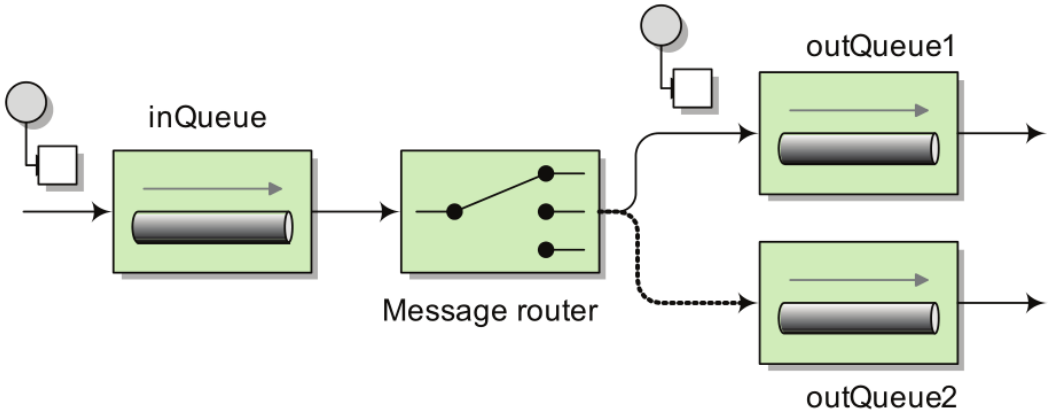
ÚTVÁLASZTÁS CAMEL-BEN

A Camel egyik legfontosabb szolgáltatása az útválasztás (routing), e nélkül a Camel gyakorlatilag csak egy egyszerű függvénykönyvtár lenne. Ebben a fejezetben a Camel útválasztásának rejtelmeiben merülhetünk el.

Mindennapi életünk során is gyakran találkozhatunk az útválasztással. Ha például elküldünk egy levelet a postán, annak valószínűleg sok városon keresztül „vezet az útja" mire a címzetthez ér. Egy elküldött email is számos számítógépes hálózaton keresztül érkezik meg a tényleges címzetthez. Mindkét esetben, az útválasztó (router) feladata az egyes üzenetek továbbítása.

A vállalati üzenetküldő rendszerek esetében az útválasztás egy olyan folyamat, melyben az üzenetet kivesszük a bemeneti sorról (`inQueue`), és bizonyos feltételek alapján továbbítjuk valamelyik kimeneti sorra (`outQueue1`, `outQueue2`), ahogy ez az alábbi ábrán látható. Ez gyakorlatilag azt jelenti, hogy a bemeneti és kimeneti sorok nincsenek tudatában a közöttük fennálló feltételeknek. Az útvonalválasztó feltételes logikája így szépen elkülönül az üzenet termelőjétől (producer) és fogyasztójától (consumer).



1. ábra: Egy útvonalválasztó üzeneteket fogad egy bemeneti csatornáról, és bizonyos feltételek alapján továbbítja azt az egyik kimenő csatornára

Az Apache Camel-ben az útválasztás ettől általánosabb fogalom. Definíciója szerint az üzenetek lépésenkénti mozgása, amely üzenetek egy consumer szerepkörű végponttól származnak. A fogyasztó kaphatja az üzenetet egy külső szolgáltatástól is, lekérdezheti azt időnként bármely rendszerből, de akár saját maga is létrehozhatja azt. Ez az üzenet aztán keresztülmegy egy feldolgozóegységen, amely lehet akár egy EIP (Enterprise Integration Pattern), egy feldolgozó, elfogó (interceptor), vagy bármi más is. Az üzenet végül eléri célját, egy termelő szerepkörű végpontot. Az útvonal tartalmazhat több feldolgozóegységet, melyek módosítják az eredeti üzenetet, vagy elküldik azt más helyre is, de az útvonalon lehet, hogy nincs egy feldolgozóegység sem, ekkor egy egyszerű csővezetéknek felel meg.

Ebben a fejezetben először egy képzeletbeli vállalatot mutatunk be, ezt fogjuk példaként használni a továbbiakban. A vállalati folyamatok támogatására megtanuljuk majd, hogy hogyan

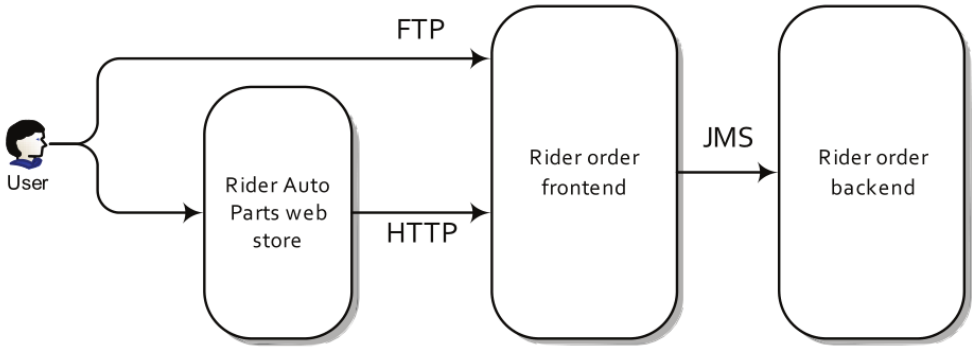
kommunikálhatunk FTP és Java Message Service (JMS) segítségével Camel végpontokat használva. Aztán részletesen bemutatjuk a Java és a Spring alapú konfigurációs útvonal létrehozó formátumokat. Továbbá betekintést nyerhetünk abba, hogy hogyan tervezzünk és implementáljunk megoldásokat a különböző vállalati integrációs problémákra EIP- és Camel felhasználásával. A fejezet végére elég gyakorlatot szerezhetünk, hogy elkezdjünk használható útvonalválasztási alkalmazásokat alkalmazásokat készíteni Camel-lel.

Kezdésként nézzük meg a képzeletbeli vállalatot, amellyel példálózni fogunk az egész könyvben!

A „Rider autóalkatrészek” bemutatása

A képzeletbeli motor alkatrészeket áruló cégünk, a Rider autóalkatrészek, motorgyártóknak szállít alkatrészeket. A cégnél az évek során többször is megváltozott a rendelések fogadásának módja. Először a rendeléseket egy CSV fájlban kapták, melyet a megrendelők feltöltöttek egy FTP szerverre. A formátum később megváltozott XML-re. Jelenleg a rendelések feladása egy honlapon keresztül történik, a rendelések XML formátumban utaznak HTTP protokoll segítségével.

A cég új ügyfelei a webes felületen keresztül végzik a megrendeléseket, de a régi ügyfelekkel fennálló megállapodások miatt fent kell tartani az összes régebbi interfészt is. Mindegyik üzenetfajta egy POJO-vá (Plain Old Java Object) konvertálódik a feldolgozás előtt. A rendeléseket feldolgozó rendszer működése látható nagyvonalakban az alábbi ábrán.



2. ábra: Az ügyfelek kétféleképpen küldhetik el rendelésüket a Rider-nek: feltölthetik a nyers fájlokat egy FTP szerverre, vagy elküldhetik azt a Rider webshopjában. A végén minden rendelést JMS-sel küldenek el feldolgozásra a backendnek.

A cég egy igen gyakori problémával áll szemben: az évek során egyre több, akkor éppen népszerű átviteli módszert és formátumot használ. Azonban ez nem okoz gondot egy olyan integrációs keretrendszernek, mint a Camel. Ebben a fejezetben és a könyv többi részében is az lesz az olvasó feladata, hogy segítsen a Rider autóalkatrészek cégnek az új követelmények és új funkciók megvalósításában, természetesen a Camel felhasználásával.

Első feladat az FTP modul lefejlesztése a megrendelőrendszer frontendjében. A fejezet további részében a backend szolgáltatások fejlesztését is bemutatjuk. Az FTP modul fejlesztése az alábbi lépésekből áll:

1. Az FTP szerverről bizonyos időközönként az új rendelések letöltése
2. Az új megrendelések fájljait JMS üzenetekké konvertálása
3. Az üzeneteket az `incomingOrders` JMS sorra elküldeni

Az 1. és 3. lépések megvalósításához tudnunk kell, hogy hogyan kommunikáljunk FTP-n és JMS-en keresztül Camel cégpontok segítségével. A feladat teljes elkészítéséhez ismernünk kell továbbá, hogyan működik az útvonalválasztás Java DSL-el. Először is lássuk, hogy hogyan használhatunk Camel végpontokat!

