# Оглавление

# Вступление

Реактивные системы всегда откликаются, что и требуется большинству предприятий. Разработка таких систем – сложная задача, требующая глубокого понимания предмета. К счастью, разработчики Spring Framework создали новую, реактивную версию проекта.

Прочитав книгу *«Практика реактивного программирования в Spring 5»*, вы познакомитесь с увлекательным процессом разработки реактивных систем на основе фреймворка Spring Framework 5.

Эта книга начинается со знакомства с основами реактивного программирования в Spring. Вы получите представление о возможностях фреймворка и узнаете об основах реактивного программирования. Далее вашему вниманию будут представлены методы реактивного программирования, их использование для организации взаимодействий с базами данных и между серверами. Все эти задачи будут продемонстрированы на примере реального проекта, что позволит вам попрактиковать полученные навыки.

А теперь просим всех на борт реактивной революции в Spring 5!

## Кому адресована эта книга

Эта книга адресована разработчикам на Java, использующим фреймворк Spring для своих приложений и желающих получить возможность создавать надежные и реактивные приложения, которые можно масштабировать в облаке. Предполагается базовое знание распределенных систем и асинхронного программирования.

## Содержание книги

Глава 1, *В чем выгоды Reactive Spring?*, описывает случаи, для которых подходит реактивность. Здесь вы узнаете, чем реактивное решение лучше проактивного, увидите несколько примеров кода, демонстрирующих разные способы связи между серверами, а также познакомитесь с современными потребностями и требованиями бизнеса к современному фреймворку Spring Framework.

Глава 2, *Реактивное программирование в Spring* – основные понятия, раскрывает потенциал реактивного программирования и его основные понятия на примерах кода. Затем демонстрирует мощь реактивного, асинхронного, неблокирующего программирования в Spring Framework на примерах кода и его применение на практике. Здесь вы познакомитесь с моделью издатель/подписчик, с реактивными событиями Flow и особенностями применения этих методов на практике.

Глава 3, *Reactive Streams* – новый стандарт потоков, описывает проблемы реактивных расширений Reactive Extensions. На примерах кода раскрывается природа проблем и исследуются разные подходы к их решению. Эта глава также знакомит со спецификацией Reactive Streams, которая вводит новые компоненты в хорошо известную модель издатель/подписчик.

Глава 4, *Project Reactor* – основа реактивных приложений, рассматривает реализацию реактивной библиотеки, полностью реализующей спецификацию Reactive Streams. Сначала в этой главе сначала описываются преимущества Reactor, а затем рассматриваются причины, побудившие разработчиков Spring заняться созданием нового решения. Кроме того, эта глава охватывает основы использования этой впечатляющей библиотеки – здесь вы получите представление о Mono и Flux, а также об особенностях применения реактивных типов.

Глава 5, *Добавление реактивности с помощью Spring Boot 2*, знакомит с реактивными модулями, имеющимися в Spring, которые пригодятся при разработке реактивных приложений. Здесь вы узнаете, как начать использовать модули и как Spring Boot 2 помогает разработчикам быстро настраивать приложения.

Глава 6, *Неблокирующие и асинхронные взаимодействия с WebFlux*, охватывает один из основных модулей, Spring WebFlux, являющийся важным инструментом для организации асинхронных и неблокирующих взаимодействий с пользователем и внешними службами. Эта глава дает обзор преимуществ этого модуля и сравнивает его с Spring MVC.

Глава 7, *Реактивный доступ к базам данных*, описывает модель реактивного программирования доступа к данным на основе Spring 5. Основное внимание в этой главе уделяется поддержке реактивности в модулях Spring Data и исследуются средства, входящие в состав Spring 5, Reactive Streams и Project Reactor. Здесь вы увидите код, демонстрирующий реактивный подход к взаимодействиям с разными базами данных, такими как SQL и NoSQL.

Глава 8, *Масштабирование с Cloud Streams*, познакомит с реактивными возможностями Spring Cloud Streams. Перед началом знакомства с новыми потрясающими возможностями модуля вашему вниманию будет предложен обзор проблем и недостатков, с которыми можно столкнуться при организации масштабирования на разных серверах. Эта глава раскроет вам возможности решения Spring Cloud и продемонстрирует его реализацию на примерах кода с соответствующей конфигурацией Spring Boot 2.

Глава 9, *Тестирование реактивных приложений*, охватывает основы тестирования реактивных приложений. Эта глава знакомит с модулями Spring 5 Test и Project Reactor Test. Здесь вы увидите, как управлять частотой следования событий, перемещать временные интервалы, расширять пулы потоков выполнения, проверять результаты и переданные сообщения.

Глава 10, *И, наконец, выпуск!*, содержит подробное пошаговое руководство по развертыванию и мониторингу решения. Здесь вы увидите, как осуществляется мониторинг реактивных микрослужб с использованием модулей Spring 5. Здесь также рассматриваются инструменты, которые пригодятся для сбора данных мониторинга и их отображения.

# Что потребуется для работы с книгой

Разработка реактивных систем – сложная задача, требующая глубокого понимания предметной области, поэтому вам обязательно потребуется знакомство с распределенными системами и асинхронным программированием.

# Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф в разделе *«Читателям – Файлы к книгам»*.

Кроме того, примеры кода к книге доступны на сайте GitHub, по адресу: https://github.com/PacktPublishing/Hands-On-Reactive-Programming-in-Spring-5.

# Скачивание цветных иллюстраций

Мы также подготовили документ PDF, содержащий цветные иллюстрации со скриншотами/диаграммами, используемыми в книге. Его можно загрузить по адресу: https://www.packtpub.com/sites/default/files/downloads/9781787284951_ColorImages.pdf.

# Типографские соглашения

В этой книге используется несколько разных стилей оформления текста, с целью обеспечить визуальное отличие информации разных типов. Ниже приводится несколько примеров таких стилей оформления и краткое описание их назначения.

Программный код в тексте, имена таблиц баз данных, имена папок, имена файлов, расширения файлов, пути в файловой системе, адреса URL, ввод пользователя и ссылки в Twitter оформляются, как показано в следующем предложении: «При первом обращении будет вызван обработчик `onSubscribe()`, который сохранит подписку Subscription локально и известит издателя `Publisher` о готовности подписчика принимать события через метод `request()`.»

Блоки программного кода оформляются так:

```
@Override
public long maxElementsFromPublisher() {
   return 1;
}
```

Ввод или вывод в командной строке будет оформляться так:

```
./gradlew clean build
```

Новые термины и важные определения будут выделяться в обычном тексте жирным.

Внимание. Так будут оформляться предупреждения и важные примечания.


Совет. Так будут оформляться советы и рекомендации.

# Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или может быть не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте `www.dmkpress.com`, зайдя на страницу книги и оставить комментарий в разделе *«Отзывы и рецензии»*. Также можно послать письмо главному редактору по адресу `dmkpress@gmail.com`, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу `http://dmkpress.com/authors/publish_book/` или напишите в издательство по адресу `dmkpress@gmail.com`.

# Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки всё равно случаются. Если вы найдёте ошибку в одной из наших книг – возможно, ошибку в тексте или в коде – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдёте какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу `dmkpress@gmail.com`, и мы исправим это в следующих тиражах.

## Нарушение авторских прав

Пиратство в Интернете по-прежнему остается насущной проблемой. Издательства ДМК Пресс и Packt очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в Интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли принять меры.

