PRO ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

bhv®

Проектирование информационных систем

Профессиональный подход

- Архитектурные принципы и шаблоны
- Проектирование программных систем
- Тестирование
- Развертывание
- Автоматизация процессов
- Балансировка нагрузки
- Анализ изменяющихся требований



Владимир Туров

Владимир Туров

Проектирование информационных систем

Профессиональный подход

Санкт-Петербург
«БХВ-Петербург»
2025

УДК 004.41
ББК 32.973.26-018.1
Т88

Туров В. П.

Т88 Проектирование информационных систем. Профессиональный подход. — СПб.: БХВ-Петербург, 2025. — 208 с.: ил. — (Профессиональное программирование)

ISBN 978-5-9775-2068-3

Рассмотрены подходы и технологии, применяющиеся для проектирования информационных систем и подбора инструментов с учетом развития и поддержки программных продуктов. Разобрана теория развертывания, тестирования, автоматизации, балансировки нагрузки и поддержка новых возможностей в развивающемся продукте. Материал теоретической части рассмотрен на основе гипотетической информационной системы для сказочных персонажей. В практической части описанные теоретические концепции применяются в рамках реализации технического задания и в условиях ограниченности ресурсов. Среди рассмотренных примеров — разработка телеграм-бота, интернет-магазина и плагина к редактору для 3D-моделирования.

Для программистов, бизнес-аналитиков, архитекторов IT-решений

УДК 004.41
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта	*Олег Сивченко*
Зав. редакцией	*Людмила Гауль*
Редактор	*Григорий Добин*
Компьютерная верстка	*Ольги Сергиенко*
Дизайн обложки	*Зои Канторович*

"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20

ISBN 978-5-9775-2068-3

© Туров В. П., 2025
© Оформление. ООО "БХВ-Петербург", ООО "БХВ", 2025

Оглавление

Предисловие **9**

ЧАСТЬ I. ТЕОРИЯ ПРОЕКТИРОВАНИЯ **11**

Глава 1. Методологии разработки ПО **13**
Водопадная модель 13
Спиральная модель 15
Rational Unified Process 16
Язык моделирования UML 18
Сущности UML 18
Структурные сущности UML 18
Поведенческие и дополнительные сущности UML 19
Комментирующие сущности UML 19
Отношения UML 19
Диаграммы UML 20
Диаграмма классов 20
Диаграмма состояний 21
Диаграмма развертывания 21
Диаграмма прецедентов 22
Диаграмма последовательности 22
Гибкие методологии 23
Scrum 24
Feature-Driven Development 26

Глава 2. Выяснение требований **29**
Функциональные требования 30
Нефункциональные требования 31

Глава 3. Архитектурные шаблоны и стили **33**
Монолитная архитектура 33
Микросервисная архитектура 34
Микроядерная архитектура 35
Событийно-ориентированная архитектура 36
Многоуровневая архитектура 37

Representational State Transfer (REST)....38
Распределенная транзакция....39
Протокол двухфазного согласования (2 Phase Commit, 2PC)....40
Протокол трехфазного согласования (3 Phase Commit, 3PC)....41
Оркестрация....41
Хореография....42
Паттерн SAGA....42

Глава 4. Принципы разработки ПО....45
Принцип KISS....45
Принцип YAGNI....47
Принцип SOLID....47
S — Single Response Principle (принцип единой ответственности)....47
O — Open-Closed Principle (принцип открытости/закрытости)....48
L — Liskov substitution principle (принцип подстановки Барбары Лисков)....49
I — interface segregation principle (принцип разделения интерфейса)....51
D — dependency inversion principle (принцип инверсии зависимостей)....52
Инверсия управления....53
Внедрение зависимостей....54

Глава 5. Технологии....57
Системы управления базами данных....57
Реляционная модель данных....58
Столбцово-ориентированная модель....60
Документо-ориентированная модель....60
Графовая модель....61
Ключ-значение....61
Базы данных для временны́х рядов....62
Брокер сообщений....62
Параллельные вычисления....64
Закон Амдала....64
Многопоточность....65
Синхронизация в многопоточных программах....65
Псевдопараллелизм....66
Сопрограммы (корутины)....67
Многопроцессорные вычисления....67
Межпроцессное взаимодействие....67
Распределенные вычисления....68
Синхронизация в распределенных системах....69
Вызов функций в распределенных системах....70
GraphQL....70
WebSocket....71
Кеширование....73
Кеширование при чтении (сквозное чтение, read-through)....73
Кеширование при записи (сквозная запись, write-through)....73
Чтение на стороне (Read aside)....73
Запись на стороне (Write aside)....75
Кеширование на опережение (Cache Ahead, Refresh Ahead)....75
Политики инвалидации....76

Идентификация, аутентификация и авторизация ... 77
Идентификация ... 77
Аутентификация ... 77
Авторизация ... 77

Глава 6. Тестирование ... 79
Выбор данных для тестирования ... 81
Юнит-тестирование ... 83
Интеграционное тестирование ... 83
Сквозное тестирование (end-to-end) ... 84
Полное тестирование ... 84
Ручное тестирование ... 85
Заглушки ... 86

Глава 7. Развертывание ... 87
Автоматизация развертывания ... 87
Непрерывная интеграция ... 88
Непрерывная доставка ... 88
Непрерывное развертывание ... 89
Масштабирование ... 91
Вертикальное масштабирование ... 91
Горизонтальное масштабирование ... 92
Выделенные и облачные серверы ... 92
Выделенный сервер ... 93
Облачный сервер ... 93
DBaaS (Database as a Service) ... 94
Объектное хранилище ... 94
Развертывание: контейнеризация ... 95
Технология chroot ... 96
Технология Jail ... 96
Технология Docker ... 97
Технология Kubernetes ... 99
Технология Nomad ... 100
Развертывание: обнаружение сервисов ... 101

Глава 8. Балансировка нагрузки ... 103
Алгоритмы балансировки ... 103
DNS ... 105
GeoDNS ... 106
Обратный прокси-сервер ... 107

Глава 9. Обслуживание ... 109
Логирование ... 109
Мониторинг ... 111
Оповещение ... 111
Анализаторы исходного кода ... 112
Сканеры уязвимостей ... 113
Аудит-логи ... 113
Аварийные игры ... 114

Миграция схемы и миграция данных ... 115
Версионирование ... 116

Глава 10. Отказоустойчивость, резервирование и надежность ... 119
Надежность системы ... 119
Отказоустойчивость ... 120
Резервирование ... 120
Резервное копирование ... 121
Схемы резервирования ... 122
Восстановление после катастроф ... 122

ЧАСТЬ II. ПРИМЕРЫ ПРОЕКТИРОВАНИЯ СИСТЕМ ... 125

Глава 11. Проектирование неинтерактивного редактора изображений ... 127
Исходные требования ... 127
Монолитный подход ... 127
Модульный подход ... 130
Внезапное требование ... 134
Рекомендуемые технологии ... 138
Распространение ... 138
Заключение ... 139

Глава 12. Проектирование текстового блога ... 141
Исходные требования ... 141
Быстрый минимальный вариант ... 142
Проектирование архитектуры ... 145
Рекомендуемые технологии ... 150
Безопасность ... 151
Аутентификация ... 152
Поиск ... 152
Масштабирование ... 153
Локализация контента ... 153
Логирование и мониторинг ... 155
Отслеживание популярности материалов ... 155
Рекомендательная система ... 156
Соблюдение локальных норм ... 156
Резервное копирование ... 156
Заключение ... 157

Глава 13. Проектирование сервиса конвертации файлов ... 159
Исходные требования ... 159
Проектирование архитектуры ... 159
Проектирование веб-сервера ... 163
Проектирование агента ... 164
Проектирование взаимодействия с инструментами конвертации ... 165
Проектирование фонового процесса ... 166
Выбор технологий ... 167
Ограничение анонимных пользователей ... 167
Запрет использования сервиса в качестве файлового хранилища ... 167

Ограничение количества действий с одного адреса ... 168
Запрет регистрации с фальшивых почтовых адресов ... 169
Ограничение по объему и форматам ... 169
Безопасность ... 170
Взаимодействие с платежными системами ... 171
Приватность данных ... 171
Пароль доступа ... 171
Шифрование данных ... 171
Логирование и мониторинг ... 173
Распределенная архитектура ... 174
Улучшение пользовательского опыта ... 175
Подробные статусы ... 175
Хеширование входных файлов ... 176
Определение некорректных файлов ... 177
Добавление новой функциональности ... 177
Заключение ... 177

Глава 14. Проектирование интернет-магазина в Telegram ... 179
Исходные требования ... 179
Анализ возможностей ... 180
Ограничения ... 182
Проектирование интерфейса бота ... 183
Проектирование зоны администратора ... 186
Проектирование архитектуры ... 188
Выбор технологий ... 190
Рассылка уведомлений ... 190
Горизонтальное масштабирование ... 192
Мониторинг ... 193
Дальнейшие улучшения ... 193
Заключение ... 194

Глава 15. Проектирование плагина для отправки проекта из программы 3D-моделирования ... 195
Исходные требования ... 195
Ограничения ... 195
Проектирование простой архитектуры ... 196
Проектирование составной архитектуры ... 198
Выбор технологий ... 199
Логирование ... 199
Потенциальные технические улучшения ... 200
Заключение ... 200

Заключение ... 201

Предметный указатель ... 203

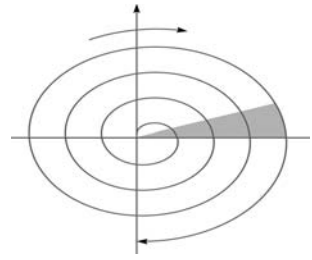
Предисловие

Проектирование информационных систем — важная тема, которая сопровождает разработчика с самых первых размышлений о создании нового проекта до последних минут существования этого проекта. Правильно выбранные инструменты и подходы позволят разрабатываемой системе гибко адаптироваться к изменяющимся условиям и требованиям бизнеса.

Эта книга состоит из двух частей: теоретической и практической. *Первая часть* предоставляет «сухой» обзор подходов и технологий с их положительными и отрицательными чертами. Во *второй части* демонстрируется разбор примеров с обсуждением, какие технологии стоит выбирать и почему.

При проектировании важно помнить, что разрабатываемая информационная система в первую очередь должна решать поставленную задачу и только во вторую очередь быть идеальной.

ЧАСТЬ I



Теория проектирования

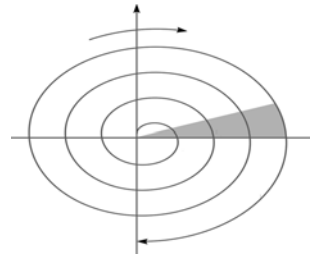
В главах этой части описываются различные подходы и технологии, используемые для разработки программного обеспечения. Для каждого подхода можно найти пример из реальной жизни, но использование разных информационных систем для разных задач не покажет справедливого сравнения. Более того, успешность или неуспешность программных решений будет негативно влиять на оценку подходов. У читателя может сложиться субъективное мнение: «в (крупной компании / успешном стартапе) используется такой подход, а потому это (отличное/ужасное) решение». Пресечем появление этого побочного эффекта в самом начале: выдумаем игрушечную информационную систему, на которую будем примерять все технологии и подходы.

Игрушечная информационная система должна решать настоящую проблему какого-нибудь игрушечного мира. Возьмем основной сюжет детских сказок: принцессу украл злой дракон, а король объявил награду за спасение — возможность жениться на дочери и полцарства в придачу. В этом простом сюжете можно найти или придумать интересы для каждой стороны:

- Король хочет спасти дочь. Или, может быть, хочет выдать ее замуж за достойного.
- Принцесса, возможно, не прочь сократить количество поклонников, оставив только самых смелых и решительных.
- Рыцари будут рады посоревноваться в силе между собой, а «доска объявлений» с возможными подвигами и наградами — это точно к рыцарям.
- Драконы могут отказаться от похищений принцесс и за достойное вознаграждение стать охранниками королевской сокровищницы. Или заняться наукой.

Способов автоматизации взаимодействий в сказке достаточно: от турнирной таблицы и доски объявлений до сложных процессов с организацией битв с драконами за право поцеловать спящую принцессу.

ГЛАВА 1



Методологии разработки ПО

В самом начале жизненного цикла программного обеспечения следует выбрать подход к ведению разработки. Составленный перечень ценностей, требований и бизнес-процессов позволяет задокументировать «контрольные точки» и признаки их достижения в жизненном цикле проекта.

Выбор методологии проектирования ПО не является обязательным для начала разработки программного обеспечения и может быть пропущен для индивидуальных домашних и академических проектов — таких как лабораторные и курсовые работы. Однако для разработки в командах очень рекомендуется выбрать методологию под свои цели.

Водопадная модель

Каскадная, или водопадная, модель (рис. 1.1) известна как минимум с 1956 года, но была формализована в 1970 году в статье американского ученого в области информатики У. Ройса[1], а свое название получила позднее.

Модель состоит из следующих последовательных шагов:

1. Планирование — разрабатывается план проекта, определяются сроки, бюджет, команды, распределяются обязанности.
2. Анализ требований — сбор и документация требований.
3. Проектирование — проектируются структуры данных, компоненты системы, их взаимодействие, пользовательский интерфейс.
4. Разработка — написание кода в соответствии с документацией, которая сгенерирована на предыдущих этапах.
5. Тестирование — проверка разработанного программного обеспечения на наличие дефектов и их устранение.

[1] См. **https://goo.su/u8kDs**.

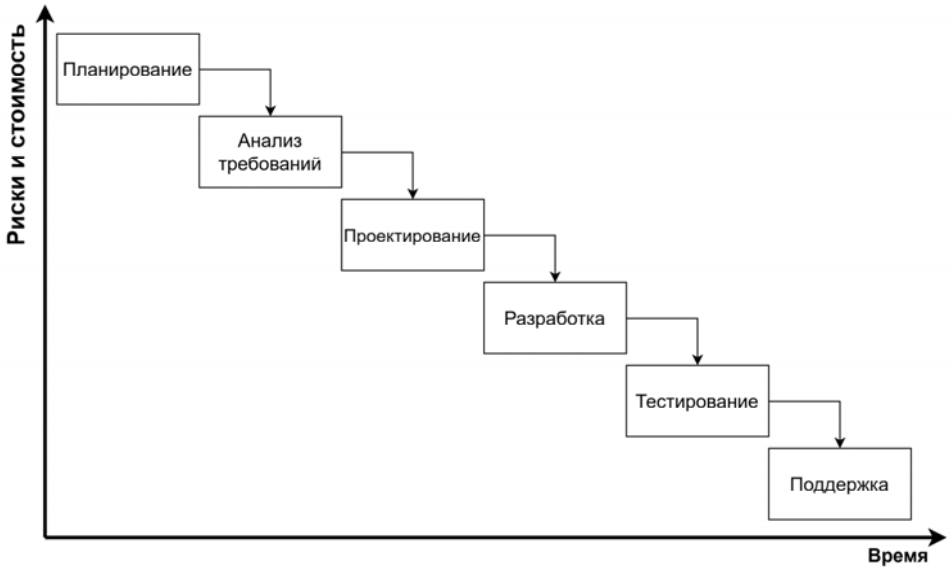


Рис. 1.1. Водопадная модель

6. Поддержка — оказание технической поддержки, исправление ошибок, обновления.

К характеристикам водопадной модели можно отнести:

- Легкость управления проектом.

  Модель определяет фиксированное количество этапов и имеет строгое ограничение — к следующему этапу можно приступать только после завершения всех работ на текущем этапе.

- Фиксированные требования.

  Требования к информационной системе собираются и фиксируются в самом начале жизненного цикла, что упрощает разработку. Это же является минусом — внесение изменений в требования на поздних этапах разработки затруднительно. Заказчик разработки привлекается два раза: в начале, при сборе требований, и в конце, при демонстрации разработанного программного продукта.

- Исчерпывающая документация.

  Процесс разработки разделен на этапы, на каждом этапе генерируется документация, описывающая проект с разных сторон. Помимо непосредственного документирования процесса, это добавляет прозрачности ходу разработки.

- Длительный цикл разработки.

  Строгое требование к последовательному выполнению этапов и необходимость составления исчерпывающей документации делает процесс разработки долгим.

Водопадная (каскадная) модель подходит для ситуаций, когда есть фиксированный набор требований, а заинтересованные лица хотят минимизировать свое взаимодействие с процессом разработки.

В терминах нашего сказочного мира каскадная модель подошла бы для разработки турнирной системы для рыцарей. Король сказал «хочу», рыцари поделились правилами проведения турниров, и мы приступили к разработке. Король занимается своими королевскими делами, а турнирные правила следующее тысячелетие меняться не будут. Как разработали систему — так всем и показали.

Спиральная модель

Спиральная модель (рис. 1.2), предложенная американским специалистом в области экономики программного обеспечения Барри Боэмом в 1986 году[1], сочетает этапы и итеративность.

Разработка модели происходит итеративно, каждая итерация разбита на четыре этапа:

1. Определение целей — собираются и документируются требования и цели. Заинтересованные лица могут высказать свои пожелания.

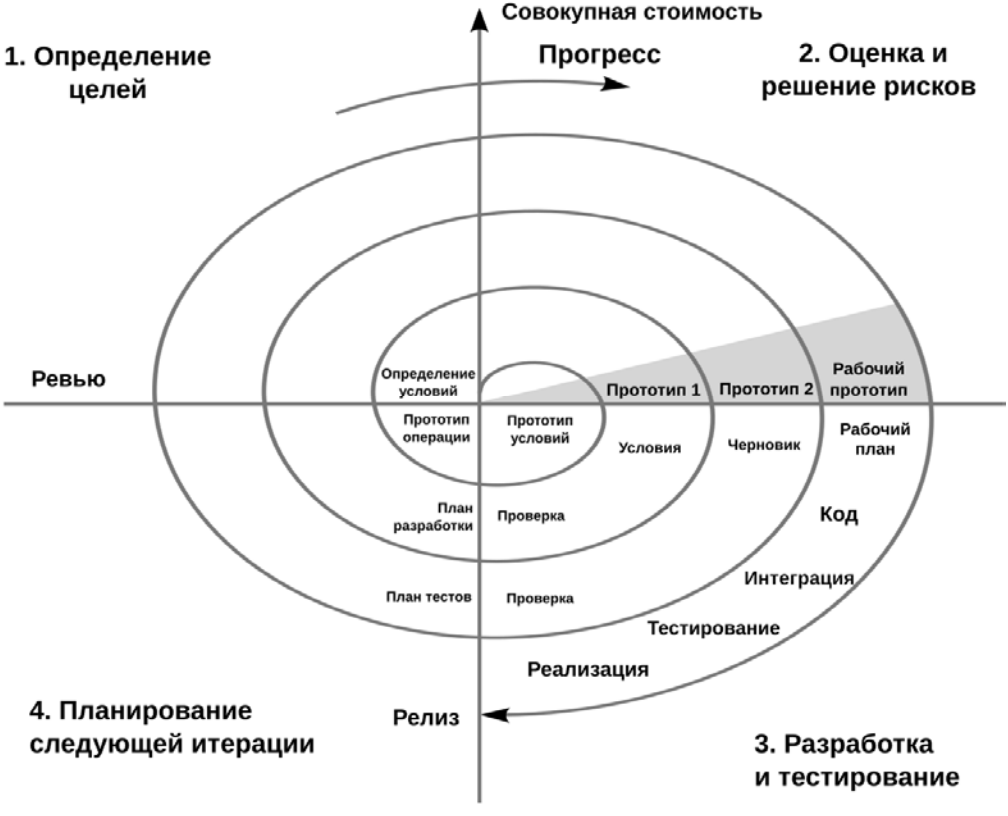


Рис. 1.2. Спиральная модель

[1] См. **https://goo.su/Dmslx**.

2. Оценка и решение рисков — определение и приоритизация рисков. Создание стратегий смягчения возможных последствий наступления рисков.
3. Разработка и тестирование — написание кода и тестирование.
4. Обзор и планирование следующей итерации — Демонстрация результатов заинтересованным лицам.

К характеристикам спиральной модели можно отнести:

- Работу с рисками.

  Это основное отличие спиральной модели. На каждой итерации уделяется время определению рисков — событий, которые могут помешать достижению целей, и выработке способов их смягчения.

- Возможность работать с нечеткими требованиями.

  Итеративность спиральной модели позволяет раз за разом возвращаться к этапу формирования требований, а в конце каждой итерации показывать результат заказчику. Это позволяет гибко адаптироваться к изменяющимся требованиям. Тем не менее постоянные изменения требований противоречат фиксированным срокам, из-за чего процесс разработки может оказаться бесконечно долгим.

- Пригодность для крупных проектов.

  Итеративность разработки позволяет сделать минимальный рабочий продукт, который можно выпустить раньше и впоследствии дорабатывать. Таким образом, итеративная модель разработки подходит в основном для крупных проектов, потому что она сложнее и требует опыта в работе с рисками.

В сказочном мире эта модель могла быть применена, если требование короля — создать под его чутким контролем социальную сеть, где будут турниры рыцарей, доски объявлений, реестр драконов и т. д. На первой итерации тогда можно сделать турнирную систему, чтобы выбрать лучшего рыцаря и отправить его спасать принцессу. На второй — добавить в турнир магов. На третьей — пересмотреть турнирные правила. Итеративная модель справится с хаотичными требованиями, главное — заинтересованность заказчика.

Rational Unified Process

Rational Unified Process (RUP) — методология разработки программного обеспечения, созданная компанией Rational Software. Согласно RUP процесс разработки (рис. 1.3) разбит на четыре фазы: «Начало», «Уточнение», «Конструирование», «Внедрение». Каждая фаза состоит из итераций длиной от двух до шести недель.

RUP строго регламентирует весь процесс в каждой фазе и определяет около 30 ролей, задействованных в разработке программного обеспечения. Каждая роль выполняет свои действия и производит артефакты, которые впоследствии используются другими ролями на следующих итерациях и этапах. Более подробно про RUP

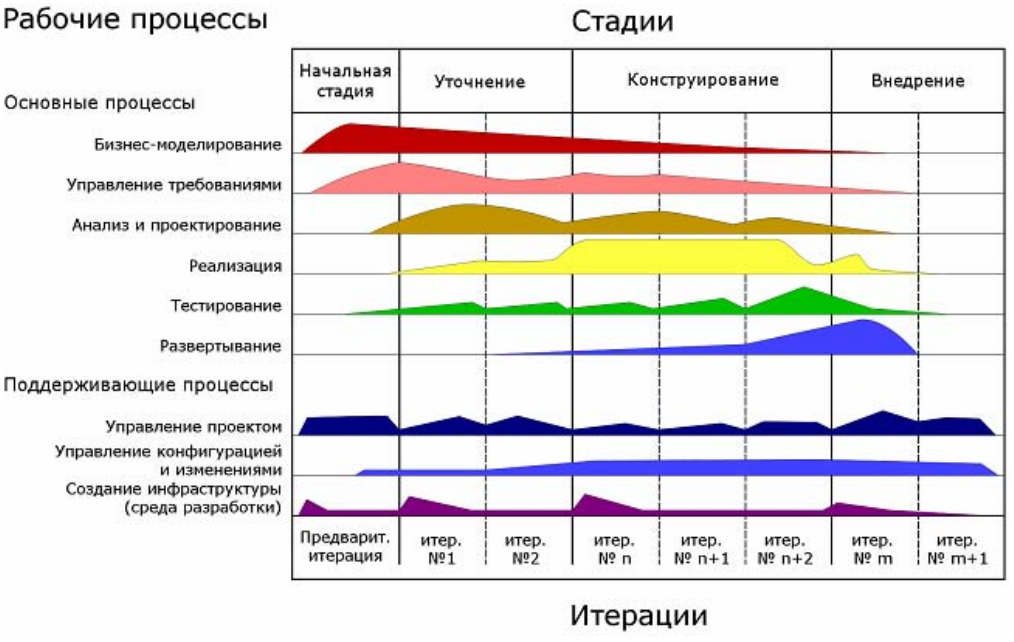


Рис. 1.3. Методология RUP

можно прочитать в книге Филиппа Крачтена «Введение в Rational Unified Process» (2-е изд.)[1], а мы лишь остановимся на некоторых ее особенностях:

- Гибкость, несмотря на формальность.

