Rest DSL

Rest DSL

Available as of Camel 2.14

Apache Camel offers a REST styled DSL which can be used with Java or XML. The intention is to allow end users to define REST services using a REST style with verbs such as **`GET`, `POST`**, **`DELETE`** etc.

How it works

The Rest DSL is a facade that builds Rest endpoints as consumers for Camel routes. The actual REST transport is leveraged by using Camel REST components such as Restlet, Spark-rest, and others that has native REST integration.

Components supporting Rest DSL

The following Camel components supports the Rest DSL. See the bottom of this page for how to integrate a component with the Rest DSL.

- camel-coap
- camel-netty-http (also supports Swagger Java)
- camel-netty4-http (also supports Swagger Java)
- camel-jetty (also supports Swagger Java)
- camel-restlet (also supports Swagger Java)
- camel-servlet (also supports Swagger Java)
- camel-spark-rest (also supports Swagger Java from **Camel 2.17**)
- camel-undertow (also supports Swagger Java from **Camel 2.17**)

Rest DSL with Java

To use the Rest DSL in Java then just do as with regular Camel routes by extending the **`RouteBuilder`** and define the routes in the **`configure()`** method.

A simple REST service can be define as follows, where we use **`rest()`** to define the services as shown below:

```
    protected RouteBuilder createRouteBuilder() throws Exception {
        return new RouteBuilder() {
            @Override
            public void configure() throws Exception {
                rest("/say")
                    .get("/hello").to("direct:hello")
                    .get("/bye").consumes("application/json").to("direct:bye")
                    .post("/bye").to("mock:update");

                from("direct:hello")
                    .transform().constant("Hello World");
                from("direct:bye")
                    .transform().constant("Bye World");
            }
        };
    }
```

This defines a REST service with the following URL mappings:

Base Path	URI Template	Verb	Consumes
`/say`	`/hello`	`GET`	*all*
`/say`	`/bye`	`GET`	`application/json`
`/say`	`/bye`	`POST`	*all*

Notice that in the REST service we route directly to a Camel endpoint using the **`to()`**. This is because the Rest DSL has a short-hand for routing directly to an endpoint using **`to()`**. An alternative is to embed a Camel route directly using **`route()`** - there is such an example further below.

Rest DSL with XML

The REST DSL supports the XML DSL also using either Spring or Blueprint. The example above can be define in XML as shown below:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <rest path="/say">
    <get uri="/hello">
      <to uri="direct:hello"/>
    </get>
    <get uri="/bye" consumes="application/json">
      <to uri="direct:bye"/>
    </get>
    <post uri="/bye">
      <to uri="mock:update"/>
    </post>
  </rest>
  <route>
    <from uri="direct:hello"/>
    <transform>
      <constant>Hello World</constant>
    </transform>
  </route>
  <route>
    <from uri="direct:bye"/>
    <transform>
      <constant>Bye World</constant>
    </transform>
  </route>
</camelContext>
```

Using Base Path

The REST DSL allows to define base path to make the DSL a bit more DRY. For example to define a customer path, we can set the base path in **`rest("/customer")`** and then provide the URI templates in the verbs, as shown below:

```
rest("/customers/")
    .get("/{id}").to("direct:customerDetail")
    .get("/{id}/orders").to("direct:customerOrders")
    .post("/neworder").to("direct:customerNewOrder");
```

And using XML DSL it becomes:

```
  <rest path="/customers/">
    <get uri="/{id}">
      <to uri="direct:customerDetail"/>
    </get>
    <get uri="/{id}/orders">
      <to uri="direct:customerOrders"/>
    </get>
    <post uri="/neworder">
      <to uri="direct:customerNewOrder"/>
    </post>
  </rest>
```

The REST DSL will take care of duplicate path separators when using base path and URI templates. In the example above the rest base path ends with a slash (`/`) and the verb starts with a slash (`/`). But Apache Camel will take care of this and remove the duplicated slash.

It is not required to use both base path and URI templates. You can omit the bast path and define the base path and URI template in the verbs only. The example above can be defined as:

