codecentric Blog (https://blog.codecentric.de/)

Overview (https://blog.codecentric.de/en/category/architecture/)

X.509 client certificates with Spring Security (https://blog.codecentric.de/en/2018/08/x-509-client-certificates-with-spring-security/)

Easy integration between services with Apache Camel (https://blog.codecentric.de/en/2018/08/easy-integration-between-services-with-apache-camel/)

08/20/18 by **Jan Martijn Roetman (https://blog.codecentric.de/en/author/jan-martijn-roetman/)**

No Comments (https://blog.codecentric.de/en/2018/08/easy-integration-between-services-with-apache-camel/#comments)

For a couple of months now I have been working on an application that uses Apache Camel. I am not sure if it's a good choice for this application because it does not deal with many sources of information. But I am convinced that Apache Camel can provide easy-to-read integration code and it's

a good choice for some services in a microservices architecture. The Apache Camel project is already running for some time, and I wonder: is it ready for the future? First I will explain a bit what I think Apache Camel is and why it is useful. I will also give some code examples.

What is Apache Camel?

Apache Camel is a framework full of tools for routing data within an application. It is a framework you use when a full-blown Enterprise Server Bus is not (yet) needed. It focusses on getting different kinds of messages from different kinds of sources to their destination.
Using Apache Camel intensively in an application means it becomes message-based. It provides an implementation of the Enterprise Integration Patterns, which are described in the book 'Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions', using a domain-specific language.

Apache Camel's main building block is a 'Route' which contains flow and integration logic. In the route you can specify the sources and endpoints using the DSL. You can also define which transformations need to be done during the trip from source to endpoint. In your route you can define URIs to receive data provided by different sources, transport protocols or messaging models and also send data to them. For example, HTTP, JMS, Amazon's SQS, Docker, MQTT and many more. Also Beans can be endpoints, but cannot be defined as a source. Apache Camel in general works nicely together with Spring. A Camel Spring Boot autoconfiguration and starter module are available.

Why use Apache Camel?

It is quite difficult to explain why one would need Apache Camel, but I will try. I think Apache Camel is a great tool when your application receives data from many different sources. At a certain moment, when adding more sources, the code is getting littered with various client libraries and custom code that does message transformation, which is when it is maybe time to look into Apache Camel. The DSL provides a clear way to define the integration and transformation required for the data from these sources. Besides, you can easily set up in-memory queueing to prevent overloading of certain calls in the application using for example the SEDA component. SEDA creates a pool of threads to process incoming messages. Also, Direct VM and VM components are provided to send messages to applications running on the same Java virtual machine. In the DSL you have the 'choice' construct that enables conditional routing. This means you can determine if a message for example needs to be sent to a specific endpoint.

The framework also provides one set of concepts and models to argue about integration issues. The same concepts of endpoint and consumer can be used when getting data from an MQTT topic or when files are dropped in a directory or when processing a REST request. While Apache Camel is expressive and declarative, it does add complexity. A language is introduced in the codebase that a lot of developers are not familiar with.

Some examples

A simple pseudo-code example:

```
from(source)
  .choice()
    .when(condition).to(endpoint)
  .otherwise()
    .to(anotherEndpoint)
  .end();
```

More extensive example:

```
from("file:" + getDirectory() + "?move=.done")
   .routeId("extensiveRouteId")
   .routePolicyRef("cronPolicy")
   .unmarshal("dataFormatter")
   .process("Processor1")
   .process("Processor2")
   .to("bean:outputBean?method=process(${body},${header." + fieldName + "})")
```

In the second example, the route listens to a directory and every file there is picked up. When finished, the file is moved to the .done sub directory. The route policy defines when a route is active and the unmarshal defines how the file contents are transformed to a new format like a bean. The process call enables you to get the message in form of an 'Exchange' object in a Processor where you can read it and change it.
At the end, the message is sent to a method 'process' of the bean with the name 'outputBean'. The two arguments of the method are provided using the 'Simple Expression Language' which is part of Camel. The body is just the main message content and the header provides metadata which often is automatically provided by a component. Like the 'CamelFileName' for the 'file:' component.

Below I give an example how you could create an integration test for a Route.

```
@RunWith(CamelSpringRunner.class)
@ContextConfiguration(loader = AnnotationConfigContextLoader.class)
public class SplitRouteIT
{
        public static final String MOCK_RESULT = "mock:result";

        @Produce(uri = DIRECT_SPLIT)
        private ProducerTemplate template;

        @Autowired
        private CamelContext camelContext;

        @EndpointInject(uri = MOCK_RESULT)
        protected MockEndpoint mockEndpoint;

        @Before
        public void setup() throws Exception
        {
                AdviceWithRouteBuilder builder = new AdviceWithRouteBuilder()
                {
                        @Override
                        public void configure() throws Exception
                        {
                                weaveByToString("To[" + DIRECT_SENDER + "]").repla
                        }
                };
                camelContext.getRouteDefinition(SplitRoute.ROUTE_ID).adviceWith(ca
        }

        @Test
        @DirtiesContext
        public void shouldSplitMessages() throws Exception
        {
                mockEndpoint.expectedBodiesReceived(
                        "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcde
                        "ijklmnopqrstuvwxyz1",
                        "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcde
                        "ijklmnopqrstuvwxyz2");
                template.sendBody(SplitRoute.DIRECT_SPLIT, "abcdefghijklmnopqrstuv
                template.sendBody(SplitRoute.DIRECT_SPLIT, "abcdefghijklmnopqrstuv
                mockEndpoint.assertIsSatisfied();
        }

        @Test
        @DirtiesContext
        public void shouldSplitMessage() throws Exception
        {
                mockEndpoint.expectedBodiesReceived("abcdefghijklmnopqrstuvwxyzabc
```

```
                template.sendBody(DIRECT_SPLIT, "abcdefghijklmnopqrstuvwxyzabcdefg
                mockEndpoint.assertIsSatisfied();
        }

        @ComponentScan(basePackageClasses = { CamelContext.class, SplitRoute.class
        @Configuration public static class ContextConfiguration
        {
        }
}
```

