

Разработка интернет-магазина на Spring Framework

Spring Integration

Spring Integration.

Оглавление

[Spring Integration](#)

[Введение](#)

[EIP](#)

[Некоторые паттерны интеграции](#)

[Ответственность интеграционного фреймворка](#)

[Spring Integration](#)

[Принципы Spring Integration](#)

[Настройка. Зависимости. Пуск](#)

[Hello World](#)

[Интеграция разрозненных приложений](#)

[Компоненты Spring Integration](#)

[Сообщение \(Message\)](#)

[Канал \(Channel\)](#)

[Шлюз \(Gateway\)](#)

[Заключение](#)

[Практическое задание](#)

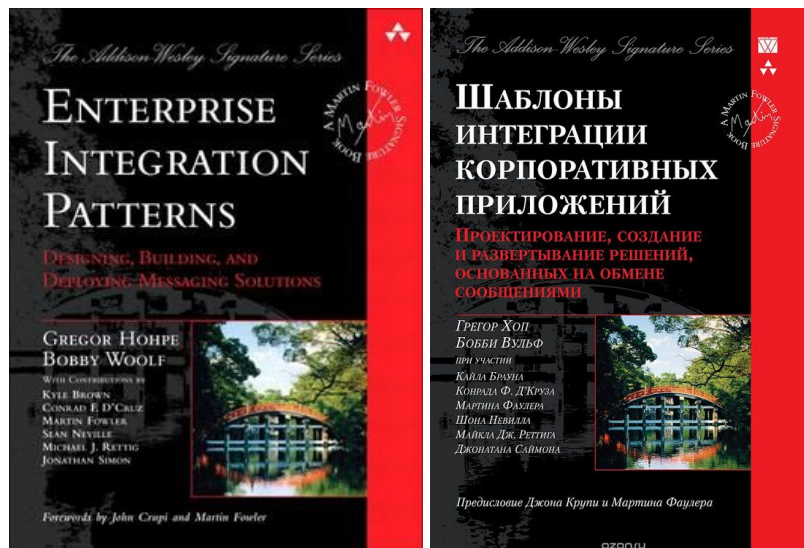
[Дополнительные материалы](#)

[Используемая литература](#)

Spring Integration

Введение

В области интеграции приложений, как и проектирования, тоже есть свои паттерны, известные как [Enterprise Integration Patterns](#), или шаблоны интеграции корпоративных приложений. Они описаны в одноименной книге Грегора Хопа и Бобби Вульфа.



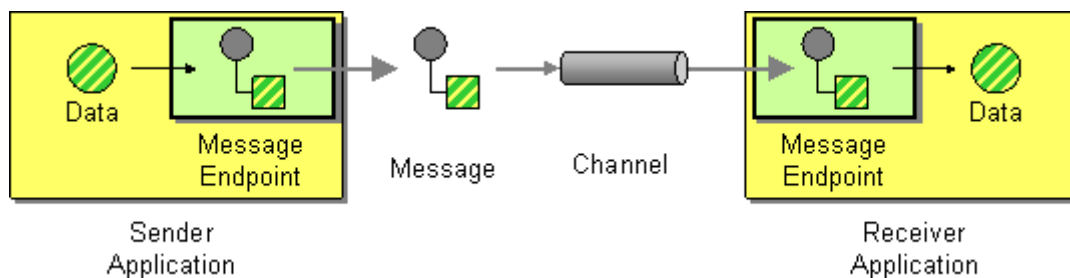
EIP

Данный набор описывает 65 паттернов, сгруппированных в 9 категорий. Они рассматривают интеграцию как передачу сообщений между узлами систем по связующим их каналам:



Источник: Грегор Хоп, Бобби Вульф. Шаблоны интеграции корпоративных приложений. — Вильямс, 2007.

