Documentation

Release 0.8.0

Crossing-Tech SA

September 04, 2015

Babel is an elegant way to write your integration solution. It tries to provide as much as possible validation during the definition of your integration solution in order to avoid time spent into testing or deploying invalid code.

Babel is a layer on top of the main integration frameworks and may be used from Scala and Java source code. The following documentation should guide you into your journey toward a new way to write integration in a secure and efficient way.

Currently, Babel provides an API on top of Apache Camel which may be used in Scala. Java API and other integration frameworks implementation would be implemented into the Babel experimental[1] project.

To use Babel on top of Camel, you may use the Babel Camel module. Please have a look to the *Quick start guide* (page 6) and to the *Babel Camel User guide* (page 8) for more details and examples.

To have a better intuition of what is Babel, you would find it in the *Babel Opinion* (page 2) and the *Babel Architecture* (page 4) pages.

In the following code snippet, we compare Babel and Camel Scala DSL. Those two routes are summing the a list of number, provided as a String and routing this sum depending on its positivity.

In the following documentation,

- Keywords are written such as **from**, **to** or **process**
- Classes, packages, modules are written such as `RouteBuilder` or `io.xtech.babel`

In the following sections,

[1] https://github.com/crossing-tech/babel-experimental

CHAPTER

ONE

BABEL OPINION

Babel opinion is that code which defines integration should be easy to write and to read. From the experience we have got using Apache Camel (Java DSL), we have raise the following points:

- A DSL needs to be simple to make it easy to write
- A DSL needs to be explicit to make it easy to read

We have been influenced by the Scala language. In other words, we just want to have the same features we already had with Scala at the route layer. Let's take a look at those points:

1.1 Simple DSL

Simple means we would provide one way to do things. This means selection. Currently, the amount of possibilities does not facilitate the coding process.

Actually, "simple" means composed by one thing, without duplicity nor ornamentation but plain. At this point, simple means also easy in the sense we have tried to restrict the possibilities to the most interesting. Babel adds as less as possible new Interfaces and tries as much as possible to rely on existing and basic tools such as Functions. As we will see latter, clear auto-completion provided by your IDE increase the coding process and adds confidence into what you are actually writing. Select the best way to achieve your task and then *transform a good practice into API* could be the main goal of the Babel project.

1.2 Explicit DSL

As a first taste, explicit means to avoid some implicits. In Apache Camel, an implict we found really dangerous and not advantageous was the Implict type converters: Apache Camel implicitely add type conversion depending on your code expectations. In other words, with Apache Camel, if your code expects some type concerning its input which is not the output of the previous step, this will be fixed implicitely at runtime. On the contrary, with Babel, the input type is as explicit as possible. For convenience, it is simply computed from the previous step output. Even if you do not see it in your code, your IDE may compute it at compile time and this is used to validate your code before having actually run it.

More important is the fact that adding information about the input type helps to read your code (and also to review it). Babel tends also to separate (explicit) type conversion from input transformation.

1.3 Explicit and Simple DSL

Now let's have a look to what does those two feature provide together to the coding process. Actually, we may also resume those two aspects as **typed** and **functional**.

In other words, Babel lets you define processing in functional way and raise Scala type inference at the route level.

Auto completion takes advantages from the typing as much as from the simplicity:

- Having only the possible keywords in completion helps to find the good one
- Type inference checks and provides inputs type to your functions

CHAPTER

TWO

BABEL ARCHITECTURE

2.1 Babel overview

Babel is a Domain Specific Language (DSL). This means Babel helps you to specify what you want to achieve and not to make it directly.

You may see Babel as one more DSL for Apache Camel, among well-known ones such as the Java, Scala and XML DSL:

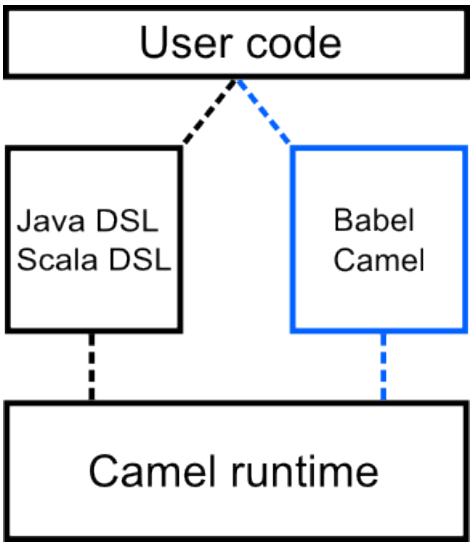


Fig. 2.1: Babel Camel is another way to write Apache Camel routes.

2.2 Babel Modules

Babel is splitted into modules:

- **babel-fish** is the core of Babel
- **babel-camel** is a connector for Apache Camel
- **babel-spring** is an experimental connector for Spring Integration (coming soon)

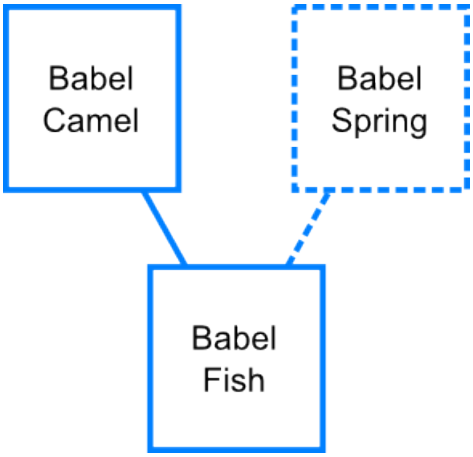


Fig. 2.2: Babel's modules : **babel-fish**, **babel-camel** and the experimental **babel-spring**

CHAPTER

THREE

QUICK START GUIDE

Babel depends mainly on the following artifacts:

- org.scala-lang/scala-library/2.10.4
- org.apache.camel/camel-core/2.12.4

3.1 Using Maven

To add Babel Camel in a Maven project, just add the following dependencies to your pom.xml file:

```
<dependency>
  <groupId>io.xtech.babel</groupId>
  <artifactId>babel-camel-core</artifactId>
  <version>BABEL_VERSION</version>
</dependency>
```

If you don't want to build the Babel project on your machine (otherwise, see *Build Babel with Maven* (page 32)), use the Sonatype Snapshot repository to your Maven configuration:

```
<repository>
     <id>oss-sonatype</id>
     <name>oss-sonatype</name>
     <url>https://oss.sonatype.org/content/repositories/snapshots/</url>
     <snapshots>
         <enabled>true</enabled>
     </snapshots>
 </repository>
```

If you also want to use the Babel Camel Mock extension for your tests, you may also add:

```
<dependency>
  <groupId>io.xtech.babel</groupId>
  <artifactId>babel-camel-mock</artifactId>
  <version>BABEL_VERSION</version>
  <scope>test</scope>
</dependency>
```

Where BABEL_VERSION is replaced by the actual Babel version.

3.2 Using Sbt

To add Babel Camel in a Sbt project, you may just add the following dependencies and resolver to your build configuration:

```
resolvers += "Sonatype OSS Snapshots" at
                  "https://oss.sonatype.org/content/repositories/snapshots

libraryDependencies += "io.xtech.babel" %% "babel-camel-core" % "BABEL_VERSION"
//if you want to use the mock keyword to simplify your tests:
libraryDependencies += "io.xtech.babel" %% "babel-camel-mock" % "BABEL_VERSION"
                                                                    % "test"
```

Where BABEL_VERSION is replaced by the actual Babel version.