Пожалуйста, свяжитесь с нами по адресу электронной почты `dmkpress@gmail.com` со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, и помогающую нам предоставлять вам качественные материалы.

# 1

# Причины выбора Spring

В этой главе мы объясним понятие **реактивности** и покажем, почему реактивные подходы лучше традиционных. Для этого мы рассмотрим примеры, когда традиционные подходы терпят неудачу. В дополнение к этому мы исследуем фундаментальные принципы построения надежных систем, которые в большинстве своем являются **реактивными системами**. Мы также познакомимся с основными причинами, объясняющими необходимость использования механизмов рассылки сообщений для организации взаимодействий между распределенными серверами, и покажем случаи, в которые реактивность вписывается как нельзя лучше. Затем распространим приемы реактивного программирования для создания модульной реактивной системы. Мы также обсудим причины, по которым команда разработчиков Spring Framework решила включить реактивный подход в ядро фреймворка **Spring Framework 5**. Прочитав эту главу, вы поймете важность реактивности, и почему стоит перенести свои проекты в реактивный мир.

В этой главе рассматриваются следующие темы:

- основные преимущества реактивности;
- основные принципы создания реактивных систем;
- случаи, когда реактивный дизайн подходит лучше всего;
- приемы программирования реактивных систем;
- причины включения поддержки реактивности в фреймворк Spring Framework.

## Основные преимущества реактивности

В наши дни стало модным использовать слово **реактивный** – оно такое волнующее и непонятное. Однако, стоит ли продолжать популяризовать реактивность, даже после того, как это слово заняло почетное место на разнообразных международных конференциях? Если погуглить по слову «реактивный», можно обнаружить, что оно чаще всего встречается в паре со словом «программирование», и вместе они обозначают модель программирования. Однако это не единственный смысл реактивности. За этим словом стоят фундаментальные принципы проектирования, направленные на создание надежных систем. Чтобы понять

ценность реактивности, как важнейшего принципа проектирования, представим, что мы развиваем малое предприятие.

Допустим, что наше малое предприятие – это интернет-магазин, продающий современные товары по привлекательной цене. Как и в большинстве проектов в этой области, мы наняли разработчиков программного обеспечения, которые помогут нам справиться со всеми возникающими проблемами. Мы решили выбрать традиционный подход к разработке и в ходе нескольких циклов создали магазин.

Каждый час наш онлайн-магазин обычно посещает тысяча пользователей. Чтобы справиться с этим потоком, мы купили современный компьютер и запустили на нем веб-сервер Tomcat, настроив пул с 500 предварительно созданными потоками выполнения. Среднее время отклика на большинство запросов составило около 250 миллисекунд. Простейшие расчеты показывают, что такая конфигурация позволит нам обслуживать до 2 000 запросов в секунду. Согласно статистике, вышеупомянутое число пользователей в среднем производит около 1 000 запросов в секунду. Следовательно, текущей производительности системы вполне достаточно для обслуживания средней нагрузки.

Итак, мы настроили приложение с неплохим запасом по производительности. Более того, наш интернет-магазин работал вполне стабильно до… последней пятницы ноября, то есть до Черной пятницы.

Черная пятница – важный день и для покупателей, и для продавцов. Покупатели получают возможность купить товар со скидкой, а продавцы – разрекламировать товары и получить дополнительную прибыль. Однако этот день характеризуется необычным наплывом клиентов и это может стать причиной сбоев в работе интернет-магазина.

И, конечно же, мы потерпели сокрушительное фиаско! В какой-то момент нагрузка превысила все наши ожидания. В пуле не оказалось свободных потоков выполнения для обработки запросов. Сервер резервного копирования не справился с таким наплывом покупателей и, в конце концов, время отклика возросло, и периодически наблюдались сбои. В этот момент мы начали терять некоторые запросы, в результате наши клиенты испытали чувство неудовлетворенности и переметнулись к конкурентам.

В итоге мы потеряли большое число потенциальных клиентов, остались без дополнительной прибыли, а рейтинг магазина рухнул. Все это стало результатом ухудшения времени отклика в условиях возросшей нагрузки.

Но не волнуйтесь, эта проблема далеко не нова. Даже такие гиганты как Amazon и Walmart сталкивались с этой проблемой и давно нашли пути ее решения. Но не будем спешить и пройдем тот же путь, что прошли наши предшественники, чтобы понять основные принципы проектирования надежных систем и затем дать им общее определение.

Узнать больше о проблемах, возникающих у гигантов можно по адресам:

- Amazon.com, проблема с отключениями (`https://www.cnet.com/news/amazon-com-hit-with-outages/`);
- Amazon.com, как отказы приводили к потерям до 66 240 долларов в минуту (`https://www.forbes.com/sites/kellyclay/2013/08/19/amazon-com-goes-down-loses-66240-per-minute/#3fd8db37495c`);
- Walmart, провал в Черную пятницу: веб-сайт не справился с нагрузкой (`https://techcrunch.com/2011/11/25/walmart-black-friday/`).

Теперь главный вопрос, ответ на который мы должны найти: как на это реагировать? Из примера, описанного выше, очевидно вытекает, что приложение должно как-то реагировать на изменения – оно должно реагировать на изменение нагрузки и на изменение доступности внешних служб. Иначе говоря, приложение должно активно реагировать на любые изменения, которые могут сказаться на доступности системы и ее способности откликаться на запросы пользователей.

Один из путей к главной цели лежит через увеличение **эластичности**. Под этим термином понимается способность сохранять отзывчивость при различной рабочей нагрузке, то есть пропускная способность системы должна автоматически увеличиваться с увеличением числа пользователей и уменьшаться со снижением спроса. Эта особенность улучшает отзывчивость системы, потому что в любой момент пропускная способность системы может увеличиться и обеспечить приемлемое среднее время задержки.

*Время задержки* – важная характеристика отзывчивости. В отсутствие должной эластичности, рост нагрузки увеличит время задержки, которое напрямую влияет на отзывчивость системы.

Например, увеличить пропускную способность системы можно, увеличивая вычислительные мощности или запуская дополнительные экземпляры. В результате возрастет отзывчивость системы. С другой стороны, если поток пользователей уменьшился, система в ответ должна уменьшить потребление ресурсов, уменьшив тем самым накладные расходы. Добиться желаемой эластичности можно путем масштабирования, горизонтальной или вертикальной. Однако, масштабирование распределенной системы – сложная задача и обычно ограничивается узкими местами или точками синхронизации в системе. С теоретической и практической точек зрения эти проблемы объясняются законом Амдала и универсальной моделью масштабирования Гюнтера Нейла. Мы обсудим их позже, в `главе 6` *Неблокирующие и асинхронные взаимодействия с WebFlux*.

Здесь под накладными расходами понимается стоимость развертывания дополнительных экземпляров в облаке или дополнительное потребление электроэнергии в случае установки дополнительных компьютеров.