```
    <rest>
      <get uri="/customers/{id}">
        <to uri="direct:customerDetail"/>
      </get>
      <get uri="/customers/{id}/orders">
        <to uri="direct:customerOrders"/>
      </get>
      <post uri="/customers/neworder">
        <to uri="direct:customerNewOrder"/>
      </post>
    </rest>
```

Using Dynamic `to()`

Available as of Camel 2.16

The Rest DSL supports the new **`.toD`** or **`<toD>`** as dynamic to in the **`rest-dsl`**. For example to do a request/reply over JMS where the queue name is dynamic defined:

```
 public void configure() throws Exception {
   rest("/say")
     .get("/hello/{language}").toD("jms:queue:hello-${header.language}");
}
```

And in XML DSL

```
<rest uri="/say">
  <get uri="/hello//{language}">
    <toD uri="jms:queue:hello-${header.language}"/>
  </get>
<rest>
```

See more details at Message Endpoint about the dynamic to, and what syntax it supports. By default it uses the Simple language, but it has more power than so.

Embedding Camel Routes

Each of the rest service becomes a Camel route, so in the first example we have **`2 x GET`** and **`1 x POST`** REST service, which each become a Camel route. We also have two regular Camel routes. Therefore we have **`3 + 2 = 5`** routes in total.

There are two route modes with the Rest DSL:

- mini using a singular to
- embedding a Camel route using route

The first example is using the former with a singular **`to()`**. That's why we end up with **`3 + 2 = 5`** total routes.

The same example could use embedded Camel routes:

```
    protected RouteBuilder createRouteBuilder() throws Exception {
        return new RouteBuilder() {
            @Override
            public void configure() throws Exception {
                rest("/say/hello")
                    .get().route().transform().constant("Hello World");
                rest("/say/bye")
                    .get().consumes("application/json").route().transform().constant("Bye World").endRest()
                    .post().to("mock:update");
        };
    }
```

In the example above, we are embedding routes directly in the rest service using **`.route()`**.

Note: we need to use **`.endRest()`** to tell Camel where the route ends, so we can *go back* to the Rest DSL and continue defining REST services.
Configuring route options

In the embedded route you can configure the route settings such as **`routeId`**, **`autoStartup`** and various other options you can set on routes today.

```
.get().route().routeId("myRestRoute").autoStartup(false).transform().constant("Hello World");
```

Managing Rest Services

Each of the rest service becomes a Camel route, so in the first example we have **`2 x GET`** and **`1 x POST`** REST service, which each become a Camel route. This makes it *the same* from Camel to manage and run these services - as they are just Camel routes. This means any tooling and API today that deals with Camel routes, also work with the REST services.

This means you can use JMX to stop/start routes, and also get the JMX metrics about the routes, such as number of message processed, and their performance statistics.

There is also a Rest Registry JMX MBean that contains a registry of all REST services which has been defined.

Binding to POJOs Using

The Rest DSL supports automatic binding **`json/xml`** contents to/from POJOs using Camels Data Format. By default the binding mode is off, meaning there is no automatic binding happening for incoming and outgoing messages.

You may want to use binding if you develop POJOs that maps to your REST services request and response types. This allows you as a developer to work with the POJOs in Java code.

The binding modes are:

Binding Mode	Description
`off`	Binding is turned off. This is the default option.
`auto`	Binding is enabled and Camel is relaxed and support JSON, XML or both if the needed data formats are included in the classpath. **Note**: if, for example, **`camel-jaxb`** is not on the classpath, then XML binding is not enabled.
`json`	Binding to/from JSON is enabled, and requires a json capabile data format on the classpath. By default Camel will use **`json-jackson`** as the data format. See the INFO box below for more details.
`xml`	Binding to/from XML is enabled, and requires **`camel-jaxb`** on the classpath. See the INFO box below for more details.
`json_xml`	Biding to/from JSON and XML is enabled and requires both data formats to be on the classpath. See the INFO box below for more details.

From **Camel 2.14.1**: when using **`camel-jaxb`** for XML bindings, then you can use the option **`mustBeJAXBElement`** to relax the output message body must be a class with JAXB annotations. You can use this in situations where the message body is already in XML format, and you want to use the message body as-is as the output type. If that is the case, then set the **`dataFormatProperty`** option **`mustBeJAXBElement=false`**.
From **Camel 2.16.3**: the binding from POJO to JSON/JAXB will only happen if the **`content-type`** header includes the word **`json`** or **`xml`** respectively. This allows you to specify a custom **`content-type`** if the message body should not attempt to be marshaled using the binding. For example if the message body is a custom binary payload etc.

To use binding you must include the necessary data formats on the classpath, such as **`camel-jaxb`** and/or **`camel-jackson`**. And then enable the binding mode. You can configure the binding mode globally on the rest configuration, and then override per rest service as well.

To enable binding you configure this in Java DSL as shown below:

```
restConfiguration().component("restlet").host("localhost").port(portNum).bindingMode(RestBindingMode.auto);
```

And in XML DSL:

```
<restConfiguration bindingMode="auto" component="restlet" port="8080"/>
```

When binding is enabled Camel will bind the incoming and outgoing messages automatic, accordingly to the content type of the message. If the message is JSON, then JSON binding happens; and so if the message is XML then XML binding happens. The binding happens for incoming and reply messages. The table below summaries what binding occurs for incoming and reply messages.

Message Body	Direction	Binding Mode(s)	Message Body
XML	Incoming	• `auto` • `xml` • `json_xml`	POJO
POJO	Outgoing	• `auto` • `xml` • `json_xml`	XML
JSON	Incoming	• `auto` • `json` • `json_xml`	POJO
POJO	Outgoing	• `auto` • `json` • `json_xml`	JSON

When using binding you must also configure what POJO type to map to. This is mandatory for incoming messages but optional for outgoing.

For example, to map from **`xml/json`** to a POJO class **`UserPojo`** you do this in Java DSL as shown below:

```
// configure to use restlet on localhost with the given port
// and enable auto binding mode
restConfiguration().component("restlet").host("localhost").port(portNum).bindingMode(RestBindingMode.auto);

// use the rest DSL to define the rest services
rest("/users/")
    .post().type(UserPojo.class)
        .to("direct:newUser");
```

Notice we use `type` to define the incoming type. We can optionally define an outgoing type (which can be a good idea, to make it known from the DSL and also for tooling and JMX APIs to know both the incoming and outgoing types of the REST services.). To define the outgoing type we use **`outType`** as shown below:

```
// configure to use restlet on localhost with the given port
// and enable auto binding mode
restConfiguration().component("restlet").host("localhost").port(portNum).bindingMode(RestBindingMode.auto);

// use the rest DSL to define the rest services
rest("/users/")
    .post().type(UserPojo.class).outType(CountryPojo.class)
        .to("direct:newUser");
```

The **`UserPojo`** is just a plain POJO with getter/setter as shown:

```
public class UserPojo {
    private int id;
    private String name;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