  Компании могут изменять и адаптировать процессы RUP под себя, RUP предлагает процессы в качестве основы, но не накладывает строгих ограничений.

- Дух RUP.

  Хотя разработка программного обеспечения — это регламентированный формальный рабочий процесс, предполагается наличие особого отношения к исполнению своих обязанностей, которое получило название «Дух RUP». Дух RUP стал важной предпосылкой к появлению гибких методологий. Вот его основные постулаты:

  - атаковать риски как можно раньше. Чем раньше обнаружен риск — тем больше времени на реакцию и выше вероятность избежать наступления риска;
  - обеспечить выполнение требований именно заказчиков;
  - концентрироваться на исполняемом коде;
  - готовиться к изменениям с самого начала;
  - создавать систему из компонентов;

[1] См. **https://goo.su/UJjfk0s**.

- разработать архитектуру как можно раньше;
- работать единой командой;
- сделать качество основной идеей.

Язык моделирования UML

Вместе с RUP появился язык моделирования UML (Unified Modeling Language), основная цель которого — графическое представление различных аспектов разработки информационных систем. Диаграммы UML позволяют описывать как структуру программы, так и течение бизнес-процессов. Рассмотрим некоторые основные детали UML.

UML состоит из трех основных элементов: сущности, отношения и диаграммы. Сущности — это существительные в терминах UML, отношения — это семантическая связь двух и более сущностей, а диаграммы — это способ визуализации, состоящий из набора сущностей и отношений.

Сущности UML

Структурные сущности UML

К структурным сущностям UML (рис. 1.4) относятся:

- Объект — это сущность, обладающая уникальностью, состоянием и поведением. Например, меч Арондит — это меч, накапливающий энергию (состояние) и повышающий характеристики носителя (поведение).
- Класс — описание множества объектов с общими атрибутами и методами. Имеет разнообразный синтаксис, заимствованный из разных языков, поддерживающих объектно-ориентированную парадигму. Может быть представлен в качестве класса «меч» или абстрактного класса «одноручное оружие».

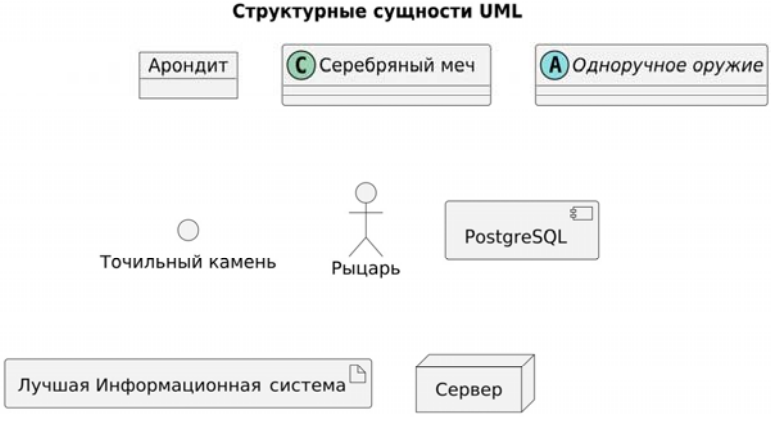


Рис. 1.4. Структурные сущности UML

- Интерфейс — именованное множество операций, предоставляемое поставщиком потребителю. В примере с мечом интерфейсом может быть инструмент для ухода за мечами — например, точильный камень или кузнечная мастерская.
- Действующее лицо (актор) — это сущность, внешняя по отношению к системе, инициирующая взаимодействие с системой. Это может быть роль (рыцарь), другая система или время.
- Компонент — модульная часть системы с определенным набором функций и интерфейсом. На рис. 1.4 — это PostgreSQL.
- Артефакт — результат работы. Это может быть как исполняемый файл (ЛучшаяИнформационнаяСистема.exe), так и UML-диаграмма или документация.
- Узел — вычислительный ресурс, на котором размещаются компоненты и артефакты. Например, сервер.

Поведенческие и дополнительные сущности UML

К поведенческим и дополнительным сущностям UML (рис. 1.5) относятся:

- Прецедент или вариант использования (use case) — сущность, которая определяет акт (действие или набор действий) использования системы актором. Например, забронировать место и время для битвы с драконом.
- Состояние — это период в общем жизненном цикле объекта, характеризующийся различным набором значений внутренних атрибутов. Например, когда дракон отдыхает на золоте между битвами, то он восстанавливает силы и здоровье.
- Пакет — это логическая группировка сущностей. Например, все объекты-драконы можно группировать в пакет «драконы».



Рис. 1.5. Поведенческие и дополнительные сущности UML

Комментирующие сущности UML

Комментирующие сущности UML — это заметки на диаграммах для облегчения чтения схем.

Отношения UML

К отношениям UML (рис. 1.6) относятся:

- Зависимость — исходный класс зависит от целевого класса. Если целевой класс будет изменен, то изменения могут потребоваться и для исходного класса.

В примере, показанном на рис. 1.6, у рыцаря есть зависимость от меча. Если меч станет двуручным или вовсе арбалетом, то рыцарю придется адаптироваться.

- Ассоциация — простая связь двух сущностей: король связан с королевством.
- Агрегация — связь сущностей, при которой одна сущность состоит из других, но все сущности могут существовать по отдельности. Например, латы рыцаря включают наручи. Но латы могут быть без наручей, хоть это и не очень безопасно, а наручи могут использоваться отдельно от лат.
- Композиция — связь сущностей, при которой одна из сущностей не существует без второй. Например, у рыцаря может быть удостоверение рыцаря. Но удостоверения без рыцаря быть не может.
- Наследование (обобщение) — сущность является специализацией более общей сущности. Змей Горыныч (специфичная сущность) — это дракон (общая сущность) с некоторыми модификациями.
- Реализация — сущность выполняет контракт (реализует интерфейс) определенный целевой сущностью.

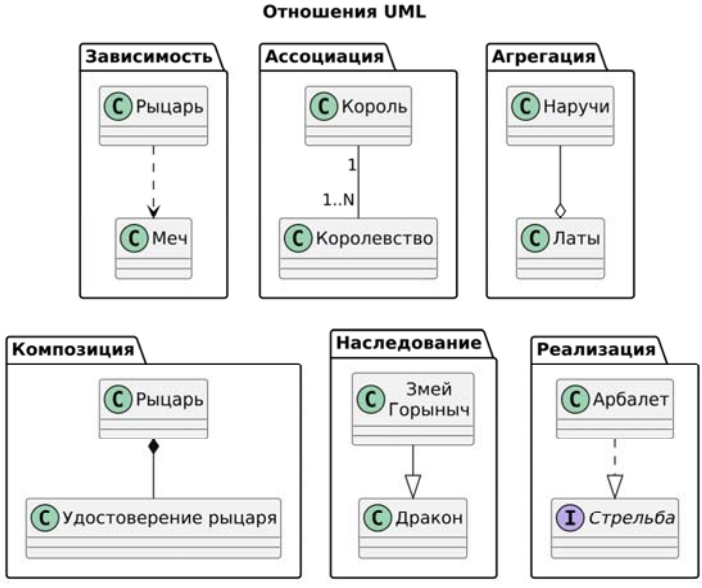


Рис. 1.6. Отношения UML

Диаграммы UML

Диаграмма классов

Диаграмма классов показывает классы, которые составляют систему, и связи между ними. Диаграмма, демонстрирующая отношения в UML (см. рис. 1.6), является техническим примером диаграммы классов.

Диаграмма состояний

Диаграмма состояний также называется *диаграммой конечных автоматов*. Она используется для моделирования поведения объекта в зависимости от различных состояний, через которые он проходит в течение своего жизненного цикла. Диаграмма наглядно показывает, как объект изменяет свое состояние в ответ на определенные события и действия. Этот тип диаграммы особенно полезен для описания сложных объектов с разветвленными логическими переходами и последовательностью операций. Диаграмма, показанная на рис. 1.7, демонстрирует возможные состояния и переходы дракона, который испытывает рыцарей, желающих доказать свое мужество на деле.

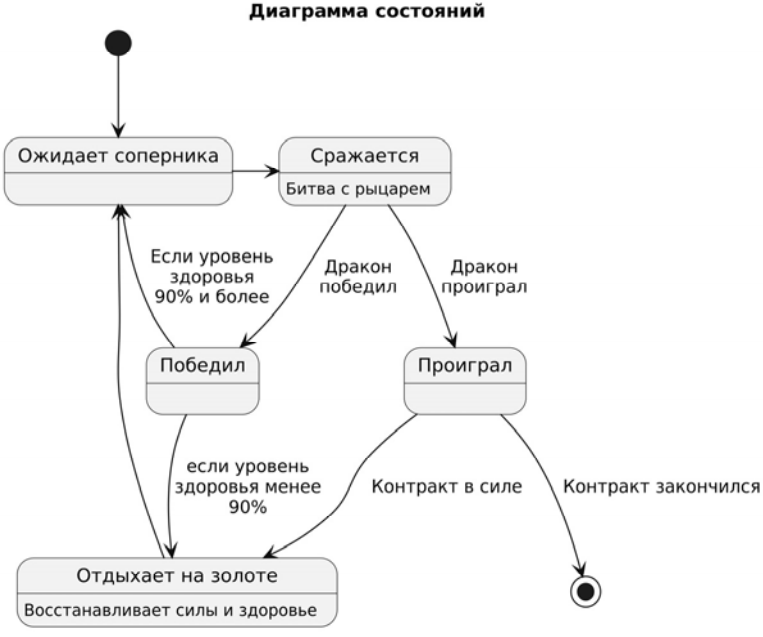


Рис. 1.7. Диаграмма состояний

Диаграмма развертывания

Диаграмма развертывания представляет собой средство для визуализации физического распределения компонентов системы на аппаратных узлах. Она показывает, как программные модули и компоненты системы размещены и взаимодействуют на различных физических устройствах, таких как серверы, рабочие станции и мобильные устройства. Эта диаграмма особенно важна для проектирования архитектуры системы, особенно в распределенных или клиент-серверных средах, где взаимодействие между аппаратными и программными частями должно быть четко определено. Диаграмма развертывания, приведенная на рис. 1.8, демонстрирует особенности микросервисной архитектуры, которая обсуждается в *главе 3*.

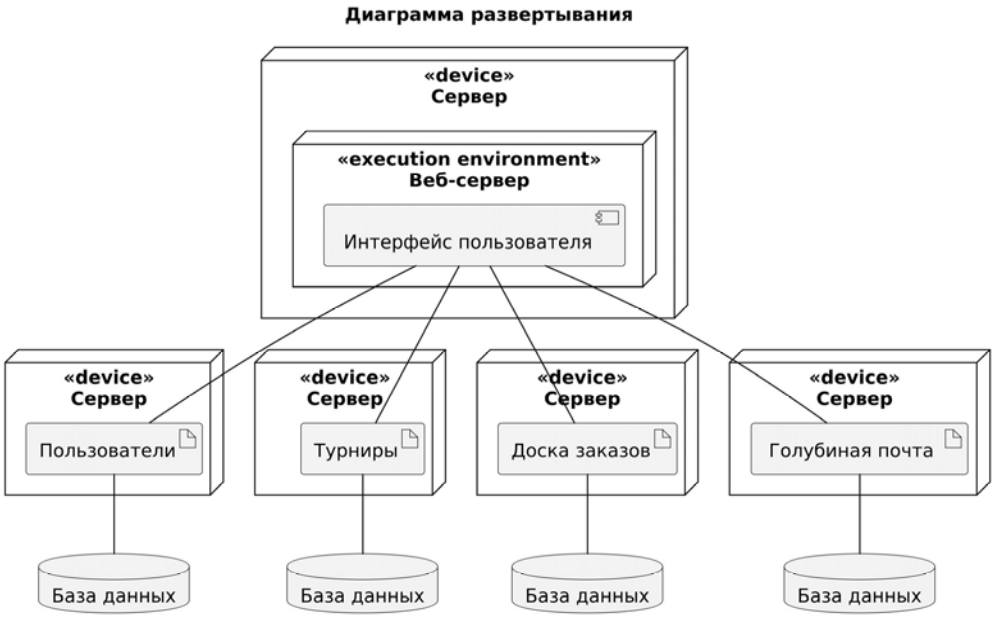


Рис. 1.8. Диаграмма развертывания

Диаграмма прецедентов

Диаграмма прецедентов (рис. 1.9) позволяет моделировать взаимодействие пользователей и системы, показывая, как различные роли (акторы) используют систему для достижения определенных целей. Эта диаграмма фокусируется на функциональных возможностях, которые система предоставляет пользователям, отображая сценарии, в которых пользователи выполняют задачи или получают полезные результаты с помощью системы.

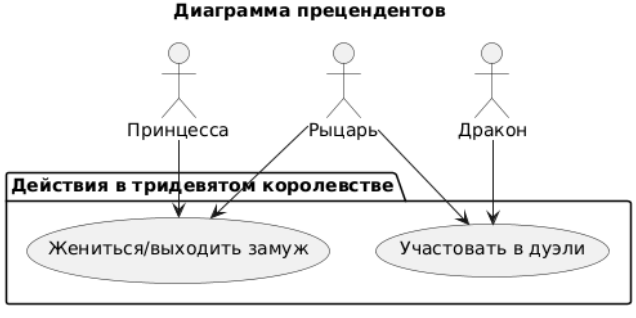


Рис. 1.9. Диаграмма прецедентов

Диаграмма последовательности

Диаграмма последовательности (рис. 1.10) используется для моделирования взаимодействия сущностей системы во времени, показывающего порядок вызова опе-

раций и сообщений. Она фокусируется на описании временно́й последовательности событий и обмена данными, чтобы отобразить, как конкретный процесс или сценарий выполняется в системе. Основой диаграммы последовательности является временна́я ось, идущая сверху вниз. Каждый участник взаимодействия представлен на диаграмме вертикальной линией, называемой *линией жизни*, которая отражает его существование на протяжении сценария. Взаимодействие между объектами изображается с помощью стрелок, которые показывают вызовы методов, обмен данными и возвращение результатов. Стрелки идут от одного объекта к другому, отражая направление и содержание сообщений, передаваемых между ними.

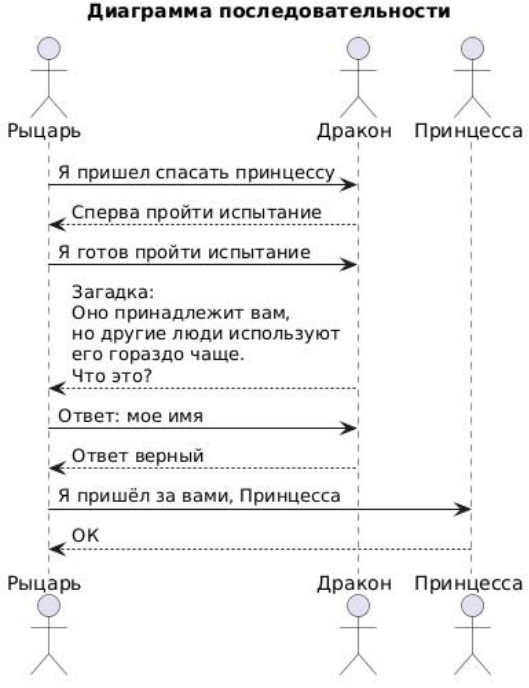


Рис. 1.10. Диаграмма последовательности

Гибкие методологии

Гибкие методологии — это семейство подходов, которые придерживаются принципов, описанных в манифесте гибких методологий разработки программного обеспечения (Agile Manifesto). В этом манифесте отражены четыре основные идеи:

- Люди и взаимодействие важнее процессов и инструментов.
- Работающий продукт важнее исчерпывающей документации.
- Сотрудничество с заказчиком важнее согласования условий контракта.
- Готовность к изменениям важнее следования первоначальному плану.

Как следует из этих принципов, гибкие методологии выше всего ценят готовый продукт и человеческое взаимодействие.

К базовым характеристикам гибких методологий относятся:

- Гибкость, открытость, быстрая реакция на проблемы.

  Процесс разработки под управлением гибких методологий, как следует из их названия, обладает высокой адаптивностью. В любой момент можно принять новые условия, требования и сразу же учесть их при разработке. При возникновении проблем их также можно быстро устранять.

- Отсутствие документации.

  Гибкие методологии позволяют разработчикам акцентировать внимание на написании кода и игнорировать неинтересное им написание документации. Хотя это повышает скорость разработки, уход «старого» сотрудника приведет к потере части документации, а появление нового — к необходимости отвлекаться на адаптацию сотрудника.

- Смещение фокуса.

  При разработке по гибким методологиям конечный продукт может существенно отличаться от первоначального видения. В некоторых случаях можно увлечься мелочами в проекте и потерять глобальную цель.

Хотя манифест гибких методологий был выпущен в 2001 году, в список гибких методологий входят подходы, появившиеся ранее. Среди них: быстрая разработка приложений (Rapid Application Development, RAD), фреймворк гибкого управления проектами Scrum, подход «программист-прагматик» (Pragmatic Programming) и экстремальное программирование (Extreme Programming, XP).

> ***Метод утенка***
>
> В книге «Программист-прагматик» Эндрю Ханта и Дейва Томаса[1] впервые описывается популярный способ отладки программ под названием «метод утенка». Этот способ предполагает, что при попытке объяснить проблему вымышленному собеседнику — резиновой уточке — человек формулирует вопрос, который содержит половину ответа и дает правильное направление мыслей для самостоятельного нахождения ответа.

Важно понимать, что гибкость — это не значит неограниченная свобода и расхлябанность. Экстремальное программирование, например, выводит проверку (ревью) кода на «экстремальный» уровень. Так, вместо последовательности «один написал, отдал второму — второй проверил, вернул назад» предлагается использовать парное программирование: один разработчик пишет код, а второй уже в момент написания производит его проверку.

Scrum

Scrum — это фреймворк гибкого управления проектами, помогающий командам структурировать работу и управлять ею на основе определенного набора ценно-

[1] См. **https://goo.su/rX7HVge**.

стей, принципов и практик. Scrum может использоваться не только для разработки ПО, но и в производстве, и в организации технической поддержки. Scrum предлагает роли и артефакты, которые позволяют управлять маленькой командой и достигать поставленных целей с минимальными затратами. Важно отметить, что Scrum — это фреймворк для организации рабочего процесса, который разделяет принципы Agile и поэтому причисляется к списку гибких методологий. Но Scrum не дает дополнительных знаний о том, как именно нужно разрабатывать программное обеспечение.

Работа в Scrum ведется на маленьких, от недели до четырех, временны́х промежутках, называемых *спринтами* (рис. 1.11). В начале спринта организуется встреча планирования, где определяется объем работ на ближайший спринт. В рамках спринта в начале каждого рабочего дня проводятся короткие ежедневные встречи, на которых каждый участник команды рассказывает о своих достижениях («Что я сделал?»), планах на текущий день («Что планирую делать?») и затруднениях («Что мешает?»), если такие есть.

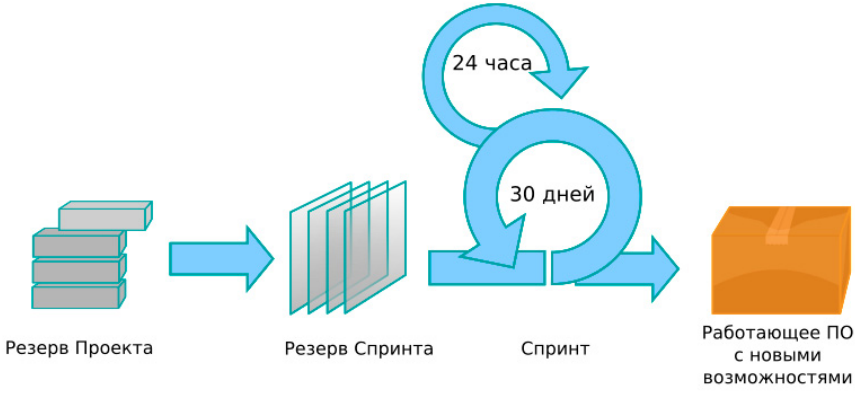


Рис. 1.11. Спринты Scrum

В процессе спринта поставленные задачи выполняются и выпускается обновление программного обеспечения — *инкремент*. В конце спринта производится демонстрация (Demo), на которой заинтересованным лицам показывают инкремент продукта и организуется ретроспектива — встреча с целью совершенствования процесса работы. Затем процесс повторяется. Все встречи проходят под контролем Scrum-мастера, который следит за соблюдением принципов Scrum, разрешает противоречия и не позволяет отвлекаться.

Существуют также модификации Scrum: ScrumBut или ScrumAnd — в которых допускаются некоторые отступления от основных его подходов.

При работе по Scrum используются следующие артефакты:

- Журнал пожеланий проекта (Project Backlog). Здесь хранятся все истории пользователя, относящиеся к проекту. *История пользователя* — это «задача», которая формулируется «Я, как пользователь А, хочу сделать Б и получить С».

- Журнал пожеланий спринта (Sprint Backlog). Сюда помещаются истории пользователей, которые они собираются реализовать в текущем спринте.
- Инкремент (Increment) продукта — представляет собой готовую к использованию часть продукта, которая должна быть реализована к завершению спринта.

Scrum эффективен в небольших самодостаточных командах — порядка 7–11 человек. В случае, если команда больше, то используется модификация Scrum of Scrums, когда весь коллектив разбивается на несколько Scrum-команд.

Как отмечалось ранее, Scrum не дает принципов построения архитектуры, поэтому бессмысленно примерять его к игрушечной информационной системе.

Feature-Driven Development

Feature-Driven Development (разработка, управляемая функциональностью) — это итеративная методология разработки программного обеспечения, также отвечающая принципам Agile. Эту методологию предложил Джефф Де Люка в 1997 году при разработке программного обеспечения для крупного сингапурского банка[1].

Подход, основанный на Feature-Driven Development (FDD), помогает объединить лучшие практики разработки программного обеспечения в единую методологию. FDD трактует создание программных продуктов как последовательную разработку и доставку отдельных функций системы, представляющих собой небольшие, легко управляемые части продукта, создающие ценность для пользователя.

Каждая функция в FDD имеет конкретное значение для конечного пользователя или клиента и должна быть завершена в короткие сроки — обычно от нескольких дней до двух недель. Этот подход помогает команде сосредоточиться на создании функций, которые напрямую связаны с требованиями заказчика, а не на выполнении абстрактных задач. Функции также могут быть легко приоритизированы и отслеживаемы, что делает их идеальными для построения сложных систем по частям.

Методология FDD реализуется в рамках следующих этапов:

1. Разработка доменной модели.

   На первом этапе команда создает высокоуровневую модель всей системы, что позволяет получить общее представление о структуре и функциях продукта. Это помогает команде лучше понять масштаб проекта и его ключевые компоненты.

2. Построение списка функций.

   Здесь происходит разбиение системы на функции, которые описываются в формате «действие — результат — объект». Такая структура помогает уточнить, какую ценность приносит каждая функция и как она будет использоваться.

3. Планирование функций.

   После определения функций команда приступает к их планированию. Здесь создается план функциональной разработки, который включает приоритизацию,

[1] См. **https://goo.su/zzHmbm**.

