Kostenloses eBook

LERNEN apache-camel

Free unaffiliated eBook created from
Stack Overflow contributors.

#apache-camel

Inhaltsverzeichnis

Über ... 1

Kapitel 1: Erste Schritte mit Apache-Kamel ... 2

Bemerkungen ... 2

Examples ... 2

Installation oder Setup ... 2

Maven-Abhängigkeit ... 2

Gradle ... 2

Spring Boot ... 3

Camel-Domänenspezifische Sprache ... 3

Kapitel 2: Integrationstest auf bestehenden Routen mit Apache-Camel und Spring (And DBUnit ... 5

Einführung ... 5

Parameter ... 5

Bemerkungen ... 6

Examples ... 6

Kamelroute Beispiel ... 6

Kamelprozessor Beispiel ... 7

Beispiel für Testklassen der Camel Integration ... 8

Kapitel 3: Pub / Sub mit Camel + Redis ... 12

Bemerkungen ... 12

Examples ... 12

RedisPublisher ... 12

RedisSubscriber ... 12

Abonnentenfederkontext ... 13

Publisher Spring-Kontext ... 13

ManagedCamel ... 14

Credits ... 16

Über

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: apache-camel

It is an unofficial and free apache-camel ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official apache-camel.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Kapitel 1: Erste Schritte mit Apache-Kamel

Bemerkungen

Apache Camel ist ein Framework, das in erster Linie die Lösung von Herausforderungen bei der Unternehmensintegration erleichtert. Im Grunde kann man es sich als Engine Engineer für Routing-Engine vorstellen. Im Wesentlichen können Sie damit Systeme (Endpunkte) über Routen verbinden. Diese Routen akzeptieren Nachrichten, die einen beliebigen Datentyp haben können.

Das Apache Camel-Framework enthält außerdem einen vollständigen Satz von EIP (Enterprise Integration Patterns) wie Splitter, Aggregatoren, inhaltsbasiertes Routing usw. Da das Framework in verschiedenen Java-Anwendungen als Standalone implementiert werden kann, kann es in verschiedenen Anwendungsservern wie WildFly und Tomcat oder auf einem voll entwickelten Enterprise Service Bus als Integrationsframework betrachtet werden.

Um mit dem Framework zu beginnen, müssen Sie es mit einer der folgenden Methoden zu einem Projekt hinzufügen:

1. Maven
2. Gradle
3. Spring Boot
4. Einfache alte JAR-Bibliotheksreferenz, die Ihrem Projekt hinzugefügt wurde.

Examples

Installation oder Setup

Detaillierte Anweisungen zum Hinzufügen der erforderlichen Camel-Abhängigkeiten.

Maven-Abhängigkeit

Eine der häufigsten Möglichkeiten, Apache Camel in Ihre Anwendung einzubinden, ist die Abhängigkeit von Maven. Durch Hinzufügen des Abhängigkeitsblocks unten wird Maven die Camel-Bibliotheken und Abhängigkeiten für Sie auflösen.

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-core</artifactId>
  <version>2.17.3</version>
</dependency>
```

Gradle

Eine andere übliche Methode, Apache Camel in Ihre Anwendung aufzunehmen, ist die Abhängigkeit von Gradle. Fügen Sie einfach die Abhängigkeitszeile unten hinzu, und Gradle importiert die Camel-Bibliothek und ihre Abhängigkeiten für Sie.

```
// https://mvnrepository.com/artifact/org.apache.camel/camel-core
compile group: 'org.apache.camel', name: 'camel-core', version: '2.17.3'
```

Spring Boot

Ab Camel 2.15 können Sie jetzt die Spring Boot-Abhängigkeit von Apache Camel nutzen. Der Unterschied zu dieser Camel-Bibliothek besteht darin, dass sie eine befürwortete Autokonfiguration bietet, einschließlich der automatischen Erkennung von Camel-Routen.

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-spring-boot</artifactId>
    <version>${camel.version}</version> <!-- use the same version as your Camel core version -
->
</dependency>
```

Camel-Domänenspezifische Sprache

Camels DSL (Domain Specific Language) ist eine der Funktionen, die Camel von anderen Integrationsframeworks abhebt. Während einige andere Frameworks auch ein DSL-Konzept aufweisen, normalerweise in Form einer XML-Datei, war das DSL in solchen Fällen immer eine benutzerdefinierte Sprache.

