10

RESTful web services

This chapter covers

- Understanding RESTful web services fundamentals
- Building RESTful web services with JAX-RS and Apache CXF
- Building RESTful web services with camel-restlet and camel-cxf components
- Using Camel's Rest DSL
- Using Rest DSL with various Camel components
- Binding between POJOs and XML/JSON
- Documenting your RESTful services by using Swagger
- Browsing the Swagger API via the Swagger UI

For over a decade, web services have played an important role in integrating loose and disparate systems together. As a natural evolution of humankind and the IT industry, we learn and get smarter. Recently, RESTful web services (a.k.a. REST services) have taken over as the first choice for integrating services. This chapter covers a lot of ground about developing and using RESTful web services with Camel.

Section 10.1 covers the fundamentals and principles of RESTful services, so you're ready to tackle the content of this chapter. We provide a lot of examples with source code that focus on a simple use-case that could happen in the real world, if Rider Auto Parts were an actual company. The section continues by explaining how to implement REST services using various REST frameworks and Camel components.

Section 10.2 introduces you to the Rest DSL, which allows you to define REST services *the Camel way*. You can use the natural verbs from REST together with the existing Camel routing DSL, combining both worlds.

Section 10.3 ends our REST trilogy by covering how to document your REST services by using Swagger APIs. This allows consumers to easily obtain contracts of your services.

USING SOAP WEB SERVICES WITH CAMEL

If you're looking for information about using SOAP web services with Apache Camel, this was covered in the first edition of this book in chapter 7 (section 7.4). You can find up-to-date source code examples of using SOAP web services in the chapter10/code-first and chapter10/contract-first directories that comes with this second edition of the book.

Okay, are you ready? Let's start with the new and existing stuff around RESTful services.

10.1 RESTful services

RESTful services, also known as *REST services*, has become a popular architectural style used in modern enterprise projects. REST was defined by Roy Fielding in 2000 when he published his paper, and today REST is a foundation that drives the modern APIs on the web. You can also think of it as a modern web service, in which the APIs are RESTful and HTTP based so they're easily consumable on the web. This section uses an example from Rider Auto Parts to explain the principles of a RESTful API and then moves on to cover implementing this service by using the following REST frameworks and Camel components:

- *Apache CXF*—Using only CXF to build REST services
- *Apache CXF with Camel*—Same with Camel included
- *Camel Restlet component*—Using one of the simplest Camel REST components
- *Camel CXF REST component*—Using Camel with the CXF REST component

Before we start the show, let's cover the basic principles of a RESTful service.

10.1.1 Understanding the principles of a RESTful API

At Rider Auto Parts, you've recently developed a web service that allows customers to retrieve information about their orders. The web service was previously implemented using old-school SOAP web services. Because you're a devoted developer, and because the company offered limited resources, you had to take matters into your own hands, and over a weekend you sketched the RESTful API design detailed in table 10.1.

Table 10.1 The RESTful API for the Rider Auto Parts order web service

Verb	http://rider.com/orders	http://rider.com/orders/{id}
`GET`	Retrieves a list of all the orders	Retrieves the order with the given ID
`PUT`	N/A	Updates the order with the given ID
`POST`	Creates a new order	N/A
`DELETE`	Cancels all the orders	Cancels the order with the given ID

The Rider Auto Parts order web service offers an API over HTTP at the base path http://rider.com/orders. REST services reuse the same HTTP verbs that are already well understood and established from HTTP. The `GET` verb is like a SQL query function to access and return data. The `PUT` and `POST` verbs are similar to SQL `UPDATE` or `INSERT` functions that create or update data. And finally, the `DELETE` verb is for deleting data.

To allow legacy systems to access the service, you decide to support both XML and JSON as the data representation of the service.

We begin our REST journey with JAX-RS, which is a standard Java API for developing REST services.

10.1.2 Using JAX-RS with REST services

Java API for RESTful Web Services (JAX-RS) is a Java standard API that provides support for creating web services. Numerous REST frameworks implement the JAX-RS specification, such as Apache CXF, JBoss RESTEasy, Restlet, and Jersey. JAX-RS is primarily developed with the help of a set of

Java annotations that you use in Java classes representing the web services.

Before we dive into JAX-RS, let's simplify the use case from Rider Auto Parts by removing two of the services from table 10.1 that aren't necessary. This leaves the four services listed in table 10.2.

Table 10.2 The first set of the RESTful API for the Rider Auto Parts order web service

Verb	http://rider.com/orders	http://rider.com/orders/{id}
`GET`	N/A	Retrieves the order with the given ID
`PUT`	N/A	Updates the order with the given ID
`POST`	Creates a new order	N/A
`DELETE`	N/A	Cancels the order with the given ID

You can then map the services from table 10.2 to a Java interface:

```
public interface OrderService {

    Order getOrder(int orderId);

    void updateOrder(Order order);

    String createOrder(Order order);

    void cancelOrder(int orderId);
}
```

And you create a Java model class to represent the order details:

```
@XmlRootElement(name = "order")
@XmlAccessorType(XmlAccessType.FIELD)
```

```
public class Order {
    @XmlElement
    private int id;
    @XmlElement
    private String partName;
    @XmlElement
    private int amount;
    @XmlElement
    private String customerName;

    // getter/setter omitted
}
```

The model class is annotated with JAXB annotations, making message translation between Java and XML/JSON much easier, because frameworks such as JAXB and Jackson know how to map payloads between XML/JSON and the Java model, and vice versa.

The actual REST service is defined in a JAX-RS resource class implementation, as shown in the following listing.

Listing 10.1 JAX-RS implementation of the Rider Auto Parts REST service

```
import javax.ws.rs.DELETE;   ❶
```

❶

Java imports the JAX-RS annotations

```
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Response;

@Path("/orders/")   ❷
```

2

Defines the entry path of the RESTful web service

```
@Produces("application/xml")
public class RestOrderService {

    private OrderService orderService;

    public void setOrderService(OrderService orderService) {    3
```

3

To inject the OrderService implementation

```
        this.orderService = orderService;
    }

    @GET
    @Path("/{id}")
    public Response getOrder(@PathParam("id") int orderId) {    4
```

4

GET order by ID

```
        Order order = orderService.getOrder(orderId);
        if (order != null) {
            return Response.ok(order).build();    5
```

5

The Response builder to return the order model in the response

```
        } else {
            return Response.status(Response.Status.NOT_FOUND).build();
        }
    }
    @PUT
    public Response updateOrder(Order order) {    ❻
```

❻ PUT to update an existing order

```
        orderService.updateOrder(order);
        return Response.ok().build();
    }

    @POST
    public Response createOrder(Order order) {    ❼
```

❼ POST to create a new order

```
        String id = orderService.createOrder(order);
        return Response.ok(id).build();
    }

    @DELETE
    @Path("/{id}")
    public Response cancelOrder(@PathParam("id") int orderId) {    ❽
```

❽ DELETE to cancel an existing order

```
        orderService.cancelOrder(orderId);
        return Response.ok().build();
    }
}
```

At first, the class imports JAX-RS annotations ❶. A class is defined as a JAX-RS resource class by annotating the class with `@Path` ❷. The `@Path` annotation on the class level defines the entry of the context-path the service is using. In this example, all the REST services in the class must have their context-path begin with `/orders/`. The class is also annotated with `@Produces("application/xml")` ❷, which declares that the service outputs the response as XML. You'll later see how to use JSON or support both.

Because you want to separate the JAX-RS implementation from your business logic, you use dependency injection to inject the `OrderService` instance ❸ that holds the business logic.

The service that gets orders by ID is annotated with `@Path("/{id}")` ❹, as well as in the method signature using `@PathParam ("id")`. The parameter is mapped to the context-path with the given key. If a client calls the REST service with `/orders/123`, for example, then 123 is resolved as the ID path parameter, which will be mapped to the corresponding method parameter.

If the service returns an order, notice that the response builder includes the order instance when the `ok` response is being created ❺. And, likewise, if no order could be found, the response builder is used to create an HTTP 404 (Resource Not Found) error response.

The two services to update ❻ and create ❼ an order are just two lines of code. But notice that these two methods use the `Order` type as a parameter. And because you've annotated the `Order` type with JAXB annotations, Apache CXF will automatically map the incoming XML payload to the `Order` POJO class.

The last method to cancel an order by ID ❽ uses similar path parameter mapping as the get orders by ID service.

The book's source code contains this example in the chapter10/cxf-rest directory, and you can try the example by using the following Maven goal:

```
mvn compile exec:java
```

The example includes two dummy orders, which makes it easier to try the example. For example, to get the first order, you can open the following URL from a web browser:

```
http://localhost:9000/orders/1
```

The second order, not surprisingly, has the order ID of 2, so you can get it using the following:

```
http://localhost:9000/orders/2
```

TESTING REST SERVICES FROM A WEB BROWSER

Testing REST services from a web browser is easy only when testing `GET` services, as those can be accessed by typing the URL in the address bar. But to try `POST`, `PUT`, `DELETE`, and other REST verbs is much harder. Therefore, you can install third-party plugins that provide a REST client. Google Chrome, for example, has the popular Postman plugin.

Using JSON instead of XML

Apache CXF allows you to plug in various binding providers, so you can plug in Jackson for JSON support. To do this, you need to add the following two Maven dependencies to the pom.xml file:

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-rs-extension-providers</artifactId>
  <version>3.2.1 </version>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.jaxrs</groupId>
  <artifactId>jackson-jaxrs-json-provider</artifactId>
  <version>2.8.10</version>
</dependency>
```

Sadly, CXF doesn't support autodiscovering the providers from the classpath, so you need to enlist the Jackson provider on the CXF RS server:

```
JAXRSServerFactoryBean sf = new JAXRSServerFactoryBean();
sf.setProvider(JacksonJsonProvider.class);
```

The chapter10/cxf-rest-json directory in the book's source code contains this example, which you can try using the following Maven goal:

```
mvn compile exec:java
```

APACHE CXF VS. JERSEY

Both Apache CXF and Jersey are web frameworks that support RESTful web services. Apache Camel had first-class integration with CXF for a long time, provided by the camel-cxf component. Numerous people working on Apache Camel are also CXF committers. At the time of this writing, Camel has no Jersey component, and there are no such plans. Apache CXF is also a sister Apache project and therefore recommended to use over Jersey.

In this section, we've covered using Apache CXF without Camel. Because this is a book about Apache Camel, it's time to get back on the saddle and ride. Let's see how to add Camel to the Rider Auto Parts order service application in the sections to follow. But first, we'll summarize the pros and cons of using a pure Apache CXF model for REST services in table 10.3.

Table 10.3 **Pros and cons of using a pure CXF-RS approach for REST services**

Pros	Cons
Uses the JAX-RS standard.	Not possible to define REST services without having to write Java code.
Apache CXF-RS is a well-established REST library.	Camel isn't integrated first class.
The REST service allows developers full control of what happens as they can change the source code.	Configuring CXF-RS can be nontrivial. There's no *CXF in Action* book.

Okay, let's try to add Camel to the example.

10.1.3 Using Camel in an existing JAX-RS application

You managed to develop the Rider Auto Parts order application by using JAX-RS with CXF. But you know that down the road, changes are in the pipeline that require integration with other systems. For that, you want Camel to play its part. How can you develop REST services by using the JAX-RS standard and still use Camel? One way is to let CXF integrate with Camel.

Using Camel from any existing Java application can be done in many ways. Possibly one of the easiest is to use Camel's `ProducerTemplate` from the Java application to send messages to Camel. The following listing shows the modified JAX-RS service implementation that uses Camel.

Listing 10.2 Modified JAX-RS REST service that uses Camel's `ProducerTemplate`

```
@Path("/orders/")
@Produces("application/json")
public class RestOrderService {

    private ProducerTemplate producer;

    public void setProducerTemplate(ProducerTemplate producerTemplate) {    (1)
```

(1)
To inject Camel's ProducerTemplate

```
        this.producer = producerTemplate;
    }

    @GET
    @Path("/{id}")
    public Response getOrder(@PathParam("id") int orderId) {
        Order order = producer.requestBody("direct:getOrder",
                                orderId, Order.class);    (2)
```

(2) Calls Camel route to get the order by ID

```
        if (order != null) {
            return Response.ok(order).build();
        } else {
            return Response.status(Response.Status.NOT_FOUND).build();
        }
    }

    @PUT
    public Response updateOrder(Order order) {
        producer.sendBody("direct:updateOrder", order);    (3)
```

(3) Calls Camel route to update order

```
        return Response.ok().build();
    }

    @POST
    public Response createOrder(Order order) {
        String id = producer.requestBody("direct:createOrder",
                                order, String.class);    (4)
```

(4) Calls Camel route to create order

```
        return Response.ok(id).build();
    }

    @DELETE
    @Path("/{id}")
    public Response cancelOrder(@PathParam("id") int orderId) {
        producer.sendBody("direct:cancelOrder", orderId);    (5)
```

❺

Calls Camel route to cancel order

```
        return Response.ok().build();
    }
}
```

This code looks similar to the code in listing 10.1, but differs in that Camel's `ProducerTemplate` ❶ will be used to route messages from the REST web service to a Camel route (❷ ❸ ❹ ❺). The Camel route is responsible for routing the message to the Rider Auto Parts back-end system, giving access to the order details.

Because you have both CXF and Camel independently operating in the same application, you need to set them individually and configure them in the right order. To keep things simple, you decide to run the application as a standalone Java application with a single main class to bootstrap the application, as shown in the following listing.

Listing 10.3 Running CXF REST web service with Camel integrated from `Main` class

```
public class RestOrderServer {

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

❶

Camel route that accesses the back-end order system

```
        route.setOrderService(new BackendOrderService());

    CamelContext camel = new DefaultCamelContext();    ❷
```

❷

Creates CamelContext and adds the route

```
        camel.addRoutes(route);

        ProducerTemplate producer = camel.createProducerTemplate(); (3)
```

(3) Creates ProducerTemplate

```
        RestOrderService rest = new RestOrderService();
        rest.setProducerTemplate(producer); (4)
```

(4) Injects ProducerTemplate to REST web service

```
        JAXRSServerFactoryBean sf = new JAXRSServerFactoryBean(); (5)
```

(5) Sets up CXF REST web service

```
        sf.setResourceClasses(RestOrderService.class); (5)
        sf.setResourceProvider(RestOrderService.class, (5)
                    new SingletonResourceProvider(rest)); (5)
        sf.setProvider(JacksonJsonProvider.class); (6)
```

(6) Uses Jackson for JSON support

```
        sf.setAddress("http://localhost:9000/"); (7)
```

(7) The URL for published REST web service

```
        Server server = sf.create();

        camel.start();  ❽
```

❽

Starts Camel and CXF

```
        server.start();

        // code that keeps the JVM running omitted  ❾
```

❾

Keeps the JVM running until it's stopped

```

        camel.stop();  ❿
```

❿

Stops Camel and CXF

```
        server.stop();
    }
}
```

The first part of the application is to set up Camel by creating the route ❶ used to call the back-end system. Then Camel itself is created with `CamelContext` ❷, where the route is added. To integrate Camel with CXF, you create `ProducerTemplate` ❸ that gets injected into the REST resource class ❹. After setting up Camel, it's CXF's turn. To use a JAX-RS server in CXF, you create and configure it by using `JAXRSServerFactoryBean` ❺. Support for JSON is provided by adding the Jackson provider ❻. Then the URL for the entry point for clients to call the service is configured ❼ as the last part of the configuration. The application is started by starting Camel and CXF in that order ❽. To keep the application going, some code keeps the JVM running until it’s signaled to stop ❾. To let the application stop gracefully, CXF and Camel are stopped ❿ before the JVM terminates.

