**Создание заглушек для интеграционного тестирования на Apache Camel и Scala DSL**

**Создание заглушек на Apache Camel**

**Создание заглушек на Apache Camel для интеграционного тестирования**

**Заглушки на Apache Camel и Scala DSL для интеграционного тестирования**

**Интеграционное тестирование: создание заглушек на Apache Camel и Scala DSL**

**Apache Camel + Scala DSL = быстрые заглушки для тестирования**

В статье рассмотрены примеры использования Apache Camel 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 на Java, Scala или JMeter. Но нам нужна система, которая постоянно работает, отвечает на запросы и не требует действий со стороны тестировщика — запустил и работает. Для решения этой задачи мы использовали приложение, основанное на фреймворке Apache Сamel.

Мыимсценарий 1:ч –сценарий :з-сценарий :рJGET-сценарий :чсценарий :мяЕсть ещесценарийPOST- (; ).

**Коротко об инструментах, используемых для решения задачи**

Apache Camel ( <http://camel.apache.org/> ) — java-фреймворк, предназначенный для реализации обмена сообщениями между компонентами информационной системы. Реализует так называемые Enterprise Integration Patterns (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 реализует обмен между производителем сообщения (Producer) и потребителем сообщения (Consumer).

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.

**Подготовительный этап**  
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**(

*// Компоненты для Camel*

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

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

*// Для каждого компонента своя зависимость*

“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,

*//Добавим логирование*

“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”,

*// Драйвер БД H2*

“com.h2database” % “h2” % “1.4.192”,

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

*// Драйвер для брокера activemq*

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

)

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

<configuration>

<appender name="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**}

**object** **FromFileToFileApp** **extends** **App** **with** **RouteBuilderSupport** {

*//Создаем Camel Main класс и контекст в нем*

**val** mainApp = **new** **Main**

**val** context = mainApp.getOrCreateCamelContext

*// Привязываем классы с маршрутами*

mainApp.addRouteBuilder(**new** **FromFileToFileRoute**(context))

*// Запускаем*

mainApp.run

}

**class** **FromFileToFileRoute**(context: **CamelContext**) **extends** **ScalaRouteBuilder**(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 FromFileToFileRoute не происходит никаких преобразований с содержимым сообщения, отсутствует маршрутизация. Команда extends App определяет запускаемый объект для приложения и позволяет производить запуск командой sbt run. После запуска приложения в папке проекта будут автоматически созданы папки inbox и outbox. При попадании в директорию inbox файл автоматически считывается и исчезает 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 по расписанию и сохранение сообщения в Redis**

В этом примере 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), используется метод process. В Redis отправка значений производится с помощью пары заголовков: CamelRedis.Key/CamelRedis.Value.  
По умолчанию выполняется команда set CamelRedis.Key CamelRedis.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**

**object** **FromHTTPToRedisApp** **extends** **App** **with** **RouteBuilderSupport**{*//*

**val** mainApp = **new** **Main**

*//Прописываем вместо стандартного кастомный stringSerializer для Redis*

mainApp.bind("stringSerializer",**new** **StringRedisSerializer**)

**val** context = mainApp.getOrCreateCamelContext

mainApp.addRouteBuilder(**new** **FromHTTPToRedisRoute**(context))

mainApp.run

}

**class** **FromHTTPToRedisRoute** (context: **CamelContext**) **extends** **ScalaRouteBuilder**(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**]))

})

*// Логирование как endpoint позволяет просмотреть сообщение и его атрибуты*

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

to("log:FromHTTPToRedisApp")

*// Отправляем данные в Redis*

*// #stringSerializer - объявленный нами ранее кастомный сериалайзер*

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

}

}

Чтобы сохранять сообщения в 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-mode no

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

Добавить пару скринов

В этом примере 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.

**object** **JettyApp** **extends** **App** **with** **RouteBuilderSupport**{

**val** mainApp = **new** **Main**

**val** context = mainApp.getOrCreateCamelContext

mainApp.addRouteBuilder(**new** **JettyRoute**(context))

mainApp.run

}

**class** **JettyRoute**(context: **CamelContext**) **extends** **ScalaRouteBuilder**(context) {

*// Определяем порт и адрес сервиса*

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

delay(2 seconds)

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

*// Извлекаем значение параметра UUID из GET-запроса к сервису*

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

*// Определяем паттерн для параметра*

**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*

**def** **responseText** = 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")

*// Возвращаем XML с ответом.*

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 требует иного подхода по сравнению с примерами, рассмотренными выше. Если в предыдущих сценариях мы проводили настройку, изменяя параметры в строке endpoint, то для работы с очередями и БД нужно 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.  
Таблица создается следующим запросом:

