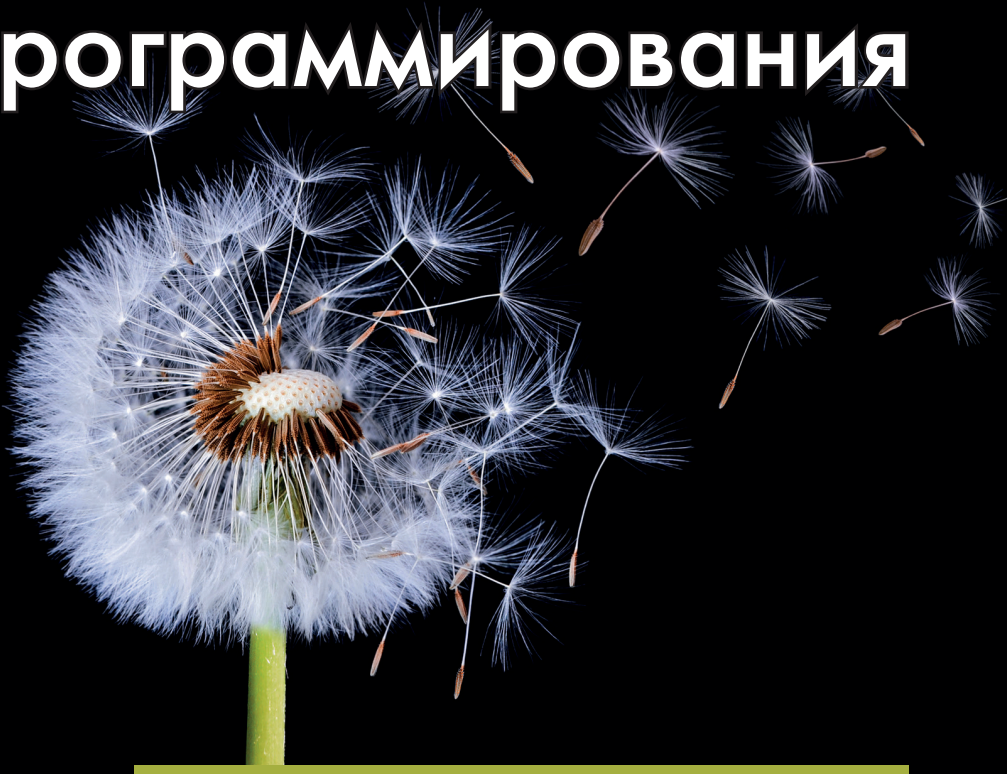
Олег Докука
Игорь Лозинский

Практика реактивного программирования в Spring 5



ДМК Пресс
издательство

УДК 004.432
ББК 32.972.1
Д63

Д63 Олег Докука, Игорь Лозинский
Практика реактивного программирования в Spring 5. –
М.: ДМК Пресс, 2019. – 508 с.

ISBN 978-5-97060-747-3

Данная книга посвящена реактивному программированию в Spring. Описаны многочисленные возможности построения эффективных реактивных систем с помощью Spring 5 и других инструментов, таких как WebFlux, Spring Boot и Project Reactor. Приведены методы реактивного программирования и их использование для взаимодействий с базами данных и между серверами. Рассмотрено создание независимых и высокопроизводительных микросервисов с помощью Spring Cloud Streams.

Издание предназначено разработчикам на Java, использующим фреймворк Spring для своих задач и желающим научиться создавать надежные и реактивные приложения, способные автоматически масштабироваться в облаке.

УДК 004.432
ББК 32.972.1

ISBN 978-5-97060-747-3 (рус.)
ISBN 978-1-78728-495-1 (анг.)

