Spring Integration fornece uma extensão do modelo de programação Spring para suportar os conhecidos Enterprise Integration Patterns . Ele permite mensagens leves em aplicativos baseados em Spring e oferece suporte à integração com sistemas externos por meio de adaptadores declarativos. Esses adaptadores fornecem um nível mais alto de abstração sobre o suporte do Spring para comunicação remota, mensagens e agendamento.

O objetivo principal do Spring Integration é fornecer um modelo simples para a construção de soluções de integração empresarial, mantendo a separação de interesses que é essencial para a produção de código passível de manutenção e teste.

Visão geral da integração do Spring

Este capítulo fornece uma introdução de alto nível aos principais conceitos e componentes do Spring Integration. Inclui algumas dicas de programação para ajudá-lo a aproveitar ao máximo a integração do Spring.

Fundo

Um dos principais temas do Spring Framework é a Inversão de Controle (IoC). Em seu sentido mais amplo, isso significa que a estrutura lida com responsabilidades em nome dos componentes que são gerenciados dentro de seu contexto. Os próprios componentes são simplificados, porque são dispensados dessas responsabilidades. Por exemplo, a injeção de dependência libera os componentes da responsabilidade de localizar ou criar suas dependências. Da mesma forma, a programação orientada a aspectos alivia os componentes de negócios de interesses transversais genéricos, modularizando-os em aspectos reutilizáveis. Em cada caso, o resultado final é um sistema mais fácil de testar, entender, manter e estender.

Além disso, a estrutura e o portfólio Spring fornecem um modelo de programação abrangente para a construção de aplicativos corporativos. Os desenvolvedores se beneficiam da consistência desse modelo e especialmente do fato de que ele é baseado em práticas recomendadas bem estabelecidas, como programação para interfaces e favorecimento da composição em vez da herança. As abstrações simplificadas do Spring e as poderosas bibliotecas de suporte aumentam a produtividade do desenvolvedor ao mesmo tempo em que aumentam o nível de testabilidade e portabilidade.

A integração do Spring é motivada por esses mesmos objetivos e princípios. Ele estende o modelo de programação Spring para o domínio de mensagens e se baseia no suporte de integração empresarial existente do Spring para fornecer um nível ainda mais alto de abstração. Ele oferece suporte a arquiteturas orientadas a mensagens em que a inversão de controle se aplica a questões de tempo de execução, como quando determinada lógica de

negócios deve ser executada e para onde a resposta deve ser enviada. Ele suporta o roteamento e a transformação de mensagens para que diferentes transportes e diferentes formatos de dados possam ser integrados sem afetar a testabilidade. Em outras palavras, as questões de mensagens e integração são tratadas pela estrutura. Os componentes de negócios são ainda mais isolados da infraestrutura e os desenvolvedores são dispensados de complexas responsabilidades de integração.

Como uma extensão do modelo de programação Spring, Spring Integration fornece uma ampla variedade de opções de configuração, incluindo anotações, XML com suporte a namespace, XML com elementos genéricos de “bean” e uso direto da API subjacente. Essa API é baseada em interfaces de estratégia bem definidas e adaptadores não invasivos de delegação. O design do Spring Integration é inspirado no reconhecimento de uma forte afinidade entre os padrões comuns no Spring e os padrões bem conhecidos descritos em *Enterprise Integration Patterns* , de Gregor Hohpe e Bobby Woolf (Addison Wesley, 2004). Os desenvolvedores que leram esse livro devem se sentir imediatamente à vontade com os conceitos e a terminologia do Spring Integration.

Objetivos e Princípios

A integração do Spring é motivada pelos seguintes objetivos:

- Fornece um modelo simples para implementar soluções complexas de integração empresarial.
- Facilite o comportamento assíncrono orientado por mensagens em um aplicativo baseado em Spring.
- Promova a adoção incremental e intuitiva para usuários existentes do Spring.

A integração Spring é guiada pelos seguintes princípios:

- Os componentes devem ser fracamente acoplados para modularidade e testabilidade.
- A estrutura deve impor a separação de interesses entre a lógica de negócios e a lógica de integração.
- Os pontos de extensão devem ser de natureza abstrata (mas dentro de limites bem definidos) para promover a reutilização e a portabilidade.

Componentes principais

De uma perspectiva vertical, uma arquitetura em camadas facilita a separação de interesses e os contratos baseados em interface entre as camadas promovem um acoplamento fraco. Os aplicativos baseados em Spring são normalmente projetados dessa maneira, e a estrutura e o

portfólio do Spring fornecem uma base sólida para seguir essa prática recomendada para toda a pilha de um aplicativo corporativo. As arquiteturas orientadas por mensagens adicionam uma perspectiva horizontal, mas esses mesmos objetivos ainda são relevantes. Assim como a “arquitetura em camadas” é um paradigma extremamente genérico e abstrato, os sistemas de mensagens geralmente seguem o modelo abstrato de “tubos e filtros”. Os “filtros” representam quaisquer componentes capazes de produzir ou consumir mensagens, e os “tubos” transportam as mensagens entre os filtros para que os próprios componentes permaneçam fracamente acoplados. É importante observar que esses dois paradigmas de alto nível não são mutuamente exclusivos. A infraestrutura de mensagens subjacente que suporta os “canais” ainda deve ser encapsulada em uma camada cujos contratos são definidos como interfaces. Da mesma forma, os próprios “filtros” devem ser gerenciados dentro de uma camada que está logicamente acima da camada de serviço do aplicativo, interagindo com esses serviços por meio de interfaces da mesma forma que uma camada da web faria.