Camel bietet mehrere DSLs in Programmiersprachen wie Java, Scala, Groovy und in XML an.

Eine einfache Dateikopierroute kann beispielsweise auf verschiedene Arten durchgeführt werden, wie in der nachstehenden Liste gezeigt

- Java DSL

  ```
  from("file:data/in").to("file:data/out");
  ```

- Blueprint / Spring DSL (XML)

  ```
  <route>
    <from uri="file:data/inbox"/>
    <to uri="file:data/out"/>
  </route>
  ```

- Scala DSL

  ```
  from "file:data/inbox" -> "file:data/out"
  ```

Erste Schritte mit Apache-Kamel online lesen: https://riptutorial.com/de/apache-

camel/topic/3511/erste-schritte-mit-apache-kamel

Kapitel 2: Integrationstest auf bestehenden Routen mit Apache-Camel und Spring (And DBUnit)

Einführung

In diesem Wiki erfahren Sie, wie Sie Integrationstests mit Apache Camel ausführen.

Genauer gesagt, können Sie eine vorhandene Route von Anfang bis Ende starten (mit oder ohne Ihre tatsächliche Datenbank) oder den Austausch zwischen jedem Teil der Route abbrechen und prüfen, ob Ihre Kopfzeilen oder der Text korrekt sind oder nicht.

Das Projekt, an dem ich dies durchgeführt habe, verwendet klassische Spring mit XML-Konfiguration und DBUnit, um eine Testdatenbank zu simulieren. Ich hoffe, das gibt Ihnen ein paar Hinweise.

Parameter

Parameter / Funktion	Einzelheiten
Austausch	Der Austausch wird innerhalb des Kamelprozessors verwendet, um Objekte zwischen Teilen Ihrer Route zu übergeben
CamelContext	Der Kamelkontext wird im Test verwendet, um den Kontext manuell zu starten und zu stoppen.
ProducerTemplate	Ermöglicht das Senden von Nachrichten auf Ihrer Route, das manuelle Einstellen des vollständigen Austauschs oder das Senden von Dummy-Kopfzeilen / -körpern
BeratungWith	Hilft Ihnen beim Definieren einer vorhandenen Route mit dem aktuellen Kontext
WeaveById	Verwendet den Ratschlag mit der Konfiguration und teilt den *Teilnehmern* Ihre Route mit, wie sie sich verhalten *sollen.*
MockEndpoint	Der Mockendpoint ist ein Punkt, den Sie für Ihren Test definieren. In Ihrer weaveById können Sie Ihrer Route die gewohnte Verarbeitung mitteilen und zu einem mockEnpoint gehen, anstatt der üblichen Route zu folgen. Auf diese Weise können Sie die Nachrichtenzahl, den Austauschstatus überprüfen ...

Bemerkungen

Einige der hier gegebenen Definitionen sind nicht genau, aber sie helfen Ihnen, den obigen Code zu verstehen. Hier finden Sie einige Links für detailliertere Informationen:

- *Informationen* zur Verwendung von *AdviceWith* und *weaveById* (oder anderen Wegen zum Auslösen von Routen) finden Sie in der offiziellen Apache-Kamel-Dokumentation: Siehe diesen Link
- Informationen zur Verwendung von *ProducerTemplate* finden Sie in der offiziellen Dokumentation erneut: Siehe diesen Link
- Um wirklich zu verstehen, worum es bei Kamel geht: detaillierte Dokumentation der Entreprise Integrationsmuster

Diese spezielle Testmethode ist selbst bei Stapelüberlauf ziemlich schwer zu finden. Das ist ziemlich spezifisch, aber zögern Sie nicht, nach mehr Details zu fragen, vielleicht kann ich Ihnen helfen.

Examples

Kamelroute Beispiel

Die folgende Route hat ein einfaches Ziel:

- Zunächst wird geprüft, ob ein **ImportDocumentProcess-** Objekt in der Datenbank vorhanden ist, und es wird als *Exchange-Header* **hinzugefügt**
- Dann fügt es ein **ImportDocumentTraitement** (welches mit dem vorherigen ImportDocumentProcess verknüpft ist) in der Datenbank hinzu