оценку времени выполнения и распределение функций между участниками команды. Члены команды могут работать над функциями параллельно, что ускоряет процесс разработки и делает его более эффективным.

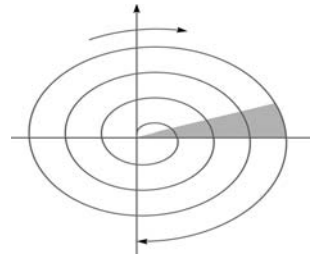
4. Проектирование функций.

   Когда план готов, руководитель выбирает функции, которые будут реализованы в ближайшие две недели. Вместе с заинтересованными лицами руководитель проектирует функции и строит диаграммы, описывающие поведение выбранных функций.

5. Реализация функций.

   Команда приступает к детальной разработке каждой функции. Разработка функции включает шесть контрольных точек: анализ области, дизайн, инспекцию дизайна, кодирование, инспекцию кода и включение в сборку. Первые три контрольные точки достигаются на этапах от разработки доменной модели до проектирования функций, а вторые три контрольные точки — на этапе реализации функций.

Feature-Driven Development имеет несколько преимуществ, особенно для больших проектов. За счет создания общей модели и плана функций команда получает ясное представление о структуре продукта и его перспективах. Четко очерченные функции позволяют лучше управлять приоритетами и адаптироваться к изменениям в требованиях, сохраняя гибкость и прозрачность. Параллельная разработка функций помогает ускорить процесс создания продукта и позволяет команде быстрее выпускать обновления и добавлять новую функциональность.



ГЛАВА 2

Выяснение требований

Выяснение требований реализуется в рамках анализа *пирамиды потребностей* (рис. 2.1), упоминаемой в книге Дина Леффингуэлла и Дона Уидрига «Принципы работы с требованиями к программному обеспечению. Унифицированный подход»[1]. Пирамида потребностей состоит из области проблемы, т. е. собственно потребностей, и области решения, т. е. функций и требований.



Рис. 2.1. Пирамида потребностей

Потребности — это менее формальное описание проблемы. Нередко заинтересованные лица не имеют компетенций в области построения информационных систем и потому не могут сформулировать задачу в соответствующих терминах. Это особенно актуально применительно к нашей сказке. Король говорит: «Пусть храбрый рыцарь спасет принцессу!», рассказывает про дракона и дает направление к замку, где сидит принцесса. Эти требования дают лишь первоначальное представление о системе и требуют уточнения.

[1] См. **https://goo.su/EuV8BI**.

При анализе потребностей можно выделить набор функций, которые необходимы системе, чтобы соответствовать пожеланиям заинтересованных лиц. На этом уровне нужны четкость и формализм, однако описания функций должны быть понятны заинтересованным лицам: «Рыцарь должен быть вооружен серебряным мечом, чтобы наносить урон дракону. Рыцарь должен быть экипирован зельем сопротивления к огню, чтобы защититься от магического воздействия дракона». К формулированию требований к программному обеспечению можно приступать после того, как заинтересованные лица утвердят список функций.

Требования к программному обеспечению однозначно, полно, непротиворечиво и корректно уточняют детали реализации функций. Для формулирования требований используют слова *должен/должна*: «Меч рыцаря должен состоять из серебра не менее 960-й пробы, длина меча — не менее 1.5 метра, гарда V-образная, угол заточки 30 градусов».

Требования помечаются атрибутами, которые позволяют выполнять сортировку и приоритизацию. Один из важных атрибутов — значение приоритета. Конечно, можно использовать цифровые значения, но также есть словесное описание категорий MoSCoW, представляющее собой метод расстановки приоритетов, основанный на классификации задач по четырем категориям:

- Must have — обязательное, фундаментальное требование;
- Should have — важное требование;
- Could have — потенциально возможное требование;
- Would like to have / wish — может быть реализовано в следующих версиях.

Помимо приоритета, также важно разделять требования по разным категориям на функциональные (что делает система?) и нефункциональные (каков опыт взаимодействия с системой?). В 1992 году американский экономист Роберт Грэйди предложил классификацию требований к системе под названием FURPS+. Эта же классификация используется в методологии RUP. Каждый символ аббревиатуры обозначает отдельный тип требований:

- Functionality — функциональность;
- Usability — удобство использования;
- Reliability — надежность;
- Performance — производительность;
- Supportability — поддерживаемость;
- + — дополнительные ограничения.

Далее мы рассмотрим их более подробно.

Функциональные требования

Функциональные требования описывают основные свойства и функции системы. Иначе говоря, это список того, что должна уметь система, чтобы удовлетворять потребностям заинтересованных лиц. Помимо очевидных требований к системе, сле-

дующих из потребностей, функциональные требования также могут содержать свойства или функции вне предметной области системы.

- ❒ Функциональные требования из предметной области.

  Система должна иметь возможность просматривать список всех принцесс. Каждая страница принцессы из этого списка должна содержать информацию о принцессе и ее статус (допустимые статусы: «в процессе похищения», «в заточении», «замужем»).

- ❒ Функциональные требования не из предметной области.

  Система должна отправлять уведомления посредством голубиной почты.

- ❒ Безопасность.

  - Отслеживание действий пользователей путем записи в журнал конкретных типов событий: система должна записывать все попытки входа не из тридевятого королевства.
  - Средства защиты информации, содержащейся в системе: система должна обеспечивать двухфакторную аутентификацию королей по паре «логин-пароль» и биометрическим данным.

Все остальные требования не описывают функции системы, не уточняют, как система должна решать задачи, и потому являются нефункциональными.

Нефункциональные требования

Нефункциональные требования, как следует из их названия, описывают не функции, которые должна выполнять система, а качественную составляющую при взаимодействии с системой.

- ❒ Usability (удобство использования).

  Требования к удобству использования в основном определяют взаимодействие человека и интерфейса разрабатываемой системы. Ответы системы на действия пользователя должны быть «человечными»: не слишком быстрыми, чтобы не путать пользователя, но и не слишком медленными, чтобы не заставлять его ждать. Также в удобство использования включается наличие документации или даже автоматизированного мастера (Wizard), который облегчает жизнь пользователям — задает направляющие вопросы и делает всё сам. И последнее по списку, но не по важности — эстетические требования, определяющие внешний вид интерфейсов системы. Эти требования могут отсутствовать, но в крупных компаниях есть брендбуки (BrandBook), которые подробно и досконально описывают корпоративный стиль, которого нужно придерживаться в разрабатываемых системах. Создание специальных компонентов интерфейсов для людей с ограниченными возможностями здоровья также является частью эстетических требований. Пример требования: «Интерфейс системы должен быть читаемым для магических существ с монохромным зрением». Для реализации этих требований применяется подход User Experience Design (UX).

- Reliability (надежность).

  Требования к надежности фиксируют способность программного обеспечения работать в течение определенного периода времени и определяют возможные сценарии отказа, их критичность для заинтересованных лиц и порядок действий обслуживающего персонала. Метриками для надежности выступают *среднее время между отказами* (Mean Time Between Failures, MTBF) и *коэффициент готовности*, в процентах определяющий, сколько времени в год система будет работать безотказно. В маркетинговых материалах можно встретить требование «пяти девяток» — 99,999%-ной доступности. В пересчете на время — это пять минут недоступности в год. Дополнительно указывается точность, если программное обеспечение используется для проведения математических расчетов: «Система должна обеспечивать доступность не менее 99%».

- Performance (производительность).

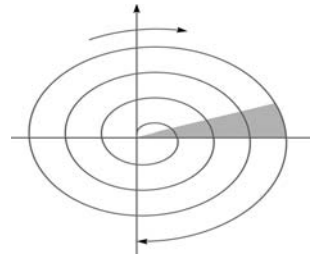
  Требования к производительности представляют собой набор взаимосвязанных условий, регламентирующих время выполнения вычислительных задач, отношение времени, потраченного на полезные задачи ко времени, потраченного на системные задачи, и т. п. Требования к производительности наиболее актуальны для систем реального времени, когда выход за рамки допустимого временно́го интервала может нарушить целостность управляемого объекта. Например: «Время ответа сервера не должно превышать 10 секунд при скорости Интернета у пользователя не менее 10 Мбит/с».

- Supportability (поддерживаемость).

  Требования к поддерживаемости отдельно выделяются для сложных корпоративных систем. Среди этих требований: расширяемость и масштабируемость — система должна адаптироваться к возрастающему количеству запросов. Более того, для крупных систем формируется требование останавливать определенные компоненты системы без прерывания обслуживания запросов от пользователей. Отдельно для систем прописываются требования функционирования в разных условиях: одинаково отображаться в разных браузерах или одинаково работать на разных операционных системах и разном оборудовании пользователя.

- \+ (дополнительные ограничения).

  Ограничения — это дополнение к классификации FURPS, которое появилось позднее. Ограничения могут являться следствиями разных условий, в том числе корпоративных и юридических. Ограничения проектирования: «Необходимо использовать методологию RUP». Ограничения реализации: «Для написания исходного кода необходимо использовать блокнот». Ограничения интерфейсов: «Аутентификацию необходимо сделать через корпоративный сервис секретов». Физические ограничения: «Система должна работать при температуре от –20°C до 50°C».



ГЛАВА 3

Архитектурные шаблоны и стили

Когда общее ви́дение информационной системы зафиксировано, можно приступать к написанию программного кода. Существует множество разных подходов к высокоуровневой организации программного кода, которые называются «архитектура». При этом один программный продукт может комбинировать несколько различных подходов.

Монолитная архитектура

Монолитная архитектура (рис. 3.1) — самый простой и очевидный подход при создании информационных систем. Монолитное приложение объединяет всю кодовую базу в один исполняемый модуль, который решает множество различных задач в зависимости от входных данных и параметров запуска. Монолитную архитектуру в разных операционных системах использует множество самостоятельных программ — от простых утилит командной строки до различных мультимедиаредакторов и игр.



Рис. 3.1. Монолитная архитектура

Монолитная архитектура характеризуется следующими особенностями:

- Легкость разработки маленькой системы.

  «Монолит» предполагает единую кодовую базу. Это снижает общую сложность разработки на ранних этапах существования системы. Компоненты информационной системы тесно связаны и могут обращаться непосредственно друг к другу. Помимо единой точки входа в систему, «монолит» также отличается возможностью агрегировать данные разных компонентов системы, а единая точка входа в хранилище данных на корню пресекает проблемы, свойственные другим архитектурным подходам. Но об этом мы поговорим позднее.

- Сложность разработки в больших системах.

  С ростом кодовой базы монолитные информационные системы теряют первоначальную легкость разработки. Каждое исправление требует пересборки всего приложения. Эта проблема частично решается инкрементальной сборкой, когда приложение собирается с нуля один раз, а при внесении исправлений пересобираются только затронутые модули. Тем не менее если над единой кодовой базой работает большое количество разработчиков, то неизбежно будут возникать конфликты слияния, когда несколько человек внесли изменения в один файл. Разрешение конфликтов требует дополнительного времени разработчиков.

- Недоступность функциональности при высокой нагрузке.

  Тесная связь компонентов внутри монолитного приложения имеет свои недостатки. В случае, если какой-либо компонент системы выполняет длительное задание, все остальные компоненты окажутся временно недоступны. Например, если в нашем игрушечном мире рыцарские турниры станут проводиться в разработанной системе виртуально, то система, хотя и будет занята турнирами, но «тормозить» начнут все компоненты, в том числе регистрация новых пользователей. Можно, конечно, развернуть еще один экземпляр системы, но тогда получится нерациональное использование ресурсов — ведь новый экземпляр содержит и обслуживает все компоненты системы, а не только «проблемные» турниры.

Микросервисная архитектура

Микросервисная архитектура (рис. 3.2) — это «антипод» монолитной архитектуры. В микросервисном подходе приложение разбивается на самостоятельные компоненты, каждый компонент решает свой набор задач и при необходимости обращается к другим сервисам.

Микросервисная архитектура характеризуется следующими особенностями:

- Независимая разработка сервисов.

  Каждый компонент (сервис) может разрабатываться независимо. В теории все сервисы системы могут быть написаны на разных языках программирования с использованием уникальных наборов библиотек и фреймворков. Главное — чтобы способ взаимодействия сервисов между собой был описан до начала разработки.

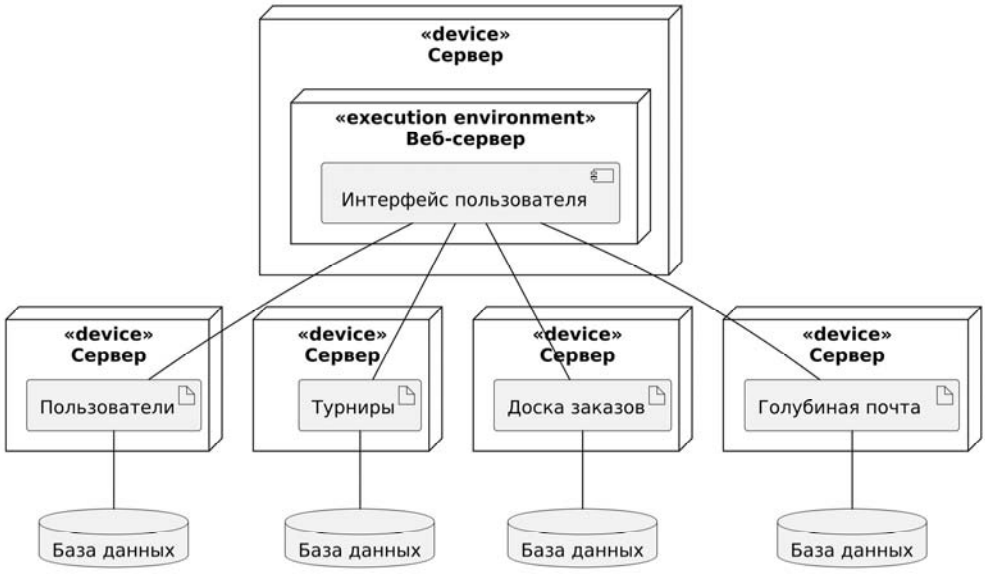


Рис. 3.2. Микросервисная архитектура

- Отказоустойчивость.

  При отказе одного из сервисов работоспособность системы частично сохраняется: продолжают работать функции, не зависящие от «упавшего» сервиса.

- Сложность межсервисного взаимодействия.

  Взаимодействие микросервисов между собой также требует особого подхода. Поскольку микросервисы работают автономно, они общаются друг с другом через сетевые запросы, что вносит определенные сложности, связанные с управлением взаимодействиями и обработкой ошибок.

- Масштабируемость.

  Микросервисная архитектура позволяет более гибко масштабировать систему. В монолитных системах масштабирование означает увеличение мощности всей системы, даже если нагрузка возрастает только на один ее компонент. В микросервисной архитектуре каждый сервис можно масштабировать независимо. Это позволяет распределять ресурсы более эффективно и обеспечивает оптимальную производительность приложения при минимальных затратах.

Микроядерная архитектура

Микроядерная архитектура (рис. 3.3) — это подход к созданию программного обеспечения, в котором ядро (основная часть) минимизировано до основных функций, а остальные задачи выполняются отдельными модулями, работающими вне ядра. Чаще всего такая архитектура применяется при разработке операционных систем.

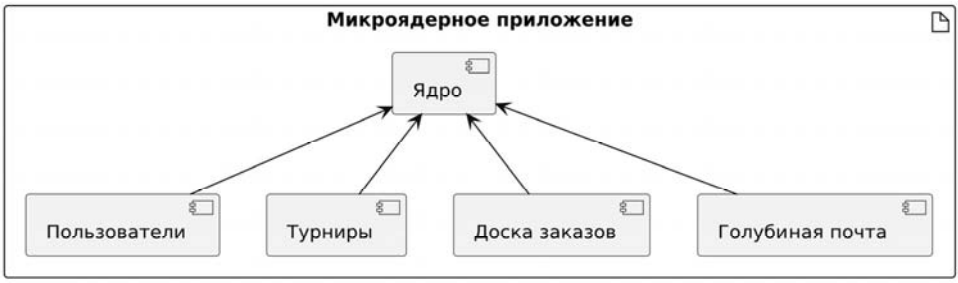


Рис. 3.3. Микроядерная архитектура

Микроядерная архитектура характеризуется следующими особенностями:

- Высокая модульность.

  Поскольку большинство компонентов системы работают вне ядра и взаимодействуют через заранее определенные интерфейсы, компоненты можно разрабатывать, обновлять и отлаживать независимо от основного ядра.

- Масштабируемость и надежность.

  Учитывая, что система состоит из независимых модулей, добавление новых модулей не является сложной задачей. Выполнение модулей отдельно от основного ядра повышает стабильность системы — сбой в модуле не повлияет на выполнение ядра и остальных модулей.

- Накладные расходы.

  Так как модули являются отдельными компонентами, появляются дополнительные расходы на обеспечение связи между ядром и модулем.

Событийно-ориентированная архитектура

Событийно-ориентированная архитектура (Event-Driven Architecture, EDA) — это подход к построению программного обеспечения, в котором взаимодействие компонентов приложения строится вокруг событий. Различные компоненты системы обмениваются *событиями* — уведомлениями о том, что в системе произошло некоторое изменение. Когда происходит событие, оно отправляется в «канал событий», и другие компоненты, которые ожидают события такого типа, могут его обработать. События могут обрабатываться синхронно и асинхронно. При этом асинхронная обработка дает возможность издателю события не ждать ответа от потребителя, а продолжать свою работу, обеспечивая высокую производительность и низкую задержку.

Событийно-ориентированный подход часто используется в фреймворках для создания графических интерфейсов пользователя (GUI).

Событийно-ориентированная архитектура характеризуется следующими особенностями:

- Низкая связанность компонентов системы.

  Компоненты системы не зависят от интерфейсов друг друга, а обмениваются данными через события, что позволяет развивать и масштабировать их независимо. В случае необходимости добавить новую функциональность, достаточно подписать модуль на нужные события, и это не потребует изменений в существующих компонентах. Такой подход упрощает поддержку и расширение системы, поскольку новые модули могут добавляться без изменения структуры основного кода.

- Сложность в управлении состоянием распределенной системы.

  В распределенных системах, где события обрабатываются асинхронно, может быть сложно обеспечить согласованность данных, особенно если несколько событий зависят друг от друга или требуют определенного порядка обработки.

Многоуровневая архитектура

Многоуровневая архитектура — подход к проектированию программных приложений, при котором функциональность системы разделяется на несколько логических уровней (слоев): каждый уровень выполняет свою задачу, и вместе они создают структурированную и легко управляемую систему.

У многоуровневой архитектуры есть частный случай — трехуровневая архитектура: уровень представления, бизнес-уровень и уровень данных. Уровень представления отвечает за взаимодействие с пользователем, т. е. за интерфейс и отображение данных. Этот слой принимает данные от пользователя и отправляет их в систему, а также получает ответы из бизнес-уровня, чтобы показать их пользователю в удобном формате. Бизнес-уровень, или логический слой, содержит основную бизнес-логику приложения, включая правила и алгоритмы обработки данных. Здесь происходят основные операции — такие как вычисления, обработка транзакций, валидация данных и выполнение бизнес-правил. Уровень данных, или слой хранения данных, предназначен для работы с базой данных или другими хранилищами. Он отвечает за выполнение операций чтения, записи и управления данными, обеспечивая доступ к информации, необходимой для работы бизнес-уровня.

Многоуровневая архитектура, предоставляя слои с разными задачами, создает каркас для реализации принципов доменно-ориентированной разработки (Domain-Driven Development, DDD). DDD, в свою очередь, наполняет этот каркас содержанием, определяя, каким образом логика предметной области (домен) и бизнес-процессы могут быть организованы и разделены между слоями.

Многоуровневую архитектуру характеризует следующая особенность:

- Разграничение обязанностей.

  Каждый уровень выполняет свой набор действий, и зависит только от интерфейса предыдущего уровня, что позволяет изменять реализацию уровней без конфликтов между ними. Разграничение обязанностей также может быть и недостатком. Даже в трехуровневой архитектуре запись данных, полученных в слое

представления, должна пройти все слои приложения. Это усложняет отладку и может влиять на скорость обработки запроса.

Representational State Transfer (REST)

REpresentational State Transfer (REST) — это архитектурный стиль для построения распределенных систем, разработанный Роем Филдингом в его докторской диссертации, опубликованной в 2000 году[1]. REST определяет ряд принципов и ограничений, которые делают взаимодействие между клиентом и сервером простым, масштабируемым и независимым от состояния.

Доступ к ресурсу в REST производится через уникальные идентификаторы (URI). *Ресурс* — это любая информация, которая может быть представлена в виде данных, например, объект в базе данных, документ или изображение. В REST-запросах существует независимость состояния, т. е. каждый запрос от клиента к серверу содержит всю информацию, необходимую для его выполнения. Сервер не хранит информацию о предыдущих взаимодействиях с клиентом, и запросы обрабатываются независимо друг от друга. Благодаря этому система становится проще и легче масштабируется, т. к. сервер не нагружается сохранением сессий и состоянием пользователей.

Для взаимодействия в REST используются стандартные методы HTTP: GET, POST, PUT и DELETE. GET служит для получения данных о ресурсе, POST — для создания нового ресурса, PUT — для обновления существующего ресурса и DELETE — для его удаления.

Для соответствия подходу REST нужно соблюдать пять условий и опционально шестое:

- Клиент-сервер.

  Архитектура должна состоять из сервера, который занимается хранением данных, и клиента, который занимается отображением данных. Это позволяет упростить серверную часть и сделать клиентскую часть более переносимой.

- Отсутствие состояния.

  Все запросы, посылаемые к серверу, должны иметь всю необходимую информацию для выполнения запроса.

- Кеширование.

  Ответы на запросы могут кешироваться для уменьшения времени ответа на одинаковые запросы. Однако кеш может навредить, если будет хранить устаревшую или неактуальную информацию. Поэтому ответы должны иметь отметку — кешируемые они или некешируемые.

- Единообразие интерфейса.

  Для обращения к ресурсам используется уникальный идентификатор (URI). Сервер возвращает представление ресурса в формате JSON, XML или каком-

[1] См. **https://goo.su/yaKGWJ**.

либо другом формате, но не сам ресурс. Каждое сообщение, запрос или ответ содержит достаточное количество информации, чтобы знать, каким образом обрабатывать это сообщение.

- Слои.

  REST допускает наличие промежуточных слоев между клиентом и сервером — таких как балансировщики нагрузки, кеширующие серверы и прокси-серверы. Эти слои улучшают производительность и масштабируемость, не нарушая взаимодействия между клиентом и сервером.

- Код по требованию (опционально).

  Сервер может передавать клиенту исполняемый код для расширения функциональности. Это может быть, например, скрипт JavaScript, который загружается на клиенте и выполняет дополнительные задачи, связанные с обработкой данных или визуализацией.

Распределенная транзакция

Распределенная транзакция — это транзакция, охватывающая несколько независимых систем или баз данных, в отличие от обычной транзакции, которая работает в рамках одного хранилища данных. В распределенных системах такие транзакции необходимы, когда требуется сохранить целостность данных, находящихся в разных базах данных или на разных серверах. Примером может служить банковская система, где перевод денег с одного счета на другой требует операций сразу на нескольких серверах, каждый из которых управляет своим счетом.

Транзакция гарантирует, что все изменения будут применены атомарно: либо все, либо ничего. Это относительно простая задача в одиночных сервисах, но в распределенных системах с множеством сервисов, у которых имеется собственная база данных, появляются сложности.

Представим, что есть два сервера, и на каждом запущен сервис с локальной базой данных (рис. 3.4). Допустим, пользователь обращается к турнирному сервису и выполняет действие, которое должно обновить данные в БД турнирного сервиса и в БД сервиса пользователей.