TIP You can also set up CXF REST web services by using an XML configuration, which we cover later in this chapter.

This example is shipped as part of the book’s source code in the chapter10/cxf-rest-camel directory; you can run the example by using the following Maven goal:

```
mvn compile exec:java
```

Combining CXF with Camel gives you the best of two worlds. You can define and code the JAX-RS REST services by using the JAX-RS standard in Java code, and still integrate Camel by using a simple dependency-injection technique to inject `ProducerTemplate` or a Camel proxy. But it comes with the cost of having to write this in Java code, and manually having to control the lifecycle of both CXF and Camel. These pros and cons are summarized in table 10.4.

Table 10.4 Pros and cons of using a CXF-RS with Camel approach

Pros	Cons
Uses the JAX-RS standard.	Not possible to define REST services without having to write Java code.
Apache CXF-RS is a well-established REST library.	Configuring CXF-RS can be nontrivial.
The REST service allows developers full control of what happens as they can change the source code.	Need to set up both CXF and Camel lifecycle separately.
Camel is integrated using dependency injection into the JAX-RS resource class.	There's no *CXF in Action* book.

Later you'll see how to use CXF REST services from a pure Camel approach, in which Camel handles the lifecycle of CXF by using the camel-cxf component. We'll leave CXF for a moment and look at alternative REST libraries you can use. For example, you'll learn how the service can be implemented without JAX-RS, and using only Camel with the camel-restlet component.

10.1.4 Using camel-restlet with REST services

The camel-restlet component is one of the easiest components to use for hosting REST services as Camel routes. The component has also been part of Apache Camel for a long time (it was included at the end of 2008). Let's see it in action.

To implement the services from table 10.2 as REST services with camel-restlet, all you need to do is define each service as a Camel route and call the order service bean that performs the action, as shown in the following listing.

Listing 10.4 Camel route using camel-restlet to define four REST services

```
public class OrderRoute extends RouteBuilder {

  public void configure() throws Exception {
    from("restlet:http://0.0.0.0:8080/orders?restletMethod=POST")     (1)
```

(1) REST service to create order

```
        .bean("orderService", "createOrder");

    from("restlet:http://0.0.0.0:8080/orders/{id}?restletMethod=GET")     (2)
```

(2) REST service to get order by ID

```
        .bean("orderService", "getOrder(${header.id})");

    from("restlet:http://0.0.0.0:8080/orders?restletMethod=PUT")     (3)
```

(3) REST service to update order

```
        .bean("orderService", "updateOrder");

    from("restlet:http://0.0.0.0:8080/orders/{id}?restletMethod=DELETE")     (4)
```

(4) REST service to cancel order by ID

```
        .bean("orderService", "cancelOrder(${header.id})");
  }
}
```

These REST web services are defined with a route per REST operation, so there are four routes in total. Notice that the restlet component uses the `restletMethod` option to specify the HTTP verb in use, such as `POST` ❶, `GET` ❷, `PUT` ❸, and `DELETE` ❹. The restlet component allows you to map dynamic values from the context-path by using the `{ }` style, which you use to map the ID from context-path to a header on the Camel message, as illustrated in figure 10.1.

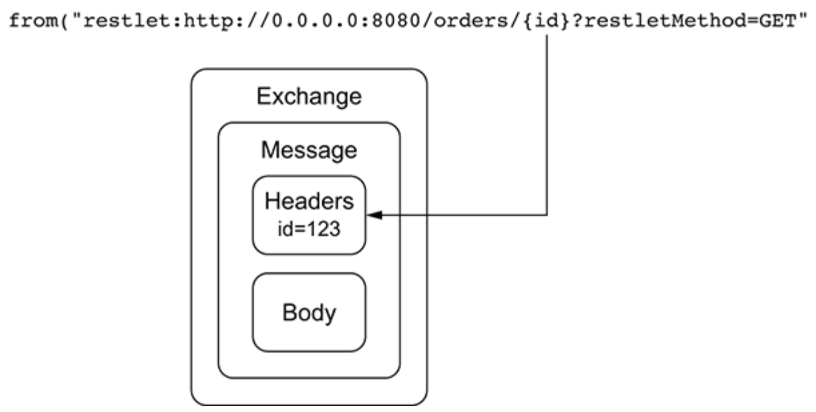

from("restlet:http://0.0.0.0:8080/orders/{id}?restletMethod=GET"
Exchange
Message
Headers
id=123
Body

Figure 10.1 Mapping of dynamic values from context-path to Camel message headers with the Camel Restlet component

In figure 10.1, the REST web service to get the order by ID from listing 10.4 is shown at the top. Each dynamic value in the context-path declared by `{key}` is mapped to a corresponding Camel message header when Camel routes the incoming REST call.

It's important to not mix this syntax with Camel's property placeholder. The mapping syntax uses a single `{ }` pair, whereas Camel's property placeholder uses double `{{ }}` pairs.

REST DSL VS. EIP DSL

The routes in listing 10.4 use the normal Java DSL syntax: routes begin with `from` followed by `bean` or `to` with the EIP verbs. Later in this chapter, you'll learn about the Rest DSL that allows you to define REST web services by using REST verbs (`GET`, `POST`, `PUT`, and so on) instead of the EIP (`from`, `to`, `bean`, and so on) verbs.

You can try this example, which is available in the book's source code in the chapter10/restlet directory, by using the following Maven goals:

```
mvn test -Dtest=OrderServiceTest
mvn test -Dtest=SpringOrderServiceTest
```

The restlet component performs no automatic binding between XML or JSON to Java objects, so you'd need to add camel-jaxb or camel-jackson to the classpath to provide this.

Binding XML to/from POJOs using camel-jaxb

The example in chapter10/restlet supports XML binding by including camel-jaxb as a dependency. By adding camel-jaxb to the classpath, a *fall-back type converter* is added to Camel's type-converter registry. A fallback type converter is capable of converting between multiple types decided at runtime, whereas regular type converters can convert from only one type to another. By using a fallback type converter, Camel is able to determine at runtime whether converting to/from a POJO is possible if the POJO has been annotated with JAXB annotations.

When you want to use JSON, you need to use camel-jackson.

Binding JSON to/from POJOs using camel-jackson

What camel-jaxb does for converting between XML and POJOs is similar to what camel-jackson does. But a few extra steps are required when using camel-jackson:

- Enable camel-jackson as a fallback type converter
- Optionally annotate the POJOs with JAXB annotations
- Optionally annotate the POJOs with Jackson annotations

Both camel-jaxb and camel-jackson can use the same set of JAXB annotations, which allow you to customize the way the mapping to your POJOs should happen. But Jackson can perform mapping to any POJOs if the POJO has plain getter/setters, and the JSON payload matches the names of those getter/setter methods. You can use JAXB to further configure the mapping, and Jackson provides annotations on top of that. You'll find more details on the Jackson website: https://github.com/FasterXML/jackson-annotations/wiki/Jackson-Annotations.

To enable camel-jackson as a fallback type converter, you need to configure properties on `CamelContext`. From Java code, you do as follows:

```
context.getProperties().put("CamelJacksonEnableTypeConverter", "true");
context.getProperties().put("CamelJacksonTypeConverterToPojo", "true");
```

You need to configure this before you start `CamelContext`—for example, from the `RouteBuilder` where you set up your Camel routes. When using XML, you need to configure this inside `<camelContext>`:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <properties>
    <property key="CamelJacksonEnableTypeConverter" value="true"/>
    <property key="CamelJacksonTypeConverterToPojo" value="true"/>
  </properties>
  ...
</camelContext>
```

The accompanying source code includes an example of this in the chapter10/restlet-json directory; you can try the example by using the following Maven goals:

```
mvn test -Dtest=OrderServiceTest
mvn test -Dtest=SpringOrderServiceTest
```

TIP camel-restlet can also be used to call a REST service from a Camel route (for example, as a producer). As an example, see the source code of the chapter10/restlet-json example, where the unit test uses `ProducerTemplate` with a camel-restlet URI to call the REST web service.

The camel-restlet component was one of the first Camel components to integrate with REST. The component is easy to use and requires little configuration to get going. The pros/cons of using camel-restlet have been summarized in table 10.5.

Table 10.5 Pros and cons of using camel-restlet

Pros	Cons
Easy to get started.	Doesn't use the JAX-RS standard.
Restlet is an established REST library.	camel-restlet hasn't been integrated with Google Gson or Jackson to make using JSON easy.

Let's look at one more REST library before ending this section. We'll circle back to Apache CXF and take a look at the Camel CXF-RS component.

10.1.5 Using camel-cxf with REST services

The camel-cxf component includes support for both REST and SOAP web services. As we noted previously, the difference between using plain Apache CXF and the camel-cxf component is that the latter is Camel *first*, whereas CXF is CXF *first*. By *first*, we mean that either CXF or Camel is the primary driver behind how the wheel is spinning.

When using camel-cxf, you need to configure the CXF REST server, which can be done in two ways:

- Camel-configured endpoint
- CXF-configured bean

Camel-configured endpoint

Using a Camel-configured endpoint means that you configure the CXF REST server as a regular Camel endpoint by using URI notation, as shown in the following listing.

Listing 10.5 Camel route using camel-cxf to define four REST services

```
public class OrderRoute extends RouteBuilder {

  public void configure() throws Exception {
    from("cxfrs:http://localhost:8080
           ?resourceClasses=camelinaction.RestOrderService
      &bindingStyle=SimpleConsumer")     1
```

1
Camel-configured endpoint of REST service using CXF-RS

```
    .toD("direct:${header.operationName}");  2
```

2
Calls the route that should perform the selected REST operation

```
  from("direct:createOrder")  3
```

3
REST service to create order

```
        .bean("orderService", "createOrder");

  from("direct:getOrder")  4
```

4
REST service to get order by ID

```
        .bean("orderService", "getOrder(${header.id})");

  from("direct:updateOrder")  5
```

5
REST service to update order

```
        .bean("orderService", "updateOrder");

  from("direct:cancelOrder")  6
```

❻

REST service to cancel order by ID

```
        .bean("orderService", "cancelOrder(${header.id})");
    }
  }
```

When using camel-cxf, the REST web service is defined in a single route ❶ where you configure the hostname and port the REST server uses. When using camel-cxf, it's recommended to use the `SimpleConsumer` binding style, which binds to the natural JAX-RS types and Camel headers. The default binding style uses the `org.apache.cxf.message.MessageContentsList` type from CXF, which is a lower-level type like Camel's `Exchange`. The last configured option is the `resourceClasses` option ❶, which refers to a class or interface that declares the REST web service by using the JAX-RS standard.

USING A CLASS OR INTERFACE FOR THE RESOURCE CLASS

Because CXF uses JAX-RS, the REST web service is defined as a JAX-RS resource class that lists all the REST operations. The resource class code is in listing 10.6. The resource class is, in fact, not a class but an interface; the reason for this is that camel-cxf will use JAX-RS only as a contract to set up REST web services. At runtime, the incoming request is routed in the Camel routes instead of invoking the Java code in the resource class. And because of that, it's more appropriate to use an interface instead of a class with empty methods. Another reason to use an interface over a class is that the class would have all empty methods and not be in use. This may lead developers to become confused about why this class is empty and to think that it could be a mistake or a bug.

The REST web service has four services (as shown in listing 10.6), and the header indicates which operation was called by using the name `operationName`. You use this to route the message via the Dynamic To EIP pattern ❷, whereby you let each operation be a separate route (❸ ❹ ❺ ❻), linked together using the direct component.

The actual REST web service is specified as a JAX-RS resource class, shown in the following listing.

Listing 10.6 JAX-RS interface of the Rider Auto Parts REST service

```
@Path("/orders/")
@Consumes(value = "application/xml,application/json")
@Produces(value = "application/xml,application/json")
public interface RestOrderService {      1
```

1
Using an interface instead of a class

```
    @GET
    @Path("/{id}")
    Order getOrder(@PathParam("id") int orderId);      2
```

2
REST operations

```
    @PUT
    void updateOrder(Order order);      2

    @POST
    String createOrder(Order order);      2

    @DELETE
    @Path("/{id}")
    void cancelOrder(@PathParam("id") int orderId);      2
}
```

In this example, you use an interface ❶, for the reason explained in the preceding sidebar. Each REST operation is declared using standard JAX-RS annotations ❷ that allow you to define the REST web services in a nice, natural way. Notice that the return types in the operations are the model types (such as `Order`, which returns the order details). And the `create-Order` operation returns a `String` with the ID of the created order.

Now compare listing 10.6 with listing 10.1. In listing 10.6, you can see from the interface what the REST service operations return, such as an `Order` type in the `getOrder` operation, and a `String` type in the `createOrder` operation. In listing 10.1, all the REST service operations return the same JAX-RS `Response` type. When doing so, you lose the ability to know exactly what the operation returns.

This problem can be remedied using a third-party API documentation framework such as Swagger. Swagger provides a set of annotations you can use in the JAX-RS resource classes to specify all sorts of details, such as what the REST operations take as input parameters, and what they return. In section 10.3, you'll learn how to use Swagger with Camel.

You can try this example, available in the chapter10/camel-cxf-rest directory of this book's source code, using the following Maven goals:

```
mvn test -Dtest=RestOrderServiceTest
mvn test -Dtest=SpringRestOrderServiceTest
```

You can also configure the CXF REST server by using a bean style, as explained next.

CXF-configured bean

Apache CXF was created many years ago, when the Spring Framework was popular and using Spring was dominated by XML configuration. This affected CXF, as the natural way of configuring CXF is the Spring XML style. The *CXF-configured bean* is such a style.

To use camel-cxf with Spring XML (or Blueprint XML), you need to add the camel-cxf namespace to the XML stanza (highlighted in bold). This allows you to set up the CXF REST server by using the `rsServer` element, as shown in the following listing.

Listing 10.7 Configuring CXF REST server using XML style

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

```
            http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans.xsd
            http://camel.apache.org/schema/cxf
            http://camel.apache.org/schema/cxf/camel-cxf.xsd
            http://camel.apache.org/schema/spring
            http://camel.apache.org/schema/spring/camel-spring.xsd">

  <bean id="orderService" class="camelinaction.DummyOrderService"/>

  <cxf:rsServer id="restOrderServer" address="http://localhost:8080"     1
```

1

Sets up CXF REST server using rsServer element

```
                  serviceClass="camelinaction.RestOrderService">
  </cxf:rsServer>

  <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="cxfrs:bean:restOrderServer?bindingStyle=SimpleConsumer"/>     2
```

Camel route with the CXF-configured bean

```
      <toD uri="direct:${header.operationName}"/>     3
```

Calling the route that should perform the selected REST operation

```
    </route>
    <route>
      <from uri="direct:createOrder"/>
      <bean ref="orderService" method="createOrder"/>
    </route>
    <route>
      <from uri="direct:getOrder"/>
      <bean ref="orderService" method="getOrder(${header.id})"/>
    </route>
    <route>
```