Hier ist der Code dieser Route:

```
@Component
public class TestExampleRoute extends SpringRouteBuilder {

    public static final String ENDPOINT_EXAMPLE = "direct:testExampleEndpoint";

    @Override
    public void configure() throws Exception {
        from(ENDPOINT_EXAMPLE).routeId("testExample")
            .bean(TestExampleProcessor.class,
"getImportDocumentProcess").id("getImportDocumentProcess")
            .bean(TestExampleProcessor.class,
"createImportDocumentTraitement").id("createImportDocumentTraitement")
            .to("com.pack.camel.routeshowAll=true&multiline=true");
    }

}
```

Die *ID* auf den Routen ist nicht obligatorisch, Sie können die Bean-Strings auch danach

verwenden. Ich denke jedoch, dass die Verwendung von *IDs* als bewährte *Methode* angesehen werden kann, falls sich Ihre Routenzeichenfolgen in der Zukunft ändern.

Kamelprozessor Beispiel

Der Prozessor enthält nur die von der Route benötigten Methoden. Es ist nur ein klassischer Java Bean, der mehrere Methoden enthält. Sie können auch *Processor implementieren* und die *Prozessmethode* überschreiben.

Siehe den Code unten:

```
@Component("testExampleProcessor")
public class TestExampleProcessor {

    private static final Logger LOGGER = LogManager.getLogger(TestExampleProcessor.class);

    @Autowired
    public ImportDocumentTraitementServiceImpl importDocumentTraitementService;

    @Autowired
    public ImportDocumentProcessDAOImpl importDocumentProcessDAO;

    @Autowired
    public ImportDocumentTraitementDAOImpl importDocumentTraitementDAO;

    // ---- Constants to name camel headers and bodies
    public static final String HEADER_ENTREPRISE = "entreprise";

    public static final String HEADER_UTILISATEUR = "utilisateur";

    public static final String HEADER_IMPORTDOCPROCESS = "importDocumentProcess";

    public void getImportDocumentProcess(@Header(HEADER_ENTREPRISE) Entreprise entreprise,
Exchange exchange) {
        LOGGER.info("Entering TestExampleProcessor method : getImportDocumentProcess");

        Utilisateur utilisateur = SessionUtils.getUtilisateur();
        ImportDocumentProcess importDocumentProcess =
importDocumentProcessDAO.getImportDocumentProcessByEntreprise(
                entreprise);

        exchange.getIn().setHeader(HEADER_UTILISATEUR, utilisateur);
        exchange.getIn().setHeader(HEADER_IMPORTDOCPROCESS, importDocumentProcess);
    }

    public void createImportDocumentTraitement(@Header(HEADER_ENTREPRISE) Entreprise
entreprise,
            @Header(HEADER_UTILISATEUR) Utilisateur utilisateur,
            @Header(HEADER_IMPORTDOCPROCESS) ImportDocumentProcess importDocumentProcess,
Exchange exchange) {
        LOGGER.info("Entering TestExampleProcessor method : createImportDocumentTraitement");

        long nbImportTraitementBefore =
this.importDocumentTraitementDAO.countNumberOfImportDocumentTraitement();
        ImportDocumentTraitement importDocumentTraitement =
this.importDocumentTraitementService.createImportDocumentTraitement(
                entreprise, utilisateur, importDocumentProcess, "md5_fichier_example_test",
"fichier_example_test.xml");
```

```
        long nbImportTraitementAfter =
this.importDocumentTraitementDAO.countNumberOfImportDocumentTraitement();

        exchange.getIn().setHeader("nbImportTraitementBefore",
Long.valueOf(nbImportTraitementBefore));
        exchange.getIn().setHeader("nbImportTraitementAfter",
Long.valueOf(nbImportTraitementAfter));
        exchange.getIn().setHeader("importDocumentTraitement", importDocumentTraitement);
    }
// Rest of the code contains getters and setters for imported dependencies
}
```

Nicht viel zu sagen, außer, dass wir den Austausch verwenden, um Objekte von einem Teil zum anderen zu übertragen. Dies wird normalerweise in meinem Projekt ausgeführt, da wir sehr komplexe Prozesse abwickeln müssen.

