**Создание заглушек для интеграционного тестирования на ApacheCamel и ScalaDSL**

**Создание заглушек на ApacheCamel**

**Создание заглушек на ApacheCamel для интеграционного тестирования**

**Заглушки на ApacheCamel и Scala DSL для интеграционного тестирования**

**Интеграционное тестирование: создание заглушек на ApacheCamel и Scala DSL**

**ApacheCamel + Scala DSL = быстрые заглушки для интеграционного тестирования**

В статье рассмотрены примеры использования ApacheCamel для создания тестовых заглушек, а также компонентов информационной системы.

Часто возникает необходимость эмулировать работу какой-либо части системы для интеграционного тестирования, сделать заглушку или написать простой компонент интеграции. Это может быть веб-сервис, возвращающий нужные ответы, или тест, наполняющий базу данных, или приложение, которое считывает сообщение из очереди и возвращает результат обработки, или генератор файлов, или другие компоненты.  
Для разовой проверки интеграции мы бы взяли простое приложение, написанное на на Java,Scala или реализовали сценарий JMeter. Но нам нужна система, которая постоянно работает, отвечает на запросы и не требует действий со стороны тестировщика — запустил и забыл. Для решения этой задачи мы используем приложение, основанное на фреймворке ApacheСamel.

Мы рассмотрим 5 сценариев, от простого к сложному:  
сценарий 1:чтение файлов в одной кодировке, запись – в другой;  
сценарий2:запрос к веб-сервису порасписанию и сохранение сообщения в Redis;  
сценарий3:реализация веб-сервиса с использованием Jetty, который возвращает сообщение в зависимости от параметра GET-запроса;  
сценарий4:чтение сообщения из очереди и отправка сообщения в БД;  
сценарий5:маршрутизация по содержимому файла.  
Есть еще отличный сценарий с чтением сообщения из файла и отправкой его POST-запросом. Пример немного устаревший, но может быть полезен(<http://www.lightbend.com/activator/template/camel-http>; <https://github.com/hilton/activator-camel-http#master>).

**Коротко обинструментах, используемых для решения задачи**

ApacheCamel( <http://camel.apache.org/> ) — java-фреймворк, предназначенный для реализации обмена сообщениями между компонентами информационной системы. Реализует подход EnterpriseIntegrationPatterns(EIP). Позволяет работать с файлами, БД, менеджерами очередей, веб-сервисами и другими компонентами — их порядка 240 на странице проекта (<http://camel.apache.org/component.html>). Компонент может реализовывать производителя сообщения(Producer), потребителя(Consumer), либо может выполнять обе роли.

Порядок работы с Camel таков:  
1) описываем источник сообщения(файл, очередь, БД, сервис, таймер и т.п.);  
2) описываем правила преобразования данных и форматов;  
3) описываем потребителя(потребителей) сообщения(файл, очередь, БД, сервис, вывод в консоль и т.п.) и логику маршрутизации;  
4) запускаем приложение, которое слушает источник, и при появлении сообщения преобразует его и маршрутизирует до получателей.

Scala DSL – язык, который мы будем использовать для описания правил маршрутизации и преобразования сообщений (<http://camel.apache.org/scala-dsl-eip.html>), систему сборки проекта sbt.

**Подготовительныйэтап**  
1) Создадим проект в IDEA на основе SBT.Пример создания проекта можно посмотреть на <https://habrahabr.ru/company/cit/blog/269293/>.

2) В файле build.sbt пропишем настройки:

name := “camel-scaladsl”

version := “1.0”

scalaVersion := “2.11.8”