Оглавление

Предисловие 7

Глава 1. Причины выбора Spring 22

Основные преимущества реактивности 22

Взаимодействия на основе обмена сообщениями 25

Примеры использования реактивности 30

Причины добавления поддержки реактивности в Spring 33

Реактивность на уровне служб 34

Заключение 42

Глава 2. Реактивное программирование в Spring. Основные понятия 44

Первые реактивные решения в Spring 44

Шаблон «Наблюдатель» 45

Примеры использования шаблона «Наблюдатель» 49

Шаблон «Публикация/Подписка» с использованием @EventListener 52

Создание приложений с @EventListener 54

Создание приложения на основе Spring 54

Реализация бизнес-логики 55

Асинхронные взаимодействия по HTTP с помощью Spring Web MVC 57

Публикация конечной точки SSE 57

Настройка поддержки асинхронного выполнения 59

Создание пользовательского интерфейса с поддержкой SSE 60

Проверка приложения 61

Критический обзор решения 61

RxJava как реактивный фреймворк 62

«Наблюдатель» плюс «Итератор» равно «реактивный поток» 63

Производство и потребление потоков 65

Генерация последовательности асинхронных событий 68

Преобразование потоков и диаграммы Marble 69

Оператор map 69

Оператор filter 70

Оператор count 71

Оператор zip 71

Требования и преимущества RxJava 72

Переделка приложения с RxJava 75

Реализация бизнес-логики 75

Нестандартный SseEmitter 77

Публикация конечной точки SSE 78

Конфигурация приложения 79

Краткая история развития реактивных библиотек 80

Реактивный ландшафт 82

Заключение 84

Глава 3. Reactive Streams – новый стандарт потоков 85

Реактивность для всех 85

Проблема несовместимости API 86

Модели обмена PULL и PUSH 89
Проблема управления потоком данных 95
Медленный производитель и быстрый потребитель 95
Быстрый производитель и медленный потребитель 96
Неограниченная очередь 96
Ограниченная очередь со сбросом избыточных элементов 97
Ограниченная очередь с блокировкой 97
Решение 99
Основные положения стандарта Reactive Streams 99
Требования Reactive Streams в действии 106
Введение в понятие обработчика Processor 109
Проверка совместимости с Reactive Streams 113
Проверка издателя Publisher 115
Проверка подписчика Subscriber 117
JDK 9 121
Асинхронный и параллельный API в Reactive Streams 123
Преображение реактивного ландшафта 125
Изменения в RxJava 125
Изменения в Vert.x 129
Усовершенствования в Ratpack 130
Драйвер MongoDB с поддержкой Reactive Streams 131
Комбинирование реактивных технологий на практике 132
Заключение 135

Глава 4. Project Reactor – основа реактивных приложений 137
Краткая история Project Reactor 137
Project Reactor 1.x 138
Project Reactor 2.x 141
Основы Project Reactor 142
Добавление библиотеки Reactor в проект 144
Реактивные типы: Flux и Mono 145
Flux 145
Mono 147
Реактивные типы из RxJava 2 148
Observable 148
Flowable 148
Single 148
Maybe 149
Completable 149
Создание последовательностей Flux и Mono 149
Подписка на реактивный поток 151
Реализация своих подписчиков 154
Преобразование реактивных последовательностей с помощью операторов 156
Отображение элементов реактивных последовательностей 157
Фильтрация реактивных последовательностей 158
Сбор данных из реактивных последовательностей 160

Сокращение элементов потока. 161
Комбинирование реактивных потоков. 164
Пакетная обработка элементов потока 164
Операторы flatMap, concatMap и flatMapSequential 168
Извлечение выборки элементов 170
Преобразование реактивных последовательностей
в блокирующие структуры. 170
Просмотр элементов при обработке последовательности. 171
Материализация и дематериализация сигналов 172
Поиск подходящего оператора 173
Создание потоков данных программным способом 173
Фабричные методы push и create 173
Фабричный метод generate. 174
Передача одноразовых ресурсов в реактивные потоки. 175
Обертывание транзакций с помощью фабричного метода usingWhen. 178
Обработка ошибок. 180
Управление обратным давлением 183
Горячие и холодные потоки данных 184
Широковещательная рассылка элементов потока данных 185
Кеширование элементов потока 186
Совместное использование элементов из потока 187
Работа со временем 188
Компоновка и преобразование реактивных потоков 188
Процессоры. 190
Тестирование и отладка Project Reactor. 191
Дополнения к Reactor. 192
Продвинутые средства в Project Reactor 193
Жизненный цикл реактивных потоков данных. 193
Этап сборки. 193
Этап подписки 195
Выполнение 196
Модель планирования потоков выполнения в Reactor 199
Оператор publishOn 199
Параллельная обработка с помощью publishOn 201
Оператор subscribeOn 202
Оператор parallel. 204
Планировщик 204
Контекст. 205
Особенности внутренней реализации Project Reactor 209
Макрослияние 210
Микрослияние 211
Заключение 214
Глава 5. Добавление реактивности с помощью Spring Boot 2 216
Быстрый старт как ключ к успеху. 217
Использование Spring Roo для ускорения разработки приложений 219
Spring Boot как ключ к созданию быстро растущих приложений. 219

