Spring

Spring Support

Apache Camel is designed to work nicely with the Spring Framework in a number of ways.

- Camel uses Spring Transactions as the default transaction handling in components like JMS and JPA
- Camel works with Spring 2 XML processing with the Xml Configuration
- Camel Spring XML Schema's is defined at Xml Reference
- Camel supports a powerful version of Spring Remoting which can use powerful routing between the client and server side along with using all of the available Components for the transport
- Camel provides powerful Bean Integration with any bean defined in a Spring **`ApplicationContext`**.
- Camel integrates with various Spring helper classes; such as providing Type Converter support for Spring Resources etc
- Allows Spring to dependency inject Component instances or the CamelContext instance itself and auto-expose Spring beans as components and endpoints.
- Allows you to reuse the Spring Testing framework to simplify your unit and integration testing using Enterprise Integration Patterns and Camel's powerful Mock and Test endpoints
- From **Camel 2.15**: Camel supports Spring Boot using the `camel-spring-boot` component.
- From **Camel 2.17.1**: Camel supports Spring Cache based Idempotent repository

Using Spring to configure the CamelContext

You can configure a **`CamelContext`** inside any **`spring.xml`** using the CamelContextFactoryBean. This will automatically start the CamelContext along with any referenced Routes along any referenced Component and Endpoint instances.

- Adding Camel schema
- Configure Routes in two ways:
  - Using Java Code
  - Using Spring XML

Adding Camel Schema

For Camel 1.x you need to use the following namespace:

```
http://activemq.apache.org/camel/schema/spring
```

with the following schema location:

```
http://activemq.apache.org/camel/schema/spring/camel-spring.xsd
```

You need to add Camel to the **`schemaLocation`** declaration

```
http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd
```

So the XML file looks like this:

```
<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.xsd
       http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd
    ">
```

Using `camel:` Namespace

Or you can refer to the camel XSD in the XML declaration:

```
xmlns:camel="http://camel.apache.org/schema/spring"
```

... so the declaration is:

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

... and then use the **`camel:`** namespace prefix, and you can omit the inline namespace declaration:

```
<camel:camelContext id="camel5">
  <camel:package>org.apache.camel.spring.example</camel:package>
</camel:camelContext>
```

Advanced Configuration Using Spring

See more details at Advanced configuration of CamelContext using Spring

Using Java Code

You can use Java Code to define your RouteBuilder implementations. These can be defined as beans in spring and then referenced in your camel context e.g.

```
<camelContext id="camel5" xmlns="http://camel.apache.org/schema/spring">
  <routeBuilder ref="myBuilder" />
</camelContext>

<bean id="myBuilder" class="org.apache.camel.spring.example.test1.MyRouteBuilder"/>
```

Using <package>

Camel also provides a powerful feature that allows for the automatic discovery and initialization of routes in given packages. This is configured by adding tags to the camel context in your spring context definition, specifying the packages to be recursively searched for RouteBuilder implementations. To use this feature in **`1.X`**, requires a **`<package></package>`** tag specifying a comma separated list of packages that should be searched e.g.

camelContextRouteBuilderRef.xml

```
<camel:camelContext id="camel5">
  <camel:package>org.apache.camel.spring.example</camel:package>
</camel:camelContext>
```

Use caution when specifying the package name as **`org.apache.camel`** or a sub package of this. This causes Camel to search in its own packages for your routes which could cause problems.

Will ignore already instantiated classes

The **`<package>`** and **`<packageScan>`** will skip any classes which has already been created by Spring etc. So if you define a route builder as a spring bean tag then that class will be skipped. You can include those beans using **`<routeBuilder ref="theBeanId"/>`** or the **`<contextScan>`** feature.

Using <packageScan>

In Camel 2.0 this has been extended to allow selective inclusion and exclusion of discovered route classes using Ant like path matching. In spring this is specified by adding a **`<packageScan/>`** tag. The tag must contain one or more **`package`** elements (similar to **`1.x`**), and optionally one or more **`includes`** or **`excludes`** elements specifying patterns to be applied to the fully qualified names of the discovered classes. e.g.,

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

Exclude patterns are applied before the include patterns. If no include or exclude patterns are defined then all the Route classes discovered in the packages will be returned.

In the above example, camel will scan all the **`org.example.routes`** package and any subpackages for **`RouteBuilder`** classes. Say the scan finds two **`RouteBuilders`**, one in **`org.example.routes`** called **`MyRoute`** and another **`MyExcludedRoute`** in a subpackage **`excluded`**. The fully qualified names of each of the classes are extracted (**`org.example.routes.MyRoute`**, **`org.example.routes.excluded.MyExcludedRoute`**) and the include and exclude patterns are applied.

The exclude pattern **`**.*Excluded*`** is going to match the FQCN **`org.example.routes.excluded.MyExcludedRoute`** and veto camel from initializing it.

Under the covers, this is using Spring's AntPatternMatcher implementation, which matches as follows

? matches one character * matches zero or more characters ** matches zero or more segments of a fully qualified name

For example:

`**.*Excluded*` would match **`org.simple.Excluded`**, **`org.apache.camel.SomeExcludedRoute`** or **`org.example.RouteWhichIsExcluded`**.