CHAPTER

FOUR

BABEL CAMEL USER GUIDE

In most examples, we are writing a *route* which represents the flow of your integration solution. Those *routes* are usually defined in a *RouteBuilder*. Most of the keywords are dealing with the Camel Message, unless specified as body specific (for example, **process** deals with a Camel Message and **processBody** deals with the body of the Camel Message).

> **Warning:** In the Message interface, its body is represented as an Option[T] to handle the nullable case. Thus, the keywords, such as **process**, which handles a Message should be aware of this point. The keywords, such as **processBody**, which handles directly the Message body do not have to care about that.

4.1 Babel Camel using the Spring Application Context

This page explains how to integrate a Babel Camel route with a Spring Application Context.

A route defined with the Babel Camel DSL may be transparently loaded into a Spring Application Context, such as that:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
  ~
  ~  Copyright 2010-2014 Crossing-Tech SA, EPFL QI-J, CH-1015 Lausanne, Switzerland.
  ~  All rights reserved.
  ~
  ~ ============================================================================
  -->

<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://camel.apache.org/schema/spring
       http://camel.apache.org/schema/spring/camel-spring.xsd">

    <camelContext xmlns="http://camel.apache.org/schema/spring" id="babel-camel-context">
        <packageScan>
            <package>io.xtech.babel.camel.builder.spring</package>
        </packageScan>
    </camelContext>

</beans>
```

With the corresponding Babel Camel route:

```
package io.xtech.babel.camel.builder.spring

import io.xtech.babel.camel.builder.RouteBuilder
```

```
class MyRouteBuilder extends RouteBuilder {
  from("direct:babel-rb-1").routeId("route1").to("mock:babel-rb")
  from("direct:babel-rb-2").routeId("route2").to("mock:babel-rb")
}
```

Warning: Unfortunately, the injection using setters may cause Babel initialization to fail because Babel may get intialized before every required Spring Bean. To handle this, please use the `io.xtech.cf.babel.camel.builder.SpringRouteBuilder` and define your route in the *configure* method body. You may also use the constructor injection if you prefer to rely on the basic `io.xtech.babel.camel.builder.RouteBuilder`.

```
import org.springframework.beans.factory.annotation.Autowired

import scala.beans.BeanProperty

class SetterInjectionRouteBuilder extends SpringRouteBuilder {

  @Autowired
  @BeanProperty
  var aBean: MyBeanProcessor = _

  def configure(): Unit = {
    from("direct:babel-rb-setter").as[String].
      processBody(aBean.doSomething).
      to("mock:babel-rb-setter")
  }
}
```

4.2 Babel Camel Samples

4.2.1 Process - Split - Router Sample

This snippet of code defines a Route which receives a list of country id and dispatch to the corresponding country.

```
val routeDef = new BabelRouteBuilder {

  val splitString = (body: String) => body.split(",").toList

  val isForSwitzerland = (msg: Message[String]) => msg.body.fold(false)(_ == "CH")

  val isForGermany = (msg: Message[String]) => msg.body.fold(false)(_ == "D")

  val filterGermanyErrors = (msg: Message[String]) => {
    !msg.headers.contains("GermanyCurrencyFailure")
  }

  val isForFrance = (msg: Message[String]) => msg.body.fold(false)(_ == "F")

  from("direct:input").as[String]
    .processBody(splitString)
    .splitBody(list => list.iterator)
    .processBody(string => string.toUpperCase)
    .choice {
      c =>
        c.when(isForSwitzerland).to("mock:switzerland")
        c.when(isForFrance).to("mock:france")
        c.when(isForGermany).filter(filterGermanyErrors).to("mock:germany")
    }
```

```
    .to("mock:output")
}
```

4.3 Babel Camel Basics

The Babel Camel Basics part exposes which basic statement may be defined.

4.3.1 Messages

In Babel Camel, the base interface which models a message which passes through the route is called Message. A Message contains a payload called body. From Camel point of view, a Babel Message may be understood as the in Message of an Exchange with required methods to read and write the Exchange properties.

For more details, please have a look at the Transformations part.

4.3.2 Basics

A simple route without changing or routing the message. The output will be the same as the input.

```
import io.xtech.babel.camel.builder.RouteBuilder

val routeDef = new RouteBuilder {
  //sends what is received in the direct
  //  endpoint to the mock endpoint
  from("direct:input").to("mock:output")
}
```

The producer may set the exchange as InOnly by setting the second argument of to, to false. This would override the default behaviour of the producer.

```
import io.xtech.babel.camel.builder.RouteBuilder

val routeDef = new RouteBuilder {
  //the mock endpoint is set in InOnly Exchange Pattern
  from("direct:input").to("mock:output", false)
}
```

The producer may set the exchange as InOut by setting the second argument of to, to true. This would override the default behaviour of the producer.

```
import io.xtech.babel.camel.builder.RouteBuilder

val routeDef = new RouteBuilder {
  //the mock endpoint is set in InOut Exchange Pattern
  from("direct:input").to("mock:output", true)
}
```

4.3.3 Route Id

The **routeId** will give an id to the route, then this id can be used for the monitoring or during testing.

```
val routeBuilder = new RouteBuilder {
  from("direct:input").
    //the routeId of this route will be "bla"
    routeId("bla").
    //the routeId keyword needs to be at the beginning of the route
    //   (enforced by the Babel DSL)
```

```
    to("mock:output")
}
```

The **routeId** can not be specified as null nor an empty string.

```
val route = new RouteBuilder {
  from("direct:input").
    //a routeId may not be empty
    routeId("") must throwA[IllegalArgumentException]
}
```

```
val route = new RouteBuilder {
  from("direct:input").
    //a routeId may not be null
    routeId(null) must throwA[IllegalArgumentException]
}
```

Note: The **routeId** keyword is member of a set of keywords which should follow directly the **from** keyword or any keyword of this set.

4.3.4 As

A basic example with type transformation. The keyword *as* will coerce the type of the message passing within a route to a given type.

```
val routeDef = new RouteBuilder {
  //message bodies are converted to String if required
  from("direct:input").as[String]
    //the processBody concatenates received String with "4"
    .processBody(_ + "4")
    //sends the concatenated string to the mock endpoint
    .to("mock:output")
}
```

4.3.5 RequireAs

A basic example with type requirement. The *requireAs* will type the exchange body for the next keyword and will accept only a message with the given type.

```
import io.xtech.babel.camel.builder.RouteBuilder

val routeDef = new RouteBuilder {
  //Input Message bodies should be of type String
  //  or would throw an Exception
  from(directConsumer).requireAs[String].
    processBody(_ + "4").to(s"mock:$mockProducer")
}
val producer = camelContext.createProducerTemplate()
producer.sendBody(directConsumer, 123) must throwA[CamelExecutionException]
```

The **requiredAs** lets you ensure you will always receive the expected body type. For example, the following may not work.