Mensagem

No Spring Integration, uma mensagem é um wrapper genérico para qualquer objeto Java combinado com metadados usados pela estrutura ao manipular esse objeto. Consiste em uma carga útil e cabeçalhos. A carga útil pode ser de qualquer tipo e os cabeçalhos contêm informações comumente necessárias, como ID, carimbo de data / hora, ID de correlação e endereço de retorno. Os cabeçalhos também são usados para passar valores de e para transportes conectados. Por exemplo, ao criar uma mensagem a partir de um arquivo recebido, o nome do arquivo pode ser armazenado em um cabeçalho para ser acessado pelos componentes downstream. Da mesma forma, se o conteúdo de uma mensagem for finalmente enviado por um adaptador de correio de saída, as várias propriedades (para, de, cc, assunto e outros) podem ser configuradas como valores de cabeçalho de mensagem por um componente upstream. Os desenvolvedores também podem armazenar quaisquer pares de valores-chave arbitrários nos cabeçalhos.

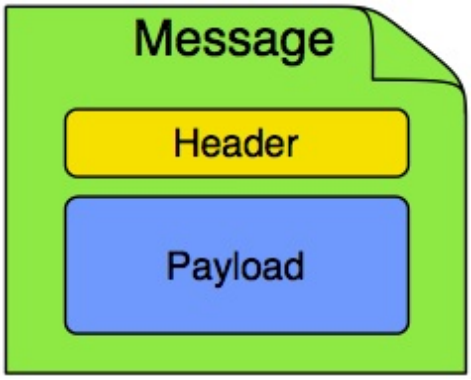

Figura 1. Mensagem

Canal de Mensagem

Um canal de mensagem representa o “tubo” de uma arquitetura de tubos e filtros. Os produtores enviam mensagens para um canal e os consumidores recebem mensagens de um canal. O canal de mensagem, portanto, desacopla os componentes de mensagem e também fornece um ponto conveniente para interceptação e monitoramento de mensagens.

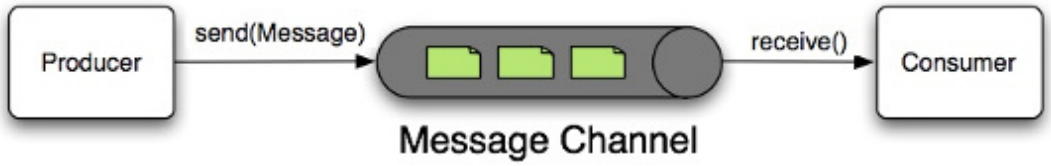

Figura 2. Canal de mensagem

Um canal de mensagem pode seguir a semântica ponto a ponto ou publicar-assinar. Com um canal ponto a ponto, não mais do que um consumidor pode receber cada mensagem enviada ao canal. Os canais de publicação-assinatura, por outro lado, tentam transmitir cada mensagem a todos os assinantes do canal. O Spring Integration oferece suporte a esses dois modelos.

Enquanto “ponto a ponto” e “publicar-assinar” definem as duas opções para quantos consumidores finalmente recebem cada mensagem, há outra consideração importante: O canal deve armazenar as mensagens em buffer? Na integração Spring, os canais pesquisáveis são capazes de armazenar mensagens em buffer em uma fila. A vantagem do armazenamento em buffer é que ele permite controlar as mensagens de entrada e, assim, evitar sobrecarregar um consumidor. No entanto, como o nome sugere, isso também adiciona alguma complexidade, uma vez que um consumidor só pode receber as mensagens de tal canal se um poller estiver configurado. Por outro lado, um consumidor conectado a um canal assinável é simplesmente orientado por mensagens. Implementações de canal de mensagem apresenta uma discussão detalhada sobre a variedade de implementações de canal disponíveis no Spring Integration.

Endpoint de mensagem

Um dos principais objetivos do Spring Integration é simplificar o desenvolvimento de soluções de integração empresarial por meio da inversão de controle. Isso significa que não deve ser necessário implementar consumidores e produtores diretamente e nem mesmo criar mensagens e invocar operações de envio ou recebimento em um canal de mensagem. Em vez disso, você deve ser capaz de se concentrar em seu modelo de domínio específico com uma implementação baseada em objetos simples. Então, ao fornecer configuração declarativa, você pode “conectar” seu código específico de domínio à infraestrutura de mensagens fornecida pelo Spring Integration. Os componentes responsáveis por essas conexões são terminais de mensagens. Isso não significa que você deve necessariamente conectar o código do aplicativo existente diretamente. Qualquer solução de integração corporativa do mundo real requer alguma quantidade de código focada em questões de integração, como roteamento e transformação. O importante é conseguir a separação de interesses entre a lógica de integração e a lógica de negócios. Em outras palavras, como com o paradigma Model-View-Controller (MVC) para aplicativos da web, o objetivo deve ser fornecer uma camada fina, mas dedicada, que traduz as

solicitações de entrada em invocações da camada de serviço e, em seguida, traduz os valores de retorno da camada de serviço em respostas de saída. A próxima seção fornece uma visão geral dos tipos de terminal de mensagem que lidam com essas responsabilidades e, nos próximos capítulos, você pode ver como as opções de configuração declarativa do Spring Integration fornecem uma maneira não invasiva de usar cada um deles. O importante é conseguir a separação de interesses entre a lógica de integração e a lógica de negócios. Em outras palavras, como com o paradigma Model-View-Controller (MVC) para aplicativos da web, o objetivo deve ser fornecer uma camada fina, mas dedicada, que traduz as solicitações de entrada em invocações da camada de serviço e, em seguida, traduz os valores de retorno da camada de serviço em respostas de saída. A próxima seção fornece uma visão geral dos tipos de terminal de mensagem que lidam com essas responsabilidades e, nos próximos capítulos, você pode ver como as opções de configuração declarativa do Spring Integration fornecem uma maneira não invasiva de usar cada um deles. O importante é conseguir a separação de interesses entre a lógica de integração e a lógica de negócios. Em outras palavras, como com o paradigma Model-View-Controller (MVC) para aplicativos da web, o objetivo deve ser fornecer uma camada fina, mas dedicada, que traduz as solicitações de entrada em invocações da camada de serviço e, em seguida, traduz os valores de retorno da camada de serviço em respostas de saída. A próxima seção fornece uma visão geral dos tipos de terminal de mensagem que lidam com essas responsabilidades e, nos próximos capítulos, você pode ver como as opções de configuração declarativa do Spring Integration fornecem uma maneira não invasiva de usar cada um deles. o objetivo deve ser fornecer uma camada fina, mas dedicada, que traduz as solicitações de entrada em invocações da camada de serviço e, em seguida, converte os valores de retorno da camada de serviço em respostas de saída. A próxima seção fornece uma visão geral dos tipos de terminal de mensagem que lidam com essas responsabilidades e, nos próximos capítulos, você pode ver como as opções de configuração declarativa do Spring Integration fornecem uma maneira não invasiva de usar cada um deles. o objetivo deve ser fornecer uma camada fina, mas dedicada, que traduz as solicitações de entrada em invocações da camada de serviço e, em seguida, converte os valores de retorno da camada de serviço em respostas de saída. A próxima seção fornece uma visão geral dos tipos de terminal de mensagem que lidam com essas responsabilidades e, nos próximos capítulos, você pode ver como as opções de configuração declarativa do Spring Integration fornecem uma maneira não invasiva de usar cada um deles.

