eBook Gratuit

APPRENEZ apache-camel

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#apache-camel

Table des matières

À propos 1

Chapitre 1: Commencer avec apache-camel 2

Remarques 2

Examples 2

Installation ou configuration 2

Dépendance Maven 2

Gradle 2

Botte de printemps 3

Langue spécifique au domaine de chameau 3

Chapitre 2: Pub / Sub utilisant Camel + Redis 5

Remarques 5

Examples 5

RedisPublisher 5

RedisSubscriber 5

Contexte printemps abonné 6

Contexte printemps éditeur 6

ManagedCamel 7

Chapitre 3: Tests d'intégration sur des routes existantes avec Apache-Camel et Spring (et 9

Introduction 9

Paramètres 9

Remarques 10

Examples 10

Exemple de route de chameau 10

Camel Processor exemple 11

Exemple de classe de test d'intégration Camel 12

Crédits 16

À propos

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

Chapitre 1: Commencer avec apache-camel

Remarques

Apache Camel est un framework qui facilite principalement la résolution des problèmes d'intégration de l'entreprise. À la base, il peut être considéré comme un moteur de moteur de routage. Essentiellement, il vous permet de connecter des systèmes (terminaux) via des routes. Ces itinéraires acceptent les messages qui peuvent être de n'importe quel type de données.

Le framework Apache Camel contient également un ensemble complet de modèles d'intégration d'entreprise (EIP) tels que le séparateur, les agrégateurs, le routage basé sur le contenu, etc. Étant donné que l'infrastructure peut être déployée dans diverses applications Java autonomes, sur divers serveurs d'applications tels que WildFly et Tomcat ou sur un bus de services d'entreprise à part entière, elle peut être considérée comme une infrastructure d'intégration.

Pour commencer avec le framework, vous devez l'ajouter à un projet en utilisant l'une des méthodes suivantes:

1. Maven
2. Gradle
3. Botte de printemps
4. Plain Ancienne référence de bibliothèque JAR ajoutée à votre projet.

Examples

Installation ou configuration

Instructions détaillées sur l'ajout des dépendances Camel requises.

Dépendance Maven

L'une des façons les plus courantes d'inclure Apache Camel dans votre application est d'utiliser une dépendance Maven. En ajoutant le bloc de dépendance ci-dessous, Maven va résoudre les bibliothèques et les dépendances Camel pour vous.

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

Gradle

Une autre méthode courante pour inclure Apache Camel dans votre application consiste à utiliser une dépendance Gradle. Ajoutez simplement la ligne de dépendance ci-dessous et Gradle importera la bibliothèque Camel et ses dépendances pour vous.

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

Botte de printemps

À partir de Camel 2.15, vous pouvez désormais utiliser les dépendances Spring Boot d'Apache Camel. La différence avec cette bibliothèque Camel réside dans le fait qu'elle fournit une configuration automatique fiable, y compris la détection automatique des routes Camel.

```
<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>
```

Langue spécifique au domaine de chameau

Le langage DSL (Domain Specific Language) de Camel est l'une des fonctionnalités qui permet à Camel de se démarquer des autres infrastructures d'intégration. Alors que certains autres frameworks disposent également d'un concept DSL, généralement sous la forme d'un fichier XML, le DSL était dans ce cas toujours un langage personnalisé.

Camel propose plusieurs DSL dans des langages de programmation tels que Java, Scala, Groovy et XML.

Par exemple, un simple itinéraire de copie de fichier peut être effectué de différentes manières, comme indiqué dans la liste ci-dessous.

- 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"
  ```

Lire Commencer avec apache-camel en ligne: https://riptutorial.com/fr/apache-camel/topic/3511/commencer-avec-apache-camel

Chapitre 2: Pub / Sub utilisant Camel + Redis

Remarques

En utilisant l'éditeur:

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

Utiliser le "createProducer ()" dans ManagedCamel pour créer le producerTemplate.

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));
    }
}
```

La méthode «méthode» à l'intérieur du grain injecté traitera les massages reçus.

Contexte printemps abonné

```
<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>
```

Contexte printemps éditeur

```
<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();
    }

}
```

Lire Pub / Sub utilisant Camel + Redis en ligne: https://riptutorial.com/fr/apache-camel/topic/7105/pub---sub-utilisant-camel-plus-redis

Chapitre 3: Tests d'intégration sur des routes existantes avec Apache-Camel et Spring (et DBUnit)

Introduction

Le but de ce wiki est de vous montrer comment exécuter des tests d'intégration avec Apache Camel.

Plus précisément, vous pourrez ainsi lancer une route existante du début à la fin (avec ou sans votre base de données réelle) ou intercepter l'échange entre chaque partie de la route et vérifier si vos en-têtes ou votre corps sont corrects ou non.

Le projet sur lequel je fais cela utilise Spring classique avec la configuration de xml et DBUnit pour simuler une base de données de test. J'espère que cela vous donnera quelques pistes.