```
      <from uri="direct:updateOrder"/>
      <bean ref="orderService" method="updateOrder"/>
    </route>
    <route>
      <from uri="direct:cancelOrder"/>
      <bean ref="orderService" method="cancelOrder(${header.id})"/>
    </route>
  </camelContext>
  </beans>
```

The CXF REST server is configured in the `rsServer` element ❶ to set up the base URL of the REST service and to refer to the resource class. The `rsServer` allows additional configuration such as logging, security, payload providers, and much more. All this configuration is done within the `rsServer` element and uses standard CXF naming. You can find many examples and details on the Apache CXF website.

The routes within `<camelContext>` are the XML equivalent of the Java-based example in listing 10.5.

This example is provided as source code in the chapter10/camel-cxf-rest directory; you can try the example using the following Maven goal:

```
mvn test -Dtest=SpringRestBeanOrderServiceTest
```

The example uses XML as a payload during testing, but you can also support JSON.

Adding JSON support to camel-cxf

JSON is supported in CXF by different providers. The default JSON provider is Jettison, but today the most popular JSON library is Jackson. To use Jackson, all you have to do is add the following Maven dependencies to your pom.xml file:

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-rs-extension-providers</artifactId>
  <version>3.2.1</version>
</dependency>
<dependency>
```

```
    <groupId>com.fasterxml.jackson.jaxrs</groupId>
    <artifactId>jackson-jaxrs-json-provider</artifactId>
    <version>2.8.10</version>
  </dependency>
```

In the Spring or Blueprint XML file, add the Jackson provider as a bean:

```
<bean id="jsonProvider"
      class="com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider"/>
```

And finally, add the provider to the CXF REST server configuration:

```
<cxf:rsServer id="restOrderServer" address="http://localhost:8080"
              serviceClass="camelinaction.RestOrderService">
  <cxf:providers>
    <ref bean="jsonProvider"/>
  </cxf:providers>
</cxf:rsServer>
```

We provide an example with the book's source code in the chapter10/camel-cxf-rest directory. You can try the example using the following Maven goal:

```
mvn test -Dtest=SpringRestBeanOrderServiceJSonTest
```

USING CDI, APACHE KARAF, OR SPRING BOOT

The example is also available for running on CDI, Karaf, or Spring Boot, which you can find in the chapter10/camel-cxf-rest-cdi, chapter10/camel-cxf-rest-karaf, and chapter10/camel-cxf-rest-spring-boot directories, respectively. Each example has instructions for running it in the accompanying readme file.

Table 10.6 details the pros and cons of using camel-cxf.

Table 10.6 Pros and cons of using camel-cxf for REST services

Pros	Cons
Uses the JAX-RS standard.	Not possible to define REST services without having to write Java code.
Apache CXF-RS is a well-established REST library. CXF-RS is integrated with Camel as first class.	The code in the REST service isn't executed, and developers are allowed full control only of what happens in Camel routes.
	There's no *CXF in Action* book.

That was a lot of coverage on using various REST frameworks with Camel. You saw how Camel bridges to REST services by using Camel components and Camel routes. It's as if the REST and EIP worlds are separated. What if you could take the REST verbs into the EIP world so you could define REST services by using HTTP verbs in the same style as your EIPs in Camel routes? This is the topic of the next section.

10.2 The Camel Rest DSL

Several REST frameworks have emerged and gained huge popularity recently—for example, the Ruby-based web framework called Sinatra, the Node-based Express framework, and from Java the small REST and web framework called Spark Java. What they all share is putting a Rest DSL in front of the end user, making it easy to do REST development.

The story of the Rest DSL all began with conversations between James Strachan and Claus Ibsen, discussing how making REST services with Apache Camel wasn't as straightforward as they'd like it to be. For example, the semantics of the REST and HTTP verbs became *less visible* in the Camel endpoints. What if we could define REST services using a *REST-like* DSL in Camel? And so it began. Claus started hacking on the Rest DSL, and now two years later, he writes about it in this book.

REST DSL DESIGN

The goal of Rest DSL is to make defining REST services at a high abstraction level easy and quick, and to do it *the Camel way*. The Rest DSL is intended to support 95% of use cases. For some advanced use cases, you may have to opt out of the Rest DSL and use an alternative solution, such as using JAX-RS directly with Apache CXF or Jersey. Initially, the Rest DSL supported only exposing RESTful services from Camel (for example, as a consumer) and there was no support for calling existing external RESTful services. But from Camel 2.19 onward, the Rest DSL has preliminary support for calling RESTful services as a producer. This chapter focuses on the consumer side and has only limited coverage of the producer side.

The Rest DSL debuted at end of 2014 as part of Apache Camel 2.14. But it has taken a couple of releases to further harden and improve the Rest DSL, making it a great feature in the Camel toolbox.

Before covering all the ins and outs of the Rest DSL, let's see it in action.

10.2.1 Exploring a quick example of the Rest DSL

The Camel Rest DSL was inspired by the Java Spark REST library, and so camel-spark-rest was the first component that's fully Rest DSL integrated. Before the Rest DSL, camel-restlet probably came closest to using the REST and HTTP verbs in the endpoint URIs. Take a moment to look at the camel-restlet example in listing 10.4 and then compare it to the implementation of a similar solution with the Rest DSL in the following listing. You should notice that the latter *says more* with REST and HTTP terms.

Listing 10.8 Camel route using Rest DSL to define four REST services

```
public class OrderRoute extends RouteBuilder {

  public void configure() throws Exception {

    restConfiguration()
    .component("spark-rest").port(8080);    1
```

❶

Configures Rest DSL to use camel-spark-rest component

```
    rest("/orders") ❷
```

❷

Uses REST verbs to define the four REST services with GET, POST, PUT and DELETE

```
        .get("{id}")
          .to("bean:orderService?method=getOrder(${header.id})")
        .post()
          .to("bean:orderService?method=createOrder")
        .put()
          .to("bean:orderService?method=updateOrder")
        .delete("{id}")
          .to("bean:orderService?method=cancelOrder(${header.id})");
      }
  }
```

When using the Rest DSL, it must be configured first; you need to indicate which REST component to use ❶ and additional configurations such as the context-path and port number. In this example, you haven't configured any context-path, so it uses `/` by default. Then you define a set of REST services by using `rest("/orders")` ❷ followed by each service defined using REST verbs such as `get`, `post`, `put`, and `delete`. The four REST services will at runtime be mapped to the following URLs:

- `GET /orders/{id}`
- `POST /orders`
- `PUT /orders`
- `DELETE /orders/{id}`

Dynamic values in context-path can be mapped by using the `{ }` syntax, which is done in the case of `get` and `delete`.

You can also use the Rest DSL in the XML DSL, as shown in the following listing.

Listing 10.9 Camel route using Rest DSL to define four REST services using XML DSL

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <restConfiguration component="spark-rest" port="8080"/>    1
```

1

Configures Rest DSL to use camel-spark-rest component

```
  <rest path="/orders">    2
```

2

Uses REST verbs to define the four REST services with GET, POST, PUT and DELETE

```
      <get uri="{id}">
        <to uri="bean:orderService?method=getOrder(${header.id})"/>
      </get>
      <post>
        <to uri="bean:orderService?method=createOrder"/>
      </post>
      <put>
        <to uri="bean:orderService?method=updateOrder"/>
      </put>
      <delete uri="{id}">
        <to uri="bean:orderService?method=cancelOrder(${header.id})"/>
      </delete>
    </rest>
</camelContext>
```

As you can see, the Rest DSL in Java or XML DSL is structured in the same way. At first you need to configure the Rest DSL ❶, followed by defining the REST services by using the REST verbs ❷.

You can try this example, provided as part of the source code for the book, by running the following Maven goals in the chapter10/spark-rest directory:

```
mvn test -Dtest=OrderServiceTest
mvn test -Dtest=SpringOrderServiceTest
```

That's it for a quick example of the Rest DSL in action. Let's return to Camel in action and talk about how the Rest DSL works under the covers and which Camel components it supports.

10.2.2 Understanding how the Rest DSL works

In a nutshell, the Rest DSL works as a DSL extension (to the existing Camel routing DSL), using REST and HTTP verbs, that's then mapped to the existing Camel DSL. By mapping to the Camel DSL, it's all just routes from the Camel point of view.

This is better explained with an example, so let's take the `GET` service from listing 10.8

```
<rest path="/orders">
  <get uri="{id}">
    <to uri="bean:orderService?method=getOrder(${header.id})"/>
  </get>
  ...
</rest>
```

and see how it's mapped to the Camel routing DSL:

```
<route>
  <from uri="rest:get:/orders/{id}?componentName=spark-rest"/>
  <to uri="bean:orderService?method=getOrder(${header.id})"/>
  ...
</route>
```

As you can see, this route is just a regular Camel route with a `<from>` and a `<to>`. The meat of the mapping happens between `<get>` and `<from>:`

```
<rest path="/orders">                          "rest:get:/orders/{id}?
  <get uri="{id}">               →              componentName=spark-rest"
```

You can break down the mapping as illustrated in figure 10.2.

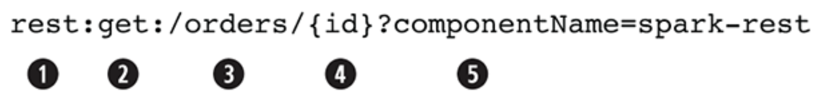


Figure 10.2 Key points of mapping the Rest DSL to five individual parts in the Camel rest endpoint

The key points are as follows:

1. `rest` —Every REST service is mapped to use the generic Camel rest component, which is provided out of the box in camel-core.
2. `get` —The HTTP verb, such as `GET` , `POST` , or `PUT` .

3 `/orders` —The base URL of the REST services, configured in `<rest path="/orders">` .

4 `/{id}` —Additional URI of the REST service, configured in `<get uri="{id}">` .

5. `componentName=spark-rest` —The Camel component to use for hosting the REST services. Additional parameters can be passed on from the REST configuration.

Notice that points 3 and 4 combine the base URL with the URI of the REST service.

The mapping for the four REST services is as follows:

```
rest:get:/orders/{id}?componentName=spark-rest
rest:post:/orders?componentName=spark-rest
rest:put:/orders?componentName=spark-rest
rest:delete:/orders/{id}?componentName=spark-rest
```

You've now established that the Rest DSL is glorified syntax sugar on top of the existing DSL that rewrites the REST services as small Camel routes of `from` → `to` . REST services are just Camel routes, which means all the existing capabilities in Camel are being harnessed. This also means that managing these REST services becomes as easy as managing your existing Camel routes—yes, you can start and stop those REST routes, and so on.

REST DSL = CAMEL ROUTES

To emphasize one more time: the Rest DSL builds regular Camel routes, which are run at runtime. So you have nothing to be afraid of. All you've learned about Camel routes is usable with the Rest DSL.

If the Rest DSL is just Camel routes, what's being used to set up an HTTP server and servicing the requests from clients calling the REST services?

10.2.3 Using supported components for the Rest DSL

When using Rest DSL, you need to pick one of the Rest DSL–capable components that handle the task of hosting the REST services. At the time of this writing, the following components support Rest DSL:

- *camel-jetty*—Using the Jetty HTTP server
- *camel-netty4-http*—Using the Netty library
- *camel-restlet*—Using the Restlet library
- *camel-servlet*—Using Java Servlet for servlets or Java EE servers
- *camel-spark-rest*—Using the Java Spark REST library
- *camel-undertow*—Using the JBoss Undertow HTTP server

You may be surprised that the list contains components that aren't primarily known as REST components. Only camel-restlet and camel-spark-rest are REST libraries; the remainder are typical HTTP components. This is on purpose, as the Rest DSL was designed to be able to sit on top of any HTTP component and just work (famous last words). A prominent component absent from the list is Apache CXF. At the time of this writing, CXF is too tightly coupled to the programming model of JAX-RS, requiring the REST services to be implemented as JAX-RS service classes. This might change in the future, as some Apache Camel and CXF committers want to work on this so CXF can support Camel's Rest DSL.

With plenty of choices in the list of components, you can pick and choose what best suits your use case. For example, you can use camel-servlet if you run your application in a servlet container such as Apache Tomcat or

WildFly. You can also “go serverless” and run your Camel REST applications standalone with Jetty, Restlet, Spark Rest (uses Jetty), Undertow, or the Netty library.

Why do you need to use one of these components with the Rest DSL? Because you don’t want to reinvent the wheel. To host REST services, you need an HTTP server framework, and it’s much better to use an existing one than to build your own inside Camel. The Rest DSL has very little code to deal with HTTP and REST, but leaves that entirely up to the chosen component.

The Rest DSL makes it easy to switch between these components. All you have to do is change the component name in the REST configuration. But that’s not entirely true, as all these components have their own configuration you can use. For example, you’d need to do this to set up security and other advanced options.

10.2.4 Configuring Rest DSL

Configuring Rest DSL can be grouped into the following six categories:

- *Common*—Common options
- *Component*—To configure options on the chosen Camel component to be used
- *Endpoint*—To configure endpoint-specific options from the chosen component on the endpoint that are used to create the consumer in the Camel route hosting the REST service
- *Consumer*—To configure consumer-specific options in the Camel route that hosts the REST service
- *Data format*—To configure data-format-specific options that are used for XML and JSON binding
- *CORS headers*—To configure CORS headers to include in the HTTP responses

We cover the first four categories next. The data format category is explained in section 10.2.5, and the CORS headers category is covered in section 10.3 as part of API documentation of your REST services using Swagger.

Configuring common options

The common options are listed in table 10.7.

Table 10.7 Common options for the Rest DSL

Option	Default	Description
`component`		*Mandatory*. The Camel component to use as the HTTP server. The chosen name should be one of the supported components listed in section 10.2.3.
`scheme`	`http`	Whether to use HTTP or HTTPS.
`hostname`		The hostname for the HTTP server. If not specified, the hostname is resolved according to the `restHostNameResolver`.
`port`		The port number to use for the HTTP server. If you use the servlet component, the port number isn't in use, as the servlet container controls the port number. For example, in Apache Tomcat the default port is 8080, and in Apache Karaf / ServiceMix, it's 8181.
`contextPath`		Configures a base context-path that the HTTP server will use.
`restHostNameResolver`		Resolves the hostname to use, if no hostname has been explicitly configured. You can specify `localHostName` or

Option	Default	Description
		`localIp` to use either the hostname or IP address.

All the REST configuration uses `restConfiguration` or `<restConfiguration>`, as shown here in Java:

```
restConfiguration()
  .component("spark-rest").contextPath("/myservices").port(8080);
```

And in XML:

```
<restConfiguration component="spark-rest"
                   contextPath="/myservices" port="8080"/>
```

You can try this yourself. For example, let's change the example in chapter10/spark-rest to use another component such as camel-undertow. All you have to do is change the dependency in pom.xml from camel-spark-rest to camel-undertow and change the component name to undertow as shown here:

Here's the changed pom.xml file:

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

And here's the change in the `OrderRoute` Java source code:

```
restConfiguration()
  .component("undertow").port(8080);
```

Also remember to change the component in the SpringOrderServiceTest.xml file:

```
<restConfiguration component="undertow" port="8080"/>
```

Then run the example by using the following Maven goal in the chapter10/spark-rest directory:

```
mvn clean test
```

When using Rest DSL, it uses the default settings of the chosen Camel component. You can configure this on three levels: the component, the endpoint, and the consumer. We'll now take a look at how to do that.

Configuring component, endpoint, and consumer options in Rest DSL

Camel components often offer many options you can tweak to your needs. We covered using components in chapter 6, which explains how configuration works. The Rest DSL allows you to configure the component directly in the REST configuration section on three levels:

- Component level
- Endpoint level
- Consumer level

The component level isn't used as often as the endpoint level. As you should know this far into the book, Camel relies heavily on endpoints that you configure as URI parameters. Therefore, you'll find most of the options at this level. The consumer level is seldom in use, as consumers are configured from endpoint options. Table 10.8 shows the option name to use in Rest DSL to refer to these levels.

Table 10.8 Component-related configuration options for the Rest DSL

Option	Description
`componentProperty`	To configure component-specific options. You can use multiple options to specify more than one option.
`endpointProperty`	To configure endpoint-specific options. You can use multiple options to specify more than one option.
`consumerProperty`	To configure consumer-specific options. You can use multiple options to specify more than one option.

A common use-case for the need to configure Rest DSL on components is to set up security or configure thread pool settings. At Rider Auto Parts, the RESTful order application you've been working on has a new requirement for clients to authenticate before they can access the services. Because Jetty supports HTTP basic authentication, you choose to give it a go with Jetty. In addition, the load on the application is expected to be minimal, so you want to reduce the number of threads used by Jetty. All this is configured on the Jetty component level, which is easily done with the Rest DSL, as shown here:

```
restConfiguration()
  .component("jetty").port(8080)
    .componentProperty("minThreads", "1")
    .componentProperty("maxThreads", "8");
```

And in XML DSL:

```
<restConfiguration component="jetty" port="8080">
  <componentProperty key="minThread" value="1"/>
  <componentProperty key="maxThread" value="8"/>
</restConfiguration>
```

Notice how easy that is, by using `componentProperty` to specify the option by key and value. In the preceding example, you lower the Jetty thread pool to use only one to eight worker threads to service incoming calls from clients.

Setting up security with Jetty requires much more work, depending on the nature of the security. Security in Jetty is implemented using special handlers, which in our case is `ConstraintSecurityHandler`. You create and configure this handler in the `JettySecurity` class, as shown in the following listing.

Listing 10.10 Set up Jetty authentication using basic auth

```
public class JettySecurity {

  public static ConstraintSecurityHandler createSecurityHandler() {
    Constraint constraint = new Constraint("BASIC", "customer");    1
```

1

Uses HTTP Basic Auth

```
    constraint.setAuthenticate(true);

    ConstraintMapping mapping = new ConstraintMapping();
    mapping.setConstraint(constraint);
    mapping.setPathSpec("/*");    2
```

2

Applies to all incoming requests matching `/*`

```


    ConstraintSecurityHandler handler = new ConstraintSecurityHandler();
    handler.addConstraintMapping(mapping);
    handler.setAuthenticator(new BasicAuthenticator());
    handler.setLoginService(new HashLoginService("RiderAutoParts",
                      "src/main/resources/users.properties"));    3
```

3

Loads user names from external file

```
      return handler;
    }
  }
```

To use HTTP basic authentication, you specify `BASIC` as a parameter to the created `Constraint` object ❶. Then you tell Jetty to apply the security settings to the incoming requests that match the pattern `/*` ❷, which means all of them. The last part is to put all this together in a handler object that's returned. The known username and passwords are validated using `LoginService`. Rider Auto Parts keeps it simple by using a plain-text file to store the username and passwords ❸—well, at least for this prototype. Later, when you go into production, you'll have to switch to an LDAP-based `LoginModule` instead.

The last piece of this puzzle is to configure Rest DSL to use this security handler, which is done on the endpoint level instead of the component level. The REST configuration is updated as shown here:

```
    restConfiguration()
     .component("jetty").port(8080)
       .componentProperty("minThreads", "1")
       .componentProperty("maxThreads", "8")
       .endpointProperty("handlers", "#securityHandler");
```

And in XML DSL:

```
  <restConfiguration component="jetty" port="8080">
    <componentProperty key="minThread" value="1"/>
    <componentProperty key="maxThread" value="8"/>
    <endpointProperty key="handlers" value="#securityHandler"/>
  </restConfiguration>
```

As you can see, Jetty uses the `handlers` option on the endpoint level to refer to one or more Jetty security handlers. Notice that you use the `#`

prefix in the value to tell Camel to look up the security handler from the Camel registry.

TIP Chapter 4 covers the Camel registries.

All that's left to do is create an instance of the handler by calling the static method `createSecurityHandler` on the `JettySecurity` class. For example, in Spring XML, you can do this:

```
    <bean id="securityHandler" class="camelinaction.JettySecurity"
          factory-method="createSecurityHandler"/>
```

The book's source code contains this example in chapter10/jetty-rest-security, and you can try the example using the following Maven goals:

```
mvn compile exec:java
```

From a web browser, you can access http://localhost:8080/orders/1, which should present a login box. You can pass in `jack` as the username and `123` as the password.

The example also provides unit tests, which you can run with the following Maven goals:

```
mvn test -Dtest=OrderServiceTest
mvn test -Dtest=SpringOrderServiceTest
```

USING CDI OR APACHE KARAF

The example is also available for running on CDI or Karaf, which you can find in the chapter10/jetty-rest-security-cdi and chapter10/jetty-rest-security-karaf directories. Each example has instructions in the accompanying readme file.

When working with REST services, it's common to use data formats in XML or JSON. The following section covers how to use XML and JSON with the Rest DSL.

10.2.5 Using XML and JSON data formats with Rest DSL

The Rest DSL supports automatic binding of XML/JSON content to/from POJOs using Camel data formats. You may want to use binding if you develop POJOs that map to your REST services request and response types. This allows you, as a developer, to work with the POJOs in Java code.

Rest DSL supports the binding modes listed in table 10.9.

Table 10.9 Binding modes supported by Rest DSL

Mode	Description
`off`	Binding is turned off. This is the default mode.
`auto`	Binding is automatically enabled if the necessary data format is available on the classpath. For example, providing camel-jaxb on the classpath enables support for XML binding. Having camel-jackson on the classpath enables support for JSON binding.
`json`	Binding to/from JSON is enabled and requires a JSON-capable data format on the classpath, such as camel-jackson.
`xml`	Binding to/from XML is enabled and requires camel-jaxb on the classpath.
`json_xml`	Binding to/from JSON and XML is enabled and requires both data formats to be on the classpath.

By default, the binding mode is off. No automatic binding occurs for incoming or outgoing messages. Section 10.2.2 covered how the Rest DSL works by mapping the Rest DSL into Camel routes. When binding is enabled, a `RestBindingProcessor` is injected into each Camel route as the first processor in the routing tree. This ensures that any incoming messages can be automatically bound from XML/JSON to POJO. Likewise, when the Camel route is completed, the binding is able to convert the message body back from POJO to XML/JSON.

Figure 10.3 illustrates this principle.

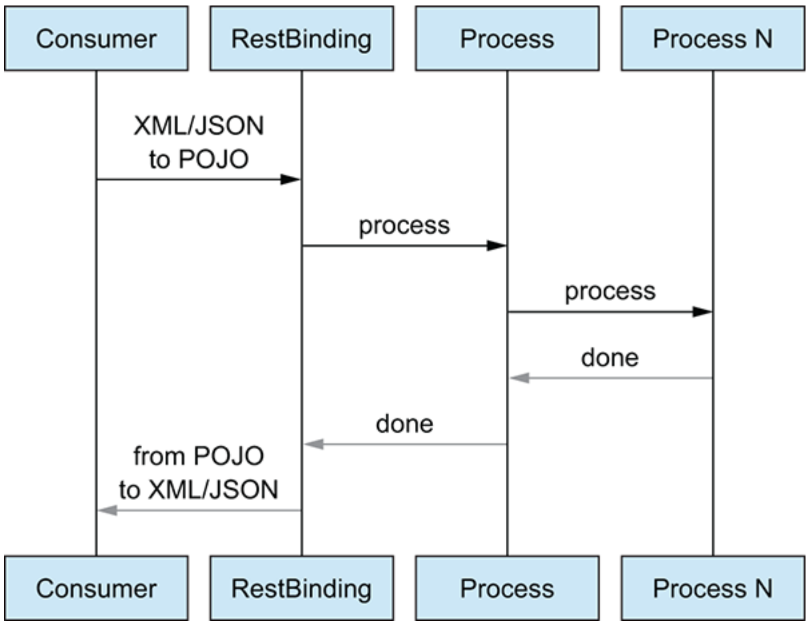


Figure 10.3 `RestBinding` sits right after the consumer to ensure that it's able to bind the incoming messages firsthand from XML/JSON to POJO classes, before the message is routed in the Camel route tree by the processors. When the routing is done, the outgoing message is reversed, from POJO to XML/JSON format, right before the consumer takes over and sends the outgoing message to the client that called the REST service.

The Rest DSL binding is designed to make it easy to support the two most common data formats with REST services, XML and JSON. As a rule of thumb, you configure the binding mode to be XML, JSON, or both, depending on your needs. Then you add the necessary dependencies to the classpath, and you're finished.

How this works, as illustrated in figure 10.3, is achieved from the `RestBinding-Processor` woven into each of the Camel routes that processes the REST services. This happens for any supported REST component (listed in section 10.2.3), and therefore you have binding out of the box.

AVOID DOUBLE BINDING

Some REST libraries, such as Apache CXF and Restlet, provide their own binding support as well. Section 10.1.2 covered using Apache CXF with JAX-RS using JSON. In those use-cases, it's not necessary to enable their binding support as well, because it's provided out of the box with Camel's Rest DSL.

`RestBindingProcessor` is responsible for handling content negotiation of the incoming and outgoing messages, which depends on various factors.

Binding negotiation for incoming and outgoing messages

`RestBindingProcessor` adheres to the following rules (in order of importance) about whether XML/JSON binding to POJO is applied for the *incoming* messages:

1. If binding mode is off, no binding takes place.
2. If the incoming message carries a `Content-Type` header, the value of the header is used to determine whether the incoming message is in XML or JSON format.
3. If the Rest DSL has been configured with a `consumes` option, that information is used to detect whether binding from XML or JSON format is allowed.
4. If you haven't yet determined whether the incoming message is in XML or JSON format, and the binding mode allows both XML and JSON, then the processor will check whether the message payload contains XML content. If the content isn't XML, it's assumed to be JSON. (At the time of writing, the Rest DSL binding supports only XML or JSON.)
5. If the binding mode and the incoming message are incompatible, an exception is thrown.

For *outgoing* messages, the rule set is almost identical:

1. If the binding mode is off, no binding takes place.
2. If the outgoing message carries an `Accept` header, the value of the header is used to determine whether the client accepts the message in either XML or JSON format.
3. If the outgoing message doesn't carry an `Accept` header, the `Content-Type` header is checked to see whether the message body is either in XML or JSON format.
4. If the Rest DSL has been configured with a `produces` option, that information is used to detect whether binding to XML or JSON format is expected.
5. If the binding mode and the outgoing message are incompatible, an exception is thrown.

The rules can also be explained as a way of ensuring that the configured binding mode and the actual incoming or outgoing message *align* so the binding can be carried on.

It’s almost time to see this in action, but first you need to know how to configure the binding.

Configuring Rest DSL Binding

Table 10.10 lists the binding options.

Table 10.10 Binding options for Rest DSL

Option	Default	Description
`bindingMode`	`off`	To enable binding where the Rest DSL can automatically bind the incoming and outgoing payload to XML or JSON format. The possible choices are `off`, `auto`, `json`, `xml`, and `json_xml`. The modes are further detailed in table 10.9.
`skipBindingOnErrorCode`	`true`	Whether to use binding when returning an error as the response. This allows you to build custom error messages that don't bind to JSON/XML, as success messages otherwise will do.
`jsonDataFormat`	`json-jackson`	Name of specific data format to use for JSON binding.
`xmlDataFormat`	`jaxb`	Name of specific XML data format to use for XML binding.
`dataFormatProperties`		Allows you to configure the XML and JSON data format with data-format-specific options. For example,

Option	Default	Description
		when you need to enable JSON pretty-print mode.

The binding configuration is done using `restConfiguration` in Java:

```
restConfiguration()
  .component("spark-rest").port(8080)
  .bindingMode(RestBindingMode.json)
  .dataFormatProperty("prettyPrint", "true");
```

And in XML:

```
<restConfiguration component="spark-rest" port="8080" bindingMode="json"
  <dataFormatProperty key="prettyPrint" value="true"/>
</restConfiguration>
```

Here you've configured the binding mode to be JSON only. And to specify that the JSON output should be in pretty-print mode, you configure this by using the `dataFormatProperty` option.

CONFIGURING DATA FORMAT OPTIONS

The REST configuration uses the option `dataFormatProperty` to configure the XML and JSON data formats. The default XML and JSON data formats are camel-jaxb and camel-jackson. Both happen to provide an option named `prettyPrint` to turn on outputting XML/JSON in pretty-print mode.

When we started covering the Rest DSL in section 10.2.1, the first example used the camel-spark-rest component, and XML was the supported content-type of the REST services. Let's see what it takes to improve this example by using JSON instead, and then thereafter supporting both XML and JSON.

Using JSON binding with Rest DSL

First you need to add the JSON data format by adding the following dependency to the Maven pom.xml file:

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

Then you need to enable JSON binding in the REST configuration, and you also turn on pretty-print mode so the JSON output from Camel is structured in a nice, human-readable style. The following listing shows the code in Java.

Listing 10.11 Using Rest DSL with JSON binding with Java DSL

```
restConfiguration()
  .component("spark-rest").port(8080)
  .bindingMode(RestBindingMode.json) 1
```

1

Using JSON binding mode

```
  .dataFormatProperty("prettyPrint", "true"); 2
```

2

Turns on pretty printing for JSON output

```
rest("/orders")
  .get("{id}").outType(Order.class) 3
```

3

The output type of this REST service is Order class.

```
    .to("bean:orderService?method=getOrder(${header.id})")
  .post().type(Order.class)    ❹
```

❹

The input type of this REST service is Order class.

```
    .to("bean:orderService?method=createOrder")
  .put().type(Order.class)    ❺
```

❺

The input type of this REST service is Order class.

