.

Camel in action – IV. fejezet.

Levente Matusz
11/10/2013

Tartalomjegyzék

Bean-ek használata

Ha több mint öt éve foglalkozol szoftver fejlesztéssel, találkozhattál már több különböző komponens modellel, mint például a CORBA, EJB, JBI, SCA vag az OSGi. Ezekre a modellekre általában igaz volt és igaz most is, hogy túl nagy a függőségek és kötöttségek száma, megszabja, hogy mit tudsz és mit nem tudsz megvalósítani, nem beszélve a túl komplex csomagolási és telepítési modellekről. Bizonyos esetekben sokkal több idő ment el az alapvető környezet felépítésével, mint magával a valódi üzleti alkalmazással. Az alkalmasok komplexitásának növelésével egyre jobban átláthatatlanabbá vált a rendszer és a fejlesztők között ez egyre nagyobb frusztációt okozott. Míg nem jött egy egyszerű, átlátható keret rendszer a nyílt forráskódú alkalmazások fejlesztők közösségétől a POJO modell. Később Spring keretrendszer.

A Spring keretrendszer egy új megoldást adott, bizonyítva hogy a POJO programozási modell és a könnyű konténer valóban megfelelnek az elvárásoknak a mai vállalkozások számára. Sőt, az egyszerű programozási modell és könnyű konténer koncepció sokkal jobbnak bizonyult a túlságosan összetett vállalati alkalmazások és integrációs szerverekkel szemben, amelyeket korábban használtak.

Hogyan jönnek mindezek a Camelhöz? Nos a Camel nem szabja meg, hogy milyen komponenseket használjunk vagy hogy milyen programozási modellt. Nem szükséges hosszú és részletes specifikációkat olvasni, hogy elkezdhessünk fejleszteni. A Camelnek nincs szüksége hogy újra struktúráld a korábbi könyvtáraidat. A Camel hasonló a Spring keretrendszerhez, mindkettő könnyű kontért használnak POJO programozási modellel.

Valójában a Camel hasznosítja a POJO programozási modellt és egy hasznos mankóval egészíti ki, melynek segítségével dolgozhatunk a beanjeinkkel. A Camel nem csak csökkentet csatolást biztosít a beanekkel, hanem laza csatolást is a Camel vonalaival. Például, akár három csapat is dolgozhat ugyan azon a rendszeren, saját útvonalaikon keresztül.

A következendőekben bemutatjuk, hogy hogyan ne használjuk a beaneket Camellel, ami hozzá segít ahhoz, hogy megértsük, hogyan is működik ez valójában. Majd egy rész következik a Service Activator EIP implementációjáról. Végezetől bemutatjuk a bean-biding(bean rögzítő) processzt, melynek segítségével. Lehet elsőre kicsit komplexen hangzik, de ne aggódj, nem sokára minden tiszta lesz.

Beanek használata, a nehéz út és a járható út

Ebben a részben egy példán keresztül mutatom be, hogyan ne használjuk a beaneket Camellel. – Ez a nehéz út. Majd bemutatom, hogyan használhatjuk a beaneket könnyebben (Járható út).
Van egy már létező bean, ami egy opetációt (szervízt) biztosít, amit integrálnunk kell az alkalmazásunkban. Például egy, *HelloBean* ami egy *hello* metódust biztosít, mint szervíz:

```
public class HelloBean{
      public String hello(String name){
            return ”Hello ”+ name;
      }
}
```

A következendőekben nézzük meg milyen módokon tudjuk használni ezt a beant az alkalmazásunkban.

Bean hívása a tiszta Java környezetből

Processor használata, hogy meghívjuk a hello metódust a HelloBeanben

```
public class InvokeWithProcessorRoute extends RouteBuilder {
public void configure() throws Exception {
      from("direct:hello")
            .process(new Processor() {
            public void process(Exchange exchange) throws
            Exception {
              String name =
            exchange.getIn().getBody(String.class);
              HelloBean hello = new HelloBean();
              String answer = hello.hello(name);
              exchange.getOut().setBody(answer);
            }
      });
}
```

A fenti kódrészletben egy RouteBuilder látható, ami az útvonalat definiálja. A kódban található egy beágyazott Camel Processor, ami biztosítja a process metódust, melynek segítségével az üzenetet fellehet dolgozni egyszerű Java utasítások segítségével. Először, kikell csomagolni az üzenet body(tartalmi) részét, amit majd később felhasznál a bean

hívásakor paraméterként. Majd inicializálni kell a beant és meghívni. Végezetül bekell állítani a kimenetet a beanben kimeneti üzenetként.

A fentiekben bemutatott módszerben Java DSL-t használtunk, a nehezebbik út megvalósításához, most lássuk hogyan oldható meg Spring XML-ben.

Bean definíció hívása Springben.

Ha Springet használunk bean konténernek és a beaneket XML-ben definiáljuk.

Spring beállítása, Camel használatához, HelloBean használatához

```
<bean id="helloBean" class="camelinaction.HelloBean"/>
<bean id="route"
class="camelinaction.InvokeWithProcessorSpringRoute"/>
<camelContext id="camel"
xmlns="http://camel.apache.org/schema/spring">
     <routeBuilder ref="route"/>
</camelContext>
```

Először is definiálnunk kell a *HelloBean*t a Spring XML fájljában, helloBean id-val. Ebben az is muszáj Java DSL-t használni, hogy buildeljük az útvonalat, szóval szükség van egy másik bean deklarációjára, mely tartalmazza az útvonalat. Végezetül a *CamelContext* definíciója, mely a Spring és Camel közti kommunikációt valósítja meg.