Ismerkedés a végpontokkal

Ahogyan a legelső fejezetben is olvashattuk, egy végpontot úgy képzelhetünk el, mint egy olyan csatorna végét, amelyen keresztül a rendszer képes üzeneteket küldeni vagy fogadni. Ebben a részben bemutatjuk, hogy hogyan használhatunk URI-kat a Camel konfigurálására FTP-vel és JMS-el. Lássuk először az FTP-t!

Fájlok és FTP

A Camel használatát nagyban megkönnyítik a végpont URI-k. Egy URI megadásával azonosíthatjuk a használni kívánt komponenst és annak konfigurációját. Ezután küldhetünk vagy fogadhatunk üzeneteket az ezzel az URI-val azonosított komponenstől.

Nézzük például az első Rider autóalkatrészekkel kapcsolatos feladatot. Az új megrendelések FTP szerverről való letöltéséhez a következőket kell tennünk:

1. Csatlakozzunk a rider.com című FTP szerver az alapértelmezett 21-es porton
2. Adjuk meg a „rider” felhasználónevet és a „secret” jelszót
3. Lépjünk az „orders” mappába
4. Töltsük le az új megrendeléseket tartalmazó fájlokat

Az alábbi ábrán látható, hogy milyen könnyen megtehetjük ezt Camel-lel URI leírást használva.

3. ábra: Egy Camel végpont három részből áll, ezek sorban: séma, kontextus útvonal, és opciólista.

A Camel először az `ftp` sémát keresi ki a komponens jegyzékből, amely visszaadja az `FtpComponent`-et. Az `FtpComponent` aztán létrehoz egy `FtpEndpoint`–ot az URI további részeinek felhasználásával.

A `rider.com/orders` kontextus útvonal (context path) alapján az `FtpComponent` bejelentkezik a rider.com FTP szerverre az alapértelmezett porton és belép az „orders” mappába. Végül az opciókként megadott `username` és `password` mondják meg, hogyan kell bejelentkezni az FTP szerverre.

TIPP Az FTP komponensnél a felhasználónevet és a jelszót megadhatjuk az URI kontextus útvonal részében is. Az alábbi URI így megfelelel a 2.3 ábrán szereplőnek.

Az `FtpComponent` nem része a Camel magjának, ezért hozzá kell azt adnunk függőségként a projekthez. Maven használatakor csak az alábbi függőséget kell hozzáadnunk a POM-hoz:

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-ftp</artifactId>
   <version>2.5.0</version>
</dependency>
```

Habár ez az URI ugyanilyen jól működne termelő-fogyasztó esetben is, most arra fogjuk használni, hogy rendeléseket töltsünk le az FTP szerverről. Ehhez a Camel DSL egy `from` node-jában kell felhasználnunk:

```
from("ftp://rider.com/orders?username=rider&password=secret")
```

Mindösszesen csak ennyi kell ahhoz, hogy fájlokat fogadhassunk egy FTP szervertől.

Következő feladatunk, hogy az FTP szerverről letöltött rendeléseket elküldjük egy JMS sorra. Ehhez egy kicsit több előkészületre van szükség, de ez sem lesz túl nehéz.

Üzenet küldése egy JMS sorra

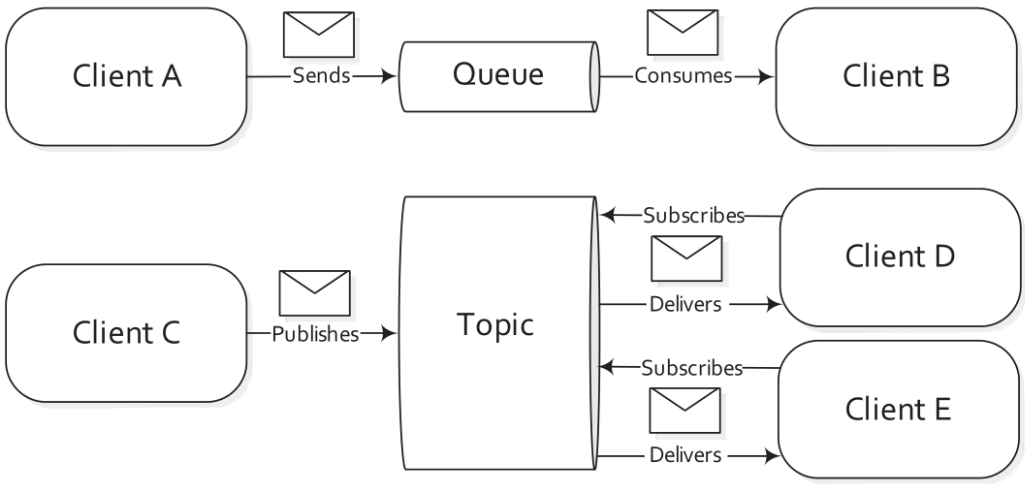
A Camel széleskörű támogatást nyújt a JMS-hez, ezt részletesen a 7. fejezetben fogjuk tárgyalni. Jelenleg csak annyit mutatunk be ebből, ami az első Rider feladat teljesítéséhez szükséges. Ha már nem emlékeznénk, az FTP szerverről letöltött rendelések adatait kell továbbítanunk egy JMS sorra.

MI IS AZ A JMS?

A JMS (Java Message Service) egy Java API, amely segítségével üzeneteket küldhetünk, fogadhatunk, ill. hozhatunk létre. Biztosítja továbbá az aszinkron üzenetkezelést, és bizonyos szintű megbízhatóságot, mint például hogy garantált kézbesíti az üzeneteket, és ezt csakis egyszer teszi meg. A JMS az egyeduralkodó üzenetkezelő rendszer a Java világában.

JMS, az üzenet termelők és fogyasztók egy közvetítőn, egy JMS állomáson (destination-ön) keresztül beszélnek egymással. Mint ahogy az alábbi ábra is bemutatja, egy állomás lehet egy queue (sor), vagy egy topic (téma). A queue-k szigorúan pont-pont kapcsolatot biztosítanak, ahol minden üzenetnek csak egy címzettje van. A topic-ok a publish/subscribe (közzététel/feliratkozás) séma alapján működnek, egy üzenet több fogyasztóhoz is eljuthat ha feliratkoztak a topic-ra.

A JMS biztosít egy `ConnectionFactory`-t is, mely segítségével a kliensek (mint például a Camel) kapcsolódhatnak egy JMS szolgáltatóhoz (provider). A JMS szolgáltatókat általában inkább broker-eknek (ügynököknek) nevezzük, mivel ők kezelik az üzenet feladója és címzettje közötti kommunikációt.



4. ábra: Kétféle JMS destination létezik: queue és topic. A queue (sor) egy pont-pont csatorna, ahol minden üzenetnek csak egy címzettje van. Egy topic minden viszont minden feliratkozott címzettnek eljuttatja az üzenet egy másolatát.

JMS PROVIDER BEÁLLÍTÁSA CAMEL-BEN

Ahhoz, hogy a Camel-t összekössük egy JSM provider-rel, be kell állítani a Camel JMS komponensét egy megfelelő `ConnectionFactory`-vel.

Az Apache ActiveMQ az egyik legnépszerűbb nyílt forráskódú JMS provider, valamint a Camel fejlesztői is ezt használják a JMS komponensük teszteléséhe, ezért a könyvben is ezt fogjuk használni a JMS alapjainak bemutatásához. Ha többet szeretne tudni az ActiveMQ-ról, ajánljuk az *Active MQ in Action* c. könyvet (írta Bruce Snyder, DejanBosanac, és Rob Davies, kiadja a Manning Publications).

Tehát ActiveMQ esetén egy ActiveMQConnectionFactory objektumot kell létrehoznunk, amely a futó ActiveMQ broker-re mutat.