```
import io.xtech.babel.camel.builder.RouteBuilder

val routeDef = new RouteBuilder {
  //no input Message may satisfies both type constraints,
  //   thus any message sent would throw an Exception.
  from(directConsumer).requireAs[String].requireAs[Int].
```

```
    to(s"mock:$mockProducer")
}
val producer = camelContext.createProducerTemplate()
producer.sendBody(directConsumer, "123") must throwA[CamelExecutionException]
```

Warning: Camel also provides tools to handle data type at runtime (which may be referred to as "runtime typing"). This may cause the regular typing to modify your data after the *requireAs* keyword depending on your ecosystem. Unfortunately, there is no way for Babel to prevent such variable behaviour.

4.3.6 Logging

With a **log**, you can log a defined string (which may use Camel Simple Expression Language) and define:

- the Log level
- the Log name
- a marker for this Log event

```
import io.xtech.babel.camel.builder.RouteBuilder
import org.apache.camel.LoggingLevel

val routeBuilder = new RouteBuilder {
  from("direct:input")
    //logs to the Trace level message such as "received ID-3423 -> toto"
    .log(LoggingLevel.TRACE, "my.cool.toto", "foo", "received: ${id} -> ${body}")
    //logs to the Info level message such as "ID-3423 -> toto"
    .log(LoggingLevel.INFO, "${id} -> ${body}")
    .to("mock:output")
}
```

You may also use a function to define what should be logged:

```
val routeBuilder = new RouteBuilder {
  from("direct:input")
    .log(LoggingLevel.TRACE, "my.cool.toto", "foo", msg => "FOO")
    .log(msg => s"BAR : ${msg.headers}")
    .to("mock:output")
}
```

4.3.7 Route configuration

Callbacks may be added to a given route in order to manage its lifecycle such as :

- **onInit**
- **onStart**
- **onSuspend**
- **onResume**
- **onStop**
- **onRemove**

```
var success: Boolean = false
val routeBuilder = new RouteBuilder {
  from("direct:input").
    //As the route is initialing, the success variable is set to true
    onInit(route => success = true).
    to("mock:output")
}
```

Concerning the exchange lifecycle :

- **onExchangeBegin**
- **onExchangeDone**

```
var success: Boolean = false
val routeBuilder = new RouteBuilder {
  from("direct:input").
    //At each time an exchange reach the end of the route,
    //   the success variable is set to true
    onExchangeDone((exchange, route) => success = true).
    to("mock:output")
}
```

Moreover, you may prevent a route from being started automatically using the **autoStartup** keyword.

```
val routeBuilder = new RouteBuilder {
  from("direct:input").routeId("babel").
    //The route is told not starting with the Camel Context
    //   but wait until beeing started especially.
    autoStartup(false).
    to("mock:output")
}
```

Finally, you may define your own RoutePolicy using the **routePolicy** keyword such as:

```
val throttlingInflightRoutePolicy = new ThrottlingInflightRoutePolicy()
val simpleScheduledRoutePolicy = new SimpleScheduledRoutePolicy()
simpleScheduledRoutePolicy.setRouteStartDate(new Date())
val routeBuilder = new RouteBuilder {
  from("direct:input").
    routeId("babel").
    routePolicy(throttlingInflightRoutePolicy, simpleScheduledRoutePolicy).
    to("mock:output")
}
```

4.3.8 Id

The **id** will set an id to the previous EIP. This may be useful for visualizing your route or for statisticts.

```
val routeBuilder = new RouteBuilder {
  from("direct:input").
    routeId(routeId).
    //the id of the processor will be "myProcess"
    processBody(x => x).id("myProcess").
    //the id of the mock endpoint will be "mock"
    to("mock:babel").id("mock")
}
```

The **id** may also set the consumer id, using the **routId** keyword.

```
  from("direct:input").
    //the id may not be configured for the from
    routeId(routeId).
    to("mock:output")
}
```

4.3.9 Default ids

Babel provides a way to define eip ids by default (without using **id**).

To modify this default behavior, you may create your own naming strategy in your `RouteBuilder` such as:

```
val routeBuilder = new RouteBuilder {

  override protected implicit val namingStrategy: NamingStrategy = new NamingStrategy {
    var index = 0

    override def name(stepDefinition: StepDefinition): Option[String] = {
      index += 1
      Some(s"id-$index")
    }

    override protected[babel] def newRoute(): Unit = {}
  }

  //the id of the from (and thus the routeId) will be "id-1"
  from("direct:input").
    //the id of the endpoint which allows the subroute will be "id-2"
    processBody(x => x).
    //the id of the mock endpoint will be "id-4"
    to("mock:babel-sub")
}
```

You may also define your naming strategy depending on the pattern type:

```
import io.xtech.babel.camel.model.{ LogDefinition, LogMessage }
import io.xtech.babel.fish.NamingStrategy
import io.xtech.babel.fish.model.StepDefinition

val routeDef = new RouteBuilder {

  override protected implicit val namingStrategy = new NamingStrategy {
    override def name(stepDefinition: StepDefinition): Option[String] =
      stepDefinition match {
        //set the id of endpoints to their uri
        case LogDefinition(LogMessage(message)) => Some(s"log:$message")
        //do not modify other EIP ids
        case other                              => None
      }

    override def newRoute(): Unit = {}
  }

  from("direct:input").routeId("babel")
    .process(msg => msg.withBody(_ + "bli"))
    //the id of log EIP will be "log:body ${body}"
    .log("body ${body}")
    //the other pattern id will not be changed by babel
    .to("mock:output")
}
```

4.4 Babel Camel Transformations

A transformation is a way to modify the message.

4.4.1 Marshalling

A marshaller is a way to change the format of the message. For example : XML <-> JSON

With the *marshal* and *unmarshal* keyword, you can choose the direction of the marshalling.

The keywords accepts an instance of a Camel DataFormat object or a reference to an object in a registry (ex : Spring Context).

With a Dataformat

```
import io.xtech.babel.camel.builder.RouteBuilder

val routeDef = new RouteBuilder {
  from("direct:input").
    //Message body type is transformed using a bean id defined lower
    marshal("csvMarshaller").
    to("mock:output")
}

val registry = new SimpleRegistry
//csvDataFormat is a org.apache.camel.spi.DataFormat instance
registry.put("csvMarshaller", csvDataFormat)
```

With a reference

```
import io.xtech.babel.camel.builder.RouteBuilder

val routeDef = new RouteBuilder {
  from("direct:input").
    //csvDataFormat is a org.apache.camel.spi.DataFormat instance
    marshal(csvDataFormat).
    to("mock:output")
}
```

4.4.2 Sorting

The *sort* keyword give a way to sort a part of a message. It accepts two parameters:

- a function that split a part of the message.
- an optional comparator that give how the ordering will be done.

The result of the processing of this keyword will be of type java.util.List

```
val routeBuilder = new RouteBuilder {
  //the input string is "4,3,1,2"
  from("direct:input").as[String].
    //the message body is split and then its output is sorted
    sort(msg => msg.body.getOrElse("").split(",")).
    //the output is List("1", "2", "3", "4")
    to("mock:output")
}
```

You may also provide an *Ordering* to the *sort*:

```
import java.util.{ List => JList }

import io.xtech.babel.camel.builder.RouteBuilder
import org.apache.camel.builder.Builder

val routeBuilder = new RouteBuilder {
  //The sort keyword expects Java list type
  from("direct:input").as[JList[Int]].
    //the exchanges are sorted based on their body
    sort(Builder.body(), EvenOddOrdering).
```

```
    to("mock:output")
}
```

```
import scala.math.Ordering.IntOrdering

//The EvenOddOrdering would order the Integer depending on
//   if they are even or odd
//1,2,3,4 becomes 2,4,1,3
object EvenOddOrdering extends IntOrdering {
  override def compare(a: Int, b: Int): Int = (a, b) match {
    case (a, b) if (a % 2) == (b % 2) =>
      Ordering.Int.compare(a, b)
    case (a, b) if a % 2 == 0 =>
      -1
    case (a, b) =>
      1
  }
}
```

4.4.3 Resequencer

The *resequence* keyword is useful for sorting messages coming out of order. There are two algorithms :

Batch resequencing collects messages into a batch, sorts the messages and sends them to their output.