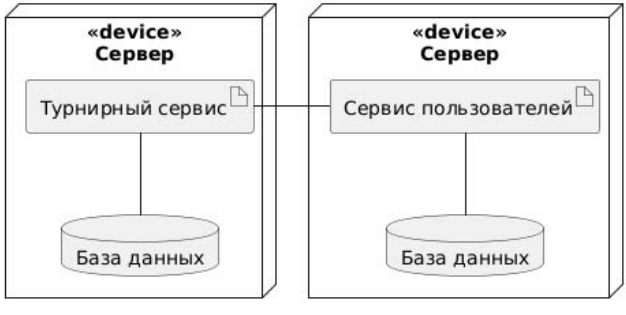


Рис. 3.4. Пример распределенной транзакции

Важно, чтобы изменения были применены сразу в обеих базах данных, а в случае ошибки по той части транзации, что уже была сделана, выполнен откат. Проблема здесь в том, что сбой может произойти в любой момент. Например, турнирный сервис записал информацию в свою базу данных, передал информацию в сервис пользователей, а сервис пользователей в этот момент аварийно завершился вместе с сервером, на котором он был развернут. После восстановления сервера получится, что в базе данных турнирного сервиса изменения применены, а в базе данных пользователей — нет. Это нарушение определения «транзакция».

Чтобы гарантировать атомарность транзакций, разработано несколько служебных протоколов и дополнительных способов.

Протокол двухфазного согласования (2 Phase Commit, 2PC)

В *двухфазном протоколе согласования* (рис. 3.5) участвуют координатор и несколько участников. Координатор управляет выполнением всей транзакции, а каж-

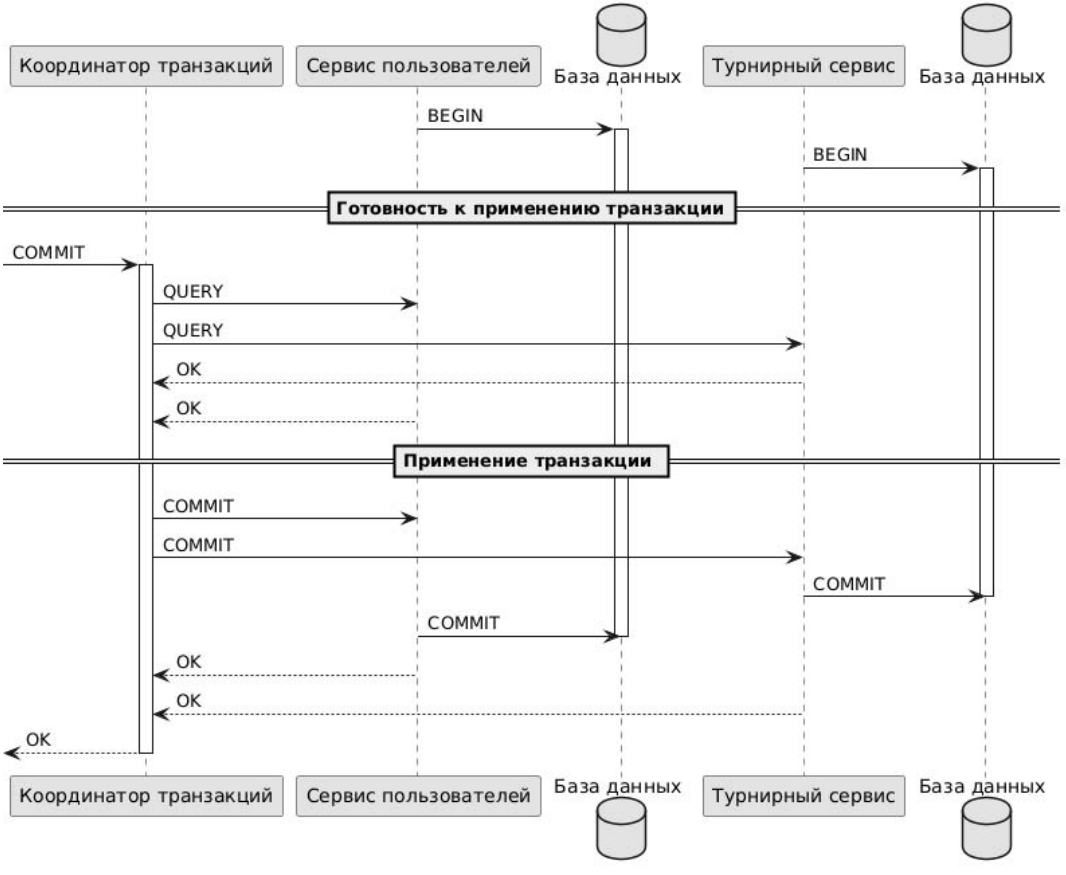


Рис. 3.5. Двухфазный протокол согласования

дый участник отвечает за выполнение своей части операции. Протокол работает в две фазы: сначала координатор отправляет всем участникам запрос на подтверждение готовности выполнить операцию. Участники проверяют возможность выполнения своей части и отвечают «готов» или «не готов». Если все участники готовы, координатор отправляет команду на выполнение (коммит) и завершает транзакцию. Если хотя бы один участник отказывает в готовности, транзакция отменяется (rollback).

Несмотря на простоту и эффективность двухфазного коммита, этот протокол имеет несколько слабых мест. Если координатор аварийно завершится вместе с одним из участников транзакции во время второй фазы, то нельзя однозначно сказать, получил ли «упавший» участник команду на применение транзакции и если получил, то успел ли применить. Получается ситуация, в которой один участник мог зафиксировать изменения, в то время как остальные участники процесса даже не узнали о решении сделать коммит.

Протокол трехфазного согласования (3 Phase Commit, 3PC)

Трехфазный протокол согласования добавляет между первым и вторым этапом 2PC шаг «префиксация». При этом первый этап ожидает, что все участники могут сделать фиксацию транзакции, второй этап уведомляет всех, что координатор получил готовность всех участников и скоро даст команду на фиксацию, а третий этап — непосредственно фиксация. Это исправляет проблему 2PC, но делает сам процесс сложнее.

Оркестрация

В информационных системах термин «оркестрация» имеет несколько значений. Оркестрация как система управления контейнерами рассматривается в *разд. «Развертывание: контейнеризация» главы 7*. Здесь же, говоря об оркестрации, мы имеем в виду способ организации распределенных транзакций.

В системах, использующих *оркестрацию*, предумотрен центральный контроллер (оркестратор), который управляет каждым шагом процесса и следит за тем, чтобы все части распределенной транзакции выполнились успешно. Оркестрация дает возможность организовать согласованное выполнение шагов, а также решает задачи мониторинга, обработки ошибок и отката действий в случае неудачи. Оркестратор, кроме того, помогает управлять длительными транзакциями, поддерживая их выполнение в асинхронном режиме, — он может ставить задачи в очередь и оповещать об успешном завершении или неудаче, поддерживая состояние для каждой части транзакции. Это особенно важно в масштабных системах, где транзакция может занять длительное время из-за сетевых задержек, высокой нагрузки или необходимости внешних вызовов к другим системам.

Хореография

Хореография — это подход, при котором право управления распределенной транзакцией и координация выполнения ее шагов передаются между сервисами во время выполнения без участия центрального контроллера. В отличие от оркестрации, где единый оркестратор управляет последовательностью действий, хореография позволяет каждому сервису самостоятельно реагировать на события и инициировать следующие шаги транзакции. Это делает хореографию более децентрализованным подходом, который лучше подходит для микросервисных систем, где важна автономность и независимость каждого сервиса.

Хореография работает с помощью событий, которые происходят в системе и на которые сервисы реагируют. В распределенной транзакции, построенной по принципу хореографии, каждый шаг транзакции вызывает определенное событие после успешного завершения своей части работы. Это событие служит сигналом для других сервисов, которые могут быть заинтересованы в его обработке и запуске своих собственных операций. Таким образом, каждый сервис знает, что ему делать после получения конкретного события, и взаимодействует с другими сервисами через цепочку событий, без необходимости в централизованном координирующем компоненте.

Подход, основанный на хореографии, имеет несколько важных преимуществ для распределенных транзакций. Во-первых, он повышает гибкость и масштабируемость системы, т. к. каждый сервис может быть изменен или масштабирован независимо, без необходимости изменения логики централизованного управления. Во-вторых, хореография хорошо подходит для асинхронной обработки, позволяя сервисам работать в собственном темпе и обрабатывать события по мере их поступления.

Однако хореография также накладывает определенные сложности на разработку и поддержку. Поскольку управление транзакцией распределено, становится сложнее отслеживать состояние транзакции и устранять ошибки. Каждому сервису необходимо знать, как правильно реагировать на события, что может увеличить сложность кода.

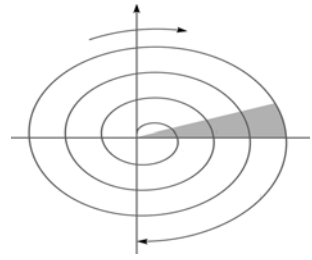
Паттерн SAGA

SAGA — это архитектурный подход к управлению распределенными транзакциями в системах, где традиционные механизмы, такие как двухфазный или трехфазный протокол согласования, неэффективны или неприменимы. Подход SAGA используется для обеспечения согласованности данных в распределенных системах, где транзакция включает несколько независимых сервисов, каждый из которых работает со своим хранилищем данных. Вместо того чтобы пытаться завершить все операции одновременно и немедленно, как это происходит в традиционных транзакциях, SAGA делит каждую сложную транзакцию на серию более мелких, независимых шагов, которые можно выполнить отдельно и в случае необходимости откатить.

Подход SAGA работает по принципу последовательности независимых локальных транзакций. Каждая локальная транзакция выполняет определенную часть общей задачи, и когда она завершается успешно, следующее действие либо запускается, либо инициируется асинхронно в зависимости от того, используется оркестрация или хореография. В случае ошибки на любом из этапов система выполняет компенсирующие транзакции, чтобы откатить предыдущие действия и вернуть систему к согласованному состоянию. Компенсации в SAGA играют ключевую роль. Поскольку шаги транзакции выполняются поэтапно и независимо, возможность отката достигается через операции, которые выполняют действия, обратные уже сделанным шагам. Компенсационная транзакция может содержать меньше шагов, чем оригинальная транзакция, потому что не все события могут быть компенсированы. Например, перезагрузка сервера или отправка SMS-сообщения не имеют «отменяющего» события, потому что сделанное уже не вернуть.

Паттерн SAGA позволяет поддерживать согласованность данных в условиях высокой нагрузки и больших объемов данных, не блокируя ресурсы на время выполнения всей транзакции. Это делает его подходящим для систем, в которых важна горизонтальная масштабируемость и отказоустойчивость. Тем не менее его использование требует тщательного проектирования, особенно в плане написания компенсирующих операций, т. к. ошибки в компенсациях могут привести к рассогласованности данных.

ГЛАВА 4



Принципы разработки ПО

Грамотный выбор архитектуры программного обеспечения позволяет легко «читать» высокоуровневое описание системы, но архитектура не регламентирует способы написания кода. Поэтому даже при идеально спроектированной архитектуре разработка может быть затруднена из-за запутанного исходного кода. Способы решения этой проблемы основаны на различных принципах, о которых мы поговорим в этой главе.

Описанные здесь принципы являются именно принципами, а не законами, поэтому придерживаться их не обязательно.

Принцип KISS

KISS — это акроним, который расшифровывается как «Keep It Simple Stupid» — в переводе: «Делай проще, балда». В различных источниках можно обнаружить разные вариации его расшифровки, избегающие грубости оригинала: «Keep It Simple, Silly» (делай проще, глупенький) или «Keep It Simple and Straightforward» (делай просто и понятно).

Основная мысль принципа — делать проектируемые системы простыми, без излишней сложности. В книге «Искусство программирования для UNIX» Эрика Реймонда[1] говорится, что принцип KISS лежит в основе философии UNIX, которая кратко формулируется так: «Делайте что-то одно, но делайте хорошо». Выполнение принципа KISS легко обнаружить в большинстве утилит командной строки в семействе операционных систем *NIX. Смотрите:

- утилита cat позволяет последовательно объединять указанные файлы (конкатенировать). Побочный эффект: можно выводить содержимое одного файла;
- утилита head выводит первые *N* строк/байтов указанного файла;
- утилита tail выводит последние *N* строк/байтов указанного файла;

[1] См. **https://goo.su/qRTq5**.

- утилита sort сортирует строки, переданные на вход;
- утилита uniq отыскивает дублирующиеся строки в отсортированных данных.

Каждая из представленных утилит делает что-то одно, а их комбинации позволяют решать различные задачи. При написании исходного кода принцип KISS можно трактовать так:

- каждая функция, процедура или метод должны решать одну задачу;
- функции, процедуры или методы должны быть маленькими;
- избегайте большой вложенности.

Большая вложенность легко достигается из-за буквального трактования условий. Давайте представим, что мы пишем функцию, которая проверяет, что пользователь аутентифицирован, что он является драконом и имеет права администратора. На языке C «вредный» пример выглядит так:

```
int checkUser(User user) {
  if (user.isAuth) {
    if (user.isDragon) {
      if (user.isAdmin) {
        return 1;
      } else {
        printf("Не админ!");
      }
    } else {
      printf("Не дракон!");
    }
  } else {
    printf("Не аутентифицирован!");
  }
  return 0;
}
```

Получается тройная вложенность, что затрудняет прочтение условий даже с однострочными действиями. Но т. к. выполнение каждого условия является критичным и останавливает проверку, то мы можем избавиться от вложенности, выходя из функции сразу после проваленной проверки:

```
int checkUser(User user) {
  if (user.isAuth) {
    printf("Не аутентифицирован!");
    return 0;
  }

  if (user.isDragon) {
    printf("Не дракон!");
    return 0;
  }
```

```
  if (user.isAdmin) {
    printf("Не админ!");
    return 0;
  }

  return 1;
}
```

Функция проверяет то же условие, но теперь нет вложенных условных операторов, и код стал понятнее.

Принцип YAGNI

Принцип YAGNI (You Aren't Gonna Need It, Вам это не понадобится) — это один из подходов в разработке программного обеспечения, который направлен на минимизацию избыточной работы и упрощение процессов. Его суть заключается в том, чтобы не реализовывать функциональность, которая может понадобиться в будущем, но на текущий момент не является необходимой.

Этот принцип помогает избежать создания избыточного кода, который усложняет проект, увеличивает время разработки и повышает риск ошибок. Принцип YAGNI предупреждает, что предположения о будущих требованиях часто оказываются неверными, а попытки заранее учесть все возможные сценарии приводят к значительным затратам без ощутимых преимуществ.

Применение принципа YAGNI требует дисциплины и умения фокусироваться на текущих задачах. Разработчики часто испытывают соблазн предусмотреть дополнительную функциональность, аргументируя это тем, что это может быть полезно. Однако реальность разработки показывает, что такое происходит не всегда.

Этот принцип тесно связан с методологиями гибкой разработки, где акцент делается на создании минимально жизнеспособного продукта (Minimum Viable Product, MVP) и итеративном улучшении системы на основе реальных потребностей пользователей. Вместо того чтобы пытаться предусмотреть все заранее, разработчики, следуя YAGNI, предпочитают реагировать на изменения по мере их появления.

Принцип SOLID

SOLID — это мнемонический акроним, объединяющий пять основных принципов объектно-ориентированного программирования. Если придерживаться принципов SOLID, то разработанное программное обеспечение будет легко поддерживать и расширять в течение долгого времени.

S — Single Response Principle (принцип единой ответственности).

Принцип единой ответственности гласит, что каждый объект должен иметь единственную ответственность. Это похоже на принцип KISS, но не для целых систем, а

для маленьких компонентов. Ранее, в примерах архитектуры, отдельно выделялись турнирный модуль и модуль отправки уведомлений голубиной почтой — и это полностью соответствует принципу единой ответственности: турнирный модуль управляет турнирами, модуль уведомлений — уведомлениями.

Можно допустить плохую мысль: «в модуле уведомлений всего одна функция, не хочу делать отдельный модуль, реализую эту функцию в турнирном модуле». Конечно, код будет работать, но принцип явно нарушается. Последствия будут заметны позднее, при развитии системы. Во-первых, представим, что после реализации турнирного модуля мы приступили к реализации доски объявлений, которая тоже должна рассылать уведомления посредством голубиной почты. Доска объявлений будет вынуждена обращаться к турнирному модулю для отправки уведомлений, что немного контринтуитивно. Во-вторых, может оказаться, что для важных заданий надо использовать более надежный вид транспорта — гонца на лошади. Для этого придется изменять турнирный модуль и надеяться, что ничего в турнирах не сломается.

O — Open-Closed Principle (принцип открытости/закрытости)

Принцип открытости/закрытости предлагает держать модули открытыми для расширения, но закрытыми для изменений.

Предположим, что у нас есть класс `Character` (рис. 4.1), который в общем представляет человека с некоторым запасом очков здоровья (`HP`). У этого класса есть наследники: рыцарь (`Knight`) и маг (`Wizard`). Наследники расширяют функциональность базового класса: рыцарь может наносить урон другим персонажам своим мечом, а волшебник имеет очки магии (`MP`) и может творить огненные шары. Первая половина принципа выполняется.

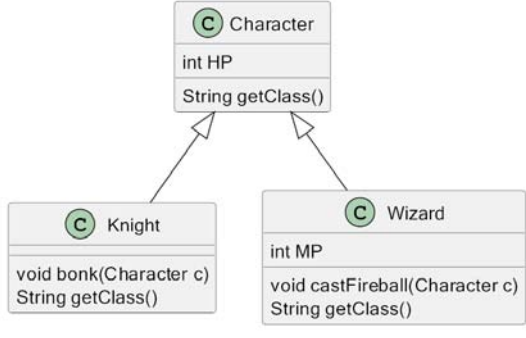


Рис. 4.1. Пример реализации принципа открытости/закрытости

Закрытость предполагает, что код родительского класса остается неизменным при появлении новых наследников. В UML-диаграмме, показанной на рис. 4.1, имеется метод `getClass`, который для родительского класса должен возвращать "`Человек`", а

для наследников — "Рыцарь" и "Маг" соответственно. Для соблюдения принципа открытости/закрытости родительский класс должен реализовать метод, а наследники — переопределить его.

Вот пример кода на языке Java, где принцип открытости/закрытости нарушается, — наличие в родительском классе «всезнающего» метода, который по каким-то данным «гадает» и возвращает ответ:

```
class Character {
  String getClass() {
    String className = this.getClass().getSimpleName();
    if(className.equalsTo("Wizard")) return "Маг";
    if(className.equalsTo("Knight")) return "Рыцарь";
    return "Человек";
  }
}
```

В этом случае при добавлении нового наследника — например, повара, придется изменять метод родительского класса.

L — Liskov substitution principle (принцип подстановки Барбары Лисков)

Принцип подстановки Барбары Лисков гласит, что наследующий класс должен дополнять, а не замещать поведение базового класса. Код, принимающий аргумент базового класса, должен работать одинаково как с базовым классом, так и с любым его наследником.

Из этих определений можно выделить конкретные рекомендации:

1. Метод наследника не должен изменять свойства базового класса, которые не изменяет метод базового класса.
2. Метод наследника не должен выбрасывать исключения, которые не предусмотрены базовым методом.

Важность первого пункта в различных источниках демонстрируют на геометрических фигурах. Представим, что у нас есть прямоугольник (Rectangle) и квадрат (Square). Что из них базовый класс, а что — наследник? Геометрия говорит, что квадрат — это частный случай прямоугольника, в котором высота равна ширине. Значит, прямоугольник — базовый класс, а квадрат — наследник. В коде на Java это выглядит так:

```
class Rectangle {
  private float width;
  private float height;

  public void setWidth(float width) {
    this.width = width;
  }
```

```
  public void setHeight(float height) {
    this.height = height;
  }

  public float calculateArea() {
    return this.width * this.height;
  }
}

class Square extends Rectangle {
  public void setWidth(float width) {
    this.width = width;
    this.height = height;
  }

  public void setHeight(float height) {
    this.width = width;
    this.height = height;
  }
}
```

На первый взгляд выглядит логично. Квадрат является наследником прямоугольника, подсчет площади корректен. Правда в сеттеры высоты и ширины пришлось добавить «костыль»: когда задается высота квадрата, то метод также меняет ширину, чего не было в методе базового класса. Это нарушает принцип подстановки Лисков! На первый взгляд это не проблема, но искусственному примеру — искусственный крайний случай. Представим, что где-то в системе есть такая функция:

```
void SomeChecks(Rectangle r) {
  r.setWidth(100);
  r.setHeight(500);
  assert r.calculateArea() == 500000;
}
```

Эта функция принимает прямоугольник или его наследника, изменяет ширину и высоту и проверяет, что площадь соответствует ожидаемой. Квадрат как наследник прямоугольника такую проверку не пройдет, что приведет к сбою в работе программы. Как можно решить проблему прямоугольника и квадрата?

Первый вариант: пересмотреть наследование и сделать базовым классом фигуру (`Shape`), а квадрат и прямоугольник — самостоятельными наследниками со своими атрибутами. От базового класса в наследниках доступен только метод вычисления площади, который необходимо переопределить (рис. 4.2). Второй вариант — сделать прямоугольник и квадрат неизменяемыми. Тогда отпадает необходимость в сеттерах и нарушение принципа Лисков устраняется.

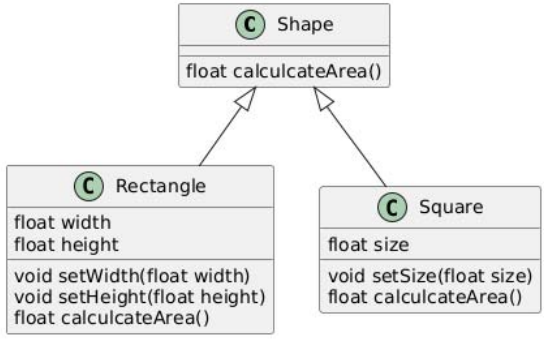


Рис. 4.2. Пример реализации принципа подстановки Барбары Лисков

I — interface segregation principle (принцип разделения интерфейса)

Принцип разделения интерфейса предлагает использовать множество узкоспециализированных интерфейсов вместо одного универсального. Вот пример неправильного «универсального» интерфейса:

```
interface IFighter {
  void bonk(Character c);
  void sharpenBlade();

  void castFireball(Character c);
  void castConcentration();

  void Rest();
}
```

Универсальный интерфейс для турниров `IFIghter` содержит методы, которые полезны как рыцарю, так и магу. Но если мы захотим реализовать интерфейс на рыцаре, то нам придется реализовывать ненужные методы, которые будут выбрасывать исключение: «Рыцарь не умеет колдовать». Лучше разбить этот интерфейс на три отдельных интерфейса: «общий» — для отдыха, «рыцарский» — для атак мечом и «магический» — для колдовства:

```
interface IPhysicalFighter {
  void bonk(Character c);
  void sharpenBlade();
}

interface IMagicFighter {
  void castFireball(Character c);
  void castConcentration();
}
```

```
interface IRest {
  void Rest();
}
```

При разделении интерфейсов каждый класс выбирает только необходимые для себя интерфейсы и не реализует ненужные методы.

D — dependency inversion principle (принцип инверсии зависимостей)

Абстракции не должны зависеть от деталей — наоборот: детали должны зависеть от абстракций. *Принцип инверсии зависимостей* предлагает опираться на интерфейсы, а не на конкретную реализацию.

Предположим, что к нашей игрушечной информационной системе предъявляют требование использовать реляционную базу данных. Создаем класс, который занимается взаимодействием с реляционной базой данных, добавляем ссылку в турнирную систему. Готово! Однако при таком подходе турнирная система жестко завязана на реляционную базу данных (рис. 4.3).