```
ConnectionFactory connectionFactory =
  new ActiveMQConnectionFactory("vm://localhost");
```

A `vm://localhost` URI azt jelenti, hogy a „localhost” nevű, az aktuális JVM-en belüli beépített broker-hez szeretnénk csatlakozni. ActiveMQ-ban a `vm` csatoló igény szerint létrehoz egy broker-t ha még nem fut, ezért nagyon egyszerűen és gyorsan használható JMS alkalmazások tesztelésére, de „éles” környezetben ajánlott egy már futó broker-hez csatlakozni. Éles környezetben ajánlott továbbá a connection pool-ok használata (a 7. fejezetben olvashat erről bővebben).

Amikor létrehozzuk a CamelContext objektumot, hozzáadhatjuk a JMS komponenst az alábbiak szerint:

```
CamelContext context = new DefaultCamelContext();
context.addComponent("jms",
  JmsComponent.jmsComponentAutoAcknowledge(connectionFactory));
```

A JMS és az ActiveMQ komponensek nem részei az alap Camel-nek, ezért használatukhoz hozzá kell adnunk néhány függőséget a Maven projekthez. Az alap JMS kompenenshez a következőt:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jms</artifactId>
  <version>2.5.0</version>
</dependency>
```

A ConnectionFactory az ActiveMQ része, tehát ehhez a következő függőség szükséges:

```
<dependency>
<groupId>org.apache.activemq</groupId>
<artifactId>activemq-core</artifactId>
<version>5.3.2</version>
</dependency>
```

Miután megfelelően konfiguráltuk a JMS komponenst egy JMS broker-hez való csatlakozásra, nézzük, hogy hogyan használhatunk URI-kat az egyes destination-ök megadására.

URI, MINT DESTINATION

Miután a JMS komponenst konfiguráltuk, már simán küldhetünk és fogadhatunk JMS üzeneteket. Mivel URI-kat használunk, nagyon könnyen használhatjuk ezt is.

Tegyük fel, hogy egy JMS üzenetet szeretnénk küldeni az `incomingOrders` nevű queuera, ekkor a megfelelő URI így néz ki:

```
jms:queue:incomingOrders
```

Magától értetődően, a „jms" rész azt jelenti, hogy az előbb konfigurált JMS komponenst szeretnénk használni. Az URI további része pedig azt jelenti, hogy az `incomingOrders` nevű queue-nak küldjük az üzeneteket. A queue minősítőt el is hagyhattuk volna, mivel ez az alapértelmezett.

> **MEGJEGYZÉS** Néhány végpontnak rengeteg opciója, beállítása lehet. Például a JMS komponensnek kb. 60 opciója van, azonban ezek többségét csak speciális esetekben kell megadnunk. A Camel fejlesztői törekedtek arra, hogy az alapértelmezett beállítások az esetek többségében megfelelőek legyenek. Az alapértelmezett beállításokat mindig megtekintheti a megfelelő komponens online dokumentációjában. A JMS komponens dokumentációját itt találja: http://camel.apache.org/jms.html.

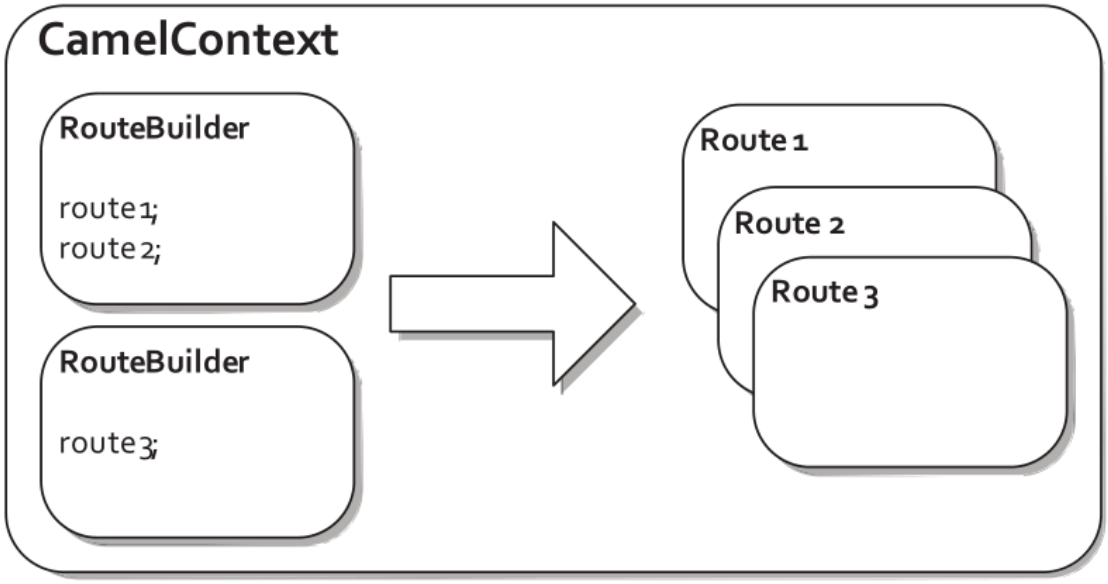
Java DSL-ben az `incomingOrders` queue-ra a `to` parancs segítségével az alábbi módon küldhetünk egy üzenetet:

```
...to("jms:queue:incomingOrders")
```

Most hogy már megtanultuk az FTP és JMS és kommunikáció alapjait Camel-ben, térjünk vissza a fejezet eredeti témájához, az útválasztáshoz!

Útvonalak létrehozása Java-ban

Az első fejezetben láthattuk, hogy az egyes `CamelContext` objektumok több útvonalat tartalmaznak, és hogy hogyan hozhatunk létre útvonalakat a `RouteBuilder`-rel. Azonban lehet hogy nem volt egyértelmű, hogy egy `RouteBuilder` nem határozza meg a `CamelContext`-ben a használt futásidejű útvonalat, a `RouteBuilder` csak létrehoz egy vagy több útvonalat, és ezek az útvonalak belekerülnek a `CamelContext` –be (lásd az alábbi ábrát).



5. ábra: Camelben a RouteBuilder-ek segítségével hozhatunk létre útvonalakat, egy RouteBuilder-rel akár többet is.

A `CamelContext addRoutes` metódusának megadhatunk bármilyen objektumot amely a `RoutesBuilder` interfészt implementálja. Ez az interfész egy metódust tartalmaz:

```
void addRoutesToCamelContext(CamelContext context) throws Exception;
```

Tehát akár írhatunk egy saját osztályt is Camel útvonalak létrehozására. Azonban általában a beépített `RouteBuilder` osztályt használják (természetesen ez is implementálja a `RoutesBuilder` interfészt). A `RouteBuilder` biztosítja továbbá DSL-t az útvonalak létrehozásához.

A következő részben megtanuljuk, hogyan használhatjuk a `RouteBuilder`-t és Java DSL-t egyszerű útvonalak létrehozására. Ezután megtudhatjuk, hogyan használjuk az útvonalakat a Spring DSL-ben és EIP-knél.

A RouteBuilder használata

Camel-ben igen gyakran használjuk az `org.apache.camel.builder.RouteBuilder` absztrakt osztályt. Minden alkalommal ezt használjuk amikor útvonalakat kell létrehoznunk Java-ban. Ehhez készítsünk egy új osztályt, amely a `RouteBuilder`-ből származik, és implementáljuk a `configure` metódust:

```
class MyRouteBuilder extends RouteBuilder {
  public void configure() throws Exception {
    ...
  }
}
```

Ezután adjuk hozzá a `CamelContext`-hez az `addRoutes` metódus segítségével:

```
CamelContext context = new DefaultCamelContext();
context.addRoutes(new MyRouteBuilder());
```

Vagy használhatunk akár egy anonymous osztályt is:

```
CamelContext context = new DefaultCamelContext();
context.addRoutes(new RouteBuilder() {
  public void configure() throws Exception {
    ...
  }
});
```

A `configure` metódusban az útvonalakat Java DSL segítségével hozzuk létre. A DSL-t részletesen a következő részben fogjuk tárgyalni, de alapok megismeréséhez most hozzunk létre egy egyszerű útvonalat.