Terminais de mensagem

Um Message Endpoint representa o “filtro” de uma arquitetura de tubos e filtros. Conforme mencionado anteriormente, a função principal do terminal é conectar o código do aplicativo à estrutura de mensagens e fazer isso de maneira não invasiva. Em outras palavras, o código do aplicativo idealmente não deve ter consciência dos objetos de mensagem ou dos canais de mensagem. Isso é semelhante ao papel de um controlador no paradigma MVC. Assim como um controlador lida com solicitações HTTP, o ponto de extremidade de mensagem lida com

mensagens. Assim como os controladores são mapeados para padrões de URL, os terminais de mensagens são mapeados para canais de mensagens. O objetivo é o mesmo em ambos os casos: isolar o código do aplicativo da infraestrutura. Esses conceitos e todos os padrões que se seguem são discutidos detalhadamente no *Enterprise Integration Patterns*livro. Aqui, fornecemos apenas uma descrição de alto nível dos principais tipos de endpoint suportados pelo Spring Integration e as funções associadas a esses tipos. Os capítulos a seguir elaboram e fornecem códigos de amostra, bem como exemplos de configuração.

Transformador de Mensagem

Um transformador de mensagem é responsável por converter o conteúdo ou estrutura de uma mensagem e retornar a mensagem modificada. Provavelmente, o tipo mais comum de transformador é aquele que converte a carga útil da mensagem de um formato para outro (como de XML para `java.lang.String`). Da mesma forma, um transformador pode adicionar, remover ou modificar os valores do cabeçalho da mensagem.

Filtro de Mensagens

Um filtro de mensagem determina se uma mensagem deve ser passada para um canal de saída. Isso simplesmente requer um método de teste booleano que pode verificar um determinado tipo de conteúdo de carga útil, um valor de propriedade, a presença de um cabeçalho ou outras condições. Se a mensagem for aceita, ela é enviada para o canal de saída. Caso contrário, ele é eliminado (ou, para uma implementação mais severa, um `Exception`poderia ser lançado). Os filtros de mensagens são frequentemente usados em conjunto com um canal de publicação-assinatura, onde vários consumidores podem receber a mesma mensagem e usar os critérios do filtro para restringir o conjunto de mensagens a serem processadas.

> Tenha cuidado para não confundir o uso genérico de “filtro” dentro do padrão arquitetônico de tubos e filtros com este tipo de ponto de extremidade específico que seletivamente restringe as mensagens que fluem entre dois canais. O conceito de tubos e filtros de um “filtro” corresponde mais de perto ao terminal de mensagem do Spring Integration: qualquer componente que pode ser conectado a um canal de mensagem para enviar ou receber mensagens.

Roteador de Mensagens

Um roteador de mensagem é responsável por decidir qual canal ou canais (se houver) devem receber a mensagem a seguir. Normalmente, a decisão é baseada no conteúdo da mensagem

ou nos metadados disponíveis nos cabeçalhos das mensagens. Um roteador de mensagem é freqüentemente usado como uma alternativa dinâmica para um canal de saída configurado estaticamente em um ativador de serviço ou outro terminal capaz de enviar mensagens de resposta. Da mesma forma, um roteador de mensagens fornece uma alternativa proativa aos filtros de mensagens reativos usados por vários assinantes, conforme descrito anteriormente.

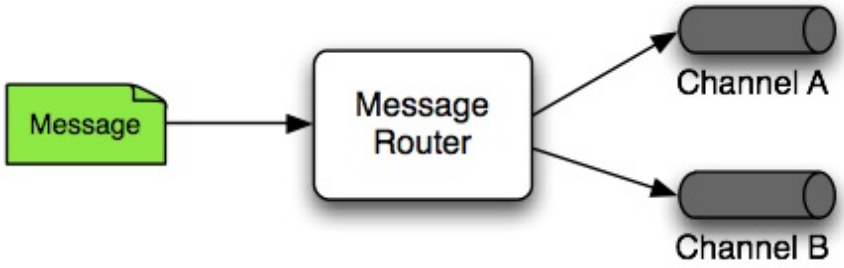

Figura 3. Roteador de mensagens

Divisor

Um divisor é outro tipo de terminal de mensagem cuja responsabilidade é aceitar uma mensagem de seu canal de entrada, dividir essa mensagem em várias mensagens e enviar cada uma delas para seu canal de saída. Isso é normalmente usado para dividir um objeto de carga útil “composto” em um grupo de mensagens contendo as cargas úteis subdivididas.

Agregador

Basicamente, uma imagem espelhada do divisor, o agregador é um tipo de terminal de mensagem que recebe várias mensagens e as combina em uma única mensagem. Na verdade, os agregadores costumam ser consumidores downstream em um pipeline que inclui um divisor. Tecnicamente, o agregador é mais complexo do que um divisor, porque é necessário para manter o estado (as mensagens a serem agregadas), para decidir quando o grupo completo de mensagens está disponível e para o tempo limite, se necessário. Além disso, em caso de tempo limite, o agregador precisa saber se deve enviar os resultados parciais, descartá-los ou enviá-los para um canal separado. O Spring Integration fornece um `CorrelationStrategy`, `ReleaseStrategy` ae configurações configuráveis para o tempo limite, se deve enviar resultados parciais após o tempo limite e um canal de descarte.