**val**camelVersion = “2.17.1”

libraryDependencies ++= **Seq**(

*// Компонентыдля Camel*

“org.apache.camel”%“camel-core” % camelVersion,

“org.apache.camel”%“camel-scala” % camelVersion,

*// Длякаждогокомпонентасвоязависимость*

“org.apache.camel”%“camel-quartz” % camelVersion,

“org.apache.camel”%“camel-spring-redis” % camelVersion,

“org.apache.camel”%“camel-http” % camelVersion,

“org.apache.camel”%“camel-jetty” % camelVersion,

“org.apache.camel”%“camel-jms” % camelVersion,

“org.apache.camel”%“camel-jdbc” % camelVersion,

*//Добавимлогирование*

“ch.qos.logback”%“logback-classic” % “1.1.2”,

«org.slf4j» % «slf4j-api» % «1.7.7»,

*//Компонент для работы xml в скала*

“org.scala-lang.modules”%“scala-xml\_2.11” % “1.0.5”,

*// ДрайверБД H2*

“com.h2database”%“h2” % “1.4.192”,

“org.apache.commons”%“commons-dbcp2” % “2.1.1”,

*// Драйвердляброкераactivemq*

“org.apache.activemq”%“activemq-client” % “5.13.3”

)

3) Добавим в папку src/main/resources файл logback.xml, в котором установимуровень логирования и формат сообщения:

<configuration>

<appendername="STDOUT"class="ch.qos.logback.core.ConsoleAppender">

<encoder>

<pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>

</encoder>

</appender>

<root level="INFO">

<appender-ref ref="STDOUT" />

</root>

</configuration>

Если не установить уровень логирования, по умолчанию будет задан уровень DEBUG, то есть на выходе получим слишком много информации.

**Сценарий 1: чтение файлов в одной кодировке, запись – в другой**

В этом примере создадим простое приложение, которое будет использовать компонент <http://camel.apache.org/file2.html> из пакета camel-core. Приложение будет состоять из объекта, запускающего приложение FromFileToFileApp, и класса FromFileToFileRoute, в котором описаны маршруты. Класс с маршрутами можно вынести в отдельный файл.

файл src/main/scala/FromFileToFileApp.scala

**import**org.apache.camel.**CamelContext**

**import**org.apache.camel.main.**Main**

**import**org.apache.camel.scala.dsl.builder.{**ScalaRouteBuilder**, **RouteBuilderSupport**}

**objectFromFileToFileAppextendsAppwithRouteBuilderSupport** {

*//Создаем CamelMain класс и контекст в нем*

**val**mainApp = **newMain**

**val** context = mainApp.getOrCreateCamelContext

*// Привязываемклассысмаршрутами*

mainApp.addRouteBuilder(**newFromFileToFileRoute**(context))

*// Запускаем*

mainApp.run

}

**classFromFileToFileRoute**(context: **CamelContext**) **extendsScalaRouteBuilder**(context) {

*// Читаем содержимое файла в одной кодировке из папки inbox*

"""file:inbox?charset=utf-8""" ==> {

*// Пишем в другой кодировке в директорию outbox*

to ("file:outbox?charset=Windows-1251")

}

}

В классе FromFileToFileRouteне происходит никаких преобразований с содержимым сообщения, отсутствует маршрутизация. Команда extendsApp определяет запускаемый объект для приложения и позволяет производить запуск командой sbtrun. После запуска приложения в папке проекта будут автоматически созданы папки inbox иoutbox. При попадании в директорию inbox файл автоматически считывается иисчезает из папки. Затем он появляется в директории outbox в другой кодировке. При этом в папке inbox в отдельной подпапке будут храниться сообщения, «поглощенные» Camel.

**Сценарий 2: запрос к веб-сервису порасписанию и сохранение сообщения в Redis**

В этомпримере по таймеру будем собирать данные о курсе валют и отправлять в Redis. Как правило, у сообщения есть тело и заголовки. Для того чтобы выполнить действия над сообщением (записать тело и заголовки), используетсяметод process.ВRedis отправка значений производится с помощью пары заголовков:CamelRedis.Key/CamelRedis.Value.  
По умолчанию выполняется команда setCamelRedis.KeyCamelRedis.Value.  
Таким образом, нам необходимо извлечь тело сообщения, которое возвращает HTTP-запрос, и сделать его заголовком "CamelRedis.Value"  
Ключ будем генерировать уникальный — текущее время в миллисекундах.