Az első fejezet olvasása közben már letöltöttük a könyv honlapját és beállítottuk az Apache Maven-t. Ha mégsem tette meg, kérjük tegye meg most. Lépjünk a chapter2/ftp-jms mappába a konzolban, és írjuk be a következő parancsot:

```
mvn eclipse:eclipse
```

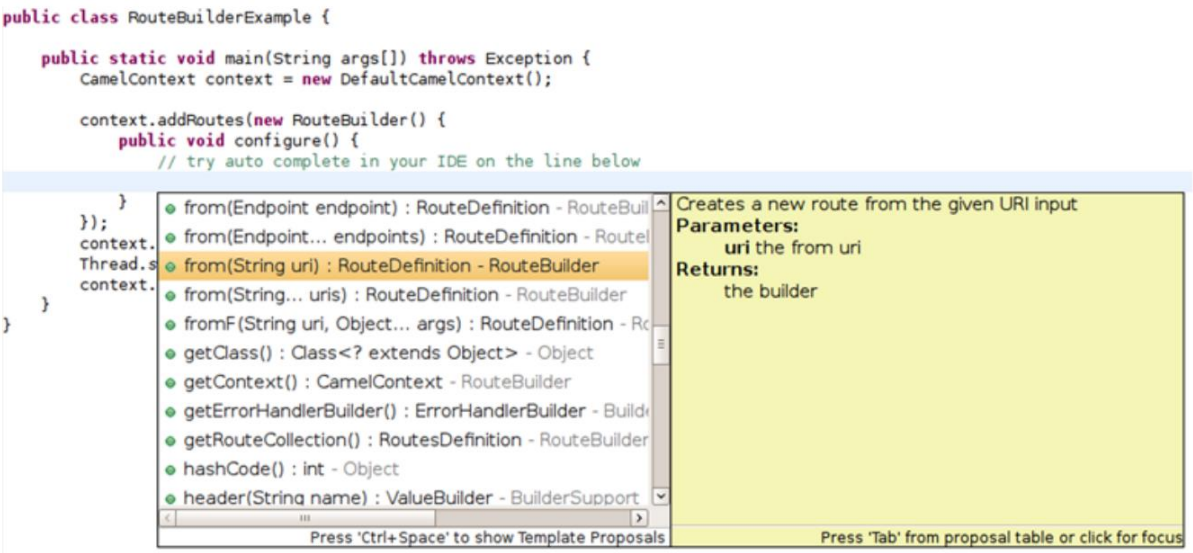
Ez létrehoz egy Eclipse projekt fájlt.

MEGJEGYZÉS Az Eclipse (http://www.eclipse.org/) egy népszerű, nyílt forráskódú integrált fejlesztőkörnyezet. A könyvben az Eclipse 3.5.2 verzióját használjuk.

Amikor lefutott a fenti parancs, importáljuk be a létrehozott projektet Eclipse-be a *File >Import > Existing Projects into Workspace* menüpont segítségével. Camel projektek Eclipse-ben való fejlesztéséről a 11. fejezetben olvashat részletesebben.

MEGJEGYZÉS Nem szükséges integrált fejlesztőkörnyezetet használnia Camel fejlesztéshez, de ezek használata nagyban megkönnyíti a munkát. Ha mégsem szeretne integrált fejlesztőkörnyezetet használni, nyugodtan kihagyhatja a következő részt.

Ha betöltődött Eclipse-be az ftp-jms projekt, nyissuk meg az src/main/java/camelinaction/RouteBuilderExample.java fájlt. Ahogy az alábbi ábrán is látszik, az automatikus kódkiegészítés (Ctrl+Space) felhoz jó néhány lehetséges metódust. Egy útvonal elkezdéséhez a `from` metódust használhatjuk.



6. ábra: Használjuk az automatikus kódkiegészítést útvonalak írásához. Minden útvonal a from metódussal kezdődik.

A `from` metódus egy végpont URI-t vár paraméterként. A Rider autóalkatrészek FTP szerveréhez a következőképpen csatlakozhatunk:

```
from(”ftp://rider.com/orders?username=rider&password=secret”)
```

A `from` metódus egy `RouteDefinition` objektummal tér vissza, mely számos különböző metódust nyújt a különböző EIP-k vagy egyéb üzenetkezelési módok megvalósítására.

Gratulálunk, Ön most már sikeresen használja a Java DSL-t! Most nézzük meg egy kicsit alaposabban, hogy mi is történik a hattérben.

A Java DSL

A terület specifikus nyelvek (Domain-specific languages - DSL) számítógépes nyelvek, amelyek egy speciális területre koncentrálnak, szemben az általános célú programozási nyelvekkel.

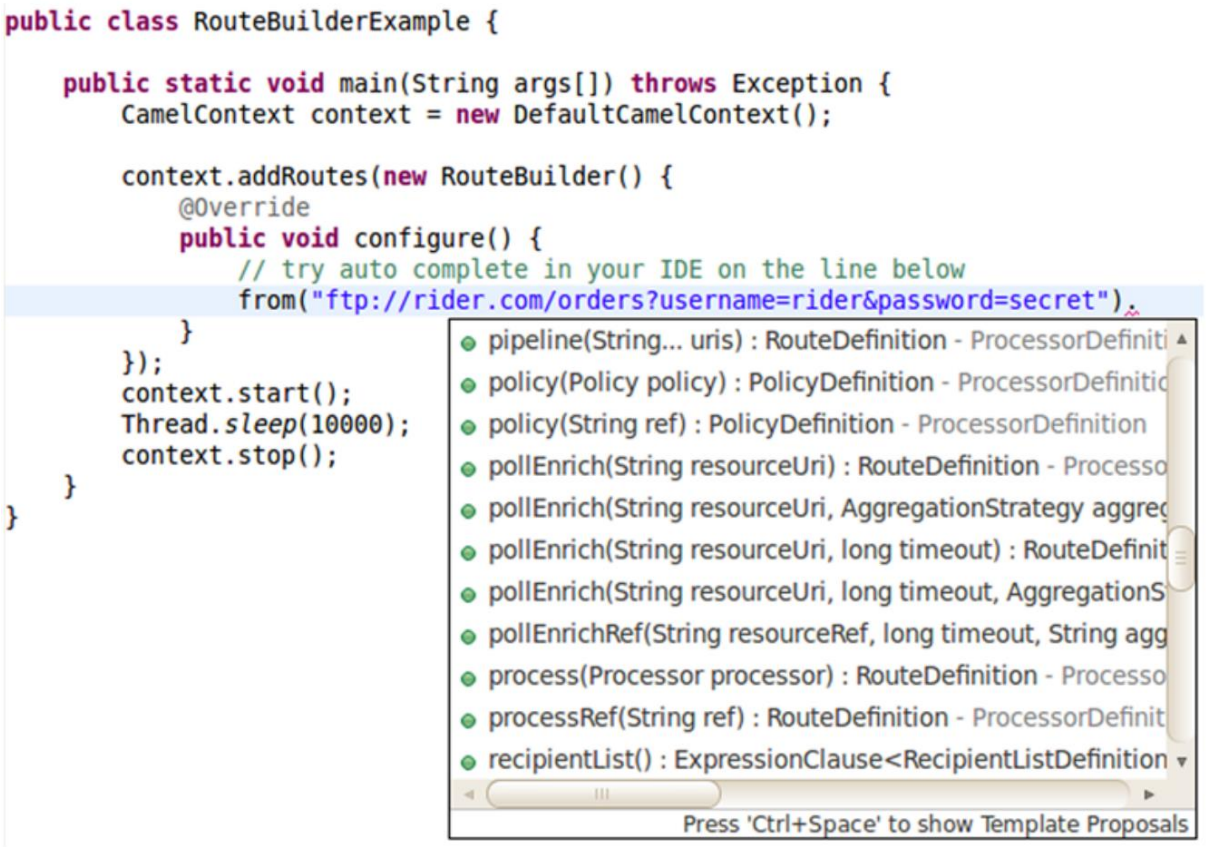
Például már biztosan használta a reguláris kifejezések tiszta és tömör DSL nyelvét, amely szövegek illesztésére és keresésére szolgál. Java-ban a szövegek illesztése és keresése már nem lenne olyan egyszerű. A reguláris kifejezések külső DSL-t felelnek meg – egyedi szintaxissal rendelkeznek és így egy külön fordítóra vagy értelmezőre van szükség a futtatásukhoz.

