

Оглавление

| | |
|---|----|
| Предисловие..... | 16 |
| Благодарности..... | 19 |
| О книге..... | 21 |
| Кому следует прочитать эту книгу | 21 |
| Структура издания..... | 21 |
| О коде..... | 23 |
| Онлайн-ресурсы | 23 |
| Об авторе..... | 23 |
| Об иллюстрации на обложке | 24 |
| От издательства | 25 |
| Глава 1. Побег из монолитного ада | 26 |
| 1.1. Медленным шагом в монолитный ад | 27 |
| 1.1.1. Архитектура приложения FTGO..... | 28 |
| 1.1.2. Преимущества монолитной архитектуры..... | 29 |
| 1.1.3. Жизнь в монолитном аду | 29 |
| 1.2. Почему эта книга актуальна для вас | 32 |

| | |
|---|-----------|
| 1.3. Чему вы научитесь, прочитав эту книгу..... | 33 |
| 1.4. Микросервисная архитектура спешит на помощь | 34 |
| 1.4.1. Куб масштабирования и микросервисы | 34 |
| 1.4.2. Микросервисы как разновидность модульности | 37 |
| 1.4.3. У каждого сервиса есть своя база данных..... | 38 |
| 1.4.4. Микросервисная архитектура для FTGO | 38 |
| 1.4.5. Сравнение микросервисной и сервис-ориентированной архитектур | 40 |
| 1.5. Достоинства и недостатки микросервисной архитектуры..... | 41 |
| 1.5.1. Достоинства микросервисной архитектуры | 41 |
| 1.5.2. Недостатки микросервисной архитектуры..... | 44 |
| 1.6. Язык шаблонов микросервисной архитектуры | 46 |
| 1.6.1. Микросервисная архитектура не панацея | 47 |
| 1.6.2. Шаблоны проектирования и языки шаблонов | 48 |
| 1.6.3. Обзор языка шаблонов микросервисной архитектуры..... | 51 |
| 1.7. Помимо микросервисов: процесс и организация..... | 58 |
| 1.7.1. Организация разработки и доставки программного обеспечения | 59 |
| 1.7.2. Процесс разработки и доставки программного обеспечения | 60 |
| 1.7.3. Человеческий фактор при переходе на микросервисы | 61 |
| Резюме..... | 62 |
| Глава 2. Стратегии декомпозиции | 63 |
| 2.1. Что представляет собой микросервисная архитектура..... | 64 |
| 2.1.1. Что такое архитектура программного обеспечения и почему она важна | 64 |
| 2.1.2. Обзор архитектурных стилей | 67 |
| 2.1.3. Микросервисная архитектура как архитектурный стиль | 70 |
| 2.2. Определение микросервисной архитектуры приложения..... | 74 |
| 2.2.1. Определение системных операций..... | 76 |
| 2.2.2. Разбиение на сервисы по бизнес-возможностям | 82 |
| 2.2.3. Разбиение на сервисы по проблемным областям | 85 |
| 2.2.4. Методические рекомендации по декомпозиции..... | 87 |
| 2.2.5. Трудности при разбиении приложения на сервисы | 88 |
| 2.2.6. Определение API сервисов | 92 |
| Резюме..... | 95 |

| | |
|---|-----|
| Глава 3. Межпроцессное взаимодействие в микросервисной архитектуре | 97 |
| 3.1. Обзор межпроцессного взаимодействия в микросервисной архитектуре | 98 |
| 3.1.1. Стили взаимодействия..... | 99 |
| 3.1.2. Описание API в микросервисной архитектуре | 100 |
| 3.1.3. Развивающиеся API | 101 |
| 3.1.4. Форматы сообщений..... | 103 |
| 3.2. Взаимодействие на основе удаленного вызова процедур | 105 |
| 3.2.1. Использование REST..... | 106 |
| 3.2.2. Использование gRPC..... | 109 |
| 3.2.3. Работа в условиях частичного отказа с применением шаблона «Предохранитель»..... | 111 |
| 3.2.4. Обнаружение сервисов | 114 |
| 3.3. Взаимодействие с помощью асинхронного обмена сообщениями..... | 119 |
| 3.3.1. Обзор механизмов обмена сообщениями | 119 |
| 3.3.2. Реализация стилей взаимодействия с помощью сообщений | 122 |
| 3.3.3. Создание спецификации для API сервиса на основе сообщений..... | 124 |
| 3.3.4. Использование брокера сообщений | 125 |
| 3.3.5. Конкурирующие получатели и порядок следования сообщений | 129 |
| 3.3.6. Дублирование сообщений..... | 130 |
| 3.3.7. Транзакционный обмен сообщениями..... | 132 |
| 3.3.8. Библиотеки и фреймворки для обмена сообщениями..... | 136 |
| 3.4. Использование асинхронного обмена сообщениями для улучшения доступности | 139 |
| 3.4.1. Синхронное взаимодействие снижает степень доступности | 139 |
| 3.4.2. Избавление от синхронного взаимодействия | 141 |
| Резюме | 144 |
| Глава 4. Управление транзакциями с помощью повествований | 146 |
| 4.1. Управление транзакциями в микросервисной архитектуре | 147 |
| 4.1.1. Микросервисная архитектура и необходимость в распределенных транзакциях..... | 148 |
| 4.1.2. Проблемы с распределенными транзакциями | 148 |
| 4.1.3. Использование шаблона «Повествование» для сохранения согласованности данных..... | 150 |
| 4.2. Координация повествований | 154 |
| 4.2.1. Повествования, основанные на хореографии..... | 154 |
| 4.2.2. Повествования на основе оркестрации | 159 |