Однако, построение масштабируемой распределенной системы без возможности оставаться отзывчивой, независимо от отказов, – сложная задача. Представьте ситуацию, когда какая-то часть системы вдруг оказывается недоступной. Допустим, что отказала внешняя платежная система. В этой ситуации любые попытки пользователя произвести оплату будут терпеть неудачу. Это нарушит отзывчивость системы, что в некоторых случаях совершенно неприемлемо. Например, если пользователи не смогут с легкостью совершать покупки, они наверняка сбегут в интернет-магазин конкурента. Чтобы обеспечить качественное обслуживание клиентов, мы должны позаботиться об отзывчивости системы. Критерием приемлемости для системы является способность оставаться отзывчивой в случае отказов или, говоря другими словами, оставаться устойчивой. Этого можно добиться путем изоляции функциональных компонентов системы, помогающей изолировать внутренние сбои и обеспечить независимость. Вернемся к интернет-магазину Amazon. В Amazon имеется большое число разных функциональных компонентов, отвечающих, например, а вывод списка заказов, оплату, рекламу, прием отзывов от пользователей и многие другие. Например, в случае сбоя в платежной системе мы можем принять заказ пользователя и запланировать автоматическое повторение запроса, защитив пользователя от сбоев. Другим примером может служить изоляция от службы приема отзывов пользователей. Если служба приема отзывов окажется недоступной, это никак не должно сказаться на возможности оформлять заказы и делать покупки.

Также важно отметить, что эластичность и устойчивость тесно связаны между собой, и получить по-настоящему отзывчивую систему, можно только уделив должное внимание обеим. Масштабируемость позволяет иметь несколько реплик компонента, чтобы в случае сбоя в одной можно было быстро переключиться на другую и таким способом обеспечить бесперебойную работу системы.

Более подробное описание терминологии можно найти по ссылкам:

- эластичность (`https://www.reactivemanifesto.org/ru/glossary#Elasticity`);
- отказ (`https://www.reactivemanifesto.org/ru/glossary#Failure`);
- изоляция (`https://www.reactivemanifesto.org/ru/glossary#Isolation`);
- компонент (`https://www.reactivemanifesto.org/ru/glossary#Component`).

## Взаимодействия на основе обмена сообщениями

Единственное, что пока остается неясным – как связываются компоненты распределенной системы и одновременно остаются независимыми, изолированными и простыми для масштабирования. Рассмотрим связь между компонентами по протоколу HTTP. Следующий фрагмент кода, реализующий взаимодействия по HTTP в Spring Framework 4, наглядно демонстрирует эту идею:

```
@RequestMapping("/resource")                                    // (1)
public Object processRequest() {
   RestTemplate template = new RestTemplate();                  // (2)

   ExamplesCollection result = template.getForObject(           // (3)
      "http://example.com/api/resource2",                       //
      ExamplesCollection.class                                  //
   );                                                           //

   ...                                                          // (4)

   processResultFurther(result);                                // (5)
}
```

Этот код выполняет следующие действия:

- Объявляет обработчик запросов с использованием аннотации `@RequestMapping`.
- Создает экземпляр `RestTemplate` – самого популярного веб-клиента в Spring Framework 4 для организации взаимодействий типа запрос/ответ между службами.
- Конструирует и посылает запрос. Здесь, используя `RestTemplate`, мы конструируем HTTP-запрос и тут же посылаем его. Обратите внимание, что ответ автоматически отображается в Java-объект и возвращается как результат. Тип ответа определяется вторым параметром метода `getForObject`. Кроме того, префикс `getXxxXxxxxx` определяет HTTP-метод, в данном случае `GET`.
- Здесь выполняются дополнительные операции, которые были опущены в этом примере для краткости.
- Выполняется следующий этап обработки ответа.

В предыдущем примере мы определили обработчик запросов, который вызывается в ответ на получение запроса от пользователя. Этот обработчик, в свою очередь, посылает дополнительный HTTP-запрос внешней службе, а затем передаем его на следующий этап обработки. Несмотря на то что логика работы этого кода выглядит знакомо и понятно, в нем есть некоторые недостатки. Чтобы понять, что не так в этом примере, рассмотрим, как протекают события во времени:

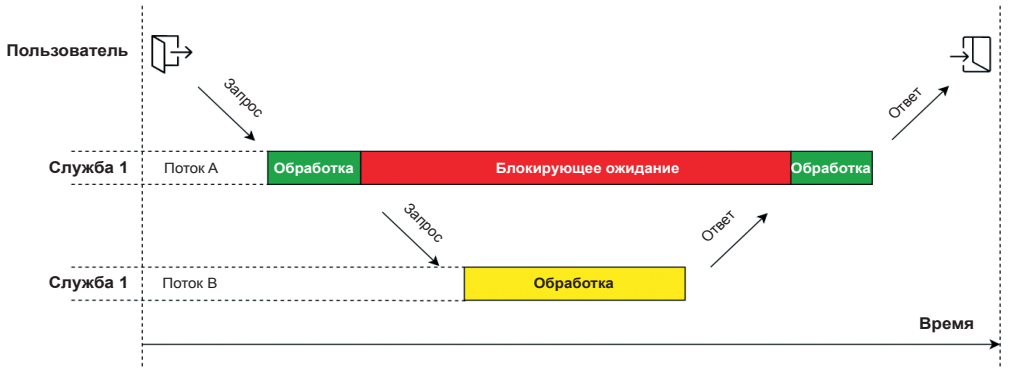


Рис. 1.1. Взаимодействия компонента во времени

Эта диаграмма отражает фактическое поведение кода в примере выше. Как можно заметить, процессор занят фактической работой только часть времени, тогда как другую часть времени поток проводит в ожидании завершения операции ввода/вывода и не может использоваться ля обслуживания других запросов.

В некоторых языках программирования, таких как C#, Go и Kotlin, этот же код может действовать в неблокирующем режиме при использовании зеленых потоков выполнения. Однако в Java такая возможность пока отсутствует. По этой причине поток выполнения фактически будет заблокирован.

С другой стороны, в мире Java имеются пулы потоков выполнения, способные запускать дополнительные потоки. Однако, при работе под высокой нагрузкой этот механизм крайне неэффективен и не может использоваться для обработки новых заданий ввода/вывода. Мы еще вернемся к этой проблеме ниже, а также исследуем ее в `главе 6` *Неблокирующие и асинхронные взаимодействия с WebFlux*.

Очевидно, что для более эффективного использования ресурсов при выполнении большого количества операций ввода/вывода необходимо задействовать модель асинхронных и неблокирующих взаимодействий. В реальной жизни такой способ взаимодействий организован как обмен сообщениями. Получив сообщение (SMS или по электронной почте), все свое время мы уделяем его чтению и составлению ответа. Но мы обычно не ждем ответа и продолжаем заниматься другими делами. Безусловно, при таком подходе мы работаем более эффективно и более рационально используем свое время. Взгляните на рис. 1.2.

Более подробное описание терминологии можно найти по ссылкам:

- неблокирующие операции (`https://www.reactivemanifesto.org/ru/glossary#Non-Blocking`);
- ресурс (`https://www.reactivemanifesto.org/ru/glossary#Resource`).