The **`UserPojo`** only supports JSON, as XML requires to use JAXB annotations, so we can add those annotations if we want to support XML also:

```
@XmlRootElement(name = "user")
@XmlAccessorType(XmlAccessType.FIELD)
public class UserPojo {
    @XmlAttribute
    private int id;
    @XmlAttribute
    private String name;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

By having the JAXB annotations the POJO supports both JSON and XML bindings.

Configuring Rest DSL

The Rest DSL allows to configure the following options using a builder style

Option	Default	Description
`component`		The Camel Rest component to use for the REST transport, such as **`restlet`**, **`spark-rest`**. If no component has been explicit configured, then Camel will lookup if there is a Camel component that integrates with the Rest DSL, or if a **`org.apache.camel.spi.RestConsumerFactory`** is registered in the registry. If either one is found, then that is being used.
`scheme`	`http`	The scheme to use for exposing the REST service. Usually **`http`** or **`https`** is supported
`hostname`		The hostname to use for exposing the REST service.
`port`		The port number to use for exposing the REST service. **Note**: if you use servlet component then the port number configured here does not apply, as the port number in use is the actual port number the servlet component is using, e.g., if using Apache Tomcat its the tomcat HTTP port, if using Apache Karaf it's the HTTP service in Karaf that uses port 8181 by default etc. Though in those situations setting the port number here, allows tooling and JMX to know the port number, so its recommended to set the port number to the number that the servlet engine uses.
`contextPath`		Sets a leading context-path the REST services will be using. This can be used when using components such as SERVLET where the deployed web application is deployed using a context-path.

`restHostNameResolver`		If no hostname has been explicit configured, then this resolver is used to compute the hostname the REST service will be using. The resolver supports: • **`allLocalIp`** (**from Camel 2.17**) • **`localHostName`** • **`localIp`** For **Camel 2.16.x** or older: **localHostName** From **Camel 2.17**: **`allLocalIp`**
`bindingMode`	`off`	Whether binding is in use. See further above for more details.
`skipBindingOnErrorCode`	`true`	**Camel 2.14.1**: Whether to skip binding on output if there is a custom HTTP error code header. This allows to build custom error messages that do not bind to JSON/XML etc, as success messages otherwise will do. See below for an example.
`enableCORS`	`false`	**Camel 2.14.1:** Whether to enable CORS headers in the HTTP response.
`jsonDataFormat`		Name of specific JSON data format to use. By default **`json-jackson`** will be used. **Important:** This option is only for setting a custom name of the data format, not to refer to an existing data format instance. **Note:** Currently Jackson is what we recommend and are using for testing.
`xmlDataFormat`		Name of specific XML data format to use. By default **`jaxb`** will be used. **Important:** This option is only for setting a custom name of the data format, not to refer to an existing data format instance. **Note:** Currently only **`jaxb`** is supported.
`componentProperty`		Allows to configure as many additional properties. This is used to configure component specific options such as for Restlet / Spark-Rest etc. The options value can use the # notation to refer to a bean to lookup in the Registry
`endpointProperty`		Allows to configure as many additional properties. This is used to configure endpoint specific options for Restlet / Spark-Rest etc. The options value can use the # notation to refer to a bean to lookup in the Registry
`consumerProperty`		Allows to configure as many additional properties. This is used to configure consumer specific options for Restlet / Spark-Rest etc. The options value can use the # notation to refer to a bean to lookup in the Registry
`dataFormatProperty`		Allows to configure as many additional properties. This is used to configure the data format specific options. For example set property **`prettyPrint=true`** to have JSON outputted in pretty mode. From **Camel 2.14.1:** the keys can be prefixed with either: ▪ `json.in.` ▪ `json.out.` ▪ `xml.in.` ▪ `xml.out.` to denote that the option is only for either JSON or XML data format, and only for either the in or the out going. For example a key with value **`xml.out.mustBeJAXBElement`** is only for the XML data format for the outgoing. A key without a prefix is a common key for all situations. From **Camel 2.17**: the options value can use the # notation to refer to a bean to lookup in the Registry
`corsHeaderProperty`		Allows to configure custom CORS headers.

For example to configure to use the spark-rest component on port **`9091`**, then we can do as follows:

```
restConfiguration().component("spark-rest").port(9091).componentProperty("foo", "123");
```

And with XML DSL