Beispiel für Testklassen der Camel Integration

> Vergessen Sie nicht, die Kameltestunterstützung und die Federkameltestunterstützung zu Ihren Projektabhängigkeiten hinzuzufügen. Für Maven-Benutzer sehen Sie Folgendes:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-test</artifactId>
  <version>${camel.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-test-spring</artifactId>
    <version>${camel.version}</version>
     <scope>test</scope>
</dependency>
```

Diese Klasse wird Tests auf der Beispielroute auslösen und ausführen. Diese Tests verwenden **DBUnit auch** zum Simulieren einer Datenbank. Sie können Ihren Kontext jedoch so konfigurieren, dass er eine echte oder eine andere Art von Datenbank verwendet.

Zunächst verwenden wir eine abstrakte Klasse, um gemeinsame Anmerkungen zwischen den einzelnen Camel Integration Test-Klassen zu teilen, die wir später verwenden werden:

```
@RunWith(CamelSpringRunner.class)
@BootstrapWith(CamelTestContextBootstrapper.class)
@ContextConfiguration(locations = { "classpath:/test-beans.xml" })
@DbUnitConfiguration(dataSetLoader = ReplacementDataSetLoader.class)
@TestExecutionListeners({ DependencyInjectionTestExecutionListener.class,
DirtiesContextTestExecutionListener.class,
        DbUnitTestExecutionListener.class })
@DirtiesContext(classMode = ClassMode.AFTER_EACH_TEST_METHOD)
public abstract class AbstractCamelTI {

}
```

Achten Sie darauf **, keine Anmerkungen zu vergessen, da** sonst Ihre DAOs nicht korrekt injiziert werden. Sie können jedoch die DBUnit-Anmerkungen sicher entfernen, wenn Sie die in Ihrer Kontextkonfiguration abgebildete Datenbank nicht verwenden möchten.

WICHTIG EDIT: Ich habe den zusätzlichen `@DirtiesContext(classMode = ClassMode.AFTER_EACH_TEST_METHOD)` vor kurzem. Auf diese Weise wird das *Kamel Kontext* für jeden Test neu geladen. Sie können jeden Teil Ihrer Route individuell testen. Wenn Sie das wirklich wollen, müssen Sie *remove ()* für die Teile der ausgewählten Route verwenden, die Sie nicht durchlaufen möchten. Einige würden argumentieren, dass dies kein echter Integrationstest ist, und sie hätten recht. Wenn Sie jedoch, wie ich, große Prozessoren haben, die Sie umgestalten müssen, können Sie dort anfangen.

Der folgende Code zeigt den Beginn der Testklasse (siehe unten für die eigentlichen Tests):

```
@DatabaseSetup(value = { "/db_data/dao/common.xml",
"/db_data/dao/importDocumentDAOCommonTest.xml" })
public class TestExampleProcessorTest extends AbstractCamelTI {

    @Autowired
    protected CamelContext camelContext;

    @EndpointInject(uri = "mock:catchTestEndpoint")
    protected MockEndpoint mockEndpoint;

    @Produce(uri = TestExampleRoute.ENDPOINT_EXAMPLE)
    protected ProducerTemplate template;

    @Autowired
    ImportDocumentTraitementDAO importDocumentTraitementDAO;

    // -- Variables for tests
    ImportDocumentProcess importDocumentProcess;

    @Override
    @Before
    public void setUp() throws Exception {
        super.setUp();

        importDocumentProcess = new ImportDocumentProcess();
        //specific implementation of your choice
    }
}
```

Der folgende Test soll den ersten Teil der Route auslösen und zu einem `mockEndpoint` damit wir testen können, ob der *ImportDocumentProcess* richtig ausgewählt und in die Kopfzeilen *eingefügt* wurde:

```
@Test
public void processCorrectlyObtained_getImportDocumentProcess() throws Exception {
    camelContext.getRouteDefinitions().get(0).adviceWith(camelContext, new
AdviceWithRouteBuilder() {

        @Override
        public void configure() throws Exception {
            weaveById("getImportDocumentProcess").after().to(mockEndpoint);
```

```
        }
    });

    // -- Launching the route
    camelContext.start();
    template.sendBodyAndHeader(null, "entreprise", company);

    mockEndpoint.expectedMessageCount(1);
    mockEndpoint.expectedHeaderReceived(TestExampleProcessor.HEADER_UTILISATEUR, null);
    mockEndpoint.expectedHeaderReceived(TestExampleProcessor.HEADER_IMPORTDOCPROCESS,
importDocumentProcess);
    mockEndpoint.assertIsSatisfied();

    camelContext.stop();
}
```

Der letzte Test löst die gesamte Route aus:

```
@Test
public void traitementCorrectlyCreated_createImportDocumentTraitement() throws Exception {
    camelContext.getRouteDefinitions().get(0).adviceWith(camelContext, new
AdviceWithRouteBuilder() {