Service Activator

Um Service Activator é um ponto de extremidade genérico para conectar uma instância de serviço ao sistema de mensagens. O canal de mensagem de entrada deve ser configurado e, se o método de serviço a ser chamado for capaz de retornar um valor, um Canal de mensagem de saída também pode ser fornecido.

O canal de saída é opcional, pois cada mensagem também pode fornecer seu próprio cabeçalho de 'Endereço de retorno'. Esta mesma regra se aplica a todos os terminais do consumidor.

O ativador de serviço invoca uma operação em algum objeto de serviço para processar a mensagem de solicitação, extraindo a carga útil da mensagem de solicitação e convertendo (se o método não esperar um parâmetro digitado na mensagem). Sempre que o método do objeto de serviço retorna um valor, esse valor de retorno é igualmente convertido em uma mensagem de resposta, se necessário (se ainda não for um tipo de mensagem). Essa mensagem de resposta é enviada para o canal de saída. Caso nenhum canal de saída tenha sido configurado, a resposta é enviada para o canal especificado no “endereço de retorno” da mensagem, se disponível.

Um ponto de extremidade ativador de serviço de solicitação-resposta conecta o método de um objeto de destino aos canais de mensagem de entrada e saída.

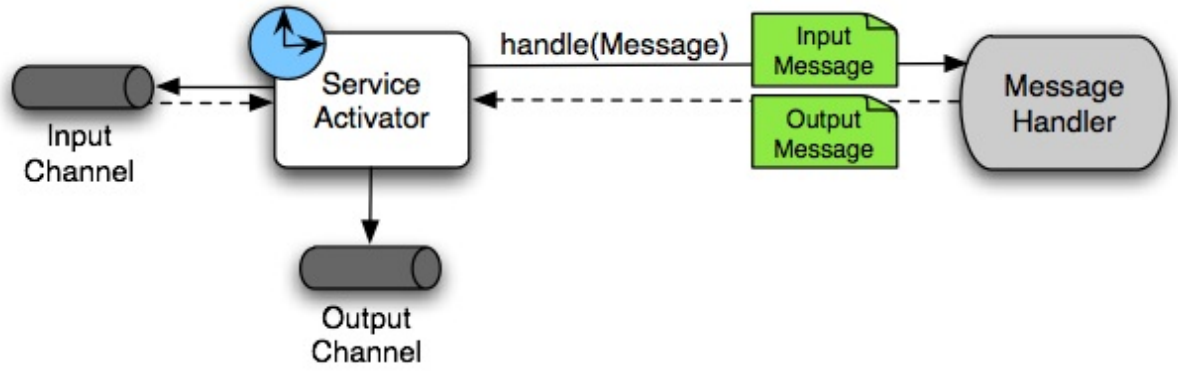

Figura 4. Ativador de serviço

Conforme discutido anteriormente, no Message Channel , os canais podem ser pesquisáveis ou assináveis. No diagrama anterior, isso é representado pelo símbolo do “relógio” e a seta sólida (pesquisa) e a seta pontilhada (assinar).

Adaptador de canal

Um adaptador de canal é um nó de extremidade que conecta um canal de mensagem a algum outro sistema ou transporte. Os adaptadores de canal podem ser de entrada ou de saída. Normalmente, o adaptador de canal faz algum mapeamento entre a mensagem e qualquer objeto ou recurso recebido ou enviado para o outro sistema (arquivo, Solicitação HTTP, mensagem JMS e outros). Dependendo do transporte, o adaptador de canal também pode preencher ou extrair os valores do cabeçalho da mensagem. O Spring Integration fornece vários adaptadores de canal, que são descritos nos próximos capítulos.

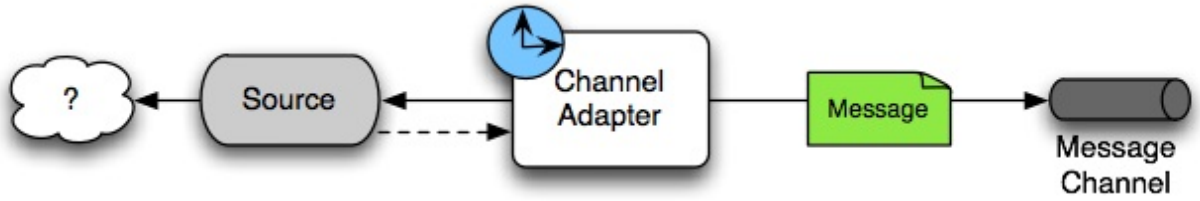

Figura 5. Um terminal de adaptador de canal de entrada conecta um sistema de origem a um `MessageChannel`.

> As fontes de mensagens podem ser pesquisáveis (por exemplo, POP3) ou orientadas por mensagens (por exemplo, IMAP ocioso). No diagrama anterior, isso é representado pelo símbolo “relógio” e a seta sólida (pesquisa) e a seta pontilhada (orientada por mensagem).

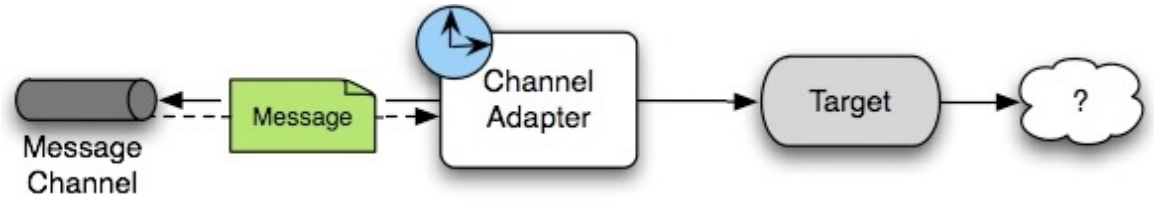

Figura 6. Um terminal de adaptador de canal de saída conecta a `MessageChannel` a um sistema de destino.