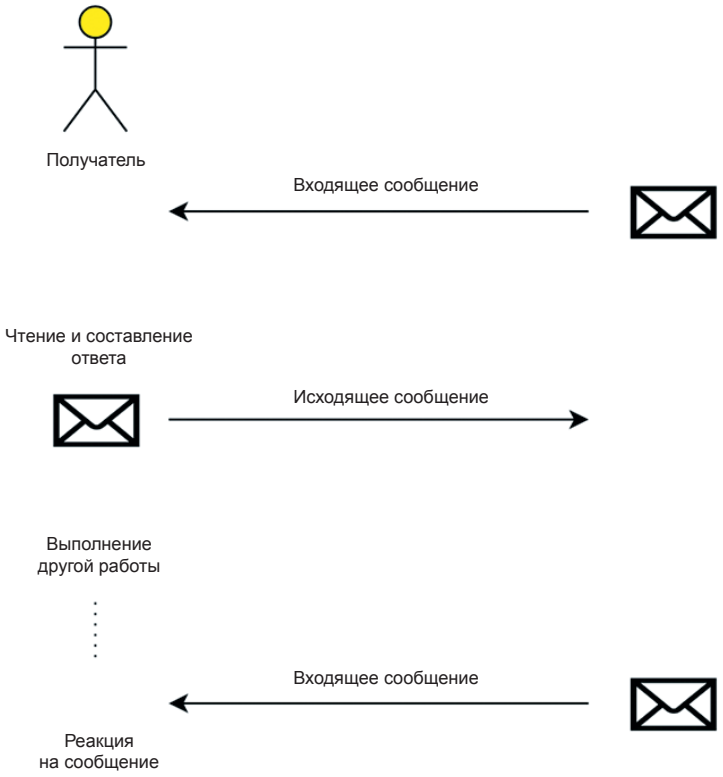


Рис. 1.2. Неблокирующий обмен сообщениями

В целом, чтобы добиться эффективного использования ресурсов при взаимодействии служб в распределенной системе, мы должны взять на вооружение принцип взаимодействий на основе обмена сообщениями. В общих чертах взаимодействие между службами можно описать так: каждый элемент, ожидающий поступления сообщений, реагирует на них при получении, а остальное время пребывает в спящем состоянии, и наоборот, компоненты должны иметь возможность посылать сообщения в неблокирующем режиме. Такой подход к взаимодействиям улучшает масштабируемость системы за счет поддержки независимости от местоположения. Отправляя электронное письмо, мы должны лишь правильно написать электронный адрес получателя, а хлопоты по его доставке на одно из устройств пользователя возьмет на себя почтовый сервер. Это освобождает нас от хлопот в выборе устройства и дает получателям возможность использовать столько устройств, сколько они пожелают. Кроме того, этот подход повышает отказоустойчивость, поскольку отказ одного из устройств не помешает получателю прочитать свою почту с помощью другого устройства.

Один из способов реализации взаимодействий на основе сообщений – использование **брокера сообщений**. В этом случае, осуществляя мониторинг очереди

сообщений, система может управлять эластичностью и нагрузкой. Кроме того, обмен сообщениями делает поток управления более ясным и упрощает общий дизайн. Мы не будем вдаваться в подробности в этой главе, потому что наиболее популярные приемы организации взаимодействий на основе сообщений рассматриваются в `главе 8` *Масштабирование с Cloud Streams*.

Фраза **пребывает в спящем состоянии** взята из следующего оригинального документа, который стремится подчеркнуть преимущества взаимодействий на основе обмена сообщениями: `https://www.reactivemanifesto.org/ru/glossary#Message-Driven`.

Предыдущие утверждения определяют основополагающие принципы построения реактивных систем, как показано на рис. 1.3.

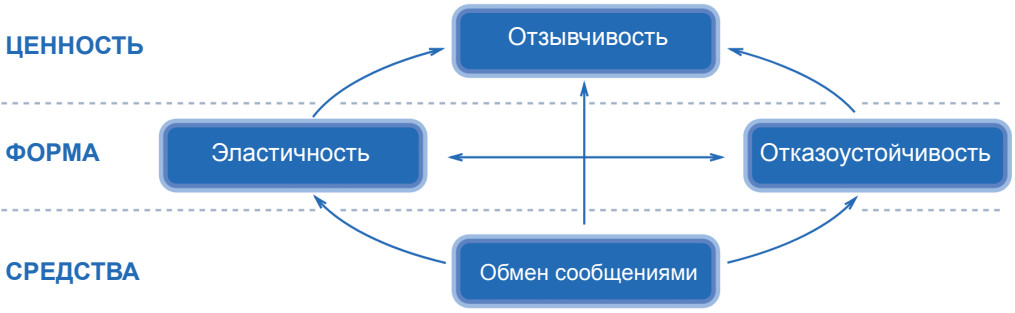


Рис. 1.3. Манифест реактивных систем

Как можно заметить на рис. 1.3, главной ценностью распределенной системы для любого бизнеса, является отзывчивость. Методы достижения отзывчивости имеют форму эластичности и отказоустойчивость. Наконец, одним из фундаментальных средств обеспечения отзывчивости, эластичности и отказоустойчивости является организация взаимодействий посредством сообщений. Кроме того, системы построенные на основе этих принципов, просты в сопровождении и легко расширяются, потому что все компоненты системы изолированы и независимы.


Мы не будем подробно обсуждать все понятия, перечисляемые в манифесте реактивных систем, но настоятельно рекомендуем посетить глоссарий по адресу: `https://www.reactivemanifesto.org/ru/glossary/`.

Все эти понятия не новы и определены в манифесте реактивных систем, который одновременно является глоссарием, описывающим понятия реактивных систем. Этот манифест был создан с целью гарантировать одинаковое понимание традиционных идей разработчиками и предпринимателями. Отметим особо, что реактивные системы и манифест реактивных систем – это архитектурные понятия и поэтому могут применяться и к большим распределенным решениям, и к малым приложениям, выполняющимся на единственном узле.

Важность манифеста реактивных систем (https://www.reactivemanifesto.org/ru) объясняется Йонасом Бонером (Jonas Bonér), основателя и директора компании Lightbend, по адресу: https://www.lightbend.com/blog/why_do_we_need_a_reactive_manifesto%3F.

# Примеры использования реактивности

В предыдущем разделе мы узнали о важности реактивности и основополагающих принципах реактивных систем, и почему организация взаимодействий посредством сообщений является важнейшей составляющей реактивной экосистемы. Но, чтобы закрепить новые знания, необходимо познакомиться с примерами использования реактивности. Прежде всего, под понятием «реактивная система» подразумевается архитектура, которая может применяться где угодно – для реализации простых веб-сайтов, больших корпоративных решений и даже систем потоковой передачи или обработки больших данных. Но начнем с самого простого – рассмотрим пример интернет-магазина, который мы в первый раз увидели в предыдущем разделе. В этом разделе мы рассмотрим возможные усовершенствования и изменения в конструкции, которые могут помочь в достижении реактивности. На рис. 1.4 показана общая архитектура предлагаемого решения.