        @Override
        public void configure() throws Exception {
            weaveById("createImportDocumentTraitement").after().to(mockEndpoint);
        }
    });

    // -- Launching the route
    camelContext.start();

    Exchange exchange = new DefaultExchange(camelContext);
    exchange.getIn().setHeader(TestExampleProcessor.HEADER_ENTREPRISE, company);
    exchange.getIn().setHeader(TestExampleProcessor.HEADER_UTILISATEUR, null); // No user in
this case
    exchange.getIn().setHeader(TestExampleProcessor.HEADER_IMPORTDOCPROCESS,
importDocumentProcess);

    long numberOfTraitementBefore =
this.importDocumentTraitementDAO.countNumberOfImportDocumentTraitement();

    template.send(exchange);

    mockEndpoint.expectedMessageCount(1);
    mockEndpoint.assertIsSatisfied();

    camelContext.stop();

    long numberOfTraitementAfter =
this.importDocumentTraitementDAO.countNumberOfImportDocumentTraitement();
    assertEquals(numberOfTraitementBefore + 1L, numberOfTraitementAfter);
}
```

Es ist auch möglich, die aktuelle Route zu einem anderen Prozess umzuleiten. Aber ich ziehe es vor, auf einen `mockEndpoint` . Es ist ein bisschen interessanter, weil Sie wirklich Zwischentests an Ihrem Körper und Kopf des Austauschs durchführen können.

WICHTIGER HINWEIS : In diesem Beispiel verwende ich den folgenden Code, um meine Routen `adviceWith` und `adviceWith` zu verwenden:

```
camelContext.getRouteDefinitions().get(0).adviceWith(camelContext, new
AdviceWithRouteBuilder() { [...] });
```

Es ist jedoch möglich, die Route anhand einer zuvor als Zeichenfolge definierten ID **abzurufen** :

```
 camelContext.getRouteDefinition("routeId").adviceWith(camelContext, new
AdviceWithRouteBuilder() {  [...]  });
```

Ich kann diese Methode wärmstens empfehlen. Sie kann viel Zeit sparen, um herauszufinden, warum Ihre Tests fehlschlagen

Integrationstest auf bestehenden Routen mit Apache-Camel und Spring (And DBUnit) online lesen: https://riptutorial.com/de/apache-camel/topic/10630/integrationstest-auf-bestehenden-routen-mit-apache-camel-und-spring--and-dbunit-

Kapitel 3: Pub / Sub mit Camel + Redis

Bemerkungen

Verwendung des Herausgebers:

```
producerTemplate.asyncSendBody("direct:myprocedure", massageBody);
```

Verwenden des "createProducer ()" in ManagedCamel zum Erstellen der Erzeuger-Vorlage.

Examples

RedisPublisher

```
public class RedisPublisher extends RouteBuilder {

    public static final String CAMEL_REDIS_CHANNEL = "CamelRedis.Channel";
    public static final String CAMEL_REDIS_MESSAGE = "CamelRedis.Message";

    @Value("${redis.host}")
    private String redisHost;
    @Value("${redis.port}")
    private int redisPort;
    @Value("${redis.channel.mychannel}")
    private String redisChannel;

    private String producerName;

    @Required
    public void setProducerName(String producerName) {
        this.producerName = producerName;
    }

    @Override
    public void configure() throws Exception {
        from(producerName)
                .log(String.format("Publishing with redis in channel: %s, massage body:
${body}", redisChannel))
                .setHeader(CAMEL_REDIS_CHANNEL, constant(redisChannel))
                .setHeader(CAMEL_REDIS_MESSAGE, body())
                .to(String.format("spring-redis://%s:%s?command=PUBLISH&redisTemplate=#%s",
redisHost, redisPort, ManagedCamel.REDIS_TEMPLATE));
    }
}
```

RedisSubscriber

```
public class RedisSubscriber extends RouteBuilder {

    @Value("${redis.host}")
    private String redisHost;
    @Value("${redis.port}")
```

```
    private int redisPort;
    @Value("${redis.channel.mychannel}")
    private String redisChannel;

    private Object bean;
    private String method;

    @Required
    public void setBean(Object bean) {
        this.bean = bean;
    }

    @Required
    public void setMethod(String method) {
        this.method = method;
    }