Реактивность в Spring Boot 2.0 220
Реактивность в Spring Core 221
Поддержка преобразования реактивных типов 221
Реактивный ввод/вывод 222
Реактивность в Web 224
Реактивность в Spring Data 226
Реактивность в Spring Session 227
Реактивность в Spring Security 228
Реактивность в Spring Cloud 228
Реактивность в Spring Test 229
Реактивность в мониторинге 229
Заключение 230

Глава 6. Неблокирующие и асинхронные взаимодействия с WebFlux 231
WebFlux как основа реактивного сервера 231
Реактивное веб-ядро 234
Реактивные фреймворки Web и MVC 238
Чисто функциональные приемы в WebFlux 242
Неблокирующие взаимодействия между службами с WebClient 246
Реактивный WebSocket API 249
Серверный WebSocket API 250
Клиентский WebSocket API 251
Сравнение WebFlux WebSocket и Spring WebSocket 252
Реактивный поток SSE и легковесная замена WebSocket 253
Реактивные механизмы шаблонов 255
Реактивная безопасность 258
Реактивный доступ к SecurityContext 258
Использование реактивной безопасности 261
Взаимодействия с другими реактивными библиотеками 262
Сравнение WebFlux и Web MVC 263
Законы сравнения фреймворков 264
Закон Литтла 264
Закон Амдала 265
Универсальный закон масштабируемости 269
Анализ и сравнение 272
Модели обработки в WebFlux и Web MVC 272
Влияние моделей обработки на пропускную способность и задержку 274
Проблемы модели обработки в WebFlux 282
Потребление памяти разными моделями обработки 285
Влияние модели обработки на удобство 291
Практическое применение WebFlux 292
Системы на основе микросервисов 292
Системы, обслуживающие клиентов с медленными соединениями 294
Потоковые системы или системы реального времени 294
WebFlux в действии 295
Заключение 299

Глава 7. Реактивный доступ к базам данных 301
Модели обработки данных в современном мире 302
Предметно-ориентированное проектирование 302
Хранение данных в эпоху микросервисов 303
Использование хранилищ разного типа 306
База данных как услуга 307
Разделение данных между микросервисами 309
Распределенные транзакции 310
Событийно-ориентированные архитектуры 310
Согласованность в конечном счете 311
Шаблон SAGA 312
Регистрация событий 312
Разделение ответственности на команды и запросы 313
Бесконфликтно реплицируемые типы данных 314
Система обмена сообщениями как хранилище данных 315
Синхронная модель извлечения данных 316
Протокол связи для доступа к базе данных 316
Драйвер базы данных 318
JDBC 319
Управление соединениями 320
Реактивный доступ к базе данных 321
Spring JDBC 322
Spring Data JDBC 323
Добавление реактивности в Spring Data JDBC 326
JPA 326
Добавление реактивности в JPA 327
Spring Data JPA 327
Добавление реактивности в Spring Data JPA 328
Spring Data NoSQL 329
Ограничения синхронной модели 332
Достоинства синхронной модели 333
Реактивный доступ к данным с использованием Spring Data 334
Реактивное хранилище на основе MongoDB 336
Объединение операций с хранилищем 339
Как работают реактивные хранилища 344
Поддержка разбиения на страницы 345
Детали реализации ReactiveMongoRepository 345
Использование ReactiveMongoTemplate 346
Использование реактивных драйверов (MongoDB) 348
Использование асинхронных драйверов (Cassandra) 350
Реактивные транзакции 352
Реактивные транзакции в MongoDB 4 352
Распределенные транзакции с шаблоном SAGA 361
Реактивные коннекторы в Spring Data 361
Реактивный коннектор MongoDB 361
Реактивный коннектор Cassandra 362