| | | |
|--------|--|-----|
| 4.3. | Что делать с недостаточной изолированностью | 164 |
| 4.3.1. | Обзор аномалий..... | 165 |
| 4.3.2. | Контрмеры на случай нехватки изолированности | 166 |
| 4.4. | Архитектура сервиса Order и повествования Create Order | 170 |
| 4.4.1. | Класс OrderService | 172 |
| 4.4.2. | Реализация повествования Create Order | 173 |
| 4.4.3. | Класс OrderCommandHandlers | 181 |
| 4.4.4. | Класс OrderServiceConfiguration..... | 182 |
| | Резюме | 184 |

Глава 5. Проектирование бизнес-логики в микросервисной архитектуре..... 185

| | | |
|--------|---|-----|
| 5.1. | Шаблоны организации бизнес-логики | 186 |
| 5.1.1. | Проектирование бизнес-логики с помощью шаблона «Сценарий транзакции» | 188 |
| 5.1.2. | Проектирование бизнес-логики с помощью шаблона «Доменная модель» | 189 |
| 5.1.3. | О предметно-ориентированном проектировании..... | 190 |
| 5.2. | Проектирование доменной модели с помощью шаблона «Агрегат» из DDD | 191 |
| 5.2.1. | Проблемы с расплывчатыми границами | 192 |
| 5.2.2. | Агрегаты имеют четкие границы..... | 194 |
| 5.2.3. | Правила для агрегатов | 195 |
| 5.2.4. | Размеры агрегатов..... | 198 |
| 5.2.5. | Проектирование бизнес-логики с помощью агрегатов..... | 199 |
| 5.3. | Публикация доменных событий | 200 |
| 5.3.1. | Зачем публиковать события об изменениях | 200 |
| 5.3.2. | Что такое доменное событие | 201 |
| 5.3.3. | Обогащение события | 202 |
| 5.3.4. | Определение доменных событий | 202 |
| 5.3.5. | Генерация и публикация доменных событий..... | 204 |
| 5.3.6. | Потребление доменных событий..... | 207 |
| 5.4. | Бизнес-логика сервиса Kitchen | 208 |
| 5.4.1. | Агрегат Ticket | 210 |
| 5.5. | Бизнес-логика сервиса Order..... | 214 |
| 5.5.1. | Агрегат Order..... | 215 |
| 5.5.2. | Класс OrderService | 220 |
| | Резюме | 222 |

| | |
|---|-----|
| Глава 6. Разработка бизнес-логики с порождением событий | 223 |
| 6.1. Разработка бизнес-логики с использованием порождения событий..... | 224 |
| 6.1.1. Проблемы традиционного сохранения данных | 225 |
| 6.1.2. Обзор порождения событий..... | 227 |
| 6.1.3. Обработка конкурентных обновлений с помощью оптимистичного блокирования | 234 |
| 6.1.4. Порождение и публикация событий | 235 |
| 6.1.5. Улучшение производительности с помощью снимков..... | 236 |
| 6.1.6. Идемпотентная обработка сообщений | 238 |
| 6.1.7. Развитие доменных событий..... | 239 |
| 6.1.8. Преимущества порождения событий..... | 241 |
| 6.1.9. Недостатки порождения событий..... | 242 |
| 6.2. Реализация хранилища событий..... | 244 |
| 6.2.1. Принцип работы хранилища событий Eventuate Local | 245 |
| 6.2.2. Клиентский фреймворк Eventuate для Java | 248 |
| 6.3. Совместное использование повествований и порождения событий | 252 |
| 6.3.1. Реализация повествований на основе хореографии с помощью порождения событий | 253 |
| 6.3.2. Создание повествования на основе оркестрации | 254 |
| 6.3.3. Реализация участника повествования на основе порождения событий | 256 |
| 6.3.4. Реализация оркестраторов повествований с помощью порождения событий | 260 |
| Резюме | 262 |
| Глава 7. Реализация запросов в микросервисной архитектуре | 264 |
| 7.1. Выполнение запросов с помощью объединения API | 265 |
| 7.1.1. Запрос findOrder()..... | 265 |
| 7.1.2. Обзор шаблона «Объединение API»..... | 266 |
| 7.1.3. Реализация запроса findOrder() путем объединения API..... | 268 |
| 7.1.4. Архитектурные проблемы объединения API | 269 |
| 7.1.5. Преимущества и недостатки объединения API | 272 |
| 7.2. Применение шаблона CQRS..... | 273 |
| 7.2.1. Потенциальные причины использования CQRS | 274 |
| 7.2.2. Обзор CQRS | 277 |
| 7.2.3. Преимущества CQRS | 280 |
| 7.2.4. Недостатки CQRS | 281 |