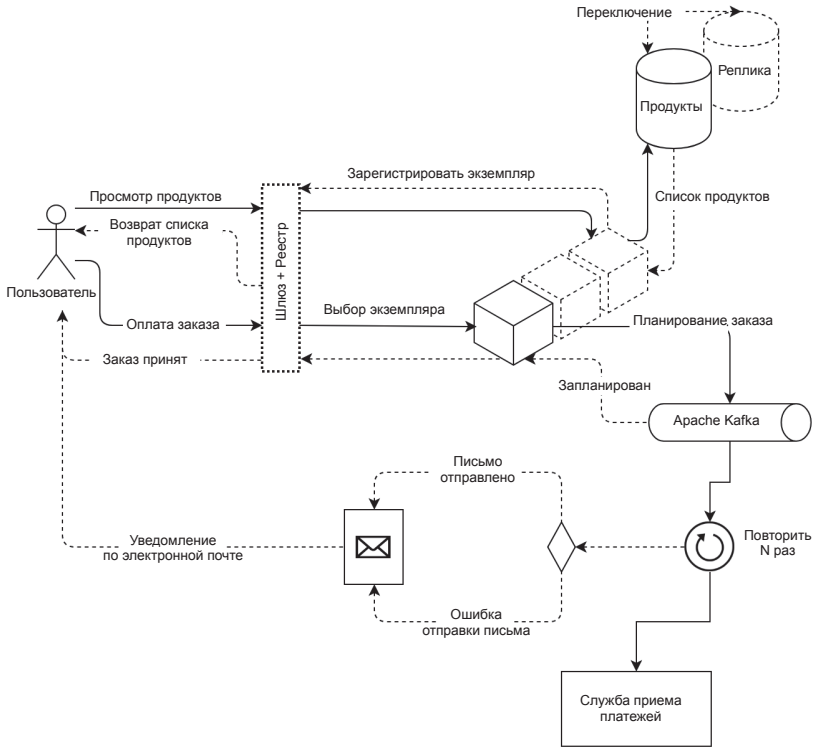


Рис. 1.4. Пример архитектуры интернет-магазина

Диаграмма на рис. 1.4 расширяет список полезных приемов реализации реактивных систем. Здесь мы немного усовершенствовали наш маленький интернет-магазин, применив современный подход на основе микрослужб. В данном случае для обеспечения независимости от местоположения используется шаблон «Шлюз API». Он обеспечивает идентификацию конкретного ресурса, не зная, какие конкретные службы отвечают за обработку тех или иных запросов.

Это, однако, означает, что клиент по меньшей мере должен знать название ресурса. Получив название службы в составе URI запроса, шлюз API может определить конкретный адрес для дальнейшей передачи запроса, обратившись к службе реестра.

Ответственность за поддержание в актуальном состоянии информации о доступных службах возлагается на службу реестра. Следует отметить, что службы шлюза и реестра в предыдущем примере действуют на одной машине, что может быть полезно для небольших распределенных систем. Кроме того, высокая отзывчивость системы достигается применением репликации к службе. С другой стороны, отказоустойчивость обеспечивается организацией взаимодействий посредством обмена сообщениями с использованием Apache Kafka и независимого прокси для доступа к платежной системе (обозначен на *рис. 1.4* точкой с подписью **Повторить N раз**), который отвечает за повторение попыток провести платеж в случае недоступности внешней системы. Также для отказоустойчивости использован прием репликации базы данных на случай, если одна из реплик выйдет из строя. Для достижения высокой отзывчивости мы тут же сообщаем о приеме заказа и асинхронно обрабатываем его и посылаем пользовательскую информацию службе платежей. Окончательное уведомление доставляется позже, по одному из поддерживаемых каналов, например, по электронной почте. Наконец, этот пример изображает только одну часть системы – в действительности общая диаграмма может быть намного шире и включать более конкретные методы реализации реактивных систем.

Подробнее о принципах проектирования, их достоинствах и недостатках мы поговорим в `главе 8` *Масштабирование с Cloud Streams*.

Поближе познакомиться с такими шаблонами, как «Шлюз API», «Служба реестра» и другими, используемыми для создания распределенных систем, можно по ссылке: `http://microservices.io/patterns`.

Помимо примера простого интернет-магазина, который кому-то может показаться действительно сложным, рассмотрим еще одну, более сложную область, где уместен системный подход. Этим более сложным примером нам послужит **аналитика**. Под термином «аналитика» в данном случае подразумевается способность системы обрабатывать гигантские объемы данных, преобразовывать их в процессе работы, держать пользователя в курсе оперативной статистики и т.д.

Представьте, что мы разрабатываем систему мониторинга телекоммуникационной сети и обрабатывающую данные сотовой связи. Согласно последнему статистическому отчету в 2016 году в США действовало 308 334 базовых станций сотовой связи.

Упомянутый статистический отчет доступен по адресу: `https://www.statista.com/statistics/185854/monthly-number-of-cell-sites-in-the-united-states-since-june-1986/`.

К сожалению, мы можем лишь приближенно судить о реальной нагрузке, создаваемой этими базовыми станциями. Однако, у нас не вызывает сомнения, что обработка такого огромного объема данных и обеспечение мониторинга состояния телекоммуникационной сети в реальном времени, действительно сложная задача.

При проектировании этой системы мы можем последовать за одним из эффективных архитектурных методов, который называется **потоковой обработкой данных**. На рис. 1.5 представлена абстрактная архитектура такой потоковой системы:

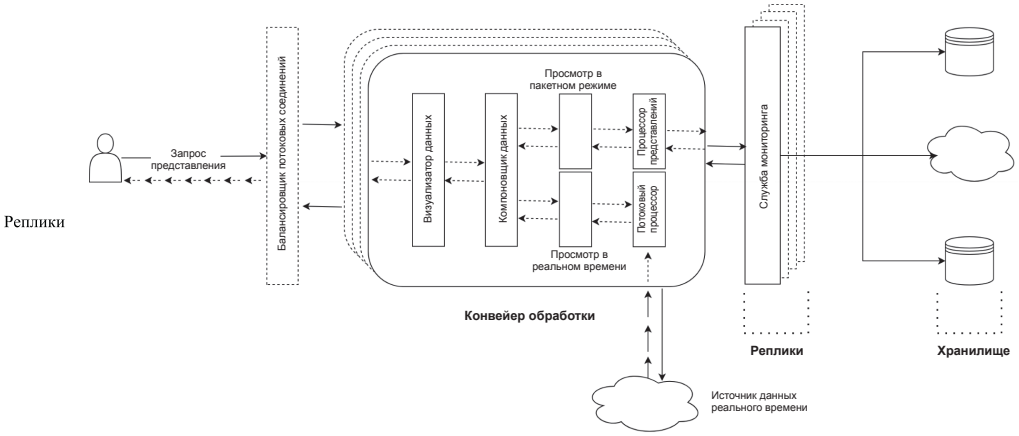


Рис. 1.5. Пример архитектуры системы аналитической обработки данных в режиме реального времени

Как можно заметить на рис. 1.5, задача потоковой архитектуры – организация конвейера обработки и преобразования данных. В целом такая система характеризуется низкой задержкой и высокой пропускной способностью. Одновременно с этим очень важную роль играет способность откликаться или просто доставлять результаты анализа состояния телекоммуникационной сети. То есть, для создания такой системы с высокой доступностью мы должны взять на вооружение фундаментальные принципы, изложенные в манифесте реактивных систем. Например, высокая отказоустойчивость может быть достигнута включением поддержки обратного давления. Под обратным давлением понимается сложный механизм управления распределением рабочей нагрузкой между этапами обработки с целью не допустить перегрузки. Добиться эффективного управления

рабочей нагрузкой можно с использованием организации обмена сообщениями через надежного брокера сообщений, который может сохранять сообщения внутри и пересылать их по требованию.

Другие приемы поддержки обратного давления мы рассмотрим в главе 3 *Reactive Streams – новый стандарт потоков*.

Более того, организовав надлежащее масштабирование каждого компонента, мы получим возможность гибко изменять пропускную способность системы.

Более подробное описание терминологии можно найти по ссылке:

обратное давление: `https://www.reactivemanifesto.org/ru/glossary#Back-Pressure`.

На практике потоки данных могут сохраняться в базах данных и обрабатываться пакетами или частично обрабатываться в режиме реального времени с применением методов машинного обучения. Как бы то ни было, здесь применимы все фундаментальные принципы манифеста реактивных систем, независимо от предметной области.

Подводя итог, можно сказать, что существует масса областей, где можно с успехом использовать принципы построения реактивных систем. Область применения реактивных систем не ограничивается предыдущими примерами, поскольку соответствующие принципы можно применять для создания практически любых распределенных систем, ориентированных на предоставление пользователям эффективной обратной связи.

