Service Activator

The service activator is the endpoint type for connecting any Spring-managed object to an input channel so that it may play the role of a service. If the service produces output, it may also be connected to an output channel. Alternatively, an output-producing service may be located at the end of a processing pipeline or message flow, in which case the inbound message's `replyChannel` header can be used. This is the default behavior if no output channel is defined. As with most of the configuration options described here, the same behavior actually applies for most of the other components.

Configuring Service Activator

To create a service activator, use the 'service-activator' element with the 'input-channel' and 'ref' attributes, as the following example shows:

```xml
<int:service-activator input-channel="exampleChannel" ref="exampleHandler"/>
```

The preceding configuration selects all the methods from the `exampleHandler` that meet one of the messaging requirements, which are as follows:

- annotated with `@ServiceActivator`
- is `public`
- not return `void` if `requiresReply == true`

The target method for invocation at runtime is selected for each request message by their `payload` type or as a fallback to the `Message<?>` type if such a method is present on target class.

Starting with version 5.0, one service method can be marked with the `@org.springframework.integration.annotation.Default` as a fallback for all non-matching cases. This can be useful when using content-type conversion with the target method being invoked after conversion.

To delegate to an explicitly defined method of any object, you can add the `method` attribute, as the following example shows:

```xml
<int:service-activator input-channel="exampleChannel" ref="somePojo" method="someMethod"/>
```

In either case, when the service method returns a non-null value, the endpoint tries to send the reply message to an appropriate reply channel. To determine the reply channel, it first checks whether an `output-channel` was provided in the endpoint configuration, as the following example shows:

```xml
<int:service-activator input-channel="exampleChannel" output-channel="replyChannel"
                       ref="somePojo" method="someMethod"/>
```

If the method returns a result and no `output-channel` is defined, the framework then checks the request message's `replyChannel` header value. If that value is available, it then checks its type. If it is a `MessageChannel`, the reply message is sent to that channel. If it is a `String`, the endpoint tries to resolve the channel name to a channel instance. If the channel cannot be resolved, a `DestinationResolutionException` is thrown. It it can be resolved, the message is sent there. If the request message does not have a `replyChannel` header and the `reply` object is a `Message`, its `replyChannel` header is consulted for a target destination. This is the technique used for request-reply messaging in Spring Integration, and it is also an example of the return address pattern.

If your method returns a result and you want to discard it and end the flow, you should configure the `output-channel` to send to a `NullChannel`. For convenience, the framework registers one with the name, `nullChannel`. See Special Channels for more information.

The service activator is one of those components that is not required to produce a reply message. If your method returns `null` or has a `void` return type, the service activator exits after the method invocation, without any signals. This behavior can be controlled by the `AbstractReplyProducingMessageHandler.requiresReply` option, which is also exposed as `requires-reply` when configuring with the XML namespace. If the flag is set to `true` and the method returns null, a `ReplyRequiredException` is thrown.

The argument in the service method could be either a message or an arbitrary type. If the latter, then it is assumed to be a message payload, which is extracted from the message and injected into the service method. We generally recommend this approach, as it follows and promotes a POJO model when working with Spring Integration. Arguments may also have `@Header` or `@Headers` annotations, as described in Annotation Support.

> The service method is not required to have any arguments, which means you can implement event-style service activators (where all you care about is an invocation of the service method) and not worry about the contents of the message. Think of it as a null JMS message. An example use case for such an implementation is a simple counter or monitor of messages deposited on the input channel.