Реактивный коннектор Couchbase 362
Реактивный коннектор Redis 363
Ограничения и ожидаемые улучшения 364
Асинхронный доступ к базам данных 365
Реактивное соединение с реляционной базой данных 367
Использование R2DBC вместе с Spring Data R2DBC 369
Преобразование синхронного хранилища в реактивное 370
С помощью библиотеки rxjava2-jdbc 371
Обертывание синхронного CrudRepository 373
Реактивный Spring Data в действии 378
Заключение 382

Глава 8. Масштабирование с Cloud Streams 383
Брокеры сообщений как основа систем, управляемых сообщениями 384
Балансировка нагрузки на стороне сервера 384
Балансировка нагрузки на стороне клиента с Spring Cloud и Ribbon 386
Брокеры сообщений как эластичный и надежный слой для передачи сообщений 392
Рынок брокеров сообщений 396
Spring Cloud Streams как мост в экосистему Spring 397
Реактивное программирование в облаке 406
Spring Cloud Data Flow 407
Модульная организация приложений с Spring Cloud Function 409
Spring Cloud – функция как часть конвейера обработки данных 416
RSocket для реактивной передачи сообщений с низкой задержкой 420
RSocket и Reactor-Netty 421
RSocket в Java 425
RSocket и gRPC 428
RSocket в Spring Framework 430
RSocket в других фреймворках 432
Проект ScaleCube 432
Проект Proteus 433
В заключение о RSocket 433
Заключение 433

Глава 9. Тестирование реактивных приложений 435
Почему реактивные потоки данных сложно тестировать? 435
Тестирование реактивных потоков с помощью StepVerifier 436
Основы StepVerifier 436
Продвинутые приемы тестирования с использованием StepVerifier 440
Виртуальное время 442
Проверка реактивного контекста 445
Тестирование WebFlux 445
Тестирование контроллеров с помощью WebTestClient 446
Тестирование WebSocket 451

Заключение 455

Глава 10. И наконец, выпуск! 456

Важность поддержки идеологии DevOps в приложениях 456

Мониторинг реактивных Spring-приложений 460

Spring Boot Actuator 460

Добавление механизма мониторинга в проект 460

Конечная точка для получения информации о службе 461

Конечная точка для получения информации о работоспособности 463

Конечная точка для получения информации о параметрах работы 466

Конечная точка управления журналированием 467

Другие важные конечные точки 468

Реализация своей конечной точки для Actuator 469

Безопасность конечных точек 470

Micrometer 472

Параметры по умолчанию в Spring Boot 473

Мониторинг реактивных потоков данных 474

Мониторинг потоков в Reactor 474

Мониторинг планировщиков в Reactor 475

Реализация своих параметров Micrometer 477

Распределенная трассировка с Spring Boot Sleuth 478

Пользовательский интерфейс Spring Boot Admin 2.x 480

Развертывание в облаке 482

Развертывание в Amazon Web Services 485

Развертывание в Google Kubernetes Engine 486

Развертывание в Pivotal Cloud Foundry 487

Обнаружение RabbitMQ в PCF 488

Обнаружение MongoDB в PCF 489

Развертывание в PCF без конфигурации с помощью Spring Cloud Data Flow . . 491

Knative для FaaS на основе Kubernetes и Istio 491

Советы по успешному развертыванию приложений 492

Заключение 493

Указатель 495

Глава 1

Причины выбора Spring

В этой главе мы объясним понятие **реактивности** и расскажем, почему реактивные подходы лучше традиционных, для чего рассмотрим примеры, когда традиционные подходы терпят неудачу. Затем исследуем фундаментальные принципы построения надежных систем, которые в большинстве своем являются **реактивными системами**. Узнаем, каковы основные причины, объясняющие необходимость использования механизмов рассылки сообщений для организации взаимодействий между распределенными серверами. Мы покажем, в какие случаи реактивность вписывается как нельзя лучше, расскажем о приемах **реактивного программирования** для создания модульной реактивной системы. Кроме того, обсудим, почему команда разработчиков Spring Framework решила включить реактивный подход в ядро фреймворка **Spring Framework 5**. Прочитав главу, вы поймете важность реактивности и то, почему стоит перенести свои проекты в реактивный мир.