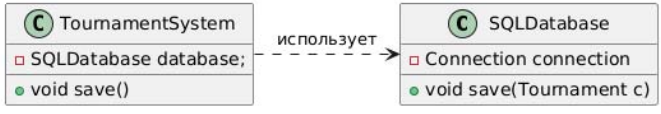


Рис. 4.3. Турнирная система жестко завязана на реляционную базу данных

Правильным решением будет объявить интерфейс с методами, которые нам нужны от базы данных. Теперь в диаграмме классов (рис. 4.4) объявлен один метод — сохранение турнира, и мы можем реализовать сохранение данных в любом хранилище, будь то реляционная база данных или документоориентированная.

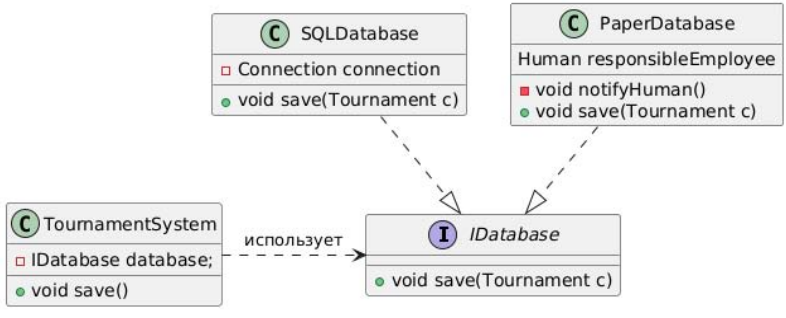


Рис. 4.4. Пример реализации принципа инверсии зависимостей

Инверсия управления

Инверсия управления (Inversion of Control, IoC) — это принцип проектирования, в котором программист пишет отдельные компоненты, которые выполняются сторонним программным решением (фреймворком). Рассмотрим этот принцип подробнее.

Разработчик пишет исходный код, компилирует в исполняемый файл и затем запускает его. Программа выполняется так, как написал разработчик, — т. е. весь порядок исполнения (Flow of Control) «принадлежит» разработчику. Чем больше функций реализует программа, тем больше разработчики исследуют готовые решения, которые можно переиспользовать в своей программе. Здесь мы впервые подходим к терминам «библиотека» и «фреймворк».

Библиотека — это внешний сборник подпрограмм, которые используются для разработки программного обеспечения. С точки зрения операционной системы библиотеки делятся на два типа: статические и динамические. *Статические библиотеки* связываются с программой на этапе построения — т. е. программа содержит в своем исполняемом файле статические библиотеки. *Динамические библиотеки* распространяются отдельно, а при запуске программы операционная система «подкладывает» в адресное пространство программы все необходимые динамические библиотеки.

Для программиста библиотеки, как статические, так и динамические, выглядят как набор функций, которые можно использовать. Когда программист в коде вызывает библиотечную функцию, то он передает управление из своей программы в библиотеку, где функция совершает полезную работу и затем возвращает управление обратно. В этом случае программист временно отдает управление из своих «рук», но основной контроль всё еще остается в руках программиста.

При разработке программ для современных операционных систем неявно используются библиотеки, которые находятся в системе. Даже самая первая программа многих разработчиков — «Hello, World!» — содержит как минимум одну библиотеку. Вот пример «Hello, World!» на языке C:

```
#include <stdio.h>

int main(int argc, char* argv[]) {
  printf("Hello, World!");
  return 0;
}
```

Функция `main()` — точка входа в программу — содержит количество аргументов командной строки (`argc`) и массив строк (`argv`), т. е. сами аргументы. Хорошо, передать аргументы — это не проблема, но кто подсчитал их количество? В семействе операционных систем Linux этим занимается системная библиотека glibc, несколько функций которой «внедряются» в начало и конец программы компилятором. Таким образом, использование библиотек — это очень распространенная практика. В чем же отличие от фреймворка?

Фреймворк (каркас) — это слой абстракции, который предоставляет инструменты общего назначения для разработки программ и абстрагирует разработчика от уже решенных проблем и задач. В самом простом случае фреймворк — это набор программ, которые вызывают функции, написанные программистом. Например, фреймворк FastAPI позволяет в несколько строк Python-кода развернуть HTTP-сервер:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def read_root():
    return {"Hello": "World"}
```

Программист пишет программу, которая использует объекты и функции фреймворка, а фреймворк абстрагирует программиста от деталей и «лишней» работы. В случае с FastAPI фреймворк прослушивает сокеты, принимает подключения, а также разбирает входящие HTTP-сообщения и собирает исходящие. Программисту остается выбрать точку подключения (эндпоинт) и написать функцию-обработчик, которая по входным данным выдаст результат.

При таком подходе код, который написал программист, напрямую не запускается. Запускается основная программа фреймворка, которая регистрирует внутри себя модули, классы или функции, написанные разработчиком, и вызывает их, когда выполняются заданные условия. Это и называется *инверсией управления* — выполняется сторонняя программа, которая решает, когда выполнять код программиста.

Фреймворки — это отличный инструмент, который абстрагирует разработчика от распространенных задач и ускоряет разработку. Цена такого удобства — высокая сложность добавления функций, которые фреймворк не поддерживает.

Внедрение зависимостей

Вернемся к принципам SOLID, в частности к букве D — принципу инверсии зависимостей. Этот принцип предлагает создавать интерфейсы и делать более специфичные реализации этих интерфейсов. Однако реализация интерфейса не появится «магически» там, где она нужна. На диаграмме с примером реализации принципа инверсии зависимостей (см. рис. 4.4) показан класс `TournamentSystem`, которому нужна база данных, которую реализует интерфейс `IDatabase`, но при этом класс не хочет знать, какие реализации бывают. Соответственно, класс не знает, как инициализировать конкретную реализацию и не может ею воспользоваться. Эту проблему решает *внедрение зависимостей* (dependency injection).

Внедрение зависимостей использует относительную терминологию «сервисов» и «клиентов». *Сервис* — это реализация интерфейса, которую необходимо внедрить в клиента, а *клиент* — это объект, который нуждается в сервисе. Существуют три способа внедрения зависимостей:

- Внедрение в конструкторе.

  В этом способе конструктор объекта принимает в аргументах все зависимости:

```
class TournamentSystem {
  private IDatabase database;

  public TournamentSystem(IDatabase database) {
    this.database = database;
  }
}
```

  После построения объекта все зависимости готовы к использованию. Это является достоинством — объект не может быть создан без всех зависимостей.

- Внедрение через метод.

  В этом способе у каждой зависимости есть сеттер — метод, который позволяет назначить соответствующий член класса:

```
class TournamentSystem {
  private IDatabase database;

  public setDatabase(IDatabase database) {
    this.database = database;
  }
}
```

  Достоинство этого способа вытекает из его недостатка — зависимости могут быть внедрены в любой момент, не обязательно при создании объекта. Это позволяет подменять реализации в процессе выполнения, но вместе с тем методы объекта могут обратиться к зависимости до ее внедрения.

- Внедрение через интерфейс.

  Интерфейсное внедрение похоже на внедрение через метод — отличие лишь в том, что сеттер выделен в отдельный интерфейс, что позволяет «собирать» объекты разных классов, которым нужна одна зависимость, в одну коллекцию:

```
interface DatabaseSetter {
  public setDatabase(IDatabase database);
}

class TournamentSystem implements DatabaseSetter {
  private IDatabase database;

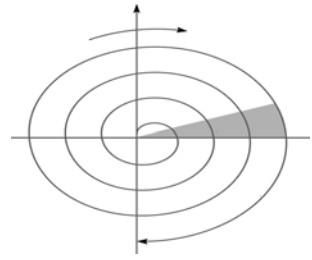
  @Override
  public setDatabase(IDatabase database) {
    this.database = database;
  }
}
```

Каждый из способов описывает способ доставки зависимости в экземпляр класса, но как инициализируются сервисы, как создаются клиенты и как сервисы находят своих клиентов? Самое простое решение — делать это все вручную:

```
IDatabase database = new SQLDatabase("somedb://127.0.0.0.1:27015");
TournamentSystem tournamentSystem = new TournamentSystem(database);
```

Здесь в момент инициализации программы мы создаем сервисы и инициализируем клиентов.

В этом фрагменте кода используется способ с внедрением зависимостей через конструктор. Разрешение зависимостей, т. е. определение, в каком порядке должны создаваться объекты классов, и внедрение зависимостей полностью лежит на программисте.



ГЛАВА 5

Технологии

При разработке программного обеспечения используются библиотеки и фреймворки, которые умеют решать определенные задачи. Библиотеки и фреймворки привязаны к языкам программирования и предоставляют лишь набор интерфейсов, которые можно комбинировать для достижения поставленной задачи. Однако существует набор задач, решение которых — это готовый продукт, который можно использовать независимо от языка программирования. Такие задачи — это хранение и передача данных.

Системы управления базами данных

Как известно, в операционных системах для хранения информации используются файлы, а для структурирования хранимой информации — каталоги. Долгое время обычных файлов хватало для большинства задач, но с развитием информационных систем объем хранимых данных только увеличивался. С ростом объема данных требовались инструменты, способные хранить и быстро обрабатывать большие объемы данных. Так появились системы управления базами данных (СУБД).

Обратите внимание, что терминология стандарта ISO/IEC 2382:2015 утверждает, что *база данных* (БД) — это совокупность данных, организованных в соответствии с концептуальной структурой, а программное обеспечение, которое организует базу данных, называется *системой управления базами данных*.

С ростом объемов данных базы данных разделились на две как бы категории — по объему обрабатываемых данных и времени отклика на команду: OLAP и OLTP:

- OLAP (Online Analytical Processing, интерактивная аналитическая обработка) — это подход, ориентированный на сложные аналитические запросы и агрегацию данных, что позволяет изучать данные с разных точек зрения и в различных разрезах.

  OLAP-системы применяются в бизнес-аналитике, финансовом планировании, маркетинговых исследованиях и других областях, где нужно глубоко анализировать данные и выявлять скрытые закономерности.

- OLTP (Online Transaction Processing) — транзакционная система, основная цель которой заключается в обеспечении высокой скорости и надежности транзакционной обработки, где каждая транзакция — это последовательность операций, которая должна быть выполнена целиком и точно.

  Основополагающий принцип работы OLTP — это транзакционность, гарантирующая целостность данных даже при высокой нагрузке и возникновении ошибок. Чтобы обеспечить надежность, OLTP придерживается модели ACID, основанной на четырех важных свойствах: Atomicity (атомарность), Consistency (согласованность), Isolation (изоляция) и Durability (устойчивость):

  - *Атомарность* гарантирует, что транзакция будет применена полностью или не будет применена вовсе;
  - *Согласованность* гарантирует, что применение любой транзакции сохраняет согласованность базы данных. Иными словами, транзакция не может нарушить схему и записать строку в числовой столбец;
  - *Изоляция* гарантирует, что параллельные транзакции не должны оказывать влияние друг на друга;
  - *Устойчивость* гарантирует, что изменения, сделанные примененной транзакцией, должны быть сохранены независимо от внешних проблем. Иными словами, если пользователь получил подтверждение, что транзакция применена, то даже немедленная экстренная перезагрузка сервера с базой данных не должна повлиять на внесенные изменения.

На текущий момент существует множество СУБД с разными подходами в основе. Все СУБД можно классифицировать несколькими способами. Основная классификация — по модели данных.

Реляционная модель данных

Реляционная модель данных — это способ организации и представления данных в виде таблиц и связей между ними, которые называются *отношениями*. В основе реляционной модели лежит размещение данных в таблицах, где все строки в таблице имеют одинаковую структуру, а значения в каждом столбце однородны. *Строки* представляют записи с фиксированным количеством атрибутов, а *столбцы* имеют имя и представляют конкретный атрибут. При этом на пересечении строки и столбца может быть либо строго одно значение, либо ни одного. У каждой строки в реляционных базах данных должен быть *ключ* — набор атрибутов, по которому можно однозначно определить строку. Чаще всего используется самостоятельный «первичный ключ», который использует сквозную нумерацию всех строк в базе данных. Все связи между разными таблицами в реляционной базе данных хранятся явно в данных. Для указания на другую таблицу служит *внешний ключ* — набор атрибутов, которые соответствуют ключу в другой таблице.

Для работы с реляционными базами данных разработан декларативный язык структурированных запросов — Structured Query Language (SQL). Он используется как для работы с данными (DML, Data Manipulation Language), так и для работы со

схемой (DDL, Data Definition Language) и разграничением доступов (DCL, Data Control Language). Хотя язык SQL является стандартизированным, реализация СУБД может добавлять собственные конструкции, специфичные для этой реализации. Все СУБД, использующие SQL для доступа к данным, называются SQL-базами, а все остальные — NoSQL.

Реляционные СУБД реализуют подход OLTP. Наиболее популярными реализациями реляционных СУБД являются MySQL, PostgreSQL, Oracle Database, Microsoft SQL.

Стандартизированный язык запросов SQL позволяет извлекать строки данных из разных СУБД, что создает дополнительную проблему: необходимо конвертировать извлеченные данные в представления используемого языка программирования. Чаще всего извлеченные данные необходимо конвертировать в объекты.

Для решения этой проблемы используется *техника объектно-реляционного отображения* (ORM, Object-Relational Mapping), которая позволяет взаимодействовать с базами данных на уровне объектно-ориентированного кода, а не с использованием SQL-запросов. ORM автоматизирует преобразование объектов программы в строки таблиц базы данных и наоборот. В традиционном подходе работы с базой данных разработчик пишет SQL-запросы для выполнения операций с данными, а при использовании ORM эти операции абстрагируются в методы и свойства объектов, что позволяет работать с базой данных, не беспокоясь о синтаксисе SQL или специфике конкретной СУБД.

Модели в ORM — это классы, которые описывают структуру и правила взаимодействия с базой данных. Например, модель может представлять таблицу в базе данных, где каждый атрибут класса соответствует столбцу таблицы, а экземпляры класса — строкам таблицы. ORM позволяет настроить связи между моделями, например, «один ко многим» или «многие ко многим» при помощи структур данных языка программирования, что упрощает построение сложных запросов.

Когда программист использует ORM, он работает с базой данных через эти модели. Например, чтобы добавить запись в таблицу, достаточно создать объект модели и вызвать метод для сохранения. Чтение данных осуществляется через методы, которые позволяют фильтровать, сортировать и объединять записи, используя привычные объектно-ориентированные подходы вместо написания SQL-запросов. Дополнительно ORM включает механизмы для работы с миграциями базы данных. Это означает, что изменения в структуре базы данных можно описывать прямо в коде (например, добавление новых полей или изменение связей между таблицами). После этого ORM автоматически синхронизирует базу данных с описанием в моделях, упрощая управление версиями базы данных.

Одно из преимуществ ORM — переносимость — является следствием стандартизации SQL. ORM позволяет абстрагироваться от конкретной реализации СУБД, т. к. все реляционные СУБД реализуют SQL, что упрощает переносимость приложения. Однако ORM также поддерживает специфические особенности некоторых СУБД, что позволяет программисту не беспокоиться об оптимизации для конкретной СУБД. Например, вместо написания запросов, зависящих от особенностей MySQL

или PostgreSQL, вы пишете универсальный код, а ORM берет на себя задачу преобразования его в SQL, совместимый с используемой базой данных.

Однако, несмотря на удобство, подход, основанный на ORM, имеет и свои ограничения. Он добавляет слой абстракции, который может снижать производительность по сравнению с использованием «чистого» SQL. Более того, сложные операции могут генерировать неоптимальный с точки зрения производительности SQL, для исправления которого может потребоваться ручное написание запроса.

Примеры популярных библиотек ORM включают SQLAlchemy для Python, и Hibernate для Java. Некоторые ORM могут работать с нереляционными СУБД, но чаще всего нереляционные СУБД имеют собственные клиенты для разных языков программирования.

Столбцово-ориентированная модель

Столбцово-ориентированная модель данных представляет собой модификацию реляционной модели, которая заключается в способе хранения данных. В реляционной модели данные хранятся построчно, что позволяет быстро получать данные к строкам и, следовательно, соответствовать OLTP и модели ACID. В столбцовой модели данные хранятся по столбцам. То есть сначала данные первого столбца, потом данные второго и т. д. Такое хранение данных делает столбцовую модель неэффективной для маленьких изменений, но показывает большую производительность при работе с большими объемами данных.

Столбцовые СУБД реализуют подход OLAP и зачастую не полностью соответствуют модели ACID. Столбцовая модель может быть использована в некоторых реляционных СУБД: MariaDB, Microsoft SQL и Oracle Database. Также существуют СУБД, реализующие непосредственно столбцовую модель: ClickHouse, Apache Cassandra.

Документо-ориентированная модель

Документо-ориентированные базы данных позволяют хранить данные в гибком формате — чаще всего в виде JSON- или BSON-документов. Каждый документ в такой базе данных представляет собой отдельный объект, содержащий информацию о конкретной сущности, и может включать в себя любые атрибуты и вложенные структуры.

В основе документо-ориентированных баз данных лежит идея, что данные организованы вокруг документа, который содержит все необходимые сведения об объекте и может легко расширяться. Такая структура данных отлично подходит для приложений, где данные часто изменяются и имеют нефиксированную структуру. В отличие от реляционных баз данных, где добавление нового поля требует изменений в структуре таблиц, в документо-ориентированной базе данных можно просто добавить новое поле в желаемый документ. Однако в документо-ориентированных базах данных нет отношений в том же смысле, как в реляционных базах данных, — «связи» реализуются помещением одного объекта в другой объект.

Вместо SQL документо-ориентированные базы данных используют собственные API или языки запросов для поиска и фильтрации документов по различным критериям. Например, MongoDB — одна из самых популярных документо-ориентированных баз данных, позволяет выполнять сложные запросы на основе JavaScript-подобного синтаксиса. MondoDB реализует подход OLTP.

Графовая модель

Графовая модель — это особый тип NoSQL баз данных, предназначенный для работы с данными, которые тесно связаны между собой. Графовые базы данных представляют данные в виде графа, состоящего из узлов и ребер. *Узлы* представляют объекты, а *ребра* (связи между узлами) отражают отношения между этими объектами. Каждый узел и ребро могут содержать атрибуты, которые добавляют больше информации о сущностях и их связях.

Основное преимущество графовых баз данных заключается в их способности эффективно хранить и обрабатывать данные, у которых много сложных взаимосвязей. В реляционной модели такие связи между данными требуют сложных SQL-запросов, что замедляет работу системы при большом объеме данных и множестве связей. В графовых базах данных связи хранятся как объекты, поэтому запросы, требующие обхода графа, обрабатываются намного быстрее. Это делает графовые базы данных идеальным выбором для задач, где важны не только сами данные, но и их взаимосвязи.

Запросы к графовым базам данных выполняются с помощью специализированных языков. Популярные графовые базы данных: Neo4j и Apache TinkerPop. Эти базы данных реализуют подход OLTP.

Ключ-значение

Это простейший и один из самых производительных типов NoSQL баз данных, где данные хранятся в виде пар «ключ-значение». Каждая запись в такой базе представлена уникальным ключом, ассоциированным с определенным значением. Ключи служат идентификаторами для значений, а сами значения могут быть любыми данными: строками, числами, JSON-объектами или даже сериализованными сложными структурами. Такой формат хранения делает базы данных «ключ-значение» особенно быстрыми, поскольку для поиска значения необходимо просто найти его по ключу, без сложных операций соединения или анализа структуры, как в реляционных или графовых базах данных.

Благодаря своей структуре базы данных «ключ-значение» позволяют обрабатывать миллионы запросов в секунду, что делает их идеальными для приложений, требующих минимальной задержки при доступе к данным. Поэтому такие базы данных часто используются в качестве системы кеширования. Наиболее популярные базы данных «ключ-значение»: Redis и memcached.

Базы данных для временны́х рядов

Хотя для хранения временны́х рядов не выделяют отдельной модели, в информационных системах есть потребность в хранении данных, привязанных к определенной временно́й метке. Для такой задачи используются существующие модели данных с оптимизациями под особенности временны́х рядов. Наиболее популярные решения для хранения временны́х рядов: TimescaleDB (расширение для PostgreSQL) и InfluxDB.

* * *

Базы данных обычно хранят данные, необходимые для работы любой информационной системы. Потеря этих данных может приводить к внезапному завершению жизненного цикла программного обеспечения, а также обязательно негативно скажется на репутации тех, кто занимался поддержкой «упавшей» базы данных. Чтобы этого не случилось, необходимо делать резервные копии данных. Тема резервного копирования баз данных обсуждается в *главе 10*.

Брокер сообщений

При проектировании архитектуры, которая включает в себя более одного независимого сервиса, возникает вопрос надежной коммуникации между сервисами. В самом простом случае можно использовать протокол HTTP и обмениваться информацией с его помощью. Всё меняется, если наложить условия на гарантию доставки сообщений. Что делать, если сообщение нужно точно доставить, а принимающий сервис занят и сообщения отбрасывает?

Брокер сообщений — это промежуточное программное обеспечение, которое управляет обменом сообщениями между различными компонентами информационной системы. Он служит «посредником», который принимает сообщения от отправителя (производителя), сохраняет их и передает получателю (потребителю). Брокеры сообщений помогают различным частям системы общаться друг с другом асинхронно и независимо, что делает их идеальными для распределенных систем, где важно обеспечить надежность передачи данных, масштабируемость и гибкость.

Брокеры сообщений поддерживают несколько режимов доставки: «очереди» и «топики» (от *англ.* topic — тема). В *режиме очередей* каждое сообщение доставляется одному потребителю, что полезно, когда требуется разделение задач между несколькими обработчиками. В *режиме топиков* сообщения доставляются всем подписчикам, что подходит для уведомлений, где один источник данных должен обновлять сразу несколько систем или пользователей.

Главная функция брокера сообщений — обеспечение доставки сообщений. Брокеры поддерживают системы, где отправитель и получатель могут работать независимо друг от друга, т. е. не обязаны быть активными в одно и то же время. Очередь действует как буфер, где сообщения хранятся до момента, пока получатель не будет готов их обработать. Это особенно полезно в ситуациях, когда скорость отправки сообщений превышает скорость их обработки или когда отправитель и

получатель работают в разное время. Такой подход помогает минимизировать зависимость между компонентами системы и делает ее более устойчивой к сбоям.

Важной характеристикой очередей является порядок обработки сообщений. Чаще всего они работают по принципу FIFO (First In, First Out) — это означает, что сообщения обрабатываются в том же порядке, в каком они были отправлены. Однако в некоторых системах возможны и другие стратегии — например, приоритетные очереди, где сообщения с более высоким приоритетом обрабатываются раньше.

Очереди могут быть как точкой взаимодействия одного отправителя с одним получателем, так и поддерживать сложные топологии. Например, в режиме fan-out одно сообщение, помещенное в очередь, доставляется сразу нескольким получателям. Это используется для реализации систем публикации и подписки, где различные компоненты получают информацию, соответствующую их интересам.

Брокеры могут анализировать метаданные или содержимое сообщений и принимать решения о том, каким получателям они должны быть отправлены. Многие брокеры также поддерживают топологии маршрутизации. Например, сообщения — чтобы достичь получателей — могут проходить через несколько узлов или обменников. Это позволяет строить сложные сети взаимодействий, где сообщения могут быть переработаны или преобразованы на промежуточных этапах.

Для обеспечения надежности работы очередей брокеры сообщений используют механизмы подтверждения и повторной доставки. Когда получатель извлекает сообщение, он подтверждает его обработку. Если подтверждение не поступает, сообщение может быть возвращено в очередь для повторной попытки обработки. Это гарантирует, что ни одно сообщение не будет потеряно, даже если получатель выйдет из строя. Сообщения передаются согласно одной из трех стратегий:

- Не более одного раза.

  Это наименее надежная модель, при которой сообщение может быть доставлено либо один раз, либо вообще не доставлено. Такая стратегия подразумевает, что если в процессе передачи или обработки произошла ошибка, то сообщение может быть потеряно. Этот уровень подходит для ситуаций, когда данные не критичны и потеря сообщения не повлияет на работу системы. Примером стратегии может быть передача метрик мониторинга, где отсутствие одного из показателей не приведет к сбоям.

- Не менее одного раза.

  Сообщение будет доставлено один или несколько раз — т. е. оно гарантированно дойдет до получателя, но возможны дублирования. В случае ошибки брокер сообщений будет повторно отправлять сообщение, пока не получит подтверждение о его доставке. Такая модель используется, когда важнее получить данные, даже если это произойдет несколько раз.

- Точно один раз.

  Это самая надежная и сложная в реализации модель, при которой сообщение доставляется ровно один раз. Брокер сообщений гарантирует, что сообщение будет доставлено только один раз и не будет продублировано, даже в случае

сбоев. Эта модель особенно востребована в финансовых и банковских приложениях, где дублирование транзакций недопустимо. Для реализации такой гарантии требуются механизмы подтверждения на нескольких уровнях, включая запись статуса доставки, что делает ее более ресурсоемкой по сравнению с другими уровнями.

Очереди также обладают настройками для управления временем хранения сообщений. Если сообщение не обработано в течение заданного времени, оно может быть удалено или перемещено в специальную «мертвую» (dead) очередь для последующего анализа. Это позволяет эффективно использовать ресурсы и предотвращать переполнение очередей.

Среди популярных брокеров сообщений можно выделить RabbitMQ, Apache Kafka, ActiveMQ и Eclipse Mosquitto.

Параллельные вычисления

Параллельные вычисления — это метод обработки данных, при котором задачи или их отдельные части выполняются одновременно на нескольких процессорах или ядрах. Такой подход применяется для ускорения вычислений, повышения эффективности использования ресурсов и решения задач, которые требуют значительных вычислительных мощностей. Основная идея параллелизма заключается в разделении работы на более мелкие подзадачи, которые могут выполняться одновременно, а затем объединении их результатов.

Закон Амдала

Хотя может показаться, что использование параллельных вычислений является «серебряной пулей» для ускорения любых вычислений, это не так. Более формально проблему ускорения описывает *закон Амдала*: ускорение задачи ограничено количеством кода, исполняемого последовательно. Представим задачу, которая состоит из двух частей: параллельной и последовательной. Чем больше мы добавим вычислителей (серверов, потоков, сервисов и т. п.), тем быстрее выполнится параллельная часть, потому что этой задачей занимается больше вычислителей. При этом последовательная часть всегда будет выполняться на одном вычислителе за фиксированное время, вне зависимости от количества выделенных на параллельную часть вычислителей. Поэтому, даже если вычислителей будет бесконечное количество, а параллельная часть станет выполняться практически мгновенно, общее время выполнения задачи будет определяться временем последовательной части.

Закон Амдала подчеркивает, что эффективность параллельных вычислений напрямую зависит от доли последовательного кода. Соответственно, добавление большего числа вычислителей не всегда приводит к линейному росту производительности. При организации параллелизма информационные системы сталкиваются с дополнительными накладными расходами на управление, синхронизацию и передачу данных между вычислителями, что еще больше снижает эффективность.

Многопоточность

Наиболее простой способ организации параллельных вычислений — *многопоточность*. В многопоточности несколько потоков разделяют память и ресурсы процесса, но при этом каждый выполняет свой код. Это позволяет им выполнять разные части программы одновременно, обмениваясь данными через общие ресурсы. Доступ к общим (разделяемым) ресурсам в параллельных вычислениях — это еще одна цена за возможное ускорение.

Поскольку потоки разделяют память, они могут обращаться к одним и тем же данным одновременно. Это может привести к состояниям *гонки*, когда результат выполнения программы зависит от порядка доступа потоков. Для предотвращения таких проблем используются *механизмы синхронизации*. Однако механизмы синхронизации — это последовательный код, который вынуждает потоки ждать завершения работы другого потока при работе с общим ресурсом. Помимо состояния гонки при отсутствии синхронизации, в многопоточных системах существует и другая проблема — взаимная блокировка (*англ*. deadlock), при которой потоки бесконечно ожидают друг друга. Представим, что выполняются поток 1 и поток 2, и каждый из них хочет обратиться к общим ресурсам А и Б. Поток 1 через механизмы синхронизации заявляет об обращении к А, а поток 2 в этот же момент заявляет об обращении к Б. Когда же поток 1 обращается к Б, механизм синхронизации предлагает ему подождать, пока поток 2 освободит Б. В этот же момент поток 2 обращается к ресурсу А и также получает предложение подождать, когда поток 1 освободит А. В такой ситуации потоки будут ожидать бесконечно. Отладка проблем с синхронизацией может стать весьма трудоемкой задачей.

Обратите внимание, что поток — это «академический» перевод английского термина thread. Однако этот перевод нарушает однозначность, т. к. конфликтует с потоками данных (stream). Альтернативные термины для потоков из темы многопоточности — «нить» или «тред».

Синхронизация в многопоточных программах

Для предотвращения состояния гонок и взаимных блокировок в многопоточных программах используются различные механизмы синхронизации, обеспечивающие согласованность данных и корректное поведение программы.

- Один из наиболее распространенных методов синхронизации — использование *мьютексов* (mutex). Они действуют как замки, которые поток устанавливает перед доступом к общему ресурсу и снимает после завершения работы. Пока мьютекс заблокирован одним потоком, другие потоки вынуждены ждать, что предотвращает одновременное изменение данных.
- *Семафоры* (semaphore) предоставляют более гибкий способ управления доступом. Они работают со счетчиком, который определяет, сколько потоков одновременно могут использовать ресурс. Это особенно полезно для ограничения числа одновременно работающих потоков — например, при доступе к сетевым соединениям или базам данных.

- *Условные переменные* (condition variables) позволяют потокам взаимодействовать друг с другом более гибко. Каждый поток может ждать определенного условия и возобновить работу, как только это условие станет истинным.
- Также применяются *механизмы блокировки чтения и записи* (rwlock), которые разделяют доступ к ресурсу на две эти категории. Потоки могут читать данные одновременно, но запись блокируется до завершения всех операций чтения, что помогает оптимизировать производительность в сценариях с частым чтением и редким изменением данных.

Псевдопараллелизм

Псевдопараллелизм создает иллюзию одновременного выполнения нескольких задач, хотя на самом деле процессор выполняет их последовательно, переключаясь между ними. Этот подход основан на концепции разделения времени и используется в системах с вычислительным процессором (ядром). Каждой задаче выделяется небольшой промежуток времени, называемый *квантом времени*. Когда этот промежуток истекает или происходит событие — например, запрос на ввод/вывод, процессор переключается на выполнение другой задачи. Такие переключения настолько быстрые, а кванты времени настолько маленькие в человеческих мерках, что пользователю кажется, будто все процессы выполняются одновременно.

Хотя псевдопараллелизм формально не имеет отношения к параллельным вычислениям, он позволяет программисту решать разные задачи изолированно друг от друга и эффективно использовать доступные вычислительные ресурсы.