> Conforme discutido anteriormente em Message Channel , os canais podem ser pesquisáveis ou assináveis. No diagrama anterior, isso é representado pelo símbolo do “relógio” e a seta sólida (pesquisa) e a seta pontilhada (assinar).

Nomes de bean de endpoint

Os terminais de consumo (qualquer coisa com um `inputChannel`) consistem em dois beans, o consumidor e o manipulador de mensagens. O consumidor tem uma referência ao manipulador de mensagens e o invoca quando as mensagens chegam.

Considere o seguinte exemplo de XML:

```
<int:service-activator id = "someService" ... />
```

Dado o exemplo anterior, os nomes dos beans são os seguintes:

- Consumidor: `someService` (o `id`)

- Handler: `someService.handler`

Ao usar anotações Enterprise Integration Pattern (EIP), os nomes dependem de vários fatores. Considere o seguinte exemplo de um POJO anotado:

```
@Component
public class SomeComponent {

    @ServiceActivator(inputChannel = ...)
    public String someMethod(...) {
        ...
    }

}
```

Dado o exemplo anterior, os nomes dos beans são os seguintes:

- Consumidor: `someComponent.someMethod.serviceActivator`
- Handler: `someComponent.someMethod.serviceActivator.handler`

A partir da versão 5.0.4, você pode modificar esses nomes usando a `@EndpointId` anotação, como mostra o exemplo a seguir:

```
@Component
public class SomeComponent {

    @EndpointId("someService")
    @ServiceActivator(inputChannel = ...)
    public String someMethod(...) {
        ...
    }

}
```

Dado o exemplo anterior, os nomes dos beans são os seguintes:

- Consumidor: `someService`
- Handler: `someService.handler`

O `@EndpointId` cria nomes conforme criados pelo `id` atributo com configuração XML. Considere o seguinte exemplo de um bean anotado:

```
@Configuration
public class SomeConfiguration {

    @Bean
    @ServiceActivator(inputChannel = ...)
    public MessageHandler someHandler() {
        ...
    }

}
```

Dado o exemplo anterior, os nomes dos beans são os seguintes:

- Consumidor: `someConfiguration.someHandler.serviceActivator`
- Manipulador: `someHandler` (o `@Bean` nome)

A partir da versão 5.0.4, você pode modificar esses nomes usando a `@EndpointId` anotação, como mostra o exemplo a seguir:

```
@Configuration
public class SomeConfiguration {

    @Bean("someService.handler")
    @EndpointId("someService")
    @ServiceActivator(inputChannel = ...)
    public MessageHandler someHandler() {
        ...
    }

}
```

1. Manipulador: `someService.handler` (o nome do bean)
2. Consumidor: `someService` (o ID do endpoint)

A `@EndpointId` anotação cria nomes conforme criados pelo `id` atributo com a configuração XML, contanto que você use a convenção de anexar `.handler` ao `@Bean` nome.

Há um caso especial em que um terceiro bean é criado: Por motivos arquitetônicos, se a `MessageHandler` `@Bean` não definir um `AbstractReplyProducingMessageHandler`, a estrutura envolverá o bean fornecido em a `ReplyProducingMessageHandlerWrapper`. Este wrapper suporta o tratamento de conselhos do manipulador de solicitações e emite as mensagens normais de log de depuração 'não produziu resposta'. Seu nome de bean é o nome

do handler bean plus `.wrapper` (quando há um `@EndpointId` - caso contrário, é o nome do handler gerado normal).

Similarmente, as Fontes de Mensagens Pollable criam dois beans, um `SourcePollingChannelAdapter` (SPCA) e um `MessageSource`.

Considere a seguinte configuração XML:

```
<int:inbound-channel-adapter id = "someAdapter" ... />
```

Dada a configuração XML anterior, os nomes dos beans são os seguintes:

- SPCA: `someAdapter` (o `id`)
- Handler: `someAdapter.source`

Considere a seguinte configuração Java de um POJO para definir um `@EndpointId`:

```
@EndpointId("someAdapter")
@InboundChannelAdapter(channel = "channel3", poller = @Poller(fixedDelay = "5000
public String pojoSource() {
    ...
}
```

Dado o exemplo de configuração Java anterior, os nomes dos beans são os seguintes:

- SPCA: `someAdapter`
- Handler: `someAdapter.source`

Considere a seguinte configuração Java de um bean para definir um `@EndpointID`:

```
@Bean("someAdapter.source")
@EndpointId("someAdapter")
@InboundChannelAdapter(channel = "channel3", poller = @Poller(fixedDelay = "5000
public MessageSource<?> source() {
    return () -> {
        ...
    };
}
```

Dado o exemplo anterior, os nomes dos beans são os seguintes:

- SPCA: `someAdapter`
- Manipulador: `someAdapter.source` (contanto que você use a convenção de anexar `.source` ao `@Bean` nome)

Configuração e `@EnableIntegration`

Ao longo deste documento, você pode ver referências ao suporte de namespace XML para declarar elementos em um fluxo de integração Spring. Esse suporte é fornecido por uma série de analisadores de espaço de nomes que geram definições de bean apropriadas para implementar um componente específico. Por exemplo, muitos terminais consistem em um `MessageHandler` bean e um `ConsumerEndpointFactoryBean` no qual o manipulador e um nome de canal de entrada são injetados.

Na primeira vez que um elemento de namespace Spring Integration é encontrado, a estrutura declara automaticamente vários beans (um planejador de tarefa, um criador de canal implícito e outros) que são usados para suportar o ambiente de tempo de execução.

> A versão 4.0 introduziu a `@EnableIntegration` anotação, para permitir o registro dos beans de infraestrutura do Spring Integration (consulte o Javadoc). Esta anotação é necessária quando apenas a configuração Java é usada - por exemplo, com Spring Boot ou Spring Integration Messaging Annotation suporte e Spring Integration Java DSL sem configuração de integração XML.

A `@EnableIntegration` anotação também é útil quando você tem um contexto pai sem componentes Spring Integration e dois ou mais contextos filhos que usam Spring Integration. Ele permite que esses componentes comuns sejam declarados apenas uma vez, no contexto pai.

