Error Handling

As described in the overview at the very beginning of this manual, one of the main motivations behind a message-oriented framework such as Spring Integration is to promote loose coupling between components. The message channel plays an important role, in that producers and consumers do not have to know about each other. However, the advantages also have some drawbacks. Some things become more complicated in a loosely coupled environment, and one example is error handling.

When sending a message to a channel, the component that ultimately handles that message may or may not be operating within the same thread as the sender. If using a simple default `DirectChannel` (when the `<channel>` element that has no `<queue>` child element and no 'task-executor' attribute), the message handling occurs in the same thread that sends the initial message. In that case, if an `Exception` is thrown, it can be caught by the sender (or it may propagate past the sender if it is an uncaught `RuntimeException`). This is the same behavior as an exception-throwing operation in a normal Java call stack.

A message flow that runs on a caller thread might be invoked through a messaging gateway (see Messaging Gateways) or a `MessagingTemplate` (see `MessagingTemplate`). In either case, the default behavior is to throw any exceptions to the caller. For the messaging gateway, see Error Handling for details about how the exception is thrown and how to configure the gateway to route the errors to an error channel instead. When using a `MessagingTemplate` or sending to a `MessageChannel` directly, exceptions are always thrown to the caller.

When adding asynchronous processing, things become rather more complicated. For instance, if the 'channel' element does provide a 'queue' child element (`QueueChannel` in Java & Annotations Configuration), the component that handles the message operates in a different thread than the sender. The same is true when an `ExecutorChannel` is used. The sender may have dropped the `Message` into the channel and moved on to other things. There is no way for the `Exception` to be thrown directly back to that sender by using standard `Exception` throwing techniques. Instead, handling errors for asynchronous processes requires that the error-handling mechanism also be asynchronous.

Spring Integration supports error handling for its components by publishing errors to a message channel. Specifically, the `Exception` becomes the payload of a Spring Integration `ErrorMessage`. That `Message` is then sent to a message channel that is resolved in a way that is similar to the 'replyChannel' resolution. First, if the request `Message` being handled at the time the `Exception` occurred contains an 'errorChannel' header (the header name is defined in the `MessageHeaders.ERROR_CHANNEL` constant), the `ErrorMessage` is sent to that channel. Otherwise, the error handler sends to a “global” channel whose

bean name is `errorChannel` (this is also defined as a constant: `IntegrationContextUtils.ERROR_CHANNEL_BEAN_NAME`).

A default `errorChannel` bean is created internally by the Framework. However, you can define your own if you want to control the settings. The following example shows how to define an error channel in XML configuration backed by a queue with a capacity of `500`:

```
<int:channel id="errorChannel">
    <int:queue capacity="500"/>
</int:channel>
```

The default error channel is a `PublishSubscribeChannel`.

The most important thing to understand here is that the messaging-based error handling applies only to exceptions that are thrown by a Spring Integration task that is executing within a `TaskExecutor`. This does not apply to exceptions thrown by a handler that operates within the same thread as the sender (for example, through a `DirectChannel` as described earlier in this section).

When exceptions occur in a scheduled poller task's execution, those exceptions are wrapped in `ErrorMessage` instances and sent to the 'errorChannel' as well.

To enable global error handling, register a handler on that channel. For example, you can configure Spring Integration's `ErrorMessageExceptionTypeRouter` as the handler of an endpoint that is subscribed to the 'errorChannel'. That router can then spread the error messages across multiple channels, based on the `Exception` type.

Starting with version 4.3.10, Spring Integration provides the `ErrorMessagePublisher` and the `ErrorMessageStrategy`. You can use them as a general mechanism for publishing `ErrorMessage` instances. You can call or extend them in any error handling scenarios. The `ErrorMessageSendingRecoverer` extends this class as a `RecoveryCallback` implementation that can be used with retry, such as the `RequestHandlerRetryAdvice`. The `ErrorMessageStrategy` is used to build an `ErrorMessage` based on the provided exception and an `AttributeAccessor` context. It can be injected into any `MessageProducerSupport` or `MessagingGatewaySupport`. The `requestMessage` is stored under `ErrorMessageUtils.INPUT_MESSAGE_CONTEXT_KEY` in the `AttributeAccessor` context.

The `ErrorMessageStrategy` can use that `requestMessage` as the `originalMessage` property of the `ErrorMessage` it creates. The `DefaultErrorMessageStrategy` does exactly that.

Starting with version 5.2, all the `MessageHandlingException` instances thrown by the framework components, includes a component `BeanDefinition` resource and source to determine a configuration point form the exception. In case of XML configuration, a resource is an XML file path and source an XML tag with its `id` attribute. With Java & Annotation configuration, a resource is a `@Configuration` class and source is a `@Bean` method. In most case the target integration flow solution is based on the out-of-the-box components and their configuration options. When an exception happens at runtime, there is no any end-user code involved in stack trace because an execution is against beans, not their configuration. Including a resource and source of the bean definition helps to determine possible configuration mistakes and provides better developer experience.

Starting with version 5.4.3, the default error channel is configured with the property `requireSubscribers = true` to not silently ignore messages when there are no subscribers on this channel (e.g. when application context is stopped). In this case a `MessageDispatchingException` is thrown which may lend on the client callback of the inbound channel adapter to negatively acknowledge (or roll back) an original message in the source system for redelivery or other future consideration. To restore the previous behavior (ignore non dispatched error messages), the global integration property `spring.integration.channels.error.requireSubscribers` must be set to `false`. See Global Properties and `PublishSubscribeChannel` Configuration (if you configure a global `errorChannel` manually) for more information.

Version 5.4.4
Last updated 2021-02-17 19:24:37 UTC