| | | |
|-----------------|--|------------|
| 7.3. | Проектирование CQRS-представлений | 282 |
| 7.3.1. | Выбор хранилища данных для представления | 283 |
| 7.3.2. | Структура модуля доступа к данным..... | 285 |
| 7.3.3. | Добавление и обновление CQRS-представлений..... | 288 |
| 7.4. | Реализация CQRS с использованием AWS DynamoDB..... | 289 |
| 7.4.1. | Модуль OrderHistoryEventHandlers..... | 290 |
| 7.4.2. | Моделирование данных и проектирование запросов с помощью DynamoDB | 291 |
| 7.4.3. | Класс OrderHistoryDaoDynamoDb..... | 296 |
| | Резюме | 299 |
| Глава 8. | Шаблоны внешних API..... | 301 |
| 8.1. | Проблемы с проектированием внешних API..... | 302 |
| 8.1.1. | Проблемы проектирования API для мобильного клиента FTGO | 303 |
| 8.1.2. | Проблемы с проектированием API для клиентов другого рода | 306 |
| 8.2. | Шаблон «API-шлюз» | 307 |
| 8.2.1. | Обзор шаблона «API-шлюз»..... | 308 |
| 8.2.2. | Преимущества и недостатки API-шлюза | 315 |
| 8.2.3. | Netflix как пример использования API-шлюза..... | 316 |
| 8.2.4. | Трудности проектирования API-шлюза..... | 316 |
| 8.3. | Реализация API-шлюза | 320 |
| 8.3.1. | Использование готового API-шлюза | 320 |
| 8.3.2. | Разработка собственного API-шлюза..... | 322 |
| 8.3.3. | Реализация API-шлюза с помощью GraphQL..... | 329 |
| | Резюме | 341 |
| Глава 9. | Тестирование микросервисов, часть 1 | 343 |
| 9.1. | Стратегии тестирования микросервисных архитектур..... | 345 |
| 9.1.1. | Обзор методик тестирования | 345 |
| 9.1.2. | Трудности тестирования микросервисов | 352 |
| 9.1.3. | Процесс развертывания | 358 |
| 9.2. | Написание модульных тестов для сервиса..... | 360 |
| 9.2.1. | Разработка модульных тестов для доменных сущностей | 363 |
| 9.2.2. | Написание модульных тестов для объектов значений..... | 364 |
| 9.2.3. | Разработка модульных тестов для повествований..... | 364 |
| 9.2.4. | Написание модульных тестов для доменных сервисов | 366 |

| | |
|--|-----|
| 9.2.5. Разработка модульных тестов для контроллеров | 368 |
| 9.2.6. Написание модульных тестов для обработчиков событий и сообщений..... | 370 |
| Резюме | 371 |
| Глава 10. Тестирование микросервисов, часть 2 | 373 |
| 10.1. Написание интеграционных тестов..... | 374 |
| 10.1.1. Интеграционные тесты с сохранением | 376 |
| 10.1.2. Интеграционное тестирование взаимодействия в стиле «запрос/ответ» на основе REST | 378 |
| 10.1.3. Интеграционное тестирование взаимодействия в стиле «издатель/подписчик» | 382 |
| 10.1.4. Интеграционные тесты контрактов для взаимодействия на основе асинхронных запросов/ответов..... | 386 |
| 10.2. Разработка компонентных тестов | 391 |
| 10.2.1. Определение приемочных тестов..... | 392 |
| 10.2.2. Написание приемочных тестов с помощью Gherkin | 392 |
| 10.2.3. Проектирование компонентных тестов..... | 395 |
| 10.2.4. Написание компонентных тестов для сервиса Order..... | 396 |
| 10.3. Написание сквозных тестов..... | 401 |
| 10.3.1. Проектирование сквозных тестов | 402 |
| 10.3.2. Написание сквозных тестов | 402 |
| 10.3.3. Выполнение сквозных тестов | 403 |
| Резюме | 403 |
| Глава 11. Разработка сервисов, готовых к промышленному использованию..... | 405 |
| 11.1. Разработка безопасных сервисов | 406 |
| 11.1.1. Обзор безопасности в традиционном монолитном приложении | 407 |
| 11.1.2. Обеспечение безопасности в микросервисной архитектуре | 411 |
| 11.2. Проектирование конфигурируемых сервисов | 420 |
| 11.2.1. Вынесение конфигурации вовне с помощью пассивной модели | 421 |
| 11.2.2. Вынесение конфигурации вовне с помощью активной модели | 423 |
| 11.3. Проектирование наблюдаемых сервисов | 424 |
| 11.3.1. Использование API проверки работоспособности..... | 426 |
| 11.3.2. Применение шаблона агрегации журналов | 428 |
| 11.3.3. Использование шаблона распределенной трассировки | 430 |
| 11.3.4. Применение шаблона «Показатели приложения» | 434 |