Camel útvonal, HelloBean hívása Processor segítségével

```
@Autowired
private HelloBean hello;
public void configure() throws Exception {
     from("direct:hello")
          .process(new Processor() {
               public void process(Exchange exchange) throws
               Exception {
               String name =
          exchange.getIn().getBody(String.class);
               String answer = hello.hello(name);
               exchange.getOut().setBody(answer);

          }
          });
     }
}
```

A különbség az, hogy most a bean injektálva van Spring @Autowired annotáció segítségével, melynek köszönhetően közvetelenül lehet használni a beant.
A fentiekben két példán keresztül látható volt, hogyan érhetők el a beanek Camel útvonal segítségével ami egy kicsit túl komplikált volt.
A következendőekben felsorolok néhány okot, hogy miért is nehéz beanekkel dolgozni:

- Muszáj Java kódot használni, hogy meghívjuk a beant.
- Muszáj a Camel Processort használni, ami csak nehezebbé teszi, hogy megértsük mi is történik valójában.
- Muszáj kicsomagolni az adatot (a Camel üzenetből) és továbbítani a beanhez, majd a választ is muszály továbbítani (a beantől a Camel üzenethez).
- Muszáj inicializálni a beant vagy dependency injectiont használni.

Következzen a járható út...

Beanek használata – A járható út

Tegyük fel, hogy definiálva van a Camel útvonal a Spring XML fájlban, a RouteBuilder class helyett. A következő kód részlet bemutatja, hogyan is néz ki:

```
<bean id="helloBean" class="camelinaction.HelloBean"/>
     <camelContext id="camel"
     xmlns="http://camel.apache.org/schema/spring">
          <route>
               <from uri="direct:start"/>
               < What goes here >
          </route>
     </camelContext>
```

Először definiálni kell egy Spring beant, majd egy Camel útvonalat *direct:start* bemenettel.
A What goes here helyén muszáj lenne meghívni a HelloBeant, de ez egy XML, itt nem lehet Java kódot használni.
Camelben van egy egyszerű megoldás, hogy beaneket használjunk, még pedig a <bean> taggel.

```
<bean ref="helloBean" method="hello"/>
```

A következő útvonalat adja meg:

```
<camelContext id="camel"
xmlns="http://camel.apache.org/schema/spring">
     <route>
          <from uri="direct:start"/>
```

```
            <bean ref="helloBean" method="hello"/>
      </route>
</camelContext>
```

A Camel ugyanezt a megoldást biztosítja mikor Java DSL-t használunk. Mint például:

```
public void configure() throws Exception {
      from("direct:hello").beanRef("helloBean", "hello");
}
```

Melynek segítségével 8 sornyi kódot egy sorrá redukálhatunk. Egy sort sokkal könnyebb megérteni.

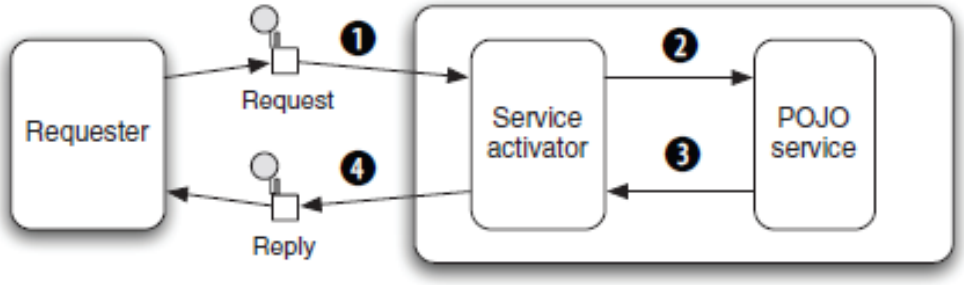
Kihagyható a hello metódus , mivel a beanek csakis egy egyedülálló metódusa van:

```
public void configure() throws Exception {
      from("direct:hello").beanRef("helloBean");
}
```

A <bean> tag egy elegáns megoldás, hogy beanekkel dolgozzunk. Enélkül muszáj a Camel Processor-t használni, hogy meghívjuk a Camel beant.

Szervízt aktiváló minta

A Service Activator minta egy üzleti minta amit a „Hohpe and Woolf's Enterprise Integration Patterns" (http://www.enterpriseintegrationpatterns.com/) könyvben fogalmaztak meg először. Amelyben egy szervízről írtak ami könnyen meghívható üzenet küldő és nem üzenetküldő szervízből egyaránt.



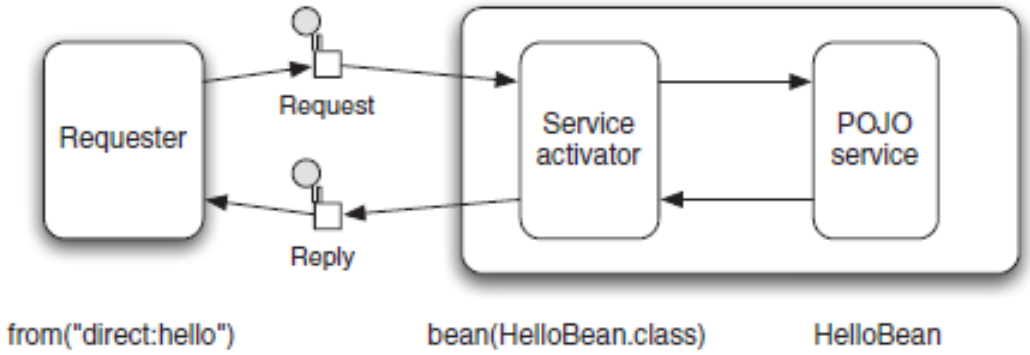
1. ábra - Service activator a kérő és a POJO szervíz között

A fenti ábrán látható a szervíz aktiváló komponens ami meghív egy szolgáltatást ami kezeli a bejövő kéréseket és vissza elindítja a választ. A szervíz aktiváló egy közvetítő a kérő és a POJO szervíz között. A kérő küld egy kérést a szervíz aktivátortnak **1.** ami feldolgozható(érthető) alakba formázza a kérést a POJO szervíz számára és **2.** továbbítja a

kérést szervíznek. A POJO szervíz vissza küldi a választ a szervíz aktivátornak **3.**, ami továbbítja (itt is formálja) a **4.** várakozó kérő számára.