В следующем разделе мы рассмотрим причины перехода фреймворка Spring Framework к использованию реактивности.

# Причины добавления поддержки реактивности в Spring

В предыдущем разделе мы рассмотрели несколько интересных примеров, где преимущества реактивных подходов проявляются во всей красе. Мы также охватили такие базовые принципы, как эластичность и устойчивость и познакомились с примерами систем на основе микрослужб, обычно используемых для создания реактивных систем.

Это дало нам понимание архитектурной перспективы, но ничего не сообщило о реализации. Однако важно подчеркнуть сложность реактивных систем и, что конструирование таких систем – не простая задача. Чтобы нам проще было создавать реактивные системы, мы должны проанализировать фреймворки, пригодные для этого, а затем выбрать один из них. Один из самых популярных способов выбора фреймворка – анализ его особенностей, релевантности, и наличие сообщества.

В мире JVM наиболее известными фреймворками для создания реактивных систем являются экосистемы Akka и Vert.x.

С одной стороны у нас есть Akka – популярный фреймворк с огромным списком возможностей и обширным сообществом. Однако первоначально Akka был частью экосистемы языка Scala и долгое время демонстрировал свою силу только в решениях на Scala. Несмотря на то что Scala является языком JVM, он заметно отличается от Java. Несколько лет назад в Akka была добавлена прямая поддержка Java, но по каким-то причинам он не получил в мире Java такой же популярности, как в Scala.

С другой стороны существует фреймворк Vert.x, который также является мощным решением для создания эффективных реактивных систем. Vert.x создавался как неблокирующая альтернатива Node.js, ориентированная на события, которая выполняется под управлением виртуальной машины Java. Однако, Vert.x вступил в конкурентную борьбу всего несколько лет назад, тогда как последние 15 лет рынок фреймворков для разработки гибких и надежных приложений уверенно удерживается фреймворком Spring Framework.

Дополнительную информацию о ландшафте инструментов Java вы найдете по ссылке: `https://www.quora.com/Is-it-worth-learning-Java-Spring-MVC-as-of-March-2016/answer/Krishna-Srinivasan-6?srid=xCnf`.

Фреймворк Spring Framework предлагает широкий спектр возможностей для создания веб-приложений с использованием дружественной модели программирования. Однако, долгое время он имел некоторые ограничения, мешающие созданию надежных реактивных систем.

## Реактивность на уровне служб

К счастью, большой спрос на реактивные системы стал причиной создания нового проекта Spring с названием Spring Cloud. Фреймворк Spring Cloud Framework – это основа для проектов, решающая конкретные проблемы и упрощающая конструирование распределенных систем. Следовательно, экосистема Spring Framework имеет непосредственное отношение к нам, занимающимся созданием реактивных систем.

Узнать больше об основных возможностях, компонентах и особенностях этого проекта можно по ссылке: `http://projects.spring.io/spring-cloud/`.

В этой главе мы не будем обсуждать детали функционирования Spring Cloud Framework, а наиболее важные его части, помогающие в разработке реактивных систем, рассмотрим в `главе 8` *Масштабирование с Cloud Streams*. Тем не менее, следует отметить, что это решение помогает с минимальными усилиями создавать надежные реактивные системы на основе микрослужб.

Однако, общий дизайн – лишь один из конструктивных элементов реактивной системы. Как отмечается в манифесте реактивных систем:

> *«Большие системы состоят из подсистем, имеющих те же свойства и, следовательно, зависят от их реактивных характеристик. То есть, принципы Реактивных Систем применяются на всех уровнях, что позволяет компоновать их между собой.»*

Поэтому очень важно обеспечить реактивный дизайн и реализацию на уровне компонентов. В данном контексте термин *дизайн* относится отношениям между компонентами и методам программирования, которые применяются для их объединения. Наиболее популярной техникой программирования на Java является **императивное программирование**.

Рассмотрим рис. 1.6, чтобы понять, насколько императивное программирование согласуется с принципами организации реактивных систем.

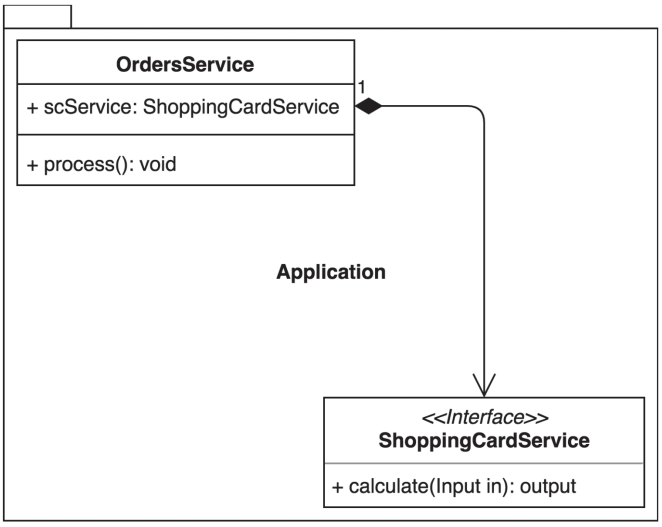


Рис. 1.6. Диаграмма UML отношений между компонентами

Здесь изображены два компонента приложения интернет-магазина. В данном случае `OrdersService` вызывает `ShoppingCardService` в процессе обработки запроса пользователя. Допустим, что `ShoppingCardService` выполняет продолжительную операцию ввода/вывода, например, посылает HTTP-запрос или обращается к удаленной базе данных. Чтобы проявить недостатки императивного программирования, рассмотрим следующую распространенную реализацию взаимодействий между этими двумя компонентами:

```
interface ShoppingCardService {                                    // (1)
   Output calculate(Input value);                                  //
}                                                                  //
```

```
class OrdersService {                                          // (2)
   private final ShoppingCardService scService;                //

   void process() {                                            //
      Input input = ...;                                       //
      Output output = scService.calculate(input);              // (2.1)
      ...                                                      // (2.2)
   }                                                           //
}                                                              //
```

Вот как действует этот код:

1. Это объявление интерфейса `ShoppingCardService`. Оно соответствует диаграмме классов на рис. 1.6 и определяет единственный метод `calculate`, который принимает один аргумент и возвращает ответ после обработки.
2. Это объявление класса `OrderService`. Здесь, в строке (`2.1`), производится синхронный вызов экземпляра `ShoppingCardService` и прием его результата. В строке (`2.2`) находится остальной код, обрабатывающий полученный результат.
3. В данном случае наши службы оказываются тесно связанными, то есть, выполнение `OrderService` сильно зависит от выполнения `ShoppingCardService`. К сожалению, при таком подходе у нас нет возможности выполнять какие-либо другие действия, пока `ShoppingCardService` занят обработкой запроса.

Как нетрудно понять из этого примера, в мире Java выполнение `scService.calculate(input)` блокирует поток `Thread`, в котором выполняется логика `OrdersService`. То есть, чтобы организовать независимую обработку в `OrderService`, мы должны запустить дополнительный поток `Thread`. Как будет показано далее в этой главе, создание нового потока `Thread` может оказаться непозволительным расточительством. Следовательно, с точки зрения реактивной системы, такое поведение абсолютно неприемлемо.

Блокирующие взаимодействия прямо противоречат принципу взаимодействий с применением сообщений, который явно предлагает возможность неблокирующих взаимодействий. Дополнительную информацию вы найдете по ссылке: `https://www.reactivemanifesto.org/ru#message-driven`.