**import**org.apache.camel.{**Exchange**, **CamelContext**}

**import**org.apache.camel.main.**Main**

**import**org.apache.camel.scala.dsl.builder.{**ScalaRouteBuilder**, **RouteBuilderSupport**}

**import**org.springframework.data.redis.serializer.**StringRedisSerializer**

**objectFromHTTPToRedisAppextendsAppwithRouteBuilderSupport**{*//*

**val**mainApp = **newMain**

*//ПрописываемвместостандартногокастомныйstringSerializerдляRedis*

mainApp.bind("stringSerializer",**newStringRedisSerializer**)

**val** context = mainApp.getOrCreateCamelContext

mainApp.addRouteBuilder(**newFromHTTPToRedisRoute**(context))

mainApp.run

}

**classFromHTTPToRedisRoute** (context: **CamelContext**) **extendsScalaRouteBuilder**(context) {

*//По таймеру, раз в минуту, обращаемся к HTTP-сервису*

"""quartz:timerName?cron=0+0/1+\*+\*+\*+?""" ==> {

*// Простое сообщение, добавленное в лог*

log("Запрос к сервису")

*// Запрос к сервису*

to("http://www.google.com/finance/info?q=CURRENCY%3aUSDRUB")

*// Создаем пару «ключ-значение» для Redis, записываем в хедер*

process((exchange: **Exchange**) => {

exchange.getOut.setHeader("CamelRedis.Key",**System**.currentTimeMillis())

exchange.getOut.setHeader("CamelRedis.Value",exchange.getIn.getBody(classOf[**String**]))

})

*// Логированиекак endpointпозволяет просмотреть сообщение и его атрибуты*

*// В данном примере тело сообщения будет пустоым (Body: [Bodyisnull]])*

to("log:FromHTTPToRedisApp")

*// Отправляем данные в Redis*

*// #stringSerializer - объявленный нами ранее кастомныйсериалайзер*

to("""spring-redis://172.16.7.58:6379?serializer=#stringSerializer""")

}

}

Чтобы сохранять сообщенияв Redis с удаленного хоста, может понадобиться разрешение. Для получения доступа можно выполнить в консоли Redis на хосте, где запущено хранилище,следующую команду:

CONFIG SET protected-modeno

**Сценарий 3: реализация веб-сервиса с использованием Jetty, который возвращает сообщение в зависимости от параметра get запроса**

Добавить пару скринов

В этомпримере с помощью компонента Jetty реализуем простой HTTP-сервер, который получает GET-запрос с некоторым параметром UUID и возвращает значение этого параметра, оборачивая в простой xml. Для сопоставления с образцом значений передаваемого параметра используем метод паттерн-матчинга. Добавим немного паттерн-матчинга, чтобы отдавать значение в зависимости от наличия и формата передаваемого параметра.Предполагается, что параметр, UUID, должен передаваться в соответствующем формате.

**objectJettyAppextendsAppwithRouteBuilderSupport**{

**val**mainApp = **newMain**

**val** context = mainApp.getOrCreateCamelContext

mainApp.addRouteBuilder(**newJettyRoute**(context))

mainApp.run

}

**classJettyRoute**(context: **CamelContext**) **extendsScalaRouteBuilder**(context) {

*// Определяем порт и адрес сервиса*

"""jetty:http://0.0.0.0:1234/myapp/myservice""" ==> {

delay(2 seconds)

process((exchange: **Exchange**) => {

*// Извлекаем значение параметра UUIDиз GET-запроса к сервису*

**val**uuidParam = exchange.getIn.getHeader("uuid")

*// Определяемпаттерндляпараметра*

**val** pattern = """[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12}""".r

*// Возвращаем ответ в зависимости от извлеченного значения*

**defresponseText** = uuidParam**match** {

**case**null =>"Uuid parameter not found"

**case**pattern() =>s"$uuidParam"

**case** \_ =>s"Uuid parameter format is not valid"

}

*// Определяем тип возвращаемого контента как XML*

exchange.getOut().setHeader(**Exchange**.**CONTENT\_TYPE**,"text/xml; charset=utf-8")

*// ВозвращаемXMLсответом.*

exchange.getOut().setBody(<uuid>{responseText}</uuid>)

*//Вариант отправки параметра как строки s"<uuid>$responseText</uuid>" тоже рабочий*

})

}

}

Примеры запросов для проверки:  
<http://localhost:1234/myapp/myservice?uuid=2a577d52-e5a1-4da5-96e5-bdba1f68e6f1>;  
<http://localhost:1234/myapp/myservice?uuid=123>;  
<http://localhost:1234/myapp/myservice>;  
<http://localhost:1234/myapp/myservice?guid=2a577d52-e5a1-4da5-96e5-bdba1f68e6f>.