A belső DSL-ek ezzel szemben egy már meglévő általános célú nyelvet használnak (mint például a Java), így a DSL egy különálló területről jövő nyelvnek tűnik. Ezt legkézenfekvőbben a metódusok és azok argumentumainak olyan elnevezésével valósíthatjuk meg, amely megfelel a szóban forgó célterületen használt fogalmaknak.

Másik népszerű megvalósítása a belső DSL nyelveknek a „folyékony” interfészek (fluent interfaces) használata. Ilyenkor úgy hozunk létre objektumokat, hogy meghívunk bizonyos metódusokat amik végrehajtanak egy műveletet, de vissza is adják az adott objektum példányt (`return this;`), a következő metódust ezen a példányon hívjuk, és így tovább.

MEGJEGYZÉS Ha többet szeretne megtudni a belső DSL-ekről, akkor Martin Fowler „Domain Specific Language" bejegyzését ajánljuk bliki (blog + wiki) oldalán: http://www.martin-fowler.com/bliki/DomainSpecificLanguage.html. Írt a folyékony interfészekről is: http://www.martinfowler.com/bliki/FluentInterface. Ha a DSL-ek érdeklik általánosabban, Debasish Ghosh Manning Publications-nél megjelent *DSLs in Action* c. könyvét ajánljuk.

A Camel célja a vállalati integráció, ezért a Java DSL-je gyakorlatilag folyékony interfészek halmaza, melyekben az elnevezések az EIP könyvet követik. Nézzük meg Eclipse-ben, hogy a kódkiegészítés milyen lehetőségeket ad a `from` metódus után a `RouteBuilder`-ben! Hasonlót kell látnia, mint ami az alábbi ábrán szerepel. Az ábrán láthatunk jó néhány EIP-t – Pipeline, Enricher, és Recipient List – a többiekről a későbbiekben lesz szó.



7. ábra: A from metódus után a fejlesztőeszköz kódkiegészítés funkcióját a használható EIP-k (mint például Pipeline, Enricher, Recipient List) és egyéb hasznos integrációs funkciók listáját láthatjuk.

Egyelőre válasszuk a `to` metódust és zárjuk le az útvonalat egy pontosvesszővel. A `RouteBuilder`-ben Minden `from`-mal kezdődő Java utasítás egy új útvonalat készít. Ezzel az új útvonallal valósítjuk meg az első Rider-es feladatunkat – rendelések felvétele az FTP szerverről és azok elküldése az `incomingOrders` nevű JMS sorra.

A feladat kész megoldását megtalálhatja a könyvhez kapcsolt forráskódok között az alábbi útvonalon: chapter2/ftp-jms/src/main/java/camelinaction/FtpToJMSExample.java.

```
import javax.jms.ConnectionFactory;

import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.camel.CamelContext;
import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.component.jms.JmsComponent;
import org.apache.camel.impl.DefaultCamelContext;

public class FtpToJMSExample {

    public static void main(String args[]) throws Exception {
        CamelContext context = new DefaultCamelContext();

        ConnectionFactory connectionFactory =
            new ActiveMQConnectionFactory("vm://localhost");
        context.addComponent("jms",
            JmsComponent.jmsComponentAutoAcknowledge(connectionFactory));

        context.addRoutes(new RouteBuilder() {
            public void configure() {
                from("ftp://rider.com/orders"
                  + "?username=rider&password=secret")
                  .to("jms:incomingOrders");
            }
        });

        context.start();
        Thread.sleep(10000);

        context.stop();
    }
}
```

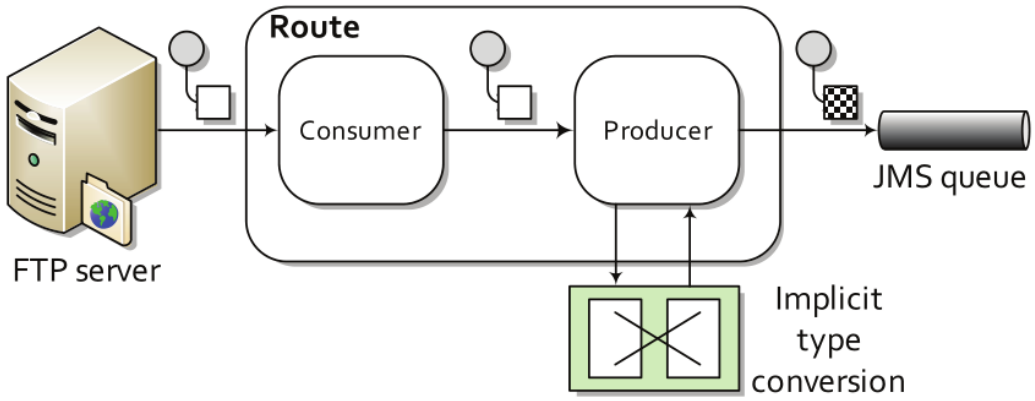
1 **Java statement that forms a route**

8. ábra: Letöltés az FTP szerverről és az üzenetek átadása az incomingOrdes queue-nak

MEGJEGYZÉS Mivel a példában az FTP szerver címe ftp://rider.com, ami a valóságban nem létezik, ezért nem futtatható ez a példa ebben a formában, a Java DSL nyelvének bemutatására szolgál. Működő FTP példákat a 7. fejezetben találhat.

Mint láthatjuk, a fenti kódrészlet tartalmaz sok sablonos beállítási és konfigurációs részt is, a probléma tényleges megoldása azonban a `configure` metóduson belül röviden, egy darab Java utasítás formájában történik. A `from` metódus adja meg a Camel-nek hogy egy FTP szerverről fogadja az üzeneteket, és a `to` metódus pedig az üzenetek egy JMS végpontra továbbítására utasítanak.

Az üzenetfolyam ezen az egyszerű útvonalon felfogható egy egyszerű csővezetékként is, ahol a fogyasztó kimenete a termelő bemenetére van kötve közvetlenül, ezt láthatjuk az alábbi ábrán.



9. ábra: A fenti kódrészlettel elkészített útvonal egy egyszerű csővezetéket képez. Az FTP fogyasztó kimenete a JMS termelő bemenetére van kötve. A letöltött fájlok JMS üzenetekké konvertálása automatikusan történik.

Mint már lehet hogy fel is tűnt Önnek, a genti példában nem végeztünk semmilyen külön konverziót az FTP-ről letöltött fájlok JMS üzenetekké alakítása érdekében – ezt megtette helyettünk automatikusan a Camel beépítet `TypeConverter` funkciója. Bár megadhatunk típus konverziókat bárhol egy útvonalban, gyakran erre egyáltalán nincs szükség. Az adatok átalakításáról és a típuskonverziókról részletesen a 3. fejezetben lesz szó.

Lehet, hogy szívesen megnézni mi történik ezen az egyszerű kis útvonal belsejében. A Camel ehhez szerencsére biztosít is különböző funkciókat. Ennek egy igazán egyszerű módja a feldogozók (processor) használata, nézzük is meg hogyan működik ez.

PROCESSOR-OK HOZZÁADÁSA

A `Processor` interfész Camel-ben a bonyolult útvonalak egyik igen fontos építőeleme. Ez egy egyszerű interfész, egyetlen metódussal:

```
public void process(Exchange exchange) throws Exception;
```

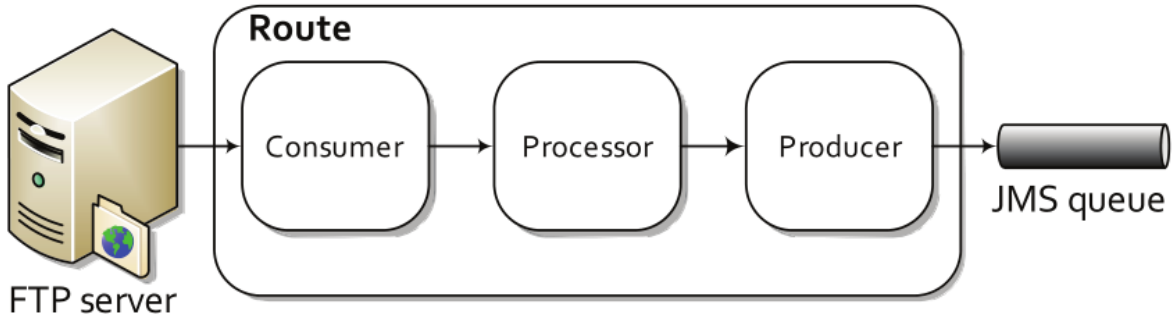
Ezzel teljes hozzáférést nyerhetünk a teljes üzenetcseréhez, szinte bármit megtehetünk magával az üzenettel vagy a fejléceivel.