Впрочем, в Java эту проблему можно решить, использовав для организации между компонентами технику обратных вызовов:

```
interface ShoppingCardService {                                // (1)
   void calculate(Input value, Consumer<Output> c);            //
}                                                              //

class OrdersService {                                          // (2)
   private final ShoppingCardService scService;                //
                                                               //
```

```
    void process() {                                                    //
       Input input = ...;                                               //
          scService.calculate(input, output -> {                        // (2.1)
          ...                                                           // (2.2)
       });                                                              //
    }                                                                   //
}                                                                       //
```

Вот как действует этот код:

1. Объявление интерфейса ShoppingCardService. На этот раз метод calculate принимает два аргумента и ничего не возвращает. То есть, вызывающий компонент не должен ждать ответа, потому что ответ будет передан указанному обратному вызову Consumer<>, когда будет готов.
2. Это объявление OrderService. Здесь в строке (2.1) выполняется асинхронный вызов ShoppingCardService, который тут же возвращает управление и выполнение OrderService продолжается. Позднее, когда ShoppingCardService передаст ответ функции обратного вызова, мы сможем обработать его (2.2).

Теперь OrdersService передает функцию обратного вызова, реализующую реакцию на окончание вычислений. Это означает, что теперь OrdersService не связан с ShoppingCardService и, как предполагает реализация метода ShoppingCardService#calculate, будет уведомлен о готовности результата посредством обратного вызова, синхронно или асинхронно:

```
class SyncShoppingCardService implements ShoppingCardService {  // (1)
   public void calculate(Input value, Consumer<Output> c) {     //
      Output result = new Output();                              //
      c.accept(result);                                          // (1.1)
   }                                                             //
}                                                                //

class AsyncShoppingCardService implements ShoppingCardService { // (2)
   public void calculate(Input value, Consumer<Output> c) {     //
      new Thread(() -> {                                         // (2.1)
         Output result = template.getForObject(...);             // (2.2)
         ...                                                     //
         c.accept(result);                                       // (2.3)
      }).start();                                                // (2.4)
   }                                                             //
}                                                                //
```

Вот как действует этот код:

1. Объявление класса SyncShoppingCardService. Эта реализация предполагает отсутствие блокирующих операций. Поскольку ввод/вывод не производится, результат можно вернуть немедленно посредством функции обратного вызова (1.1).

2. Это объявление класса AsyncShoppingCardService. В данном случае имеет место блокирующая операция ввода/вывода в строке (2.2), поэтому для обработки запроса запускается новый поток Thread (2.1) (2.4). После получения результата он передается через функцию обратного вызова.

В этом примере представлена синхронная реализация ShoppingCardService, которая остается в рамках синхронного выполнения и не дает никаких преимуществ. Однако в асинхронной версии мы покидаем синхронный мир и выполняем запрос в отдельном потоке Thread. Компонент OrdersService отделен от процесса обработки и будет уведомлен о завершении через функцию обратного вызова.

Преимущество этого приема в том, что компоненты не связаны друг с другом во времени. То есть, после вызова метода scService.calculate компонент может тут же продолжить выполнять другие операции, не дожидаясь ответа от ShoppingCardService.

А недостаток в том, что для использования техники обратных вызовов разработчик должен хорошо владеть приемами многопоточного программирования, чтобы не попасть в ловушку изменения общих данных и избежать ада обратных вызовов.

Фактически определение ад обратных вызовов зародилось в JavaScript: http://callbackhell.com, но оно также применимо к Java.

К счастью, обратные вызовы – не единственный способ организации асинхронных взаимодействий. Другой способ основан на использовании java.util.concurrent.Future, который до определенной степени скрывает свое поведение и тоже отделяет компоненты друг от друга:

```
interface ShoppingCardService {
// (1)
   Future<Output> calculate(Input value);                          //
}                                                                  //

class OrdersService {                                              // (2)
   private final ShoppingCardService scService;                    //
                                                                   //
   void process() {                                                //
      Input input = ...;                                           //
      Future<Output> future = scService.calculate(input);          // (2.1)
      ...                                                          //
      Output output = future.get();                                // (2.2)
      ...                                                          //
   }                                                               //
}                                                                  //
```

Вот как действует этот код:

1. Объявление интерфейса ShoppingCardService. Здесь метод calculate принимает один аргумент и возвращает экземпляр Future. Future – это класс обертки, позволяющей проверить доступность результата или заблокировать дальнейшее выполнение в его ожидании.
2. Объявление OrderService. Здесь, в строке (2.1), мы выполняем асинхронный вызов ShoppingCardService и получаем экземпляр Future. После этого мы можем продолжать выполнять другие операции, пока не завершится асинхронная обработка запроса. Спустя некоторое время, мы можем проверить завершение работы ShoppingCardService#calculation и получить результат. В данном случае (2.2) попытка может окончиться блокировкой или вернет результат немедленно.

Как можно заметить в предыдущем коде, класс Future позволяет нам отложить получение результата. Благодаря классу Future мы избегаем ада обратных вызовов и абстрагируемся от сложностей многопоточного программирования. В любом случае, чтобы получить нужный результат, мы должны, возможно на продолжительное время, заблокировать текущий поток Thread и синхронизироваться с внешним выполнением, что заметно ухудшает масштабируемость.

С целью исправить эту проблему, в Java 8 предлагаются CompletionStage и его прямая реализация CompletableFuture. Эти классы поддерживают promise-подобные API и позволяют писать такой код:

Узнать больше о механизмах future и promise отложенных вычислений можно по ссылке: https://ru.wikipedia.org/wiki/Futures_and_promises.

```
interface ShoppingCardService {                                    // (1)
   CompletionStage<Output> calculate(Input value);                 //
}                                                                  //

class OrdersService {                                              //(2)
   private final ComponentB componentB;                            //
   void process() {                                                //
      Input input = ...;                                           //
         componentB.calculate(input)                               // (2.1)
         .thenApply(out1 -> { ... })                               // (2.2)
         .thenCombine(out2 -> { ... })                             //
         .thenAccept(out3 -> { ... })                              //
   }                                                               //
}                                                                  //
```

Вот как действует этот код:

1. Объявление интерфейса ShoppingCardService. В данном случае метод calculate принимает один аргумент и возвращает экземпляр CompletionStage. CompletionStage – это класс обертки, похожей на Future, но позволяющий

обрабатывать и возвращать результат в декларативной манере функционального программирования.

2. Объявление `OrderService`. Здесь, в строке (`2.1`), мы асинхронно вызываем `ShoppingCardService` и немедленно получаем экземпляр `CompletionStage`. В целом `CompletionStage` действует подобно `Future`, но предлагает более гибкий API с такими методами, как `thenAccept` и `thenCombine`. Они позволяют определить преобразующие операции с результатом и конечных потребителей преобразованного результата.

