Spring Boot

Spring Boot

Available as of Camel 2.15

Spring Boot component provides auto-configuration for Apache Camel. Our opinionated auto-configuration of the Camel context auto-detects Camel routes available in the Spring context and registers the key Camel utilities (like producer template, consumer template and the type converter) as beans.

Maven users will need to add the following dependency to their **`pom.xml`** in order to use this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-spring-boot</artifactId>
    <version>${camel.version}</version> <!-- use the same version as your Camel core version -->
</dependency>
```

`camel-spring-boot` jar comes with the **`spring.factories`** file, so as soon as you add that dependency into your classpath, Spring Boot will automatically auto-configure Camel for you.

Camel Spring Boot Starter

Available as of Camel 2.17

Apache Camel ships a Spring Boot Starter module that allows you to develop Spring Boot applications using starters. There is a sample application in the source code also.

To use the starter, add the following to your spring boot **`pom.xml`** file:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-spring-boot-starter</artifactId>
    <version>2.17.0</version>
</dependency>
```

Then you can just add classes with your Camel routes such as:

```
package com.example;

import org.apache.camel.builder.RouteBuilder;
import org.springframework.stereotype.Component;

@Component
public class MyRoute extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        from("timer:foo")
          .to("log:bar");
    }
}
```

Then these routes will be started automatically. To keep the main thread blocked so that Camel stays up, either include the **`spring-boot-starter-web`** dependency, or add **`camel.springboot.main-run-controller=true`** to your **`application.properties`** or **`application.yml`** file.

You can further customize the Camel application in the **`application.properties`** or **`application.yml`** file with ***`camel.springboot.* properties`***.

Auto-Configured Camel Context

The most important piece of functionality provided by the Camel auto-configuration is **`CamelContext`** instance. Camel auto-configuration creates a **`SpringCamelContext`** for you and takes care of the proper initialization and shutdown of that context. The created Camel context is also registered in the Spring application context (under **`camelContext`** bean name), so you can access it just as any other Spring bean.

```
@Configuration
public class MyAppConfig {

  @Autowired
  CamelContext camelContext;

  @Bean
  MyService myService() {
    return new DefaultMyService(camelContext);
  }
}
```

Auto-Detecting Camel Routes

Camel auto-configuration collects all the **`RouteBuilder`** instances from the Spring context and automatically injects them into the provided **`CamelContext`**. That means that creating new Camel route with the Spring Boot starter is as simple as adding the **`@Component`** annotated class to your classpath:

```
@Component
public class MyRouter extends RouteBuilder {

  @Override
  public void configure() throws Exception {
    from("jms:invoices").to("file:/invoices");
  }
}
```

...or creating a new route **`RouteBuilder`** bean in your **`@Configuration`** class:

```
@Configuration
public class MyRouterConfiguration {

  @Bean
  RoutesBuilder myRouter() {
    return new RouteBuilder() {

      @Override
      public void configure() throws Exception {
        from("jms:invoices")
          .to("file:/invoices");
      }
    };
  }
}
```

Camel properties

Spring Boot auto-configuration automatically connects to Spring Boot external configuration (like properties placeholders, OS environment variables or system properties) with the Camel properties support. It basically means that any property defined in **`application.properties`** file:

```
route.from = jms:invoices
```

...or set via system property...

```
java -Droute.to=jms:processed.invoices -jar mySpringApp.jar
```

...can be used as placeholders in Camel route:

```
@Component
public class MyRouter extends RouteBuilder {

  @Override
  public void configure() throws Exception {
    from("{{route.from}}")
      .to("{{route.to}}");
  }
}
```

Custom Camel Context Configuration

If you would like to perform some operations on **`CamelContext`** bean created by Camel auto-configuration, register **`CamelContextConfiguration`** inst ance in your Spring context:

```
@Configuration
public class MyAppConfig {

  ...

  @Bean
  CamelContextConfiguration contextConfiguration() {
    return new CamelContextConfiguration() {
      @Override
      void beforeApplicationStart(CamelContext context) {
        // your custom configuration goes here
      }
    };
  }
}
```

Method **`CamelContextConfiguration#beforeApplicationStart(CamelContext)`** will be called just before the Spring context is started, so the **`C amelContext`** instance passed to this callback is fully auto-configured. You can add many instances of **`CamelContextConfiguration`** into your Spring context - all of them will be executed.