Camel-ben minden EIP gyakorlatilag egy processor. Az útvonalhoz akár az alábbi módon is hozzáadhatunk egy új, saját processor-t:

```
from("ftp://rider.com/orders?username=rider&password=secret").
process(new Processor() {
  public void process(Exchange exchange) throws Exception {
    System.out.println("We just downloaded: " +
      exchange.getIn().getHeader("CamelFileName"));
  }
}).
to("jms:incomingOrders");
```

Ez az útvonal így kiírja majd a megrendeléseket tartalmazó fájl nevét mielőtt elküldi az üzenetet a JMS sorra.

Mivel a processor-t az útvonal középe raktuk, így tulajdonképpen hozzáadtuk az előzőekben említett csővezetékhez, ezt láthatjuk az alábbi ábrán.



10. ábra: Egy processor hozzáadásával az FTP fogyasztó kimenete a consumer-re, annak kimenete pedig a JMS producer-re kerül.

> **MEGJEGYZÉS** Sok komponens, így köztük a `FileComponent` és az `FtpComponent` is, hasznos, az üzenetekre jellemző fejléceket állít be. Az előző példában a `CamelFileName` fejlécet használtuk az FTP-ről letöltött fájlnevének lekérdezésére. Az online dokumentációban minden komponensnél megtalálhatja, hogy milyen fejléceket bocsájtanak rendelkezésre. Az FTP komponensről például a http://camel.apache.org/ftp2.html címen találhat információkat.

Camel-ben az útvonalak létrehozásának legfőbb módja a Java DSL nyelv, amely így a Camel magjának része is. Vannak azonban egyéb módszerek az útvonalak létrehozására, ezek lehet, hogy egy-egy adott szituációhoz jobban megfelelőek. Például létrehozhatunk útvonalakat Groovy-ban, Scala-ban, és a következőkben tárgyalt Spring XML-ben is.

Útvonalak létrehozása Spring-ben

A Spring a legnépszerűbb Inversion of Control (IoC) Java kénténer. Az alap keretrendszer segítségével „összeköthetünk" bean-eket az alkalmazások elkészítéséhez. Ez az összekötés egy XML konfigurációs fájl segítségével történik.

Ebben a részben egy rövid betekintést nyújtunk, hogy hogyan is kell alkalmazásokat elkészíteni a Spring segítségével. A Spring széleskörűbb áttekintéséhez ajánljuk Craig Walls *Spring in Action* c. könyvét (http://www.manning.com/walls4/).

Aztán bemutatjuk, hogy hogyan használhatjuk Camel-ben a Spring-et a Java DSL helyett vagy mellett.

Bean injection („befecskendezés") és a Spring

Alkalmazások készítése bean-ekből Spring-gel igazán egyszerű. Mindössze csak néhány bean-re (osztályra), egy Spring XML konfigurációs fájlra és egy `ApplicationContext` objektumra van szükségünk. Az `ApplicationContext` hasonló a `CamelContext`-hez, mivel ez a Spring futásidejű konténere. Nézzünk most egy egyszerű példát!

Vegyünk egy alkalmazást, amely kiír egy üdvözlőüzenetet és azután egy felhasználónevet. Nem szeretnénk az alkalmazásba beégetni az üdvözlést, ezért ennek kiváltására az alábbi interfészt használhatjuk:

```
public interface Greeter {
  public String sayHello();
}
```

Ezt az interfészt a következő osztályok implementálják:

```
public class EnglishGreeter implements Greeter {
  public String sayHello() {
    return "Hello " + System.getProperty("user.name");
  }
}
public class DanishGreeter implements Greeter {
  public String sayHello() {
    return "Davs " + System.getProperty("user.name");
  }
}
```

Most készítsük el magát az alkalmazást:

```
public class GreetMeBean {
  private Greeter greeter;

  public void setGreeter(Greeter greeter) {
    this.greeter = greeter;
  }

  public void execute() {
    System.out.println(greeter.sayHello());
  }
}
```

Ez az alkalmazás más és más üdvözlést fog kiírni attól függően, hogy hogyan konfiguráljuk. Egy példa Spring konfigurációt láthatunk az alábbiakban:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

  <bean id="myGreeter" class="camelinaction.EnglishGreeter"/>

  <bean id="greetMeBean" class="camelinaction.GreetMeBean">
    <property name="greeter" ref="myGreeter"/>
  </bean>
```

```
</beans>
```

Ezzel az XML fájllal a következőkre utasítjuk a Spring-et:

1. Hozza létre az `EnglishGreeter` osztály egy példányát és nevezze el `myGreeter`-nek
2. Hozza létre az `GreetMeBean` osztály egy példányát és nevezze el `greetMeBean`-nek
3. Állítsa be a `greetMeBean` `greeter` property-jének a `myGreeter`-t

A beanek ezen konfigurációját nevezzük wiring-nek (összekötésnek).

Az XML fájl Spring-be való betöltéséhez használhatjuk a `ClassPathXmlApplicationContext`-et, amely az ApplicationContext keretrendszer által biztosított implementációja. Ennek segítségével betölthetjük a classpath-on levő Spring konfigurációs fájlokat, így a `GreetMeBean` végső változata:

```
public class GreetMeBean {
  ...
  public static void main(String[] args) {
    ApplicationContext context =
      new ClassPathXmlApplicationContext("beans.xml");

    GreetMeBean bean = (GreetMeBean) context.getBean("greetMeBean");
    bean.execute();
  }
}
```

A példányosított `ClassPathXmlApplicationContext` objektum betölti az előbb bemutatott konfigurációs beállításokat a beans.xml fájlból. Aztán ezen meghívjuk a `getBean` metódust, hogy megkeresse nekünk a `greetMeBean` ID-vel ellátott bean-t a Spring registry-ből. Ezzel a módszerrel minden, a beans.xml-ben definiált bean-t elérhetünk.

A példa futtatásához lépjünk a forráskódokat tartalmazó mappán belül a chapter2/spring mappába, és futtassuk le az alábbi Maven parancsot:

```
mvn compile exec:java -Dexec.mainClass=camelinaction.GreetMeBean
```

Egy ehhez hasonló kimenetet várhatunk:

```
Hello janstey
```

Ha a DanishGreeter-t „kötöttük be”, akkor viszont ilyesmit várhatunk:

```
Davs janstey
```

Habár ez a példa igen egyszerűnek tűnhet, remélhetőleg megértettük általa, hogy mi is az Spring, illetve hogy mire szolgál egy IoC konténer valójában.

Feltehetjük a kérdést, hogy a Camel hogyan lép be itt a képbe? Lényegében, a Camel-t is konfigurálhatjuk egy bean-ként. Például, az előzőekben konfigurált ActiveMQ brokerhez csatlakozó JMS komponenst megadhatjuk a Spring bean leírásának segítségével is:

```
<bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
  <property name="connectionFactory">
    <bean class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL" value="vm://localhost" />
    </bean>
  </property>
</bean>
```

Így a Camel egy `org.apache.camel.Component` típusú bean-t fog keresni, és automatikusan hozzáadja azt a `CamelContext`-hez – ezt előzőekben kézzel kellett megtennünk.

De Spring-ben hol kell a `CamelContext`-et definiálni? A könnyebb használhatóság érdekében a Camel a Spring kiterjeszthetőségét kihasználva saját XML szintaxist definiál a Spring-en belüli konfigurációhoz. Ennek tükrében egy `CamelContext` Spring-es betöltéséhez a következőket kell tennünk:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://camel.apache.org/schema/spring
       http://camel.apache.org/schema/spring/camel-spring.xsd">

  ...

  <camelContext xmlns="http://camel.apache.org/schema/spring"/>

</beans>
```

Ez automatikusan elindít egy `SpringCamelContext`-et, amely a Java DSL-ben használt `DefaultCamelContext` egy leszármazottja. Vegyük észre továbbá, hogy az XML fájlba bele kellett vennünk az http://camel.apache.org/schema/spring/camel-spring.xsd séma definíciót is – az egyedi XML elemek importálása miatt van erre szükség.