A `@EnableIntegration` anotação registra muitos componentes de infraestrutura com o contexto do aplicativo. Em particular, ele:

- Registra alguns beans embutidos, como `errorChannel` e its `LoggingHandler` , `taskScheduler` para pollers, `jsonPath` função SpEL e outros.
- Adiciona várias `BeanFactoryPostProcessor` instâncias para aprimorar o `BeanFactory` ambiente de integração global e padrão.
- Adiciona várias `BeanPostProcessor` instâncias para aprimorar ou converter e agrupar beans específicos para fins de integração.

- Adiciona processadores de anotação para analisar anotações de mensagens e registra componentes para elas com o contexto do aplicativo.

A `@IntegrationComponentScan` anotação também permite a varredura de caminho de classe. Essa anotação desempenha uma função semelhante à `@ComponentScan` anotação padrão do Spring Framework , mas é restrita a componentes e anotações que são específicos para Spring Integration, que o mecanismo de varredura de componente padrão do Spring Framework não pode alcançar. Para obter um exemplo, consulte `@MessagingGateway` Anotação .

A `@EnablePublisher` anotação registra um `PublisherAnnotationBeanPostProcessor` bean e configura o `default-publisher-channel` para aquelas `@Publisher` anotações que são fornecidas sem um `channel` atributo. Se mais de uma `@EnablePublisher` anotação for encontrada, todas devem ter o mesmo valor para o canal padrão. Consulte Configuração orientada por `@Publisher` anotação com a anotação para obter mais informações.

A `@GlobalChannelInterceptor` anotação foi introduzida para marcar `ChannelInterceptor` beans para interceptação de canal global. Esta anotação é análoga ao `<int:channel-interceptor>` elemento XML (consulte Configuração do Global Channel Interceptor). `@GlobalChannelInterceptor` as anotações podem ser colocadas no nível da classe (com uma `@Component` anotação de estereótipo) ou em `@Bean` métodos dentro das `@Configuration` classes. Em qualquer caso, o bean deve implementar `ChannelInterceptor` .

A partir da versão 5.1, os interceptores de canal global se aplicam a canais registrados dinamicamente - como beans que são inicializados usando `beanFactory.initializeBean()` ou por meio de `IntegrationFlowContext` ao usar o DSL Java. Anteriormente, os interceptores não eram aplicados quando os beans eram criados após a atualização do contexto do aplicativo.

As `@IntegrationConverter` marcas de anotação `Converter` , `GenericConverter` ou `ConverterFactory` feijão como conversores de candidatos para `integrationConversionService` . Essa anotação é um análogo do `<int:converter>` elemento XML (consulte Conversão de tipo de carga útil). Você pode colocar `@IntegrationConverter` anotações no nível da classe (com uma `@Component` anotação de estereótipo) ou em `@Bean` métodos dentro das `@Configuration` classes.

Consulte Suporte a anotações para obter mais informações sobre anotações de mensagens.

Considerações de programação

Você deve usar objetos Java simples (POJOs) sempre que possível e apenas expor a estrutura em seu código quando for absolutamente necessário. Consulte a invocação do método POJO para obter mais informações.

Se você expõe a estrutura para suas classes, há algumas considerações que precisam ser levadas em consideração, especialmente durante a inicialização do aplicativo:

- Se o seu componente for `ApplicationContextAware`, geralmente você não deve usar o `ApplicationContext` no `setApplicationContext()` método. Em vez disso, armazene uma referência e adie tais usos para mais tarde no ciclo de vida do contexto.
- Se o seu componente for um `InitializingBean` ou usar `@PostConstruct` métodos, não envie mensagens desses métodos de inicialização. O contexto do aplicativo ainda não foi inicializado quando esses métodos são chamados e o envio dessas mensagens provavelmente falhará. Se você precisar enviar mensagens durante a inicialização, implemente `ApplicationListener` e aguarde o `ContextRefreshedEvent`. Como alternativa, implemente `SmartLifecycle`, coloque seu bean em uma fase tardia e envie as mensagens do `start()` método.

Considerações ao usar potes empacotados (por exemplo, sombreados)

O Spring Integration inicializa certos recursos usando o `SpringFactories` mecanismo do Spring Framework para carregar várias `IntegrationConfigurationInitializer` classes. Isso inclui o `-core` jarro, bem como alguns outros, incluindo `-http` e `-jmx`. As informações desse processo são armazenadas em um `META-INF/spring.factories` arquivo em cada jar.

Alguns desenvolvedores preferem reempacotar seus aplicativos e todas as dependências em um único jar usando ferramentas bem conhecidas, como o Apache Maven Shade Plugin.

Por padrão, o plug-in de sombra não mescla os `spring.factories` arquivos ao produzir o jar sombreado.

Além disso `spring.factories`, outros `META-INF` arquivos (`spring.handlers` e `spring.schemas`) são usados para configuração XML. Esses arquivos também precisam ser mesclados.

> O mecanismo executável de jar do Spring Boot tem uma abordagem diferente, pois ele aninha os jars, mantendo assim cada `spring.factories` arquivo no caminho da classe.Portanto, com um aplicativo Spring Boot, nada mais é necessário se você usar seu formato jar executável padrão.

Mesmo se você não usar o Spring Boot, você ainda pode usar as ferramentas fornecidas pelo Boot para aprimorar o plug-in de sombra adicionando transformadores para os arquivos mencionados acima.