Здесь 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 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 слово заняло почетное место на разнообразных международных конференциях? Если забьем в поиск слово «реактивный», то обнаружим, что чаще всего оно встречается в паре со словом «программирование», и вместе они обозначают модель программирования. Однако это не единственный смысл реак-

тивности. За данным 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, наше малое предприятие – это интернет-магазин, продающий современные товары по привлекательным ценам. Как большинство владельцев подобных магазинов, мы также наняли разработчиков программного обеспечения, которые помогут нам справиться с проблемами. Мы выбрали традиционный подход к разработке программного обеспечения и создали магазин.

Каждый час наш онлайн-магазин обычно посещает 1 тыс. пользователей. Чтобы справиться с потоком покупателей, 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 – около 250 мс. Простейшие расчеты показывают, что такая конфигурация позволит нам обслуживать до 2 тыс. запросов в секунду. Согласно статистике, вышеупомянутое число пользователей в среднем производит около 1 тыс. запросов в секунду. Следовательно, текущей производительности системы вполне достаточно для обслуживания средней нагрузки.

Итак, мы настроили приложение с неплохим запасом 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 – получить дополнительную прибыль. Однако этот день характеризуется необычным наплывом клиентов, что 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 наплывом покупателей, время отклика возросло, периодически наблюдались сбои. Мы начали терять некоторые запросы, в результате наши клиенты, неудовлетворенные долгим обслуживанием, переметнулись к конкурентам.

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

Но не волнуйтесь, это далеко не новая проблема. Даже такие гиганты, как Amazon и Walmart, сталкивались с ней и давно нашли способы ее решения. Но не будем спешить и пройдем тот же путь, каким следовали наши предшественники, чтобы понять основные принципы проектирования надежных систем и дать им общее определение.

Узнать больше о проблемах магазинов-гигантов можно на следующих сайтах:

- 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: что с этим делать? Из примера выше следует, что приложение должно как-то реагировать на изменение нагрузки и доступности внешних служб. Иначе говоря, оно должно активно реагировать на любые изменения, которые могут ухудшить доступность системы и ее способность откликаться на запросы пользователей.

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

Время задержки – важная характеристика отзывчивости. При отсутствии должной эластичности из-за роста нагрузки увеличится время задержки, которое напрямую влияет на отзывчивость системы.

Например, увеличить пропускную способность системы можно, расширяя вычислительные мощности или запуская дополнительные экземпляры. В результате возрастет отзывчивость системы. С другой стороны, если поток пользователей уменьшился, система в ответ должна снизить потребление ресурсов, сократив тем самым накладные расходы. Добиться желаемой эластичности можно путем масштабирования – горизонтального или вертикального. Однако масштабирование распределенной системы – сложная задача. Обычно ограничиваются 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 новых экземпляров в облаке или дополнительное потребление электроэнергии в случае установки добавочных компьютеров.

Однако построение масштабируемой распределенной системы без возможности оставаться отзывчивой независимо от отказов – сложная задача. Представьте ситуацию: какая-то часть системы вдруг оказывается недоступной. Допустим, отказала внешняя платежная система и любые попытки пользователя произвести оплату терпят 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 качественно обслуживать клиентов, мы должны позаботиться об отзывчивости системы. Критерием приемлемости является способность системы оставаться отзывчивой в случае отказов или, говоря другими словами, сохранять устойчивость. Этого можно добиться путем изоляции функциональных компонентов системы, помогающей отделить внутренние сбои и обеспечить независимость.