```
<restConfiguration component="spark-rest" port="9091">
  <componentProperty key="foo" value="123"/>
</restConfiguration>
```

You can configure properties on these levels.

- component - Is used to set any options on the Component class. You can also configure these directly on the component.
- endpoint - Is used set any option on the endpoint level. Many of the Camel components has many options you can set on endpoint level.
- consumer - Is used to set any option on the consumer level. Some components has consumer options, which you can also configure from endpoint level by prefixing the option with "consumer."
- data format - Is used to set any option on the data formats. For example to enable pretty print in the JSON data format.
- cors headers - If cors is enabled, then custom CORS headers can be set. See below for the default values which are in used. If a custom header is set then that value takes precedence over the default value.

You can set multiple options of the same level, so you can can for example configure 2 component options, and 3 endpoint options etc.

Enabling or Disabling Jackson JSON Features

Available as of Camel 2.15

When using JSON binding you may want to turn specific Jackson features on or off. For example to disable failing on unknown properties e.g., JSON input has a property which cannot be mapped to a POJO, then configure this using the **`dataFormatProperty`** as shown below:

```
restConfiguration().component("jetty").host("localhost").port(getPort()).bindingMode(RestBindingMode.json)
   .dataFormatProperty("json.in.disableFeatures", "FAIL_ON_UNKNOWN_PROPERTIES");
```

You can disable more features by separating the values using comma, such as:

```
   .dataFormatProperty("json.in.disableFeatures", "FAIL_ON_UNKNOWN_PROPERTIES,
ADJUST_DATES_TO_CONTEXT_TIME_ZONE");
```

Likewise you can enable features using the **`enableFeatures`** such as:

```
restConfiguration().component("jetty").host("localhost").port(getPort()).bindingMode(RestBindingMode.json)
   .dataFormatProperty("json.in.disableFeatures", "FAIL_ON_UNKNOWN_PROPERTIES,
ADJUST_DATES_TO_CONTEXT_TIME_ZONE")
   .dataFormatProperty("json.in.enableFeatures", "FAIL_ON_NUMBERS_FOR_ENUMS,USE_BIG_DECIMAL_FOR_FLOATS");
```

The values that can be used for enabling and disabling features on Jackson are the names of the enums from the following three Jackson classes

- **`com.fasterxml.jackson.databind.SerializationFeature`**
- **`com.fasterxml.jackson.databind.DeserializationFeature`**
- **`com.fasterxml.jackson.databind.MapperFeature`**

The rest configuration is of course also possible using XML DSL

```
<restConfiguration component="jetty" host="localhost" port="9090" bindingMode="json">
  <dataFormatProperty key="json.in.disableFeatures" value="FAIL_ON_UNKNOWN_PROPERTIES,
ADJUST_DATES_TO_CONTEXT_TIME_ZONE"/>
  <dataFormatProperty key="json.in.enableFeatures" value="FAIL_ON_NUMBERS_FOR_ENUMS,USE_BIG_DECIMAL_FOR_FLOATS"
/>
</restConfiguration>
```

Default CORS Headers

Available as of Camel 2.14.1

If CORS is enabled then the follow headers is in use by default. You can configure custom CORS headers which takes precedence over the default value.

Key	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**

Defining a Custom Error Message As-is

If you want to define custom error messages to be sent back to the client with a HTTP error code e.g., such as **400**, **404** etc., then from **Camel 2.14.1**: you just set a header with the key **Exchange.HTTP_RESPONSE_CODE** to the error code (must be 300+) such as 404. And then the message body with any reply message, and optionally set the content-type header as well. There is a little example shown below:

```
restConfiguration().component("restlet").host("localhost").port(portNum).bindingMode(RestBindingMode.json);

// use the rest DSL to define the rest services
rest("/users/")
    .post("lives").type(UserPojo.class).outType(CountryPojo.class)
        .route()
            .choice()
                .when().simple("${body.id} < 100")
                    .bean(new UserErrorService(), "idToLowError")
                .otherwise()
                    .bean(new UserService(), "livesWhere");
```

In this example if the input **id** is a number that is below 100, we want to send back a custom error message, using the **UserErrorService** bean, which is implemented as shown:

```
public class UserErrorService {
    public void idToLowError(Exchange exchange) {
        exchange.getIn().setBody("id value is too low");
        exchange.getIn().setHeader(Exchange.CONTENT_TYPE, "text/plain");
        exchange.getIn().setHeader(Exchange.HTTP_RESPONSE_CODE, 400);
    }
}
```

In the **UserErrorService** bean we build our custom error message, and set the HTTP error code to **400**. This is important, as that tells **rest-dsl** that this is a custom error message, and the message should not use the output POJO binding e.g., would otherwise bind to **CountryPojo**.

Returning a Custom Error Message for JsonParserException

From **Camel 2.14.1**: you return a custom message as-is (see previous section). So we can leverage this with Camel error handler to catch **JsonParserException**, handle that exception and build our custom response message. For example to return a HTTP error code **400** with a hard-coded message, we can do as shown below:

```
onException(JsonParseException.class)
    .handled(true)
    .setHeader(Exchange.HTTP_RESPONSE_CODE, constant(400))
    .setHeader(Exchange.CONTENT_TYPE, constant("text/plain"))
    .setBody().constant("Invalid json data");
```

Parameter Default Values

You can specify default values for parameters in the **`rest-dsl`**, such as the verbose parameter below:

```
  rest("/customers/")
      .get("/{id}").to("direct:customerDetail")
      .get("/{id}/orders")
        .param().name("verbose").type(RestParamType.query).defaultValue("false").description("Verbose order
details").endParam()
          .to("direct:customerOrders")
      .post("/neworder").to("direct:customerNewOrder");
```

From **Camel 2.17**: the default value is automatically set as a header on the incoming Camel **`Message`**. So if the call the **`/customers/id/orders`** do not include a query parameter with key **`verbose`** then Camel will now include a header with key **`verbose=false`** because it was declared as the default value. This functionality is only applicable for query parameters.

Integrating a Camel Component with Rest DSL

Any Apache Camel component can integrate with the Rest DSL if they can be used as a REST service(e.g., as a REST consumer in Camel lingo. To integrate with the Rest DSL, then the component should implement the **`org.apache.camel.spi.RestConsumerFactory`**. The Rest DSL will then invoke the **`createConsumer`** method when it setup the Camel routes from the defined DSL. The component should then implement logic to create a Camel consumer that exposes the REST services based on the given parameters, such as path, verb, and other options. For example see the source code for **`camel-restlet`**, **`camel-spark-rest`**.

Swagger API

The Rest DSL supports Swagger Java by the **`camel-swagger-java`** module. See more details at Swagger and the **`camel-swagger-java`** example from the Apache Camel distribution.

From **Camel 2.16**: you can define each parameter fine grained with details such as name, description, data type, parameter type and so on, using the **`<param>`**. For example to define the **`id`** path parameter you can do as shown below:

```
<!-- this is a rest GET to view an user by the given id -->
<get uri="/{id}" outType="org.apache.camel.example.rest.User">
  <description>Find user by id</description>
  <param name="id" type="path" description="The id of the user to get" dataType="int"/>
  <to uri="bean:userService?method=getUser(${header.id})"/>
</get>
```

And in Java DSL:

```
.get("/{id}").description("Find user by id").outType(User.class)
    .param().name("id").type(path).description("The id of the user to get").dataType("int").endParam()
    .to("bean:userService?method=getUser(${header.id})")
```

The body parameter type requires to use body as well for the name. For example a REST **`PUT`** operation to create/update an user could be done as:

```
<!-- this is a rest PUT to create/update an user -->
<put type="org.apache.camel.example.rest.User">
  <description>Updates or create a user</description>
  <param name="body" type="body" description="The user to update or create"/>
  <to uri="bean:userService?method=updateUser"/>
</put>
```

And in Java DSL:

```
.put().description("Updates or create a user").type(User.class)
    .param().name("body").type(body).description("The user to update or create").endParam()
    .to("bean:userService?method=updateUser")
```

For an example see the **`examples/camel-example-servlet-rest-tomcat`** of the Apache Camel distribution.

See Also

- DSL
- Rest
- Swagger Java
- Spark-rest
- How do I import rests from other XML files