| | |
|--|------------|
| 11.3.5. Шаблон отслеживания исключений | 437 |
| 11.3.6. Применение шаблона «Ведение журнала аудита» | 439 |
| 11.4. Разработка сервисов с помощью шаблона микросервисного шасси | 440 |
| 11.4.1. Использование шасси микросервисов | 441 |
| 11.4.2. От микросервисного шасси до сети сервисов | 442 |
| Резюме | 444 |
| Глава 12. Развёртывание микросервисов | 446 |
| 12.1. Развёртывание сервисов с помощью пакетов для отдельных языков..... | 449 |
| 12.1.1. Преимущества использования пакетов для конкретных языков | 452 |
| 12.1.2. Недостатки применения пакетов для конкретных языков..... | 452 |
| 12.2. Развёртывание сервисов в виде виртуальных машин | 454 |
| 12.2.1. Преимущества развертывания сервисов в виде ВМ..... | 456 |
| 12.2.2. Недостатки развертывания сервисов в виде ВМ..... | 457 |
| 12.3. Развёртывание сервисов в виде контейнеров..... | 458 |
| 12.3.1. Развёртывание сервисов с помощью Docker..... | 460 |
| 12.3.2. Преимущества развертывания сервисов в виде контейнеров | 463 |
| 12.3.3. Недостатки развертывания сервисов в виде контейнеров..... | 463 |
| 12.4. Развёртывание приложения FTGO с помощью Kubernetes..... | 463 |
| 12.4.1. Обзор Kubernetes..... | 464 |
| 12.4.2. Развёртывание сервиса Restaurant в Kubernetes..... | 467 |
| 12.4.3. Развёртывание API-шлюза | 470 |
| 12.4.4. Развёртывание без простоя | 471 |
| 12.4.5. Использование сети сервисов для отделения развертывания от выпуска..... | 472 |
| 12.5. Бессерверное развертывание сервисов | 481 |
| 12.5.1. Обзор бессерверного развертывания с помощью AWS Lambda..... | 482 |
| 12.5.2. Написание лямбда-функции..... | 483 |
| 12.5.3. Вызов лямбда-функций..... | 484 |
| 12.5.4. Преимущества использования лямбда-функций | 485 |
| 12.5.5. Недостатки использования лямбда-функций..... | 485 |
| 12.6. Развёртывание RESTful-сервиса с помощью AWS Lambda и AWS Gateway | 486 |
| 12.6.1. Архитектура сервиса Restaurant на основе AWS Lambda..... | 487 |
| 12.6.2. Упаковывание сервиса в виде ZIP-файла | 491 |
| 12.6.3. Развёртывание лямбда-функций с помощью бессерверного фреймворка..... | 492 |
| Резюме | 493 |

| | |
|--|-----|
| Глава 13. Процесс перехода на микросервисы | 495 |
| 13.1. Переход на микросервисы..... | 496 |
| 13.1.1. Зачем переходить с монолита на что-то другое | 496 |
| 13.1.2. «Удушение» монолита | 497 |
| 13.2. Стратегии перехода с монолита на микросервисы | 501 |
| 13.2.1. Реализация новых возможностей в виде сервисов | 501 |
| 13.2.2. Разделение уровня представления и внутренних компонентов | 503 |
| 13.2.3. Извлечение бизнес-возможностей в сервисы | 505 |
| 13.3. Проектирование взаимодействия между сервисом и монолитом..... | 512 |
| 13.3.1. Проектирование интеграционного слоя | 513 |
| 13.3.2. Обеспечение согласованности данных между сервисом и монолитом | 518 |
| 13.3.3. Аутентификация и авторизация | 523 |
| 13.4. Реализация новой возможности в виде сервиса | 525 |
| 13.4.1. Архитектура сервиса Delayed Delivery | 526 |
| 13.4.2. Проектирование интеграционного слоя для сервиса Delayed Order..... | 528 |
| 13.5. Разбиение монолита на части: извлечение управления доставкой | 530 |
| 13.5.1. Обзор возможностей существующего механизма управления доставкой | 530 |
| 13.5.2. Обзор сервиса Delivery..... | 532 |
| 13.5.3. Проектирование доменной модели сервиса Delivery..... | 533 |
| 13.5.4. Структура интеграционного слоя для сервиса Delivery | 536 |
| 13.5.5. Изменение монолита для взаимодействия с сервисом Delivery | 538 |
| Резюме | 541 |

Проектирование бизнес-логики в микросервисной архитектуре

В этой главе

- Применение шаблонов организации бизнес-логики, таких как сценарий транзакции и доменная модель.
- Проектирование бизнес-логики с помощью шаблона «Агрегат» (предметно-ориентированное проектирование).
- Применение шаблона «Доменное событие» в микросервисной архитектуре.

Сердцем промышленных приложений является бизнес-логика, которая реализует бизнес-правила. Разработка сложной бизнес-логики всегда сопряжена с определенными трудностями. Приложение FTGO реализует довольно замысловатую бизнес-логику, особенно для управления заказами и доставкой. Мэри поощряла свою команду применять принципы объектно-ориентированного проектирования, поскольку, исходя из ее опыта, это лучший способ реализации сложной бизнес-логики. На некоторых участках приложения использовался процедурный шаблон «Сценарий транзакции». Но большая часть кода была реализована в соответствии с объектно-ориентированной доменной моделью, которая накладывалась на базу данных с помощью JPA.