Ez a kódrészlet magában nem sokra való. Meg kell adnunk az útvonalakat is, mint ahogy tettük ezt Java DSL-nél is. Az alábbi kóddal ugyanazt érhetjük el, mint a Java DSL-es megfelelőjével.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://camel.apache.org/schema/spring
       http://camel.apache.org/schema/spring/camel-spring.xsd">
```

```
  <bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
    <property name="connectionFactory">
      <bean class="org.apache.activemq.ActiveMQConnectionFactory">
        <property name="brokerURL" value="vm://localhost" />
      </bean>
    </property>
  </bean>

  <bean id="ftpToJmsRoute" class="camelinaction.FtpToJMSRoute"/>

  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <routeBuilder ref="ftpToJmsRoute"/>
  </camelContext>

</beans>
```

Mint láthatja, a `camelinaction.FtpToJMSRoute` osztályt egy `RouteBuilder`-ként hivatkozunk. A Java DSL-es változat reprodukálásához az ott anonymous, belső osztályként megadott `RouteBuilder`-t ki kell szervezni a saját, nevesített osztályába:

```
public class FtpToJMSRoute extends RouteBuilder {
  public void configure() {
    from("ftp://rider.com/orders?username=rider&password=secret")
      .to("jms:incomingOrders");
  }
}
```

Most hogy megismerkedtünk a Spring alapjaival, illetve hogy hogyan tölthetjük be a Camel-t Springben, megnézzük, hogyan írhatunk Camel útvonalakt tisztán XML-ben – Java DSL nélkül.

A Spring DSL

Láthattuk, hogy a Camel megfelelően integrálódik a Spring-hez, de nem használtuk ki teljesen a Spring által nyújtott programkódok nélküli konfigurációs módszerét. A Spring XML-ben az inversion of control teljes megvalósításához a Camel által nyújtott kiterjesztéseket használhatjuk, amit Spring DSL-nek is nevezhetünk. Spring DSL-ben szinte mindent megtehetünk, mint amit Java DSL is megtehettünk.

Folytassuk a Rider-es az előzőekben tárgyalt Rider Autóalkatrészek-es példát, de most a RouteBuilder helyett adjuk meg tisztán XML-ben az útvonalválasztási szabályokat! Az alábbi Spring XML kód éppen ezt teszi:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://camel.apache.org/schema/spring
```

```
      http://camel.apache.org/schema/spring/camel-spring.xsd">
  <bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
    <property name="connectionFactory">
      <bean class="org.apache.activemq.ActiveMQConnectionFactory">
        <property name="brokerURL" value="vm://localhost" />
      </bean>
    </property>
  </bean>
  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="ftp://rider.com/orders?username=rider&password=secret"/>
      <to uri="jms:incomingOrders"/>
    </route>
  </camelContext>
</beans>
```

A `camelContext` elemen belül a `routeBuilder`-t lecseréltük egy `route`-ra. A `route` elemen belül, a Java DSL `RouteBuilder`-éhez hasonló elnevezésű elemek segítségével hozzuk létre az útvonalat. Ez a kód funkcióját tekintve teljesen egyenértékű az előzőleg bemutatott tisztán Java DSL, és Spring + Java DSL-es változatokkal.

A könyvhöz tartozó forráskódban megváltoztattuk a from metódust, így az már a saját gépen lévő fájlokat fogad. Az új útvonal így néz ki:

```
<route>
<from uri="file:src/data?noop=true"/>
<to uri="jms:incomingOrders"/>
</route>
```

A fájl végpont a rendelési adatokat a relatívan megadott src/data mappából fogja betölteni. A `noop` opcióval azt állítottuk be, hogy a fájl ne változzon meg a feldolgozás során, ez igen hasznos lehet tesztelési célokra. A 7. fejezetben láthatjuk majd, hogyan tudjuk a Camel segítségével letörölni ill. áthelyezni a feldolgozott fájlokat.

Ez az útvonal azonban jelenleg nem tartalmaz semmilyen érdekességet. Próbáljuk ki, mi történik akkor, ha hozzáadunk egy processor-t is!

PROCESSOR HOZZÁADÁSA

Az egyes processing (feldolgozó) lépések hozzáadása éppoly egyszerű, mint Java DSL-ben. Mivel nem hivatkozhatunk anonymous osztályokra Spring XML-ben, ezért létre kell hoznunk a feldolgozónak megfelelő nevesített osztályt:

```
public class DownloadLogger implements Processor {
  public void process(Exchange exchange) throws Exception {
    System.out.println("We just downloaded: "
        + exchange.getIn().getHeader("CamelFileName"));
  }
}
```

Most már használhatjuk őt a Spring útvonalban:

```
<bean id="downloadLogger" class="camelinaction.DownloadLogger"/>
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="file:src/data?noop=true"/>
    <process ref="downloadLogger"/>
    <to uri="jms:incomingOrders"/>
  </route>
</camelContext>
```

Most már lefuttathatjuk a példát. Lépjünk a forráskódok között a chapter2/spring mappába és adjuk ki a következő parancsot:

```
mvn clean compile camel:run
```

Mivel csak egy fájl (message1.xml) található az src/data mappában, a parancs valami ilyesmit ír majd ki:

```
We just downloaded: message1.xml
```

Mit kell tennünk ahhoz, hogy a program ezt az üzenetet a JMS sorról való lekérdezés után írja ki? Létre kell hoznunk még egy útvonalat.

TÖBB ÚTVONAL HASZNÁLATA

Már említettük, hogy Java DSL-ben minden `from`-mal kezdődő Java utasítás egy új útvonalat kezdetét jelöli. Természetesen Spring DSL-ben is készíthetünk több útvonalat, ehhez egyszerűen adjunk hozzá egy új `route` elemet a `camelContext`-en belül.

Például tegyük a második útvonalra `DownloadLogger` bean-t, miután a rendelés elküldésre került az `incomingOrders` sorra:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="file:src/data?noop=true"/>
    <to uri="jms:incomingOrders"/>
  </route>
  <route>
    <from uri="jms:incomingOrders"/>
    <process ref="downloadLogger"/>
  </route>
```

```
</camelContext>
```

Így az incomingOrders sorról jövő üzeneteket kapjuk meg a második útvonalon, tehát a letöltési üzenet a sorra való elküldés után kerül kiírásra.

MELYIK DSL-T HASZNÁLJUK?

Gyakori kérdés a Camel felhasználók körében, hogy melyik DSL-t érdemes használni egy adott helyzetben, de ezt általában csak attól függ, ki mit részesít előnyben. Ha Ön szeret a Spring-gel és XML-ben dolgozni, akkor nyugodtan alkalmazza a tisztán Spring-es megoldást. Ha jobban szereti a Java-t, akkor a tisztán Java DSL-es megközelítés lehet a nyerő.

Akármelyik módszert is válassza, szinte az összes Camel szolgáltatást el tuja érni. A Java DSL némileg gazdagabb nyelv, hiszen ekkor a Java nyelv teljes ereje a rendelkezésünkre áll. Néhány Java DSL szolgáltatás, mint például a value builder (kifejezések és predikátumok készítésére szolgál) nem érhető el Spring DSL-ben. A másik módszert választva viszont használhatjuk a Spring nagyszerű objektum létrehozási képességeit szintúgy, mint az igen gyakran használt Spring absztrakciókat adatbázisokhoz és JMS integrációhoz.

Egy gyakori kompromisszum (és egyben a szerzők kedvelt módszere) a két módszer vegyes felhasználása, erről lesz szó a következő részben.

Camel és Spring használata

Akár Java vagy Spring DSL-ben írjuk le az útvonalakat, a Camel Spring konténerben való futtatása számos egyéb előnnyel jár. Egyik ilyen előny, hogy ha Spring-et használunk, nem kell újrafordítani a kódot az útvonalirányítási szabályok megváltoztatásakor. Továbbá így hozzáférünk a Spring által nyújtott adatbázis kapcsolókhoz, tranzakciókezeléshez, stb.

Most nézzük meg közelebbről, hogy milyen más Spring integrációs módokat használhatunk Camel-ben!

ROUTE BUILDER-EK MEGKERESÉSE

Nagyon jó módszer, hogy a Spring `CamelContext`-jét használjuk, az útvonalak fejlesztése azonban Java DSL-ben történik. Sőt, leggyakrabban így használják a Camel-t.

Láthattuk korábban, hogy konkrétan megadhatjuk a Spring `CamelContext`-jének hogy mely útvonalakat kell betöltenie, ezt a `routeBuilder` elemmel tehetjük meg:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <routeBuilder ref="ftpToJmsRoute"/>
</camelContext>
```

Ez a konkrét leírás a Camel-be betöltendő dolgok tiszta és tömör definícióját képezi.