Amint látható az ábrán, ez a szolgáltatás (szervíz aktivátor) valami olyasmi lenne Camelben ami képes alkalmazkodni a kéréshez és meghívni a szervízt. Ez a valami (Camelben) a Camel Bean komponens ami a *org.apache.camel.component.bean.BeanProcessor* használja a megvalósításhoz. Nézzük meg a Camel bean komponenst mint Service Activator minta.

Hasonlítsuk össze a szervíz aktivátor mintát a fentebb bemutatott Camel útvonal példán:



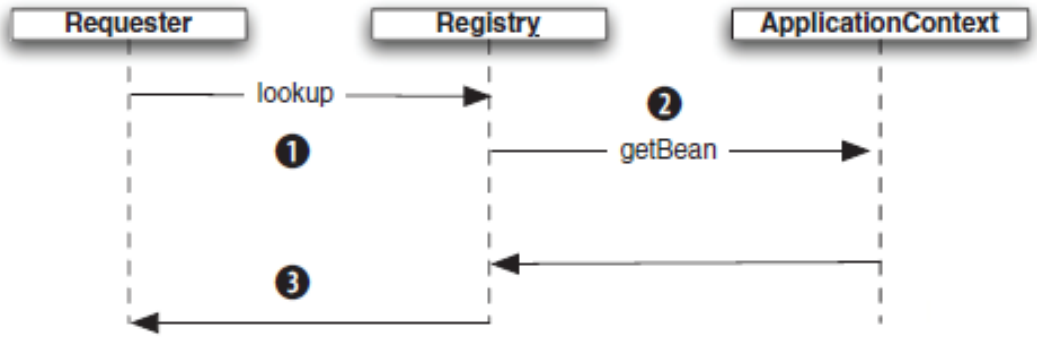
2. ábra Kapcsolat Camel útvonal és Service Activator között

A fenti ábrán látható, hogyan route maps a Service Activator EIP-hez. A kérő a csomópont ami a hamarabb jön mint a bean – ez a from("direct:hello") példából jön. A szervíz aktivátor önmagában a bean csomópont, ami BeanProcessor-ként jelenik meg a Camelben és POJO szervíz önmagában a HelloBean.

Most már érthető hogyan dolgozik a Camel beanekkel – a szervíz aktivátor mintával. De mielőtt használnák a beaneket, szükséges megtudni, hogy honnan jön. Ez az ahol a regisztáció történik ak épen Nézzük meg hogyan dolgozik a Camel külön féle regisztrációkkal.

Camel bean regisztrációk

A Camel filózófiája, hogy hatással legyen rá a legjobb elérhető keretrendszerek, szóval egy pluggable registry architektúrát használ, hogy integrálja őket. A Spring is egy ilyen keret rendszer például, a következő ábra szemlélteti is ezt:



3. ábra A kérő bekér egy beant a Camel registryvel ami a Spring ApplicationContextel meghatározza hol található a bean.

A fenti ábrán látható hogy a Camel registry egy absztrakció ami a hívó és a valós registry között van. Amikor a kérőnek szüksége van egy beanre **1**, a Camel Registryt használja. A Camel registry elvégzi a keresést a valós registryn keresztül **2**. A bean utána visszatér a kérőhöz **3**. Ez a struktúra lehetővé teszi a laza csatolást de mégis egy pluggable architektúra ami integrálja a többszörös regisztrációkat. Mint kerőnek tudnia kel, hogy hogy van kölcsön hatásban a Camel registryvel.

A regisztráció Camelben Service Provider Interface-n keresztül valósul meg ami a org.apache.camel.spi.Registry van definiálva. Az interfacek a következők:

```
Object lookup(String name);
<T> T lookup(String name, Class<T> type)
<T> Map<String, T> lookupByType(Class<T> type)
```

Leggyakrabban az első két metódust használják, hogy megkeressék az egyes beaneket név alapján. Például a HelloBean-re a következő képen nézne ki:

```
HelloBean hello =
 (HelloBean) context.getRegistry().lookup("helloBean");
```

Vagy így:

```
HelloBean hello =
 context.getRegistry().lookup("helloBean", HelloBean.class);
```

Az utolsó metódusban használta lookupByType a leggyakrabban használt Camel támógató konvenció – lehetővé teszi, hogy a Camel megkeresse a beaneket anélkül, hogy tudná a nevüket.

A regisztrációk absztraktak tehát így interfacek. A következő táblázatban a négy lehetséges implementáció látható Camelben.

Registry	**Leírás**
SimpleRegistry	Egyszerű implementáció unit teszteknél vagy ha a Google App Engine-ben fut a Camel, ahol az egyetlen korlát az elérhető JDK osztályok száma.
JndiRegistry	Egy implementáció ami a létező Java Naming és Directory Interface (JNDI) regisztrációt használja, hogy megkeresse a beaneket.
ApplicationContextRegistry	Egy implementáció ami a Spring keretrendszerrel működik együtt, hogy megkeresse a beaneket a Spring ApplicationContext segítségével. Ez az implementció az alapértelmezett és a automatikus implementált megoldás, ha a Camelt Spring környezetben használjuk.
OsgiServiceRegistry	Egy implementáció ami a OSGi szervízt használja. Az előzőhöz hasonlóan, ha OSGi környezetben használjuk a Camelt, ez az automatikusan létrejövő implementáció.

1. táblázat - Registry megvalósítások

A következő részben részletezem az egyes megvalósításokat.