В микросервисной архитектуре разрабатывать сложную бизнес-логику оказывается еще труднее, потому что она распределена между разными микросервисами. Вам необходимо решить две ключевые проблемы. Типичная доменная модель выглядит как паутина из связанных между собой классов. В монолитных приложениях в этом нет ничего плохого, но в микросервисной архитектуре, где классы разбросаны по разным сервисам, нужно избавиться от ссылок на объекты, которые пересекают

границы сервисов. Еще одна проблема заключается в проектировании бизнес-логики, которая работает в рамках ограничений, накладываемых работой с транзакциями в микросервисной архитектуре. Вы можете применять ACID-транзакции внутри одного сервиса, но, как говорилось в главе 4, для обеспечения согласованности данных между сервисами следует использовать шаблон «Повествование».

К счастью, для преодоления этих трудностей можно воспользоваться шаблоном «Агрегат» из состава DDD. Он структурирует бизнес-логику приложения в виде набора агрегатов. *Агрегат* — это кластер объектов, с которыми можно обращаться как с единым целым. Есть две причины, почему агрегаты могут пригодиться при разработке бизнес-логики в микросервисной архитектуре.

- Агрегаты исключают любую возможность того, что ссылки на объекты могут выйти за рамки одного сервиса, потому что межагрегатная ссылка — это скорее значение первичного ключа, а не объектная ссылка.
- Транзакция может создать или обновить лишь один агрегат, поэтому агрегаты соответствуют ограничениям транзакционной модели микросервисов.

Благодаря этому ACID-транзакция никогда не выйдет за пределы одного сервиса.

Я начну эту главу с описания разных способов организации бизнес-логики — шаблонов «Сценарий транзакции» и «Доменная модель». Затем вы познакомитесь с концепцией агрегатов из DDD и узнаете, почему они являются хорошими строительными блоками для бизнес-логики сервисов. После этого я опишу шаблон «Доменная модель» и объясню, почему сервису следует публиковать свои события. В конце главы будут представлены несколько примеров бизнес-логики из сервисов *Kitchen* и *Order*.

Рассмотрим шаблоны организации бизнес-логики.

5.1. Шаблоны организации бизнес-логики

На рис. 5.1 показана архитектура типичного сервиса. Как говорилось в главе 2, бизнес-логика является ядром шестигранной архитектуры. Ее окружают входящие и исходящие адаптеры. *Входящий адаптер* обрабатывает запросы от клиентов и вызывает бизнес-логику. *Исходящий адаптер*, который сам вызывается бизнес-логикой, обращается к другим сервисам и приложениям.

Этот сервис состоит из бизнес-логики и следующих адаптеров:

- *адаптера REST API* — входящего адаптера, который реализует REST API для вызова бизнес-логики;
- *OrderCommandHandlers* — входящего адаптера, который потребляет из канала командные сообщения и вызывает бизнес-логику;
- *адаптера базы данных* — исходящего адаптера, который вызывается бизнес-логикой для доступа к базе данных;
- *адаптера публикации доменных событий* — исходящего адаптера, который публикует события для брокера сообщений.