```
    .to("bean:orderService?method=updateOrder")
  .delete("{id}")
    .to("bean:orderService?method=cancelOrder(${header.id})");
```

In the REST configuration, you set up to use the camel-spark-rest component and host the REST services on port 8080. The binding mode is set to JSON ❶, and you want the JSON output to be pretty printed ❷ (using indents and new lines) instead of outputting all the JSON data on one line only.

When you use JSON binding mode, the incoming and outgoing messages are in JSON format. For example, the REST service to create an order ❹ could receive the following JSON payload in an HTTP `POST` call:

```
{
  "partName": "motor",
  "amount": 1,
  "customerName": "honda"
}
```

Because you've enabled JSON binding mode, the `RestBindingProcessor` needs to transform the incoming JSON payload to a POJO class. The JSON payload doesn't carry any kind of identifier that can be used to know which POJO class to use. Therefore, the Rest DSL needs to be marked with the class name of the POJO to use. This is done by using `type(class)` ❹ ❺. The REST service to return an order can likewise be marked with the outgoing type by using `outType(class)` ❸.

In this example, the input and output types are a single entity. The Rest DSL also supports using array/list types, which you specify as follows in Java DSL:

```
put().type(Order[].class)
get("{id}").outType(Order[].class)
```

In XML DSL, you must prepend the name with `[]`, as highlighted here:

```
<put type="camelinaction.Order[]">
<get uri="{id}" outType="camelinaction.Order[]">
```

TIP It's a good practice to mark the incoming and outgoing types in the Rest DSL, which is part of documenting your APIs. You'll learn much more about this in section 10.3, when Swagger enters the stage.

Readers who cheer for XML shouldn't be left out, so we prepared the following listing with the XML version.

Listing 10.12 Using Rest DSL with JSON binding with XML DSL

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

  <restConfiguration component="spark-rest" port="8080"
            bindingMode="json">  ❶
```

❶ Using JSON binding mode

```
  <dataFormatProperty key="prettyPrint" value="true"/>  ❷
```

2

Turns on pretty printing for JSON output

```
    </restConfiguration>

    <rest path="/orders">
      <get uri="{id}" outType="camelinaction.Order">    3
```

3

The output type of this REST service is Order class.

```
        <to uri="bean:orderService?method=getOrder(${header.id})"/>
      </get>
      <post type="camelinaction.Order">    4
```

4

The input type of this REST service is Order class.

```
        <to uri="bean:orderService?method=createOrder"/>
      </post>
      <put type="camelinaction.Order">    5
```

5

The input type of this REST service is Order class.

```
        <to uri="bean:orderService?method=updateOrder"/>
      </put>
      <delete uri="{id}">
        <to uri="bean:orderService?method=cancelOrder(${header.id})"/>
      </delete>
    </rest>

  </camelContext>
```

That would be the changes needed for using JSON binding. This example is provided with the source code in the chapter10/spark-rest-json directory. You can try this example using the following Maven goals:

```
mvn test -Dtest=OrderServiceTest
mvn test -Dtest=SpringOrderServiceTest
```

Now let's make the example support both XML and JSON.

Using both XML and JSON binding with Rest DSL

We continue the example and turn on both JSON and XML binding, which is done by having camel-jackson and camel-jaxb as dependencies in the Maven pom.xml file. To demonstrate how easy it is to switch the REST component, you change camel-spark-rest to camel-undertow instead. Therefore, the updated pom.xml file should include the following dependencies:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-undertow</artifactId>
</dependency>
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jackson</artifactId>
</dependency>
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jaxb</artifactId>
</dependency>
```

The only changes you then have to make are to reconfigure the REST configuration to use `undertow` and both XML and JSON binding, as shown here:

```
restConfiguration()
  .component("undertow").port(8080)
  .bindingMode(RestBindingMode.json_xml)
  .dataFormatProperty("prettyPrint", "true");
```

The book's source code contains this example in the chapter10/undertow-rest-xml-json directory, and you can try the example by running the following Maven goal:

```
mvn compile exec:java
```

Then from the command line, you can use `curl` (or `wget`) to call the REST service, such as getting the order number 1:

```
curl http://localhost:8080/orders/1
```

This returns the response in JSON format:

```
$ curl http://localhost:8080/orders/1
{
  "id" : 1,
  "partName" : "motor",
  "amount" : 1,
  "customerName" : "honda"
}
```

Now to get the response in XML instead, you need to provide the HTTP `Accept` header and specify to accept XML content:

```
$ curl --header "Accept: application/xml" http://localhost:8080/orders/1
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<order>
    <id>1</id>
    <partName>motor</partName>
    <amount>1</amount>
    <customerName>honda</customerName>
</order>
```

That seems easy enough. The client can now specify the acceptable output format, and the Camel Rest DSL binding makes all that happen with little for the developer to do. Well, hold your horses—er, camels. When doing data transformation between XML and POJO with JAXB, you need to do one of the following:

- Annotate the POJO classes with `@XmlRootElement`

- Provide an `ObjectFactory` class

The former is by far the easiest and most-used approach but requires you to add annotations to all your POJO classes. The latter lets you create an `ObjectFactory` class in the same package where the POJO class resides. Then the `ObjectFactory` class has methods to create the POJO classes and be able to control how to map to the fields. We, the authors of this book, favor using annotations.

TIP You can use JAXB annotations on your POJO classes, which both JAXB and Jackson can use. But Jackson has its own set of annotations you can use to control the mapping between JSON and POJO.

Using Spring Boot configuration with Rest DSL

Spring Boot users can also configure rest configuration in the application.properties or application.yaml file. The previous example can be configured as follows:

```
camel.rest.component=undertow
camel.rest.port=8080
camel.rest.binding-mode=json
camel.rest.data-format-property.prettyPrint=true
```

We have provided an example in the chapter10/springboot-json directory which you can try by running the following Maven goal:

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

From a web browser you can open http://localhost:8080/api/orders/1.

Because the example is running in Spring Boot, it’s recommended to use the servlet engine from Spring Boot instead of undertow, which is done by specifying servlet as the component name:

```
camel.rest.component=servlet
```

But you can often omit configuring this with Spring Boot because Camel is able to automatically detect that camel-servlet is on the classpath and use it. We recommend you take a look at this example for more details.

We'll now leave the REST binding and talk about one last item when using Rest DSL: how do you deal with exceptions?

Handling exceptions and returning custom HTTP status codes with Rest DSL

Integration is hard, especially the unhappy path where things fail and go wrong. This book devotes all of chapter 11 to this subject. In this section, you'll learn how to use the existing error-handling capabilities with Camel and your Rest DSL services.

At Rider Auto Parts, the order service should deal with failures as well, and you've amended the service to throw exceptions in case of failures:

```
public interface OrderService {
  Order getOrder(int orderId) throws OrderNotFoundException;
  void updateOrder(Order order) throws OrderInvalidException;
  String createOrder(Order order) throws OrderInvalidException;
  void cancelOrder(int orderId);
}
```

We've introduced two new exceptions. `OrderNotFoundException` is thrown if a client tries to get an order that doesn't exist. In that situation, you want to return an HTTP status code of 404, which means Resource Not Found. `OrderInvalidException` is thrown when a client tries to either create or update an order and the input data is invalid. In that situation, you want to return a status code 400 to the client. Finally, any other kind of exception is regarded as an internal server error, and an HTTP status code 500 should be returned.

How can you implement such a solution? Luckily, as you'll see in chapter 11 on error-handling, Camel allows you to handle exceptions by using `onException` in the DSL. You spend less than 10 minutes adding the error handling to the `OrderRoute` class that contains the Rest DSL. The following listing shows the REST configuration, error handling, and REST services all together.

Listing 10.13 Using `onException` to handle exceptions and return HTTP status codes

```
restConfiguration()
  .component("undertow").port(8080)
  .bindingMode(RestBindingMode.json_xml)
  .dataFormatProperty("prettyPrint", "true");

onException(OrderNotFoundException.class)    ❶
```

❶ Handles OrderNotFoundException and returns HTTP status 404 with empty body

```
  .handled(true)
  .setHeader(Exchange.HTTP_RESPONSE_CODE, constant(404))
  .setBody(constant(""));

onException(OrderInvalidException.class)    ❷
```

❷ Handles OrderInvalidException and returns HTTP status 400 with empty body

```
  .handled(true)
  .setHeader(Exchange.HTTP_RESPONSE_CODE, constant(400))
  .setBody(constant(""));

onException(Exception.class)    ❸
```

❸ Handles all other exceptions and returns HTTP status 500 with exception message in the HTTP body

```
  .handled(true)
  .setHeader(Exchange.HTTP_RESPONSE_CODE, constant(500))
```

```
    .setBody(simple("${exception.message}\n"));

  rest("/orders")
    .get("{id}").outType(Order.class)
      .to("bean:orderService?method=getOrder(${header.id})")
    .post().type(Order.class)
      .to("bean:orderService?method=createOrder")
    .put().type(Order.class)
      .to("bean:orderService?method=updateOrder")
    .delete("{id}")
      .to("bean:orderService?method=cancelOrder(${header.id})");
```

As you can see, all you did was add three `onException` blocks ❶ ❷ ❸ to catch the various exceptions and then set the appropriate HTTP status code and HTTP body.

The book's source code contains this example in the chapter10/undertow-rest-xml-json directory. We prepared scenarios to demonstrate the three exception handlers used. At first, start the example using the following Maven command:

```
mvn compile exec:java
```

For example, to try to call the `GET` service to get an order that doesn't exist, you can run the following command, which returns HTTP status code 404:

```
$ curl -i --header "Accept: application/json" http://localhost:8080/orde
HTTP/1.1 404 Not Found
```

Another example is to update an order using invalid input data:

```
$ curl -i -X PUT -d @invalid-update.json http://localhost:8080/orders --
HTTP/1.1 400 Bad Request
```

And you've also prepared a special request to trigger a server-side error:

```
$ curl -i -X POST -d @kaboom-create.json http://localhost:8080/orders --
HTTP/1.1 500 Internal Server Error
```

```
...
Forced error due to kaboom
```

One last feature we want to quickly cover is the latest addition to the Rest DSL: being able to call RESTful services.

10.2.6 Calling RESTful services using Rest DSL

From Camel 2.19 onward, the Rest DSL introduced the first functionality to allow Camel to easily call any RESTful service by using the rest component. The rest component builds on the principle of the Rest DSL by using REST verbs and URI templating.

Suppose you want to call an existing REST service that provides a simple API to return geographical information about a given city or country. Such an external REST service can quickly be implemented using Spring Boot and Camel together:

```
@RestController
public class GeoRestController {

  @EndpointInject
  private FluentProducerTemplate template;

  @RequestMapping("/country/{city}")      1
```

1

Defines Spring REST service

```
  public Object address(@PathVariable(name = "city") String city) {
    return template.to("geocoder:address:" + city).request();      2
```

2

Calls Camels geocoder component

```
  }
}
```

The REST service has one REST endpoint ❶ that maps to `/country/{city}`. You then use Camel's geocoder component to obtain geolocation information about the city ❷. This component will call an online service on the internet, which returns detailed information in JSON format.

Now you want to use Camel's rest component to call the REST service. To make this easy, you use a timer to trigger the Camel route periodically.

Listing 10.14 Using the rest component to call the REST service

```
restConfiguration().producerComponent("http4")    1
```

1

Configures Rest DSL producer-side

```
    .host("localhost").port(8080);    1

from("timer:foo?period=5000")
    .setHeader("city", RestProducerRoute::randomCity)    2
```

2

Sets a random city

```
    .to("rest:get:country/{city}")    3
```

3

Calls rest component

```
    .transform().jsonpath("$.results[0].formattedAddress")    4
```

4

Extracts formatted response

```
    .log("${body}");
```

When using Rest DSL to call a REST service, you also need to configure it. Listing 10.14 specifies that you want to use the http4 component as the HTTP client that will perform the REST calls ❶. You also specify the host-name and port number of the REST service ❶. If you omit this informa-tion, you'd need to provide it in the rest endpoint ❸. But if you call the same host, it's recommended that you configure this in the rest configura-tion (as done in the listing). The example then generates a random city name ❷, which will be used when calling the REST service. The REST ser-vice is then called using the `URI rest:get:country/{city}`. The rest component supports URI templating, which means the URI at runtime will be resolved to http://localhost:8080/country/Paris if the random city is Paris, and an HTTP `GET` operation is executed. The information returned is detailed, and you want to log only a short, human-readable message, so you use `jsonpath` to extract a formatted address ❹, which is then logged.

This example is provided with the source code in the chapter10/rest-pro-ducer directory, and you can try it by using this:

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

Notice the console output, which should print information about Paris, London, and Copenhagen.

Sorry, but that's all we managed to cover about the new producer side of the Rest DSL. Apache Camel will improve the Rest DSL in its upcoming re-leases and make working with REST services even easier. A rest-swagger component enables you to call a REST service by referring to operation IDs, if the REST service has Swagger API documentation associated. We also managed to add a new Maven plugin (camel-restdsl-swagger) that al-lows you to generate Camel Rest DSL Java source code based on an exist-ing Swagger API documentation. These new additions are expected to be improved over the upcoming release, so we suggest you take a look at this when you get a chance.

Okay, you've now covered a lot about Rest DSL, but there's more to come. The next section covers APIs and how to document your REST services directly in the Rest DSL by using Swagger.

10.3 API documentation using Swagger

Any kind of integration between service providers and clients (or *consumers* and *producers* in Camel lingo) requires using a data format both parties can use. RESTful services using JSON, or to a lesser extent XML, are by far the popular choices. But this is only one part of the puzzle: the way the data is structured in the payload.

To complete the jigsaw puzzle, you need pieces that address technical areas, such as the following:

- What services does a service producer offer?
- Where are the endpoints to access these services?
- What data format do the services accept?
- What data format can the services respond with?
- What's the mandatory and optional information to provide when calling a service?
- And where should said information be provided—as HTTP headers, query parameters, in the payload?
- What does the service do?
- What error codes can the service return?
- Is the service secured?
- Is there a little example?

And besides the technical questions, as human beings we'd like to know the following:

- What does the service do?
- What do the parameters mean?
- What do incoming and outgoing payloads represent?

As you probably can tell, we could add many other questions. But it all leads to the same picture. How do we document these services, so both service producers and clients can interact successfully?

WSDL AND WADL

SOAP-based web services have service contracts built in from the Web Services Description Language (WSDL) specification. This specification describes the functionality offered by the web services. For RESTful services, an attempt was made to ratify Web Application Description Language (WADL) as an official specification, but that didn't happen. In practice, writing WSDL or WADL schema files by hand is problematic for developers, even for developers trying to read and comprehend existing schema files. Therefore, most often, tools are used to machine-generate schema files based on parsing source code, known as the *source-first approach*. Both WSDL and WADL were designed by a committee and lacked what users wanted. The open source communities, aiming to build something better, brought other projects to life, and out of those, one stood on top: Swagger.

Swagger to the rescue

Swagger is both a specification and framework for describing, producing, consuming, and visualizing RESTful services. Applications written with the Swagger framework contain full documentation of methods, parameters, and models directly in the source code. This ensures that the implementation and documentation of RESTful services are always in sync.

In 2016, the Swagger specification was renamed the *OpenAPI* specification and governed by its founding group under the Linux Foundation. In this chapter, we use the popular term *Swagger* instead of *OpenAPI*.

The Swagger specification is best explained by quoting the OpenAPI website:

> *The goal of the OAI specification is to define a standard, language-agnostic interface to REST APIs that allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection. When properly defined, a consumer can understand and interact with the remote service with a minimal amount of implementation logic. Similar to what interfaces have done for lower-level programming, Swagger removes the guesswork in calling the service.*
>
> —*https://openapis.org/specification*

This section covers getting started with Swagger and documenting a JAX-RS REST service. You'll see how to provide Swagger documentation directly in the Rest DSL to easily document your REST services. We'll finish the Swagger coverage by showing you how to embed the popular Swagger UI into your Camel applications.

10.3.1 Using Swagger with JAX-RS REST services

Swagger provides a set of annotations you can use to document JAX-RS-based REST services and model classes. The annotations are fairly easy to use, but adding annotations to all your REST operations, parameters, and model classes is tedious.

To start using Swagger annotations, you add the following Maven dependency to your pom.xml file:

```
<dependency>
  <groupId>io.swagger</groupId>
  <artifactId>swagger-jaxrs</artifactId>
  <version>1.5.16</version>
</dependency>
```

The following listing shows the Order Service example from Rider Auto Parts, which has been documented using the Swagger annotations.

Listing 10.15 Using Swagger annotations to document JAX-RS REST service

```
import io.swagger.annotations.Api;
import io.swagger.annotations.ApiOperation;
import io.swagger.annotations.ApiParam;
import io.swagger.annotations.ApiResponse;
import io.swagger.annotations.ApiResponses;    (1)
```

(1) Imports Swagger annotations

```
@Path("/orders/")
@Consumes("application/json,application/xml")
@Produces("application/json,application/xml")
@Api(value = "/orders", description = "Rider Auto Parts Order Service")    (2)
```

(2) Documents the REST API

```
public class RestOrderService {

  @GET
  @Path("/{id}")
  @ApiOperation(value = "Get order", response = Order.class)    (3)
```

(3) Documents each REST operation

```
  @ApiResponses({    (5)
```

(5) Documents response of REST operation

```
    @ApiResponse(code = 200, response = String.class,    (5)
            message = "The found order"),    (5)
    @ApiResponse(code = 404, response = String.class,      ❺
                  message = "Cannot find order with the id")})      ❺
  public Response getOrder(@ApiParam(value = "The id of the order",      ❹
```

---

❹

Documents each REST input parameter

```
                                     required = true) @PathParam("id") int orderId
        Order order = orderService.getOrder(orderId);
        if (order != null) {
            return Response.ok(order).build();
        } else {
            return Response.status(Response.Status.NOT_FOUND).build();
        }
    }

    @PUT
  @ApiOperation(value = "Update existing order")      ❸
```

---

❸

Documents each REST operation

```
  public Response updateOrder(@ApiParam(value = "The order to update",      ❹
```

---

❹

Documents each REST input parameter

```
                                         required = true) Order order) {
        orderService.updateOrder(order);
        return Response.ok().build();
    }

    @POST
  @ApiOperation(value = "Create new order")      ❸
```

❸ Documents each REST operation

```
  @ApiResponses({@ApiResponse(code = 200, response = String.class,    ❺
```

❺ Documents response of REST operation

```
                    message = "The id of the created order")})
  public Response createOrder(@ApiParam(value = "The order to create",    ❹
```

❹ Documents each REST input parameter

```
                                 required = true) Order order) {
      String id = orderService.createOrder(order);
      return Response.ok(id).build();
  }

  @DELETE
  @Path("/{id}")
  public Response cancelOrder(@ApiParam(value = "The order id to cancel",    ❹
```

❹ Documents each REST input parameter

```
                             required = true) @PathParam("id") int orderId)
      orderService.cancelOrder(orderId);
      return Response.ok().build();
  }
}
```

To document JAX-RS services using Swagger, you add the Swagger annotations to the JAX-RS resource class. At first, you import the needed Swagger annotations ❶. The `@Api` annotation ❷ is used as high-level documentation of the services in this class. In this example, they're the order services. Each REST operation is then documented using `@ApiOperation` ❸; you supply a description of what the operation does as well as what the operation returns. Each operation that takes input parameters is documented using `@ApiParam` ❹, which is colocated with the parameter. Swagger allows you to further specify response values using `@ApiResponses` / `@ApiResponse` ❺. In this example, you use this to specify that the `createOrder` operation returns the ID of the created order, in the success operation. Also notice that ❺ documents both the successful and error response from the `getOrder` operation. Each `@ApiResponse` must map to an HTTP response code: 200 means success, for example, and 404 means resource not found.

The model classes can be annotated with `@ApiModel`, as shown in the following listing.

Listing 10.16 Document model classes using Swagger annotations

```
@XmlRootElement(name = "order")
@XmlAccessorType(XmlAccessType.FIELD)
@ApiModel(value = "order", description = "Details of the order")   ❶
```

❶

Documents the model class

```
public class Order {

  @XmlElement
  @ApiModelProperty(value = "The order id", required = true)   ❷
```

❷

Documents each field in the model class

```
  private int id;

  @XmlElement
  @ApiModelProperty(value = "The name of the part", required = true)   ❷
```

```
    private String partName;

    @XmlElement
  @ApiModelProperty(value = "Number of items ordered", required = true)    2
    private int amount;

    @XmlElement
  @ApiModelProperty(value = "Name of the customer", required = true)    2
    private String customerName;

      // getter/setters omitted
  }
```

Documenting model classes is easier than JAX-RS classes because you need only to document each field ❷ and the overall model ❶.

After adding all the annotations to your JAX-RS and model classes, you need to integrate Swagger with CXF. Doing this depends on whether you use CXF in a Java EE application, OSGi Blueprint, Spring Boot, or standalone.

Adding Swagger to a CXF application

To use Swagger with CXF, you need to create a `Swagger2Feature` and configure it. This can be done in a JAX-RS application class, as shown in the following listing.

Listing 10.17 Using a JAX-RS application to set up a REST service with Swagger

```
  @ApplicationPath("/")
  public class RestOrderApplication extends Application {    1
```

1

JAX-RS Application class

```


      private final RestOrderService orderService;

      public RestOrderApplication(RestOrderService orderService) {
          this.orderService = orderService;
      }

      @Override
      public Set<Object> getSingletons() {
          Swagger2Feature swagger = new Swagger2Feature();    ❷
```

❷

Configures Swagger feature in CXF

```
        swagger.setBasePath("/");    ❷
        swagger.setHost("localhost:9000");    ❷
        swagger.setTitle("Order Service");    ❷
        swagger.setDescription("Rider Auto Parts Order Service");    ❷
        swagger.setVersion("2.0.0");    ❷
        swagger.setContact("rider@autoparts.com");    ❷
        swagger.setPrettyPrint(true);    ❷

        Set<Object> answer = new HashSet<>();    ❸
```

❸

Provides a set of features to use with CXF

```
        answer.add(orderService);    ❸
        answer.add(new JacksonJsonProvider());    ❸
        answer.add(swagger);    ❸
        answer.add(new LoggingFeature());    ❸
        return answer;
    }
}
```

JAX-RS allows you to set up your JAX-RS RESTful services using your own class implementation that must extend `javax.ws.rs.core.Application`. You can see this in listing 10.17, where you create the `RestOrderApplication` class ❶. In the `getSingletons` method, you can set up the services that the RESTful application should use. To use Swagger with CXF, you need to create and configure an instance of

`Swagger2Feature` ❷. Then you configure the Rider Auto Parts `orderSer-vice` implementation, Jackson for JSON support, and Swagger, and then you enable logging by using the `LoggingFeature` ❸.

To run this application standalone, you need to set up Jetty as an embedded HTTP server. And then you add CXF as a servlet to the HTTP server. The following listing shows how this can be done.

Listing 10.18 Running Jetty with CXF JAX-RS and Swagger

```
public class RestOrderServer {

  public static void main(String[] args) throws Exception {
    DummyOrderService dummy = new DummyOrderService();
    dummy.setupDummyOrders();

    RestOrderService orderService = new RestOrderService();
    orderService.setOrderService(dummy);   ❶
```

❶

Sets up the REST Order Service using dummy implementation

```
    RestOrderApplication app = new RestOrderApplication(orderService);   ❷
```

❷

Creates the JAX-RS application instance

```
    Servlet servlet = new CXFNonSpringJaxrsServlet(app);   ❸
```

❸

Uses CXF JAX-RS servlet without Spring

```
    ServletHolder holder = new ServletHolder(servlet);
    holder.setName("rider");
    holder.setForcedPath("/");
    ServletContextHandler context = new ServletContextHandler();
    context.addServlet(holder, "/*");

    Server server = new Server(9000);    ❹
```

❹ Sets up and starts Jetty embedded server on port 9000

```
    server.setHandler(context);
    server.start();

    // keep that keeps the JVM running omitted
  }
}
```

The example is run standalone, using a plain main class. The example uses a dummy order service ❶. An instance of the JAX-RS application is created ❷, which is then configured as a CXF JAX-RS servlet ❸. And the servlet is then added to an embedded Jetty server ❹, which is started.

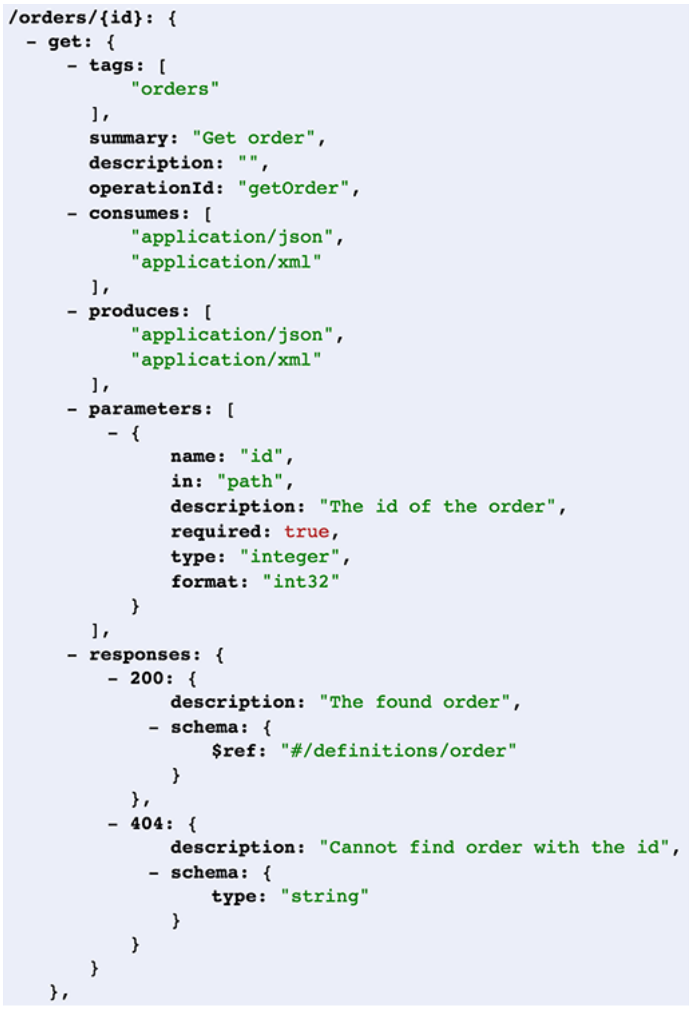
The book's source code contains this example in the chapter10/cxf-swagger directory; you can try the example using the following Maven goal:

```
mvn compile exec:java
```

You can then access the Swagger documentation from the following URL:

```
http://localhost:9000/swagger.json
```

The returned Swagger API is verbose. Figure 10.4 shows the API of the Get Order operation.



```
/orders/{id}: {
  - get: {
      - tags: [
            "orders"
        ],
        summary: "Get order",
        description: "",
        operationId: "getOrder",
      - consumes: [
            "application/json",
            "application/xml"
        ],
      - produces: [
            "application/json",
            "application/xml"
        ],
      - parameters: [
          - {
                name: "id",
                in: "path",
                description: "The id of the order",
                required: true,
                type: "integer",
                format: "int32"
            }
        ],
      - responses: {
          - 200: {
                description: "The found order",
              - schema: {
                    $ref: "#/definitions/order"
                }
            },
          - 404: {
                description: "Cannot find order with the id",
              - schema: {
                    type: "string"
                }
            }
        }
    },
```

Figure 10.4 Swagger API of the Get Order operation. Notice that the operation is detailed with a summary including the formats it consumes and produces, details of the input parameter, and the responses.

Even for this simple example with only four operations, the Swagger API can be verbose and detailed. We encourage you to try this example on your own and view the Swagger API from your web browser.

TIP You can install a JSON plugin to the browser that formats and outputs JSON in a more human-readable format. This example uses the JSONView extension in the Google Chrome browser, shown in figure 10.4.

If you're using the Rest DSL with Camel, documenting your REST services becomes much easier.

10.3.2 Using Swagger with Rest DSL

The Rest DSL comes with Swagger integration out of the box. You need to do two things to enable Swagger with Rest DSL:

- Add camel-swagger-java as a dependency
- Turn on Swagger in the Rest DSL configuration

To add camel-swagger-java as a dependency, you can add the following to your Maven pom.xml file:

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

To turn on Swagger, you need to configure which context-path to use for the Swagger API service. This is done in the REST configuration using `apiContextPath`, as shown in Java DSL:

```
restConfiguration()
    .component("undertow").port(8080)
    .bindingMode(RestBindingMode.json_xml)
    .dataFormatProperty("prettyPrint", "true")
    .apiContextPath("api-doc");
```

And in XML DSL:

```
<restConfiguration component="undertow" port="8080"
                   bindingMode="json_xml" apiContextPath="api-doc">
  <dataFormatProperty key="prettyPrint" value="true"/>
</restConfiguration>
```

The book’s source code contains this example in the chapter10/undertow-swagger directory; you can run this example using the following Maven goal:

```
mvn compile exec:java
```

The Swagger API is available at http://localhost:8080/api-doc.

TIP You can specify whether the Swagger API should provide output in JSON or YAML. Use http://localhost:8080/api-doc/swagger.json for JSON and http://localhost:8080/api-doc/swagger.yaml for YAML.

How does this work?

How the Rest DSL Swagger works

The Swagger API in the Rest DSL works in the same way as any of the other REST services defined in the Rest DSL. When `apiContextPath` is configured, the Swagger API is turned on. Under the hood, Camel creates a route that routes from `apiContextPath` to a rest-api endpoint. In Camel route lingo, this would be as follows:

```
from("undertow:http://localhost:8080/api-doc")
    .to("rest-api://api-doc");
```

The rest-api is a component from camel-core that seamlessly integrates with Swagger in the camel-swagger-java module. When Camel starts up, the rest-api endpoint will discover that camel-swagger-java is available on the Java classpath and enable Swagger integration. What happens next is that the model of the REST services from the Rest DSL is parsed by a Swagger model reader from the camel-swagger-java module. The reader then transforms the Rest DSL into a Swagger API model. After a request to the Swagger API service, the Swagger API model is generated and transformed into the desired output format (either JSON or YAML) and returned.

All this requires no usage of the Swagger annotations in your source code. But they can be used to document your model objects. Besides using the Swagger annotations, you document your services directly by using the Rest DSL.

10.3.3 Documenting Rest DSL services

In the introduction of section 10.3, we discussed the benefits of documenting your REST services in such a way that both humans and computers can understand all the service capabilities, without having to look up information from external resources such as offsite documentation or source code.

HATEOAS

Hypermedia as the Engine of Application State (HATEOAS) is an extension to RESTful principles that the services should be hypermedia driven. This principle goes the extra mile: a client needs no prior knowledge about how to interact with a service, besides the basic principles of hypermedia. An analogy is humans interacting with any website by using a web browser and following hyperlinks. HATEOAS uses the same principles: links are returned in response to clients, which can follow these links, and so on. The Rest DSL doesn't support HATEOAS.

The Rest DSL allows you to document the REST services directly in the DSL to include operations, parameters, response codes and types, and the like.

The following listing shows how the Rider Auto Parts order service application has been documented.

Listing 10.19 Document services using Rest DSL

```
rest("/orders")
   .description("Order services")    ❶
```

❶

Description of the REST service and operations

```
    .get("{id}").outType(Order.class)
    .description("Get order by id")    ❶
    .param().name("id").description("The order id").endParam()    ❷
```

❷

Input parameter documentation