Você pode querer consultar o pom spring-boot-starter-parent atual para ver as configurações atuais que o boot usa. O exemplo a seguir mostra como configurar o plug-in:

Exemplo 1. pom.xml

```
...
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <configuration>
                <keepDependenciesWithProvidedScope>true</keepDependenciesWithPro
                <createDependencyReducedPom>true</createDependencyReducedPom>
            </configuration>
            <dependencies>
                <dependency>
                    <groupId>org.springframework.boot</groupId>
                    <artifactId>spring-boot-maven-plugin</artifactId>
                    <version>${spring.boot.version}</version>
                </dependency>
            </dependencies>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>shade</goal>
                    </goals>
                    <configuration>
                        <transformers>
                            <transformer
                                implementation="org.apache.maven.plugins.shade.r
                                <resource>META-INF/spring.handlers</resource>
                            </transformer>
                            <transformer
                                implementation="org.springframework.boot.maven.P
                                <resource>META-INF/spring.factories</resource>
                            </transformer>
                            <transformer
                                implementation="org.apache.maven.plugins.shade.r
                                <resource>META-INF/spring.schemas</resource>
                            </transformer>
                            <transformer
                                implementation="org.apache.maven.plugins.shade.r
                        </transformers>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
...
```

Especificamente,

- Adicione o `spring-boot-maven-plugin` como uma dependência.
- Configure os transformadores.

Você pode adicionar uma propriedade `${spring.boot.version}` ou usar uma versão explícita.

Dicas e truques de programação

Esta seção documenta algumas das maneiras de obter o máximo da integração do Spring.

Esquemas XML

Ao usar a configuração XML, para evitar erros de validação de esquema falsos, você deve usar um IDE “Spring-aware”, como o Spring Tool Suite (STS), Eclipse com os plug-ins Spring IDE ou IntelliJ IDEA. Esses IDEs sabem como resolver o esquema XML correto do classpath (usando o `META-INF/spring.schemas` arquivo nos jars). Ao usar STS ou Eclipse com o plugin, você deve habilitar `Spring Project Nature` no projeto.

Os esquemas hospedados na Internet para determinados módulos legados (aqueles que existiam na versão 1.0) são as versões 1.0 por motivos de compatibilidade. Se o seu IDE usa esses esquemas, é provável que você veja erros falsos.

Cada um desses esquemas online tem um aviso semelhante ao seguinte:

> Este esquema é para a versão 1.0 do Spring Integration Core. Não podemos atualizá-lo para o esquema atual porque isso interromperá todos os aplicativos que usam 1.0.3 ou inferior. Para versões subsequentes, o esquema não versionado é resolvido do classpath e obtido do jar. Consulte o github:
>
> https://github.com/spring-projects/spring-integration/tree/master/spring-integration-core/src/main/resources/org/springframework/integration/config

Os módulos afetados são

- `core` (`spring-integration.xsd`)
- `file`
- `http`
- `jms`

- `mail`
- `rmi`
- `security`
- `stream`
- `ws`
- `xml`

Encontrando nomes de classes para configuração Java e DSL

Com configuração XML e suporte a Spring Integration Namespace, os analisadores XML ocultam como os beans de destino são declarados e conectados juntos. Para a configuração Java, é importante entender a API Framework para aplicativos de usuário final de destino.

Os cidadãos de primeira classe para a implementação EIP são `Message`, `Channel` e `Endpoint` (ver Componentes Principais , anteriormente neste capítulo). Suas implementações (contratos) são:

- `org.springframework.messaging.Message`: Veja a mensagem ;
- `org.springframework.messaging.MessageChannel`: Veja Canais de Mensagem ;
- `org.springframework.integration.endpoint.AbstractEndpoint`: Veja Poller .

Os dois primeiros são simples o suficiente para entender como implementar, configurar e usar. O último merece mais atenção

O `AbstractEndpoint` é amplamente utilizado em todo o Spring Framework para diferentes implementações de componentes. Suas principais implementações são:

- `EventDrivenConsumer`, usado quando assinamos um `SubscribableChannel` para ouvir mensagens.
- `PollingConsumer`, usado quando pesquisamos mensagens de a `PollableChannel`.

Ao usar anotações de mensagens ou o Java DSL, você precisa se preocupar com esses componentes, porque o Framework os produz automaticamente com anotações e `BeanPostProcessor` implementações apropriadas . Ao construir componentes manualmente, você deve usar o `ConsumerEndpointFactoryBean` para ajudar a determinar a `AbstractEndpoint` implementação do consumidor de destino a ser criada, com base na `inputChannel` propriedade fornecida .

Por outro lado, os `ConsumerEndpointFactoryBean` delegados a um outro cidadão de primeira classe no Quadro - `org.springframework.messaging.MessageHandler`. O objetivo da

implementação dessa interface é tratar a mensagem consumida pelo terminal do canal. Todos os componentes em EIP Integração da mola são `MessageHandler` implementações (por exemplo, `AggregatingMessageHandler`, `MessageTransformingHandler`, `AbstractMessageSplitter`, e outros). Os adaptadores de saída protocolo alvo (`FileWritingMessageHandler`, `HttpRequestExecutingMessageHandler`, `AbstractMqttMessageHandler`, e outros) são também `MessageHandler` implementações. Ao desenvolver aplicativos Spring Integration com configuração Java, você deve olhar para o módulo Spring Integration para encontrar uma `MessageHandler` implementação apropriada a ser usada para a `@ServiceActivator` configuração. Por exemplo, para enviar uma mensagem XMPP (consulte Suporte XMPP) você deve configurar algo como o seguinte:

```
@Bean
@ServiceActivator(inputChannel = "input")
public MessageHandler sendChatMessageHandler(XMPPConnection xmppConnection) {
    ChatMessageSendingMessageHandler handler = new ChatMessageSendingMessageHand

    DefaultXmppHeaderMapper xmppHeaderMapper = new DefaultXmppHeaderMapper();
    xmppHeaderMapper.setRequestHeaderNames("*");
    handler.setHeaderMapper(xmppHeaderMapper);

    return handler;
}
```

As `MessageHandler` implementações representam a parte de saída e processamento do fluxo de mensagens.

O lado do fluxo de mensagens de entrada possui seus próprios componentes, que são divididos em comportamentos de pesquisa e atendimento. Os componentes de escuta (orientados por mensagem) são simples e normalmente requerem apenas uma implementação de classe de destino para estar pronta para produzir mensagens. Os componentes de escuta podem ser `MessageProducerSupport` implementações unilaterais (como `AbstractMqttMessageDrivenChannelAdapter` e `ImapIdleChannelAdapter`) ou `MessagingGatewaySupport` implementações de solicitação-resposta (como `AmqpInboundGateway` e `AbstractWebServiceInboundGateway`).

