walks you through running Camel in some of the most popular microservice runtimes:

- *Standalone*—Running just Camel
- *CDI*—Running Camel with CDI
- *WildFly Swarm*—We's see how Camel runs with the lightweight Java EE server
- *Spring Boot*—Running Camel with Spring Boot

As you can see, there are four runtimes in our bulleted list, so there's a lot to cover. We start with just Camel and then work our way down the list, ending with some of the newer and more eccentric microservice runtimes.

Before continuing, we want to mention that chapter 20 (available online only) covers another microservice framework called Vert.X, which you can use to build reactive applications. Vert.X isn't your typical framework for running Camel applications, but it has potential, so we want to make sure you hear about it.

7.2.1 Standalone Camel as microservice

Throughout this section, we use a basic Camel example as the basis of a microservice. The service is an HTTP hello service that returns a simple response, as shown in the following Camel route:

```
public class HelloRoute extends RouteBuilder {

  public void configure() throws Exception {
    from("jetty:http://localhost:8080/hello")
      .transform().simple("Hello from Camel");
  }
}
```

To run this service by using standalone Camel, you follow these three steps:

1. Create a `CamelContext`.
2. Add the route with the service to the `CamelContext`.
3. Start the `CamelContext`.

```
public class HelloCamel {

  public static void main(String[] args) throws Exception {
    CamelContext context = new DefaultCamelContext();
```

Creates the CamelContext

```
    context.addRoutes(new HelloRoute());
```

Adds the route with the service

```
    context.start();
```

Starts Camel

```
    Thread.sleep(Integer.MAX_VALUE);  1
```

1

Keeps Camel (the JVM) running

```
    }
  }
```

You can try this example from the source code in the chapter7/standalone directory by running the following Maven goal:

```
mvn compile exec:java -Pmanual
```

When the example is running, you can access the service from a web browser on http://localhost:8080/hello.

Have you noticed a code smell in the `HelloCamel` class? Yeah, at ❶ we had to use `Thread.sleep` to keep Camel running. Whenever there's a code smell, there's usually a better way with Camel.

Running Camel standalone by using the Camel Main class

Camel provides an `org.apache.camel.main.Main` class, which makes it easier to run Camel standalone:

```
public class HelloMain {

    public static void main(String[] args) throws Exception {
        Main main = new Main();
```

Creates the Main class

```
main.addRouteBuilder(new HelloRoute());
```

Adds the route with the service

```
main.run();  ❶
```

❶

Keeps Camel (the JVM) running

```
    }
}
```

As you can see, running Camel by using the `Main` class is easier. The `run` method ❶ is a blocking method that keeps the JVM running. The `Main` class also ensures that you trigger on JVM shutdown signals and perform a graceful shutdown of Camel so your Camel applications terminate gracefully.

You can try this example from the source code in the chapter7/standalone directory by running the following Maven goal:

```
mvn compile exec:java -Pmain
```

To terminate the JVM, press Ctrl-C, and notice from the console log that Camel is stopping.

The `Main` class has additional methods for configuration, such as property placeholders, and for registering beans in the registry. You can find details by exploring the methods available on the main instance.

Summary of using standalone Camel for microservices

Running Camel standalone is the simplest and smallest runtime for running Camel—it's just Camel. You can quickly get started, but it has its limits. For example, you need to figure out how to package your application code with the needed JARs from Camel and third-party dependencies, and how to run that as a Java application. You could try to build a fat JAR, but that creates problems if duplicate files need to be merged together. But if you're using Docker containers, you can package your application together in a Docker image, with all the JARs and your application code separated, and still make it easy to run your application. We cover Camel and Docker in chapter 18.

The `Main` class has its limitations, and you may want to go for some of the more powerful runtimes, such as WildFly Swarm or Spring Boot, which we cover in sections to follow. But first, let's talk about Camel and CDI.

7.2.2 CDI Camel as microservice

What is CDI? Contexts and Dependency Injection (CDI) is a Java specification for dependency injection that includes a set of key features:

- *Dependency injection*—Dependency injection of beans.
- *POJOs*—Any kind of Java bean can be used with CDI.
- *Lifecycle management*—Perform custom action on bean creation and destruction.
- *Events*—Send and receive events in a loosely coupled fashion.
- *Extensibility*—Pluggable extensions can be installed in any CDI runtime to customize behavior.

Camel provides support for CDI by the camel-cdi component, which contains a set of CDI extensions that make it easy to set up and bootstrap Camel with CDI.

TIP You can find great CDI documentation at the JBoss Weld project: http://docs.jboss.org/weld/reference/latest/en-US/html/.

Hello service with CDI

The Camel route for the hello microservice can be written using CDI, as shown in the following listing.

Listing 7.1 Camel route using CDI

```
@Singleton    1
```

1

Defines scope of bean as Singleton

```
public class HelloRoute extends RouteBuilder {

    public void configure() throws Exception {
        from("jetty:http://localhost:8080/hello")
            .transform().simple("Hello from Camel");
    }
}
```

As you can see, the Camel route is just regular Java DSL code without any special CDI code. The only code from CDI is the `@Singleton` (`javax.inject.Singleton`) annotation ❶.

To make running Camel in CDI easier, you can use an `org.apache.camel.cdi.Main` class:

```
public class HelloApplication {

    public static void main(String[] args) throws Exception {
        Main main = new Main();
        main.run();
    }
}
```

You can try this example from the chapter7/cdi-hello directory by running the following Maven goal:

```
mvn compile exec:java
```

The hello service can be accessed from a web browser at http://localhost:8080/hello. To terminate the JVM, you can press Ctrl-C.

This example is simple, so let's push the bar a little and configure Camel using CDI.

Configuring CDI applications

The code in listing 7.1 returns a reply message that's hardcoded. Let's improve this and externalize the configuration to a properties file. Camel's property placeholder mechanism is a component (`PropertiesComponent`) with the special name properties. In CDI, you use `@Produces` and `@Named` annotations to create and configure beans. You use this to create and configure an instance of `PropertiesComponent`, as shown in the following listing.

Listing 7.2 Configuring Camel property placeholder by using CDI `@Produces`

```
@Singleton
public class HelloConfiguration {

    @Produces  (1)
```

(1) Declares that this method is producing (creating) a bean

```
    @Named("properties")  (2)
```

(2) ... with the name properties

```
    PropertiesComponent propertiesComponent() {  (3)
```

(3) ... and of type PropertiesComponent

```
        PropertiesComponent component = new PropertiesComponent();
        component.setLocation("classpath:hello.properties");
```

```
            return component;
        }
    }
```

Here you've created a class named `HelloConfiguration` that you use to configure your application with CDI. It's good practice to have one or more configuration classes separated from your business and Camel routing logic.

To set up Camel's property placeholder, you use a Java method that creates, configures, and returns an instance of `PropertiesComponent` ❸. Notice that the code in this method is plain Java code without using CDI. Only on the method signature do you use CDI annotations, to instruct CDI that this method is able to produce ❶ a bean of type `PropertiesComponent` with the ID `properties` ❷.

This example is shipped with the book's source code in the chapter7/cdi directory, which you can try by using the following Maven goal:

```
mvn compile exec:java
```

Let's raise the bar one more time and show you the most-used feature of CDI: using CDI dependency injection in your POJO beans with `@Inject`.

Dependency injection using @Inject

We've refactored the example to use a POJO bean to construct the reply message of the hello service:

```
@Singleton
public class HelloBean {

    public String sayHello(@PropertyInject("reply") String msg)    ❶
```

❶ Injects Camel property placeholder with key reply as parameter

```
            throws Exception {
        return msg + " from " + InetAddressUtil.getLocalHostName();
```

```
    }
}
```

The `HelloBean` class has a single method named `sayHello` that creates the reply message. The method takes one argument as input that has been annotated with Camel's `@PropertyInject` annotation. `@ProjectInject` is configured with the value `reply` ❶, which corresponds to the property key. The property placeholder file should contain this key:

```
reply=Hello from Camel CDI with properties
```

The Camel route for the hello service needs to be changed to use the `HelloBean;` this can be done by dependency-injecting the bean into the `RouteBuilder` class, as shown in the following listing.

Listing 7.3 Injecting bean using CDI `@Inject`

```
@Singleton
public class HelloRoute extends RouteBuilder {

    @Inject    ❶
```

❶ Dependency injects the HelloBean using @Inject

```
    private HelloBean hello;

    public void configure() throws Exception {
        from("jetty:http://localhost:8080/hello")
            .bean(hello, "sayHello");    ❷
```

❷ Calls the method sayHello on the injected HelloBean instance

```
    }
}
```

Because you want to use the `HelloBean` in the Camel route, you can instruct CDI to dependency-inject an instance of the bean by declaring a field and annotate the field with `@Inject` ❶. The bean can then be used in the Camel route as a bean method call ❷.

This example is provided with the book's source code in the chapter7/cdi directory. With the source code, you see how to use `@Inject` in unit tests with CDI. We've devoted all of chapter 9 to the topic of testing and cover testing CDI in much more detail.

The camel-cdi component provides a specialized dependency injection to inject Camel endpoints in POJOs.

Using @Uri to inject Camel endpoint

In your POJOs or Camel routes, you may want to inject a Camel endpoint. For example, instead of using string values for Camel endpoints in Camel routes, you can use endpoint instances instead:

```
@Singleton
public class HelloRoute extends RouteBuilder {

    @Inject
    private HelloBean hello;

    @Inject @Uri("jetty:http://localhost:8080/hello")    ❶
```

❶

Dependency injects Camel endpoint with the given URI

```
    private Endpoint jetty;

    public void configure() throws Exception {
        from(jetty)    ❷
```

❷

Uses the injected Camel endpoint in the Camel route

```
                .bean(hello, "sayHello");
        }
    }
```

In the `HelloRoute` class, we use a field to define the Camel endpoint for the incoming endpoint ❶. Notice that the field is annotated with both `@Inject` (CDI) and `@Uri` (camel-cdi). In the Camel route, the endpoint is used in the form that accepts `org.apache.camel.Endpoint` as a parameter type ❷.

You can also use `@Inject @Uri` to inject `FluentProducerTemplate`, which makes it easy to send a message to the given endpoint. The following listing uses this in unit testing the example.

Listing 7.4 Using `@Inject @Uri` to inject `FluentProducerTemplate`

```
@RunWith(CamelCdiRunner.class)     1
```

1

Unit testing with camel-cdi

```
public class HelloRouteTest {

    @Inject @Uri("jetty:http://localhost:8080/hello")     2
```

2

Dependency injects FluentProducerTemplate with default URI

```
    private FluentProducerTemplate producer;

    @Test
    public void testHello() throws Exception {
        String out = producer.request(String.class);     3
```

❸

Sends (InOut) empty message to the endpoint and receives response as String type

```
            assertTrue(out.startsWith("Hello from Camel CDI"));
        }
    }
```

The class is annotated with `@RunWith(CamelCdiRunner.class)` ❶, enabling running the test with Camel and CDI. `CamelCdiRunner` is provided by the camel-test-cdi component. The test uses `FluentProducerTemplate` ❷ to send a test message to the Camel hello service ❸. Notice that the template is injected using both `@Inject` and `@Uri` ❷. The endpoint URI defined in `@Uri` represents the default endpoint, which is optional. The unit test could have been written as follows:

```
@Inject @Uri    ❶
```

❶

Injects template without a default endpoint URI

```
private FluentProducerTemplate producer;

@Test
public void testHello() throws Exception {
    String out = producer.to("jetty:http://localhost:8080/hello")    ❷
```

❷

Specifies the URI of the endpoint to send the message to

```
                        .request(String.class);
    assertTrue(out.startsWith("Hello from Camel CDI"));
}
```

`FluentProducerTemplate` isn't configured with an endpoint URI ❶, which we then must provide when we send the message ❷.

This example is provided with the accompanying source code in the chapter7/cdi directory.

Before reaching the end of our mini coverage of using Camel CDI, we'll show you one last feature of CDI that you can use with Camel: event listening.

Listening to Camel events using CDI

CDI supports an event notification mechanism that allows you to listen for certain events and react to them. This can be used to listen to Camel lifecycle events such as when Camel Context or routes start or stop. The following code shows how to listen for Camel starting:

```
@Singleton
public class HelloConfiguration {

  void onContextStarted(@Observes CamelContextStartedEvent event) {   ❶
```

❶ Listens for CamelContextStartedEvent to happen

```
    System.out.println("*****************************************");
    System.out.println("* Camel started " + event.getContext().getName()
    System.out.println("*****************************************");
  }
}
```

To listen for events in CDI, you declare a method with a parameter that carries the event class to listen for. The parameter must be annotated with the `@Observes` annotation ❶.

This concludes our coverage of using Camel with CDI, including coverage of three of the most common, key features of CDI and camel-cdi:

- Dependency injection using `@Inject` and `@Produces`
- Injecting a Camel endpoint using `@Uri`

- Bean lifecycle using scopes such as `@Singleton`
- Event listening using `@Observes`

Before moving on to the next Camel microservice runtime, we want to share some of our thoughts on using Camel CDI.

Summary of using Camel CDI for microservices

Developers wanting to build microservices by using Java code can find value in using a dependency injection framework such as CDI or Spring Boot. Camel users who find the XML DSL attractive shouldn't despair because camel-cdi supports both Java and XML DSLs.

TIP You can find more documentation about using camel with CDI at http://camel.apache.org/cdi.

What CDI brings to the table is a Java development model using Java code and annotations to configure Java beans and specify their interrelationships.

When using CDI, you need a CDI runtime such as JBoss Weld, which we've been using in our examples. JBoss Weld isn't primarily intended as a standalone server but finds its primary use-case as a component inside an existing application server such as WildFly, WildFly Swarm, or Apache TomEE. Attempting to build your application as a fat JAR deployment and run with JBoss Weld isn't as easy. Instead, you should look at a more powerful application server such as WildFly Swarm or Spring Boot, which are covered next.

7.2.3 WildFly Swarm with Camel as microservice

WildFly Swarm is a Java EE application server in which you package your application and the bits from the WildFly Swarm server you need together in a *fat JAR* binary. You can also view WildFly Swarm as Spring Boot but for Java EE applications.

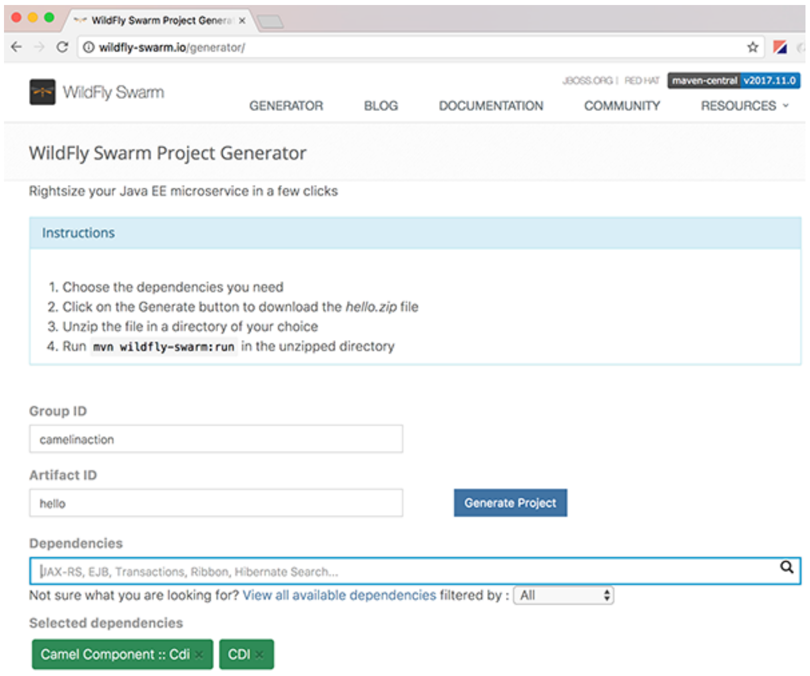
It's easy to get started with WildFly Swarm in a new project, as you set up your Maven pom.xml by doing the following:

- Import WildFly Swarm bill of materials (BOM) dependency

- Declare the WildFly Swarm dependencies you need (such as Camel, CDI, and so forth)
- Add the WildFly Swarm Maven plugin that generates the fat JAR

You can easily get started with a new project by using the generator web page shown in figure 7.1.

