В данном материале будут рассмотрены примеры использования связки Apache Camel и ScalaDSL для реализации заглушек информационных систем при тестировании.  
Часто возникает необходимость с эмулировать работу какой-либо системы для интеграционного тестирования, сделать заглушку или написать простой компонент интеграции.  
Цель - краткое описание с примерами для быстрой реализации заглушек систем.  
Apache camel ( http://camel.apache.org/ )- java фреймворк, предназначенный для реализации обмена сообщениями между компонентами информационной системы. Реализует т.н. EIP - Enterprise Integration Patterns.  
Существует возможность работы с файлами, БД, менеджерами очередей, веб-сервисами, и другими компонентами - их около 240 на странице проекта http://camel.apache.org/component.html  
Компонент может реализовывать как поставщика сообщения(Producer), так и получателя(Consumer).  
  
Общий смысл таков:  
1) Описываем источник сообщения(файл, очередь, БД, сервис, таймер и т.п.);  
2) Описываем правила преобразования данных и форматов;  
3) Описываем получателя (получателей) 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.  
Рассмотрим подробнее на примерах.  
  
Для описания правил маршрутизации и преобразования сообщений используются различные языки( http://camel.apache.org/languages.html ). Мы будем использовать Scala DSL (http://camel.apache.org/scala-dsl-eip.html), систему сборки проекта sbt.  
  
Будет рассмотрено 4 сценария, с возраствнием сложности:  
1) Чтение файлов в одной кодировке, запись в другой;  
2) Запрос к веб сервису по-расписанию и сохранение сообщения в Redis;  
3) Реализация веб-сервиса с использованием jetty, который возвращает сообщение в зависимости от параметра get запроса;  
4) Чтение сообщения из очереди и отправка сообщения в БД;

5) Пример маршрутизации по содержимому файла.  
Также существует отличный пример с чтением сообщения из файла и отправкой его http 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  
  
Подготовительные работы  
Создадим проект в idea, пример - ???  
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 логгирование* **"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"**)

Добавим в папку src/main/resources файл logback.xml, в котором можем установить уровень логгирования 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>  
  
Без этого файла по-умолчанию работает уровень DEBUG - сыпется слишком много информации.

Пример 1

Чтение файлов в одной кодировке, запись в другой.

*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}  
  
*//extends App определяет запускаемый объект для приложения и позволяет производить запуск командой "sbt run"***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 данном классе не происходит никаких преобразований с содержимым сообщения, отсутствует маршрутизация. После запуска приложеия командой "sbt run" в папке проекта будут автоматически созданы папки "inbox", "outbox". При попадании в директорию "inbox" файл автоматически считывается - исчезает из папки, и появляется в директории "outbox" в другой кодировке. При этом в папке "inbox" в отдельной подпапке будут сообщения, «поглощенные» camel.*

Пример 2

Запрос к веб сервису по-расписанию и сохранение сообщения в Redis.

Добавить скриншот, объяснить сериалайзер *'CONFIG SET protected-mode no'*

*В данном примере рассмотрим более сложный случай. По таймеру будем собирать даные о курсе доллара и отправлять в redis. Нам понадобится некоторая логика для отправки сообщения в Redis. В общем случае, у сообщения есть тело и заголовки. Для того, чтобы выпонить действия над сообщением(установить тело сообщения и заголовки), существует метод process. Для 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 просто генерировать уникальный*

**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]))  
 })  
 *// Логгирование как endpoint позволяет просмотреть сообщение и его атрибуты  
 // В данном примере тело сообщения будет пусто (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"""**)  
  
 }  
}

CONFIG SET protected-mode no

Пример 3

Реализация веб-сервиса с использованием jetty, который возвращает сообщение в зависимости от параметра get запроса

Добавить пару скринов

*В данном примере с помощью компонента jetty реализуем простой http сервер, который получает get запрос с параметром и отдает значение параметра. Добавим немного патерн-матчинга, чтобы отдавать значение в зависимости от наличия и формата передаваемого параметра. Предполагается, что параметр, uuid, должен передаваться в соответствующем формате.***object** JettyApp **extends** RouteBuilderSupport{ *//App with* **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*

*соответственно, формирование ответа можно реализовать в соответствии с пожеланиями.*

*Пример 4*

Чтение сообщения из очереди и отправка сообщения в БД.