Disabling JMX

To disable JMX of the auto-configured **`CamelContext`** use **`camel.springboot.jmxEnabled`** property (JMX is enabled by default). For example you could add the following property to your **`application.properties`** file:

```
camel.springboot.jmxEnabled = false
```

Auto-Configured Consumer and Producer Templates

Camel auto-configuration provides pre-configured **`ConsumerTemplate`** and **`ProducerTemplate`** instances. You can simply inject them into your Spring-managed beans:

```
@Component
public class InvoiceProcessor {

  @Autowired
  private ProducerTemplate producerTemplate;

  @Autowired
  private ConsumerTemplate consumerTemplate;
  public void processNextInvoice() {
    Invoice invoice = consumerTemplate.receiveBody("jms:invoices", Invoice.class);
    ...
    producerTemplate.sendBody("netty-http:http://invoicing.com/received/" + invoice.id());
  }
}
```

By default consumer templates and producer templates come with the endpoint cache sizes set to **`1000`**. You can change those values via the following Spring properties:

```
camel.springboot.consumerTemplateCacheSize = 100
camel.springboot.producerTemplateCacheSize = 200
```

Auto-Configured TypeConverter

Camel auto-configuration registers a **`TypeConverter`** instance named **`typeConverter`** in the Spring context.

```
@Component
public class InvoiceProcessor {

  @Autowired
  private TypeConverter typeConverter;

  public long parseInvoiceValue(Invoice invoice) {
    String invoiceValue = invoice.grossValue();
    return typeConverter.convertTo(Long.class, invoiceValue);
  }
}
```

Spring Type Conversion API Bridge

Spring comes with the powerful type conversion API. Spring API happens to be very similar to the Camel type converter API. As those APIs are so similar, Camel Spring Boot automatically registers a bridge converter (**`SpringTypeConverter`**) that delegates to the Spring conversion API.That means that out-of-the-box Camel will treat Spring Converters like Camel ones. With this approach you can enjoy both Camel and Spring converters accessed via Camel **`TypeConverter`** API:

```
@Component
public class InvoiceProcessor {

  @Autowired
  private TypeConverter typeConverter;

  public UUID parseInvoiceId(Invoice invoice) {
    // Using Spring's StringToUUIDConverter
    UUID id = invoice.typeConverter.convertTo(UUID.class, invoice.getId());
  }
}
```

Under the hood Camel Spring Boot delegates conversion to the Spring's `ConversionService` instances available in the application context. If no **`ConversionService`** instance is available, Camel Spring Boot auto-configuration will create one for you.

Disabling Type Conversions Features

If you don't want Camel Spring Boot to register type-conversions related features (like **`TypeConverter`** instance or Spring bridge) set the **`camel.springboot.typeConversion`** property to **`false`**.

```
camel.springboot.typeConversion = false
```

Fat Jars and Fat Wars

The easiest way to create a Camel-aware Spring Boot fat jar/war is to extend the **`org.apache.camel.spring.boot.FatJarRouter`** class:

```
package com.example;

... // imports

@SpringBootApplication
public class MyFatJarRouter extends FatJarRouter {

    @Override
    public void configure() throws Exception {
        from("netty-http:http://0.0.0.0:18080").
            setBody().simple("ref:helloWorld");
    }

    @Bean
    String helloWorld() {
        return "helloWorld";
    }
}
```

...and add the following property to your **`application.properties`** file:

```
spring.main.sources = com.example.MyFatJarRouter
```

It is also recommended to define your main class explicitly in the Spring Boot Maven plugin configuration:

```
 <plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <version>${spring-boot.version}</version>
    <configuration>
      <mainClass>org.apache.camel.spring.boot.FatJarRouter</mainClass>
    </configuration>
    <executions>
      <execution>
        <goals>
          <goal>repackage</goal>
        </goals>
      </execution>
    </executions>
</plugin>
```

In order to turn your fat jar into fat war, add the following class extending **`org.apache.camel.spring.boot.FatWarInitializer`** to your project:

```
package com.example;

... // imports

public class MyFatJarRouterWarInitializer extends FatWarInitializer {

  @Override
  protected Class<? extends FatJarRouter> routerClass() {
    return MyFatJarRouter.class;
  }
}
```