Формирование ответа реализуем в соответствии с пожеланиями.  
Реализация на Jetty сделана для примера. В некоторых случаях удобнее использовать другие компоненты для реализации сервиса-заглушки, к примеру, Spray или SoapUI.

**Сценарий 4: чтение сообщения из очереди и отправка сообщения в БД**

Работа с очередями и БД требуетиного подхода по сравнению спримерами, рассмотренными выше. Если в предыдущих сценариях мы проводилинастройку, изменяяпараметрыв строке endpoint, то для работы с очередями и БДнужно заранее создать объект, сделать на его основе компонент, который и будет использоваться в дальнейшем.  
Различие очереди и БД в том, что для БД используем BasicDataSource и создаем dataSourceName, который является частью URI camel-jdbc, а для очереди используем JmsComponent и создаем на его основе новый компонент, с кастомным названием.  
Данный код будет забирать сообщения из очереди, выполнять в БД запрос на добавление уникального идентификатора, времени и тела сообщения.  
Таблицасоздаетсяследующимзапросом:

**CREATETABLE**MESSAGETABLE(

**IDUUIDNOT**NULL PRIMARY **KEY**,

DATETIME **TIMESTAMP**,

**BODY**VARCHAR(65536)

**import**java.text.**SimpleDateFormat**

**import**java.util.{**UUID**, **Date**}

**import**org.apache.camel.component.jms.**JmsComponent**

**import**org.apache.camel.main.**Main**

**import**org.apache.camel.scala.dsl.builder.{**RouteBuilderSupport**, **ScalaRouteBuilder**}

**import**org.apache.camel.{**CamelContext**, **Exchange**}

*// Для создания подключения к БД необходим BasicDataSource*

**import** org.apache.commons.dbcp2.**BasicDataSource**

*// Для работы с месседж-брокером импортируемсоответствующий ConnectionFactory класс*

**import**org.apache.activemq.**ActiveMQConnectionFactory**

**objectFromMQToDBAppextendsAppwithRouteBuilderSupport** {

**val**mainApp = **newMain**

*// Для работы с БД создаемобъект и передаемему свойства соединения*

**val** ds = **newBasicDataSource**

ds.setDriverClassName("org.h2.Driver")

ds.setUrl("jdbc:h2:./h2db")

*// ДобавляемБДв приложение, далее в названии получателя будем указывать имя"h2db"*

mainApp.bind("h2db",ds)

*//Для работы с очередью создаем MQConnectionFactory*

**val**cf= **newActiveMQConnectionFactory**("tcp://192.168.3.38:61616")

*// создадим компонент для работы с очередью*

mainApp.bind("amq-jms", **JmsComponent**.jmsComponentAutoAcknowledge(cf))

**val** context = mainApp.getOrCreateCamelContext

mainApp.addRouteBuilder(**newFromMQToDBAppRoute**(context))

mainApp.run

}

*// Данный класс реализует чтение сообщения из очереди и запись сообщенияв БД*

**classFromMQToDBAppRoute**(context: **CamelContext**) **extendsScalaRouteBuilder**(context) {

*// Читаем сообщение из очереди. Компонент называется также, как мы его назвали ранее - "amq-jms", имя очереди передается как параметр*

*// Для каждого брокера необходимо создавать свой компонент*

"""amq-jms:queue:TESTQ""" ==> {

process((exchange: **Exchange**) => {

*// Генерируеммuuid, дату/время и извлекаем тело сообщения*

**val**uuid = **UUID**.randomUUID

**val** time = **newSimpleDateFormat**("yyyy-MM-ddHH:mm:ss").format(**newDate**())

**val**messageBody = exchange.getIn.getBody(classOf[**String**])

*// Формируемзапросспараметрами*

exchange.getOut.setBody(s"INSERT INTO PUBLIC.MESSAGETABLE (ID, DATETIME, BODY) VALUES('$uuid', '$time', '$messageBody')")