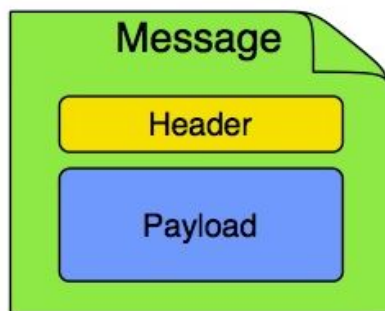
Здесь изображена минимальная передача информации между двумя узлами. Данная диаграмма наглядно показывает основную идею, на которой строятся EIP:



Данные одного приложения упаковываются в сообщение, поступают в отправную конечную точку, далее — в канал передачи между приложениями. Затем в конечной точке получателя исходные данные извлекаются из сообщения и передаются в приложение-получатель.

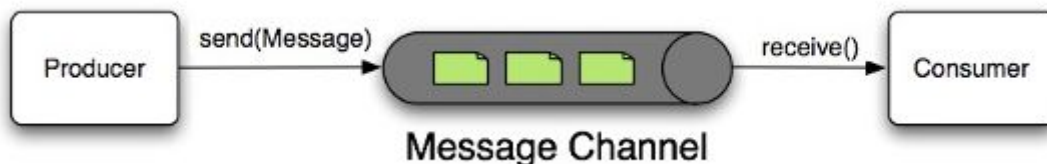
В информационном обмене участвуют три основных компонента:

- **узлы, или «конечные точки» (MessageEndpoint).** Их задача (ответственность) — обработка и хранение информации;
- **сообщения (Message)** — обертка над данными пользователя (Payload), включающая в себя также заголовок (Header), содержащий дополнительные поля. Разработчики могут также хранить в заголовках любые произвольные пары «ключ — значение».



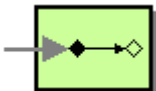
Сложилось так, что сообщения — наиболее гибкий способ интегрировать несколько разрозненных систем, так как это:

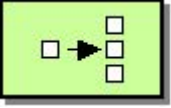
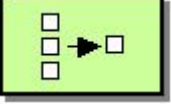
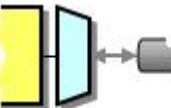
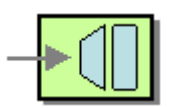
- почти полностью развязывает системы, участвующие в интеграции;
 - позволяет ее участникам быть полностью независимыми от различий, лежащих в основе протоколов, форматирования и других деталей реализации;
 - поощряет разработку и повторное использование компонентов, участвующих в интеграции.
- **каналы передачи сообщений между узлами (Message Channel).**



Паттерны интеграции описывают виды узлов и каналов. Собирая их воедино, мы можем быстро строить интеграционные решения вокруг существующих или новых приложений.

Некоторые паттерны интеграции

Символ	Паттерн	Описание
	Message Сообщение	Обертка над данными пользователя (Payload). Включает также заголовок (Header), содержащий дополнительные поля.
	Message Endpoint Конечная точка	Код конечной точки сообщения является обычным как для приложения, так и для клиентского API системы обмена сообщениями. Остальная часть приложения мало знает о форматах сообщений, каналах обмена ими или любых других элементах механики взаимодействия с прочими приложениями. Оно просто знает, что у него есть запрос или часть данных для отправки другому приложению, или ожидает их от последнего. Код конечной точки принимает эту команду или данные, превращает ее в сообщение и отправляет на конкретный канал обмена. Конечная точка получателя принимает сообщение, извлекает содержимое и предоставляет данные приложению.
	Message Channel Канал сообщений	Приложения связываются с помощью канала сообщений, где одно приложение отправляет информацию в канал, а другое — получает эту информацию из канала. В практическом применении интерес представляет не столько использование каналов, сколько выбор их типа, соответствующего задаче.
	Service activator Активатор	Активатор может быть односторонним (только запрос) или двухсторонним (запрос — ответ). Активатор обрабатывает все детали сообщения и вызывает службу, как любой другой клиент. Так что служба даже не знает, что она вызывается посредством обмена сообщениями.
	Pipes-and-Filters	Используйте архитектурный стиль Pipes and Filters , чтобы разделить большую задачу обработки на последовательность меньших независимых этапов (Filters).
	Router Маршрутизатор	Маршрутизатор получает сообщение из входного канала и передает его на один из выходных каналов в зависимости от набора условий (заголовка сообщения, payload, etc).
	Translator Транслятор	Преобразует данные (payload) из одного формата в другой.