In the previous section, you built a Camel microservice using CDI. Now you want to take that application and run it on WildFly Swarm, so you choose Camel CDI and CDI as dependencies. You then need to make one other change, because you're using Jetty as the HTTP server, but WildFly Swarm comes out of the box with its own HTTP server named Undertow. This requires you to change the source code, as shown in Listing 7.5.



WildFly Swarm Project Generator

Rightsize your Java EE microservice in a few clicks

Instructions

1. Choose the dependencies you need
2. Click on the Generate button to download the *hello.zip* file
3. Unzip the file in a directory of your choice
4. Run `mvn wildfly-swarm:run` in the unzipped directory

Group ID: camelinaction

Artifact ID: hello

Generate Project

Dependencies: JAX-RS, EJB, Transactions, Ribbon, Hibernate Search...

Not sure what you are looking for? View all available dependencies filtered by : All

Selected dependencies

Camel Component :: Cdi ×   CDI ×

Figure 7.1 Creating a new WildFly Swarm project from the generator web page (http://wildfly-swarm.io/generator/). Here we've chosen Camel CDI and CDI as the dependencies we need. Clicking the view all dependencies link shows all the dependencies you can choose from. Clicking Generate Project downloads a zip file with the generated source code.

Listing 7.5 Hello service using Undertow as HTTP server

```
@Singleton
public class HelloRoute extends RouteBuilder {

    @Inject
    private HelloBean hello;

    @Inject @Uri("undertow:http://localhost:8080/hello")
```

Use Undertow as HTTP server as it comes out of the box with WildFly Swarm

```
    private Endpoint undertow;

    @Override
    public void configure() throws Exception {
        from(undertow)
            .bean(hello, "sayHello");
    }
}
```

And you also need to add camel-undertow as a dependency to your Maven pom.xml file, as shown here:

```
<dependency>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>camel-undertow</artifactId>
</dependency>
```

Notice that the `groupId` of the dependency isn't `org.apache.camel`, but is `org.wildfly.swarm`. The reason is that WildFly Swarm provides numerous supported and curated dependencies. These dependencies are called *fragments* in WildFly Swarm. In this case, there's a fragment for making Camel and Undertow work together, hence you must use this dependency.

After this change, the service is ready to run on WildFly Swarm, which you can try from the source code by running the following Maven goal from the chapter7/wildfly-swarm directory:

```
mvn wildfly-swarm:run
```

Then from a web browser you can call the service from the following URL: http://localhost:8080/hello.

You can also run the example as a fat JAR by executing the following:

```
java -jar target/hello-swarm.jar
```

This was a brief overview of using Camel with WildFly Swarm. We covered most parts in the previous section about using CDI with Camel.

One important aspect to know when using WildFly Swarm and Camel is the importance of using fragments over Camel components.

WildFly Swarm fragments versus Camel components

When using the Java EE functionality from WildFly Swarm with Camel, you should use the provided WildFly-curated components, called *fragments*. The example uses Undertow, and therefore should use the WildFly Swarm fragment of camel-undertow, and not the regular camel-undertow component:

```
<dependency>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>camel-undertow</artifactId>
</dependency>
```

When there's no specialized fragment for a Camel component, you should use the regular component. For example, there's no fragment for the stream component, and as a user you should use camel-stream:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-stream</artifactId>
</dependency>
```

You can see a list of all known Camel fragments from the WildFly Swarm generator web page by clicking View All Available Dependencies (as was shown in figure 7.1).

CDI isn't the only option when using Camel on WildFly Swarm. You can also use Camel routes defined in Spring XML files.

Using Camel routes with Spring XML

Many Camel users are using the Camel XML DSL to define routes in XML in either Spring or OSGi Blueprint files. Spring is supported by WildFly

Swarm, so you can use Spring XML files to define Camel routes and beans. The route in listing 7.6 could be done in Spring XML:

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

<camelContext xmlns="http://camel.apache.org/schema/spring">
  <propertyPlaceholder id="properties"
                       location="classpath:hello.properties"/>
  <route>
    <from uri="undertow:http://localhost:8080/hello"/>
    <bean ref="helloBean" method="sayHello"/>
  </route>
</camelContext>
```

Spring XML files must be stored in the src/main/resources/spring directory, and the name of the files must use the suffix -camel-context.xml. The book's source code includes this example in the chapter7/wildfly-swarm-spring directory, which you can try by using the following Maven goal:

```
mvn wildfly-swarm:run
```

Monitoring Camel with WildFly Swarm

WildFly Swarm comes with monitoring and health checks out of the box provided by the `monitor` fraction. To enable monitoring, you need to add the `monitor` fraction to the Maven pom.xml file:

```
<dependency>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>monitor</artifactId>
</dependency>
```

You can try this by running the chapter7/wildfly-swarm-spring example:

```
mvn wildfly-swarm:run
```

Then, from a web browser, open http://localhost:8080/node, which shows overall status of the running WildFly Swarm node. Another endpoint is /health, which shows health information. At the time of this writing, there are no default health checks installed and the endpoint returns an HTTP Status 204:

```
$ curl -i http://localhost:8080/health
HTTP/1.1 204 No health endpoints configured!
```

But it's expected that the Camel fraction will include a Camel-specific health check in a future release.

We'll end our coverage of WildFly Swarm by sharing our thoughts of the good and bad when using Camel with WildFly Swarm.

Summary of using WildFly Swarm with Camel for microservices

WildFly Swarm is a lightweight, *just-enough* application server that provides support for all of the Java EE stack. This is appealing for users who are already using Java EE. Users not using Java EE can also use WildFly Swarm and find value in using standards such as CDI, JAX-RS, and others. Camel users preferring the XML DSL can use WildFly Swarm with the Spring XML supported out of the box.

Because WildFly Swarm does independent releases of the WildFly Camel fragments, you may find yourself using a newer version of Apache Camel that hasn't had a WildFly Swarm Camel release yet, meaning you're stuck using the older version.

WildFly Swarm isn't the only just-enough server on the market. Spring Boot is another popular choice that works well with Camel.

7.2.4 Spring Boot with Camel as microservice

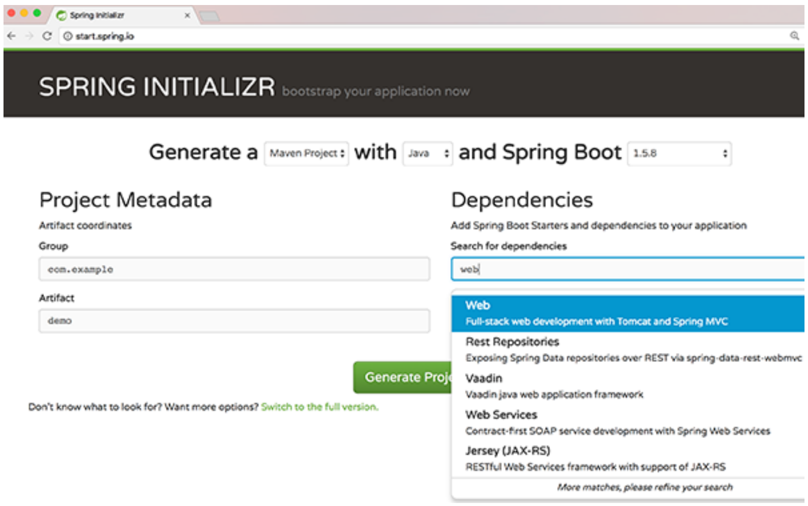
Spring Boot is an opinionated framework for building microservices. Spring Boot is designed with convention over configuration and allows developers to get started quickly developing microservices with reduced boilerplate, configuration, and fuss. Spring Boot does this via the following:

- Autoconfiguration and reduced configuration needed
- Curated list of starter dependencies
- Simplified application packaging as a standalone fat JAR
- Optional application information, insights, and metrics

Let's get started with Spring Boot. First we'll try without Camel, and then we'll add Camel to an existing Spring Boot application.

Getting started with Spring Boot

We'll be using the Spring Boot starter website (http://start.spring.io) to get started. As dependencies, we chose only Web in order to start with a plain web application. You do this by typing `web` in the Search for dependencies field and then selecting Web in the list presented. Figure 7.2 shows where we're going.



Spring Initializr
start.spring.io
SPRING INITIALIZR bootstrap your application now

Generate a Maven Project with Java and Spring Boot 1.5.8

Project Metadata
Artifact coordinates
Group: com.example
Artifact: demo

Dependencies
Add Spring Boot Starters and dependencies to your application
Search for dependencies: web

Web
Full-stack web development with Tomcat and Spring MVC
Rest Repositories
Exposing Spring Data repositories over REST via spring-data-rest-webmvc
Vaadin
Vaadin java web application framework
Web Services
Contract-first SOAP service development with Spring Web Services
Jersey (JAX-RS)
RESTful Web Services framework with support of JAX-RS
More matches, please refine your search

Generate Proj

Don't know what to look for? Want more options? Switch to the full version.

Figure 7.2 Create a new Spring Boot project from the Spring starter website. Start typing `web` in the Search for Dependencies field and then select Web in the list to add it as dependency.

You click the Generate Project button to download the project. You then unzip the downloaded source code and are ready to run the Spring Boot application by using Maven:

```
mvn spring-boot:run
```

TIP You can also create a new Spring Boot project using Spring CLI, or the camel-archetype-spring-boot Maven archetype from Apache Camel. You'll learn how to use the Camel Maven archetypes in chapter 8.

The application then starts up, and you can navigate to http://localhost:8080 in your browser and see a web page. Our application doesn't do anything yet, so let's add a REST endpoint to return a hello message.

Adding REST to Spring Boot application

You want to add a REST endpoint that returns a simple hello message. Spring provides REST support out of the box, which you can build using Java code, as shown in the following listing.

Listing 7.6 `RestController` exposes a REST endpoint using Spring Rest

```
@RestController  ❶
```

❶ Define this class as a REST endpoint

```
@RequestMapping("/spring")  ❷
```

❷ Root mapping for all requests

```
public class HelloRestController {

    @RequestMapping(method = RequestMethod.GET, value = "/hello",
                    produces = "text/plain")  ❸
```

❸ HTTP GET service mapped to /hello

```
    public String hello() {
        return "Hello from Spring Boot";
    }
}
```

The class `HelloRestController` is annotated with `@RestController` ❶, which tells Spring that this class is a REST controller exposing REST endpoints. The annotation `@RequestMapping` is used to map the HTTP URI to Java classes, methods, and parameters. For example, the annotation ❷ at the class level maps all HTTP requests starting with /spring in the context path to this controller. The annotation ❸ at the method exposes a REST service under /spring/hello as a HTTP GET service that produces plain-text content.

The book's source code carries this example in the chapter7/springboot directory; you can try the example by using the following Maven goal:

```
mvn spring-boot:run
```

Then from a web browser, open http://localhost:8080/spring/hello.

In the sections to follow, we'll show you two ways of adding Camel to Spring Boot. We'll add Camel to the existing `HelloRestController` class and then add Camel routes.

Adding Camel to existing Spring Boot REST endpoint

Suppose you have an existing Spring REST controller and want to use one of the many Camel components. You can easily add Camel to any existing Spring controller classes by dependency-injecting a Camel `ProducerTemplate` or `FluentProducerTemplate` and then using the template from the controller methods, as shown in the following listing.

Listing 7.7 Adding Camel to Spring controller class

```
@RestController
@RequestMapping("/spring")
public class HelloRestController {

    @EndpointInject(uri = "geocoder:address:current")    ❶
```

❶ Dependency-injecting Camel FluentProducerTemplate

```
    private FluentProducerTemplate producer;

    @RequestMapping(method = RequestMethod.GET, value = "/hello",
                    produces = "text/plain")
    public String hello() {
    String where = producer.request(String.class);     ❷
```

❷

Using the template to request the endpoint and receive the response as a String

```
        return "Hello from Spring Boot and Camel. We are at: " + where;
    }
}
```

One of the best ways to use Camel from an existing Java class such as a Spring controller is to inject a Camel `ProducerTemplate` or `FluentProducerTemplate` ❶, where you can define the endpoint to call. In this example, we want to know the current location where the application runs by using the camel-geocoder component. This component contacts a service on the internet that, based on your IP, can track your location (to some degree—one of our current locations was 14 kilometers off by the geocoder). The controller then uses the injected Camel template to contact the geocoder when the `hello` method is invoked ❷ so the location can be included in the response.

TIP If you want to use a Camel route from the controller class, you can use the direct component to call the Camel route.

Using Camel starter components with Spring Boot

When using Spring Boot with Camel, you should favor using the Camel starter components that are curated to work with Spring Boot. This example uses the following Camel components in the Maven pom.xml file:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring-boot-starter</artifactId>
</dependency>
<dependency>
  <groupId>org.apache.camel</groupId>
```

```
    <artifactId>camel-geocoder-starter</artifactId>
  </dependency>
```

Notice that the `artifactId` of the Camel components uses the suffix `-starter`. For every Camel component, there's a corresponding Camel Spring Boot–curated starter component that's recommended to use. The starter component has been specially adjusted to work with Spring Boot and includes support for Spring Boot autoconfiguration, which we'll cover later in this section.

The book's source code contains this example in chapter7/springboot-rest-camel, and you can try it by using the following goal:

```
  mvn spring-boot:run
```

Then from a web browser, open http://localhost:8080/spring/hello.

This example doesn't use any Camel routes. Let's take a look at building this example by using a Camel route instead of a Spring controller.

Using Camel routes with Spring Boot

Camel routes are natural to Camel applications, and they're supported in Spring Boot. You can write your Camel routes in Java or XML DSL, but because Spring Boot is Java-centric, you may take on using Java DSL instead of XML. Let's rewrite the previous example that uses a Spring controller to expose a REST service to use a Camel route instead. The following listing shows how this can be done.

Listing 7.8 Use Camel route in Spring Boot to expose a REST service

```
  @Component   1
```

1 @Component annotation to let Camel route be autodiscovered

```
  public class HelloRoute extends RouteBuilder {

      @Override
```

```
    public void configure() throws Exception {
        rest("/").produces("text/plain") ❷
```

❷ Camel Rest DSL to define a REST service in the route

```
            .get("hello")
            .to("direct:hello");

        from("direct:hello") ❸
```

❸ Camel route

```
            .to("geocoder:address:current")
            .transform().simple("Hello. We are at: ${body}");
    }
}
```

When creating Camel routes in Java DSL with Spring Boot, you should annotate the class with `@Component`, which ensures that the route is autodetected by Spring and automatically added to Camel when the application starts up ❶. Because we want to expose a rest service from a Camel route, we need to use an HTTP server component. Spring Boot comes with a servlet engine out of the box, therefore we need to configure a servlet to be used. You can do this easily with Camel by adding the camel-servlet-starter dependency to the Maven pom.xml file. Camel will then automatically detect the `CamelServlet` and integrate that with the servlet engine within Spring Boot. Camel will by default use the context-path `/camel/*`, which can be reconfigured in the application.properties file.

In Camel, you can define REST services using a DSL known as Rest DSL ❷, which will use the servlet component that's just been configured. The Camel route ❸ comes next. Notice that the Rest DSL calls the route by using the direct endpoint, which is how you can link them together.

NOTE Chapter 10 covers the Rest DSL extensively. But we thought the three lines of Rest DSL code shown in listing 7.8 wouldn't knock you down.

We've provided the source code for this example in the chapter7/springboot-camel directory; you can try it by using the following Maven goal:

```
mvn spring-boot:run
```

Then from a web browser, open http://localhost:8080/camel/hello.

Some Camel users prefer using XML DSL over Java DSL.

Using Camel XML DSL with Spring Boot

Spring Boot supports loading Spring XML files, which is how using the XML DSL would work. In your Spring Boot application, you use the `@ImportResource` annotation to specify the location of the XML file from the classpath, as shown here:

```
@SpringBootApplication
@ImportResource("classpath:mycamel.xml")
public class SpringbootApplication {
```

The XML file is a regular Spring `<beans>` XML that can contain `<bean>`s and `<camelContext>`, as shown here:

```
<beans ...>
  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="timer:foo"/>
      <log message="Spring Boot says {{hello}} to me"/>
    </route>
  </camelContext>
</beans>
```

Being able to load Spring XML files in your Spring Boot applications is useful for users who either prefer Camel XML DSL or are migrating existing Spring applications that have been configured with Spring XML files to run on Spring Boot.

We've provided an example with the source code in chapter7/springboot-xml that you can try by using the following Maven goal:

```
mvn spring-boot:run
```

The perceptive eye may notice that this example uses a property placeholder in the Spring XML file in the `<log>` EIP.

Using Camel property placeholders with Spring Boot

Spring Boot supports property configuration out of the box, which allows you to specify your placeholders in the application.properties file. Camel ties directly into Spring Boot, which means you can define placeholders in the application.properties files that Camel can use. The previous example uses the Camel property placeholder in the `<log>` EIP:

```
<log message="Spring Boot says {{hello}} to me"/>
```

We then define a property with the key `hello` in application.properties:

```
hello=I was here
```

And hey, presto, no surprise: Camel is able to look up and use that value without you having to configure anything.

Spring Boot allows you to override property values in various ways. One of them is by configuring JVM system properties. Instead of using `hello=I was here`, you can override this when you start the JVM as follows:

```
mvn spring-boot:run -Dhello='Donald Duck'
```

Another possibility is to use OS environmental variables:

```
export HELLO="Goofy"
mvn spring-boot:run
```

You can also specify the environment variable as a single command line:

```
HELLO=Goofy mvn spring-boot:run
```

Where are we going with this? This allows Spring Boot applications to easily externalize configuration of your applications.

In section 7.1.1, we talked about the characteristics of a microservice, with one of them being that they're highly configurable. Another characteristic of a microservice is that it should be observable by monitoring and management systems.

Monitoring Camel with Spring Boot

Spring Boot provides *actuator endpoints* to monitor and interact with your application. Spring Boot includes built-in endpoints that can report many kinds of information about the application.

To enable Spring Boot actuator, you'd need to add its dependency to the Maven pom.xml file as shown here:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-actuator</artifactId>
</dependency>
```

One of these endpoints is the health endpoint, which shows application health. Spring Boot allows third-party libraries to provide custom health indicators, which Camel has. This means your Camel applications can provide health indicators whether the Camel application is healthy or not.

You can try this by running the chapter7/springboot-rest-camel example:

```
mvn spring-boot:run
```

Then from a web browser, open: http://localhost:8080/health.

By always using the /health endpoint for your Spring Boot applications, you can more easily set up centralized monitoring to use this endpoint to query the health information about your running applications.

Next we have a few words to say about why Camel and Spring Boot work well together for developing microservices applications.

Summary of using Camel with Spring Boot for microservices

Spring Boot is a popular *just-enough* application runtime for running your Java applications. Camel has always had first-class support for Spring, and using Camel with Spring Boot is a great combination. We think it's a great choice. But if you're using or coming from a Java EE background, we recommend looking at WildFly Swarm instead.

No man is an island is a famous phrase from John Donne's 400-year-old poem. Likewise, no microservice is alone. Microservices are composed together to conduct business transactions. The next topic is about building microservices with Camel that call other services and the implications that brings to the table.

7.3 Calling other microservices

In the land of microservices, each service is responsible for providing functionality to other collaborators. Two of the microservices' characteristics we discussed at the beginning of this chapter are that building distributed systems is hard and that microservices must be designed to deal with failures. This section first covers the basics: how to build Camel-based microservices that call other services (without design for failures). The subsequent section covers the EIP patterns that can be applied to design for failures.

The story of the busy developer

At Rider Auto Parts, you've been dazzling a bit with microservices but haven't kicked the tires yet because of a busy work schedule. Does it all sound too familiar? Yeah, even authors of Camel books know this feeling too well. To put yourself back in the saddle again, you send your spouse and kids away to visit your in-laws, leaving you alone in the house for the entire weekend. Your goal is to enjoy an occasional beer while studying and writing some code to familiarize yourself with building a set of microservices that collectively solve a business case. You don't want to settle on a particular runtime and therefore want to implement the services using various technologies.

The business case you'll attempt to implement by using microservices is illustrated in figure 7.3.

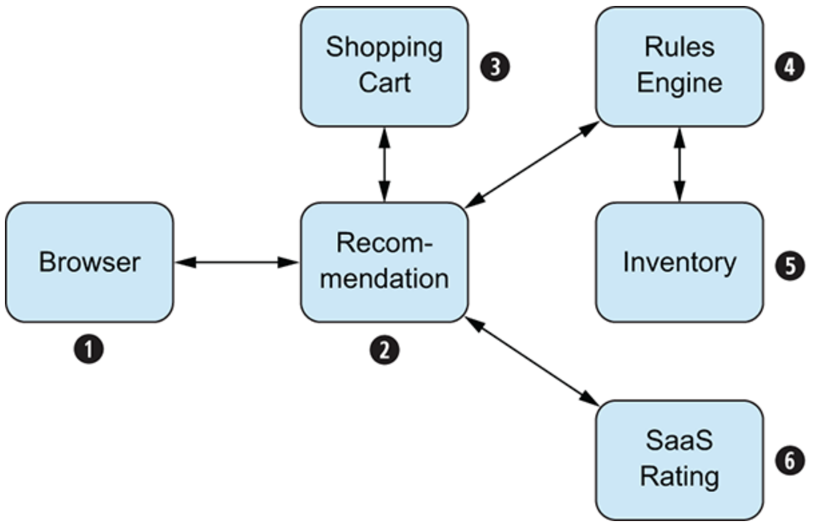


Figure 7.3 Rider Auto Parts recommendation system. From the web browser ❶, the recommendation service ❷obtains items currently in the shopping cart ❸ and then calls the rules engine ❹ to compute items to be recommended, which takes into account the number of items currently in stock ❺. Finally, an external user-based ranking system from a cloud provider ❻ is used to help rank recommendations.

Rider Auto Parts has an upcoming plan to implement a new recommendation system, and you decide to get a jump start by implementing a prototype. Users who are browsing the Rider Auto Parts website will, based on their actions, have recommended items displayed. The web browser ❶ communicates with the recommendation microservice ❷. The recommendation service calls three other microservices ❸ ❹ ❻ as part of computing the result. Items already in the shopping cart ❸ are fed into the rules engine ❹, which is used to rank recommendations. The inventory system ❺ in the back end hosts information about the items currently in stock and can help rank items that can be shipped to customers immediately. An external SaaS service is used to help rank popular items based on a worldwide user-based rating system ❻.

Technologies

The technology stack for the recommendation system is illustrated in figure 7.4.

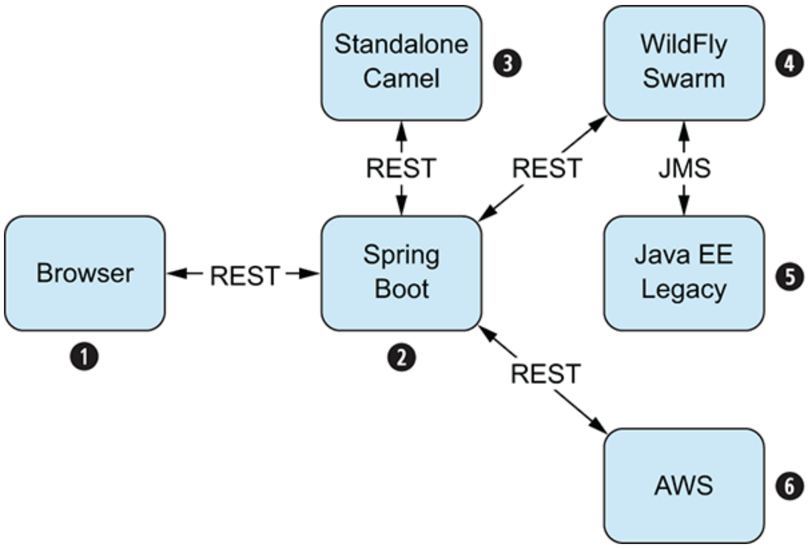


Figure 7.4 Web browser ❶ communicates using HTTP/REST to the recommendation system ❷ running on Spring Boot. The shopping cart is running standalone Camel ❸. The rules engine ❹ running inside WildFly Swarm is integrated using REST. The inventory system is hosted on a legacy Java EE application server ❺, which is integrated using JMS messaging. The rating system is hosted in the cloud on AWS ❻.

You have one weekend, so let's start hacking code. You've decided to implement each microservice as a separate module in the source code. To keep things simple, you decide to keep all the source in the same Git repository. The source code structure is like this:

```
prototype
├── cart            Shopping cart
├── inventory       Inventory system
├── recommend       Recommendation microservice
├── rules           Rules engine
└── rating          External SaaS rating system
```

You can find the source code for this example in the chapter7/prototype directory. The following section covers how to build this prototype and highlights the key parts.

7.3.1 Recommendation prototype

The prototype for the recommendation microservice is built as a vanilla Spring Boot application. This microservice is the main service, which depends on all the other microservices, and hence has more moving parts than the services to follow.

The prototype consists of the following source files:

- *CartDto.java*—POJO representing a shopping cart item
- *ItemDto.java*—POJO representing an item to be recommended

- *RatingDto.java*—POJO representing a rating of an item
- *RecommendController.java*—The REST service implementation
- *SpringBootApplication.java*—The main class to bootstrap this service
- *application.properties*—Configuration file

The *meat* of this service is the `RecommendController` class, shown in the following listing.

Listing 7.9 Recommendation REST service

```
@RestController    1
```

1

Defines class as REST controller

```
@RequestMapping("/api")
@ConfigurationProperties(prefix = "recommend")    2
```

2

Injects configuration properties with prefix “recommend”

```
public class RecommendController {

  private String cartUrl;    3
```

3

Fields with injected configuration values

```
  private String rulesUrl;    3
  private String ratingsUrl;    3

  private final RestTemplate restTemplate = new RestTemplate();    4
```

4

REST template used to call other microservices

```
  @RequestMapping(value = "recommend", method = RequestMethod.GET,
          produces = "application/json")    ❺
```

❺ Defines REST service

```
  public List<ItemDto> recommend(HttpSession session) {
    String id = session.getId();

    CartDto[] carts = restTemplate.getForObject(
            cartUrl, CartDto[].class, id);    ❻
```

❻ Calls shopping cart REST service

```
    String cartIds = cartsToCommaString(carts);

    ItemDto[] items = restTemplate.getForObject(
            rulesUrl, ItemDto[].class, id, cartIds);    ❼
```

❼ Calls rules engine REST service

```
    String itemIds = itemsToCommaString(items);

    RatingDto[] ratings = restTemplate.getForObject(
              ratingsUrl, RatingDto[].class, itemIds);    ❽
```

❽ Calls SaaS rating REST service

```
    for (RatingDto rating : ratings) {
          appendRatingToItem(rating, items);
        }
        return Arrays.asList(items);
      }
```

Spring Boot makes it easy to define the REST service in Java by annotating the class with `@RestController` and `@RequestMapping` ❶. The chapter introduction indicated the good practice of externalizing your configuration. Spring Boot allows you to automatically inject getter/setters ❸ from its configuration file by annotating the class with `@ConfigurationProperties` ❷. In this example, we've configured `recommend` as the prefix that maps to the following properties from the application.properties file:

```
recommend.cartUrl=http://localhost:8282/api/cart?sessionId={id}
recommend.rulesUrl=http://localhost:8181/api/rules/{cartIds}
recommend.ratingsUrl=http://localhost:8383/api/ratings/{itemIds}
```

To call other REST services, we use Spring's `RestTemplate` ❹. The method `recommend` is exposed as a REST service by the `@RequestMapping` annotation ❺. The service then calls three other microservices ❻ ❼ ❽ using `RestTemplate`.

AUTOMATIC MAPPING JSON RESPONSE TO DTO

Notice that the `RestTemplate getForObject` method automatically maps the returned JSON response to an array of the desired DTO classes, such as when calling the shopping cart service:

```
CartDto[] carts = restTemplate.getForObject(cartUrl,
                                CartDto[].class, id);
```

You must use an array type and can't use a `List` type because of Java type erasure in collections. For example, the following can't compile:

```
List<CartDto> carts = restTemplate.getForObject(cartUrl,
                                List<CartDto.class>, id);
```

As you can see, calling another microservice using Spring's `RestTemplate` is easy, because it requires just one line of code with `RestTemplate`. Have you seen this kind before? Yes: Camel's `ProducerTemplate` or `FluentProducerTemplate` is also easy to use for sending messages to Camel endpoints with one line of code.

In listing 7.10, the first microservices called is the shopping cart service.

7.3.2 Shopping cart prototype

A goal of building the prototype is also to gain practical experience by using Camel with different Java toolkits and frameworks. This time, you've chosen to use standalone Java with CDI and Camel for the shopping cart service. The implementation of the shopping cart is kept simple as a pure in-memory storage of items currently in the carts, as shown in the following listing.

Listing 7.10 Simple shopping cart implementation class

```
@ApplicationScoped    (1)
```

(1)

CDI application-scoped bean named cart

```
@Named("cart")
public class CartService {

  private final Map<String, Set<CartDto>> content = new LinkedHashMap<>(

  public void addItem(@Header("sessionId") String sessionId,
                      @Body CartDto dto) {    (2)
```

(2)

Method to add item to cart stored

```
      Set<CartDto> dtos = content.get(sessionId);
      if (dtos == null) {
        dtos = new LinkedHashSet<>();
        content.put(sessionId, dtos);
      }
      dtos.add(dto);
    }

  public void removeItem(@Header("sessionId") String sessionId,
                         @Header("itemId") String itemId) {    ❸
```

❸ Method to remove an item from the cart

```
    Set<CartDto> dtos = content.get(sessionId);
    if (dtos != null) {
      dtos.remove(itemId);
    }
  }

  public Set<CartDto> getItems(@Header("sessionId") String sessionId) {    ❹
```

❹ Method to get the items from the cart

```
    Set<CartDto> answer = content.get(sessionId);
    if (answer == null) {
      answer = Collections.EMPTY_SET;
    }
    return answer;
  }
}
```

The `CartService` class has been annotated with `@ApplicationScoped` ❶ because only one instance is needed at runtime. The name of the instance can be assigned by using `@Named`. The class implements three methods ❷ ❸ ❹ to add, remove, and get items from the cart. Notice that these methods have been annotated with Camel's `@Header` to bind the parameters to message headers. We do this to make it easy to call the `CartService` bean from a Camel route, which we'll cover in a little while.

TIP We covered bean parameter bindings in chapter 4.

In listing 7.10, the shopping cart uses `CartDto` in the `add` and `get` methods ❷ ❹. This class is implemented as a plain POJO, as shown here:

```
public class CartDto {
  private String itemId;
  private int quantity;

 @ApiModelProperty(value = "Id of the item in the shopping cart")    ❶
```

❶

Swagger annotation to document the fields

```
  public String getItemId() {
    return itemId;
  }

 @ApiModelProperty(value = "How many items to purchase")    ❶
  public int getQuantity() {
    return quantity;
  }

  // setter methods omitted
}
```

The `CartDto` class has two fields to store the item ID and the number of items to purchase. The getter methods ❶ have been annotated with Swagger's `ApiModelProperty` to include descriptions of the fields, which will be used in the API documentation of the microservice.

TIP Chapter 10 covers REST services and Swagger API documentation.

Earlier in this chapter, in code listing 7.5, you got a quick glimpse of Camel's Rest DSL. Now brace yourself, because this time we'll dive deeper as you implement the shopping cart microservice using Camel's Rest DSL and with automatic API documentation using Swagger. All this is done from the Camel route shown in the following listing.

