EBook Gratuito

APPRENDIMENTO
apache-camel

Free unaffiliated eBook created from
Stack Overflow contributors.

#apache-camel

Sommario

Di

Capitolo 1: Iniziare con apache-camel

Osservazioni

Apache Camel è un framework che facilita principalmente la risoluzione delle sfide di integrazione aziendale. Al suo centro può essere pensato come un costruttore di motori di routing. In sostanza, consente di connettere i sistemi (endpoint) tramite percorsi. Questi percorsi accettano messaggi che possono essere di qualsiasi tipo di dati.

Il framework Apache Camel contiene anche un set completo di EIP (modelli di integrazione aziendale) come splitter, aggregatori, routing basato sul contenuto e così via. Poiché il framework può essere implementato in varie applicazioni Java standalone, in vari server applicativi come WildFly e Tomcat o su un bus di servizio aziendale completo può essere visto come un framework di integrazione.

Per iniziare con il framework è necessario aggiungerlo a un progetto utilizzando uno dei seguenti metodi:

1. Maven
2. Gradle
3. Spring Boot
4. Semplice vecchio riferimento alla libreria JAR aggiunto al progetto.

Examples

Installazione o configurazione

Istruzioni dettagliate sull'aggiunta delle dipendenze Camel richieste.

Dipendenza da Maven

Uno dei modi più comuni per includere Apache Camel nella tua applicazione è attraverso una dipendenza Maven. Aggiungendo il blocco delle dipendenze sotto, Maven risolverà le librerie e le dipendenze Camel per te.

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

Gradle

Un altro modo comune per includere Apache Camel nella tua applicazione è attraverso una dipendenza Gradle. Aggiungi semplicemente la linea di dipendenza qui sotto e Gradle importerà la libreria Camel e le sue dipendenze per te.

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

Spring Boot

A partire da Camel 2.15, ora puoi sfruttare la dipendenza Spring Boot di Apache Camel. La differenza con questa libreria Camel è che fornisce una configurazione automatica supponente, incluso il rilevamento automatico delle rotte 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>
```

Lingua specifica del dominio cammello

Camel's DSL (Domain Specific Language) è una delle caratteristiche che rende Camel straordinario rispetto ad altri framework di integrazione. Mentre alcuni altri framework dispongono anche di un concetto DSL, in genere sotto forma di file XML, in questi casi il DSL era sempre un linguaggio personalizzato.

Camel offre più DSL in linguaggi di programmazione come Java, Scala, Groovy e in XML.

Ad esempio, un semplice percorso di copia dei file può essere eseguito in vari modi, come mostrato nell'elenco seguente

- DSL Java

  ```
  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"
  ```

Leggi Iniziare con apache-camel online: https://riptutorial.com/it/apache-camel/topic/3511/iniziare-

con-apache-camel

Capitolo 2: Pub / Sub usando Camel + Redis

Osservazioni

Utilizzando l'editore:

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

Utilizzando "createProducer ()" in ManagedCamel per creare il 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));
    }
}
```

Il metodo 'metodo' all'interno del fagiolo iniettato gestirà i massaggi ricevuti.

Contesto primaverile dell'abbonato

```
<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>
```

Contesto primavera editore

```
<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();
    }

}
```

Leggi Pub / Sub usando Camel + Redis online: https://riptutorial.com/it/apache-camel/topic/7105/pub---sub-usando-camel-plus-redis

Capitolo 3: Test di integrazione su rotte esistenti con Apache-Camel e Spring (e DBUnit)

introduzione

Il punto di questo wiki è mostrarti come eseguire test di integrazione usando Apache Camel.

Più precisamente, facendo questo sarai in grado di lanciare un percorso esistente dall'inizio alla fine (con o senza il tuo vero database) o intercettare lo scambio tra ogni parte del percorso e verificare se le intestazioni o il corpo sono corretti o meno.

Il progetto che ho fatto su utilizza la classica Spring con la configurazione xml e DBUnit per deridere un database di test. Spero che questo ti dia qualche vantaggio.

Parametri

Parametro / Funzionalità	Dettagli
Scambio	Lo scambio viene utilizzato all'interno del processore cammello per passare oggetti tra le parti del percorso
CamelContext	Il contesto cammello viene utilizzato nel test per avviare e interrompere manualmente il contesto.
ProducerTemplate	Ti consente di inviare messaggi lungo il percorso, impostando manualmente lo scambio completo o inviando intestazioni / corpo fittizi
AdviceWith	Aiuta a ridefinire una rotta esistente con il contesto attuale
WeaveById	Utilizzato all'interno del consiglio con la configurazione, indica ai brani del percorso come comportarsi (può anche usare *weaveByToString*)
MockEndpoint	Il mockendpoint è un punto che definisci per il tuo test. Nel tuo weaveById, puoi dire al tuo percorso la sua solita elaborazione e andare in un mockEnpoint piuttosto che seguire la solita rotta. In questo modo puoi controllare il numero di messaggi, lo stato di cambio ...