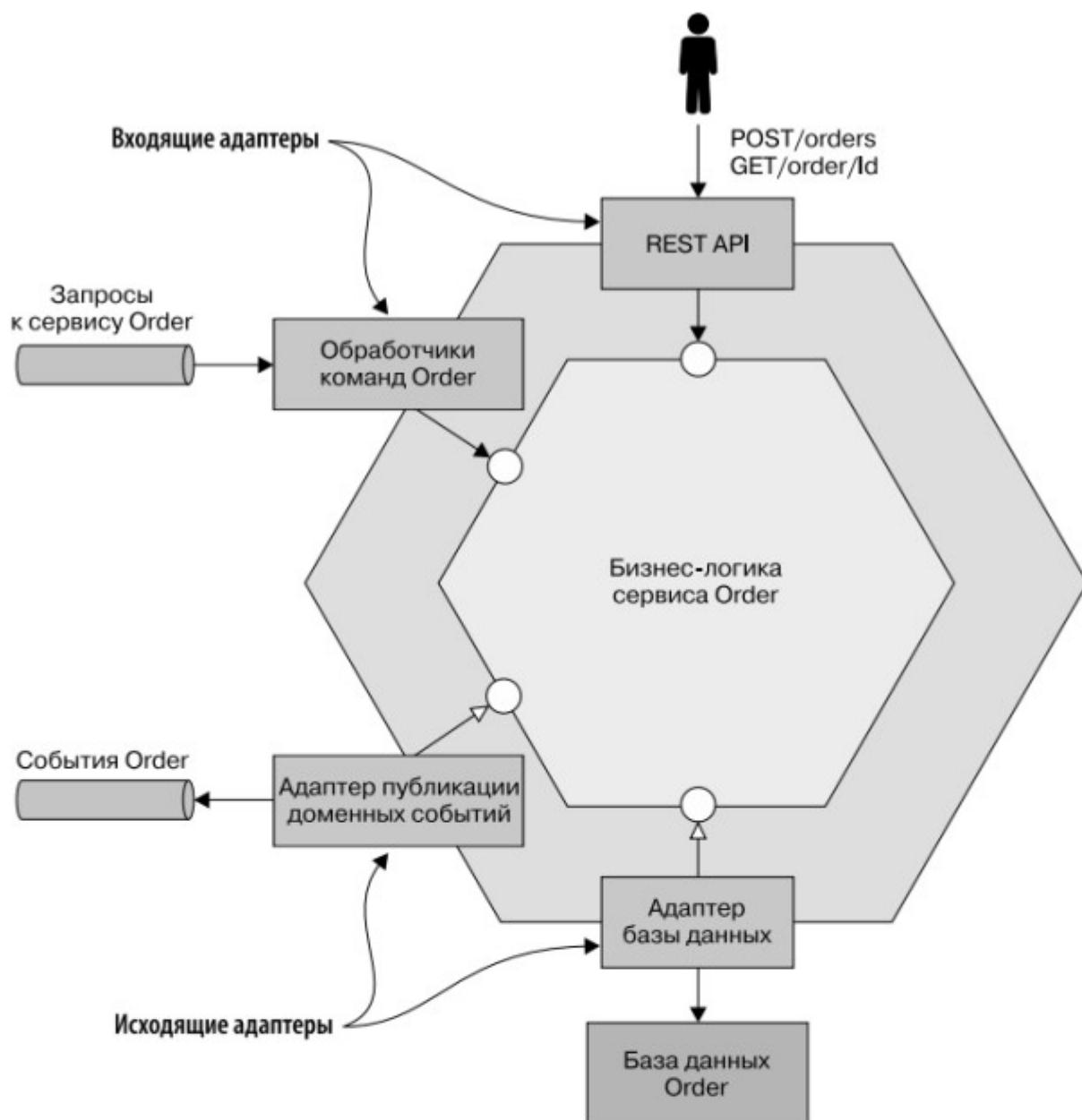


Рис. 5.1. Сервис Order имеет шестигранную архитектуру. Он состоит из бизнес-логики и одного или нескольких адаптеров для доступа к внешним приложениям или другим сервисам

Бизнес-логика обычно оказывается самой сложной частью сервиса. Вы должны осознанно организовать ее таким образом, который лучше всего подходит для вашего приложения. Я уверен, что вам уже знакомо разочарование от поддержки плохо структурированного кода, написанного кем-то другим. Большинство приложений уровня предприятия написаны на объектно-ориентированных языках, таких как Java, поэтому они состоят из классов и методов. Но использование объектно-ориентированного языка вовсе не означает, что бизнес-логика имеет объектно-ориентированную структуру. Ключевое решение, которое вам придется принять в ходе разработки, заключается в том, какой подход лучше применять — объектно-ориентированный или процедурный. Существует два основных шаблона для организации бизнес-логики: процедурный «Сценарий транзакции» и объектно-ориентированный, который называется «Доменная модель».

5.1.1. Проектирование бизнес-логики с помощью шаблона «Сценарий транзакции»

Я большой сторонник объектно-ориентированного подхода, но в некоторых ситуациях он излишен — например, при разработке простой бизнес-логики. В таких случаях лучше писать процедурный код, используя шаблон, который Мартин Фаулер в своей книге *Patterns of Enterprise Application Architecture* (Addison-Wesley Professional, 2002)¹ называет сценарием транзакции. Вместо объектно-ориентированного проектирования вы создаете метод под названием «Сценарий транзакции», который обрабатывает запросы из уровня представления. Важная характеристика этого подхода — то, что классы, реализующие поведение, отделены от классов, которые хранят состояние (рис. 5.2).