    @Override
    public void configure() throws Exception {
        from(String.format("spring-
redis://%s:%s?command=SUBSCRIBE&channels=%s&serializer=#%s", redisHost, redisPort,
redisChannel, ManagedCamel.REDIS_SERIALIZER))
                .log(String.format("Consuming with redis in channel: %s, massage body:
${body}", redisChannel))
                .process(exchange -> {
                }).bean(bean, String.format("%s(${body})", method));
    }
}
```

Die Methode "Methode" in der injizierten Bohne behandelt die erhaltenen Massagen.

Abonnentenfederkontext

```
<bean id="managedCamel" class="com.pubsub.example.ManagedCamel" >
    <constructor-arg name="routes">
        <list>
            <ref bean="redisSubscriber"/>
        </list>
    </constructor-arg>
</bean>

<bean id="redisSubscriber" class="com.pubSub.example.RedisSubscriber" >
    <property name="bean" ref="myBean"/>
    <property name="method" value="process"/>
</bean>
```

Publisher Spring-Kontext

```
<bean id="managedCamel" class="com.pubSub.example.ManagedCamel" >
    <constructor-arg name="routes">
        <list>
            <ref bean="redisPublisher"/>
        </list>
    </constructor-arg>
</bean>

<bean id="redisPublisher" class="com.pubSub.example.RedisPublisher" >
```

```
    <property name="producerName" value="direct:myprocedure"/>
</bean>
```

ManagedCamel

```
public class ManagedCamel implements Managed {

    public static final String REDIS_TEMPLATE = "redisTemplate";
    public static final String LISTENER_CONTAINER = "listenerContainer";
    public static final String REDIS_SERIALIZER = "redisSerializer";
    private DefaultCamelContext camelContext;

    private List<RouteBuilder> routes;
    @Value("${redis.host}")
    private String redisHost;
    @Value("${redis.port}")
    private int redisPort;
    @Value("${redis.password}")
    private String redisPassword;

    public ManagedCamel(List<RouteBuilder> routes) throws Exception {
        this.routes = routes;
    }

    @PostConstruct
    private void postInit() throws Exception {
        JndiRegistry registry = new JndiRegistry();
        final StringRedisSerializer serializer = new StringRedisSerializer();
        RedisTemplate<String, Object> redisTemplate = getRedisTemplate(serializer);
        registry.bind(REDIS_TEMPLATE, redisTemplate);
        RedisMessageListenerContainer messageListenerContainer = new
RedisMessageListenerContainer();
        registry.bind(LISTENER_CONTAINER, messageListenerContainer);
        registry.bind(REDIS_SERIALIZER, serializer);

        camelContext = new DefaultCamelContext(registry);
        for (RouteBuilder routeBuilder : routes) {
            camelContext.addRoutes(routeBuilder);
        }
        start();
    }

    private RedisTemplate<String, Object> getRedisTemplate(StringRedisSerializer serializer) {
        RedisTemplate<String, Object> redisTemplate = new RedisTemplate<String, Object>();
        redisTemplate.setConnectionFactory(redisConnectionFactory());
        redisTemplate.setKeySerializer(new StringRedisSerializer());
        redisTemplate.setValueSerializer(serializer);
        redisTemplate.setEnableDefaultSerializer(false);
        redisTemplate.afterPropertiesSet();
        return redisTemplate;
    }

    private RedisConnectionFactory redisConnectionFactory() {
        final JedisConnectionFactory jedisConnectionFactory = new JedisConnectionFactory();
        jedisConnectionFactory.setHostName(redisHost);
        jedisConnectionFactory.setPort(redisPort);
        jedisConnectionFactory.setPassword(redisPassword);
        jedisConnectionFactory.afterPropertiesSet();
        return jedisConnectionFactory;
```

```
    }

    public void start() throws Exception {
        camelContext.start();
    }

    public void stop() throws Exception {
        camelContext.stop();
    }

    public ProducerTemplate createProducer() {
        return camelContext.createProducerTemplate();
    }

}
```

Pub / Sub mit Camel + Redis online lesen: https://riptutorial.com/de/apache-camel/topic/7105/pub--sub-mit-camel-plus-redis

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit Apache-Kamel	Community, Michael Hoffman, Namphibian
2	Integrationstest auf bestehenden Routen mit Apache-Camel und Spring (And DBUnit)	DamienB, Flanfl, matthieusb
3	Pub / Sub mit Camel + Redis	Lior