And the actual route:

```
import org.apache.camel.builder.RouteBuilder;
import org.springframework.stereotype.Component;

@Component
public class SplitRoute extends RouteBuilder
{
        public static final String ROUTE_ID = "SPLIT_ROUTE";
        public static final String DIRECT_SPLIT = "direct:split";
        public static final String DIRECT_SENDER = "direct:sender";

        @Override public void configure() throws Exception
        {
                from(DIRECT_SPLIT)
                        .routeId(ROUTE_ID)
                        .split().method(SplitIterator.class, "splitMessage")
                        .to(DIRECT_SENDER);
        }
}
```

The route tested splits incoming messages in a new message for each 60 characters. The 'direct' scheme used in this example is useful for synchronous communication between routes. An important point is to add the *adviceWith* method which changes the output to mock:result URI. The scheme 'mock' in the URI is required when mocking. The *@DirtiesContext* is needed for the clean-up of the application context after a test.

Camel routes are not always easy to test in my opinion but there are support classes provided for JUnit. Like the 'CamelTestSupport' which provides a 'CamelContext' and a 'ProducerTemplate', the 'ProducerTemplate' is used to provide messages and these can be used as input for a route. Mocking

classes are also provided and there is the *CamelSpringRunner* class for integration tests (Used in the example).

The future

Apache Camel could be very useful in a system with microservices. In this case you have many services working together and Camel can play a role in integration. For example when creating a API Gateway like described in this article: https://developers.redhat.com/blog/2016/11/07/microservices-comparing-diy-with-apache-camel/ (https://developers.redhat.com/blog/2016/11/07/microservices-comparing-diy-with-apache-camel/). The example in the linked article really shows that it's possible to create an elegant solution to do multiple calls to different services and combine the results. It also shows that Apache Camel provides support for circuit breaking like Hystrix. Another nice addition is a component for communicating with a cache provider like Ehcache. For the future of Apache Camel I think it would be benefical to have more components for communication with cloud services. For AWS services, some components are available, but for Microsoft Azure and the Google Cloud platform not so much. Developers are still quite actively committing in the Apache Camel project so I expect more components will become available. An alternative to Apache Camel is for example Spring Integration, which has similar features, but people tend to favor the syntax of Apache Camel. Another alternative is Mule ESB, but this is a more ready-to-use platform than a framework.

Apache Camel looks like a solid framework, with a nice fluent API. It provides support for a lot of data sources. I would suggest using it in a service that is communicating and receiving data from/to a lot of different sources. For example, an API gateway or an aggregator service.

More information about Apache Camel can be found here : http://camel.apache.org/articles (http://camel.apache.org/articles).

Tags

APACHE CAMEL (HTTPS://BLOG.CODECENTRIC.DE/EN/TAG/APACHE-CAMEL/)

MESSAGE ORIENTED (HTTPS://BLOG.CODECENTRIC.DE/EN/TAG/MESSAGE-ORIENTED/)

TESTING (HTTPS://BLOG.CODECENTRIC.DE/EN/TAG/TESTING/)

Jan Martijn Roetman (https://blog.codecentric.de/en/author/jan-martijn-roetman/)

Jan Martijn Roetman is a software craftsman working at codecentric Netherlands. He is a web application developer with experience in Java and JavaScript. He is especially interested in security of applications and artificial intelligence.

(http://www.facebook.com/sharer.php?u=https://blog.codecentric.de/en/2018/08/easy-integration-between-services-with-apache-camel/)

(http://twitter.com/share?url=https://blog.codecentric.de/en/2018/08/easy-integration-between-services-with-apache-camel/&text=Easy+integration+between+services+with+Apache+Camel)

(https://www.linkedin.com/shareArticle?mini=true&url=https://blog.codecentric.de/en/2018/08/easy-integration-between-services-with-apache-camel/)

(http://reddit.com/submit?url=https://blog.codecentric.de/en/2018/08/easy-integration-between-services-with-apache-camel/&title=Easy+integration+between+services+with+Apache+Camel)

Post by **Jan Martijn Roetman**

HACKING

Web application vulnerabilities and how to prevent them (https://blog.codecentric.de/en/2019/02/web-application-vulnerabilities-and-how-to-prevent-them/)

MICROSERVICES

How and when to use JSON Web Tokens for your services (https://blog.codecentric.de/en/2017/08/use-json-web-tokens-services/)

More content about **Architecture**

ARCHITECTURE

The testable Lambda – A lightweight approach with Dependency Injection (https://blog.codecentric.de/en/2019/02/testable-lambda/)

ARCHITECTURE

Reflections on DDD Europe 2019 (https://blog.codecentric.de/en/2019/02/reflections-on-visiting-ddd-europe-2019/)

Comment

Nachricht

Name

E-Mail

SUBMIT

 (https://www.facebook.com/codecentric)  (https://twitter.com/codecentric)

IMPRINT (HTTPS://BLOG.CODECENTRIC.DE/EN/IMPRINT/)

PRIVACY POLICY (HTTPS://WWW.CODECENTRIC.DE/PRIVACY-POLICY/)

CONTACT (HTTPS://WWW.CODECENTRIC.DE/UEBER-CODECENTRIC/KONTAKT/)