```
import io.xtech.babel.camel.builder.RouteBuilder
import org.apache.camel.model.config.BatchResequencerConfig

//the resequencing would be done in a batch manner
val batchConfiguration = new BatchResequencerConfig()

val routeBuilder = new RouteBuilder {
  //message bodies are converted to Integer if required
  from("direct:input").as[Int].
    //resequencing is based on the body of the message
    resequence(m => m.body.getOrElse(0), batchConfiguration).
    //sends received Integer in a resquenced sequence to the mock endpoint
    to("mock:output")
}
```

Stream resequencing re-orders (continuous) message streams based on the detection of gaps between messages.

```
import io.xtech.babel.camel.builder.RouteBuilder
import org.apache.camel.model.config.StreamResequencerConfig

//the resequencing would be done in a streaming manner
val streamConfiguration = new StreamResequencerConfig()

val routeBuilder = new RouteBuilder {
  //message bodies are converted to Long if required
  from("direct:input").as[Long].
    //resequencing is based on the body of the message
    resequence(m => m.body.getOrElse(0), streamConfiguration).
    //sends the received Long in a resquenced sequence to the mock endpoint
    to("mock:output")
}
```

4.4.4 Enrich

The *enrich* and *pollEnrich* keywords retrieve additional data from an endpoint and let you combine the original and the new message with an aggregator.

The *enrich* is using a request-reply pattern with a endpoint (ex: Web Service) to obtain more data.

```
val routeDef = new RouteBuilder {
  from("direct:input").
    pollEnrich("seda:enrichRoute", (a, b: Any) => s"$a${b.toString}", 1000).
    to("mock:output")
}
```

The *pollEnrich* is using a Polling Consumer from an endpoint (ex: JMS, SEDA, File) to obtain more data.

```
      val routeDef = new RouteBuilder with Mock {
        from("direct:enricherRoute").mock("enricher")

        from("direct:input").
          requireAs[String].
          //enriches the input with the enricherRoute messages
          //  using the aggregationStrategy
          enrich("direct:enricherRoute", (a: String, b: Any) => s"${a}${b.toString.toInt}").
          mock("output")
      }


      routeDef.addRoutesToCamelContext(camelContext)
      camelContext.start()

      val mockEndpoint = camelContext.mockEndpoint("output")
      val enricherMockEndpoint = camelContext.mockEndpoint("enricher")
      enricherMockEndpoint.returnReplyBody(new SimpleBuilder("123"))

      mockEndpoint.expectedBodiesReceived("bla123", "bli123")

      val producer = camelContext.createProducerTemplate()
      producer.sendBody("direct:input", "bla")
      producer.sendBody("direct:input", "bli")

      mockEndpoint.assertIsSatisfied()
    }

    "do not work with a FoldBodyAggregationStrategy" in new camel {

      import io.xtech.babel.camel.builder.RouteBuilder

      case class Result(string: String) {
        def fold(next: String): Result = Result(string + next)
      }

      val aggregationStrategy = new FoldBodyAggregationStrategy[String, Result](Result(""), (a, b

      val routeDef = new RouteBuilder {
        from("direct:enricherRoute").to("mock:enricher")

        from("direct:input").
          enrich("direct:enricherRoute", aggregationStrategy).
          to("mock:output")
      }

      routeDef.addRoutesToCamelContext(camelContext)
      camelContext.start()

      val enricherMockEndpoint = camelContext.mockEndpoint("enricher")
      enricherMockEndpoint.returnReplyBody(new SimpleBuilder("123"))

      val producer = camelContext.createProducerTemplate()
      producer.sendBody("direct:input", "bla") should throwA[CamelExecutionException]
```

```
  }

  "enrich a message with the pollEnrich keyword and a reference to an aggregationStrategy" in ne

    import io.xtech.babel.camel.builder.RouteBuilder

    // pending


    val routeDef = new RouteBuilder {
      from("direct:input").
        pollEnrichRef("seda:enrichRoute", "aggregationStrategy", 1000).
        to("mock:output")
    }

    val registry = new SimpleRegistry
    registry.put("aggregationStrategy",
      new ReduceBodyAggregationStrategy[String]((a, b) => a + b))
    camelContext.setRegistry(registry)

    routeDef.addRoutesToCamelContext(camelContext)
    camelContext.start()

    val mockEndpoint = camelContext.mockEndpoint("output")
    val enricherMockEndpoint = camelContext.mockEndpoint("enricher")
    enricherMockEndpoint.returnReplyBody(new SimpleBuilder("123"))

    mockEndpoint.expectedBodiesReceived("bla123")

    val producer = camelContext.createProducerTemplate()
    producer.sendBody("seda:enrichRoute", "123")
    producer.sendBody("direct:input", "bla")

    mockEndpoint.assertIsSatisfied()
  }

  "enrich a message with the pollEnrich keyword and an instance of an aggregationStrategy" in ne

    import io.xtech.babel.camel.builder.RouteBuilder

    //pending


    val aggregationStrategy = new ReduceBodyAggregationStrategy[String]((a, b) => a + b)

    val routeDef = new RouteBuilder {
      from("direct:input").
        pollEnrichAggregation("seda:enrichRoute", aggregationStrategy, 1000).
        to("mock:output")
    }

    routeDef.addRoutesToCamelContext(camelContext)
    camelContext.start()

    val mockEndpoint = camelContext.mockEndpoint("output")
    val enricherMockEndpoint = camelContext.mockEndpoint("enricher")
    enricherMockEndpoint.returnReplyBody(new SimpleBuilder("123"))

    mockEndpoint.expectedBodiesReceived("bla123")

    val producer = camelContext.createProducerTemplate()
    producer.sendBody("seda:enrichRoute", "123")
```

```
      producer.sendBody("direct:input", "bla")

      mockEndpoint.assertIsSatisfied()
    }

    "enrich a message with the pollEnrich keyword and a aggregationFunction" in new camel {

      import io.xtech.babel.camel.builder.RouteBuilder

      //pending

      val routeDef = new RouteBuilder {
        from("direct:input").
          pollEnrich("seda:enrichRoute", (a, b: Any) => s"$a${b.toString}", 1000).
          to("mock:output")
      }

      routeDef.addRoutesToCamelContext(camelContext)
      camelContext.start()

      val mockEndpoint = camelContext.mockEndpoint("output")
      val enricherMockEndpoint = camelContext.mockEndpoint("enricher")
      enricherMockEndpoint.returnReplyBody(new SimpleBuilder("123"))

      mockEndpoint.expectedBodiesReceived("bla123")

      val producer = camelContext.createProducerTemplate()
      producer.sendBody("seda:enrichRoute", "123")
      producer.sendBody("direct:input", "bla")

      mockEndpoint.assertIsSatisfied()
    }

  }
}
```

You may also use Camel or Babel AggregationStrategy to define how the incomming message and the enriched one are merged:

```
val routeDef = new RouteBuilder {
  from("direct:input").
    pollEnrichRef("seda:enrichRoute", "aggregationStrategy", 1000).
    to("mock:output")
}

val registry = new SimpleRegistry
registry.put("aggregationStrategy",
  new ReduceBodyAggregationStrategy[String]((a, b) => a + b))
camelContext.setRegistry(registry)
```

```
val aggregationStrategy = new ReduceBodyAggregationStrategy[String]((a, b) => a + b)

val routeDef = new RouteBuilder {
  from("direct:input").
    pollEnrichAggregation("seda:enrichRoute", aggregationStrategy, 1000).
    to("mock:output")
}
```

```
val routeDef = new RouteBuilder {
  from("direct:enricherRoute").to("mock:enricher")

  from("direct:input").
    //enriches the input with the enricherRoute messages
```

```
    //  using the aggregationStrategy
    enrichRef("direct:enricherRoute", "aggregationStrategy").
    to("mock:output")
}

val registry = new SimpleRegistry
registry.put("aggregationStrategy",
  //the used aggregation strategy is stored in a registry
  new ReduceBodyAggregationStrategy[String]((a, b) => a + b))
camelContext.setRegistry(registry)
```

```
val aggregationStrategy = new ReduceBodyAggregationStrategy[String]((a, b) => a + b)

val routeDef = new RouteBuilder {
  from("direct:enricherRoute").to("mock:enricher")

  from("direct:input").
    //enriches the input with the enricherRoute messages
    //  using the aggregationStrategy
    enrich("direct:enricherRoute", aggregationStrategy).
    to("mock:output")
}
```

Warning: It's not recommended to us the enrich and pollEnrich keywords with the io.xtech.babel.camel.model.FoldBodyAggregationStrategy. The only supported Aggregation strategy are io.xtech.babel.camel.model.ReduceBodyAggregationStrategy and custom implementations of the org.apache.camel.processor.aggregate.AggregationStrategy Interface.

4.4.5 Processors

You can transform a message including your own business logic. Such data transformation may be defined either by a function or using a bean. The functional way is always preferred in the Babel philosophy.

With a function

You can transform a message with a function.

The *processBody* keyword works on message bodies.

```
import io.xtech.babel.camel.builder.RouteBuilder

val routeDef = new RouteBuilder {
  //message bodies are converted to String if required
  from("direct:input").as[String].
    //processBody concatenates received string with "bli"
    processBody(JavaProcessors.append(_)).
    //sends the concatenated string to the mock endpoint
    to("mock:output")
}

import io.xtech.babel.camel.builder.RouteBuilder

val routeDef = new RouteBuilder {
  //message bodies are converted to String if required
  from("direct:input").as[String].
    //processBody concatenates received string with "bli"
    processBody(string => string + "bli").
    //sends the concatenated string to the mock endpoint
    to("mock:output")
}
```

The *process* keyword works on messages.

```
import io.xtech.babel.camel.builder.RouteBuilder

val routeDef = new RouteBuilder {
  //message bodies are converted to String if required
  from("direct:input").as[String].
    //process redefines a new Message with Body
    process(msg => msg.withBody(_ + "bli")).
    //sends the concatenated string to the mock endpoints
    to("mock:output")
}
```

Note: Babel provides a minimal API to modify `Message` or Body you are dealing with in your transformation

- withBody creates a copy of the current `Message` and let you define how its Body is transformed (using a function)

```
//append a string to the message's body
val newMsg = message.withBody(_ + "bli")
```

- withHeader creates a copy of the current `Message` and let you add a new Header (using a two arguments: key and value)

```
val newMsg = message.withHeader("c", 42)
```

- withHeaders creates a copy of the current `Message` and let you define how its Headers is transformed (using a function)

```
//add headers to the message
val newMsg = message.withHeaders(headers => headers ++ Map("c" -> 42, "d" -> "dd"))
//replace headers by only one
val newMsg = message.withHeaders(headers => Map("c" -> 42))
```

- exchange to access directly to the wrapped Camel Exchange.

It also provides methods concerning the Exchange properties, exceptions and MessageExchangePattern.

With a Bean

You can transform a message with a bean (using camel way to handle beans)

Warning: This keyword will remove type safety for the rest of your route, thus it has been deprecated and might disappear if no user does require it.

With a reference in Camel registry (or in Spring Context):

```
val routeDef = new RouteBuilder {
  from("direct:input").
    //bean keyword is deprecated!
    bean("myBean").to(mockProducer)
}
```

```
val routeDef = new RouteBuilder {
  from("direct:input").
    //the received message are provided to the "doIt" method
    //   of the class with bean id "myBean"
    //bean keyword is deprecated!
    bean("myBean", "doIt").
    //the bean keyword destroys the type of the next keyword
```

```
    to(mockProducer)
}
```

With an instance:

```
val routeDef = new RouteBuilder {
  from("direct:input").
    //the received message are provided to
    //   the TestBean class method which corresponds
    //bean keyword is deprecated!
    bean(new TestBean).
    to(mockProducer)
}
```

With a class:

```
val routeDef = new RouteBuilder {
  from("direct:input").
    //the received message are provided to
    //   the TestBean class method which corresponds
    //bean keyword is deprecated!
    bean(classOf[TestBean]).
    to(mockProducer)
}
```

4.5 Babel Camel Routing

The Routing is used to define where the messages should be routed, or not.

4.5.1 Multicast

The **multicast** keywords defines a static list of outputs where the message is sent.

```
val routeDef = new RouteBuilder {
  from("direct:input").as[String].
    //received messages are sent to those three mock endpoints
    multicast("mock:output1", "mock:output2", "mock:output3").
    processBody(_ => "tested").
    to("mock:output4").
    processBody(_ + " by babel").
    to("mock:output5")

}
```

You may configure the multicast with an aggregation strategy to define how to merge messages that are output by the multicast's branches.

```
val aggregation = new ReduceBodyAggregationStrategy[String]((x, y) => x)

val routeDef = new RouteBuilder {
  from("direct:input").as[String].
    //received messages are sent to those three mock endpoints
    multicast("mock:output1", "mock:output2", "mock:output3").withAggregation(aggregation).
    to("mock:output4").
    processBody(_ + " by babel").
    to("mock:output5")

}
```

4.5.2 Recipient list

The **recipientList** is like the multicast keyword, but the list can be dynamic and calculated at runtime.

```
val routeDef = new RouteBuilder {
  from("direct:input").as[String].
    //received messages target is defined by the headers of each message
    recipientList(m => m.headers("recipients")).
    to("mock:output4")
}
```

4.5.3 Filter

The **filter** and **filterBody** keywords filter message with a predicate.

In this example, the predicate is a function taking a message and returning a boolean.

Note: Contrary to Apache Camel, Babel does not provide the *end* keyword. After the *filter* (or *filterBody*) keyword, only accepted message are processed by next EIPs. If you want to process also the other message, you may dispatch your message to another part of route, with a *multicast* or a *choice* for example.

4.5.4 Choice

The **choice** keyword gives you a way to choose where you are sending the message.

You configures a choice with **when**, **whenBody** and **otherwise** keywords. Each when accepts a predicate. In this example the predicates are function taking message and returning a boolean.