SimpleRegistry

A SimpleRegistry egy Map-alapú regisztráció amit általában teszteknél használnak, vagy mikor a Camel önállóan fut.
Például ha unit tesztelni akarjuk a fentebbi HelloBean példát a kód következő képen nézne ki:

```
public class SimpleRegistryTest extends TestCase {
     private CamelContext context;
     private ProducerTemplate template;
     protected void setUp() throws Exception {
          SimpleRegistry registry = new SimpleRegistry();
          registry.put("helloBean", new HelloBean());
```

```
            context = new DefaultCamelContext(registry);
            template = context.createProducerTemplate();
            context.addRoutes(new RouteBuilder() {
                 public void configure() throws Exception {
                      from("direct:hello").beanRef("helloBean")
                 ;
                 }
            });
            context.start();
      }
      protected void tearDown() throws Exception {
            template.stop();
            context.stop();
      }
      public void testHello() throws Exception {
            Object reply =
                 template.requestBody("direct:hello", "World");
            assertEquals("Hello World", reply);
      }
}
```

Először létre kell hozni egy *SimpleRegistry* példányt és feltölteni a HelloBean-el a helloBean nevével. Ahhoz, hogy használni tudjuk a registryt, muszáj továbbítani a registryt mint paramétert a DefaultCamelContext konstruktorának. A teszteléshez létre kell hozni egy ProducerTemplatet, ami lehetővé teszi, hogy könnyedén küldjünk üzeneteket a Camelnek, mint ahogy látszi a test metódusban. Végezetül amikor a teszt kész muszáj kisöpörni a forrásokat, a Camel leállításával. A routeban a beanRef metódussal hívható a HelloBean, a heloBean névvel lehet azonosítani, amit korábban adtunk neki regisztrációkor.

JndiRegistry

A JndiRegistry, mint ahogy a neve is mutatja, integrálja a JNDI- alapú registryt. Ez volt az első registry amit a Camel integrált, tehát ez az alapértelmezett registry egy Camel projectben, ha csak nem írjuk felül.

```
CamelContext context = new DefaultCamelContext();
```

A JndiRegistryt gyakran használják unit tesztekhez vagy ha a Camel önállóan fut. A legtöbb unit teszt Camelben a JndiRegistryt használja mivel hamarabb jön létre mielőtt a

SimpleRegistry hozzáadódik a Camelhöz. A JndiRegistry elég hasznos, ha a Camelt együtt használjuk egy Java EE alkalmazás szerverrel, biztosít egy JNDI-alapú registryt out of the box feuterrel. Tegyük fel, a következő példában, hogy szükségünk van egy JNDI registryre WebSphere alkalmazás szerverrel, akkor a következő képen kell implementálnunk a kódot:

```
protected CamelContext createCamelContext() throws Exception
{
     Hashtable env = new Hashtable();
     env.put(Context.INITIAL_CONTEXT_FACTORY,
     "com.ibm.websphere.naming.WsnInitialContextFactory");
     env.put(Context.PROVIDER_URL,
     "corbaloc:iiop:myhost.mycompany.com:2809");
     env.put(Context.SECURITY_PRINCIPAL, "username");
     env.put(Context.SECURITY_CREDENTIALS, "password");
     Context ctx = new InitialContext(env);
     JndiRegistry jndi = new JndiRegistry(ctx);
     return new DefaultCamelContext(jndi);
}
```

Szükség van egy Hashtablere, hogy tárolni tudjuk a JNDI Registry információit. Majd létre kell hozni egy javax.naming.Context amit a JndiRegistry használ.
Camel ezen felül lehetővé teszi, hogy a Spring XML-el együtt használjuk a JndiRegistryt. Az egyetlen dolog amire szükség van, az hogy definiáljunk egy Spring beant és a Camel automatikusan elkészíti:

```
<beanid="registry"class="org.apache.camel.impl.JndiRegistry">
```

A szokásos Spring zsargonnal továbbítható a Hashtable paraméter a JndiRegistry konstruktorában.
A következő registry, mikor együtt használjuk a Springet Camellel.

ApplicationContextRegistry

Az ApplicationContextRegistry az alapértelmezett registry Camel és Spring együtt működése esetén. Pontosabban fogalmazva, ez az alapértelmezett ha a Camel beállításokat a Spring XML-ben hozzuk létre, a következő kód részlet illusztrálja:

```
<camelContext id="camel"
xmlns="http://camel.apache.org/schema/spring">
     <route>
          <from uri="direct:start"/>
          <bean ref="helloBean" method="hello"/>
     </route>
</camelContext>
```

Camel definíciója a <camelContext> tag segítségével automatikusan tudathatjuk, hogy az ApplicationContextRegistry használjuk. Ez a registry lehetővé teszi, hogy beaneket definiáljunk a Spring XML fájlban. Például definiálhatjuk a helloBean beant a következő képen:

```
<bean id="helloBean" class="camelinaction.HelloBean"/>
```

Ha Camelt és Springet használunk egyszerre, ugyan úgy definiálhatunk Spring beaneket és a Camel képes őket használni minden további konfiguráció nélkül.
Az utolsó Registry ha Camelt és OSGi-t használunk egyszerre.