Azonban néha ettől rugalmasabbnak kell lennünk, itt lépnek képbe a `packageScan` és a `contextScan` elemek:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
<packageScan>
<package>camelinaction.routes</package>
</packageScan>
```

```
</camelContext>
```

A fenti `packageScan` elem a camelinaction.routes csomagban (és az alcsomagokban) talált összes `RouteBuilder` osztályt betölti.

Válogathatunk is a betöltendő útvonalak között:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <packageScan>
    <package>camelinaction.routes</package>
    <excludes>**.*Test*</excludes>
    <includes>**.*</includes>
  </packageScan>
</camelContext>
```

Ebben az esetben minden olyan útvonalat betöltünk a camelinaction.routes csomagból, amelynek nevében nem szerepel a „Test" szó. Az illesztésre használt kifejezések szintaxisa hasonló az Apache Ant-ban fájl illesztési mintákhoz.

A contextScan elem a Spring komponens-kereső szolgáltatását használja fel, ennek segítségével minden olyan `RouteBuilder` osztályt betölt, amelyet a `@org.springframework.stereotype.Component` annotációval megjelöltünk. Módosítsuk ennek tükrében az `FtpToJMSRoute` osztályunkat:

```
@Component
public class FtpToJMSRoute extends SpringRouteBuilder {

  public void configure() {
    from("ftp://rider.com/orders?username=rider&password=secret")
    .to("jms:incomingOrders");
  }
}
```

Vegyük észre, hogy most az `org.apache.camel.spring.SpringRouteBuilder` osztályt használtuk, amely a `RouteBuilder` Spring segédfüggvényekkel kibővített változata. Most engedélyezzük a komponensek automatikus keresését a Spring XML fájlban:

```
<context:component-scan base-package="camelinaction.routes"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">
  <contextScan/>
</camelContext>
```

Ezzel betöltünk minden `camelinaction.routes` csomagban található `RouteBuilder`-t amely tartalmazza a `@Component` annotációt.

Néhány Camel komponens (például a JMS komponens) Spring-es alapokra épül, ezért nem csoda, hogy nagyon egyszerű ezek konfigurációja Spring-ben.

KOMPONSEK ÉS VÉGPONTOK KONFIGURÁLÁSA

Ahogy az előzőekben láthattuk az egyes komponenseket konfigurálhatjuk Spring XML-ben, majd ezeket automatikusan felismeri a Camel. Példaként nézzük meg ismét a JMS komponenst:

```
<bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
  <property name="connectionFactory">
    <bean class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL" value="vm://localhost" />
    </bean>
  </property>
</bean>
```

Az `id` tulajdonsággal állíthatjuk be a bean elnevezését, ezzel rugalmasabban, az adott esethez jobban illően nevezhetjük el a komponenseket. Előfordulhat, hogy az alkalmazásunk két JMS broker-t is szeretnénk használni: például az Apache ActiveMQ-t és a SonicMQ-t:

```
<bean id="activemq" class="org.apache.camel.component.jms.JmsComponent">
  ...
</bean>
<bean id="sonicmq" class="org.apache.camel.component.jms.JmsComponent">
  ...
</bean>
```

Ezután használhatjuk az `activemq:myActiveMQQueue` és a `sonicmq:mySonicQueue` URI-kat is.

Végpontok definiálására is használható a Camel Spring kiterjesztése. Például hozzunk létre egy FTP végpontot a Rider hagyományos rendelési szerveréhez:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <endpoint id="ridersFtp"
            uri="ftp://rider.com/orders?username=rider&password=secret"/>
    <route>
      <from ref="ridersFtp"/>
      <to uri="jms:incomingOrders"/>
    </route>
</camelContext>
```

> **MEGJEGYZÉS** Mint láthatja az azonosító adatokat (felhasználónév és jelszó) is közvetlenül az URI-ba írtuk, ez sokszor nem megfelelő. Ezeket célszerű valamilyen megfelelően védett helyen tárolni, és erre hivatkozni. A 6. fejezet 6.1.6. részében olvashat arról, hogyan valósíthatjuk meg ezt a Camel Properties komponense vagy a Spring property helyőrzőinek segítségével.

A KONFIGURÁCIÓ ÉS AZ ÚTVONALAK IMPORTÁLÁSA

Amikor Spring-ben fejlesztünk, gyakran szétbontjuk a konfigurációt több XML fájlra. Ennek legfőbb célja, hogy olvashatóbb XML kódot kapjunk, mivel valószínűleg senki sem szeretne egy több száz soros, egybefüggő XML fájllal dolgozni.

A több fájl használatának másik előnye, hogy így nagy eséllyel kapunk kisebb, újrafelhasználható egységeket. Például egy másik alkalmazásban is lehet, hogy hasonlóan kell konfigurálnunk a JMS komponenst, ezért készítünk erre a célra egy jms-setup.xml fájlt:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

  <bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
    <property name="connectionFactory">
      <bean class="org.apache.activemq.ActiveMQConnectionFactory">
        <property name="brokerURL" value="vm://localhost" />
      </bean>
    </property>
  </bean>
</beans>
```

Ezt beimportálhatjuk a `CamelContext`-et tartalmazó másik XML fájlunkba az alábbi sorral:

```
<import resource="jms-setup.xml"/>
```

Így elérhető tettük a `CamelContext` számára a másik fájlban konfigurált JMS komponenst.

Hasznos lehet továbbá, ha az egyes Spring DSL útvonalakat magukat is külön fájlokban definiáljuk. Mivel a route elemeket csak egy `camelContext`-en belül hozhatunk létre, ezért most egy másik módszert mutatunk be az útvonalak létrehozására. Egy `routeContext` elemen belül az alábbi módon hozhatunk létre útvonalakat:

```
<routeContext id="ftpToJms" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="ftp://rider.com/orders?username=rider&password=secret"/>
    <to uri="jms:incomingOrders"/>
  </route>
</routeContext>
```

A `routeContext` elemet definiálhatjuk más fájlban, de akár ugyanebben a fájlban is, ezután a `routeContextRef` elemmel hivatkozhatunk az egyes `routeContext`-ekre. A `routeContextRef` elemet így kell a `camelContext`-ben használni:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <routeContextRef ref="ftpToJms"/>
</camelContext>
```

Ha több `CamelContext`-be is behúzunk egy `routeContext`-et, mindegyikhez egy új útvonal jön létre. Az előző esetben két teljesen azonos útvonal, azonos végpont URI-kkal ugyanahhoz az erőforráshoz való versengéshez vezet. Ez most konkrétan azt jelenti, hogy egyszerre csak útvonal fog megkapni egy adott fájlt az FTP szerverről. Legyünk mindig nagyon körültekintőek, amikor egy útvonalat több `CamelContext`-ben is felhasználunk.

TOVÁBBI KONFIGURÁCIÓS LEHETŐSÉGEK

Még sok további konfigurációs lehetőséget biztosít számunkra a Spring és a CamelContext, ezekről részletesen a következő fejezetekben lesz szó.