Blocking Main Thread

This feature is available starting from Camel **2.15.2**. Camel applications extending FatJarRouter by default block the main thread of the application. It means that after you start your fat jar, your application waits for **`Ctrl+C`** signal and does not exit immediately. If you would like to achieve similar behavior for non-**`FatJarRouter`** applications, retrieve **`CamelSpringBootApplicationController`** bean from your **`ApplicationContext`** and use the former to block the main thread of your application using **`CamelSpringBootApplicationController#blockMainThread()`** method.

```
public static void main(String... args) {
    ApplicationContext applicationContext = new SpringApplication(MyCamelApplication.class).run(args);
    CamelSpringBootApplicationController applicationController =
            applicationContext.getBean(CamelSpringBootApplicationController.class);
    applicationController.blockMainThread();
}
```

Adding XML Routes

By default you can put Camel XML routes in the classpath under the directory camel, which camel-spring-boot will auto detect and include. From **Camel 2.17**: you can configure the directory name or turn this off using the configuration option:

```
// turn off
camel.springboot.xmlRoutes = false
// scan in the com/foo/routes classpath
camel.springboot.xmlRoutes = classpath:com/foo/routes/*.xml
```

The XML files should be Camel XML routes (not CamelContext) such as

```
   <routes xmlns="http://camel.apache.org/schema/spring">
        <route id="test">
            <from uri="timer://trigger"/>
            <transform>
                <simple>ref:myBean</simple>
            </transform>
            <to uri="log:out"/>
        </route>
    </routes>
```

Adding Rest-DSL

Available since Camel 2.18

By default you can put Camel Rest-DSL XML routes in the classpath under the directory **`camel-rest`**, which **`camel-spring-boot`** will auto detect and include.

You can configure the directory name or turn this off using the configuration option:

```
// turn off
camel.springboot.xmlRests = false
// scan in the com/foo/routes classpath
camel.springboot.xmlRests = classpath:com/foo/rests/*.xml
```

The Rest-DSL XML files should be Camel XML rests (not CamelContext) such as

```
<rests xmlns="http://camel.apache.org/schema/spring">
   <rest>
      <post uri="/persons">
         <to uri="direct:postPersons"/>
      </post>
      <get uri="/persons">
         <to uri="direct:getPersons"/>
      </get>
      <get uri="/persons/{personId}">
          <to uri="direct:getPersionId"/>
      </get>
      <put uri="/persons/{personId}">
          <to uri="direct:putPersionId"/>
      </put>
      <delete uri="/persons/{personId}">
          <to uri="direct:deletePersionId"/>
      </delete>
   </rest>
 </rests>
```

Unit Tests

Below is a sample unit test set up for camel spring-boot.

```
@ActiveProfiles("test")
@RunWith(CamelSpringBootRunner.class)
@SpringBootTest
@DirtiesContext(classMode = ClassMode.AFTER_EACH_TEST_METHOD)
@DisableJmx(true)
public class MyRouteTest extends CamelTestSupport {

        @Autowired
        private CamelContext camelContext;

        @Override
        protected CamelContext createCamelContext() throws Exception {
                return camelContext;
        }

        @EndpointInject(uri = "direct:myEndpoint")
        private ProducerTemplate endpoint;

        @Override
        public void setUp() throws Exception {
                super.setUp();
                RouteDefinition definition = context().getRouteDefinitions().get(0);
                definition.adviceWith(context(), new RouteBuilder() {
                        @Override
                        public void configure() throws Exception {
                                onException(Exception.class).maximumRedeliveries(0);
                        }
                });
        }

        @Override
        public String isMockEndpointsAndSkip() {
                return "myEndpoint:put*";
        }

        @Test
        public void shouldSucceed() throws Exception {
                assertNotNull(camelContext);
                assertNotNull(endpoint);

                String expectedValue = "expectedValue";
                MockEndpoint mock = getMockEndpoint("mock:myEndpoint:put");
                mock.expectedMessageCount(1);
                mock.allMessages().body().isEqualTo(expectedValue);
                mock.allMessages().header(MY_HEADER).isEqualTo("testHeader");
                endpoint.sendBodyAndHeader("test", MY_HEADER, "testHeader");

                mock.assertIsSatisfied();
        }
}
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started