```
        .to("bean:orderService?method=getOrder(${header.id})")

      .post().type(Order.class).outType(String.class)
        .description("Create a new order")    1
```

1

Description of the REST service and operations

```
        .responseMessage()    3
```

3

Response message documentation

```
          .code(200).message("The created order id")
        .endResponseMessage()
          .to("bean:orderService?method=createOrder")

      .put().type(Order.class)
        .description("The order to update")    1
```

1

Description of the REST service and operations

```
        .to("bean:orderService?method=updateOrder")

      .delete("{id}")
        .description("The order to cancel")    1
        .param().name("id").description("The order id").endParam()    2
```

2

Input parameter documentation

```
        .to("bean:orderService?method=cancelOrder(${header.id})");
```

You use `description`, `param`, and `responseMessage` to document your Rest DSLs. `description` ❶ is used to provide a summary of the REST service and operations. `param` ❷ is used for documenting input parameters. And `responseMessage` ❸ is used to document responses.

Two services have input parameters ❷:

```
.param().name("id").description("The order id").endParam()
```

The parameter is named `id`, and it's the order ID, according to the description. The type of the parameter is a path parameter (default). A path parameter maps to a segment in the context-path, such as `http://localhost:8080/orders/{id}`, where `{id}` maps to the path parameter named `id` ❷.

This book's accompanying source code contains this example using Java DSL in the chapter10/undertow-swagger directory. You can try this example using the following:

```
mvn compile exec:java
```

At startup, the example outputs the HTTP URLs you can use to call the services and access the API documentation. The API documentation is available in both JSON and YAML format.

The example is also provided using XML DSL in the chapter10/servlet-swagger-xml directory. This example is deployable as a web application, so you can deploy the .war artifact that's built into Apache Tomcat or WildFly servers. You can also run the example directly from Maven using the following:

```
mvn compile jetty:run
```

You also need to document what input and output your REST service accepts and returns.

10.3.4 Documenting input, output, and error codes

This section covers configuring and documenting what input data and parameters your REST service accepts. Then we move on to documenting what your REST services can send as output responses. And we end the section covering document failures.

A RESTful service can have five types of input parameters in different forms and shapes:

- *body*—The HTTP body
- *formData*—Form data in the HTTP body
- *header*—An HTTP header
- *path*—A context-path segment
- *query*—An HTTP query parameter

The most common parameter types in use are body, path, and query. The previous examples covered so far in this chapter all used two parameter types: body and path.

The input path parameter type

The following snippet is the service used to get an order by providing an order ID:

```
.get("{id}").outType(Order.class)
    .to("bean:orderService?method=getOrder(${header.id})")
```

The path parameter type is in use only when the service uses `{ }` in the URL. As you can see from the snippet, the URL is configured as `{id}` that will automatically let the Rest DSL define API documentation for this parameter.

You can explicitly declare the parameter type in the Rest DSL and configure additional details, as shown here:

```
.param().name("id").description("The order id").endParam()
```

You can also specify the expected data type. For example, you can document that the parameter is an integer value:

```
.param().name("id").dataType("int").description("The order id").endParam
```

And in XML DSL:

```
<param name="id" type="path" dataType="int" description="The order id"/>
```

You want to do this only if you need to document the type with a human description or specify the data type. By default, the data type is `string`.

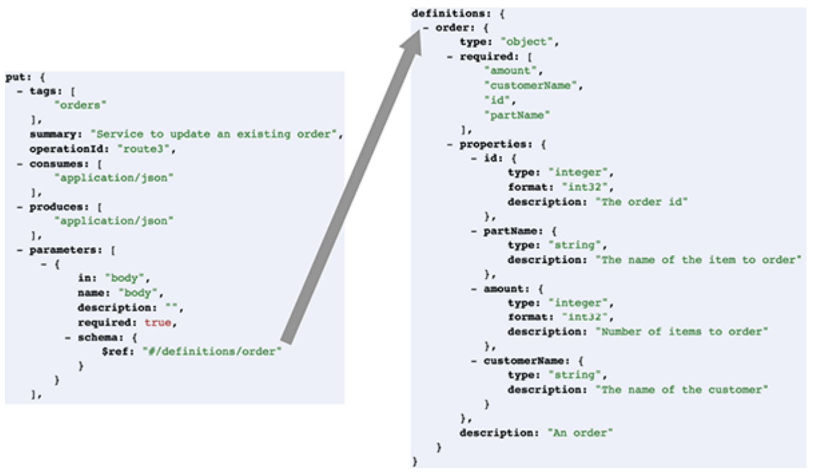
The input body parameter type

The body parameter type is used in the following snippet:

```
.put().type(Order.class)
  .to("bean:orderService?method=updateOrder")
```

It may not be obvious at first sight. When you use `type(Order.class)`, the Rest DSL automatically defines that as a body parameter type. Because the type is a POJO, the data type of the parameter is a schema type. In the generated Swagger API documentation, the schemas are listed at the bottom.

Figure 10.5 shows how the Swagger API documentation maps the body parameter type to the order definition.



```
put: {
  - tags: [
        "orders"
    ],
    summary: "Service to update an existing order",
    operationId: "route3",
  - consumes: [
        "application/json"
    ],
  - produces: [
        "application/json"
    ],
  - parameters: [
      - {
            in: "body",
            name: "body",
            description: "",
            required: true,
          - schema: {
                $ref: "#/definitions/order"
            }
        }
    ],
```

```
definitions: {
  - order: {
        type: "object",
      - required: [
            "amount",
            "customerName",
            "id",
            "partName"
        ],
      - properties: {
          - id: {
                type: "integer",
                format: "int32",
                description: "The order id"
            },
          - partName: {
                type: "string",
                description: "The name of the item to order"
            },
          - amount: {
                type: "integer",
                format: "int32",
                description: "Number of items to order"
            },
          - customerName: {
                type: "string",
                description: "The name of the customer"
            }
        },
        description: "An order"
    }
}
```

Figure 10.5 On the left, the `PUT` operation with a body input parameter refers to a schema definition/order. The schema is defined at the bottom of the Swagger API documentation, shown on the right.

Other input parameter types

All the parameter types are configured in the same style; for example, to specify a query parameter named `priority`, you do this in Java DSL:

```
.param().name("priority").type(RestParamType.query)
    .description("If the matter is urgent").endParam()
```

And in XML DSL:

```
<param name="priority" type="query" description="If the matter is urgent
```

So far, we’ve covered the input parameter types. But there are also outgoing types.

Output types

The response types are used for documenting what the RESTful services are returning. There are two kind of response types:

- *Body*—The HTTP body
- *Header*—An HTTP header

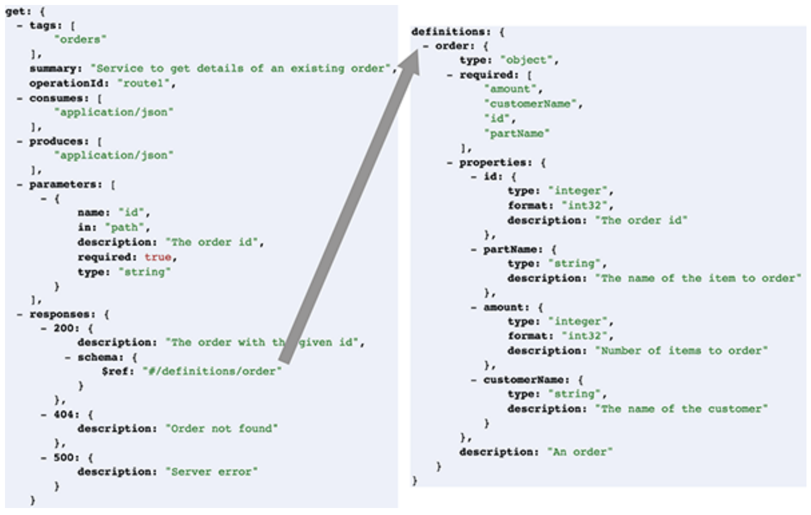
Let’s take a look at how to use those.

The body output type

The response body parameter is used in the following snippet:

```
.get("{id}").outType(Order.class)
  .param().name("id").description("The order id").endParam()
  .to("bean:orderService?method=getOrder(${header.id})")
```

Here you’re using `outType(Order.class)` that causes the Rest DSL to automatically define the response message as an `Order` type. Because the out type is a POJO, the data type of the response is a schema. Figure 10.6 shows the generated Swagger API documentation for this REST service and how it refers to the order schema.



```
get: {
  - tags: [
        "orders"
    ],
    summary: "Service to get details of an existing order",
    operationId: "route1",
  - consumes: [
        "application/json"
    ],
  - produces: [
        "application/json"
    ],
  - parameters: [
      - {
            name: "id",
            in: "path",
            description: "The order id",
            required: true,
            type: "string"
        }
    ],
  - responses: {
      - 200: {
            description: "The order with the given id",
          - schema: {
                $ref: "#/definitions/order"
            }
        },
      - 404: {
            description: "Order not found"
        },
      - 500: {
            description: "Server error"
        }
    }
```

```
definitions: {
  - order: {
        type: "object",
      - required: [
            "amount",
            "customerName",
            "id",
            "partName"
        ],
      - properties: {
          - id: {
                type: "integer",
                format: "int32",
                description: "The order id"
            },
          - partName: {
                type: "string",
                description: "The name of the item to order"
            },
          - amount: {
                type: "integer",
                format: "int32",
                description: "Number of items to order"
            },
          - customerName: {
                type: "string",
                description: "The name of the customer"
            }
        },
        description: "An order"
    }
}
```

Figure 10.6 On the left is the `GET` operation with a successful (code 200) response body that refers to a schema definition/order. The schema is defined at the bottom of the Swagger API documentation, shown on the right.

You can also explicitly define response messages using a description understood by humans, shown highlighted here:

```
.get("{id}").outType(Order.class)
  .description("Service to get details of an existing order")
    .param().name("id").description("The order id").endParam()
    .responseMessage()
      .code(200).message("The order with the given id")
    .endResponseMessage()
    .to("bean:orderService?method=getOrder(${header.id})")
```

And in XML DSL:

```
<get uri="/{id}" outType="camelinaction.Order">
  <description>Service to get details of an existing order</description>
  <param name="id" type="path" description="The order id"/>
  <responseMessage code="200" message="The order with the given id"/>
  <to uri="bean:orderService?method=getOrder(${header.id})"/>
</get>
```

The code attribute refers to the HTTP status code; 200 is a successful call. In a short while, we'll show you how to document failures.

When a REST service returns a response, you most often use the HTTP body for the response, but you can use HTTP headers also.

The header output type

You can include HTTP headers in the response. If you do, you can document these headers in the Rest DSL, as highlighted here:

```
.get("{id}").outType(Order.class)
  .description("Service to get details of an existing order")
    .param().name("id").description("The order id").endParam()
    .responseMessage()
      .code(200).message("The order with the given id")
      .header("priority").dataType("boolean")
        .description("Priority order")
      .endHeader()
    .endResponseMessage()
  .to("bean:orderService?method=getOrder(${header.id})")
```

And in XML DSL:

```
<get uri="/{id}" outType="camelinaction.Order">
  <description>Service to get details of an existing order</description>
  <param name="id" type="path" description="The order id"/>
  <responseMessage code="200" message="The order with the given id">
    <header name="priority" dataType="boolean"
            description="Priority order"/>
  </responseMessage>
  <to uri="bean:orderService?method=getOrder(${header.id})"/>
</get>
```

You can also document which failure codes a REST service can return.

Documenting error codes

The Rider Auto Parts order service application can return responses to clients using four status codes, listed in table 10.11.

Table 10.11 The four HTTP status codes the Rider Auto Parts order application can return

Code	Exception	Description
200		The request was processed successfully.
400	`OrderInvalidException`	A client attempted to create or update an order with invalid input data. Therefore, the client error code 400 is returned to indicate invalid input.
404	`OrderNotFoundException`	A client attempted to get the status of a nonexistent order. Therefore, response code 404 is returned to indicate no data.
500	`Exception`	There was a server-side error processing the request. Therefore, a generic status code 500 is returned.

All four status codes are documented in the same way in the Rest DSL. The following snippets show two of the four REST services in the Rider Auto Parts application in Java and XML:

```
.get("{id}").outType(Order.class)
  .description("Service to get details of an existing order")
  .param().name("id").description("The order id").endParam()
  .responseMessage()
    .code(200).message("The order with the given id").endResponseMessage
  .responseMessage()
    .code(404).message("Order not found").endResponseMessage()
  .responseMessage()
    .code(500).message("Server error").endResponseMessage()
  .to("bean:orderService?method=getOrder(${header.id})")
```

And the second service in XML DSL:

```
<post type="camelinaction.Order" outType="String">
  <description>Service to submit a new order</description>
  <responseMessage code="200" message="The created order id"/>
  <responseMessage code="400" message="Invalid input data"/>
  <responseMessage code="500" message="Server error"/>
  <to uri="bean:orderService?method=createOrder"/>
</post>
```

How do you handle those thrown exceptions in the Rest DSL? Well, that's done using regular Camel routes in which you use Camel's error handler, such as `onException`. The following snippet shows how to handle the generic exception and transform that into an HTTP Status 500 error message:

```
onException(Exception.class)
  .handled(true)
  .setHeader(Exchange.HTTP_RESPONSE_CODE, constant(500))
  .setBody(simple("${exception.message}\n"));
```

And in XML DSL:

```
<onException>
  <exception>java.lang.Exception</exception>
  <handled><constant>true</constant></handled>
  <setHeader headerName="Exchange.HTTP_RESPONSE_CODE">
    <constant>500</constant>
  </setHeader>
  <setBody>
    <simple>${exception.message}\n</simple>
  </setBody>
</onException>
```

TIP Don't worry if you don't completely understand all details of `onException`. Those details are extensively covered in chapter 11.

We encourage you to try the example that accompanies this book. You can find the Java-based example in chapter10/undertow-swagger, and the

XML-based example in chapter10/servlet-swagger-xml. Both examples have a readme file with further instructions.

When using API documentation, you may need to use various configuration options.

10.3.5 Configuring API documentation

The Rest DSL provides several configuration options for API documentation, as listed in table 10.12.

Table 10.12 **API documentation options for Rest DSL**

Option	Default	Description
`apiComponent`	`swagger`	Indicates the name of the Camel component to use as the REST API (such as swagger).
`apiContextPath`		Configures a base context-path the REST API server will use. Important: you must specify a value for this option to enable API documentation.
`apiContextRouteId`		Specifies the name of the route that the REST API server creates for hosting the API documentation.
`apiContextIdPattern`		Sets a pattern to expose REST APIs only from REST services that are hosted within CamelContexts matching the pattern. You use this when running multiple Camel REST applications in the same JVM and you want to filter which `CamelContext` s are exposed as REST APIs. The pattern name refers to the `CamelContext` name, to match on the current `CamelContext` only.
`apiContextListing`	`false`	Sets whether a listing of all available `CamelContext` s with REST services in the JVM is

Option	Default	Description
		enabled. If enabled, you can discover these contexts. If `false`, only the current `CamelContext` is in use.
`enableCORS`	`false`	Indicates whether to enable CORS headers in the HTTP responses.

You'll often need to use only the `apiContextPath` option, as it's the one that enables API documentation. You won't use the first option, because currently Swagger is the only library integrated with Camel to service RESTful API documentation.

The third, fourth, and fifth options are all related to filtering out which `CamelContext`s and routes to include in the API documentation. The last option is for enabling CORS, which we cover at the end of this section.

Enabling API documentation

To use the Swagger API, you need to enable it in the Rest DSL configuration. To enable the Swagger API, you configure a context-path, which is used as the base path to service the API documentation.

The following snippet highlights how to configure this in Java:

```
restConfiguration()
  .component("undertow").port(8080)
  .bindingMode(RestBindingMode.json_xml)
  .dataFormatProperty("prettyPrint", "true")
  .apiContextPath("api-doc");
```

And in XML DSL:

```
<restConfiguration component="undertow" bindingMode="json_xml"
                   port="8080" apiContextPath="api-docs">
```

```
  <dataFormatProperty key="prettyPrint" value="true"/>
</restConfiguration>
```

When using Swagger API documentation, you'll likely want to provide a set of general information, such as version and contact information. The information is configured using `apiProperty`:

```
restConfiguration()
  .component("undertow").port(8080)
  .bindingMode(RestBindingMode.json_xml)
  .dataFormatProperty("prettyPrint", "true")
  .apiContextPath("api-doc")
  .apiProperty("api.version", "2.0.0")
  .apiProperty("api.title", "Rider Auto Parts Order Services")
  .apiProperty("api.description",
    "Order Service to submit orders and query status")
  .apiProperty("api.contact.name", "Rider Auto Parts");
```

And in XML DSL:

```
<restConfiguration component="undertow" bindingMode="json_xml"
                   port="8080" apiContextPath="api-docs">
  <dataFormatProperty key="prettyPrint" value="true"/>
  <apiProperty key="base.path" value="rest"/>
  <apiProperty key="api.version" value="2.0.0"/>
  <apiProperty key="api.title" value="Rider Auto Parts Order Services"/>
  <apiProperty key="api.description"
    value="Order Service to submit orders and query status"/>
  <apiProperty key="api.contact.name" value="Rider Auto Parts"/>
</restConfiguration>
```

The possible values for the keys in the `apiProperty` are the info object from the Swagger API specification. You can find more details at http://swagger.io/specification/#infoObject.

Filtering CamelContext and routes in API documentation

You may have some REST services that you don't want to be included in the public Swagger API documentation. The easiest way to disable a REST service is to set the `apiDocs` attribute to `false`:

```
.get("/ping").apiDocs(false)
  .to("direct:ping")
```

And in XML DSL:

```
<get uri="/ping" apiDocs="false">
  <to uri="direct:ping"/>
</get>
```

The other options (`apiContextListing`, `apiContextIdPattern`, and `apiContextRouteId`) are all options you may find usable only when you run multiple Camel applications in the same JVM, such as from an application server. The idea is that you can enable API documentation in a single application that can discover and service API documentation for all the Camel applications running in the same JVM. This allows you to have a single endpoint as the entry to access API documentation for all the services running in the same JVM.

If you enable `apiContextListing`, the root path returns all the IDs of the Camel applications that have RESTful services.

The example in chapter10/servlet-swagger-xml has enabled the API context listing, which you can access using the following URL:

```
$ curl http://localhost:8080/chapter10-servlet-swagger-xml/rest/api-doc
[
{"name": "camel-1"}
]
```

As you can see from the preceding output, only one Camel application in the JVM has RESTful services. The name is camel-1, which means the API documentation from within that Camel application is available at the following URL:

```
$ curl http://localhost:8080/chapter10-servlet-swagger-xml
  /rest/api-doc/camel-1
{
  "swagger" : "2.0",
  "info" : {
    "description" : "Order Service to submit orders and query status",
    "version" : "2.0.0",
    "title" : "Rider Auto Parts Order Services",
    "contact" : {
      "name" : "Rider Auto Parts"
    }
  }
...
```

Swagger lets you visualize the API documentation via a web browser. The web browser requires online access to the Swagger API documentation. Because the web browser is separated from the context-path where the Swagger API documentation resides, you need to enable CORS.

10.3.6 Using CORS and the Swagger web console

Over the years, web browsers have become more secure and can't access resources outside the current domain. If a user visits the web page http://rider.com, that website can freely load resources from the rider.com domain. But if the web page attempts to load resources from outside that domain, the web browser would disallow that.

But it's not that simple, as some resources are always allowed (such as CSS styles, images, and scripts), but advanced requests (such as `POST`, `PUT`, and `DELETE`) aren't allowed. *Cross-Origin Resource Sharing* (*CORS*) is a way of allowing clients and servers to negotiate whether the client can access those advanced resources. The negotiation happens using a set of HTTP headers in which the client specifies the resource it wants access to, and the server then responds with a set of HTTP headers that tells the client what's allowed.

Our web browsers of today are all CORS compliant and perform these actions out of the box.

Why do we need CORS, you may ask? You may need CORS, for example, if you build web applications using JavaScript technology that calls RESTful services on remote servers.

It's easy to enable CORS when using the Rest DSL, which is done in the REST configuration:

```
restConfiguration()
  .component("undertow").port(8080)
  .bindingMode(RestBindingMode.json_xml)
  .dataFormatProperty("prettyPrint", "true")
  .enableCORS(true)
  .apiContextPath("api-doc")
```

And in XML:

```
<restConfiguration component="servlet" bindingMode="json"
                   contextPath="chapter10-swagger-ui/rest" port="8080"
                   apiContextPath="api-doc" apiContextListing="true"
                   enableCORS="true">
```

When CORS is enabled, the CORS headers in table 10.13 are in use.

Table 10.13 Default CORS headers in use

HTTP header	Value
`Access-Control-Allow-Origin`	`*`
`Access-Control-Allow-Methods`	`GET`, `HEAD`, `POST`, `PUT`, `DELETE`, `TRACE`, `OPTIONS`, `CONNECT`, `PATCH`
`Access-Control-Allow-Headers`	`Origin`, `Accept`, `X-Requested-With`, `Content-Type`, `Access-Control-Request-Method`, `Access-Control-Request-Headers`
`Access-Control-Max-Age`	`3600`

If the default value isn't what you need, you can customize the CORS headers in the REST configuration:

```
restConfiguration()
  .component("undertow").port(8080)
  .bindingMode(RestBindingMode.json_xml)
  .dataFormatProperty("prettyPrint", "true")
  .enableCORS(true)
  .apiContextPath("api-doc")
  .apiProperty("api.version", "2.0.0")
  .apiProperty("api.title", "Rider Auto Parts Order Services")
  .apiProperty("api.description",
    "Order Service to submit orders and query status")
  .apiProperty("api.contact.name", "Rider Auto Parts")
  .corsHeaderProperty("Access-Control-Allow-Origin", "rider.com")
  .corsHeaderProperty("Access-Control-Max-Age", "300");
```

And in XML DSL:

```
<restConfiguration component="servlet" bindingMode="json"
                   contextPath="chapter10-swagger-ui/rest" port="8080"
                   apiContextPath="api-doc" apiContextListing="true"
                   enableCORS="true">
  <dataFormatProperty key="prettyPrint" value="true"/>
  <apiProperty key="base.path" value="rest"/>
  <apiProperty key="api.version" value="2.0.0"/>
  <apiProperty key="api.title" value="Rider Auto Parts Order Services"/>
  <apiProperty key="api.description"
    value="Order Service to submit orders and query status"/>
  <apiProperty key="api.contact.name" value="Rider Auto Parts"/>
  <corsHeaders key="Access-Control-Allow-Origin" value="rider.com"/>
  <corsHeaders key="Access-Control-Max-Age" value="300"/>
</restConfiguration>
```

Using this configuration, only clients from the domain rider.com are allowed to access the RESTful services by using CORS. The `Max-Age` option is set to 5 minutes; the client is allowed to cache a preflight request to a resource for that amount of time before the client must renew and call a new preflight request.

Let’s put CORS to use and use the Swagger UI web application.

Embedding Swagger UI web console

The Swagger API documentation can be visualized using the Swagger UI. This web console also allows you to try calling the REST services, making it a great tool for developers. The console consists of HTML pages and scripts and has no server-side dependencies. Therefore, you can more easily host or embed the console on application servers or locally.

The book's source code contains an example in which the web console is embedded together in a Camel application. The example is located in the chapter10/swagger-ui directory; you can try the example using the following Maven goals:

```
mvn compile jetty:run-war
```

Then from a web browser, open the following URL:

```
http://localhost:8080/chapter10-swagger-ui/?url=rest/api-doc/camel-1/swa
```

This loads the Swagger API documentation from the REST services and presents the API documentation beautifully in the web console. You can then click the Expand Operations button and see all four services. Then you can open the `GET` operation, type in `1` as the order ID, and click the Execute button.

Now let's try to browse the Swagger API documentation from a remote service. We have another example in the source code, in the chapter10/spark-rest-ping directory. Now start this example by using the following:

```
mvn compile exec:java
```

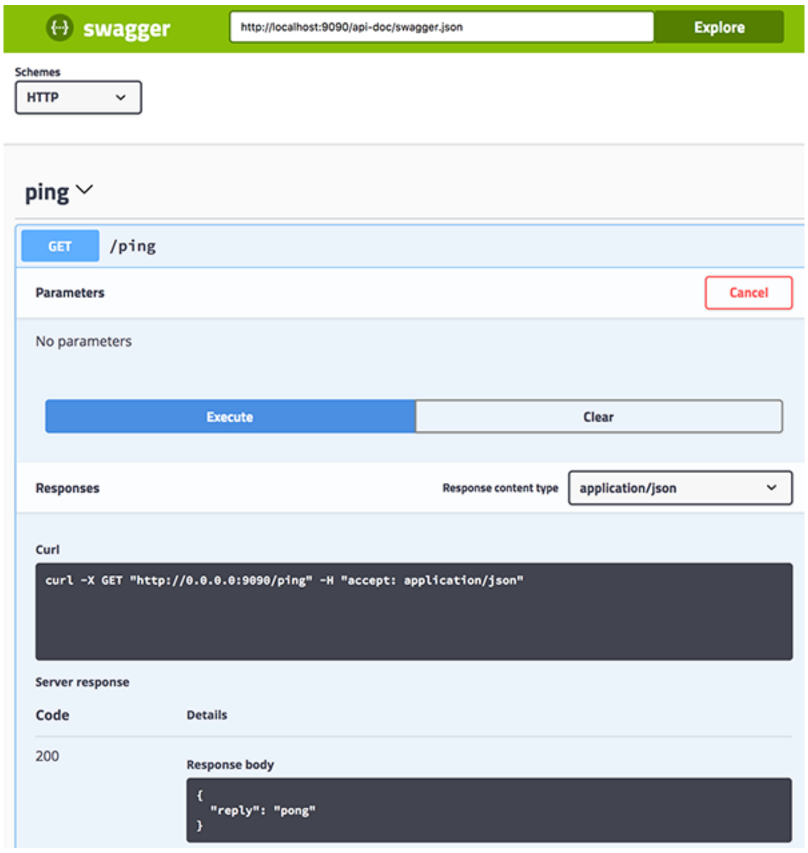
This starts a ping REST service that you can access with this:

```
http://localhost:9090/ping
```

And the Swagger API for the ping service is available here:

```
http://localhost:9090/api-doc
```

You can then, from the Swagger UI, view the ping service by typing at the top of the web page the URL for the ping API documentation; then click the Explore button, as shown in figure 10.7.



http://localhost:9090/api-doc/swagger.json

/ping

No parameters

application/json

```
curl -X GET "http://0.0.0.0:9090/ping" -H "accept: application/json"
```

200

```
{
  "reply": "pong"
}
```

Figure 10.7 Swagger UI browsing the remote ping REST API documentation. At the top of the window, you type the URL to the Swagger API you want to browse and then click the Explore button. Then you can Expand Operations and try calling the services. In the figure, we've called the ping service and received a pong reply in the Response Body section.

The Swagger UI is able to browse and call the remote REST services only because we've enabled CORS. You can view the CORS headers using `curl` or similar commands:

```
$ curl -i http://localhost:9090/ping
HTTP/1.1 200 OK
Date: Sat, 25 Mar 2017 16:57:42 GMT
Accept: */*
Access-Control-Allow-Headers: Origin, Accept, X-Requested-With, Content-
Access-Control-Allow-Methods: GET, HEAD, POST, PUT, DELETE, TRACE,
                              OPTIONS, CONNECT, PATCH
Access-Control-Allow-Origin: *
Access-Control-Max-Age: 3600
breadcrumbId: ID-davsclaus-air-63483-1458925056040-0-1
Content-Type: text/html; charset=UTF-8
Transfer-Encoding: chunked
Server: Jetty(9.2.21.v20160210)
```

```
{ "reply": "pong" }
```

Whoa, that was a lot of content to cover about RESTful services! Who would’ve thought there’s so much to learn and master when using such a popular and modern means of communication?

That’s it folks. That’s all we want to say about RESTful web services.

10.4 Summary and best practices

In this chapter, you looked at building RESTful services with Camel, using the wide range of choices Camel offers. In fact, you started without Camel, built a pure JAX-RS service using CXF, and then explored Camel in various stages. You went through all the other REST components at your disposal and reviewed a short summary of the pros and cons of each choice.

REST services is becoming a top choice for providing and consuming services. In light of this, Camel introduced a specialized Rest DSL that makes it easier for Camel developers to build REST services. This chapter covered all about the Rest DSL.

Swagger is becoming the de facto standard for an API documenting your REST services. You saw how to document your REST services in the Rest DSL, which then is directly serviced as Swagger API documentation out of the box. To view Swagger API documentation, we showed you how to use the Swagger UI web console, which when CORS is enabled, allows you to browse and access remote REST services.

We’ve listed the takeaways for this chapter:

- *JAX-RS is a good standard*—The JAX-RS specification allows Java developers to code RESTful services in an annotation-driven style. The resource class allows the developer full control of handling the REST services. Apache CXF integrates the JAX-RS specification and is a good RESTful framework.
- *Java code or Camel route*—Apache Camel allows you to expose Camel routes as if they’re RESTful services. You need to decide whether you want the full coding power that the JAX-RS resource class allows or to

go straight to Camel routes. When you need both, you can use the hybrid mode with a JAX-RS resource class and call Camel routes by using `ProducerTemplate` (as shown in section 10.1.3).

- *JAX-RS or the Rest DSL*—The Rest DSL is a powerful DSL that allows new users of REST to quickly understand and develop REST services that integrate well with the philosophy and principles of Apache Camel. The Rest DSL has started to support calling REST services from Camel 2.19 onward. It's expected that this will become even easier and smarter in upcoming Camel releases.
- *Use the right REST components*—The Rest DSL allows you to use many Camel components to host the REST services that are easy to swap out. You can start using Jetty and later change the servlet if you run in a Java EE application server. You can also go minimal with Netty or Undertow. Or change to CXF or Restlet, which are first-class RESTful frameworks.
- *Document your REST services*—Your REST services can be documented using the Rest DSL and automatically provide API documentation using Swagger at runtime.
- *Govern and manage your APIs*—There was no room in this chapter to touch on this topic, but we want to share our views that API management becomes important when you have many APIs to integrate.

We've covered a lot of ground pertaining to RESTful web services. Now we'll take a leap into another world, one that's often tackled as an afterthought in integration projects: how to handle situations when things go wrong. We've devoted an entire chapter to Camel's extensive support for error handling.