Вернемся к интернет-магазину Amazon. В нем имеется большое число разных функциональных компонентов, отвечающих, например, за вывод списка заказов, оплату, рекламу, прием отзывов от пользователей и др. Например, в случае выхода из строя платежной системы мы можем принять заказ пользователя и запланировать автоматическое повторение запроса, защитив пользователя от сбоев. Другой способ – изоляция от службы приема отзывов пользователей. Если данная служба 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. Получить по-настоящему отзывчивую систему можно, только уделив должное внимание обоим параметрам. 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 вы найдете по следующим 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, – это то, как связываются компоненты распределенной системы и в то же время остаются независимыми, изолирован-

ными 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)
}
```

Данный код выполняет следующие действия.

1. Объявляет обработчик запросов с использованием аннотации `@RequestMapping`.
2. Создает экземпляр `RestTemplate` – самого популярного веб-клиента в Spring Framework 4 для организации взаимодействий типа «запрос-ответ» между службами.
3. 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`.
4. Здесь выполняются дополнительные операции, которые были опущены в этом примере для краткости.
5. Производится следующий этап обработки ответа.

В предыдущем примере мы определили обработчик запросов, который вызывается в ответ на получение запроса от пользователя. Он, в свою очередь, посылает дополнительный HTTP-запрос внешней службе, а затем передает 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 примере, рассмотрим, как протекают события во времени.

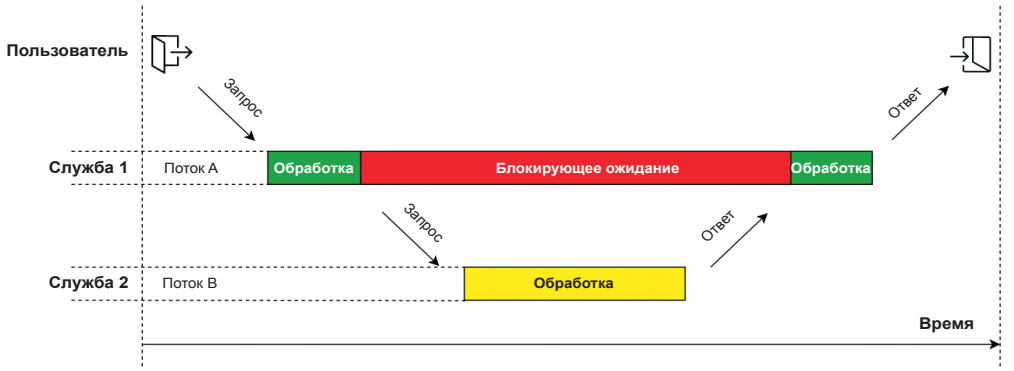


Рис. 1.1. Взаимодействия компонента во времени

Диаграмма отражает фактическое поведение кода в примере выше. Обратите внимание: процессор занят фактической работой только часть времени, тогда как остальное время поток проводит в ожидании завершения операции ввода/вывода и не может использоваться для 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, следовательно, 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 данной проблеме, 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 (на телефон или по электронной почте), мы его читаем, а также отвечаем на него, но обычно не ждем ответа и занимаемся 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`).