OsgiServiceRegistry

Ha Camelt használunk OSGi környezetben, a Camel egy két lépéses módszert alkalmaz a folyamathoz. Először megnézi, hogy létezik-e szervíz olyan néven, az OSGi szervít registryben. Ha nem, a Camel visszalép és megkeresi név alapján a „hagyományos” registryben mint a Spring ApplicationContextRegistryben.
Például, ha szeretnénk exportálni a HelloBeant mint OSGi szervíz. Ezt kell elvégeznünk:

```
<osgi:service id="helloService"
interface="camelinaction.HelloBean"
ref="helloBean"/>
```

```
<bean id="helloBean" class="camelinaction.HelloBean"/>
```

Segítségül szolgáljon, hogy az osgi:service névteret a Spring Dynamic Modules (Spring DM; http://www.springsource.org/osgi), exportálja a HelloBeant OSGi registrybe helloService név alapján. Ezután használható a HelloBean Camel útvonalból ugyan úgy mint a korábbiakban, csak itt a OSG szervíz nevére kell hivatkozni:

```
<camelContext id="camel"
     xmlns="http://camel.apache.org/schema/spring">
     <route>
          <from uri="direct:start"/>
          <bean ref="helloService" method="hello"/>
     </route>
</camelContext>
```

Ilyen egyszerűen. Az egyetlen amire emlékezni kell, hogy melyik beanből lett exportálva. A Camel mefogja keresni az OSGi szervíz regisztrációk között és a Spring bean container közösen. Ez a konvenció érvényes a registry konfigurációra.
A bean regisztrációs részről ennyit, a következendőekben leírom, hogy a Camel hogyan választja ki, melyik metódus hívódjon meg az adott beanre.

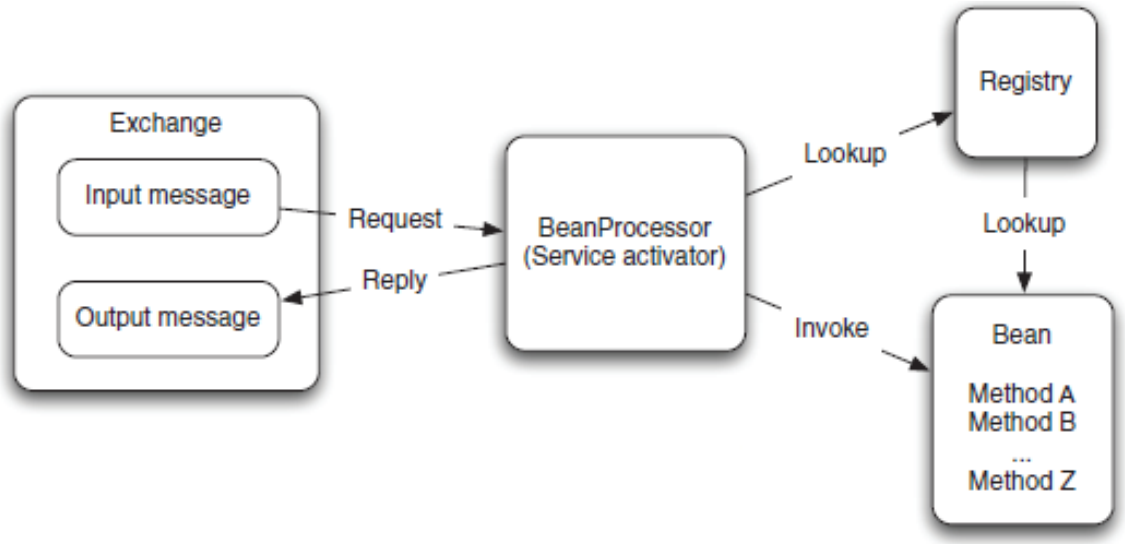
Megcímzett bean metódusai

A korábbiakban látható volt, hogyan dolgozik együtt a Camel beanekkel, az útvonal perspektívájából. Most itt az ideje, hogy mélyebbre ássunk és megnézzük, az egyes komponeseket működés közben. Az első a Camel azon mechanizmusának megértése, mely a meghívandó metódusokat választja ki.
Emlékezzünk a Camel szervíz aktivátor BeanProcessorára ami a hívó és az aktuális bean között helyezkedett el. Compile időben nincs közvetlen kapcsolat és a JVM nem tudja a hívót a beanhez linkelni – a Camel megtudja ezt oldani futásidőben.
A lentebbi ábra (4. ábra) szemlélteti azt hogyan befolyálsolja a BeanProcessor a registryt, hogy meghívja a beant.

Futásidőben a Camel exchange is routedm és pontot ad az útvonalnak, amit a BeanProcessor ér el. A BeanProcessor mindezek után feldolgozza az adatcserét és előkészíti a következő lépéseket:

1. Megkeresi a beant a registryben
2. Kiválasztja a meghívandó metódust a beanen
3. Felparaméterezi a metódust
4. Meghívódik a metódus
5. Lekezel minden felmerülő hibát
6. Beállítja a metódus válaszát (ha van neki) mint a kimeneti üzenet törzsét

A fentiekben ismertettem hogyan működik a registry keresés. A következő két lépésben (a precedencia lépcsőn a második és harmadik lépés) még összetettebb. Az oka miért komplex, hogy a Camelnek fel kell dolgozni, hogy melyik bean vagy metódus hívódjon meg futás időben, míg a java kód compile timeban van.



4. ábra Bean hívása Camelben

Ahogyan a Camel kiválasztja a bean metódusokat

Ellentétben fordítási időben, mikor a Java compiler megtudja szólítani a metódusokat, a Camel BeanProcessor kiválasztja, hogy melyik metódus hívódjon meg futásidőben.
Mint például a következő osztályban:

```
public class EchoBean {
     String echo(String name) {
     return name + " " + name;
```

```
    }
}
```

Fordítási időben, ki lehet echozni a kódot a következő képen:

```
EchoBean echo = new EchoBean();
String reply = echo.echo("Camel");
```

Ennek segítségével könnyedén megtudhatjuk, hogy az echo futás időben hívódott meg.
Más felől, az EchoBean segítségével is megoldható:

```
from("direct:start").bean(EchoBean.class,"echo").to("log:reply");
```