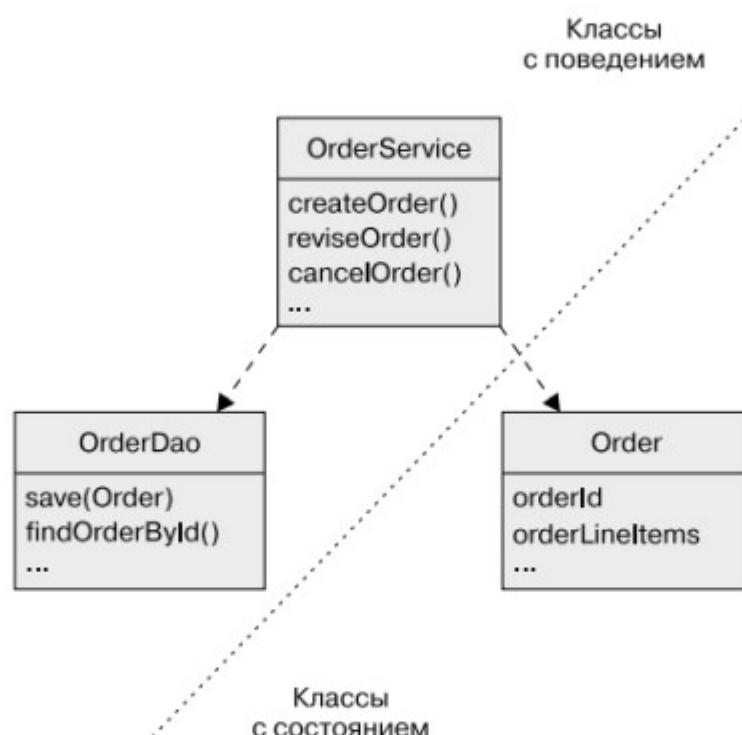


Рис. 5.2. Организация бизнес-логики в виде сценариев транзакции. В такой архитектуре один набор классов обычно реализует поведение, а другой хранит состояние. Сценарии транзакции организованы в виде классов, у которых нет состояния. Они применяют классы данных, которые, как правило, не обладают поведением

При использовании этого шаблона сценарии обычно размещаются в классе сервиса (в данном случае `OrderService`). Класс сервиса имеет по одному методу для каждого запроса или системной операции. Метод реализует бизнес-логику для определенного запроса. Он обращается к БД с помощью объектов доступа к данным (data access object, DAO), таких как `OrderDao`. Здесь примером такого объекта является класс `Order`, он предназначен исключительно для работы с данными, и у него почти (или совсем) нет никакого поведения.

¹ Фаулер М. Шаблоны корпоративных приложений. — М.: Вильямс, 2016.

Шаблон «Сценарий транзакции»

Организует бизнес-логику в виде набора процедурных сценариев транзакций, по одному для каждого типа запросов.

Этот стиль проектирования в основном является процедурным, но при этом использует несколько возможностей объектно-ориентированных языков программирования. Вы бы с помощью этого подхода писали программы на С и других языках без поддержки объектно-ориентированного программирования (ООП). Тем не менее в процедурном проектировании, если оно применяется в подходящей ситуации, нет ничего постыдного. Оно хорошо подходит для простой бизнес-логики, но сложную логику с его помощью лучше не реализовывать.

5.1.2. Проектирование бизнес-логики с помощью шаблона «Доменная модель»

Простота процедурного подхода может показаться довольно соблазнительной. Вы можете писать код без тщательного продумывания организации классов. Но проблема в том, что при довольно значительном усложнении бизнес-логики поддержка вашего кода превратится в сплошной кошмар. Как и монолитные приложения, сценарии транзакций склонны постоянно разрастаться. Поэтому, если ваше приложение не является предельно простым, следует воздерживаться от написания процедурного кода. Вместо этого стоит применять доменную модель и вести разработку в объектно-ориентированном стиле.

Шаблон «Доменная модель»

Организует бизнес-логику в виде объектной модели, состоящей из классов с состоянием и поведением.

В объектно-ориентированном проектировании бизнес-логика состоит из объектной модели — сети относительно небольших классов. Эти классы обычно напрямую соотносятся с концепциями из проблемной области. В такой архитектуре некоторые классы обладают либо состоянием, либо поведением, но многие имеют и то и другое, что является признаком хорошо спроектированного класса. Пример шаблона «Доменная модель» показан на рис. 5.3.

Как и в случае с шаблоном «Сценарий транзакции», у класса `OrderService` предусмотрено по одному методу для каждого запроса или системной операции. Но при использовании доменной модели методы сервисов обычно получаются более простыми. Это связано с тем, что существенная часть бизнес-логики делегируется доменным объектам. Метод сервиса может, например, извлечь доменный объект из базы данных и вызвать один из его собственных методов. В этом примере класс

`Order` обладает как состоянием, так и поведением. Более того, его состояние является приватным, и доступ к нему осуществляется лишь через его методы.