	Splitter Сплиттер	Разбивает составное сообщение на ряд отдельных, каждое из которых содержит данные, относящиеся к одному элементу.
	Aggregator Агрегатор	Собирает и сохраняет отдельные сообщения до тех пор, пока не будет получен полный набор связанных сообщений. Затем агрегатор отправляет одно сообщение, собранное из отдельных.
	Adapter Адаптер	Адаптер может обращаться к API приложения или данным и отправлять сообщения в канал на их основе, а также получать сообщения и вызывать функции внутри приложения.
	Gateway Шлюз	Шлюз инкапсулирует код, специфичный для обмена сообщениями, и отделяет его от остальной части кода приложения. Только код шлюза знает о системе обмена сообщениями, остальная часть — нет. Ей эту бизнес-функцию предоставляет шлюз обмена сообщениями.

Ответственность интеграционного фреймворка

Помимо реализации основных интеграционных паттернов, интеграционный фреймворк несет следующие обязанности:

- **оркестрация** — организация мелкозернистых компонентов в логические потоки процессов;
- **манипуляция и преобразование** данных между форматами или их обогащение;
- **транспортировка** — обеспечение связи во множестве протоколов, включая HTTP, FTP, SMTP, JDBC и JMS. Также возможна поддержка корпоративных протоколов Tibco, SAP, Salesforce, PayPal, Facebook и других;
- **посредничество** — развязка объектов путем предоставления слоя связи, через который они взаимодействуют (паттерн «Медиатор»).

Известно несколько реализаций интеграционных паттернов, в том числе на Java:

- Apache Camel — <http://camel.apache.org>;
- Mule ESB — <http://www.mulesoft.com>;
- Spring Integration — <http://projects.spring.io/spring-integration>.

Spring Integration

Одна из реализаций паттернов интеграции корпоративных приложений — фреймворк Spring Integration, который построен поверх инфраструктуры Spring Framework и широко использует такие его возможности, как Dependency Injection и Spring Context.

Spring Integration позволяет связывать как независимые компоненты внутри одного приложения, так и различные приложения (в том числе и не JVM), предоставляет множество мощных компонентов, которые могут значительно повысить взаимосвязанность систем и процессов в рамках архитектуры

предприятия, обеспечивает легкую передачу сообщений в Spring-приложениях и поддерживает интеграцию с внешними системами. Основная цель Spring Integration — предоставить простую модель для построения корпоративных интеграционных решений, сохраняя проверяемость, поддерживаемость и управляемость кода.

Принципы Spring Integration

1. Компоненты должны быть слабо связаны для модульности и тестируемости.
2. Структура должна обеспечивать разделение проблем между бизнес-логикой и логикой интеграции.
3. Точки расширения должны быть абстрактными по своему характеру, но в пределах четко определенных границ — для повторного использования и переносимости.

Настройка. Зависимости. Пуск

Spring Integration легко подключается указанием Maven-зависимостей в проекте.

Вариант без Spring Boot:

```
<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-core</artifactId>
  <version>LATEST</version>
</dependency>
```

Вариант с использованием Spring Boot:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-integration</artifactId>
</dependency>
```

Кроме того, исходя из набора адаптеров могут быть нужны описания, предоставляющие их зависимости. Например, для HTTP потребуется следующая зависимость:

```
<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-http</artifactId>
</dependency>
```

Hello World

Простое приложение, раскрывающее суть происходящего. Здесь мы используем аннотации для описания конфигурации Spring Integration:

```
@SpringBootApplication
@IntegrationComponentScan
public class Application {
    public static void main(String[] args) {
        ConfigurableApplicationContext context =
        SpringApplication.run(Application.class, args);
        Message<String> message = MessageBuilder
            .withPayload("body")
            .setHeader("header", "value")
            .build();