**CREATE** **TABLE** MESSAGETABLE(

**ID** **UUID** **NOT** 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 импортируем соответствующий ConnectionFactory класс*

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

**object** **FromMQToDBApp** **extends** **App** **with** **RouteBuilderSupport** {

**val** mainApp = **new** **Main**

*// Для работы с БД создаем объект и передаем ему свойства соединения*

**val** ds = **new** **BasicDataSource**

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

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

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

mainApp.bind("h2db",ds)

*//Для работы с очередью создаем MQConnectionFactory*

**val** cf= **new** **ActiveMQConnectionFactory**("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(**new** **FromMQToDBAppRoute**(context))

mainApp.run

}

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

**class** **FromMQToDBAppRoute**(context: **CamelContext**) **extends** **ScalaRouteBuilder**(context) {

*// Читаем сообщение из очереди. Компонент называется так же, 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**) => {

*// Генерируемм uuid, дату/время и извлекаем тело сообщения*

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

**val** time = **new** **SimpleDateFormat**("yyyy-MM-dd HH:mm:ss").format(**new** **Date**())

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

*// Формируем запрос с параметрами*

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

})

*// Отправляем запрос в БД*

*// Компонент называется jdbc, далее указываем конкретный 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, сократив сообщение до нужного размера либо добавив код errorHandler(deadLetterChannel("file:error")), который будет отправлять сообщения, приводящие к ошибкам, в папку error.

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

**val** ds = **new** **BasicDataSource** {

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"

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

**val** cf = **new** **MQQueueConnectionFactory** {

setHostName("myhost")

setPort(1414)

setTransportType(1)

setQueueManager("myqmname")

setChannel("mychannel")

}

Для Oracle WebLogic JMS будет еще интереснее.

Если взять за основу инструкцию https://blogs.oracle.com/soaproactive/entry/how\_to\_create\_a\_simple, то объявление компонента connection factory будет 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** = **new** **InitialContext**(env)

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

*// где jms/TestConnectionFactory - jndi для ConnectionFactory"*

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

а endpoint URI будет такого формата: """ora-jms:queue:./TestJMSModule!TestJMSQueue"""

где ./ обозначает текущий сервер, "TestJMSModule" JNDI – имя модуля "TestJMSQueue" - JNDI имя очереди.

**Сценарий 5:**  
маршрутизация по содержимому файла  
В данном примере рассмотрим маршрутизацию сообщения в зависимости от его содержимого.  
Предположим, что на входе имеется xml-сообщение, маршрутизация которого зависит от 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**

**class** **ContentOrientedRouting**(context: **CamelContext**) **extends** **ScalaRouteBuilder**(context) {

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

errorHandler(deadLetterChannel("direct:trash"))

*// Описываем таблицу маршрутизации в виде Map*

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

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

"H2" -> "direct:h2db",

"someAdress" -> "direct:outbox")

*// Выносим обработку в отдельную функцию*

**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

*// Получим имя endpoint, если такого значения нет - отправляем в "direct:trash"*

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

*// Устанавливаем значение заголовка*

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

}

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

process(addRoutingAction)

*// Извлекаем из заголовка "Destination" endpoint и отправляем туда сообщение*

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

}

*// Описываем логику для разных 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""" ==> {???}

}

Опционально возможно добавление компонента для конфигурации приложения, <https://github.com/typesafehub/config>, чтобы не зашивать параметры подключения в код, а хранить в json-файле  
в build.sbt добавляем  
libraryDependencies += "com.typesafe" % "config" % "1.3.0"  
в папке src/main/resources создаем файл application.conf  
Более подробно останавливаться на этом вопросе не будем.  
Запуск приложения выполняется командой sbt run.  
Также возможно создание файла \*.jar с 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&

Для остановки  
/usr/local/bin/camelstop

**#!/bin/bash**

pkill -f camelapp

Запуск приложения выполняется делается командой camelstart, остановка — командой camelstop. Также возможно реализовать автозапуск приложения при старте машины, как сервис.

**Заключение**

Итак, мы рассмотрели несколько примеров использования Apache Camel для создания тестовых заглушек, а также компонентов информационной системы. Можно выделить некоторые плюсы и минусы данного инструмента.

Плюсы:  
быстрая реализация 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 Apache Camel работает на JVM, приложениям, созданным на его основе, присущи все плюсы и минусы этой платформы.   
С развитием Apache Camel код устаревает, и в новых версиях помечается как @Deprecated.

Опыт использования Apache Camel в нашей компании показал, что этот фркеймворк позволяет реализовывать заглушки, компоненты интеграции, а иногда и нагрузочные тесты быстро и эффективно.