Listing 7.11 Shopping cart microservice using Camel's Rest DSL

```
@Singleton
public class CartRoute extends RouteBuilder {

  public void configure() throws Exception {

    restConfiguration("jetty").port("{{port}}")    1
```

1

Configures Rest DSL to use Jetty component

```
      .contextPath("api")    1
      .bindingMode(RestBindingMode.json)    2
```

2

Turns on JSON binding to/from POJO classes

```
      .dataFormatProperty("disableFeatures", "FAIL_ON_EMPTY_BEANS")    3
```

3

Turns off binding error on empty lists/beans

```
      .apiContextPath("api-doc")    4
```

4

Enables Swagger API documentation

```
        .enableCORS(true);
```

```
    rest("/cart").consumes("application/json").produces("application/jso
    .get()  5
```

5

GET/cart service

```
        .outType(CartDto[].class)
        .description("Returns the items currently in the shopping cart")
    .to("bean:cart?method=getItems")  8
```

8

Calls the CartService bean methods

```
    .post()  6
```

6

POST /cart service

```
        .type(CartDto.class)
        .description("Adds the item to the shopping cart")
    .to("bean:cart?method=addItem")  8
    .delete().description("Removes the item from the shopping cart")  7
```

7

DELETE /cart service

```
        .param().name("itemId")
          .description("Id of item to remove")
        .endParam()
    .to("bean:cart?method=removeItem");  8
  }
}
```

The REST service is using Camel's jetty component ❶ as the HTTP server. To work with JSON from Java POJO classes, we turned on JSON binding ❷, which will use Jackson under the covers. Jackson is instructed to not fail if binding from an empty list/bean ❸. This is needed in case the shopping cart is empty and the `GET` service ❻ is called. Swagger is turned on for API documentation ❹. The Rest DSL then exposes three REST services ❺ ❻ ❼, each calling the `CartService` bean ❽. Notice that each REST service documents its input and output types and parameters, which becomes part of the Swagger API documentation.

Running the shopping cart service

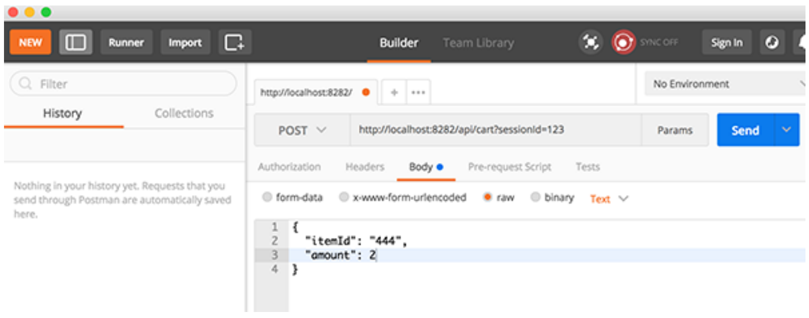
You can start the source code for the shopping cart microservice, located in the chapter7/prototype/cart directory, using the following Maven goal:

```
mvn compile exec:java
```

You can then access the Swagger API documentation from a web browser:

```
http://localhost:8282/api/api-doc
```

To test the shopping cart, you can install Postman, which has a built-in REST client, as an extension to your web browser, as shown in figure 7.5.



```
POST http://localhost:8282/api/cart?sessionId=123
```

```
{
  "itemId": "444",
  "amount": 2
}
```

Figure 7.5 Using Postman to send an HTTP POST request to a shopping cart service to add the item to the cart

Using Postman, you can then more easily call REST services with more-complex queries that involve `POST`, `PUT`, and `DELETE` operations. Notice that we hardcode the HTTP session ID to 123 as a query parameter.

Because the shopping cart microservice includes Swagger API documentation, we can use Swagger UI to test the service. You can easily run Swagger UI using Docker by running the following command line:

```
docker run -d --name swagger-ui -p 8080:8080 swaggerapi/swagger-ui
```

Open a web browser:

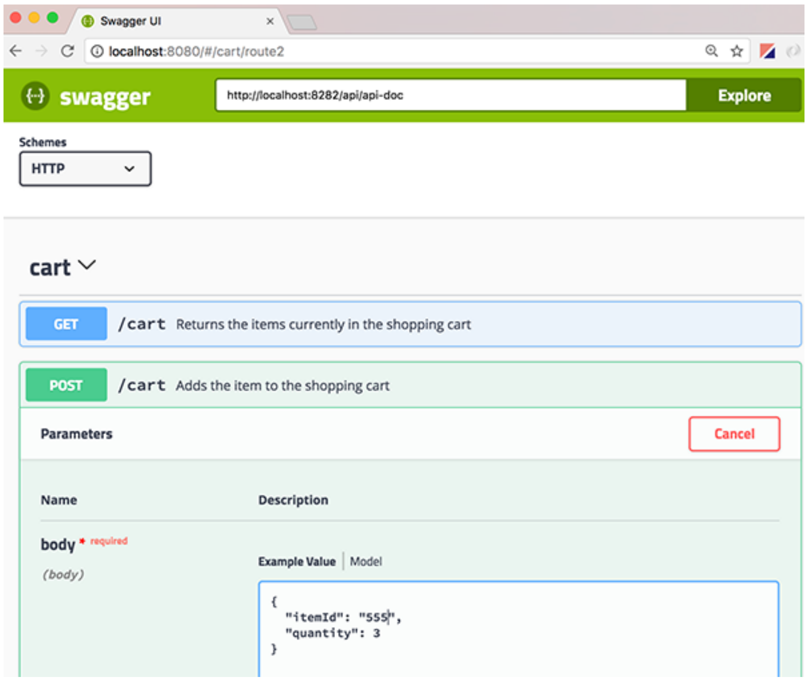
```
http://localhost:8080/
```

Then type the following URL in the URL field and click the Explore button:

```
http://localhost:8282/api/api-doc
```

You should see the cart service, which you can expand and try, as shown in figure 7.6.

This was a glimpse of some of the powers Camel provides for REST services. We've devoted all of chapter 10 to covering this in much more depth. Let's move on to the next microservice you're developing during the weekend: the rules and inventory services.



localhost:8080/#/cart/route2

http://localhost:8282/api/api-doc

Schemes: HTTP

cart

GET /cart Returns the items currently in the shopping cart

POST /cart Adds the item to the shopping cart

Parameters

Name	Description
body * required (body)	Example Value | Model

```
{
  "itemId": "555",
  "quantity": 3
}
```

Figure 7.6 Using the Swagger UI to explore the shopping cart REST service. Here we're about to call the `POST` method to add an item to the shopping cart.

7.3.3 Rules and inventory prototypes

Not all microservices use REST services. For example, the Rider Auto Parts inventory system is accessible only by using a JMS messaging broker and XML as the data format. This means the rules microservice needs to use JMS and XML, and therefore you decide to use a Java EE micro container (WildFly Swarm) to host the rules microservice. This isn't bad news, because a goal of the prototype is to use a variety of technologies to learn what works and what doesn't work.

The rules microservice depends on the inventory back end, so let's start there first.

THE INVENTORY BACK END

Because you develop the prototype from home, you don't want to access the Rider Auto Parts test environment and therefore build a quick simulated inventory back end using standalone Camel with an embedded ActiveMQ broker, which starts up using a main class. The embedded ActiveMQ broker listens for connections on its default port:

```
<transportConnector name="tcp" uri="tcp://0.0.0.0:61616"/>
```

The rules microservice can communicate using JMS to the embedded broker via localhost:61616.

The back end is implemented using the following mini Camel route:

```
  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <route id="inventory">
      <from uri="activemq:queue:inventory"/>    1
```

1

JMS request/reply to query inventory

```
      <to uri="bean:inventory"/>    2
```

2

Bean that simulates inventory back end

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

The rules microservice sends a JMS message to the inventory queue ❶, which then gets routed to the bean ❷ that returns the items in the inventory in XML format, which is returned back to the rules microservice over JMS.

The rules microservice

The rules microservice uses WildFly Swarm as the runtime with embedded Camel. The service exposes a REST service that the recommendation service will use. Because WildFly Swarm is a Java EE server, you can implement the REST service using JAX-RS technology.

This is done by creating a JAX-RS application class:

```
@ApplicationPath("/")   1
```

1
JAX-RS application using root (/) as context-path

```
public class RulesApplication extends Application {
}
```

This implementation is short—we don't need any custom configuration. The `@ApplicationPath` annotation ❶ is used to define the starting context-path that the REST services will use. The REST service is implemented in the `RulesController` class, as shown in the following listing.

Listing 7.12 JAX-RS REST implementation

```
@ApplicationScoped   1
```

1
CDI application-scoped bean

```
@Path("/api")   2
```

2

JAX-RS REST service using /api as context-path

```
public class RulesController {

  @Inject
  @Uri("direct:inventory")   3
```

3

Injects Camel's FluentProducerTemplate

```
  private FluentProducerTemplate producer;

  @GET
  @Produces(MediaType.APPLICATION_JSON)
  @Path("/rules/{cartIds}")   4
```

4

HTTP GET service

```
  public List<ItemDto> rules(@PathParam("cartIds") String cartIds) {
    List<ItemDto> answer = new ArrayList<>();

    ItemsDto inventory = producer.request(ItemsDto.class);   5
```

5

Calls Camel route to get items from inventory service

```

    for (ItemDto item : inventory.getItems()) {   6
```

6

Filters duplicate items

```
        boolean duplicate = cartIds != null
                                      && cartIds.contains("" + item.getItemNo());
        if (!duplicate) {
          answer.add(item);
        }
      }

    Collections.sort(answer, new ItemSorter());    ❼
```

❼ Sorts and ranks items to be returned

```
      return answer;
    }
  }
```

The controller class can be automatically discovered via CDI by annotating the class with `@ApplicationScoped` ❶. If this isn't done, the class would need to be registered manually in the JAX-RS application class. The `@Path` annotation ❷ is used to denote the class as hosting REST services that are serviced from the given /api context path. To call the inventory back end, you'll use Camel and therefore inject `FluentProducerTemplate` ❸ to make it easy to call a Camel route from within the class.

The rules method ❹ is annotated with `GET` to set this up as an HTTP `GET` service returning data in JSON format. Notice that `@Path` and `@PathParam` map to the `partIds` method parameter from the HTTP URL using the `{cartIds}` syntax. For example, if the REST service is called using `HTTP GET /rules/123,456` and then `partIds` is mapped to the String value `"123,456"`.

The injected `FluentProducerTemplate` ❸ is used to let Camel call the back end service in one line of code ❺. Because the service is returning a response, you use the `request` method (`request` = `InOut`, `send` = `InOnly`) and automatically convert the response to the `ItemsDto` POJO type.

The rules microservice then filters out duplicate items that aren't intended to be returned ❻. Finally, the items are sorted by using the `ItemSorter` implementation ❼ that ranks the items to custom rules.

Camel is used to call the inventory service via JMS messaging, as shown in the Camel route:

```
from("direct:inventory")
  .to("jms:queue:inventory")    1
```

1

Calls the inventory back end using JMS

```
  .unmarshal().jaxb("camelinaction");    2
```

2

Converts response from XML to POJO classes using JAXB

The route uses the Camel JMS component ❶, which needs to be configured. This is easily done in Java code using CDI `@Produces`:

```
@Produces
@Named("jms")
public static JmsComponent jmsComponent() {
  ActiveMQComponent jms = new ActiveMQComponent();
  jms.setBrokerURL("tcp://localhost:61616");
  return jms;
}
```

The last thing you need to implement is the mapping between XML and JSON, and you can do that using POJO classes and JAXB. The Camel route converts the returned XML data from the back end to POJO by using JAXB unmarshal ❷.

The REST service in listing 7.12 is declared to produce JSON, and the return type of the method is `List<ItemDto>`. That's all you need to do because WildFly Swarm will automatically convert the POJO classes to JSON using JAXB.

TIP When using JAXB, always remember to list the POJO classes in the jaxb.index file, which resides in the src/main/resources folder.

Running the rules and inventory services

You can run these services from the chapter7/prototype directory. First you need to start the inventory service:

```
cd inventory
mvn compile exec:java
```

Then from another shell, you can start the rules microservice:

```
cd rules
mvn wildfly-swarm:run
```

From a web browser, you can call the rules service using the following URL:

```
http://localhost:8181/api/rules/123,456
```

The inventory service has been hardcoded to return three items using IDs 123, 456, and 789. Therefore, you can try URLs such as these:

```
http://localhost:8181/api/rules/123
http://localhost:8181/api/rules/456,789
```

The response returned is intended to filter out items that are already present in the shopping cart, so you may have responses that don't return one or more of the item IDs provided as inputs. Only one microservice is left in the prototype to develop: the rating service.

7.3.4 Rating prototype

It's been a long, productive weekend, and you've already built four prototypes. Only the rating service is left. Building the prototype from home, you don't have access to the provider of the rating SaaS service. Instead, you decide to build a quick-and-dirty simulation of the rating service that you can run locally. When it comes to the real deal, the rating service is hosted on Amazon AWS, and Camel has the camel-aws component that offers integration with many of the Amazon technologies.

You've used Rest DSL a few times already in this chapter, so you can quickly slash out the following code to set up a Camel REST service:

```
restConfiguration().bindingMode(RestBindingMode.json);   1
```

1

Turns on JSON binding

```
rest("/ratings/{ids}").produces("application/json")   2
```

2

GETs /ratings/{ids} REST service

```
    .get().to("bean:ratingService");
```

The REST service uses JSON, so you turn on automatic JSON binding ❶, which allows Camel to bind between JSON data format and POJO classes. Only one REST service rates items ❷. In the interest of time, you implement the rating service to return random values:

```
@Component("ratingService")   1
```

1

Naming the bean ratingService

```
public class RatingService {
  public List<RatingDto> ratings(@Header("ids") String items) {    ❷
```

❷

Parameter mapping of rating IDs

```
      List<RatingDto> answer = new ArrayList<>();
      for (String id : items.split(",")) {
        RatingDto dto = new RatingDto();
        answer.add(dto);
        dto.setItemNo(Integer.valueOf(id));
        dto.setRating(new Random().nextInt(100));    ❸
```

Generating random rating value

```
      }
      return answer;
    }
  }
```

The class is annotated with `@Component("ratingService")` ❶, which allows Spring to discover the bean at runtime and allows Camel to call the bean by its name, which is done from the Camel route using `to("bean:ratingService")`. The rating method ❷ maps to the Rest DSL `/ratings/{ids}` path using the `@Header("ids")` annotation, which ensures that Camel will provide the values of the IDs upon calling the bean.

The returned value from the bean is `List<RatingDto>`, which is a POJO class. The POJO class is merely a class with two fields:

```
public class RatingDto {
    private int itemNo;
    private int rating;
    // getter and setter omitted
}
```

Because you've turned on automatic JSON binding in the Rest DSL configuration, Camel will automatically convert from `List<RatingDto>` to a

JSON representation.

Running the rating service

You can find the rating prototype in the chapter7/prototype/rating directory, which you can run using the following Maven goal:

```
mvn spring-boot:run
```

Then from a web browser you can open the following URL:

```
curl http://localhost:8383/api/ratings/123,456
```

Phew! You've now implemented prototypes for all the microservices for the Rider Auto Parts recommendation system. So far, you've been running each service in isolation. It's about time that you see how they work together.

7.3.5 Putting all the microservices together

During a weekend, you've been able to build a prototype for the Rider Auto Parts recommendation system that consists of five independent microservices, as previously illustrated in figure 7.3. Now it's time to put all the pieces together, plug in the power, and see what happens.