Благодаря `CompletionStage`, мы можем писать код в функциональном и декларативном стиле, который выглядит прозрачно и обрабатывает результат асинхронно. Кроме того, мы можем избежать ожидания результата и определить функцию для его обработки, когда он станет доступен. Более того, все предыдущие приемы прошли экспертизу разработчиков Spring и уже реализованы в большинстве проектов, входящих в состав этого фреймворка. К сожалению, несмотря на то что `CompletionStage` предлагает более широкие возможности для создания эффективного и легко читаемого кода, имеют место некоторые упущения. Например, Spring 4 MVC долгое время не поддерживал `CompletionStage` и для тех же целей предлагал собственную реализацию в форме ListenableFuture. Это случилось потому, что разработчики Spring 4 стремились сохранить совместимость со старыми версиями Java. Давайте рассмотрим пример использования `AsyncRestTemplate`, чтобы понять, как работать с классом `ListenableFuture`. Следующий код демонстрирует один из вариантов использования ListenableFuture в паре с `AsyncRestTemplate`:

```
AsyncRestTemplate template = new AsyncRestTemplate();
SuccessCallback onSuccess = r -> { ... };
FailureCallback onFailure = e -> { ... };
ListenableFuture<?> response = template.getForEntity(
   "http://example.com/api/examples",
   ExamplesCollection.class
);
response.addCallback(onSuccess, onFailure);
```

Этот код демонстрирует реализацию асинхронных взаимодействий в стиле обратных вызовов. По сути этот прием является грубым хаком и за кулисами фреймворк Spring Framework заворачивает блокирующие сетевые вызовы в отдельные потоки выполнения. Кроме того, Spring MVC опирается на Servlet API, который обязывает все реализации использовать модель выполнения запросов в отдельных потоках.

Многое изменилось с выпуском Spring Framework 5 и нового проекта Reactive `WebClient`,. Теперь, благодаря поддержке `WebClient`, появилась возможность использовать неблокирующие методы взаимодействий между службами. Также в Servlet 3.0 появилась поддержка асинхронных взаимодействий клиент-сервер, в Servlet 3.1 добавлена возможность неблокирующей записи в операциях ввода/вывод, и в целом новые асин-

хронные механизмы в Servlet 3 прекрасно интегрированы в Spring MVC. Единственная проблема состоит в том, что в Spring MVC отсутствует готовый, асинхронный и неблокирующий клиент, что сводит на нет все преимущества улучшенных сервлетов.

Это далеко не оптимальная модель. Чтобы понять причины неэффективности этого подхода, необходимо принять во внимание дороговизну многопоточного выполнения. С одной стороны, многопоточность сама по себе очень сложна. Работая с потоками выполнения, нам приходится учитывать множество самых разных мелочей, таких как доступ к общей памяти из разных потоков, синхронизация, обработка ошибок и т.д. Дизайн многопоточности в Java, в свою очередь, предполагает, что несколько потоков могут использовать единственный процессор для одновременного выполнения заданий. Так как в этом случае процессорное время распределяется между несколькими потоками, возникает необходимость **переключения контекста**. То есть, чтобы возобновить выполнение потока позже, нужно сохранить и загрузить регистры, карты памяти и выполнить другие операции с большим объемом вычислений. Из-за этого снижается эффект от применения многопоточности в случаях с большим количеством потоков и небольшим количеством процессоров.

Узнать больше о стоимости переключения контекста можно по ссылке: `https://ru.wikipedia.org/wiki/Переключение_контекста`.

Кроме того, типичный поток выполнения в Java предъявляет довольно серьезные требования к потреблению памяти. Размер стека типичного потока в 64-разрядной версии Java VM равен 1 024 Кбайт. С одной стороны, попытка обработать сразу ~64 000 запросов с использованием модели, выделяющей отдельный поток для каждого запроса, потребует около 64 Гбайт памяти, что может быть довольно дорого с точки зрения бизнеса. С другой стороны, решение на основе традиционных пулов потоков ограниченного размера и предварительно настроенной очереди запросов менее надежно и может заставить клиентов ждать ответа слишком долго.

Для этих целей в манифесте реактивных систем рекомендуется использовать неблокирующие операции, и эта рекомендация была проигнорирована в экосистеме Spring. С другой стороны, отсутствие хорошей интеграции с реактивными серверами, такими как Netty, отчасти решает проблему переключения контекста.

Получить больше информации о среднем количестве соединений можно по ссылке: https://stackoverflow.com/questions/2332741/what-is-the-theoretical-maximum-number-of-open-tcp-connections-that-a-modernlin/2332756#2332756.

Термин «поток» относится к памяти, выделенной для объекта потока и для стека потока. Подробности вы найдете по ссылке: `http://xmlandmore.blogspot.com/2014/09/jdk-8-thread-stack-size-tuning.html?m=1`.

Важно отметить, что асинхронная обработка не ограничивается простым шаблоном запрос/ответ, и иногда нам приходится иметь дело с бесконечными потоками данных и использовать для их обработки цепочки преобразований с поддержкой обратного давления:

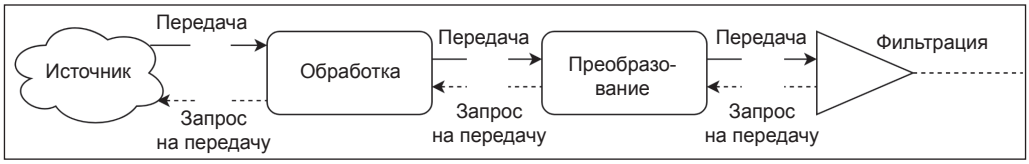


Рис. 1.7. Пример реактивного конвейера

Один из способов справиться с подобной ситуацией – использовать приемы реактивного программирования, включающие методы асинхронной обработки событий с использованием цепочки этапов преобразований. То есть, приемы реактивного программирования в полной мере соответствуют требованиям проектирования реактивных систем. В следующих главах мы рассмотрим важность и ценность использования подходов реактивного программирования для создания реактивных систем.

К сожалению, методы реактивного программирования плохо поддерживались фреймворком Spring Framework. Это стало еще одним ограничением для разработчиков современных приложений и снизило конкурентоспособность фреймворка. Как следствие, все упомянутые пробелы, связанные с реактивными системами и реактивным программированием, просто увеличили потребность в кардинальном улучшении фреймворка. Наконец, добавление поддержки реактивности на всех уровнях способствовало бы увеличению привлекательности фреймворка Spring Framework и дало бы разработчикам мощный инструмент для разработки реактивных систем. По этим причинам ключевые разработчики фреймворка решили реализовать новые модули, концентрирующие в себе всю мощь Spring Framework, как основы для реактивных систем.

# В заключение

В этой главе мы подчеркнули требования к современным и экономически эффективным IT-решениям. Мы описали, почему и как большие компании, такие как Amazon, потерпели неудачу в попытках использовать старые архитектурные решения в современных облачных окружениях.

Мы также установили, что действительно имеется потребность в новых архитектурных шаблонах и приемах программирования для удовлетворения постоянно растущего спроса на удобные, эффективные и интеллектуальные цифровые услуги. С помощью манифеста реактивных систем мы разобрали понятие реактивности и показали, как и почему эластичность, устойчивость и взаимодействие с использованием сообщений помогают добиться высокой отзывчивости, что явля-

ется основным нефункциональным требованием к системам в нашу цифровую эпоху. И, конечно же, мы привели примеры ситуаций, когда реактивные системы позволяют предприятиям добиться своих целей.

В этой главе мы подчеркнули различия между реактивной системой, как архитектурным шаблоном, и реактивным программированием, как подходом к программированию. Мы описали, как и почему сочетание этих двух сторон реактивности позволяют нам создавать высокоэффективные IT-решения.

Для углубленного исследования особенностей Reactive Spring 5 необходимо знать и понимать основы реактивного программирования, изучить основные понятия и шаблоны. Поэтому в следующей главе мы займемся изучением сути реактивного программирования, его истории и ландшафта реактивного программирования в мире Java.