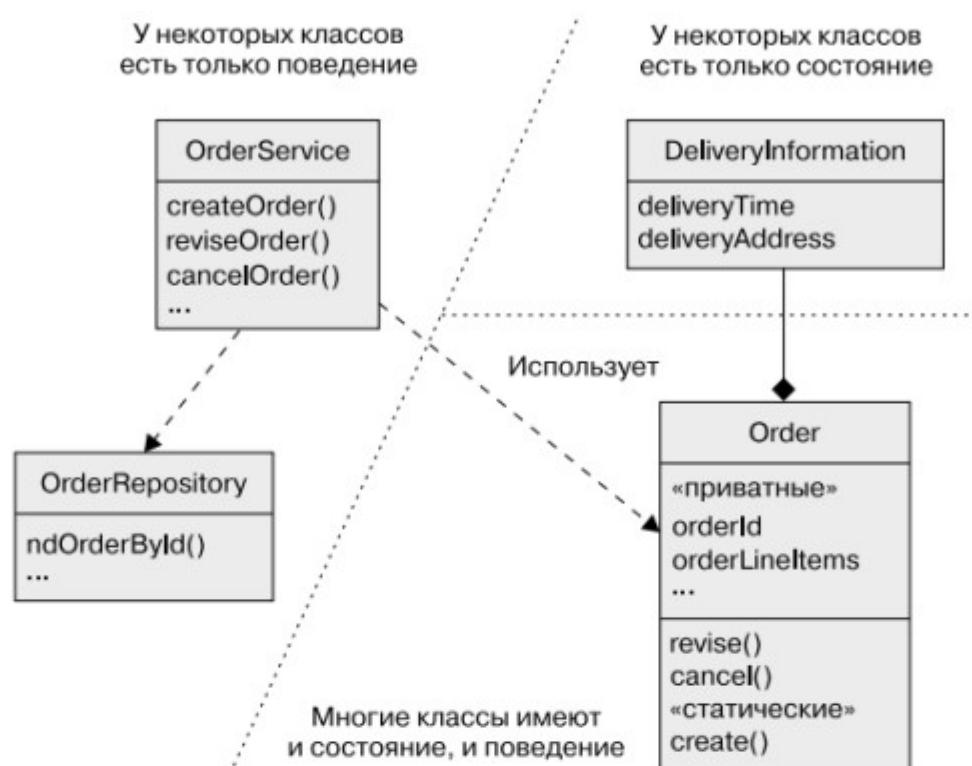


Рис. 5.3. Организация бизнес-логики в виде доменной модели. Большинство классов имеют и состояние, и поведение

Применение объектно-ориентированного проектирования имеет ряд преимуществ. Во-первых, это упрощает понимание и поддержку архитектуры. Вместо одного большого класса, который берет на себя все функции, сервис состоит из нескольких мелких классов, у каждого из которых есть свой небольшой набор обязанностей. Кроме того, такие классы, как `Account`, `BankingTransaction` и `OverdraftPolicy`, довольно точно отражают реальный мир, благодаря чему их роль в архитектуре проще понять. Во-вторых, нашу объектно-ориентированную архитектуру легче тестировать: каждый класс может и должен тестироваться отдельно. И наконец, объектно-ориентированный код проще расширять, поскольку в нем можно использовать хорошо известные шаблоны проектирования, такие как «Стратегия» и «Шаблонный метод», которые позволяют расширять компонент без изменения его кода.

Шаблон «Доменная модель» хорошо себя зарекомендовал, но у него есть целый ряд проблем, особенно в контексте микросервисной архитектуры. Чтобы разобраться с ними, нужно использовать более узкую версию ООП, известную как DDD.

5.1.3. О предметно-ориентированном проектировании

Предметно-ориентированное проектирование (DDD), описанное в книге Эрика Эванса *Domain-Driven Design*, – это более узкая разновидность ООП, предназначенная для разработки сложной бизнес-логики. Мы познакомились с DDD в главе 2 при обсуждении поддоменов и того, насколько они подходят для разбиения при-

ложений на сервисы. В DDD каждый сервис имеет собственную доменную модель, что позволяет избежать проблем с единой доменной моделью, которая охватывает все приложение. Поддомены и связанная с ними концепция изолированного контекста — это два стратегических шаблона DDD.

В DDD есть также тактические шаблоны, которые служат строительными блоками для доменных моделей. Каждый шаблон представляет собой роль, которую класс играет в доменной модели, и описывает характеристики этого класса. Разработчики широко применяют следующие строительные блоки.

- *Сущность* — объект, обладающий устойчивой идентичностью. Две сущности, чьи атрибуты имеют одинаковые значения, — это все равно разные объекты. В приложении Java EE классы, которые сохраняются с помощью аннотации `@Entity` из JPA, обычно представляют собой сущности DDD.
- *Объект значений* — объект, представляющий собой набор значений. Два объекта значений с одинаковыми атрибутами взаимозаменяемы. Примером таких объектов может служить класс `Money`, который состоит из валюты и суммы.
- *Фабрика* — объект или метод, реализующий логику создания объектов, которую ввиду ее сложности не следует размещать прямо в конструкторе. Фабрика также может скрывать конкретные классы, экземпляры которых создает. Она реализуется в виде статического метода или класса.
- *Репозиторий* — объект, предоставляющий доступ к постоянным сущностям и инкапсулирующий механизм доступа к базе данных.
- *Сервис* — объект, реализующий бизнес-логику, которой не место внутри сущности или объекта значений.

Многие разработчики используют эти строительные блоки. Некоторые из них поддерживаются такими фреймворками, как JPA и Spring. Но есть еще одна концепция, которую обычно игнорируют все (и я тоже!), за исключением истинных ценителей DDD. Речь идет об агрегатах. Несмотря на свою непопулярность, этот строительный блок чрезвычайно полезен при разработке микросервисов. Давайте рассмотрим некоторые неочевидные проблемы классического ООП, которые можно решить с помощью агрегатов.

5.2. Проектирование доменной модели с помощью шаблона «Агрегат» из DDD

В традиционном объектно-ориентированном проектировании доменная модель описывает набор классов и отношения между ними. Классы обычно сгруппированы в пакеты. Например, на рис. 5.4 показана часть доменной модели приложения FTGO. Это типичная доменная модель, представляющая собой паутину взаимосвязанных классов.

Этот пример содержит несколько классов, которые соотносятся с бизнес-объектами: `Consumer`, `Order`, `Restaurant` и `Courier`. Но что интересно, в традиционной доменной модели не хватает четких границ между разными бизнес-объектами.