Starting with version 4.1, the framework correctly converts message properties (`payload` and `headers`) to the Java 8 `Optional` POJO method parameters, as the following example shows:

```java
public class MyBean {
    public String computeValue(Optional<String> payload,
            @Header(value="foo", required=false) String foo1,
            @Header(value="foo") Optional<String> foo2) {
        if (payload.isPresent()) {
            String value = payload.get();
            ...
        }
        else {
            ...
        }
    }

}
```

We generally recommend using a `ref` attribute if the custom service activator handler implementation can be reused in other `<service-activator>` definitions. However, if the custom service activator handler implementation is only used within a single definition of the `<service-activator>`, you can provide an inner bean definition, as the following example shows:

```xml
<int:service-activator id="exampleServiceActivator" input-channel="inChannel"
            output-channel = "outChannel" method="someMethod">
    <beans:bean class="org.something.ExampleServiceActivator"/>
</int:service-activator>
```

Using both the `ref` attribute and an inner handler definition in the same `<service-activator>` configuration is not allowed, as it creates an ambiguous condition and results in an exception being thrown.

If the `ref` attribute references a bean that extends `AbstractMessageProducingHandler` (such as handlers provided by the framework itself), the configuration is optimized by injecting the output channel into the handler directly. In this case, each `ref` must be to a separate bean instance (or a `prototype`-scoped bean) or use the inner `<bean/>` configuration type. If you inadvertently reference the same message handler from multiple beans, you get a configuration exception.

Service Activators and the Spring Expression Language (SpEL)

Since Spring Integration 2.0, service activators can also benefit from SpEL.

For example, you can invoke any bean method without pointing to the bean in a `ref` attribute or including it as an inner bean definition, as follows:

```xml
<int:service-activator input-channel="in" output-channel="out"
 expression="@accountService.processAccount(payload, headers.accountId)"/>

 <bean id="accountService" class="thing1.thing2.Account"/>
```

In the preceding configuration, instead of injecting 'accountService' by using a `ref` or as an inner bean, we use SpEL's `@beanId` notation and invoke a method that takes a type compatible with the message payload. We also pass a header value. Any valid SpEL expression can be evaluated against any content in the message. For simple scenarios, your service activators need not reference a bean if all logic can be encapsulated in such an expression, as the following example shows:

```xml
<int:service-activator input-channel="in" output-channel="out" expression="payload * 2"/>
```

In the preceding configuration, our service logic is to multiply the payload value by two. SpEL lets us handle it relatively easily.

See Service Activators and the `.handle()` method in the Java DSL chapter for more information about configuring service activator.

Asynchronous Service Activator

The service activator is invoked by the calling thread. This is an upstream thread if the input channel is a `SubscribableChannel` or a poller thread for a `PollableChannel`. If the service returns a `ListenableFuture<?>`, the default action is to send that as the payload of the message sent to the output (or reply) channel. Starting with version 4.3, you can now set

the `async` attribute to `true` (by using `setAsync(true)` when using Java configuration). If the service returns a `ListenableFuture<?>` when this the `async` attribute is set to `true`, the calling thread is released immediately and the reply message is sent on the thread (from within your service) that completes the future. This is particularly advantageous for long-running services that use a `PollableChannel`, because the poller thread is released to perform other services within the framework.

If the service completes the future with an `Exception`, normal error processing occurs. An `ErrorMessage` is sent to the `errorChannel` message header, if present. Otherwise, an `ErrorMessage` is sent to the default `errorChannel` (if available).

Service Activator and Method Return Type

The service method can return any type which becomes reply message payload. In this case a new `Message<?>` object is created and all the headers from a request message are copied. This works the same way for most Spring Integration `MessageHandler` implementations, when interaction is based on a POJO method invocation.

A complete `Message<?>` object can also be returned from the method. However keep in mind that, unlike transformers, for a Service Activator this message will be modified by copying the headers from the request message if they are not already present in the returned message. So, if your method parameter is a `Message<?>` and you copy some, but not all, existing headers in your service method, they will reappear in the reply message. It is not a Service Activator responsibility to remove headers from a reply message and, pursuing the loosely-coupled principle, it is better to add a `HeaderFilter` in the integration flow. Alternatively, a Transformer can be used instead of a Service Activator but, in that case, when returning a full `Message<?>` the method is completely responsible for the message, including copying request message headers (if needed). You must ensure that important framework headers (e.g. `replyChannel`, `errorChannel`), if present, have to be preserved.