Os pontos de extremidade de entrada de pesquisa são para aqueles protocolos que não fornecem uma API de ouvinte ou não se destinam a tal comportamento, incluindo qualquer protocolo baseado em arquivo (como FTP), quaisquer bancos de dados (RDBMS ou NoSQL) e outros.

Esses terminais de entrada consistem em dois componentes: a configuração do poller, para iniciar a tarefa de pesquisa periodicamente, e uma classe de origem da mensagem para ler dados do protocolo de destino e produzir uma mensagem para o fluxo de integração de

recebimento de dados. A primeira classe para a configuração do poller é a `SourcePollingChannelAdapter`. É mais uma `AbstractEndpoint` implementação, mas especialmente para polling para iniciar um fluxo de integração. Normalmente, com as anotações de mensagens ou Java DSL, você não deve se preocupar com esta classe. O Framework produz um bean para ele, com base na `@InboundChannelAdapter` configuração ou em uma especificação do construtor Java DSL.

Os componentes de origem da mensagem são mais importantes para o desenvolvimento do aplicativo de destino e todos implementam a `MessageSource` interface (por exemplo, `MongoDbMessageSource` e `AbstractTwitterMessageSource`). Com isso em mente, nossa configuração para ler dados de uma tabela RDBMS com JDBC pode ser semelhante a:

```
@Bean
@InboundChannelAdapter(value = "fooChannel", poller = @Poller(fixedDelay="5000")
public MessageSource<?> storedProc(DataSource dataSource) {
    return new JdbcPollingChannelAdapter(dataSource, "SELECT * FROM foo where st
}
```

Você pode encontrar todas as classes de entrada e saída necessárias para os protocolos de destino no módulo Spring Integration específico (na maioria dos casos, no respectivo pacote). Por exemplo, os `spring-integration-websocket` adaptadores são:

- `o.s.i.websocket.inbound.WebSocketInboundChannelAdapter`: Implementa `MessageProducerSupport` para escutar frames no socket e produzir mensagem para o canal.
- `o.s.i.websocket.outbound.WebSocketOutboundMessageHandler`: A `AbstractMessageHandler` implementação unilateral para converter as mensagens recebidas no quadro apropriado e enviar pelo websocket.

Se você estiver familiarizado com a configuração do Spring Integration XML, começando com a versão 4.3, fornecemos informações nas definições de elemento XSD sobre quais classes de destino são usadas para declarar beans para o adaptador ou gateway, como mostra o exemplo a seguir:

```
<xsd:element name="outbound-async-gateway">
    <xsd:annotation>
                <xsd:documentation>
Configures a Consumer Endpoint for the 'o.s.i.amqp.outbound.AsyncAmqpOutboundGat
that will publish an AMQP Message to the provided Exchange and expect a reply Me
The sending thread returns immediately; the reply is sent asynchronously; uses '
       </xsd:documentation>
        </xsd:annotation>
```

Invocação do método POJO

Conforme discutido em Considerações de programação , recomendamos o uso de um estilo de programação POJO, como mostra o exemplo a seguir:

```
@ServiceActivator
public String myService(String payload) { ... }
```

Nesse caso, a estrutura extrai uma `String` carga útil, chama seu método e agrupa o resultado em uma mensagem para enviar ao próximo componente no fluxo (os cabeçalhos originais são copiados para a nova mensagem). Na verdade, se você usar a configuração XML, nem mesmo precisará da `@ServiceActivator` anotação, como mostram os exemplos emparelhados a seguir:

```
<int:service-activator ... ref="myPojo" method="myService" />
```

```
public String myService(String payload) { ... }
```

Você pode omitir o `method` atributo, desde que não haja ambigüidade nos métodos públicos da classe.

Você também pode obter informações de cabeçalho em seus métodos POJO, como mostra o exemplo a seguir:

```
@ServiceActivator
public String myService(@Payload String payload, @Header("foo") String fooHeader
```

Você também pode cancelar a referência de propriedades na mensagem, como mostra o exemplo a seguir:

```
@ServiceActivator
public String myService(@Payload("payload.foo") String foo, @Header("bar.baz") S
```

Como várias invocações de método POJO estão disponíveis, as versões anteriores à 5.0 usavam SpEL (Spring Expression Language) para invocar os métodos POJO. SpEL (mesmo interpretado) é geralmente “rápido o suficiente” para essas operações, quando comparado ao trabalho real normalmente realizado nos métodos. No entanto, a partir da versão 5.0, o `org.springframework.messaging.handler.invocation.InvocableHandlerMethod` é usado por padrão sempre que possível. Essa técnica geralmente é mais rápida de executar do que o SpEL interpretado e é consistente com outros projetos de mensagens Spring. O `InvocableHandlerMethod` é semelhante à técnica usada para invocar métodos de controlador no Spring MVC. Existem certos métodos que ainda são sempre invocados ao usar o SpEL. Os exemplos incluem parâmetros anotados com propriedades não referenciadas, conforme discutido anteriormente. Isso ocorre porque SpEL tem a capacidade de navegar por um caminho de propriedade.

Pode haver alguns outros casos extremos que não consideramos e que também não funcionam com `InvocableHandlerMethod` instâncias. Por esse motivo, voltamos automaticamente a usar o SpEL nesses casos.

Se desejar, você também pode configurar seu método POJO de forma que sempre use SpEL, com a `UseSpelInvoker` anotação, como mostra o exemplo a seguir:

```
@UseSpelInvoker(compilerMode = "IMMEDIATE")
public void bar(String bar) { ... }
```

Se a `compilerMode` propriedade for omitida, a `spring.expression.compiler.mode` propriedade do sistema determina o modo do compilador. Consulte a compilação do SpEL para obter mais informações sobre o SpEL compilado.

Versão 5.4.4
Última atualização 2021-02-17 19:24:37 UTC