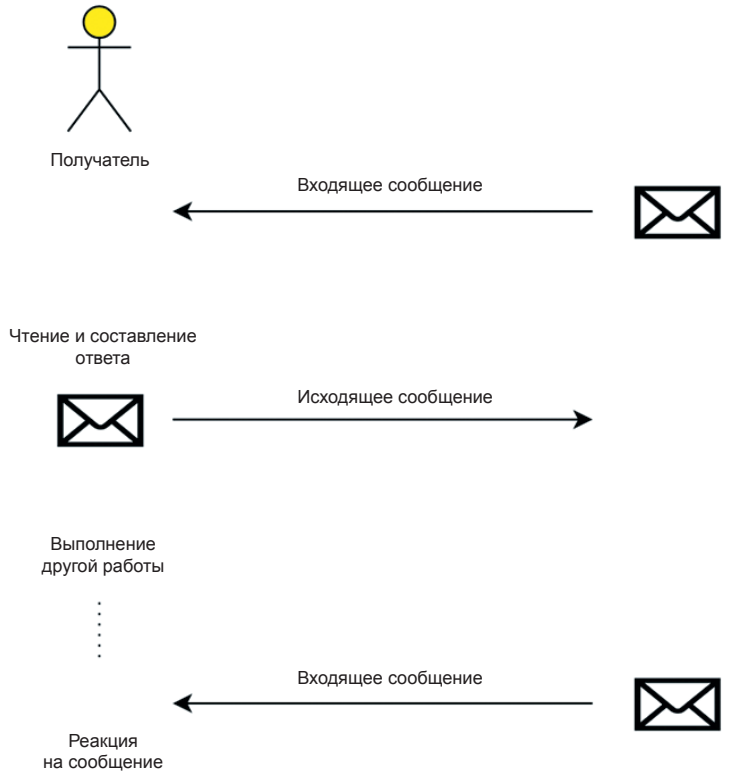


Рис. 1.2. Неблокирующий обмен сообщениями

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 выбора устройства и дает получателям возможность использовать столько устройств, сколько они пожелают. Кроме того, повышается 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 сейчас вдаваться в подробности, 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.

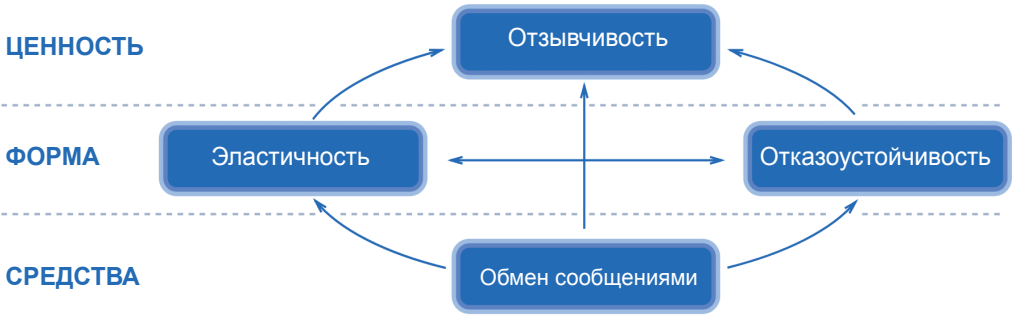


Рис. 1.3. Манифест реактивных систем

На рис. 1.3 мы видим, что главная ценность распределенной системы для любого бизнеса – это отзывчивость. Методы достижения отзывчивости имеют форму эластичности и отказоустойчивости. 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: `https://www.reactivemanifesto.org/ru/glossary/`.

Все эти понятия не новы, их определение дается в манифесте реактивных систем, который является и глоссарием, описывающим реактивные системы. Данный манифест создан для того, чтобы гарантировать одинаковое понимание традиционных идей разработчиками и предпринимателями. Отметим, что реактивные системы и манифест реактивных систем – это архитектурные понятия, они могут применяться и к большим распределенным решениям, и к малым приложениям, выполняющимся на единственном узле.

Важность манифеста реактивных систем (`https://www.reactivemanifesto.org/ru`) Йонас Бонер (Jonas Bonér), основатель и директор компании Lightbend, объясняет здесь: `https://www.lightbend.com/blog/why_do_we_need_a_reactive_manifesto%3F`.

Примеры использования реактивности

В предыдущем разделе мы узнали о важности реактивности, об основополагающих принципах реактивных систем, о том, почему организация взаимодействий посредством сообщений является важнейшей составляющей реактивной экосистемы. 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, о котором мы уже говорили. Теперь обсудим возможные усовершенствования и изменения в конструкции, которые помогут достичь реактивности. На рис. 1.4 показана общая архитектура предлагаемого решения.

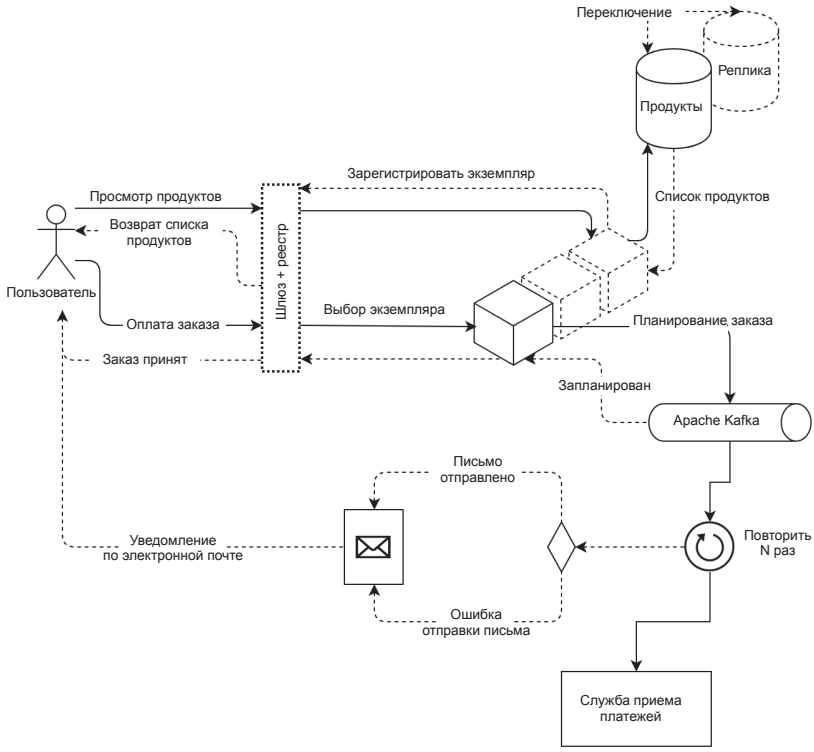


Рис. 1.4. Пример архитектуры интернет-магазина