The book's source code contains the entire prototype in the chapter7/prototype directory. You need to open five command shells at once, and for each shell run the following commands in the given order:

1. Start the shopping cart service:

   ```
   cd cart
   mvn compile exec:java
   ```

2. Start the inventory service:

   ```
   cd inventory
   mvn compile exec:java
   ```

3. Start the rules service:

```
cd rating
mvn spring-boot:run
```

4. Start the recommendation service:

```
cd rules
mvn wildfly-swarm:run
```

5. Start the rating service:

```
cd recommend
mvn spring-boot:run
```

Then you can open a web browser and call the recommendation service using the following URL:

```
http://localhost:8080/api/recommend
```

The response should contain three items being recommended. That may seem like no big deal, but you have five Java applications running isolated in their own JVM, integrated together to implement a business solution.

The weekend is coming to a close, and you want to summarize what you've learned so far.

What you've learned

First, you've learned that it's good from time to time to clear your schedule and set aside one or two days to let you fully concentrate when learning new skill sets. The busy workday and the rapid evolution of software are having tremendous impacts on business. Many companies are getting more and more worried about becoming "uberized" (beaten by new, mobile-based businesses). As a forward thinker, you take the matter into your own hands to ensure that you up your game and don't become obsolete in the job market.

With that in mind, you also gained a fair amount of hands-on experience building small microservices with various technologies. For example, you have no worries about Camel, as it fits in any kind of Java runtime of your choosing. You learned that Spring Boot is a great opinionated framework for building microservices. The curated starting dependencies are a snap to install, and Spring Boot's autoconfiguration makes configuration consistent and easy to learn and use. Camel works well with Spring Boot, and you have great power with testing as well (more on that to come in chapter 9). You dipped your toes into WildFly Swarm and learned a few things. It's a small and fast application server on which you can easily include only the Java EE parts you need. But WildFly Swarm doesn't yet have as good a story as Spring Boot when it comes to configuration management, and you struggled a bit with configuring WildFly Swarm and your Camel applications in a seamless fashion. You learned that using Camel with WildFly requires knowledge of CDI, which is a similar programming model to Spring's annotations, so the crossover is fairly easy to learn. You want to do a few more things before the family returns and your weekend comes to a close. At the beginning of this chapter, you learned about some of the characteristics of microservices, which you want to compare with what you've done this weekend. Your comments are marked in table 7.1.

Table 7.1 Comments about microservices characteristics

Characteristic	Comment
Small in size	Each microservice is surely small in size and does one thing and one thing only.
Observable	Microservices should be observable from centralized monitoring and management. This is covered in chapter 16.
Design for failure	An important topic that we haven't covered sufficiently in this chapter. But there's more to come in chapter 11, covering error handling.
Highly configurable	Spring Boot is highly configurable. WildFly Swarm doesn't yet offer configuration on the same level as Spring Boot. But you can use the Camel properties component for Camel-only configuration.
Smart endpoints and dumb routes	Can easily be done using Camel in smaller Camel routes.
Testable	Camel has great support for testing, which is covered in chapter 9.

You've come to realize that you haven't focused so much on the aspect of designing for failure. You still have the five microservices running, so you quickly play the devil's advocate and stop the inventory service, and then refresh the web browser. Kaboom—the recommendation service fails with an HTTP error 500, and you find errors logged in the consoles. Oh boy—how could you forget about this?

When building microservices or distributed applications, it's paramount to assume that failures happen, and you must design with that in mind. Designing for and dealing with failures is covered in chapters 11, 17, and 18. But we won't leave you in the dark—let's take a moment to lay out the landscape and show you how to use a design pattern to handle failures, and then we'll return to this prototype before we reach the end of this chapter.

7.4 Designing for failures

Murphy's law—*Whatever can go wrong will go wrong*—is true in a distributed system such a microservice architecture. You could spend a lot of time preventing errors from happening, but even so, you can't predict every case of how microservices could fail in a distributed system. Therefore, you must face the music and design your microservices as resilient and fault-tolerant. This section touches on the following patterns you can use to deal with failures:

- Retry
- Circuit Breaker
- Bulkhead

The first pattern, Retry, is the traditional solution to dealing with failures: if something fails, try again. The second and third are patterns that have become popular with microservice architectures; they're often combined, and you get this combination with the Netflix Hystrix stack.

But let's start from the top with the traditional way.

7.4.1 Using the Retry pattern to handle failures

The Retry pattern is intended for handling transient failures, such as temporary network outages, by retrying the operation with the expectation that it'll then succeed. The Camel error handler uses this pattern as its primary functionality. This section only briefly touches on the Retry pattern and Camel's error handler because chapter 11 is entirely devoted to this topic. But you can use the Retry pattern in the rules prototype containing the following Camel route that calls the inventory service:

```
public class InventoryRoute extends RouteBuilder {

  public void configure() throws Exception {
    from("direct:inventory")
      .to("jms:queue:inventory") 1
```

❶
Calling inventory microservice, which may fail

```
        .unmarshal().jaxb("camelinaction");
    }
  }
```

To handle failures when calling the inventory microservice ❶, we can configure Camel's error handler to retry the operation:

```
    public void configure() throws Exception {
      errorHandler(defaultErrorHandler()
               .maximumRedeliveries(5)    ❶
```

❶
Configures error handler to retry up to 5 times

```
                    .redeliveryDelay(2000));

      from("direct:inventory")
        .to("jms:queue:inventory")
        .unmarshal().jaxb("camelinaction");
    }
```

As you can see, you've configured Camel's error handler to retry ❶ up to five times, with a two-second delay between each attempt. For example, if the first two attempts fail, the operation succeeds at the third attempt and can continue to route the message. Only if all attempts fail does the entire operation fail, and an exception is propagated back to the caller from the Camel route.

If only the world were so easy. But you must consider these five factors when using the Retry pattern.

Which failures

Not all failures are candidates for retrying. For example, network operations such as HTTP calls or database operations are good candidates. But in every case, it depends. An HTTP server may be unresponsive, and retrying the operation in a bit may succeed. Likewise, a database operation may fail because of a locking error due to concurrent access to a table, which may succeed on the next retry. But if the database fails because of wrong credentials, retrying the operation will keep failing, and therefore it isn't a candidate for retries.

Camel's error handler supports configuring different strategies for different exceptions. Therefore, you can retry only network issues, and not exceptions caused by invalid credentials. Chapter 11 covers Camel's error handler.

How often to retry

You also have to consider how frequently you may retry an operation, as well as the total duration. For example, a real-time service may be allowed to retry an operation only a few times, with short delays, before the service must respond to its client. In contrast, a batch service may be allowed many retries with longer delays.

The retry strategy should also consider other factors, such as the SLAs of the service provider. For example, if you call the service too aggressively, the provider may throttle and degrade your requests, or even blacklist the service consumer for a period of time. In a distributed system, a service provider must build in such mechanisms—otherwise, consumers of the services can overload their systems or degrade downstream services. Therefore, the service provider often provides information about the remaining request count allowed. The retry strategy can read the returned request count and attempt only within the permitted parameters.

TIP Camel's error handler offers a `retryWhile` functionality that allows you to configure to retry only while a given predicate returns a `true` value.

Idempotency

Another factor to consider is whether calling a service is idempotent. A service is *idempotent* if calling the service again with the same input parameters yields the same result. A classic example is a banking service:

calling the bank balance service is idempotent, whereas calling the withdrawal service isn't idempotent.

Distributed systems are inherently more complex. Because of remote networks, a request may have been processed by the remote service but not received by the client, which then assumes the operation failed, and retries the same operation, which then may cause unexpected side effects.

TIP In a distributed system, it's difficult to ensure "once and only once" guarantees. Design with idempotency and duplicates in mind.

The service provider can use the Idempotent Consumer EIP pattern to guard its service with idempotency. We cover this pattern in chapter 12, section 12.5.

Monitoring

Monitoring and tracking retries are important. For example, if some operations have too many retries before they either fail or succeed, it indicates a problem area that needs to be fixed. Without proper monitoring in place, retries may remain unnoticed and affect the overall stability of the system.

Camel comes with extensive metrics out of the box that track every single message being routed in fine-grained detail. You'll be able to pinpoint exactly which EIP or custom processor in your Camel applications is causing retries or taking too long to process messages. You'll learn all about monitoring your Camel applications in chapter 16.

TIP Camel uses the term *redelivery* when a message is retried.

Time-out and SLAs

When using the Retry pattern, and retries kick in, the overall processing time increases by the additional time of every retry attempt. In a microservice architecture, you may have SLAs and consumer time-outs that should be factored in. Take the maximum time allowed to handle the request as specified in SLAs and consumer times and then calculate appropriate retry and delay settings.

The example at the start of section 7.4.1 uses Camel's error handler for retries. What if you don't use Camel? How can you do retries?

7.4.2 Using the Retry pattern without Camel

If you don't use Camel, you can still use the Retry pattern by using good old-fashioned Java code with a `try ... catch` loop. For example, in the recommendation service that's using Spring Boot and Java, you could implement a Retry pattern, as shown in the following listing.

Listing 7.13 Retry pattern using Java `try ... catch` loop

```
String itemIds = null;
ItemDto[] items = null;
int retry = 5 + 1;
for (int i = 0; i < retry; i++) {   1
```

1

Retry loop

```
    try {
        LOG.info("Calling rules service {}", rulesUrl);
        items = restTemplate.getForObject(rulesUrl,
                                          ItemDto[].class, id, cartIds);
        itemIds = itemsToCommaString(items);
        LOG.info("Inventory items {}", itemIds);
        break;   2
```

2

Break out loop if succeeded

```
    } catch (Exception e) {
        if (i == retry - 1) {
            throw e;   3
```

❸
Rethrow exception if end of loop

```
            }
        }
    }
```

In the Java code, you can implement a Retry pattern by using a `for` loop ❶. You keep looping until either the operation succeeds ❷ or keeps on failing until the end of the loop ❸. As you can see, implementing this code is cumbersome, error prone, and *ugly*. Isn't there a better way? Well, if you're using Apache Camel, we've already shown you the elegancy of Camel's error handler; but some of our microservices aren't using Camel, so what you can do? You can use a pattern called Circuit Breaker, which has become popular with microservice architectures.

7.4.3 Using Circuit Breaker to handle failures

The previous section covered the Retry pattern and the ways it can overcome transient errors, such as slow networks, or a temporary overloaded service, by retrying the same operation. But sometimes failures aren't transient—for example, a downstream service may not be available, may be overloaded, or may have a bug in the code, causing application-level exceptions. In those situations, retrying the same operation and putting additional load on an already struggling system provide no benefit. If you don't deal with these situations, you risk degrading your own service, holding up threads, locks, and resources, and contributing to cascading failures that can degrade the overall system, and even take down a distributed system.

Instead, the application should detect that the downstream service is unhealthy and deal with that in a graceful matter until the service is back and healthy again. What you need is a way to detect failures and fail fast in order to avoid calling the remote service until it's recovered.

The solution for this problem is the Circuit Breaker pattern, described by Michael Nygard in *Release It!* (Pragmatic Bookshelf, 2007).

Circuit Breaker pattern

The Circuit Breaker pattern is inspired by the real-world electrical circuit breaker, which is used to detect excessive current draw and fail fast to protect electrical equipment. The software-based circuit breaker works on the same notion, by encapsulating the operation and monitoring it for failures. The Circuit Breaker pattern operates in three states, as illustrated in figure 7.7.

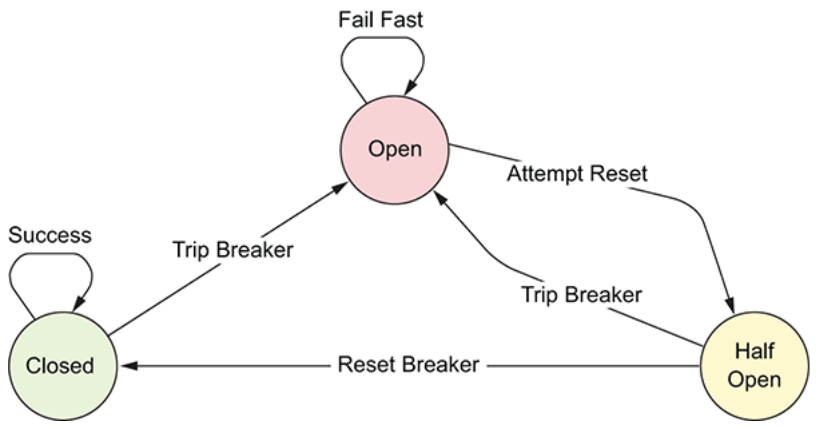


Figure 7.7 The Circuit Breaker pattern operates in three states. In the closed state (green), the operation is successful. Upon a number of successive failures, the breaker trips into the open state (red) and fails fast. After a short period, the breaker attempts to reset in the half-open state (yellow). If it's successful, the break resets into a closed state (green), and in case of failure transfers back to an open state (red).

The states are as follows:

- *Closed*—When operating successfully.
- *Open*—When failure is detected and the breaker opens to short-circuit and fail fast. In this state, the circuit breaker avoids invoking the protected operation and avoids putting additional load on the struggling service.
- *Half Open*—After a short period in the open state, an operation is attempted to see whether it can complete successfully, and depending on the outcome, it will transfer to either open or closed state.

Speaking from practical experience with using the Circuit Breaker pattern, you may mix up the open and closed states. It takes a while to make your brain accept the fact that closed is good (green) and open is bad (red).

At first sight, the Retry and Circuit Breaker patterns may seem similar. Both make an operation resilient to failures from downstream services. But the difference is that the Retry pattern invokes the same operation in hopes it will succeed, whereas the Circuit Breaker prevents overloading struggling downstream systems. The former is good for dealing with transient failures, and the latter for more permanent and long-lasting failures.

In microservice architecture and distributed systems, the Circuit Breaker pattern has become a popular choice. In more traditional systems, the Retry pattern has been the primary choice. But one doesn't exclude the other; both patterns can be used to deal with failures.

What support does Camel have for circuit breakers?

Circuit Breaker and Camel

Camel offers two implementations:

- Load Balancer with Circuit Breaker
- Netflix Hystrix

The first implementation of the Circuit Breaker pattern is an extension to the Load Balancer pattern. This implementation comes out of the box from camel-core and provides a basic implementation. But recently the Netflix Hystrix library has become popular, so the Camel team integrated Hystrix as a native EIP pattern in Camel. We recommend using the latter, which is also the implementation covered in this book.

7.4.4 Netflix Hystrix

Hystrix is a latency and fault-tolerant library designed to isolate points of access to remote services and stop cascading failures in complex distributed systems where failures are inevitable. Hystrix provides the following:

- Protection against services that are unavailable
- Time-out to guard against unresponsive services
- Separation of resources using the Bulkhead pattern
- Shedding loads and failing fast instead of queueing
- Self-healing
- Real-time monitoring

We'll describe each of these bullet items, but let's start with a simple Hystrix example.

Using Hystrix with plain Java

Hystrix provides the `HystrixCommand` class, which you extend by implementing the `run` method containing your potential faulty code, such as a remote network call. The following listing shows how this is done.

Listing 7.14 Creating a custom Hystrix command

```
public class MyCommand extends HystrixCommand<String> {    1
```

1

Extending HystrixCommand

```
    public MyCommand() {
        super(HystrixCommandGroupKey.Factory.asKey("MyGroup"));    2
```

2

Specifying group

```
    }

    protected String run() throws Exception {    3
```

3

All unsafe code goes into run method

```
        COUNTER++;
        if (COUNTER % 5 == 0) {
            throw new IOException("Forced error");    4
```

4

Simulated error every fifth call

```
    }
    return "Count " + COUNTER;
  }
}
```

As you can see, we've created our custom Hystrix command in the `MyCommand` class by extending `HystrixCommand<String>` ❶ with `String` specified as the type, which then corresponds to the return type of the `run` method ❸. Every Hystrix command must be associated with a group, which we configured in the constructor ❷. All the potentially unsafe and faulty code is put inside the `run` method ❸. In this example, we've built into the code a failure that happens for every fifth call to simulate some kind of error ❹.