`**.??cluded*` would match **`org.simple.IncludedRoute`**, **`org.simple.Excluded`** but *not* match **`org.simple.PrecludedRoute`**.

Using `contextScan`

Available as of Camel 2.4

You can allow Camel to scan the container context, e.g. the Spring **`ApplicationContext`** for route builder instances. This allow you to use the Spring **`<component-scan>`** feature and have Camel pickup any **`RouteBuilder`** instances which was created by Spring in its scan process.

```
<!-- enable Spring @Component scan -->
<context:component-scan base-package="org.apache.camel.spring.issues.contextscan"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <!-- and then let Camel use those @Component scanned route builders -->
    <contextScan/>
</camelContext>
```

This allows you to just annotate your routes using the Spring **`@Component`** and have those routes included by Camel:

```
@Component
public class MyRoute extends SpringRouteBuilder {
 @Override public void configure() throws Exception {
    from("direct:start") .to("mock:result");
 }
}
```

You can also use the ANT style for inclusion and exclusion, as mentioned above in the **`<packageScan>`** documentation.

how do i import routes from other xml files

Test Time Exclusion.

At test time it is often desirable to be able to selectively exclude matching routes from being initialized that are not applicable or useful to the test scenario. For instance you might a spring context file **`routes-context.xml`** and three Route builders **`RouteA`,** `RouteB` and **`RouteC`** in the **`org.example.routes`** package. The **`packageScan`** definition would discover all three of these routes and initialize them.

Say **`RouteC`** is not applicable to our test scenario and generates a lot of noise during test. It would be nice to be able to exclude this route from this specific test. The **`SpringTestSupport`** class has been modified to allow this. It provides two methods (**`excludedRoute`** and **`excludedRoutes`**) that may be overridden to exclude a single class or an array of classes.

```
public class RouteAandRouteBOnlyTest extends SpringTestSupport {
  @Override
  protected Class excludeRoute() {
    return RouteC.class;
  }
}
```

java

In order to hook into the **`camelContext`** initialization by spring to exclude the **`MyExcludedRouteBuilder.class`** we need to intercept the spring context creation. When overriding **`createApplicationContext`** to create the spring context, we call the **`getRouteExcludingApplicationContext()`** method to provide a special parent spring context that takes care of the exclusion.

```
@Override
protected AbstractXmlApplicationContext createApplicationContext() {
  return new ClassPathXmlApplicationContext(
        new String[] {"routes-context.xml"}, getRouteExcludingApplicationContext());
}
```

`RouteC` will now be excluded from initialization. Similarly, in another test that is testing only **`RouteC`**, we could exclude **`RouteB`** and **`RouteA`** by overriding:

```
@Override
protected Class[] excludeRoutes() {
 return new Class[]{RouteA.class, RouteB.class};
}
```

Using Spring XML

You can use Spring 2.0 XML configuration to specify your Xml Configuration for Routes such as in the following example.

```
<camelContext id="camel-A" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="seda:start"/>
    <to uri="mock:result"/>
  </route>
</camelContext>
```

Configuring Components and Endpoints

You can configure your Component or Endpoint instances in your Spring XML as follows in this example.

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <jmxAgent id="agent" disabled="true"/>
</camelContext>

<bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">
  <property name="connectionFactory">
    <bean class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL" value="vm://localhost?broker.persistent=false&amp;broker.useJmx=false"/>
    </bean>
  </property>
</bean>
```

Which allows you to configure a component using some name (**`activemq`** in the above example), then you can refer to the component using **`activemq:[queue:|topic:]destinationName`**. This works by the **`SpringCamelContext`** lazily fetching components from the spring context for the scheme name you use for Endpoint URIs.

For more detail see Configuring Endpoints and Components.

Spring Cache Idempotent Repository

Available as of **Camel 2.17.1**

```
<bean id="repo" class="org.apache.camel.spring.processor.idempotent.SpringCacheIdempotentRepository">
 <constructor-arg>
   <bean class="org.springframework.cache.guava.GuavaCacheManager"/>
</constructor-arg>
 <constructor-arg value="idempotent"/>
</bean>
<camelContext xmlns="http://camel.apache.org/schema/spring">
 <route id="idempotent-cache">
  <from uri="direct:start" />
    <idempotentConsumer messageIdRepositoryRef="repo" skipDuplicate="true">
      <header>MessageId</header>
      <to uri="log:org.apache.camel.spring.processor.idempotent?level=INFO&amp;showAll=true&amp;multiline=true"
/> <to uri="mock:result"/>
    </idempotentConsumer>
 </route>
</camelContext>
```

CamelContextAware

If you want to be injected with the CamelContext in your POJO just implement the CamelContextAware interface; then when Spring creates your POJO the **`CamelContext`** will be injected into your POJO. Also see the Bean Integration for further injections.

Integration Testing

To avoid a hung route when testing using Spring Transactions see the note about Spring Integration Testing under Transactional Client.

See also

- Spring JMS Tutorial
- Creating a new Spring based Camel Route
- Spring example
- Xml Reference
- Advanced configuration of CamelContext using Spring
- How Do I Import Routes From Other XML Files?