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 микросервисов. В данном случае для обеспечения независимости от местоположения используется шаблон «Шлюз API». Он обеспечивает идентификацию конкретного ресурса, не зная, какие 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. Отметим, 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 надписью **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, 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 поговорим в `главе 8` *«Масштабирование с Cloud Streams»*.

Поближе познакомиться с такими шаблонами, как «Шлюз API», «Служба реестра» и т. д., используемыми для создания распределенных систем, можно на сайте http://microservices.io/patterns.

Помимо примера простого интернет-магазина (который тем не менее кому-то может показаться сложным), рассмотрим более замысловатую область, где уместен системный подход. Это аналитика. Под термином «аналитика» 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 обработки данных сотовой связи. Согласно последнему статистическому отчету, в 2016 году в США действовало 308 334 базовые станции сотовой связи.

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

К сожалению, мы можем лишь приблизительно судить о реальной нагрузке, создаваемой этими базовыми станциями. Однако у нас нет сомнений, что обработка такого огромного объема данных и мониторинг состояния телекоммуникационной сети в реальном времени – действительно сложная задача.

При проектировании данной 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.

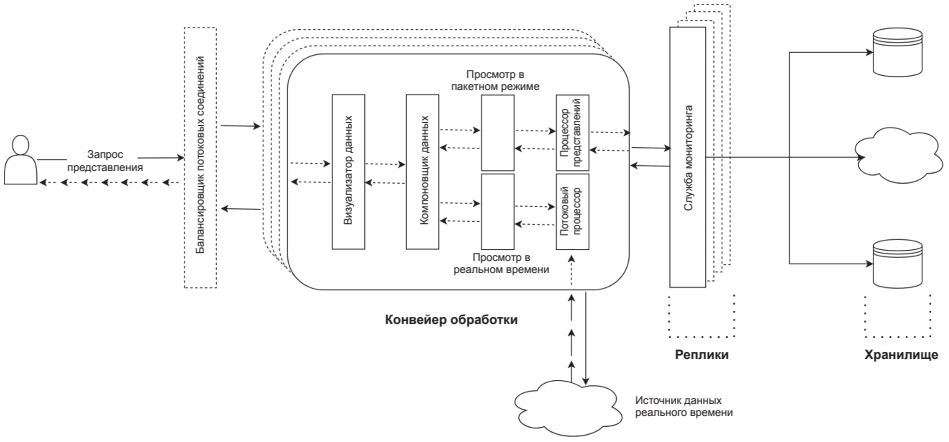


Рис. 1.5. Пример архитектуры системы аналитической обработки данных в режиме реального времени

Рисунок 1.5 демонстрирует, что задача потоковой архитектуры – организация конвейера обработки и преобразования данных. В целом такая система характеризуется низкой задержкой и высокой пропускной способностью. В то же время очень важную роль играет способность откликаться или просто доставлять результаты анализа состояния телекоммуникационной сети. То есть для создания системы с высокой доступностью мы должны взять на вооружение фундаментальные принципы, изложенные в манифесте реактивных систем. Например, большая отказоустойчивость может быть достигнута включением поддержки обратного давления. Под обратным давлением понимается сложный механизм управления распределением рабочей нагрузки между этапами обработки с целью