```
val routeDef = new RouteBuilder {
  from("direct:babel").as[String].choice {
    c =>
      c.when(msg => msg.body == Some("1")).
        processBody(body => body + "done").to("mock:output1")
      c.when(msg => msg.body == Some("2")).
        processBody(body => body + "done").to("mock:output2")
      c.when(msg => msg.body == Some("3")).
        processBody(body => body + "done").to("mock:output3")
      c.otherwise.
        processBody(body => body + "done").
        to("mock:output4")
  }
    .to("mock:output5")
```

4.5.5 Splitter

The **split** keyword is the way to split a message in pieces, the **splitBody** does the same directly on the message body.

In this example the splitting is done with a function which takes the message body and returns an Iterator.

The **splitReduceBody** and the **splitFoldBody** define higher-level EIPs: The possibility to split a content, apply a modification on each part of the content and then merge those parts into a new content. The two keywords differs in their way to merge the final content. They are inspired by the `AggregationStrategy` that are defined in the `Aggregation` part.

The **splitReduceBody** let you define a simple aggregation which does not change the type during the aggregation:

```
val routeDef = new RouteBuilder {
  //message bodies are converted to String if required
  from("direct:babel").as[String].splitReduceBody(_.split(",").iterator) {
    _.to("mock:babel1").requireAs[String]
      .processBody(_.toInt + 1 + "")
      .to("mock:babel2").requireAs[String]
  }((x, y) => s"$x,$y").
    processBody(x => x).
    to("mock:babel3")
}
```

The **splitFoldBody** let you define a more complexe aggregation which does change the type during the aggregation:

```
val routeDef = new RouteBuilder {
  //message bodies are converted to String if required
  from("direct:babel").as[String].splitFoldBody(_.split(",").iterator) {
    _.to("mock:babel1").requireAs[String]
      .processBody(_.toInt + 1)
      .to("mock:babel2").requireAs[Int]
  }("1")((x, y) => s"$x,$y").
    processBody(x => x).
    to("mock:babel3")
}
```

Additionnal configuration

- propagateException when an exception is raised in a sub message, the exception is exposed to the initial message.
- stopOnException when an exception is raised in a sub message, the processing of the initial message is stopped.

4.5.6 Aggregation

An aggregation is a way to combine several messages in a new message. An aggregation is declared with :

- How do you combine the messages?
- How do you group the messages?
- When the aggregation is complete?
  - When a number of message is aggregated? CompletionSize
  - After a period of time? (CompletionInterval)
  - Or a combination?

The DSL contains some default implementations we will show :

- Reduce combines messages with the same type and creates a new message with the same type.
- Fold takes a seed and combines the message with this seed and creates a new message with the type of the seed.
- CamelAggregation and CamelReferenceAggregation (from the *io.xtech.babel.camel.model* package) defines an aggregation using camel specific vocabulary.

Reduce

```
// inputs (1,2,3,4,5,6,7,8,9) -> outputs (6,15,24)
val reduceBody = ReduceBody(
  //defines how message bodies are aggregated
  reduce = (a: Int, b: Int) => a + b,
  //defines when message may be aggregated
  groupBy = (msg: Message[Int]) => "a",
  //defines the size of the aggregation (3 messages)
  completionStrategies = List(CompletionSize(3), CompletionTimeout(1000), ForceCompletionOnStop))

import io.xtech.babel.camel.builder.RouteBuilder

val routeDef = new RouteBuilder {
  from("direct:babel").as[Int].
    aggregate(reduceBody).
    to("mock:output")
}
```

Fold

```
// inputs (1,2,3,4,5,6,7,8,9) -> outputs ("123","456","789")
val foldBody = FoldBody("",
  //defines how message bodies are aggregated
  (a: String, b: Int) => a + b,
  //defines when message may be aggregated
  (msg: Message[Int]) => "a",
  //defines the size of the aggregation (3 messages)
  completionStrategies = List(CompletionSize(3), CompletionFromBatchConsumer))

import io.xtech.babel.camel.builder.RouteBuilder

val routeDef = new RouteBuilder {
  from("direct:babel").as[Int].
    aggregate(foldBody).
    to("mock:output")
}
```

Camel Aggregation

```
import io.xtech.babel.camel.builder.RouteBuilder
import org.apache.camel.processor.aggregate.GroupedExchangeAggregationStrategy

val camelAggr = CamelAggregation(MessageExpression((msg: Message[String]) => "1"),
  aggregationStrategy = new GroupedExchangeAggregationStrategy,
  completionStrategies = List(CompletionSize(3), CompletionInterval(1000)))

val routeDef = new RouteBuilder {
  //message bodies are converted to String if required
  from(directConsumer).as[String].
    //aggregates strings based on the camelAggr defined higher
    aggregate(camelAggr).
    //sends the aggregated string to the mock endpoint
    to("mock:output")
}
```

```
//defines when message should be aggregated
val camelExp = new MessageExpression((a: Message[String]) => "1")

import io.xtech.babel.camel.model.Aggregation.CamelReferenceAggregation
```

```
val camelAggr = CamelReferenceAggregation[String, String](
  correlationExpression = camelExp,
  //defines the string id of the aggregation strategy in the bean registry
  "aggregationStrategy",
  completionStrategies = List(CompletionSize(3)))

val routeDef = new RouteBuilder {
  //message bodies are converted to String if required
  from(directConsumer).as[String].
    //aggregates strings based on the camelAggr defined higher
    aggregate(camelAggr).
    //sends the aggregated string to the mock endpoint
    to("mock:output")
}
```

4.5.7 Wire-Tap

The **wiretap** keyword is the way to route messages to another location while they keep beeing process by the regular flow.

```
from("direct:input-babel").
  //Incoming messages are sent to the direct endpoint
  //   and to the next mock endpoint
  wiretap("direct:babel-tap")
  .to("mock:output-babel")
```

4.5.8 Validate

The **validate** keyword validates messages passing through a route using a function or a Camel predicate.

A message will be valid only if the expression or function is returning true. Otherwise, an exception is thrown.

Camel Predicate

```
val routeBuilder = new RouteBuilder {

  from("direct:input").as[Int].
    validate(Builder.body().isEqualTo(1)).
    to("mock:output")
}
```

Message Function

```
val routeBuilder = new RouteBuilder {

  from("direct:input").as[Int].
    validate(msg => msg.body == Some(1)).
    to("mock:output")
}
```

Body Function

```
val routeBuilder = new RouteBuilder {

  from("direct:input").as[Int].
```

```
    validateBody(body => body == 1).
    to("mock:output")
}
```

4.6 Babel Camel Error management

Babel provides mainly two ways to deal with errors : depending on if the handling is Exception specific or not. In any case, the error handling is done using the **handle** keyword.

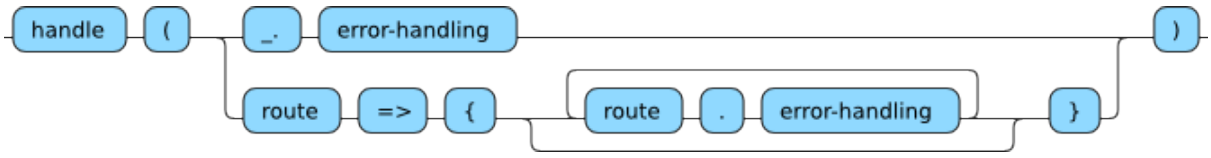


Fig. 4.1: Error handling **handle** can handle one or several error-handling

Note: The error handling keywords, when defined in a route, are member of a set of keywords which should follow directly the **from** keyword or any keyword of this set.

4.6.1 General error handling

This part concerns error handling which does not take into account the type of the raised exception. It just let you define strategies that are used for any raised exception:

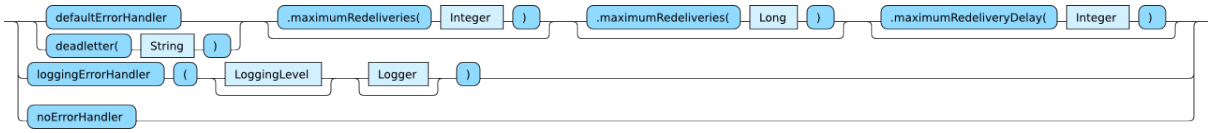


Fig. 4.2: the four error-handling policy we will have a look at and their configuration.

DeadLetter channel

The **deadletter** will send received exception to the defined endpoint.