Paramètres

Paramètre / Fonction	Détails
Échange	L'échange est utilisé à l'intérieur du processeur camel pour transmettre des objets entre les différentes parties de votre itinéraire.
CamelContext	Le contexte camel est utilisé dans le test pour démarrer et arrêter manuellement le contexte.
ProducerTemplate	Vous permet d'envoyer des messages sur votre itinéraire, en définissant l'échange complet manuellement ou en envoyant des en-têtes / corps factices.
Des conseils	Vous aide à redéfinir une route existante avec le contexte actuel
WeaveById	Utilisé dans les conseils de configuration, indique à votre itinéraire comment se comporter (peut également utiliser *weaveByToString*)
MockEndpoint	Le point de repère est un point que vous définissez pour votre test. Dans votre weaveById, vous pouvez indiquer votre itinéraire à son traitement habituel et aller dans un mockEnpoint plutôt que de suivre l'itinéraire habituel. De cette façon, vous pouvez vérifier le nombre de messages, l'état de l'échange…

Remarques

Certaines définitions données ici ne sont pas parfaitement exactes, mais elles vous aideront à comprendre le code ci-dessus. Voici quelques liens pour des informations plus détaillées:

- À propos de l'utilisation de *AdviceWith* et de *weaveById* (ou d'autres moyens de déclencher des itinéraires), consultez la documentation officielle apache-camel: voir ce lien
- A propos de l'utilisation de *ProducerTemplate* , consultez à nouveau la documentation officielle: voir ce lien
- Pour bien comprendre ce qu'est le camel: Documentation détaillée de Enterprise Integration Patterns

Cette méthode particulière de test est assez difficile à trouver, même en cas de débordement de pile. C'est assez spécifique mais n'hésitez pas à demander plus de détails, peut-être que je pourrai vous aider.

Examples

Exemple de route de chameau

L'itinéraire suivant a un objectif simple:

- D'abord, il vérifie si l'objet **ImportDocumentProcess** est présent dans la base de données et l'ajoute en tant qu'en- *tête d'échange.*
- Ensuite, il ajoute un **ImportDocumentTraitement** (qui est lié au précédent ImportDocumentProcess) dans la base de données

Voici le code de cette 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");
    }

}
```

L' *identifiant* sur les routes n'est pas obligatoire, vous pouvez également utiliser les chaînes de bean ensuite. Cependant, je pense que l'utilisation des *identifiants* peut être considérée comme

une bonne pratique, au cas où vos chaînes de route changeraient dans le futur.

Camel Processor exemple

Le processeur ne contient que les méthodes requises par l'itinéraire. C'est juste un bean Java classique contenant plusieurs méthodes. Vous pouvez également *implémenter Processor* et remplacer la méthode de *processus* .

Voir le code ci-dessous:

```
@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
}
```

Pas grand chose à dire ici, sauf que nous utilisons l'échange pour transférer des objets d'une partie à une autre. C'est comme cela que l'on fait habituellement sur mon projet, car nous avons des processus très complexes à gérer.

Exemple de classe de test d'intégration Camel

> N'oubliez pas d'ajouter le support de test de chameau et le support de test de camel de printemps aux dépendances de votre projet. Voir les informations suivantes pour les utilisateurs de maven:

```
<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>
```

Cette classe va déclencher et exécuter des tests sur l'exemple de route. Ces tests utilisent également **DBUnit** pour simuler une base de données, mais vous pouvez configurer votre contexte pour utiliser une base de données réelle ou autre.

Tout d'abord, nous utilisons une classe abstraite afin de partager des annotations communes entre chaque classe de test d'intégration Camel que nous utiliserons plus tard:

```
@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 {

}
```

Attention à ne pas oublier les annotations ou vos DAO ne seront pas injectés correctement. Cela dit, vous pouvez supprimer en toute sécurité les annotations DBUnit si vous ne souhaitez pas utiliser la base de données décrite dans votre configuration de contexte.

IMPORTANT : J'ai ajouté le `@DirtiesContext(classMode = ClassMode.AFTER_EACH_TEST_METHOD)` récemment. De cette façon, le *contexte camel* est rechargé pour chaque test. Vous pouvez vraiment tester chaque partie de votre itinéraire individuellement. Cependant, si vous le voulez vraiment, vous devez utiliser *remove ()* sur les parties de l'itinéraire choisi que vous ne souhaitez pas parcourir. Certains diront que ce n'est pas un véritable test d'intégration et qu'ils auront raison. Mais si, comme moi, vous avez besoin de gros processeurs, vous devez commencer par là.

Le code ci-dessous décrit le début de la classe de test (voir ci-dessous pour les tests réels):

```
@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
    }
}
```

Le test suivant est supposé déclencher la première partie de la route et l'amener à un `mockEndpoint` afin que nous puissions tester si *ImportDocumentProcess* a été correctement sélectionné et placé dans les en-têtes:

```
@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();
}
```

Le dernier test déclenche l'intégralité de l'itinéraire:

```
@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);
}
```

Il est également possible de rediriger l'itinéraire actuel vers un autre processus. Mais je préfère rediriger vers un `mockEndpoint` . C'est un peu plus intéressant car vous pouvez vraiment faire des tests intermédiaires sur votre corps d'échange et vos en-têtes.

REMARQUE IMPORTANTE : Dans cet exemple, j'utilise le code suivant pour obtenir mes

itinéraires et utiliser le `adviceWith` :

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

Cependant, il est possible d'obtenir la route par un identifiant précédemment défini comme une chaîne, comme ceci:

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

Je recommande fortement cette méthode, cela peut économiser beaucoup de temps à comprendre pourquoi vos tests échouent

Lire Tests d'intégration sur des routes existantes avec Apache-Camel et Spring (et DBUnit) en ligne: https://riptutorial.com/fr/apache-camel/topic/10630/tests-d-integration-sur-des-routes-existantes-avec-apache-camel-et-spring--et-dbunit-

Crédits

S. No	Chapitres	Contributeurs
1	Commencer avec apache-camel	Community, Michael Hoffman, Namphibian
2	Pub / Sub utilisant Camel + Redis	Lior
3	Tests d'intégration sur des routes existantes avec Apache-Camel et Spring (et DBUnit)	DamienB, Flanfl, matthieusb