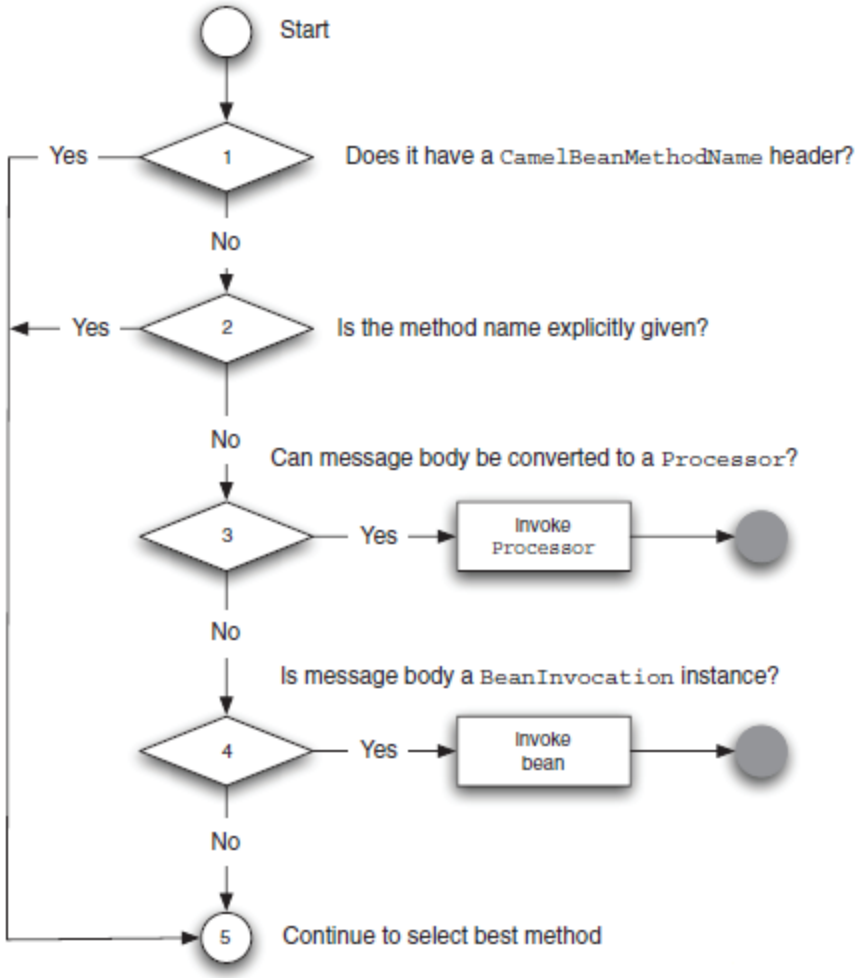
Amikro a fordító fordítja a kódot, nem látható ha az echo metódust hívjuk az EchoBeanen. A fordító szemszögéből az EchoBean.class és az echo a bean metódus paraméterei. Az egész fordító le tudja ellenőrizni, hogy az EchoBean osztály létezik. Ha elgépelt a metódus név, például „ekko” a fordító nem tudja lekezelni ezt a hibát. A hiba futásidőben váltódik ki amikor a BeanProcessor MethodNotFoundExceptiont dob, hogy az „ekko” nevű név nem létezik.
A Camel lehetővé teszi, hogy ne nevezzük el a metódusainkat. Például a következő útvonal bemutatja ezt:

```
from("direct:start").bean(EchoBean.class).to("log:reply");
```

A Camel metódus választó algoritmusa

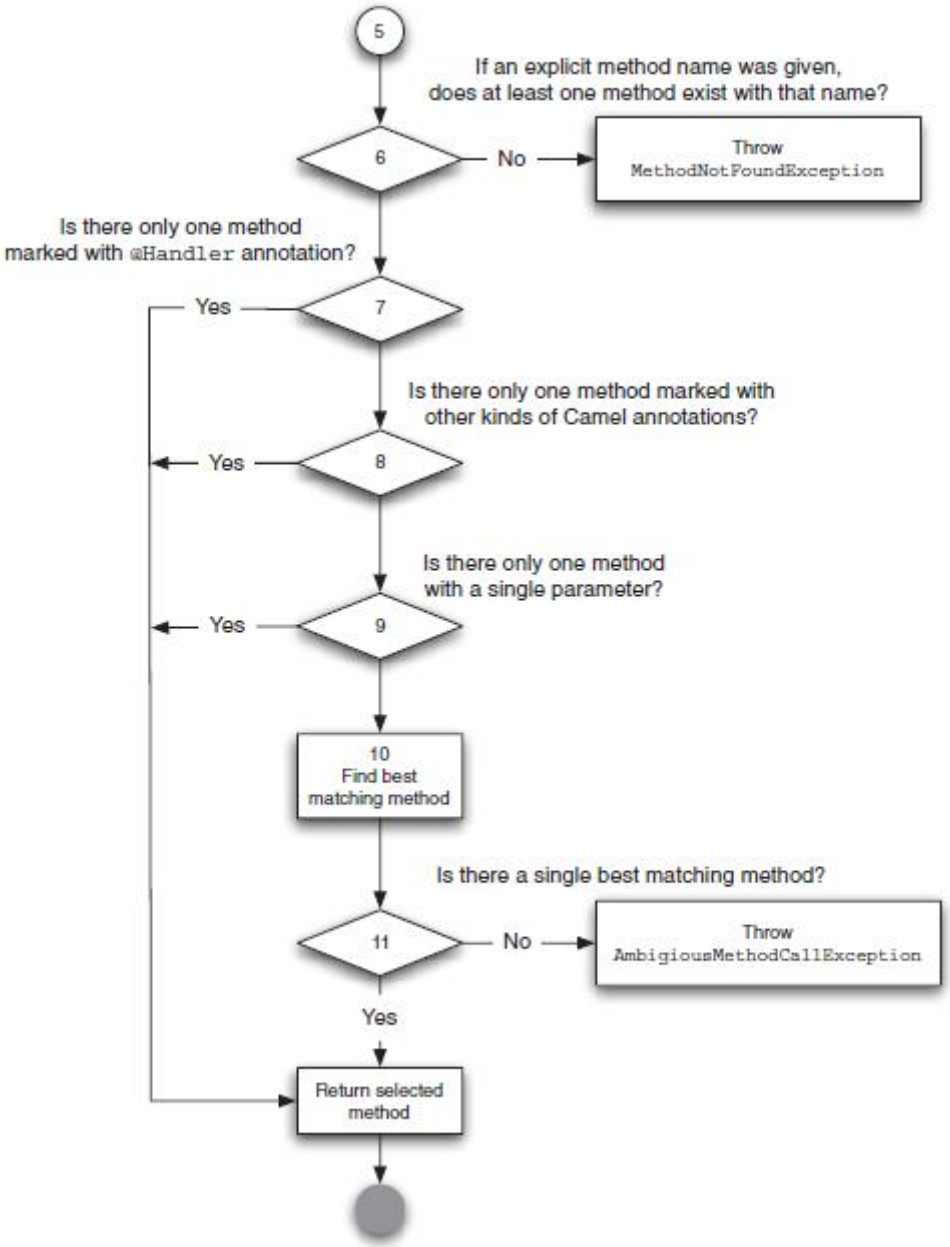
A BeanProcessor egy komplex algoritmust használ, hogy kiválassza melyik metódus hívódjon meg a beanen. Nem létfontonsságú minden lépést megérteni vagy emlékezni rájuk – csupán csak reprezentatív jellegű információ arról, hogyan dolgozik a Camel beanekkel a háttérben, mindezt elrejtve, hogy a fejlesztől csak a letisztult, és redukált kódot használhassák. Az alábbi ábra (5. ábra) az algoritmus első felét mutatja be és a folytatása a 6. ábra.
Az algoritmus a következő képen választja ki, hogy melyik metódus hívódjon meg:

1. Ha a Camel üzenet headerje tartalmazza a CamelBeanMethodNamet kulccsal, irány az ötödik lépés.
2. Ha a metódus explicite definiált, a Camel ezt felhaszálja és irány az ötödik lépés



5. ábra - Ahogy a Camel kiválasztja melyik metódus hívodjon meg. 1. rész

3. Ha a bean konvertálható, a Processor a Camel tipus-konverter segítségével feldolgozza az üzenetet. Ez lehet kicsit furcsának hangzik, de ez teszi lehetővé hogy a Camel bármelyik beant átváltoztassa üzenet vezérelet beanné (Message Driven Bean). Például ezzel a technikával, a Camel bármelyik javax.jms.Message-Listener beant közvetlenül hívhatja, bármilyen integráció nélkül. Ezt a metódust ritkán használják, de elég hasznos trükk lehet.
4. Ha a Camel üzenet törzse konvertálható org.apache.camel.component.bean. BeanInvocation formára, akkor a metódus meghívódik és továbbítja az argumentumokat a beannek. Ezt is elég ritkán szokták használni.
5. Az algoritmus második felével folytatódva az 6. ábrán.



6. ábra Ahogy a Camel kiválasztja melyik metódus hívodjon meg. 2. rész

6. Ha a metódus nevéhez nem tartozik létező név, NotFoundException kivételt dob.
7. Ha csak egy metódus van ellátva @Handler annotációval, azt választja ki.
8. Ha csak egy metódus is használ, minden egyéb Camel bean paraméterrel ellátott annotációt, mint például a @Body, @Header, stb, akkor azt választja ki.
9. Ha a bean metódusai között, ha csak egynek is van egy paramétere azt választja ki. Például ez a szituáció volt az EchoBean példánál, aminek csak egy echo metódusa volt, egy paraméterrel. Az egyedül álló paraméterű metódusok preferáltabbak mert könnyen mappolhatók adatcserénél.

10. Most a számítás már kezd komplex lenni. Már többszörös metódus jelöltek vannak és a Camelnek el kell döntenie, hogy melyik metódus a legjobb megoldás. A stratégia, hogy végig megy a metódus jelölteken, és kiszüri a nem illeszkedő metódusokat. A Camel ezt úgy végzi el, hogy az első paramétert össze hasonlítja a metódus jelölt első paraméterével, ha a paraméter nem azonos típusú, és ha a kényszerített típus konverzió sem lehetséges, a metódus „kiesik”. A végén egyetlen egy metódus marad ami a megfelelő lesz.
11. Ha a Camel még sem talál illeszkedő metódust, AmbigiousMethodCallException kivételt dob, a kétértelmú metódusokkal.

Most már tisztábban látható, hogyan választja ki a Camel a megfelelő metódust. A következőekben bemutatom, hogyan működik élesben az algoritmus.

Néhány metódus választó példa

Ha látni szeretnénk hogyan működik az algoritmus, a korábban használt EchoBeant elővesszük, de hozzá kell adni egy másik metódust – a bar metódust – a többszörös metódus jelölt kiválasztás érdekében.

```
public class EchoBean {
     public String echo(String echo) {
          return echo + " " + eco;
     }

     public String bar() {
          return "bar";
     }
}
```

A következő útvonallal fogjuk kezdeni:

```
from("direct:start").bean(EchoBean.class).to("log:reply");
```

Ha küldön egy String üzenetet „Camel” a Camel útvonalnak, a válasz naplózó „Camel Camel” ír a kimenetre. Annak a ténynek az ellenére, hogy a EchoBeannek két metódusa, echo és bar, csakis az echo metódusnak van egyedül álló paramétere. Ez az ami a 9. lépésben volt megfogalmazva – A Camel azt a metódust fogja választani aminek egy paramétere van.