```
from("direct:babel")
  //Message causing exception would be sent to the deadletter
  .handle(_.deadletter("mock:error").maximumRedeliveries(2))
```

LoggingErrorHandler

The `LoggingErrorHandler` will just received exception to the defined logger at the defined logging level.

```
import org.apache.camel.LoggingLevel
import org.slf4j.LoggerFactory

from("direct:babel")
  //logs raised Exception at the Trace level
  .handle(_.loggingErrorHandler(level = LoggingLevel.TRACE,
    logger = LoggerFactory.getLogger("my.cool.tata")))
```

```
  .processBody(_ => throw new Exception())
  .to("mock:success")
```

DefaultErrorHandler

The default error handler is, as its name claims, implicit. This may be used to override inherited error handler.

```
from("direct:babel")
  //The exchange may get redelivered twice before the Exception is raised higher
  .handle(_.defaultErrorHandler.maximumRedeliveries(2))
```

NoErrorHandler

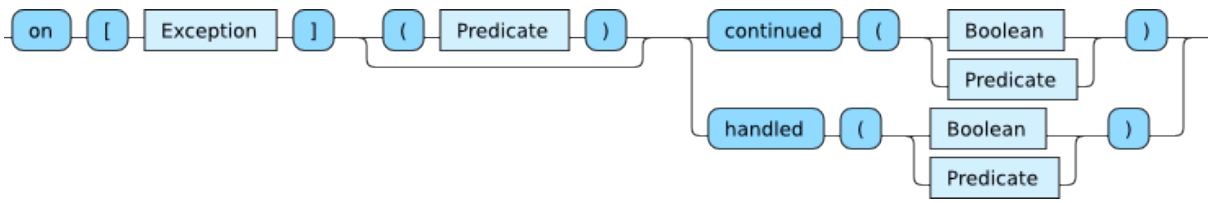
The NoErrorHandler overrides errorhandling defined by a route which is given as input of the current route.

```
val masterRoutes = new RouteBuilder {
  from("direct:babel")
    .handle(_.on[IllegalArgumentException].continuedBody(true))
    .to("direct:channel").to("mock:babel")
}
val errorRoutes = new RouteBuilder {
  from("direct:channel")
    //erase the error handling policy defined in the higher route
    .handle(_.noErrorHandler)
    .process(any => throw new IllegalArgumentException(s"blah:$any")).
    to("mock:exception")
}
```

4.6.2 Exception clauses

You may define specific error management depending on the type of the thrown exception using the **on** keyword:

- **when** the exception should be treated, which is the parameter of the **on** keyword
- **continued**
- **handled**



when

The **when** parameter allows you to be more precise about the exception which should be process through this exception clause. In the below example, the exceptions which message contains “toto” are received in the output. The exceptions which message contains “tata” are handled but would not reach the output.

```
from("direct:input")
  .handle {
    route =>
      //Message containing "toto" and causing an Exception should continue the route
      route.onBody[Exception]((x: Any) => x.toString.contains("toto"))
        .continuedBody(true)
```

```
      //Message containing "tata" and causing an Exception should stop and
      //    the Exceptionshould be tagged as handled
      route.onBody[Exception]((x: Any) => x.toString.contains("tata"))
        .handledBody(true).handlingRoute("direct:exception")
  }
```

continued

The **continued** keyword allows to specify if the exchange should continue processing within the original route. The continued keyword accepts a `Camel Predicate` or a `Boolean function`.

```
from("direct:input")
  .handle {
    route =>
      //Messages which causes IllegalArgumentException
      route.on[IllegalArgumentException].
        //are catch and continue the flow if body is "toto"
        continuedBody((x: Any) => x.toString == "toto")
  }
```

handled

The **handled** keyword accepts a `Camel Predicate` or a `Boolean function`. If the parameter to handled keyword evaluates to true then the `exception` will not be raised with a caller:

```
from("direct:input")
  .handle {
    route =>
      route.on[IllegalArgumentException]
        .handledBody((x: Any) => x.toString.contains("toto"))

  }
```

Handling errors

You may use the **handlingRoute** keyword to define an endpoint which should receive this exception

```
from("direct:input")
  .handle {
    //Any message which cases an IllegalArgumentException
    _.on[IllegalArgumentException].
      //should be transfered, via another route called consuming from "direct:exception"
      handlingRoute("direct:exception")
  }
```

4.6.3 Configure several routes

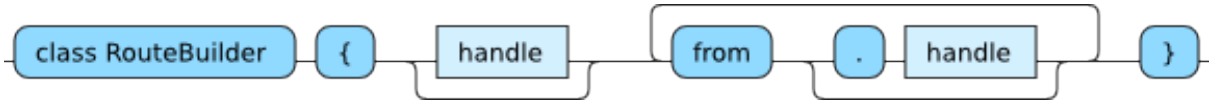


Fig. 4.3: Every Error handling keyword may also be used for every route defined in the RouteBuilder by using the **handle** keyword as if it was the beginning of a route:

```
handle {
  route =>
    //Message raising Exception in any of the following route
    //    will be sent to the deadletter
    route.deadletter("mock:error").maximumRedeliveries(2)
}
from("direct:blah").to("mock:blah")
from("direct:blih").to("mock:blih")
```

In the example above, the two routes send their exceptions to the *mock:error* endpoint.

4.7 Mock Extension

Mock is the first extension of the Babel Camel DSL. It provides both a DSL and some helpers.

4.7.1 Description

An extension for Babel Camel declaring some helpers for testing.

Requirement

The `babel-camel-mock` module needs to in the classpath.

4.7.2 Usage

Import the `io.xtech.babel.camel.mock._` package and extends the `RouteBuilder` with the `Mock` trait.

mock component

For testing a mock endpoint can be declared with the mock endpoint.