})

*// Отправляем запрос в БД*

*// Компонент называется jdbc, далее указываемконкретный DataSource*

to("jdbc:h2db")

}

}

Следует помнить о том, что при попытке записи в БД сообщения больше длины поля (у нас 65536)возникнет ошибка. Ее можно решить, сокративсообщениедо нужного размера либо добавив код errorHandler(deadLetterChannel("file:error")), который будет отправлять сообщения, приводящие к ошибкам, в папку error.

Для других БД нужнодобавить библиотеку в build.sbt, определить имя класса драйвера иURL.Возможно,понадобится указать другие свойства подключения, к примеру, имя пользователя и пароль.  
Например, для отправки сообщения вPostgres потребуется указать следующие реквизиты:

**val** ds = **newBasicDataSource** {

setDriverClassName("org.postgresql.Driver")

setUrl(conf.getString("jdbc:postgresql://myhost:5432/mydb"))

setUsername(conf.getString("myusername"))

setPassword(conf.getString("mypassword"))

}

**build**.sbt

libraryDependencies += "org.postgresql" % "postgresql" % "9.4.1207"

Сочередяминесколькосложнее.  
Для некоторых библиотеки не открыты для доступа в репозиториях и используются файлы \*.jar, записываемые в папку lib.  
В любом случае, нужно получить connectionfactory.  
К примеру, код, обеспечивающий соединение для IBM Websphere MQ, будет таким:

**val**cf = **newMQQueueConnectionFactory** {

setHostName("myhost")

setPort(1414)

setTransportType(1)

setQueueManager("myqmname")

setChannel("mychannel")

}

Для OracleWebLogicJMS будетеще интереснее.

Если взять за основу инструкцию https://blogs.oracle.com/soaproactive/entry/how\_to\_create\_a\_simple, то объявление компонента connectionfactoryбудет таким:

**val**env = **new**util.**Hashtable**[**String**, **String**]

env.put(javax.naming.**Context**.**INITIAL\_CONTEXT\_FACTORY**, "weblogic.jndi.WLInitialContextFactory")

env.put(javax.naming.**Context**.**PROVIDER\_URL**, "t3://myhost:7001")

**val**ic: **InitialContext** = **newInitialContext**(env)

**val**connectionFactory = ic.lookup("jms/TestConnectionFactory").asInstanceOf[**QueueConnectionFactory**]

*// гдеjms/TestConnectionFactory - jndiдляConnectionFactory"*

mainApp.bind("ora-jms", **JmsComponent**.jmsComponentAutoAcknowledge(connectionFactory))

а endpoint URI будеттакогоформата: """ora-jms:queue:./TestJMSModule!TestJMSQueue"""

где ./ обозначаеттекущийсервер, "TestJMSModule" JNDI – имямодуля"TestJMSQueue" - JNDI имяочереди.

**Сценарий5:**  
маршрутизация по содержимому файла  
В данном примере рассмотриммаршрутизацию сообщенияв зависимости от его содержимого.  
Предположим, что на входе имеетсяxml-сообщение, маршрутизация которогозависитот значения элемента "То".  
К примеру, нам известно, что сообщение, в котором элемент «То» имеет значение «ActiveMQ»,нужно отправить в очередь;сообщение, в котором «То» означает «H2», требуется обработать каким-то образом и отправить в БД; а сообщение, где в строке «То» указан«someAdress»,обработать еще каким-то образом.  
В сообщение будет добавлен заголовок "Destination" с именем endpoint, в который надо будет отправить сообщение дальше.  
Если возникнет ошибка при обработке сообщения или в таблице маршрутизации не будет соответствующего значения, то отправляем сообщение в "direct:trash"  
??? — конструкция скала, которая позволяет заменить несуществующий блок кода для успешной компиляции. Предполагается замена на рабочий код.