Calling a Hystrix command from Java

How do you use this command? That's simple with Hystrix: you create an instance of it and call the—no, not the `run` method—but the `execute` method, as shown here:

```
MyCommand myCommand = new MyCommand();
String out = myCommand.execute();
```

Here's the result of running this command:

```
Count 1
```

If you run it again like this

```
MyCommand myCommand = new MyCommand();
String out = myCommand.execute();
String out2 = myCommand.execute();
```

then you'd expect the following:

```
Count 1
Count 2
```

But that isn't what happens. Instead, you have this:

```
Count 1
com.netflix.hystrix.exception.HystrixRuntimeException: MyCommand command
```

That's an important aspect with Hystrix: each command isn't reusable; you can use a command once and only once. You can't store any state in a command, because it can be called only once, and no state is left over for successive calls.

You can find this example in the source code, in the chapter7/hystrix directory. Try the example by using the following Maven goal:

```
mvn test -Dtest=MyCommandTest
```

You may have noticed in listing 7.14 the `COUNTER` instance, which is used as a global state stored outside the Hystrix command, and the fact that the example is constructed to fail every fifth call. How do you deal with exceptions thrown from the `run` method?

Adding fallback

When the `run` method can't be executed successfully, you can use the Hystrix built-in fallback method to return an alternative response. For example, to add a fallback to listing 7.14, you override the `getFallback` method as follows:

```
    protected String getFallback() {
      return "No Counter";
    }
```

Running this command 10 times yields the following output:

```
new MyCommand().execute()  ->  Count 1
new MyCommand().execute()  ->  Count 2
new MyCommand().execute()  ->  Count 3
new MyCommand().execute()  ->  Count 4
new MyCommand().execute()  ->  No Counter
new MyCommand().execute()  ->  Count 6
new MyCommand().execute()  ->  Count 7
new MyCommand().execute()  ->  Count 8
new MyCommand().execute()  ->  Count 9
new MyCommand().execute()  ->  No Counter
```

The example is constructed to fail every fifth time, which is when the fall-back kicks in and returns the alternative response.

You can try this example, which is in the source code in the chapter7/hystrix directory, by using the following Maven goal:

```
mvn test -Dtest=MyCommandWithFallbackTest
```

Okay, what about using Hystrix with Camel?

7.4.5 Using Hystrix with Camel

Camel makes it easy to use Hystrix. All you have to do is to add the camel-hystrix dependency to your project and then use the Hystrix EIP in your Camel routes.

Let's migrate the previous example from section 7.4.4 to use Camel. The code from listing 7.14, which was implemented in the `run` method of `HystrixCommand`, is refactored into a plain Java bean:

```
public class CounterService {
  private int counter;   1
```

1

Storing counter state in the bean

```
  public String count() throws IOException {   2
```

2

The unsafe code

```
      counter++;
      if (counter % 5 == 0) {
        throw new IOException("Forced error");
      }
      return "Count " + counter;
    }
  }
```

The code that was previously in the `run` method of `HystrixCommand` is migrated as is in the `count` method ❷. We can now store the counter state directly in the bean ❶ because the bean isn't a `HystrixCommand` implementation. From section 7.4.4, you learned that it's a requirement for a Hystrix command to be stateless and executable only once, which forced us to store any state outside the `HystrixCommand`.

Using Hystrix in Camel routes is so easy, you need just a few lines of code:

```
public void configure() throws Exception {
  from("direct:start")
    .hystrix()     ❶
```

❶ Hystrix EIP

```
        .to("bean:counter")
    .end()     ❷
```

❷ End of Hystrix block

```
      .log("After calling counter service: ${body}");
}
```

In the route, you use Hystrix to denote the beginning of the protection of Hystrix. Any of the following nodes are run from within `HystrixCommand`. In this example, that would be calling the counter bean. To denote when the Hystrix command ends, you use `end` ❷ in Java DSL, which means the log node is run outside Hystrix.

You can have as many service calls and EIPs as you like inside the Hystrix block. For example, to call a second bean, you can do this:

```
.hystrix()
  .to("bean:counter")
  .to("bean:anotherBean")
.end()
```

Using Hystrix in XML is just as easy. The equivalent route is shown in the following listing.

Listing 7.15 Using Hystrix EIP using XML DSL

```
<bean id="counterService" class="camelinaction.CounterService"/> (1)
```

(1) Defines counter bean

```
  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="direct:start"/>
   <hystrix> (2)
```

(2) Hystrix EIP

```
        <to uri="bean:counterService"/>
   </hystrix> (3)
```

(3) End of Hystrix block

```
      <log message="After calling counter service: ${body}"/>
    </route>
```

```
  </camelContext>
```

In XML DSL, we first define the counter service bean as a Spring `<bean>` ❶. In the Camel route, you can easily use Hystrix by using `<hystrix>` ❷. The nodes inside the Hystrix block ❸ are then run under the control from a `HystrixCommand`.

What about using fallbacks?

Adding fallback

Using a Hystrix fallback with Camel is also easy:

```
public void configure() throws Exception {
  from("direct:start")
    .hystrix()
      .to("bean:counter")
  .onFallback()    ❶
```

❶
Hystrix fallback

```
      .transform(constant("No Counter"))
    .end()
    .log("After calling counter service: ${body}");
}
```

The fallback is added by using `.onFallback()` ❶, and the following nodes are included. In this example, we perform a message transformation to set the message body to the constant value of `No Counter`. Using fallback with XML DSL is easy as well:

```
<route>
  <from uri="direct:start"/>
  <hystrix>
    <to uri="bean:counterService"/>
  <onFallback>
```

Hystrix fallback

```
        <transform>
          <constant>No Counter</constant>
        </transform>
      </onFallback>
    </hystrix>
    <log message="After calling counter service: ${body}"/>
  </route>
```

You can find this example with the source code in the chapter7/hystrix-camel directory. Try it using the following Maven goals:

```
mvn test -Dtest=CamelHystrixWithFallbackTest
mvn test -Dtest=SpringCamelHystrixWithFallbackTest
```

One aspect we haven't delved into is that each `HystrixCommand` must be uniquely defined using a command key. Hystrix enforces this in the constructor of `HystrixCommand`, as in listing 7.14 when we weren't using Camel.

Configuring command key

Each Hystrix EIP in Camel is uniquely identified via its node ID, which by default is the Hystrix command key. For example, the following route

```
.hystrix()
  .to("bean:counter")
  .to("bean:anotherBean")
.end()
```

would use the autoassigned node IDs, which typically are named using the node and a counter such as `hystrix1`, `bean1`, `bean2`. The command key of the Hystrix EIP would be `hystrix1`. If you want to assign a specific key, you need to configure the node ID with the value shown in bold:

```
.hystrix().id("MyHystrixKey")
  .to("bean:counter")
```

```
    .to("bean:anotherBean")
  .end()
```

In XML DSL, you’d do as follows:

```
<hystrix id="MyHystrixKey">
  <to uri="bean:counterService"/>
  <to uri="bean:anotherBean"/>
</hystrix>
```

Speaking of configuring Hystrix, it has a wealth of configuration options that you can use to tailor all kind of behaviors. Some options are more important to know than others, which you’ll learn about through the Bulkhead pattern.

But first let’s cover the basics of configuring Hystrix.

7.4.6 Configuring Hystrix

Configuring Hystrix isn’t a simple task. Many options are available and you can tweak all kinds of details related to the way the circuit breaker should operate. All these options play a role in allowing fine-grained configuration. Your first encounters with these options will leave you baffled, and learning the nuances between the options and the role they play takes time. As a rule of thumb, the out-of-the box settings are a good compromise, and you’ll most often use only a few options. This section shows you how to configure Hystrix in pure Java and as well with Camel and Hystrix.

Configuring Hystrix with just Java

You configure `HystrixCommand` in the constructor by using a *somewhat special* fluent builder style. For example, to configure a command to open the circuit breaker if you get 10 or more failed requests within a rolling time period of 5 seconds, you can configure this as follows:

```
public MyConfiguredCommand() {
  super(Setter
    .withGroupKey(HystrixCommandGroupKey.Factory.asKey("MyGroup"))
    .andCommandPropertiesDefaults(
      HystrixCommandProperties.Setter()
```

```
            .withCircuitBreakerRequestVolumeThreshold(10)
            .withMetricsRollingStatisticalWindowInMilliseconds(5000)
    ));
  }
```

Yes, it takes time to learn how to configure Hystrix when using this configuration style. Plenty more options are available than shown here, and we refer you to the Hystrix documentation (https://github.com/Netflix/Hystrix/wiki) to learn more about every option you can use.

Configuring Hystrix in Camel is easier.

Configuring Hystrix with Camel

The previous example can be configured in Camel as follows:

```
.hystrix()
  .hystrixConfiguration() 1
```

1

Hystrix configuration

```
    .circuitBreakerRequestVolumeThreshold(10)
    .metricsRollingPercentileWindowInMilliseconds(5000)
  .end() 2
```

2

End of Hystrix configuration

```
  .to("bean:counter")
.end()
```

End of Hystrix EIP

As you can see, Hystrix is configured using `hystrixConfiguration()` ❶, which provides type-safe DSL with all the possible options. Pay attention to how `.end()` ❷ is used to mark the end of the configuration.

In XML DSL, you configure Hystrix using the `<hystrixConfiguration>` element:

```
<hystrix id="MyGroup">
  <hystrixConfiguration
        circuitBreakerRequestVolumeThreshold="10"
        metricsRollingStatisticalWindowInMilliseconds="5000"/>
    <to uri="bean:counterService"/>
</hystrix>
```

As mentioned, Hystrix has a lot of options, but let's take a moment to highlight the options we think are of importance.

Important Hystrix options

Table 7.2 lists the Hystrix options we think are worthwhile to learn to use first.

Table 7.2 Important Hystrix options

Option	Default value	
`commandKey`		Identifies can't be c the node
`groupKey`	`CamelHystrix`	Identifies EIP to cor propertie
`circuitBreakerRequestVolumeThreshold`	`20`	Sets the n rolling wi below thi regardles
`circuitBreakerErrorThresholdPercentage`	`50`	Indicates whole nu circuit br requests. defined in `circuitE` option.
`circuitBreakerSleepWindowInMilliseconds`	`5000`	Sets the ti breaker t trying rec
`metricsRollingStatisticalWindowInMilliseconds`	`10000`	Indicates window i metrics a
`corePoolSize`	`10`	Sets the c maximur can execu

Option	Default value	
`maxQueueSize`	`-1`	Sets the n pool task
`executionTimeoutInMilliseconds`	`1000`	Indicates point the executior

Some of these options are covered in the next section about the Bulkhead pattern.

7.4.7 Bulkhead pattern

Imagine that a downstream service is under pressure, and your application keeps calling the downstream service. The service is becoming latent, but not enough to trigger a time-out in your application nor trip the circuit breaker. In such a situation, the latency can stall (or appear to stall) all worker threads and cascade the latency all the way back to users. You want to limit the latency to only the dependency that's causing the slowness without consuming all the available resources in your application, process, or compute node.

The solution to this is the Bulkhead pattern. A *bulkhead* is a separation of resources such that one set of resources doesn't impact others. This pattern is well known on ships as a way to create watertight compartments that can contain water in the case of a hull breach. A famous tragedy is the *Titanic*, which had bulkheads that were too low and didn't prevent water from leaking into adjacent compartments, eventually causing the ship to sink.

Hystrix implements the Bulkhead pattern with thread pools. Each Hystrix command is allocated a thread pool. If a downstream service becomes latent, the thread pool can become fully utilized and fail fast by rejecting new requests to the command. The thread pools are isolated from each other, so other commands can still operate successfully.

By default, the bulkhead is enabled, and each command is assigned a thread pool of 10 worker threads. The thread pool has no task pool as backlog, so if all 10 threads are used, Hystrix will reject processing new requests and fail fast.

The option `corePoolSize` can be used to configure the number of threads in the pool, and the task queue is enabled by setting the `maxQueueSize` option to a positive size.

If the additional thread pools are a concern, Hystrix can implement the bulkhead on the calling thread with counting semaphores instead. Refer to the Hystrix documentation for more information.

Another powerful feature from Hystrix is the execution time-out.

Time-out with Hystrix

Every Hystrix command is executed within the scrutiny of a time-out period. If the command doesn't complete within the given time, Hystrix will react and cause the command to fail. This time-out is 100% controlled by Hystrix and shouldn't be mistaken for any time-outs from Camel components or downstream services. Hystrix comes with a time-out out of the box. The default value is sensitive at only 1 second, which is a low time-out. Therefore, you may find yourself often configuring this number to a value that suits your needs.

The time-out is listed last in table 7.2 and can be configured as shown in Java DSL:

```
from("direct:start")
  .hystrix()
  .hystrixConfiguration().executionTimeoutInMilliseconds(2000).end()
```

2 seconds as time-out

```
    .toD("direct:${body}")
  .end();
```

Here's the configuration in XML DSL:

```
<route>
  <from uri="direct:start"/>
  <hystrix>
   <hystrixConfiguration executionTimeoutInMilliseconds="2000"/>
```

2 seconds as time-out

```
      <toD uri="direct:${body}"/>
  </hystrix>
</route>
```

The book's source code includes an example of using Hystrix to call downstream Camel routes that process either 1 or 3 seconds before responding. This demonstrates that Hystrix with a time-out value of 2 seconds will succeed when using a fast response, and will fail with a time-out exception when using a slow response. You can try the example, located in the chapter7/hystrix-camel directory, using the following Maven goals:

```
mvn test -Dtest=CamelHystrixTimeoutTest
mvn test -Dtest=SpringCamelHystrixTimeoutTest
```

When a Hystrix time-out occurs, the Hystrix command is regarded as failed, and the Camel route fails with a time-out exception.

Time-out and fallback with Hystrix

If a time-out error happens, you may want to let Hystrix handle this by a fallback to set up an alternative response. This can easily be done with Camel by using `onFallback`:

```
from("direct:start")
  .hystrix()
    .hystrixConfiguration()
      .executionTimeoutInMilliseconds(2000)
    .end()
    .log("Hystrix processing start: ${threadName}")
    .toD("direct:${body}")
    .log("Hystrix processing end: ${threadName}")
  .onFallback()
```

Hystrix fallback when any error happens

```
    .log("Hystrix fallback start: ${threadName}")
    .transform().constant("Fallback response")
    .log("Hystrix fallback end: ${threadName}")
  .end()
  .log("After Hystrix ${body}");
```

You can find this example with the source code in the chapter7/hystrix-camel directory, which can be run with the following Maven goals:

```
mvn test -Dtest=CamelHystrixTimeoutAndFallbackTest
mvn test -Dtest=SpringCamelHystrixTimeoutAndFallbackTest
```

If you run the example, the fast and slow response tests will output the following to the console.

The fast response logs:

```
2017-10-01 20:21:14,654 [-CamelHystrix-1] INFO  route1
2017-10-01 20:21:14,660 [-CamelHystrix-1] INFO  route2
2017-10-01 20:21:15,662 [-CamelHystrix-1] INFO  route2
2017-10-01 20:21:15,663 [-CamelHystrix-1] INFO  route1
2017-10-01 20:21:15,666 [main           ] INFO  route1
```

And the slow response logs:

```
2017-10-01 20:23:08,035 [-CamelHystrix-1] INFO  route3
2017-10-01 20:23:10,023 [HystrixTimer-1 ] INFO  route1
2017-10-01 20:23:10,023 [HystrixTimer-1 ] INFO  route1
2017-10-01 20:23:10,026 [main           ] INFO  route1
     - After Hystrix Fallback response
```

Why do we show you these log lines? The clue is the information highlighted in bold. That information shows the name of the thread that's processing. In the test with the fast response, it's the same thread, `CamelHystrix-1`, that routes the Camel message in the Hystrix command (Hystrix EIP). The last log line is outside the Hystrix EIP and therefore outside the Hystrix command, so it's the caller thread that's processing the message again, which in the example is the `main` thread that started the test.

On the other hand, in the slow test, the log shows that Hystrix is using two threads to process the message, `CamelHystrix-1` and `HystrixTimer-1`. This is the Bulkhead pattern in action, where the run and fallback methods from the Hystrix command are separated and being processed by separated threads.

Figure 7.8 illustrates this principle.