*Работа с очередями и БД была выделена в отдельный пример. Проблема в том, что требуется особая их настройка. Если в предыдущих примерах настройка проводилась с помощью параметров в строке endpoint, то здесь нужно заранее создать объект, сделать на его основе компонент и использовать далее.  
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 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 нужно импортировать соответствующий ConnectionFactory класс***import** org.apache.activemq.ActiveMQConnectionFactory  
  
  
**object** FromMQToDBApp **extends** App **with** RouteBuilderSupport {  
 **val** *mainApp* = **new** Main  
 *// Для работы с БД необходимо создать объект и передать ему свойства соединения* **val** *ds* = **new** BasicDataSource  
 *ds*.setDriverClassName(**"org.h2.Driver"**)  
 *ds*.setUrl(**"jdbc:h2:./h2db"**)  
 *// Добавим бд в приложение, далее в названии получателя будем использовать "h2db"  
 mainApp*.bind(**"h2db"**,*ds*)  
 *//Для работы с очередью создадим 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 его в БД***class** FromMQToDBAppRoute(context: CamelContext) **extends** ScalaRouteBuilder(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) => {  
 *// Генерим 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 подготовленный запрос в бд  
 // Компонент называется jdbc, далее указывается конкретный DataSource* to(**"jdbc:h2db"**)  
  
 }  
}

При попытке записи в БД сообщения большедлины поля (у нас *65536*) – возникнет ошибка. Ее можно решить либо обрезая тело до нужного размера, либо добавив *errorHandler(deadLetterChannel("file:error")), который будет отправлять сообщения, приводящие к ошибкам. В папку «error».*

*Для других БД нужно, соответственно, добавить библиотеку в build.sbt, определить имя класса драйвера, url, возможно понадобятся другие свойства подключения, к примеру, Имя пользователя и пароль.  
Для Postgres:  
  
 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 \*.jar файлы, подкладываемые в папку lib проекта.  
В любом случае, нужно получить connection factory  
К примеру, для IBM Websphere MQ  
 val cf = new MQQueueConnectionFactory {  
 setHostName("myhost")  
 setPort(1414)  
 setTransportType(1)  
 setQueueManager("myqmname")  
 setChannel("mychannel")  
 }  
  
 Для oracle weblogic jms еще интереснее  
 Если взять за основу инструкцию https://blogs.oracle.com/soaproactive/entry/how\_to\_create\_a\_simple  
то объявление компонента будет такое:  
  
 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 удет такого формата: """ora-jms:queue:./TestJMSModule!TestJMSQueue"""  
 где ./ обозначает текущий сервер, "TestJMSModule" JNDI имя модуля "TestJMSQueue" - JNDI имя очереди*

Пример 5

маршрутизация по содержимому файла.

*/\*  
В данном примере будет рассмотрена маршрутизация в зависимости от содержимого сообщения.  
Мы предполагаем, что на входе будет xml сообщение, которое брабатывается по-разному в зависимости от значения элемента "То"  
К примеру, нам известно, что сообщение, в котором <To>ActiveMQ</To> - нужно отправить в очередь, а <To>H2</To> - обработать каким-то образом и отправить в БД, <To>someAdress</To> - 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 блок кода для компиляции. Предполагается замена на рабочий код.  
\*/***import** org.apache.camel.{Exchange, CamelContext}  
**import** org.apache.camel.scala.dsl.builder.ScalaRouteBuilder  
**import** scala.xml.XML  
  
*/\*\*  
 \* Created by smakhetov on 11.07.2016.  
 \*/***class** ContentOrientedRouting(context: CamelContext) **extends** ScalaRouteBuilder(context) {  
  
 *// При ошибках обработки сообщения отправляем сообщения в "direct:trash"* errorHandler(deadLetterChannel(**"direct:trash"**))  
  
 *// Опишем таблицу маршрутизации в виде Map* **val** *destMap* = *Map*(  
 **"ActiveMQ"** -> **"jms-amq:queue:inbox"**,  
 **"H2"** -> **"direct:h2db"**,  
 **"someAdress"** -> **"direct:outbox"**)  
 *// Вынесем 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 коде, 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 не будем.

Запуск приложения выполняется командой sbt run

Также возможно создание jar файла с помощью плагина sbt-assembly <https://github.com/sbt/sbt-assembly> для запуска командой java –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

т.о. запуск приложения делается командой camelstart, остановка - camelstop

Таким образом, были рассмотрены примеры использования Camel. Надеюсь, кому-нибудь данный материал будет полезен и сэкономит время на поиски и реализацию инструментов.