**import**org.apache.camel.{**Exchange**, **CamelContext**}

**import**org.apache.camel.scala.dsl.builder.**ScalaRouteBuilder**

**import** scala.xml.**XML**

**classContentOrientedRouting**(context: **CamelContext**) **extendsScalaRouteBuilder**(context) {

*// При ошибках обработки сообщения отправляем сообщения в "direct:trash"*

errorHandler(deadLetterChannel("direct:trash"))

*// ОписываемтаблицумаршрутизацииввидеMap*

**val**destMap = **Map**(

"ActiveMQ" ->"jms-amq:queue:inbox",

"H2" ->"direct:h2db",

"someAdress" ->"direct:outbox")

*// Выносимобработкувотдельнуюфункцию*

**val**addRoutingAction = (exchange: **Exchange**) => {

*// Получим значение тега "To" из XML, который пришла на вход*

**val** body = exchange.getIn.getBody(classOf[**String**])

**val**xmlBody = **XML**.loadString(body)

**val**toValue = (xmlBody \\ "To").text

*// Получимимя endpoint, еслитакогозначениянет - отправляемв "direct:trash"*

**val**dest = destMap.getOrElse(toValue,"direct:trash")

*// Устанавливаемзначениезаголовка*

exchange.getOut.setHeader("Destination", dest)

}

"""direct:inbox1""" ==> {

process(addRoutingAction)

*// Извлекаемиззаголовка "Destination" endpointиотправляемтудасообщение*

recipients(\_.in("Destination"))

}

*// Описываемлогикудляразных endpoint*

"""jms-amq:queue:inbox""" ==> {???}

"""direct:h2db""" ==> {

process((exchange: **Exchange**) => {???})

to ("jdbc:h2db")

}

"""direct:outbox""" ==> {

*// Параллельная отправка сообщения в файл и в лог*

to("file:someFile", "log:Somelog")

}

"""direct:trash""" ==> {???}

}

Опциональновозможнодобавлениекомпонентадляконфигурацииприложения, <https://github.com/typesafehub/config>,чтобынезашиватьпараметрыподключениявкод, ахранитьвjson-файле  
вbuild.sbtдобавляем  
libraryDependencies += "com.typesafe" % "config" % "1.3.0"  
впапкеsrc/main/resourcesсоздаемфайлapplication.conf  
Болееподробноостанавливатьсянаэтомвопросенебудем.  
Запуск приложения выполняется командой sbtrun.  
Также возможно создание файла \*.jarс помощью плагина sbt-assembly <https://github.com/sbt/sbt-assembly> для запуска приложения командой jar camelapp.jar.  
Для запуска приложения в фоне удобно использовать приложение nohup.  
Создаем скрипт для запуска в папке, которая входит в $PATH? К примеру, в /usr/local/bin/ для запуска по имени скрипта  
/usr/local/bin/camelstart

**#!/bin/bash**

/usr/bin/nohup java -jar /opt/camelapp.jar&

Дляостановки  
/usr/local/bin/camelstop

**#!/bin/bash**

pkill -f camelapp

Запускприложениявыполняетсяделаетсякомандойcamelstart, остановка — командойcamelstop. Также возможно реализовать автозапуск приложения при старте машины, как сервис.

**Заключение**

Итак, мы рассмотрели несколько примеров использования ApacheCamelдля создания тестовых заглушек, а также компонентов информационной системы.Можновыделить некоторые плюсы и минусы данного инструмента.

Плюсы:  
быстраяреализация приложений;  
большое количество готовых компонентов;  
многопоточность, параллельная обработка сообщений из коробки;  
возможность выбрать удобный язык описания–XML, DSL.

Минусы:  
У каждого компонента своя логика работы, иногда требуется время на понимание;  
Существует порог входа;

Кроме того, поскольку ApacheCamelработает на JVM, приложениям, созданным на его основе, присущивсе плюсы и минусы этой платформы.   
СразвитиемApacheCamelкод устаревает, и в новых версияхпомечается как @Deprecated.

Опыт использования ApacheCamelв нашей компании показал, чтоэтот фркеймворкпозволяет реализовывать заглушки, компоненты интеграции, а иногда и нагрузочные тесты быстро и эффективно.