Ahhoz, hogy ’versengés’ még kiegyenlítettebb legyen, változtatunk a bar metóduson a következő képen:

```
public String bar(String name) {
     return "bar " + name;
}
```

Jelenleg két metódus van aminek egy paramétere van. Ebben az esetben a Camel nem tud választani a két megoldás közül ezért AmbigiousMethodCallException kivételt dob, mint a 11. lépésben.
Hogyan oldható fel ez a probléma? Az egyik megoldás, hogy a metódus nevével ellátott útvonalat adok meg:

```
from("direct:start").bean(EchoBean.class,
"bar").to("log:reply");
```

Létezik egy másik megoldás, amihez nem kell megadni a metódus nevét a routeban. Használható a @Handler annotáció, hogy kiválassza a metódust, amit könnyedén hozzáadhatunk a metódushoz. Ez egyszerűen értesíti a Camelt, hogy ezt kell választania:

```
@Handler
     public String bar(String name) {
          return "bar " + name;
}
```

Most már nem kapunk AmbigiousMethodCallException kivételt, az annotációnak köszönhetően.
A következő példában megváltoztatjuk az EchoBean osztályt, két metódussal amiknek különböző a paraméter típusai:

```
public class EchoBean {
     public String echo(String echo) {
          return echo + " " + echo;
     }

     public Integer double(Integer num) {
          return num.intValue() * num.intValue();
     }
}
```

Az echo metódus együtt működik a String, double, integer metódussal. Ha nem adjuk meg a nevét a BeanProcessor fog választani köztük futásidőben.
A 10-es lépésben a Camel lehetővé teszi, hogy melyik metódus essen ki. Több metódus jelöltek paraméter típussát összehasonlítja üzenet törzsének típusával.
Ebben a példában küldünk egy üzenetet az útvonalnak aminek a törzse String típusú és a következő szerepel benne: „Camel”. Nem nehéz kitalálni, hogy a Camel az echo metódust fogja kiválasztani mert az String típusú. Másik oldalról, ha küldünk egy üzenetet, melynek típusa Integer és értéke 5, a Camel a double típusú metódust fogja kiválasztani mert az használ Integer típust.

Potenciális metódus választó problémák

A következő problémák fordulhatnak elő, ha futás időben akarunk beant hívni:

- **A meghatározott metódus nem található:** Ha a Camel nem találja a metódust a meghatározott névvel, MethodNotFoundException kivételt dob. Ez csakis akkor történhet meg ha előre definiáltuk a metódus neveket.
- **Két értelmű metódus:** Ha a Camel nem tudja egyértelműen kiválasztani a metódust, AmbigiousMethodCallException kivételt dob, egy listával mellyen a szóba jöhető metódusok szerepelnek. Ez akkor történik ha, az explicit metódus nevet definiáltunk mivel a metódus felül lett írva, ami azt jelenti, hogy a beannek több metódusa van ugyan azzal a névvel; egyedül a paraméterek száma módosíthatja.
- **Típus konverzió hiba:** Mielőtt a Camel meghívja a kiválasztott metódust, muszáj konvertálnia az üzenetet mint amilyet a metódus paraméter szignatúrája megkíván. Ez NoTypeConversionAvailableException kivételt dobhat.

A következő szituációban nézzük meg az EchoBean osztályt:

```
public class EchoBean {
     public String echo(String name) {
          return name + name;
     }
     public String hello(String name) {
          return "Hello " + name;
     }
}
```

Először megkell határozni ami nem létezik:

```
<bean ref="echoBean" method="foo"/>
```

Most már hívható a foo metódus, de nem létezik ilyen metódus tehát a Camel MethodNotFoundException kivételt dob.
Egy másik példa:

```
<bean ref="echoBean"/>
```

Ebben az esetben, a Camel nem tudja eldönteni melyik metódust hívja meg, mivel mindkét metódus mind az echo, mind a hello metódus két értelmű. Tehát a Camel AmbigiousMethod-CallException kivételt dob a két metódus listájával együtt.
Az utolsó szituációban ez akkor történhet meg ha az üzenet törzse olyan információt tartalmaz ami nem konvertálható a kívánt típusra. Ehhez a következőt kell követni:

```
public class OrderServiceBean {
     public String handleXML(Document xml) {
          ...
     }
}
```

A beant a routeban így kell definiálni:

```
from("jms:queue:orders")
     .beanRef("orderService", "handleXML")
     .to("jms:queue:handledOrders");
```

A handleXML metódusnak szüksége van egy paraméterre aminek org.w3c.dom.Document a típusa, ami egy XML típus, de a JMS nem tartalmazhat XML adatatot, csak egyszerű szöveget, mint például a „Camel rocks”. Futás időben a következő error üzenetet kapjuk:

```
Caused by: org.apache.camel.NoTypeConversionAvailableException: No
type
converter available to convert from type: java.lang.byte[] to the
required type: org.w3c.dom.Document with value [B@b3e1c9
at
org.apache.camel.impl.converter.DefaultTypeConverter.mandatoryConv
ertTo
(DefaultTypeConverter.java:115)
at
org.apache.camel.impl.MessageSupport.getMandatoryBody(MessageSuppo
rt.java
```

```
:101)
... 53 more
Caused by: org.apache.camel.RuntimeCamelException:
org.xml.sax.SAXParseException: Content is not allowed in prolog.
at
org.apache.camel.util.ObjectHelper.invokeMethod(ObjectHelper.java:
724)
at
org.apache.camel.impl.converter.InstanceMethodTypeConverter.conver
tTo
(InstanceMethodTypeConverter.java:58)
at
org.apache.camel.impl.converter.DefaultTypeConverter.doConvertTo
(DefaultTypeConverter.java:158)
at
org.apache.camel.impl.converter.DefaultTypeConverter.mandatoryConv
ertTo
(DefaultTypeConverter.java:113)
... 54 more
```

A Camel megpróbálja konvertálni a javax.jms.TextMessage, org.w3c.dom.Document típussá, de hibát dob. Ebben az esetben a Camel kicsomagolja az errort és NoTypeConverterExceptiont dob.