```
import io.xtech.babel.camel.mock._

//The Mock extension is added simply by
//  extending the RouteBuilder with
val routeDef = new RouteBuilder with Mock {
  //the mock keyword is the same as typing
  //  to("mock:output1")
  from("direct:input").
    requireAs[String].
    mock("output1").
    //the mock keyword keeps the same body type (here: String)
    processBody(x => x.toUpperCase).
    mock("output2")

}
```

The mock keyword keeps the type of the payload as the mock component do not modify the received messages in most of the cases. For more complexe cases, such as when using `returnReplyBody`, you may fallback to the legacy way to define mock endpoints.

CHAPTER

FIVE

BABEL DEVELOPMENT GUIDE

This sections provides information if you want to develop or understand better Babel.

5.1 Babel Structure

Each of the presented Babel module is composed of the following objects:

- Domain Specific Language (DSL)
- Definition Objects (model)
- Parser

5.1.1 DSL

- The DSL is used by the user when declaring the flow of a route.
- The DSL exposes an API and creates definition object used by the parser.

5.1.2 Definition Objects

- Each definition object defines the behaviour of an EIP in the route.
- All the definitions define the model of the route.

5.1.3 Parser

- A parser takes definition objects and use them to configure the runtime of the route (Apache Camel, Spring Integration).
- A route is defined with all its definition objects.

5.2 Babel Installation Guide

5.2.1 Project Requirements

Operating System	GNU/Linux, Mac OS X, MS Windows
Java VM	Oracle JDK 6
Maven	3.0.5

5.2.2 Build Babel with Maven

To build Babel from sources using Apache Maven[1], in a regular shell:

```
git clone https://github.com/babel-dsl/babel.git
cd babel
export MAVEN_OPTS="-XX:MaxPermSize=256m -Xmx1024m"
mvn install
```

5.2.3 Build Babel with Sbt

To build Babel from sources using Sbt[2], in a regular shell:

```
git clone https://github.com/babel-dsl/babel.git
cd babel
export SBT_OPTS="-XX:MaxPermSize=256m -Xmx1024m"
sbt test publish-local
```

5.2.4 Generate the documentation

to generate the documentation, first install the sphinx documentation generator and then, run:

```
git clone https://github.com/babel-dsl/babel.git
cd babel/babel-doc
make html
#browse build/html/index.html
```

Note: Akka provides some nice documentation on how to generate documentation with Sphinx: Akka Sphinx Documentation[3]

5.3 Babel development

We will define here some guide lines used during the development of Babel.

5.3.1 Build

Babel is build using Apache Maven (even if a Sbt configuration is synchronized with).

5.3.2 Source Code

The Code is reformatted while compiling the sources using sbt (which uses scalariform[4]) To reformat the code:

```
sbt test-compile
```

[1] http://maven.apache.org/
[2] http://www.scala-sbt.org/
[3] http://doc.akka.io/docs/akka/2.0/dev/documentation.html
[4] https://github.com/sbt/sbt-scalariform

5.3.3 Tests

In the Babel modules implemented using Scala, Specs2[5] is the used testing framework.

Test coverage reports are generated using scoverage through the sbt build configuration following:

```
sbt scoverage:test
#browse test reports in modules target directory (target/classes/coverage-report)
```

5.3.4 Deployment

With Maven

Snapshots are deployed locally using the Maven build configuration as following:

```
mvn install
```

With Sbt

Snapshots are deployed using the Sbt build configuration as following:

```
sbt publish-local
```

5.3.5 Release

Releases are done using the Sbt/Maven build configuration (coming soon)

5.4 Contribute to Babel

First of all, thank you for wanting to improve the Babel project. These guidelines may help you in your trip to enrich Babel with your skills, whatever they are.

5.4.1 How to contribute?

Contribute with feedback

Maybe the first way to contribute to the Babel project is to test it and to provide us feedback about your experience.

First, even if it may be obvious, have a look to the current documentation, the existing posts in the mailing list and the github issues. As computer scientists, we love when not repeating ourselves unnecessary.

The best place to ask your questions (and potentially to get answer) is the mailing list[6] . You may also (but not always) chat in our irc room (at irc.codehaus.net, channel #babel)

Contribute with code

We will examine and hopefully accept your code contribution with pleasure. Please have a look to the following points which may help you to contribute:

- Read and sign the CLA[7] .
- Make sure a github issue exists (to avoid other people to work on the same topic).

[5] http://etorreborre.github.io/specs2/
[6] https://groups.google.com/forum/#!forum/babel-user
[7] https://github.com/Crossing-Tech/babel/blob/master/babel-doc/CLA.md

- Ensure your using the Crossing-Tech copyright, you are not addding dependencies incompatible with the licence Apache2.
- Shape your contribution into a Github Pull request[8] .
- Please take the Pull request review as an opportunity to improve the quality of your contribution. Creating the Pull request is certainly not your the final goal.
- Once the Pull requested is accepted, please clean your git branch (it should be more often a single commit).

We would like to keep the contribution process as simple and effective as possible. This procedure may get improve with the time, do not hesitate if you have an opinion.

Babel copyright notice

The copyright notice should look like:

```
/*
 *
 *  Copyright STARTING_YEAR-CURRENT_YEAR Crossing-Tech SA, EPFL QI-J, CH-1015 Lausanne, Switzerla
 *  All rights reserved.
 *
 * ============================================================================================
 */
```

Where the STARTING_YEAR states the creation of the file and the CURRENT_YEAR states the last modification of the file.

Please avoid any mention of author information in the file. Instead of this, please make sure your name get mentioned with the first release of your contribution.

5.5 Babel Extension creation

This page will explains how you can extends the DSL from Babel Camel by adding new keywords and grammar parts to the existing DSL.

> **Warning:** The extension of the DSL requires some knowledge of the Scala language.

5.5.1 Existing Extensions

Mock Extension

Mock is the first extension of the Babel Camel DSL. It provides both a DSL and some helpers.

Description

An extension for Babel Camel declaring some helpers for testing.

Requirement The `babel-camel-mock` module needs to in the classpath.

Usage

Import the `io.xtech.babel.camel.mock._` package and extends the `RouteBuilder` with the `Mock` trait.

[8] https://help.github.com/articles/using-pull-requests

mock component For testing a mock endpoint can be declared with the mock endpoint.

```
import io.xtech.babel.camel.mock._

//The Mock extension is added simply by
//  extending the RouteBuilder with
val routeDef = new RouteBuilder with Mock {
  //the mock keyword is the same as typing
  //  to("mock:output1")
  from("direct:input").
    requireAs[String].
    mock("output1").
    //the mock keyword keeps the same body type (here: String)
    processBody(x => x.toUpperCase).
    mock("output2")

}
```

The mock keyword keeps the type of the payload as the mock component do not modify the received messages in most of the cases. For more complexe cases, such as when using `returnReplyBody`, you may fallback to the legacy way to define mock endpoints.

5.5.2 Create an extension

To create an extension, you need to create a new DSL, some definition objects and a Parser.

Explanation

1. Choose a package for your the new extension like `io.xtech.babel.camel.mock`
2. Create some definition objects that extends `io.xtech.babel.fish.parsing.StepDefinition`.
3. Create a DSL with your new keywords using the definition objects. The DSL will take a `io.xtech.babel.fish.BaseDSL` and extends a `io.xtech.babel.fish.DSL2BaseDSL`
4. Create a trait that extends `io.xtech.babel.camel.parsing.CamelParsing`.
   (a) Declare your parser by defining a parse method which returns a `Process` type and add it to the `steps` of the trait.
   (b) Declare an implicit method using your new DSL.

Example

```
import io.xtech.babel.camel.mock._

//The Mock extension is added simply by
//  extending the RouteBuilder with
val routeDef = new RouteBuilder with Mock {
  //the mock keyword is the same as typing
  //  to("mock:output1")
  from("direct:input").
    requireAs[String].
    mock("output1").
    //the mock keyword keeps the same body type (here: String)
    processBody(x => x.toUpperCase).
    mock("output2")

}
```