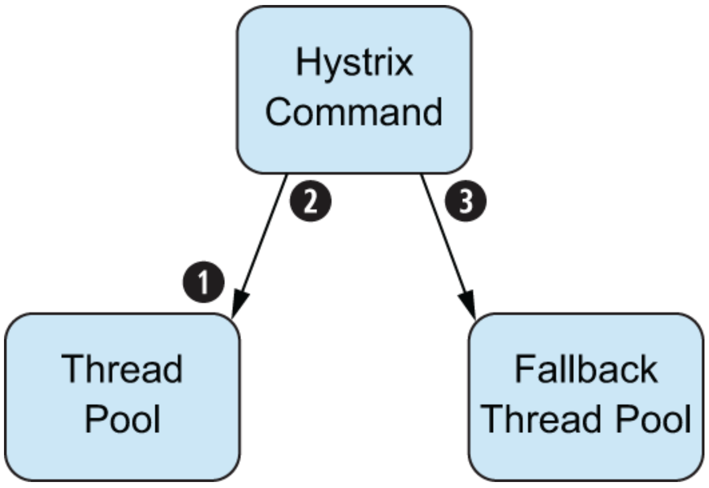


Figure 7.8 The Hystrix command uses a thread from the regular thread pool ❶ to route the Camel message. The task runs, and upon completion, the command completes successfully ❷. But the task can also fail because of a time-out or an exception thrown during Camel routing, which causes the Hystrix command to run the fallback using a thread from the fallback thread pool ❸. This ensures isolation between run and fallback, adhering to the bulkhead principle.

Hystrix fallbacks have one important caveat. In the example, you may have wondered why the fallback thread name is named `HystrixTimer-1`. That's because Hystrix triggered a time-out and executed the fallback. This time-out thread is from the time-out functionality within Hystrix that's shared across all the Hystrix commands.

Now we're down to the caveat: your Hystrix fallbacks should be written as short methods that do simple in-memory computations without any network dependency.

TIP Hystrix offers live metrics of all the states of the commands and their thread pools. This information can be visualized using the Hystrix dashboard, which is covered in section 7.4.10.

Fallback via network

If the fallback must do a network call, you have to use another Hystrix command that comes with its own thread pool to ensure total thread isolation. With Camel, this is easy, because we've implemented a special fallback for network calls that's named `onFallbackViaNetwork`:

```
from("jetty:http://0.0.0.0/myservice")
  .hystrix()
    .to("http://server-one")
 .onFallbackViaNetwork()
```

Fallback with a network call

```
    .to("http://server-backup")
```

This route exposes an HTTP service using Jetty, which proxies to another HTTP server on http://server-one. If the downstream service isn't available, Hystrix will fall back to another downstream service over the network at http://server-backup.

FALLBACK ISN'T A ONE-SIZE-FITS-ALL SOLUTION

A Hystrix fallback isn't a silver bullet that solves every problem in a microservices or distributed architecture. Whether a fallback is applicable is domain specific and determined case by case. In the Rider Auto Parts example with the recommendation service, a fallback can be useful. In this case, the recommendation service is used for displaying a personalized list for the user, but what if it isn't available or is too slow to respond? You can degrade to a generic list for users in a particular region, or just a generic list. But in other domains such as a flight-booking system, a service that performs the booking at the airline could likely not fallback because a flight seat can't be guaranteed to have been booked with the airline.

With fresh and new knowledge in your toolbelt, you arrange for another weekend where the spouse and kids are away on a retreat. To have a good time, you've stocked up with a fresh six-pack of beer and a bottle of good scotch, just in case you run out of beer.

7.4.8 Calling other microservices with fault-tolerance

The previous prototype you built was a set of microservices that collectively implemented a recommendation system for Rider Auto Parts. But the prototype wasn't designed for failures. How can you apply fault-tolerance to those microservices? This section details what you did during the weekend to add fault-tolerance to your prototype.

First, figure 7.9 illustrates the collaborations among the microservices and the places where they're in trouble.

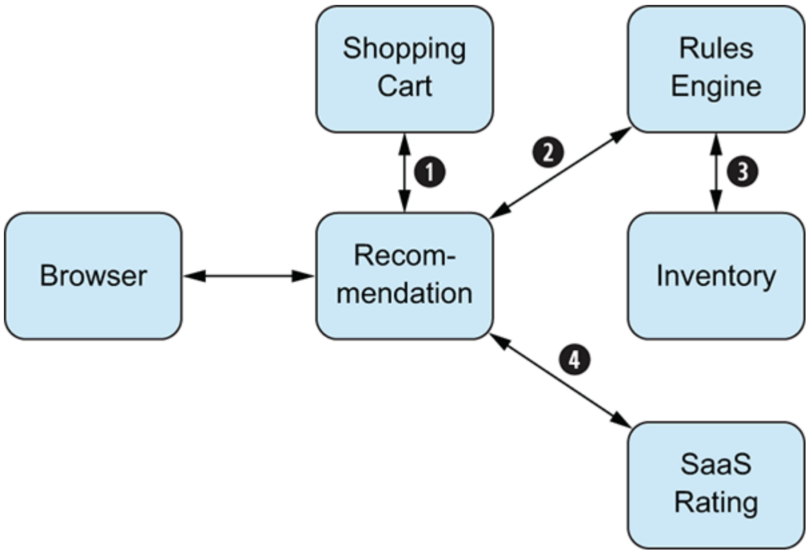


Figure 7.9 Rider Auto Parts recommendation system. Each arrow represents a dependency among the microservices. Each number represents places where you need to add fault-tolerance.

As you can see in figure 7.9, you need to add fault-tolerance four times ❶ ❷ ❸ ❹ to the prototype. Let's start from the top with the recommendation microservice.

Using Hystrix with Spring Boot

The recommendation microservice is a Spring Boot application that doesn't use Camel. To use Hystrix with Spring Boot, you need to add the following dependency to the Maven pom.xml file:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-hystrix</artifactId>
  <version>1.3.5.RELEASE</version>
</dependency>
```

The support for Hystrix comes with the Spring Cloud project, hence the name of the dependency.

Support for Hystrix on Spring Boot requires you to implement an `@Service` class, which holds the code to run as a Hystrix command. The following listing shows what you did to add fault-tolerance at number ❶ in figure 7.9.

Listing 7.16 Using Hystrix in Spring Boot as an `@Service` class

```
@Service    ❶
```

1

Annotates class as a Spring @service class

```
public class ShoppingCartService {

  private final RestTemplate restTemplate = new RestTemplate();

@HystrixCommand(fallbackMethod = "emptyCart")     2
```

2

Marks method as Hystrix command

```
  public String shoppingCart(String cartUrl, String id) {
    CartDto[] carts = restTemplate.getForObject(cartUrl,
                        CartDto[].class, id);     3
```

3

Calls downstream service

```
    String cartIds = Arrays.stream(carts)
                .map(CartDto::toString)
          .collect(Collectors.joining(","));     4
```

4

Converts response to a comma-separated String

```
    return cartIds;
  }

public String emptyCart(String cartUrl, String id) {     5
```

5

Method used as Hystrix fallback

```
      return "";
    }
  }
```

Here you create a new class, `ShoppingCartService`, which is responsible for calling the downstream shopping cart service. The class has been annotated with `@Service` ❶ (or `@Component`), which is required by Spring when using Hystrix. The `@HystrixCommand` annotation ❷ is added on the method that will be wrapped as a Hystrix command. All the code that runs inside this method will be under the control of Hystrix. Inside this method, you call the downstream service ❸ using Spring's `RestTemplate` to perform a REST call. Having a response, you transform this into a comma-separated string using Java 8 *stream magic* ❹. The `@HystrixCommand` annotation specifies the name of the fallback method that refers to the `emptyCart` method ❺. This method does what the name says: returns an empty shopping cart as its response.

Pay attention to the method signature of the two methods in play. The fallback method must have the exact same method signature as the method with the `@HystrixCommand` annotation.

You follow the same practice to implement fault-tolerance to calling the rules and rating microservices (❷ and ❹ in figure 7.9).

The last code change needed is in the `RecommendController` class, which orchestrates the recommendation service. The following listing lists the updated source code.

Listing 7.17 `RecommendController` using the fault-tolerant services

```
@EnableCircuitBreaker    ❶
```

❶
Enables Hystrix Circuit Breaker

```
@RestController
@RequestMapping("/api")
@ConfigurationProperties(prefix = "recommend")
```

```
public class RecommendController {

  private String cartUrl;
  private String rulesUrl;
  private String ratingsUrl;

 @Autowired    2
```

2

Dependency inject services to be used

```
 private ShoppingCartService shoppingCart;    2
 @Autowired    2
 private RulesService rules;    2
 @Autowired    2
 private RatingService rating;    2

  @RequestMapping(value = "recommend", method = RequestMethod.GET,
                  produces = "application/json")
  public List<ItemDto> recommend(HttpSession session) {
    String id = session.getId();

   String cartIds = shoppingCart.shoppingCart(cartUrl, id);    3
```

3

Calls shopping cart service

```
   ItemDto[] items = rules.rules(rulesUrl, id, cartIds);    4
```

4

Calls rules service

```
    String itemIds = itemsToCommaString(items);

   RatingDto[] ratings = rating.rating(ratingsUrl, itemIds);    5
```

❺

Calls rating service

```
      for (RatingDto rating : ratings) {
        appendRatingToItem(rating, items);
      }
      return Arrays.asList(items);
    }
  }
```

The annotation `@EnableCircuitBreaker` ❶ is used to turn on Hystrix with Spring Boot. You then have dependency-injected the three classes with the Hystrix command ❷, which will be used from the controller to call the downstream microservices. Because the logic to call those downstream services is now moved into those separate classes, the code in the recommend method is simpler. All you do is call those services in order (❸ ❹ ❺) as if they were regular Java method calls. But under the covers, those method calls will be under the wings of Hystrix, wrapped in a Hystrix command with fault-tolerance included in the batteries.

Trying the example

Now you're ready to try this in action. Run the code from the chapter7/prototype2/recommend2 directory using the following goal from Maven:

```
mvn spring-boot:run
```

Then open a web browser to http://localhost:8080/api/recommend, which returns the following response:

```
[{"itemNo":999,"name":"special","description":"Special Rider Auto Part p
```

What you see is that when all the downstream services aren't available, the application returns a fallback response. The fallback response is created because of these conditions:

- *ShoppingCartService*—Returns an empty cart as a response
- *RulesService*—Returns the special item with number 999 as fallback

- *RatingService*—Gives each item a rating of 6 as fallback

You turn on each of these downstream services one by one and see how the response changes accordingly.

To turn on the shopping cart, you run the following Maven command from another shell:

```
cd chapter7/prototype2/cart2
mvn compile exec:java
```

Then you call the recommendation service again from the following URL: http://localhost:8080/api/recommend, which outputs the following response:

```
[{"itemNo":999,"name":"special","description":"Special Rider Auto Partpr
```

The response is similar to before, which isn't a surprise because no items have been added to the shopping cart. Then you start the rules service by executing from another shell:

```
cd chapter7/prototype2/rules2
mvn wildfly-swarm:run
```

And by calling the recommendation service, the response is now as follows:

```
[{"itemNo":999,"name":"special","description":"Special Rider Auto Partpr
```

Oh boy, it's still the same response. Why is that? You do what every experienced developer always does when something isn't right—you go home. But you're already home, so what can you do? You take a little break and grab a beer from the six-pack. Dear reader, you're almost at the end of this chapter, so you don't have to take a break just yet, but you're surely welcome to enjoy a refreshment while reading.

Sitting back at the desk, you do as a professional developer would do: you go look in the logs. Looking at the rules service logs, you can see the following problem:

```
2017-10-06 10:52:30,984 ERROR [org.apache.camel.component.jms.DefaultJmsl
```

Ah, yeah, of course, the rules service isn't fault-tolerant and fails because it can't connect to the inventory service over JMS. This is the connection at number 3 from figure 7.9. Before you start adding Hystrix to the rules service, let's continue the test and start the last service, which is the rating service. From another shell, you run the following commands:

```
cd chapter7/prototype2/rating2
mvn spring-boot:run
```

By refreshing the call to the rating service, the response is now as follows:

```
[{"itemNo":999,"name":"special","description":"Special Rider Auto Partpr
```

At first glance, the response looks the same, but when you refresh the web browser to call the URL again, a subtle change occurs in the rating:

```
[{"itemNo":999,"name":"special","description":"Special Rider Auto Part
premium service","number":100,"rating":98}]
```

The rating service was implemented to return a random number. Notice that the first rating was 24, and the second is 98, as highlighted in bold.

Okay, the last piece of the puzzle is to add fault-tolerance to the rules service that's implemented using WildFly Swarm with Camel. So let's get back to Camel land.

Using Hystrix with Camel and WildFly Swarm

Your last task for the prototype is to add fault-tolerance to the rules service, which corresponds to ❸ in figure 7.9. This service is using Camel running on WildFly Swarm. To use Hystrix, you have to add the camel-hystrix fraction as a Maven dependency in the pom.xml file:

```
<dependency>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>camel-hystrix</artifactId>
</dependency>
```

Using Hystrix with Camel is easy, as you've already learned, so all you've done is modify the Camel route as shown here:

```
public void configure() throws Exception {
 JaxbDataFormat jaxb = new JaxbDataFormat();   ❶
```

❶

Sets up JAXB for XML

```
 jaxb.setContextPath("camelinaction");   ❶

    from("direct:inventory")
      .hystrix()   ❷
```

❷

Hystrix EIP

```
        .to("jms:queue:inventory")
        .unmarshal(jaxb)
      .onFallback()   ❸
```

❸

Hystrix fallback

```
        .transform().constant("resource:classpath:fallback-inventory.xml
  }
```

Because the response from the inventory service is in XML format, and this route is expected to return data as a POJO, you need to set up JAXB ❶, which is later used to unmarshal from XML to POJO. The call to the inventory service is protected by the Hystrix circuit breaker ❷. In case of any failures, the fallback is executed ❸, which loads a static response from an XML file embedded in the classpath. The fallback response is the following item:

```
<items>
  <item>
    <itemNo>998</itemNo>
    <name>Rider Auto Part Generic Brakepad</name>
    <description>Basic brakepad for any motorbike</description>
    <number>100</number>
  </item>
</items>
```

Coding this was so fast that you didn't finish the beer you opened a while back when you took the break. But after all, the coding required adding only a Maven dependency, and then three lines of code in the Camel route, and creating an XML file with the fallback response.

You've now implemented fault-tolerance using Hystrix EIP in all the microservices at the numbers ❶ ❷ ❸ and ❹ from figure 7.9. Let's try the complex example.

Continue trying the example

To continue trying the prototype, you start the rules microservice again, now that it's been updated with fault-tolerance. From the shell, you type this:

```
cd chapter7/prototype2/rules2
mvn wildfly-swarm:run
```

Then you call the recommendation service URL, http://localhost:8080/api/recommend:

```
[{"itemNo":998,"name":"Rider Auto Part Generic Brakepad","description":"
```

The response looks familiar. It's the fallback response from the rules service. But didn't we add fault-tolerance to this? Why is it doing this? Just as you grab another beer from the six-pack, you realize that the rules service calls the downstream inventory service, which is represented as number ❸ in figure 7.9. And it's failing with a fallback response because you haven't yet started the inventory service. You quickly open another shell and type this:

```
cd chapter7/prototype2/inventory2
mvn compile exec:java
```

And you just happen to glance at the logs from the rules service and notice that it stopped logging stacktraces and logged that it had successfully reconnected to the message broker, which was just started as part of the inventory service:

```
2017-10-06 12:01:56,558 INFO[org.apache.camel.component.jms.DefaultJmsMe
```

With that in mind, you're confident that by calling the recommendation service, it should produce a different response:

```
[{"itemNo":456,"name":"Suzuki Brakepad","description":"Basic brakepad fo
Suzuki","number":149,"rating":28},{"itemNo":789,"name":"Suzuki Engine
500","description":"Suzuki 500cc engine","number":4,"rating":80}]
```

Yay, all the microservices are up and running, and the recommendation system returns a positive response. One last thing you want to try is to see whether the rules service is fault-tolerant when the inventory service isn't available. You stop the inventory service and call the recommendation URL again, which returns the following:

```
[{"itemNo":998,"name":"Rider Auto Part Generic Brakepad",
"description":"Basic brakepad for any motorbike","number":100,"rating":3
```

That's the fallback response from the rules service, which would be intended to return a list of known items that Rider Auto Parts always has on stock.

You still have one beer left from the six-pack and the whisky to celebrate your success. So let's finish the weekend on a high note and build the rating microservice using Spring Boot.

7.4.9 Using Camel Hystrix with Spring Boot

It's easy to use Camel and Hystrix on Spring Boot. What you want to do is to migrate the rules microservices from using WildFly Swarm with Camel and Hystrix to using Spring Boot with Camel and Hystrix.

The first thing you do is set up the Maven pom.xml file to include the Spring Boot dependencies and the following Camel starter dependencies: camel-core-starter, camel-hystrix-starter, camel-jaxb-starter, and camel-jms-starter.

The second thing is to create the REST controller, as in the following listing.

Listing 7.18 Spring REST controller calling a Camel route using direct endpoint

```
@RestController   1
```

1

Spring REST controller

```
@RequestMapping("/api")
public class RulesController {

 @Produce(uri = "direct:inventory")   2
```

2

Injects Camel ProducerTemplate

```
    private FluentProducerTemplate producer;

    @RequestMapping(value = "rules/{cartIds}",
              method = RequestMethod.GET, produces = "application/json")
    public List<ItemDto> rules(@PathVariable String cartIds) {
    ItemsDto inventory = producer.request(ItemsDto.class);   3
```

3

Calls Camel route

```
    return inventory.getItems().stream()   4
```

❹

Filters, sorts, and builds response as list

```
        .filter((i) -> cartIds == null
                      || !cartIds.contains("" + i.getItemNo()))
        .sorted(new ItemSorter())
        .collect(Collectors.toList());
    }
  }
```

The REST controller uses Spring `@RestController` ❶ instead of JAX-RS used by WildFly Swarm. The controller class uses Camel `FluentProducerTemplate` ❷ to call the Camel route ❸ that does the call of the downstream service using Hystrix. The response from the Camel route is then filtered, sorted, and collected as a list ❹ by using the Java 8 stream API.

The Camel route that uses the Hystrix EIP is shown in the following listing.

Listing 7.19 Camel route using Hystrix to call downstream service using JMS

```
@Component    ❶
```

❶

Enables automatic component discovery

```
public class InventoryRoute extends RouteBuilder {

  public void configure() throws Exception {
    JaxbDataFormat jaxb = new JaxbDataFormat();
    jaxb.setContextPath("camelinaction");

    from("direct:inventory")
      .hystrix()
        .to("jms:queue:inventory")
        .unmarshal(jaxb)
      .onFallback()
        .transform().constant("resource:classpath:fallback-inventory.xml
  }
}
```

Migrating the Camel route shown in listing 7.19 from WildFly Swarm to Spring Boot required only one change. The class annotation uses Spring's `@Component` ❶ instead of WildFly Swarm using CDI's `@ApplicationScoped`. That's all that was needed. That's a testimony that Camel is indifferent and works on any runtime as is.

The last migration effort is to set up Spring Boot to use ActiveMQ and set up the connection to the message broker. You do this by adding the following dependency to the Maven pom.xml file:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-activemq</artifactId>
</dependency>
```

Then you set up the connection to the ActiveMQ broker in the `application.properties` file:

```
spring.activemq.broker-url=tcp://localhost:61616
server.port=8181
```

The other setup is to configure Spring Boot to use HTTP port 8181, which is the port number the rules service is expected to use.

With all the beers consumed, you're almost fooled by how the Camel route in Listing 7.19 knows about the Spring Boot ActiveMQ broker setting we just made in the application.properties file. That's because the Camel JMS component is using Spring JMS under the hood. And Spring JMS has, not surprisingly, support for Spring Boot and can automatically wire up to the ActiveMQ broker.

You're now ready to run this example. All the other microservices has been stopped, so when you start this example, it's the only JVM running on your computer:

```
cd chapter7/prototype2/rules2-springboot
mvn spring-boot:run
```

And this time you call the rules service directly using the following URL: http://localhost:8181/api/rules/999, which responds with the fallback response:

```
[{"itemNo":998,"name":"Rider Auto Part Generic Brakepad",
"description":"Basic brakepad for any motorbike","number":100,"rating":0
```

That's expected because the inventory service isn't running, and therefore there's no ActiveMQ broker running that the rules service can call. Therefore, you start the inventory service by running these commands from another shell:

```
cd chapter7/prototype2/inventory2
mvn compile exec:java
```

And you call the rules service again, which returns a different response:

```
[{"itemNo":456,"name":"Suzuki Brakepad","description":"Basic brakepad fo
Suzuki","number":149,"rating":0},{"itemNo":123,"name":"Honda Brakepad",
"description":"Basic brakepad for Honda","number":28,"rating":0},
{"itemNo":789,"name":"Suzuki Engine 500","description":"Suzuki 500cc
engine","number":4,"rating":0}]
```

You're almost done for the day but just want to test the fault-tolerance, so you stop the inventory service that hosts the ActiveMQ broker and call the service yet again, which returns the fallback response:

```
[{"itemNo":998,"name":"Rider Auto Part Generic Brakepad",
"description":"Basic brakepad for any motorbike","number":100,"rating":0
```

There's still time for one more adventure before the weekend comes to a close. The prototype includes four communication points that are protected by Hystrix circuit breakers (figure 7.9). When the recommendation service is called, four service calls could potentially fail and return a fallback response. How do you know where there are problems? That's a good question, and it's related to a whole other topic about monitoring and distributed systems, which we'll cover much more in chapters 16 and 18.

After six beers, we don't want to leave you hanging, so let's embark on another little adventure before wrapping up this chapter. How can we gain insight into the state of the circuit breakers?

7.4.10 The Hystrix dashboard

The Hystrix dashboard is used for visualizing the states of all your circuit breakers in your applications. For this to work, you must make sure each of your applications and microservices are able to provide real-time metrics to be fed into the dashboard.

The weekend is coming to a close, and you just want to quickly try to see what it would take to set up the Hystrix dashboard to visualize one of our microservices, the rules microservice.

To be able to feed live metrics from Hystrix into the dashboard, you need to enable *Hystrix streams*. These are easily enabled when using Spring Boot by adding the following dependencies to the Maven pom.xml file:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-hystrix</artifactId>
  <version>1.3.5.RELEASE</version>
</dependency>
```

The Spring Boot Actuator enables real-time monitoring capabilities such as health checks, metrics, and support for Hystrix streams. The support for Hystrix is provided by the Spring Cloud Hystrix dependency.

The last thing you need to do is to turn on Hystrix circuit breakers in the Spring Boot application, which you can do by adding the `@EnableCircuitBreaker` annotation to your `SpringBootApplication` class:

```
@EnableCircuitBreaker
@SpringBootApplication
public class SpringbootApplication {
```

The Hystrix EIP from the Camel route is automatically integrated with Spring Cloud Hystrix. When running the microservice, the Hystrix stream is available under the `/hystrix.stream` endpoint, which maps to the following URL: http://localhost:8181/hystrix.stream.

You're now ready to run the microservice. First you run the inventory service that the rules microservice communicates with:

```
cd chapter7/prototype2/inventory2
mvn compile exec:java
```

From another shell, you run the rules microservice:

```
cd chapter7/prototype2/rules2-springboot-hystrixdashboard
mvn spring-boot:run
```

Then you test that the microservice runs as expected by calling the rules service directly from the following URL: http://localhost:8181/api/rules/999.

Everything looks okay, and you're ready to have fun and install the Hystrix dashboard.

Installing the Hystrix Dashboard

The dashboard can be installed by downloading the standalone JAR from Bintray: https://bintray.com/kennedyoliveira/maven/standalone-hystrix-dashboard/1.5.6.

After the download, you run the dashboard with the following command:

```
java -jar standalone-hystrix-dashboard-1.5.6-all.jar
```

Then from a web browser, open the URL:

```
http://localhost:7979/hystrix-dashboard
```

On the dashboard, type `http://localhost:8181/hystrix.stream` in the Hostname field and click the Add Stream button. After adding the stream, you click the Monitor Streams button—and behold, there are graphics.

TIP WildFly Swarm has also made it easy to install the Hystrix dashboard as a .war file on WildFly. You can find more information at http://wildfly-swarm.io/tutorial/hystrix/.

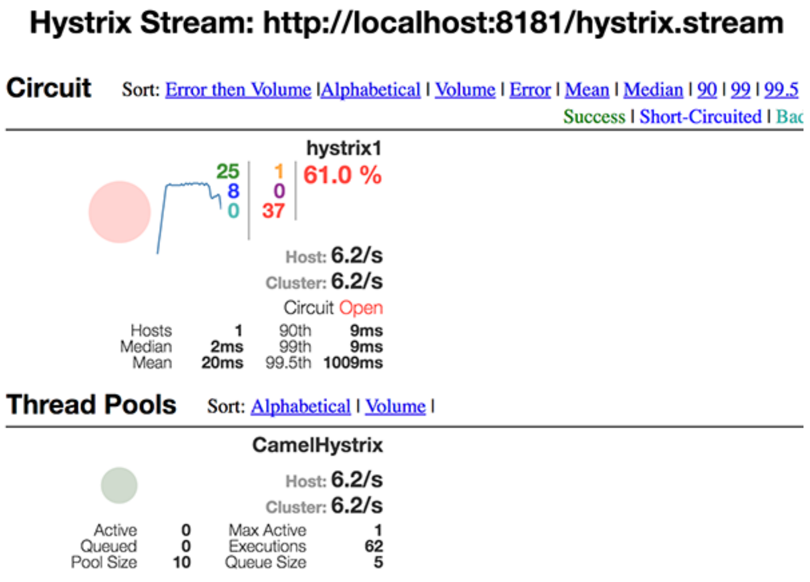
Because there are no activities, the graph isn't exciting—it's all just zeros and a flat line. So let's turn up the load. To do that, you hack up a little bash script that calls the rules service repetitively 10 times per second. You run the script as follows:

```
cd chapter7/prototype2/rules2-springboot-hystrixdashboard
chmod +x hitme.sh
./hitme.sh
```

And now there's activity in the dashboard.

NOTE Understanding what all the numbers on the Hystrix dashboard represent takes time to learn. You can find information on the Hystrix website (https://github.com/Netflix/Hystrix/wiki/Dashboard) that explains the dashboard in much more detail.

After the script has been running for a while, it's all green and happy, so you go out on a limb and stop the inventory microservice to see how the dashboard reacts to failures. And after a little while, you can see the error numbers go up, as illustrated in figure 7.10.



```
Hystrix Stream: http://localhost:8181/hystrix.stream

Circuit      Sort: Error then Volume |Alphabetical | Volume | Error | Mean | Median | 90 | 99 | 99.5
                                              Success | Short-Circuited | Bac

                               hystrix1
                    25 |  1 | 61.0 %
                     8 |  0 |
                     0 | 37 |

                             Host: 6.2/s
                          Cluster: 6.2/s
                            Circuit Open
   Hosts      1     90th     9ms
  Median    2ms     99th     9ms
    Mean   20ms   99.5th  1009ms

Thread Pools   Sort: Alphabetical | Volume |

                           CamelHystrix
                             Host: 6.2/s
                          Cluster: 6.2/s
    Active    0   Max Active     1
    Queued    0   Executions    62
 Pool Size   10   Queue Size     5
```

Figure 7.10 Hystrix dashboard visualizing the circuit breaker running in the rules microservice. The state of the breaker is open because of the recent number of errors as the downstream inventory service has been stopped.

Later you start the inventory service again and see the changes in the dashboard. The red numbers go down, and the green numbers up, and the state of the breaker changes from red to green as well.

The weekend is coming to an end, and tomorrow morning the family is returning. It's late in the evening, and you've done well. You pour yourself a neat glass of whisky and sit to reflect on what you've achieved this time.

What you have learned this time

You've learned that in a microservices and distributed architecture, it's even more important to design for errors. With so many individual services that communicate across the network in a mesh, errors are destined to happen. Luckily, with Camel you can easily handle errors using Camel's error handler or Hystrix circuit breakers.

Although circuit breakers are easy to use with Camel or the `@HystrixCommand`, they're not a silver-bullet solution for everything. The mantra about using the right tool for the right job applies. But circuit breakers are a great solution to prevent cascading errors through an entire system or apply loads on downstream systems under stress. An important setting is to apply sensitive time-outs that match your use cases. Hystrix comes with a low, one-second time setting by default, which works best for Netflix, but likely not for your organization. You certainly also learned that with circuit breakers scattered all around your individual microservices, it's important to have good monitoring of their states. For that, you can use the Hystrix dashboard.

On the flip side, you also learned that managing and running up to six microservices on a single host/laptop is cumbersome. What you need is some kind of platform or orchestration system to manage all your running microservices. And, yeah, that's a big and popular topic these days, which we cover in chapter 18. At first you need to master many other aspects of Camel.

The weekend is over and so is this chapter. Thanks for sticking with us all the way.

7.5 Summary and best practices

Getting to the end of this chapter was a long and bumpy ride. We hope you enjoyed the journey. Writing it was surely one of the most challenging of all the chapters, but we hope it was worth all our extra effort (and little delay in getting the book done).

Don't get too caught up in the microservices buzzword bingo. Microservices is a great concept, but it's not a universal super solution to every software problem. Don't panic, because numerous internet startups have been successful with microservices and are able to do all the jazz and deploy to production a hundred times per day. For regular enterprises, we're likely going to see a middle ground with an enterprise microservice that works the best for them.

What we've been focusing on in this book and this chapter is to cover how you can do microservice development with Camel. At the start of the chapter in section 7.1, we outlined six microservices characteristics that we believe are most applicable for Camel developers. We encourage you to take a second look at these six now that you've reached the end of this chapter. Add a bookmark and circle back to that section from time to time. We want you to have those concepts in your toolbelt.

As usual, we have put together a bulleted list with best practices and advice:

- *Microservices is more than technology*—You can't just build smaller applications and use REST as a communication protocol and think you're doing microservices. There's a lot more to master on both technical and nontechnical levels. The biggest challenges in organizations that want to move to doing microservices include potential friction from organizational structure, team communication, and old habits.
- *Small in size*—A good rule of thumb is the Jeff Bezos's two-pizza rule (where two pizzas should be enough to feed everybody). A microservice should be small enough that a single person could have its complete overview in their head.
- *Design for failures*—Microservices are inherently a distributed architecture. Every time you have remote networks, communication failures can happen. Design your microservices to be fault-tolerant. Camel offers great capabilities in this matter with its error handler and first-class support for Hystrix circuit breakers.
- *Observable*—With an increasing number of running microservices, it becomes important to be able to manage and monitor these services in your infrastructure. Build your microservices with monitoring capabilities such as log aggregation to centralized logging. Chapter 16 covers monitoring and management.
- *Configurable*—Design your microservices to be highly configurable. This becomes an important requirement with container-based platforms, where you deploy and run your microservices as immutable containers, which means any kind of configuration of such containers must be provided externally.
- *Testable*—Microservices should be quick to test, and at best, testing should be automated as part of a continuous integration (CI)/continuous delivery (CD) pipeline. Chapter 9 covers testing.

The journey of microservices will continue in chapter 18, where we talk about microservices in the cloud.

The next chapter takes you through developing Camel projects. You may think you already have the skills to build Camel projects, and you're surely correct. But there's more to Camel than at first sight. Without giving away too many details, chapter 8 helps new users who are less familiar with Maven and Eclipse—to ensure that anyone can set up a development environment for building Camel projects. You'll also learn how to build your own Camel components, data formats, and more. And last but not least, you'll learn an important topic: how to debug your Camel routes.