Osservazioni

Alcune definizioni fornite qui non sono perfettamente accurate, ma ti aiuteranno a capire il codice

sopra. Ecco alcuni link per informazioni più dettagliate:

- Informazioni sull'uso di *AdviceWith* e *weaveById* (o altri modi per attivare i percorsi), dai un'occhiata alla documentazione ufficiale di apache-camel: vedi questo link
- Per quanto riguarda l'utilizzo di *ProducerTemplate* , consultare di nuovo la documentazione ufficiale: vedere questo link
- Per capire veramente cos'è il cammello: documentazione dettagliata di Entreprise Integration Patterns

Questo particolare modo di test è piuttosto difficile da trovare, anche in caso di overflow dello stack. Questo è abbastanza specifico ma non esitare a chiedere maggiori dettagli, forse potrò aiutarti.

Examples

Esempio di itinerario cammello

Il seguente percorso ha un obiettivo semplice:

- Innanzitutto, controlla se l'oggetto **ImportDocumentProcess** è presente nel database e lo aggiunge come *intestazione di scambio*
- Quindi, aggiunge un **ImportDocumentTraitement** (che è collegato al precedente ImportDocumentProcess) nel database

Ecco il codice di questa rotta:

```
@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' *id* sui percorsi non è obbligatorio, puoi usare anche le stringhe di bean in seguito. Tuttavia, ritengo che l'utilizzo di *id* possa essere considerato una buona pratica, nel caso in cui le stringhe del percorso cambino in futuro.

Esempio di processore Camel

Il processore contiene solo contiene i metodi necessari per il percorso. È solo un Java Bean classico contenente diversi metodi. È inoltre possibile *implementare Processor* e sovrascrivere il metodo di *processo* .

Guarda il codice qui sotto:

```
@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
}
```

Non c'è molto da dire qui, tranne che usiamo lo scambio per trasferire oggetti da una parte all'altra. Questo è il modo in cui di solito viene fatto sul mio progetto, dal momento che abbiamo processi molto complessi da gestire.

Esempio di classe di test di integrazione Camel

> Non dimenticare di aggiungere il supporto per il test del cammello e il supporto per il test del cammello a molla alle dipendenze del progetto. Vedi quanto segue per gli utenti esperti:

```
<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>
```

Questa classe sta per attivare ed eseguire test sul percorso di esempio. Questi test utilizzano anche **DBUnit** per simulare un database, sebbene sia possibile configurare il contesto in modo che utilizzi un database reale o di altro tipo.

Innanzitutto, usiamo una classe astratta per condividere annotazioni comuni tra ogni classe di test di integrazione Camel che utilizzeremo in seguito:

```
@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 {

}
```

Fare attenzione a non dimenticare alcuna annotazione o i DAO non verranno iniettati correttamente. Detto questo, è possibile rimuovere in modo sicuro le annotazioni DBUnit se non si desidera utilizzare il database rappresentato nella configurazione del contesto.

MODIFICA IMPORTANTE : Ho aggiunto `@DirtiesContext(classMode =`

`ClassMode.AFTER_EACH_TEST_METHOD)` . In questo modo, il *contesto cammello* viene ricaricato per ogni test. Puoi davvero testare ciascuna parte del tuo percorso individualmente. Tuttavia, se lo vuoi davvero, devi usare *remove ()* sulle parti della rotta scelta che non vuoi attraversare. Alcuni sostengono che questo non è un vero test di integrazione, e avrebbero ragione. Ma se, come me, hai processori di grandi dimensioni hai bisogno di refactoring, puoi iniziare da lì.

Il codice seguente illustra l'inizio della classe di test (vedi sotto per i test effettivi):

```
@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
    }
}
```

Il seguente test dovrebbe attivare la prima parte del percorso e portarlo a un `mockEndpoint` modo che possiamo verificare se l' *ImportDocumentProcess* è stato correttamente selezionato e inserito nelle intestazioni:

```
@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();
}
```

L'ultimo test attiva l'intero percorso:

```
@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);
}
```

È anche possibile reindirizzare il percorso corrente a un altro processo. Ma preferisco il reindirizzamento a un `mockEndpoint` . È un po 'più interessante perché puoi davvero fare test intermedi sul corpo degli scambi e sulle intestazioni.

NOTA IMPORTANTE : in questo esempio sto usando la seguente parte di codice per ottenere i miei percorsi e usare i `adviceWith` questi:

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

TUTTAVIA È possibile ottenere il percorso con un ID precedentemente definito come una stringa, come questo:

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

Consiglio vivamente questo metodo, può risparmiare un sacco di tempo per capire dove sono andati a finire i test

Leggi Test di integrazione su rotte esistenti con Apache-Camel e Spring (e DBUnit) online: https://riptutorial.com/it/apache-camel/topic/10630/test-di-integrazione-su-rotte-esistenti-con-apache-camel-e-spring--e-dbunit-

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con apache-camel	Community, Michael Hoffman, Namphibian
2	Pub / Sub usando Camel + Redis	Lior
3	Test di integrazione su rotte esistenti con Apache-Camel e Spring (e DBUnit)	DamienB, Flanfl, matthieusb