        MessageChannel channel = (DirectChannel) context.getBean("channel_no5");
        channel.send(message);
    }

    @ServiceActivator(inputChannel = "channel_no5")
    public void foo(@Payload String payload, @Headers Map<String, Object>
headerMap) {
        headerMap.forEach((s, o) -> System.out.printf("%s:%s\n", s, o));
        System.out.println(payload);
    }
}
```

Вывод в консоль:

```
header:value
id:742acd41-527d-a4fa-018c-a8e5f86e40f9
timestamp:1505949152561
Hello World!
```

Разберем пример построчно:

```
@IntegrationComponentScan
```

Эта аннотация — аналогично **@ComponentScan** — сканирует классы в поисках аннотированных элементов (классов и методов), описывающих конфигурацию Spring Integration.

```
Message<String> message = MessageBuilder
    .withPayload("body")
    .setHeader("header", "value")
    .build();
```

Здесь мы используем **MessageBuilder** для построения сообщения:

```
MessageChannel channel = (DirectChannel) context.getBean("channel_no5");
channel.send(message);
```

Получаем канал из контекста и отправляем сообщение:

```
@ServiceActivator(inputChannel = "channel_no5")
public void foo(@Payload String payload, @Headers Map<String, Object> headerMap)
{
    headerMap.forEach((s, o) -> System.out.printf("%s:%s\n", s, o));
    System.out.println(payload);
}
```

Описываем активатор и функцию, которую он вызывает.

В аннотации активатора ссылаемся на нигде не описанный канал. По умолчанию, описанным в **/META-INF/spring.integration.default.properties** в **spring-integration-core**, входные каналы создаются автоматически, если не описаны явно.

Теперь рассмотрим, как этот пример будет выглядеть с использованием **xml**:

```
@SpringBootApplication
@ImportResource("integration.xml")
public class Application {
    public static void main(String[] args) {
        ConfigurableApplicationContext context=SpringApplication.
            run(Application.class,args);

        Message<String> message = MessageBuilder
            .withPayload("Hello World!")
            .setHeader("header", "value")
            .build();

        MessageChannel channel = (DirectChannel)
            context.getBean("channel_no5");

        channel.send(message);
    }

    public void foo(String payload, Map<String, Object> headerMap) {
        headerMap.forEach((s, o) -> System.out.printf("%s:%s\n", s, o));
        System.out.println(payload);
    }
}
```

Файл **integration.xml**:

```
<int:channel id="channel_no5"/>

<int:service-activator id="someService" ref="application" method="foo"
    input-channel="channel_no5"/>
```

Отличия:

```
@ImportResource("integration.xml")
```

Здесь вместо сканирования классов мы импортируем xml-файл описания конфигурации — **integration.xml**.

В файле конфигурации описаны два компонента: активатор и его входной канал. Сравнивая два метода, можно заметить, что xml-описание более наглядно, а описание аннотациями может показаться более быстрым и удобным. Эти стили можно комбинировать.

Мы рассмотрели случай интеграции компонентов внутри одного приложения. Рассмотрим теперь более сложный пример.

Интеграция разрозненных приложений

В этом примере видим HTTP-сервис, принимающий запросы. Затем это же приложение обращается к сервису и получает ответ.

```
@SpringBootApplication
@ImportResource("integration.xml")
public class ServerApplication {
    public static void main(String[] args) {
        ConfigurableApplicationContext context =
SpringApplication.run(ServerApplication.class, args);
        String response = context.getBean(RequestGateway.class).echo("hello");
        System.out.println(response);
        context.close();
    }

    // программная реализация бизнес-логики
    public String controller(String payload) {
        return String.
            format("Re: '%s' from over side via controller", payload);
    }

    // описание интерфейса для слоя бизнес-логики,
    // интеграция скрыта слоем описанным в .xml
    public interface RequestGateway {
        String echo(String request);
    }
}
```

Файл **integration.xml**:

```
<!--server side-->
<int-http:inbound-gateway request-channel="receiveChannel"
                        path="/api"
                        supported-methods="GET, POST"/>
<int:channel id="receiveChannel"/>
<!--// SpEL-реализация бизнес-логики -->
<!--<int:service-activator input-channel="receiveChannel"
expression="'Re: ' + payload + ' from the other side'"/>-->
<!--// программная реализация бизнес-логики-->
<int:service-activator input-channel="receiveChannel"
ref="serverApplication" method="controller"/>
```

```
<!--client side-->
<int:gateway id="requestGateway"
service-interface="com.example.server.ServerApplication$RequestGateway"
default-request-channel="requestChannel"/>

<int:channel id="requestChannel"/>

<int-http:outbound-gateway request-channel="requestChannel"
                        url="http://localhost/api"
                        http-method="POST"
                        expected-response-type="java.lang.String"/>
```

Иногда активатор комфортнее описать аннотацией:

```
// вариация с аннотацией
@ServiceActivator(inputChannel="receiveChannel")
public String controller(String payload) {
    return String.format("Re: '%s' from over side via controller",
                        payload);
}
```

Компоненты Spring Integration

Сообщение (Message)

Интерфейс **org.springframework.integration.Message** определяет Spring-сообщение. Блок передачи данных в контексте **Spring Integration**:

```
public interface Message<T> {
    T getPayload();
    MessageHeaders getHeaders();
}
```

Он определяет аксессоров двух ключевых элементов:

- заголовков сообщений — по существу, контейнера «ключ — значение», который может использоваться для передачи метаданных, как это определено в классе `org.springframework.integration.MessageHeaders`;
- полезной нагрузки сообщения — передаваемых данных, имеющих значение.

Канал (Channel)

Поскольку **DirectChannel** — самый простой вариант, который не добавляет накладных расходов, которые необходимы для планирования и управления потоками **poller**, этот тип канала указан по умолчанию в Spring Integration. Общая идея состоит в том, чтобы определить каналы для приложения, затем рассмотреть, какой из них должен обеспечивать буферизацию или ввод данных с дроссельной заслонкой, и, наконец, модифицировать те, которые будут использоваться на основе **PollableChannels**. Аналогично, если канал транслирует сообщения, он не должен быть **DirectChannel**, а, скорее, **PublishSubscribeChannel**.

Шлюз (Gateway)

Основная цель шлюза — скрыть API сообщений, предоставляемый Spring Integration. Это позволяет бизнес-логике приложения не знать об API-интеграции Spring и использовать шлюз. При этом код взаимодействует только с простым интерфейсом. Его реализация — задача Spring Integration, которую он решает на основании описания конфигурации (**xml** или аннотаций).

Заключение

Фреймворк Spring Integration предназначен для интеграции как существующих, так и разрабатываемых приложений. При работе с ним в меньшей степени приходится писать код или реализовывать ранее определенные фреймворком интерфейсы, и в большей — описывать интерфейсы и задавать правила. Фреймворк, следуя указанным предписаниям, оставляет реализацию сложного механизма интеграции «под капотом» итогового решения.

Практическое задание

1. Подумайте над вопросом - есть ли возможность использовать Spring Integration в рамках курсового проекта. Если да, то реализуйте вашу идею. Если нет - доработайте один из блоков разрабатываемого интернет-магазина.

Дополнительные материалы

1. Грегор Хоп, Бобби Вульф. Шаблоны интеграции корпоративных приложений. — Вильямс, 2007.
2. [Введение от одного из разработчиков.](#)
3. [Сжатый конспект о EIP.](#)

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [Официальная документация — Spring Expression Language \(SpEL\)](#).
2. [Официальная документация — Integrating Data](#).
3. Грегор Хоп, Бобби Вульф. Шаблоны интеграции корпоративных приложений. — Вильямс, 2007.
4. [Spring Integration Fundamentals](#).
5. [Spring Integration Java DSL: Line by line tutorial](#).