Обратимся к упомянутому ранее закону Амдала — он оперирует теоретическими понятиями, позволяя бесконечно увеличивать количество вычислителей. В реальной системе количество вычислителей ограничено процессором и количеством его ядер. Если количество работающих потоков больше, чем количество ядер, то операционная система будет использовать псевдопараллелизм для размещения нескольких потоков на одном ядре. Это не только не ускорит выполнение программы, но, наоборот, замедлит из-за необходимости переключать контексты.

* * *

Для организации в программе многопоточности существует несколько подходов. Более гибкий, но более сложный — непосредственное создание и управление потоками через интерфейсы операционной системы. Этот способ позволяет запускать параллельно практически всё что угодно, но требует от программиста реализовать логику запуска потоков, коммуникацию между ними и ожидание завершения. Другой способ — использование средств для распараллеливания программ наподобие OpenMP, представляющего собой набор директив препроцессора языков C, C++ и Fortran, указывающих компилятору сгенерировать код для распараллеливания. Это ограниченный вариант, который позволяет легко распараллеливать однотипные задачи, но не требует от программиста глубоких познаний в многопоточности.

Сопрограммы (корутины)

Сопрограммы — это особый вид функций или программных конструкций, которые позволяют приостанавливать и возобновлять свое выполнение в определенных точках и передавать управление в другие сопрограммы или управляющему коду. Это удобно при работе с событиями — например, при обработке пользовательского ввода, т. к. позволяет не блокировать поток в ожидании нажатия кнопки.

В отличие от псевдопараллелизма, в котором каждому потоку выделяется квант времени и за соблюдением справедливого распределения времени наблюдает планировщик процессов, сопрограммы передают управление только согласно своей логике. Таким образом, сопрограмма может занять больше одного кванта времени и долгое время не отдавать управление другим сопрограммам (рис. 5.1). При этом ничто не сможет повлиять на эту несправедливость.

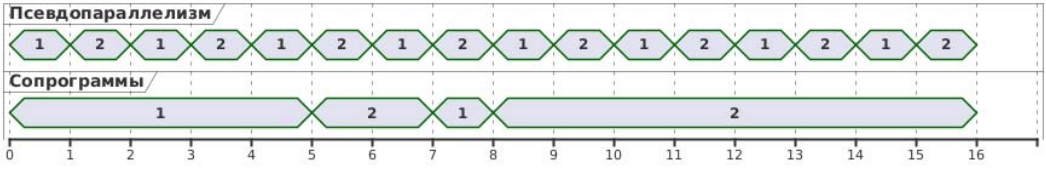


Рис. 5.1. Псевдопараллелизм и сопрограммы

Сопрограммы также называются *легковесными потоками*, т. к., с точки зрения программиста, это разные потоки, но для операционной системы все сопрограммы выполняются в одном «настоящем» потоке. В англоязычной среде сопрограммы называются *корутинами* (coroutine), а легковесные потоки — *волокнами* (fiber). В языке программирования Go используется особенное название — *горутины* (goroutine). Реализация и особенности сопрограмм и легковесных потоков зависят в основном от стандартной библиотеки используемого языка программирования.

Многопроцессорные вычисления

В этом подходе задачи распределяются между несколькими процессами, работающими одновременно. В отличие от многопоточности, где потоки имеют общую память внутри одного процесса, многопроцессные вычисления изолируют память каждого процесса. Эта изоляция делает их более устойчивыми к ошибкам, т. к. сбой в одном процессе не затрагивает другие. Многопроцессные вычисления хорошо подходят для программ, которые могут быть разбиты на независимые задачи. Например, в веб-сервере каждый процесс может обрабатывать отдельный запрос клиента. Для координации работы процессов используется межпроцессное взаимодействие (Inter-Process Communication, IPC).

Межпроцессное взаимодействие

Поскольку процессы изолированы друг от друга, прямой доступ к ресурсам в памяти другого процесса невозможен. Однако IPC предоставляет инструменты для обмена информацией и синхронизации.

Один из основных подходов к межпроцессному взаимодействию — использование каналов связи, которые передают информацию между процессами. Это могут быть однонаправленные каналы — именованные и неименованные *конвейеры* (pipe), либо двунаправленные каналы — *сокеты*. Конвейеры и сокеты обеспечивают доставку сообщений между процессами.

Еще один распространенный метод — *общая память*. В этом случае операционной системой выделяется небольшая область памяти, доступная сразу нескольким процессам. Процессы могут записывать и читать данные из этой области, что обеспечивает быструю передачу информации. Однако при использовании общей памяти требуется дополнительная синхронизация, как при многопоточных вычислениях.

Очереди сообщений предоставляют механизм для обмена данными, где сообщения помещаются в очередь одним процессом и извлекаются другим. Обратите внимание, что для межпроцессного взаимодействия задействуются механизмы очередей, которые реализованы в ядре используемой операционной системы. Эта реализация схожа с ранее рассмотренными брокерами сообщений, но существует непосредственно в операционной системе и имеет более узкую область применения.

Сигналы представляют собой минимальные стандартизированные сообщения, передаваемые процессу для уведомления о каком-либо событии — например, завершении другой программы или необходимости перезапуска. Сигналы применяются для управления процессами и выполнения простых операций.

Распределенные вычисления

Распределенные вычисления — частный случай параллельных вычислений, при котором задачи выполняются на нескольких независимых компьютерах, объединенных в сеть. Каждая машина в такой системе выполняет часть общей работы, взаимодействуя с другими через сеть для обмена данными и координации. Этот подход позволяет использовать совокупную мощность многих устройств, что делает его особенно эффективным для решения сложных задач, требующих больших вычислительных ресурсов. Вычислительные узлы могут находиться в разных местах, иметь различные конфигурации и операционные системы, но они работают совместно как единое целое.

В отличие от многопоточных вычислений, ограниченных производительностью одного компьютера, распределенные вычисления можно масштабировать практически неограниченно. Примером распределенных вычислений может служить проект GIMPS (Great Internet Mersenne Prime Search) по поиску простых чисел Мерсенна[1]. Добровольцы, принимающие участие в этом проекте, могут предоставить свои вычислительные ресурсы для поиска простых чисел, которые вычисляются по формуле $2^P - 1$. Проект основан в 1996 году и действует до сих пор. Наиболее актуальное достижение на момент подготовки книги (октябрь 2024 года) — найдено число длиной в 41 миллион знаков.

[1] См. **https://goo.su/sfJ376Z**.

Проект по поиску простых чисел частично реализует *парадигму MapReduce* для обработки больших данных. Она основана на разделении работы на две стадии: map и reduce, что позволяет эффективно распределить задачу между множеством узлов в системе. На стадии map данные разбиваются на небольшие части, которые распределяются между узлами кластера. Каждый узел применяет заданную функцию к своей части данных, преобразуя их или извлекая из них полезную информацию. Результатом этой стадии становятся результаты локальных вычислений. Затем начинается стадия reduce. Здесь результаты, полученные от разных узлов, группируются, а затем обрабатываются для получения итогового результата. Эта стадия собирает, агрегирует или комбинирует данные, завершая процесс обработки. Например, если задача заключается в подсчете числа слов в большом наборе текстов, на этапе map каждое слово подсчитывается в отдельной части текста, а на этапе reduce результаты объединяются для получения общего количества.

Парадигма MapReduce используется в основном для обработки данных, однако это не единственная задача, которая решается распределенными системами. Помимо «локальной» распределенности, как в микросервисном подходе, системы могут быть распределенными и географически. Так, географическая распределенность позволяет пользователям получать данные с ближайших серверов, что уменьшает время доступа к данным и снижает нагрузку на магистральных провайдеров связи. Примером географически распределенной системы может служить сеть доставки контента Content Delivery Network (CDN). Географическая распределенность может быть связана и с бизнес-процессами, и обеспечением надежности. Например, региональные филиалы банков могут иметь локальные экземпляры большой банковской системы, а все региональные экземпляры составляют большую распределенную сеть. В отличие от парадигмы MapReduce, в таких системах тоже нужна синхронизация.

Синхронизация в распределенных системах

В распределенных системах узлы взаимодействуют через сеть с непредсказуемыми задержками, и каждый узел имеет собственные часы, которые могут быть не синхронизированы. Эти особенности создают уникальные проблемы, связанные с синхронизацией, особенно в географически распределенных системах.

У каждого вычислительного узла в распределенной системе существует свое представление о виртуальном глобальном состоянии всей системы в целом. Важно поддерживать это представление одинаковым у всех участников распределенной системы. Проблема в том, что оповещения о событиях, произошедших у определенных участников распределенной системы, могут в разном порядке приходить другим участникам системы из-за различных задержек в канале связи. Для упорядочивания событий в распределенной системе используется логическое время: скалярное, векторное и матричное.

У распределенных систем, как и у многопоточных, и многопроцессорных, могут быть свои общие ресурсы, доступ к которым должен иметь не более чем один узел в каждый момент времени. Для этого используются *алгоритмы взаимного исклю-*

чения, которые позволяют равноправным участникам распределенной системы договариваться о доступе к общему ресурсу.

Тема алгоритмов в распределенных системах достаточно обширна и выходит за рамки теоретического введения, необходимого для проектирования информационных систем. Однако с этими алгоритмами можно ознакомиться в учебном пособии М. Косякова «Введение в распределенные вычисления»[1].

Вызов функций в распределенных системах

RPC (Remote Procedure Call, удаленный вызов процедур) — это технология, позволяющая программам вызывать функции или процедуры на удаленных системах так, будто они находятся локально. Основная цель RPC заключается в упрощении взаимодействия между распределенными компонентами системы, скрывая сложность передачи данных по сети.

Когда клиент вызывает удаленную процедуру, RPC автоматически обрабатывает передачу аргументов вызова через сеть на сервер, где процедура исполняется. После выполнения сервер возвращает результат обратно клиенту. Это осуществляется благодаря генерации специального кода, который включает в себя часть на клиентской стороне, называемую *заглушкой* (stub), и *скелет* (skeleton) на серверной. Клиентская часть формирует запрос, упаковывает данные, а серверная часть получает запрос, выполняет процедуру и формирует ответ.

RPC базируется на идее прозрачности местоположения: вызов удаленной процедуры не сильно отличается от локального вызова, что снижает сложность разработки распределенных систем. Тем не менее использование RPC имеет свои ограничения. Например, в отличие от локальных вызовов, удаленный вызов может быть подвержен сетевым задержкам, отказам или несоответствию данных между клиентом и сервером.

Существует множество реализаций RPC, включая классические технологии вроде ONC RPC, современные реализации на базе JSON-RPC или gRPC, а также протоколы, использующие HTTP/2 и Protocol Buffers для улучшения производительности и совместимости.

GraphQL

GraphQL — это язык запросов для взаимодействия с API, а также серверный инструмент для их обработки. Он был создан в 2012 году и опубликован как open-source в 2015 году. GraphQL предоставляет клиентам гибкий и эффективный способ работы с данными, позволяя запрашивать только те данные, которые действительно нужны.

В основе GraphQL лежит схема, представляющая собой описание всех типов данных, доступных через API, а также их связей и возможных операций. Эта схема иг-

[1] См. **https://goo.su/HFMjq**.

рает роль интерфейса между клиентом и сервером. На ее основе клиент формирует запросы, которые сервер валидирует и выполняет. Благодаря этому исключаются ошибки, связанные с неправильными запросами. В GraphQL клиент может задавать структуру ответа. Если в традиционных REST API клиент получает заранее определенный набор данных для каждого эндпоинта, то в GraphQL клиент сам указывает, какие поля ему нужны, и сервер возвращает только их. Это уменьшает избыточность данных, снижает нагрузку на сеть и упрощает разработку. GraphQL поддерживает три основных типа операций:

- Запросы (queries).

  Используются для получения данных. Клиент формирует запрос с описанием полей, вложенных объектов и аргументов, а сервер возвращает данные в точно таком же формате.

- Мутации (mutations).

  Предназначены для изменения данных на сервере. Они могут использоваться для создания, обновления или удаления записей. Результатом выполнения мутации обычно является измененный объект или подтверждение операции.

- Подписки (subscriptions).

  Позволяют клиенту получать обновления в реальном времени при изменении данных на сервере. Это достигается с использованием технологий типа WebSocket.

GraphQL имеет строгую типизацию, благодаря которой схема API однозначно определяет, какие типы данных могут быть возвращены или отправлены. Это упрощает тестирование, документацию и обнаружение ошибок на этапе разработки. Для работы с вложенными структурами данных GraphQL использует концепцию *резолверов* (resolvers) — функций, которые отвечают за извлечение данных для конкретного поля или типа. Это позволяет API объединять данные из разных источников, таких как базы данных, сторонние API или кеш.

GraphQL задействует унифицированный эндпоинт для всех операций вместо множества URL-адресов, как в REST. Однако эта лаконичность является как достоинством, так и недостатком. Единый URL усложняет кеширование, поскольку запросы формируются клиентом и могут быть уникальными, а стандартные механизмы HTTP-кеширования становятся менее эффективными. Кроме того, гибкость запросов может привести к избыточной нагрузке на сервер, если клиент запрашивает слишком много данных или делает сложные вложенные запросы.

WebSocket

У протокола HTTP есть недостаток: невозможность обмениваться данными в реальном времени. Клиент отправляет запрос, а сервер отвечает. В этой схеме у сервера нет возможности отправить клиенту данные, как только они будут готовы. Из-за этого в классическом HTTP для проверки готовности какой-либо фоновой операции делают запросы через определенные промежутки времени. Это, очевидно, создает нагрузку на сеть.

Для устранения этого недостатка был придуман *«длинный опрос»* (Long polling) — технология взаимодействия между клиентом и сервером, которая используется для получения данных с сервера в режиме реального времени. Она отличается от обычного HTTP-запроса тем, что клиент инициирует запрос, а сервер удерживает соединение открытым до тех пор, пока не появится новая информация для передачи.

Клиент при этом отправляет запрос на сервер и ждет ответа, не закрывая соединение. Сервер, в свою очередь, не отвечает сразу, а задерживает ответ до момента, когда у него появятся данные, которые нужно отправить клиенту. Как только данные будут переданы, соединение закрывается, а клиент немедленно отправляет новый запрос, чтобы снова быть готовым принять информацию.

Такая схема эффективна в условиях, когда обновления происходят нерегулярно, т. к. она минимизирует частоту запросов по сравнению с постоянным опросом сервера, но при этом сохраняет возможность получения данных почти в реальном времени. Однако Long polling не является идеальным решением, поскольку на один запрос отправляется один ответ. Длинные запросы позволяют отказаться от периодических запросов, но всё еще не позволяют серверу отправлять данные клиенту по своему желанию. Этот недостаток решается в WebSocket.

WebSocket — это протокол связи, обеспечивающий постоянное двустороннее взаимодействие между клиентом и сервером. Он позволяет устанавливать долговременный канал связи по одному TCP-соединению, что делает его более эффективным для передачи данных в режиме реального времени по сравнению с традиционными методами, такими как HTTP-запросы.

Передаваемые данные в WebSocket кодируются в компактном бинарном формате, что снижает накладные расходы на передачу. По сравнению с HTTP, где каждое сообщение требует полного набора заголовков, WebSocket позволяет минимизировать объем служебной информации. Это делает протокол более производительным при частом обмене небольшими сообщениями.

WebSocket поддерживает как текстовые, так и бинарные данные, а также может быть защищен с использованием TLS. Для управления состоянием соединения WebSocket задействует пинг- и понг-сообщения, которые позволяют проверить доступность другой стороны и поддерживать связь активной.

Одной из сложностей при работе с WebSocket является необходимость обработки закрытия соединений и обработки ошибок. Соединение может разорваться по разным причинам, включая сетевые сбои или намеренное завершение сеанса. Клиенты и серверы должны быть готовы к восстановлению соединения или корректному завершению обмена данными.

WebSocket активно используется в современных веб-приложениях благодаря своей низкой задержке и высокой производительности, особенно в сценариях, где традиционные запросы на основе HTTP были бы слишком медленными или избыточными.

Кеширование

Кеширование — это процесс хранения часто запрашиваемых данных в быстрой памяти, позволяющий сократить время на их повторное получение и уменьшить нагрузку на системы, которые генерируют эти данные. Основная идея кеширования заключается в том, чтобы данные, которые ранее были получены или рассчитаны, могли быть быстро доступны без повторного выполнения ресурсоемких операций.

Кеш может быть реализован на различных уровнях. На уровне процессора существуют аппаратные кеши, которые хранят данные, недавно использованные для выполнения инструкций, ускоряя работу программ. На уровне программного обеспечения кеширование широко применяется в интернет-приложениях, базах данных и сетевых системах для повышения производительности. В программном обеспечении также есть кеши на разных уровнях. Например, браузеры хранят кеш статических ресурсов, таких как изображения, стили и скрипты, чтобы ускорить загрузку веб-страниц. Промежуточные прокси-серверы и CDN (Content Delivery Networks) кешируют ресурсы для доставки контента пользователям из ближайших к ним серверов. В базах данных применяются механизмы кеширования запросов и индексов, дающие возможность минимизировать нагрузку на дисковую подсистему.

Для управления кешированием используются различные стратегии кеширования и политики инвалидации (вытеснения) данных в кеше.

Кеширование при чтении (сквозное чтение, read-through)

При этой стратегии кеширования данные извлекаются из источника и добавляются в кеш при первом запросе. Если данные уже есть в кеше, они при следующем запросе возвращаются немедленно, минуя источник (рис. 5.2). Это удобный подход для часто запрашиваемой информации, т. к. кеширование происходит только при необходимости, а ресурсы не тратятся на данные, которые могут никогда не понадобиться. В этом случае кеш «прозрачен» для приложения.

Кеширование при записи (сквозная запись, write-through)

Эта стратегия предполагает, что данные при обновлении сразу записываются как в кеш, так и в основной источник (рис. 5.3). Такой подход обеспечивает консистентность между кешем и базой данных, но может снизить производительность из-за необходимости синхронной записи.

Чтение на стороне (Read aside)

В этой стратегии, в отличие от сквозного чтения, приложение обращается к источнику данных напрямую и при необходимости самостоятельно помещает данные в кеш (рис. 5.4).

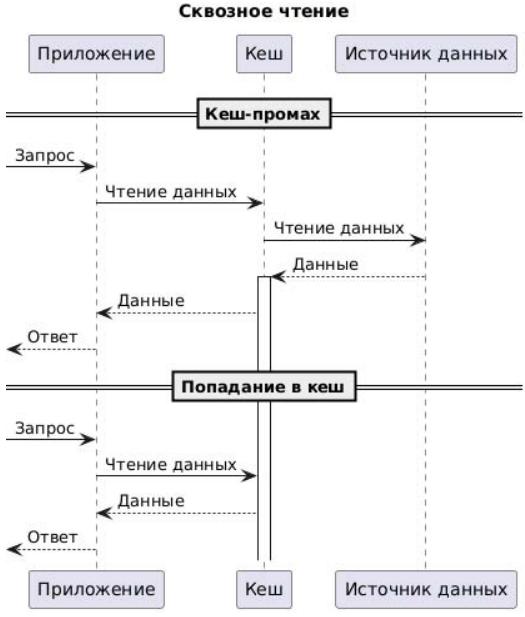


Рис. 5.2. Кеширование при чтении (сквозное чтение)

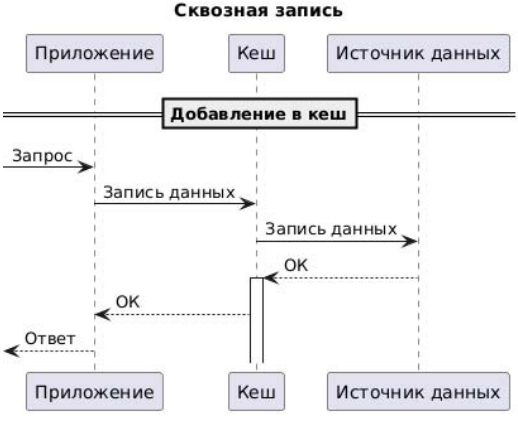


Рис. 5.3. Кеширование при записи (сквозная запись)

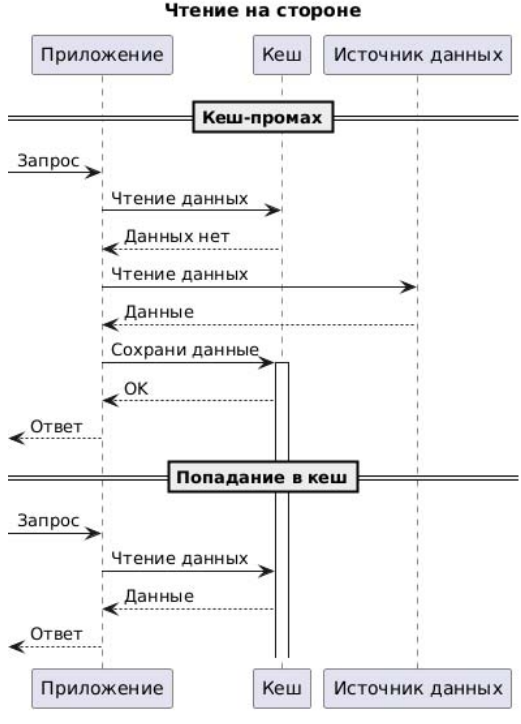


Рис. 5.4. Чтение на стороне

Запись на стороне (Write aside)

Аналогично чтению на стороне в стратегии записи на стороне приложение явно кладет в кеш данные при операции записи (рис. 5.5).

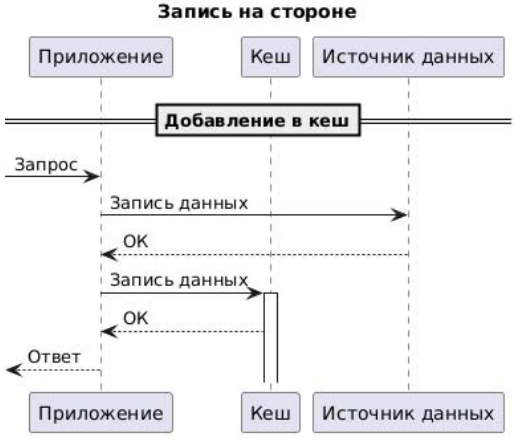


Рис. 5.5. Запись на стороне

Кеширование на опережение (Cache Ahead, Refresh Ahead)

Кеширование на опережение (рис. 5.6) подразумевает, что кеш самостоятельно, в фоновом режиме, обращается к источнику данных и обновляет кешированные значения без обращений от приложения, предполагая, что приложение в ближайшем будущем запросит уже кешированные данные.

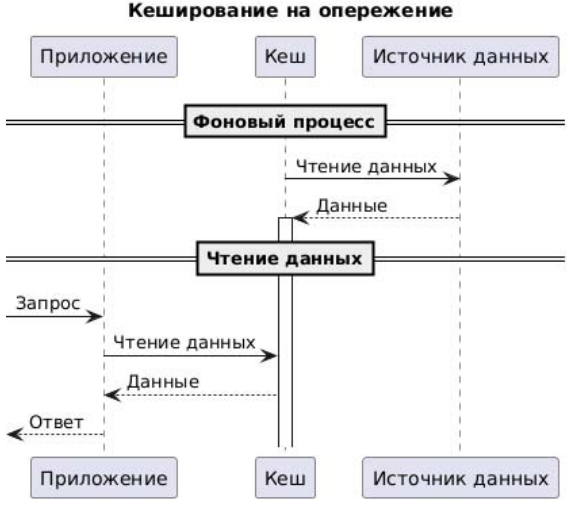


Рис. 5.6. Кеширование на опережение

Политики инвалидации

Поскольку кеш — это не бесконечный ресурс, то применяются *политики инвалидации* — вытеснения устаревших и неактуальных данных.

- FIFO (First In, First Out).

  Простейшая политика инвалидации кеша, которые реализует очередь. Очередь работает по принципу «первый вошел — первый обслужен». Кеш на очереди помещает новое значение в конце очереди и удаляет самое старое значение из начала очереди. Эта стратегия никак не учитывает «популярность» кешированных данных.

- LRU (Least Recently Used).

  Кеш записывает время последнего обращения к каждому элементу, и когда придет время, удалению подлежат те элементы, к которым давно не обращались. Эта политика часто используется как стратегия по умолчанию.

- LFU (Least Frequently Used).

  Кеш учитывает количество обращений к элементам и отмечает для удаления те, к которым обращались реже всего.

- TTL (Time-to-Live).

  Значения в кеше имеют ограничение по времени и удаляются по истечении указанного времени.

- MRU (Most Recently Used).

  Специфическая стратегия, вытесняющая недавние элементы и сохраняющая элементы, к которым давно не обращались.

- Сегментированные политики.

  Сегментированные политики кеширования предполагают несколько уровней кеша. Например, SNLRU — это три уровня кеша с политикой LRU. При частом кеш-попадании значение переносится на более «горячий» уровень. В результате каждый уровень хранит данные согласно их популярности: самый «холодный» уровень — наиболее популярные среди редко используемых данных, а самый «горячий» — самые популярные среди популярных. Это позволяет эффективно хранить разнородные данные и предотвращает конфликт за место между часто используемыми данными и очень часто используемыми данными. В эту категорию также входят 2Q и MQ.

- Сложные алгоритмы вытеснения.

  В некоторых случаях базовые политики вытеснения не справляются, и тогда применяются более сложные алгоритмы. Отметим их в порядке упоминания: ARC (Adaptive Replacement Cache), CLOCK и GCLOCK. Политика вытеснения ARC используется в файловой системе ZFS, СУБД PostgreSQL и в объектном хранилище VMware vSAN.

Идентификация, аутентификация и авторизация

Авторизация, аутентификация и идентификация — три ключевых процесса в обеспечении информационной безопасности, которые часто путают, хотя каждый из них имеет свое значение и роль.

Идентификация

Этап, на котором система определяет, кто пытается получить доступ (рис. 5.7). Это как предъявление документа, удостоверяющего личность, но без его проверки. Например, когда рыцарь вводит свой идентификационный номер, он заявляет системе: «Я — Примеров Иван Иванович». Это не подтверждение, что рыцарь именно Примеров Иван Иванович, а просто заявление о личности.

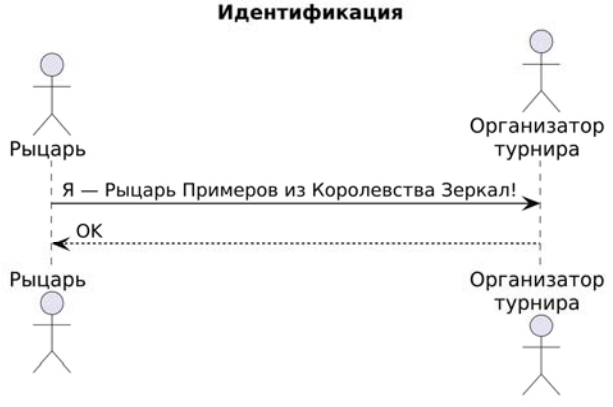


Рис. 5.7. Идентификация

Аутентификация

Отвечает за проверку достоверности того, что пользователь действительно тот, за кого себя выдает (рис. 5.8). Здесь проверяются учетные данные и удостоверяется, что они соответствуют ранее установленным критериям. Например, после ввода имени идентификационного номера рыцаря система просит пароль, который знает только истинный владелец учетной записи. Современные методы аутентификации включают не только пароли, но и биометрические данные — такие как отпечатки пальцев или распознавание лица, а также одноразовые коды (OTP).

Авторизация

Следующий этап после успешной аутентификации (рис. 5.9). Она определяет, какие действия и ресурсы доступны пользователю. Даже если система знает, кто вы, и уверена в вашей личности, это не значит, что вам автоматически позволено делать

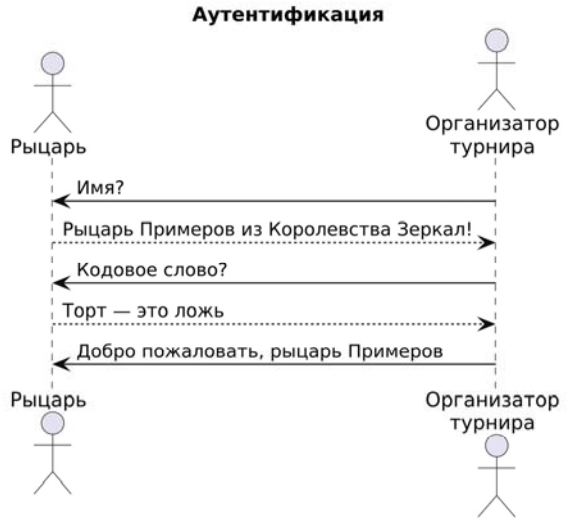


Рис. 5.8. Аутентификация

всё. Например, рыцарь авторизован записать себя на турнир, но не авторизован изменять время проведения турнира. Таким образом, авторизация устанавливает границы полномочий пользователя в рамках системы.

Каждый из этих процессов связан с другим, но выполняет свои уникальные функции, обеспечивая безопасное и управляемое использование системы.

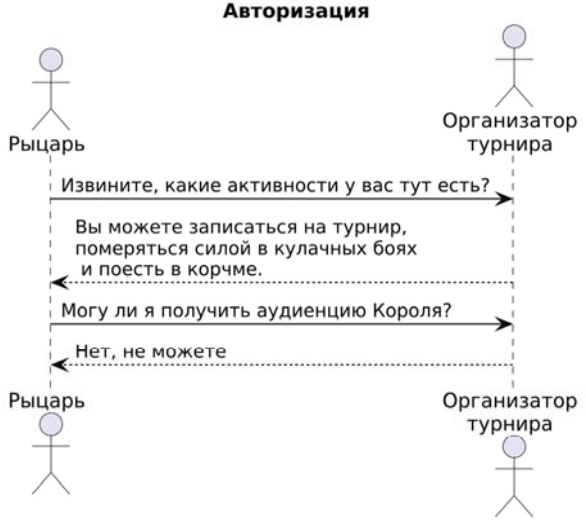
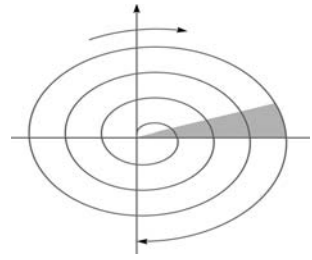


Рис. 5.9. Авторизация

ГЛАВА 6



Тестирование

Тестирование — важная часть разработки программного обеспечения. Основная цель тестирования — повышение уровня доверия к программному обеспечению в том смысле, что оно функционирует корректно при любых возможных обстоятельствах. Хотя тестирование — это шаг, который не добавляет функциональности, но занимает время, отсутствие тестирования может привести к катастрофическим последствиям.

Так, 4 июня 1996 года ракета-носитель «Ариан-5» (Arian-5) была уничтожена на 34-й секунде полета из-за ошибки в модуле управления, зависшем при конвертации числа. Подпрограмма, которая могла бы обработать ошибку, была удалена во время разработки в угоду быстродействию системы. В июле 2024 года компания CrowdStrike выпустила обновление для своего антивирусного обеспечения, в котором содержался дефект, приводящий к аварийному завершению операционной системы и невозможности запустить ОС. Этот инцидент привел к отмене более 5000 авиарейсов по всему миру, что составляет около 4,6% всего их количества. Дефектное обновление также повлияло в том числе и на медицинские, и на финансовые информационные системы. В официальном пресс-релизе было указано, что дефект успешно прошел все автоматические тесты.

Все эти и другие возможные примеры демонстрируют, что тестирование может выявить наличие дефектов в программном обеспечении, но не может доказать их отсутствие. Рассмотрим этот вывод на простом примере функции, которая принимает два числа и выдает результат:

```
public long sum(int a, int b) {
  return a + b;
}
```

Чтобы практически доказать, что эта функция лишена дефектов, нужно проверить результат выполнения функции при всех комбинациях входных параметров. В указанной функции это возможно: целочисленный тип int в Java имеет ограничение — он принимает значения от –2147483648 до 2147483647. При этом количество всех комбинаций значений двух параметров примерно равно $4{,}6 \times 10^{18}$. Если проверять

по два с половиной миллиона комбинаций ежесекундно, что грубо эквивалентно вычислительной мощности современных компьютеров, то проверка займет... Шестьдесят две тысячи лет! Некоторые типы данных — например, строковые, имеют еще большую вариативность, что делает невозможным проведение тестирования на полном тестовом покрытии. Это лишь один из принципов тестирования, по мнению международного квалификационного совета по тестированию программного обеспечения (International Software Testing Qualifications Board, ISTQB):

- Исчерпывающее тестирование невозможно.

  Этот принцип мы уже рассмотрели. Тестирование на всех возможных комбинациях входных данных выполняется очень долго даже для простейших функций.

- Тестирование демонстрирует наличие дефектов, а не их отсутствие.

  Как утверждает предыдущий принцип, исчерпывающее тестирование невозможно, следовательно, нельзя доказать, что дефектов нет. Но при проведении тестирования можно выявить дефекты и принять меры по их устранению.

- Заблуждение об отсутствии дефектов.

  Не существует системы без дефектов. Это утверждение стоит принять как аксиому. Тестирование помогает обнаружить и устранить наиболее критичные из дефектов.

- Раннее тестирование экономит ресурсы.

  Чем раньше будет выявлен дефект, тем проще и дешевле его исправить. Обнаружение и исправление ошибки на этапе проектирования займет немного времени, потому что система существует только на бумаге. Исправлять ту же ошибку после развертывания у пользователя гораздо сложнее.

- Принцип скопления дефектов.

  Дефекты проявляются в наиболее сложных компонентах системы. Принцип скопления дефектов предлагает внимательно исследовать компоненты, в которых уже нашлись дефекты, — скорее всего, там будут еще дефекты.

- Тестирование зависит от контекста.

  Разным системам — разное тестирование. Тестирование сайта-визитки может быть минимальным, а большинство выявленных дефектов могут расцениваться как субъективное мнение пользователя о дизайне и не приниматься во внимание. При разработке медицинского оборудования или авиационных систем тестирование должно быть максимально строгим, разносторонним и объективным. И пройдет много времени, прежде чем разработка дойдет до массового пользователя.

- Парадокс пестицида.

  Если использовать одно и то же средство для борьбы с вредителями, то вредители приобретают иммунитет, а средство становится менее эффективным. Аналогично и с тестированием. Если однажды создать тестовый набор и переиспользовать его, то со временем его эффективность будет снижаться. Необходимо до-

бавлять новые тесты и пересматривать старые, изучать новые методы тестирования и привлекать к тестированию новых сотрудников.

Тестирование неразрывно связано с проектированием на первых этапах разработки системы, когда создается документация, которая делится на несколько категорий. Во-первых, на функциональное и нефункциональное. Как следует из названия, функциональное тестирование проверяет соответствие функциональным требованиям, а нефункциональное тестирование — нефункциональным (виды требований подробно рассматривались в *главе 2*). Во-вторых, на статическое и динамическое:

- Статическое тестирование — это тестирование, которое не подразумевает выполнение программы. В эту категорию входит код-ревью, когда члены команды проверяют код друг друга в процессе разработки.
- Динамическое тестирование — это тестирование через запуск отдельных компонентов системы или целой системы.

Помимо статического и динамического тестирования, выделяют два вида проверок соответствия ожиданиям: верификацию и валидацию:

- Верификация — это процесс проверки того, что разрабатываемый продукт соответствует установленным в его спецификации требованиям. Верификация производится практически всегда. Пример: кузнец делает мечи для рыцарей. Всё, что имеет эфес и острый клинок, проходит верификацию. Даже если это деревянный меч для обучения.
- Валидация — это проверка того, что потребности пользователя удовлетворены. Если рыцарь приходит к кузнецу и просит у него меч для сражения с драконом, то валидацию пройдут только серебряные мечи, а деревянные и железные — нет.

Выбор данных для тестирования

Множество видов тестирования и множество комбинаций входных данных приводят к вопросу: «А как выбрать данные для тестирования?». Источников достаточно:

- Метод «черного ящика».

  В спецификациях и требованиях указывается, что должна получать система на вход и что подавать на выход. Запускаем систему на значениях, предусматриваемых спецификацией, и сверяем результат с эталонным.

- Метод «белого ящика».

  Если для тестирования доступен исходный код, то можно провести анализ путей исполнения программы и всех условий ветвления.

- Опыт — в некоторых случаях для тестирования можно применять личный опыт, полученный ранее.

Несколько способов получить данные для тестов только усложняют выбор, и возникает следующий вопрос: «Сколько тестов нужно сделать?» Тут надо найти компромисс между скоростью и качеством:

- чем больше тестов — тем больше тестовое покрытие и тем лучше качество;
- чем меньше тестов — тем быстрее разработка и быстрее выпуск системы.

Один из подходов для выбора тестового покрытия — это *эквивалентное разбиение* всех допустимых вариантов на области и выбор в тестовое покрытие всех граничных значений и нескольких значений из каждой области. Вернемся к функции, которая принимает на вход два значения и сумму. Входные параметры — это два числа, которые могут быть как положительными, так и отрицательными. Отсюда можно выделить четыре эквивалентные области:

- `a` — положительное, `b` — положительное;
- `a` — отрицательное, `b` — положительное;
- `a` — положительное, `b` — отрицательное;
- `a` — отрицательное, `b` — отрицательное.

Отдельно для каждого параметра есть три граничных случая, и каждый из них надо проверить как для отрицательного, так и положительного значения второго параметра:

- параметр принимает минимальное допустимое значение;
- параметр принимает значение 0;
- параметр принимает максимальное допустимое значение.

И граничные случаи для пары параметров:

- один параметр принимает максимальное значение, а второй — минимальное;
- оба параметра принимают максимальное значение;
- оба параметра принимают минимальное значение;
- оба параметра принимают значение 0.

Исходя из выполненного анализа эквивалентности, тестовое покрытие будет состоять из 15 вариантов входных данных (в случае, если из каждой области эквивалентности выбрана одна точка). При этом выбранное тестовое покрытие подходит и для тестирования бинарных алгебраических операций.

Вот еще краткое описание нескольких подходов для выбора тестового покрытия:

- Таблица решений.

  Таблица решений применяется в системах со сложной логикой и с множеством состояний и условий переходов и представляет собой упорядоченный список всех условий с возможными исходами (табл. 6.1). Таблица решений — это мощный инструмент тестирования, но ее составление занимает много времени.

- Таблица переходов.

  Для систем с множеством состояний можно построить таблицу переходов — конечный автомат со всеми состояниями и разрешенными переходами между ними. Тестирование по таблице переходов предполагает проверку того, что разрешенные переходы происходят, а запрещенные — нет.

Таблица 6.1. *Пример таблицы условий*

	1	2	...	N
Условия				
Условие 1	Нет	Нет	Да	Да
Условие 2	Нет	Да	Нет	Да
Условие 3	Да	Да	Да	Нет
Действия				
Действие 1	+	+	+	
Действие 2		+		+

- Сценарии использования (use-case).

  Для каждого сценария проверяем, что программа выполняет то, что указано в этом сценарии.

Юнит-тестирование

Юнит-тестирование (модульное тестирование) — это тестирование отдельных компонентов программного обеспечения: методов, классов или модулей. Модульное тестирование компонента производится изолированно от остальных компонентов системы. Так как тестируемый компонент может взаимодействовать с другими компонентами системы, то используются заглушки, которые соответствуют интерфейсу «заглушаемого» модуля, но имеют простейшую логику. Подробнее заглушки рассматриваются далее.

Обычно юнит-тесты пишет программист во время разработки соответствующей функциональности. Необходимость создавать заглушки к другим модулям побуждает программиста следовать принципам SOLID (см. *разд. «Принцип SOLID» главы 4*).

В некоторых случаях при написании юнит-тестов игнорируют изоляцию модулей друг от друга, и тогда выполняется другой вид тестирования — интеграционное.

Интеграционное тестирование

Интеграционное тестирование — это тестирование взаимодействия модулей или систем между собой. Такое тестирование возможно при разработке как минимум двух модулей системы, которые взаимодействуют друг с другом. При этом проверяется:

- на уровне классов — вызовы методов класса или библиотеки;
- в рамках системы — взаимодействие бизнес-логики (бэкенда) и интерфейса пользователя (фронтенда);
- между системами — сетевое взаимодействие двух систем.

Интеграционное тестирование выполняется инструментами, которые также выполняют модульное тестирование. Большинство таких инструментов имеют в названии фрагмент unit: CppUnit, PyUnit, JUnit. Тем не менее модульное тестирование изолированно тестирует один модуль, а интеграционное — взаимодействие двух и более модулей.

Сквозное тестирование (end-to-end)

Сквозное (end-to-end, end2end) тестирование — это тестирование, при котором программное обеспечение проверяется целиком: от и до, со всеми интеграциями, в том числе с внешними системами. В ходе сквозного тестирования производится эмуляция действий реального пользователя программного обеспечения, а тестирование чаще всего выполняется методом «черного ящика».

Модульное и интеграционное тестирование проверяют корректность реализации компонентов программы и их взаимодействия, а сквозное тестирование предлагает проверить систему в целом, в том числе и на соответствие нефункциональным требованиям. Тестировать всю систему целиком — это сложное действие, поэтому сквозное тестирование можно разделить на шаги и идти от простого к сложному:

- Возможности.

  Сначала проверяем соответствие функциональным требованиям в идеальных условиях: один пользователь, одно действие за раз. На вход подаются только корректные данные. Окружение соответствует требованиям.

- Стабильность.

  Повышаем реалистичность действий: пользователи всё еще вводят корректные данные, но пользователей уже несколько, и они выполняют действия одновременно, в том числе один пользователь пытается сделать несколько действий сразу, если такое возможное. Система запускается на долгий срок с целью определить утечки памяти или неэффективное использование других доступных вычислительных ресурсов.

- Устойчивость к сбоям.

  Помещаем программный продукт в худшие условия: пользователи вводят неправильные данные, в сетевой коммуникации происходят разрывы и потеря данных. Внешние системы отказывают в обслуживании.

Полное тестирование

Полное тестирование программного обеспечения можно разделить на несколько категорий. В каждой из этих категорий в общем выполняется одно и то же тестирование, но его выполняют разные люди с разной мотивацией, и получается разный результат:

- Системное тестирование.

  Проводится непосредственно разработчиками и тестировщиками компании на окружении, приближенном к продуктовому окружению. Каждый найденный дефект должен заноситься в журнал учета и устраняться в будущем. Этот вид тестирования может быть автоматизирован с формированием отчетов.

- Альфа-тестирование.

  В альфа-тестировании к сборке программного обеспечения, запущенного в тестовом окружении, подпускают ограниченное количество пользователей, чаще всего из числа сотрудников компании-заказчика. Иными словами, разработчики имеют контролируемое окружение, но действия выполняют люди, которые будут пользоваться программным обеспечением. Этот подвид тестирования позволяет применить действия пользователя к системе в контролируемой среде. Результат — отзывы пользователей на ранних этапах разработки.

- Бета-тестирование.

  Проводится пользователем в продуктовом окружении, но под контролем разработчика. Этот подвид тестирования позволяет проверить систему в действии на различных окружениях: операционных системах, браузерах, языковых настройках и т. п. Бета-тестирование не может быть автоматизировано. Результат — разработчики получают отзывы от настоящих пользователей, пользователи получают возможность проверить разрабатываемую функциональность и повлиять на нее.

- Приемочное тестирование.

  Этот вид тестирования производится пользователем или заказчиком самостоятельно. Результат этого вида тестирования — решение об использовании или неиспользовании программного обеспечения.

Ручное тестирование

Ручное тестирование — это любой из описанных видов тестирования, который проводится вручную, т. е. человеком. Чаще всего ручное тестирование применяется в системном тестировании. Хотя автоматизация действий исключает человеческий фактор и выполняется значительно быстрее, автоматическое тестирование не сможет вытеснить ручное тестирование именно из-за человечности в проведении тестов:

- Оценка пользовательского опыта.

  При ручном тестировании человек может проверять удобство использования программного обеспечения, что невозможно автоматизировать.

- Адаптация к изменениям.

  При многочисленных существенных изменениях в интерфейсе и логике программного обеспечения тестировщики могут быстро адаптироваться к новым условиям — в отличие от автоматических тестов, обновление которых может занять время, сопоставимое со временем разработки.

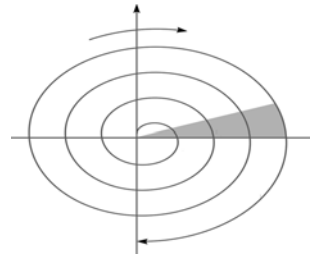
Заглушки

Для тестирования отдельных компонентов системы приходится изолировать тестируемые компоненты от связанных модулей или эмулировать поведение других систем, которые по каким-либо причинам нельзя использовать в тестировании. Задачу эмуляции других модулей решают *заглушки*, в некоторых источниках называемые *тестовыми дублями* (test doubles), — объекты, которые совпадают по интерфейсу взаимодействия с модулями программы, но содержат упрощенную логику.

Для обозначения заглушек используется несколько терминов:

- Пустышка (Dummy) — объекты, которые передаются, потому что этого требует интерфейс, но не используются тестируемой функциональностью модуля.
- Подделка (Fake) — упрощенная реализация существующих модулей. Например, использование словаря (ассоциативного массива) вместо драйвера базы данных «ключ-значение» — это fake-модуль. Такие модули реализуют интерфейс в полной мере, но представляют собой упрощенную реализацию, которая не может быть использована в продуктовом окружении.
- Заглушка (Stub) — частичная реализация интерфейса. Такие заглушки не реализуют настоящую логику модуля и применимы исключительно в тестах, для которых они были созданы.
- Шпионы (Spy) — это улучшенные заглушки, которые записывают информацию об обращениях. Например, «заглушка» почтового сервера будет возвращать ответ «письмо отправлено», а «шпион» запишет, сколько писем было отправлено за время выполнения теста.
- Макеты (Mocks) — это объекты, который не только возвращают какой-либо ответ, но и содержат настраиваемые *ожидания* к запросам от тестируемого модуля. При тестировании можно проверить соответствие этим ожиданиям.

ГЛАВА 7



Развертывание

Развертывание (deploy) — это один из этапов жизненного цикла программного обеспечения, в ходе которого программное обеспечение доставляется пользователям. Это может быть как первый выпуск (релиз, от *англ*. release), так и последующие обновления для исправления дефектов.

Во времена до широкого распространения Интернета развертывание программного обеспечения было медленным. Программы распространялись на портативных носителях информации. В случае игр для игровых приставок развертывание было еще и «одноразовым», т. к. носитель игры мог быть защищен от записи. Невозможность обновить выпущенное программное обеспечение — это серьезная мотивация заниматься тестированием, но также и источник неудобств для разработчика.

К счастью или к сожалению, но эти времена прошли. Сейчас множество компьютеров имеют доступ к сети Интернет, что позволяет быстро рассылать пользователям обновления ПО. Помимо «классического» программного обеспечения, которое устанавливается на компьютер пользователя, также распространены «облачные» сервисы. Основное отличие облачных сервисов — полезная работа приложения выполняется на удаленных серверах. Тем не менее приложения, будь то классические или облачные, нужно обновлять. Всегда можно делать обновление вручную, но в долгосрочной перспективе — это провальное решение.

Автоматизация развертывания

Автоматизация может быть различной. В простейшем случае это неинтерактивный скрипт, который собирает исходный код в исполняемый файл, запаковывает этот файл в архив, а архив загружает на удаленный сервер, откуда архив скачивают пользователи. В современном мире для решения задач развертывания используется комбинация из двух практик, называемая CI/CD: CI — это Continuous Integration, непрерывная интеграция, а CD — Continuous Delivery, или Continuous Deploy, непрерывная доставка или непрерывное развертывание.

Непрерывная интеграция

Непрерывная интеграция (Continuous Integration) — это практика, при которой члены команды постоянно *интегрируют* свою работу в основную (общую) ветку системы контроля версий и проверяют, что изменения не «ломают» разрабатываемое программное обеспечение. Практика CI приучает разработчиков реализовывать функциональность малыми частями, постоянно проверяя стабильность продукта после обновления.

Очевидно, что при работе нескольких человек над одной программой выполнять проверку стабильности вручную неэффективно. В практике CI предлагается выполнять автоматические ежедневные сборки (daily build) программного обеспечения. При каждой сборке реализуется следующая последовательность действий:

1. Получение исходного кода из системы контроля версий.
2. Компиляция исходного кода.
3. Выполнение автотестирования.
4. Формирование и отправка отчета.

Для больших программных продуктов компиляция и выполнение всех тестов могут занимать существенное время, поэтому применяются «ночные сборки» (nightly builds): в конце рабочего дня все сотрудники вносили свои изменения, уходили отдыхать, а в начале следующего рабочего дня могли изучить отчеты о тестировании. Итеративность разработки и ежедневные сборки — это первопричина достоинств практики непрерывной интеграции:

- каждый день генерируется сборка программного обеспечения наиболее актуальной версии с самой новой функциональностью. Это программное обеспечение может использоваться для тестирования и демонстрации;
- между внесением изменений и выполнением тестов проходит не более одного дня, что позволяет обнаруживать дефекты программного обеспечения раньше. Чем раньше будет обнаружен дефект, тем дешевле его исправить.

Главный недостаток практики CI — это затраты на вычислительные ресурсы и необходимость поддерживать инфраструктуру, которая выполняет действия CI.

Непрерывная доставка

Вторая практика из CI/CD — это *непрерывная доставка* (Continuous Delivery) программного обеспечения пользователям. Как было отмечено ранее, один из артефактов непрерывной интеграции — это сборка программного обеспечения, которую можно использовать для тестирования или демонстрации заказчику. Задачу развертывания таких сборок решает практика непрерывной доставки.

Непрерывная доставка позволяет автоматизировать развертывание актуальной версии программного обеспечения на тестовых окружениях для ручного тестирования, запускать автоматическое приемочное тестирование и доставлять обновление пользователям. Преимущества непрерывной доставки легко заметны:

- автоматизация рутинных действий снижает риски ошибки из-за человеческого фактора и экономит время разработчиков и администраторов;
- уменьшает количество дефектов и повышает качество продукта из-за дополнительного тестирования;
- уменьшает время выхода на рынок (time to market) новой функциональности программного обеспечения.

Тем не менее непрерывная доставка, несмотря на положительные стороны автоматизации, не является решением всех проблем.

В идеальном варианте тестовые окружения должны быть максимально приближены к действующему продуктовому окружению. Но это требование с подвохом:

- продуктовое окружение может быть очень большим. Создавать одно или несколько полноразмерных тестовых окружений может быть экономически нецелесообразно;
- тестовое окружение должно содержать данные из продуктового окружения. Это создает риск информационной безопасности и открывает лишний доступ к чувствительным данным, т. к. разработчики обычно имеют более высокий уровень доступа на тестовых стендах;
- отказ от использования данных из продуктового окружения создает риск пропустить дефект, который не проявляется на искусственных данных в тестовом окружении;
- пользователи могут быть недовольны обновлениями, особенно если от них невозможно отказаться;
- в некоторых отраслях непрерывная доставка не может быть реализована в полной мере, т. к. требует более сложного тестирования, которое невозможно автоматизировать. К таким отраслям относится, например, авиация.

Непрерывное развертывание

Непрерывное развертывание (Continuous Deployment) — это «финальная» форма непрерывной доставки, в которой решение о развертывании программного обеспечения принимает не человек, а компьютер — в зависимости от пройденных этапов тестирования. Проще говоря, если все автоматические тесты прошли успешно, то новая версия немедленно разворачивается в продуктовое окружение. Это еще больше ускоряет процесс доставки обновлений до конечного пользователя.

* * *

Каждая из описанных практик имеет ограниченную область ответственности и набор задач, которые решаются соответствующей практикой. Однако с точки зрения реализации интеграция и развертывание — это набор команд, которые выполняются инструментами автоматизации. Вот популярные инструменты автоматизации CI/CD-практик:

- Travis CI — облачный сервис, предоставляющий автоматизацию CI/CD.
- Github Actions — является частью облачного Github. Github совмещает в себе систему контроля версий, позволяет отслеживать дефекты ПО и автоматизацию CI/CD.
- Gitlab CI/CD — является частью Gitlab, аналога Github, но с возможностью локального развертывания.
- Jenkins — локальный инструмент для автоматизации CI/CD.

Автоматизация практик CI/CD в этих сервисах реализуется *пайплайнами* (от *англ.* pipeline — конвейер). Каждый пайплайн состоит из шагов, которые называют *джобами* (от *англ.* job — работа), и условий запуска каждого шага (рис. 7.1).

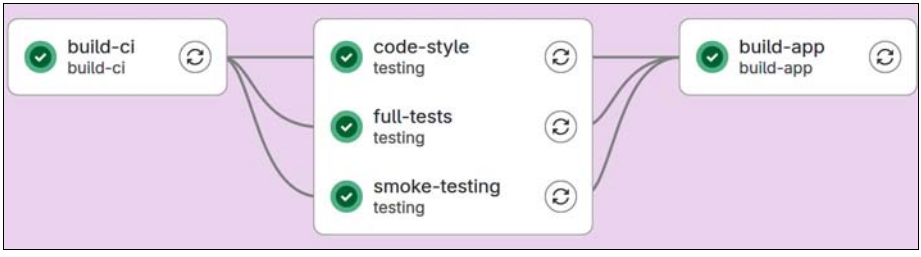


Рис. 7.1. Пример визуалиазции пайплайнов в Gitlab

Например, если в текущем состоянии проект не может быть собран из-за синтаксической ошибки в исходном коде, то сборки не получится. Не получится сборки — нет нужды проводить автоматическое тестирование. Как отмечалось ранее, с точки зрения инструмента автоматизации между практиками CI и CD нет разницы. Представим наиболее очевидный пайплайн и разберем его по шагам:

1. Компиляция программного обеспечения из исходного кода. Запуск программы для сборки — практика CI.
2. Выполнение автоматических тестов. Запуск программы для тестирования — практика CI.
3. Создание сборки. Запуск программы или скрипта, которая упакует программу вместе с документацией в архив или установщик — практика CI.
4. Развертывание на тестовом окружении. Загрузка сборки в окружение и выполнение команд для развертывания — практика CD.
5. Проведение автоматических тестов на тестовом окружении. Запуск программ, которые эмулируют поведение пользователя на тестовом окружении, — практика CD.
6. Развертывание сборки на продуктовом окружении. Аналогично п. 4 — практика CD.

Этот пример еще раз подтверждает, что с точки зрения инструмента автоматизации нет различия между CI и CD — каждый шаг, вне зависимости от практики, является набором команд, запускающих программное обеспечение. Это размывает грани-

цы между CI и CD и приводит к использованию неверной терминологии. В разговорной речи путаница между названиями практик не является критичной, но корректность — основа продуктивной коммуникации.

Масштабирование

Однажды при развертывании очередной версии приложения вам не хватит вычислительных ресурсов. Придется расширяться. Но как это сделать?

> **Масштабируемость** — это способность системы справляться с увеличением нагрузки путем добавления ресурсов. Масштабирование бывает двух типов: вертикальное и горизонтальное.

Вертикальное масштабирование

Вертикальное масштабирование — это увеличение производительности каждого элемента вычислительной системы. Например, замена процессора сервера на более производительный (рис. 7.2). Это наиболее простой способ масштабирования, который не требует модификации программного обеспечения.

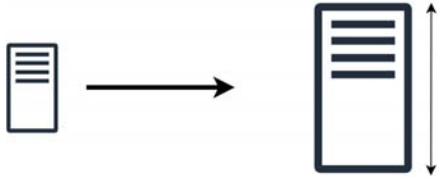


Рис. 7.2. Вертикальное масштабирование

Представим программу, которая реализует в тридевятом королевстве счетчик радости — аналог «лайков» без привязки к жителю: счетчик и HTTP-интерфейс, который позволяет узнать текущее значение счетчика и инкрементировать его. Затем делаем «тумбочку» с кнопкой и экраном — на экране отображается текущее значение счетчика, а кнопка делает вызов к программе и увеличивает значение на единицу. Разместим это изобретение в поселке на 100 грустных человек, где люди будут нажимать на кнопку один раз в час. С такой нагрузкой справится даже самый маленький и старый сервер.

Вдруг король дает распоряжение: поставить изобретение в каждом населенном пункте королевства. Вы выполняете требование и осознаете, что сейчас у вас появятся десятки тысяч кнопок, расположенных в местах, где гораздо больше радостных людей. И вместо одного запроса в час у вас получится пять запросов в секунду, а сервер способен обрабатывать в секунду только один запрос. Остальные четыре будут отбрасываться. Для людей это не проблема — после нажатия на кнопку они увидят увеличение числа, но с точки зрения статистики для короля такая ситуация критична — ведь будут подсчитываться только 20% от реального числа нажатий. В этом случае поможет вертикальное масштабирование. Вы просите у короля более

мощный и дорогой сервер, который, по вашим расчетам, способен выдерживать до ста запросов в секунду. Проблема решена.

Несмотря на простоту, вертикальное масштабирование имеет существенный недостаток — развитие не может быть бесконечным. Представим, что в описанном примере сервер, который справляется с сотней запросов в секунду, — это самый новый и самый дорогой сервер в королевстве. В случае, если к счетчику будет выполняться больше запросов, то часть запросов снова будут отбрасываться, а увеличить вычислительную мощность сервера уже не получится. Здесь поможет только горизонтальное масштабирование.

Горизонтальное масштабирование

Горизонтальное масштабирование — это разнесение структурных компонентов системы на разные физические серверы с возможностью увеличивать количество серверов (рис. 7.3).

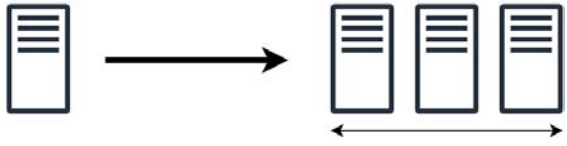


Рис. 7.3. Горизонтальное масштабирование

Но не каждая система может быть масштабируема горизонтально, и в некоторых случаях для этого потребуются существенные доработки. Так, описанный в предыдущем разделе «счетчик лайков» в тридевятом королевстве нельзя масштабировать горизонтально в исходном виде — если запустить счетчик на двух разных серверах, то это будут два самостоятельных счетчика.

Для возможности горизонтального масштабирования нужно внести в систему доработки. Наиболее простой вариант: отказаться от глобального счетчика на все королевство и сделать локальные счетчики для каждого населенного пункта — чтобы каждый локальный счетчик один раз в час отправлял статистику королю. При этом получится, что нагрузка на локальный счетчик — это один запрос в час, с которым мы умеем работать, а рост количества охваченных счетчиками населенных пунктов решается увеличением количества локальных счетчиков. Это и есть горизонтальное масштабирование.

Впрочем, у масштабирования на практике есть дополнительные сложности, которые зависят от используемой инфраструктуры.

Выделенные и облачные серверы

Провайдеры инфраструктуры предоставляют разные типы серверов: выделенные и облачные. Разница между ними существенна.

Выделенный сервер

Выделенный сервер — это полный доступ к физическому серверу, который стоит в дата-центре. Доступ к физическому серверу дает полный доступ ко всем его компонентам — это похоже на удаленный персональный компьютер. Главное достоинство выделенного сервера — отсутствие «соседей», которые претендуют на его вычислительные ресурсы. Арендатор волен единолично делать с сервером все, что ему угодно. Однако вместе с большой свободой приходит большая ответственность. Физическое оборудование может выходить из строя. Это значит, что нужно следить за износом накопителей, чтобы не потерять данные. Также может выйти из строя любой компонент сервера, из-за чего сервер выключится, и запущенные на нем системы будут недоступны до устранения физической неисправности. Такие проблемы случаются не часто, но все-таки случаются, поэтому эти риски необходимо предусматривать.

Облачный сервер

Облачный сервер — это виртуальная машина, запущенная в кластере физических серверов, который носит название «облако». Виртуальная машина, как следует из названия, существует только в виде представления внутри кластера. Кластер и виртуальное представление абстрагируют пользователя от множества физических задач и проблем:

- кластер распределяет данные виртуальных накопителей в несколько копий по физическим накопителям. Выход из строя одного из физических накопителей кластера незаметен пользователям виртуальной машины;
- выход из строя вычислительного узла кластера может привести к выключению виртуальной машины и ее перезапуску на другом вычислительном узле кластера. Перезагрузка — это неприятно, однако придется ждать проведения диагностики и замены физического сервера;
- отсутствие физического воплощения у виртуальных машин позволяет быстро изменять производительность виртуальной машины при перезагрузке. Если для решения задач нужно добавить процессорных ядер и оперативной памяти, то в облачном сервере для этого понадобится только перезагрузиться. В выделенном сервере это более трудоемкая задача;
- отсутствие физического воплощения у накопителей позволяет быстро создавать копии виртуальных машин для горизонтального масштабирования;
- при загрузке виртуальная машина не занимается инициализацией и тестированием оборудования, а практически сразу загружает операционную систему;
- оплата за облачные серверы может быть почасовой, поминутной или даже посекундной. Вместе с быстрым запуском это позволяет экономически эффективно производить горизонтальное масштабирование при увеличении нагрузки, а также уменьшать количество облачных серверов при снижении нагрузки.

Удобства облачного сервера выглядят очень заманчиво, но и «облака» не лишены отрицательных моментов:

- у облачных серверов есть характеристика «виртуальные ядра» — vCPU. В идеальном мире предполагается, что виртуальное ядро — это 100% времени одного физического ядра сервера. На самом деле, это может быть 50 или даже 10% от времени физического ядра. Виртуальные машины-«соседи» на вычислительном узле станут бороться за одно ядро, и кто-то будет проигрывать;
- накопители виртуальной машины располагаются на нескольких узлах кластера, и не всегда данные хранятся на вычислительных узлах. Это значит, что доступ к виртуальным накопителям будет медленнее, чем у выделенного сервера к своим накопителям. Однако эта проблема частично решается особыми вычислительными узлами с локальными накопителями.

Облачные серверы — это более легкий и зачастую более выгодный вариант, совместимый с горизонтальным масштабированием. Выделенные серверы требуют больше внимания, но предоставляют большую производительность. Компромиссом между облачным и выделенным сервером является «частное облако» — собственный кластер, конфигурируемый под нужды арендатора.

Гибкость и масштабируемость облачных серверов положены в основу при создании SaaS (Software as a Service) — предоставления не виртуальной машины, а программного обеспечения на основе облачных серверов с возможностью масштабирования. В контексте проектирования информационных систем следует различать возможности моделей DBaaS и объектного хранилища.

DBaaS (Database as a Service)

DBaaS (Database as a Service) — это модель предоставления базы данных как облачной услуги, при которой пользователи получают доступ к полностью управляемым и масштабируемым базам данных.

DBaaS снимает с разработчиков и администраторов задачи, связанные с установкой, настройкой, администрированием, обновлением и резервным копированием баз данных. При использовании DBaaS разработчики могут сосредоточиться на работе с данными и разработке приложений, не тратя время на инфраструктурные задачи. Облачный провайдер берет на себя всю операционную часть: развертывание базы данных, управление производительностью, мониторинг, обеспечение безопасности и автоматическое восстановление в случае сбоя. Пользователи получают доступ к базе через удобный интерфейс, будь то веб-консоль, API или командная строка.

Одним из ключевых преимуществ DBaaS является простота масштабирования. Если приложению требуется больше ресурсов для обработки данных, провайдер DBaaS позволяет масштабировать базу данных вертикально или горизонтально практически без вмешательства пользователя.

Объектное хранилище

Объектное хранилище — это модель хранения данных, ориентированная на управление большими объемами неструктурированной информации, такой как медиафайлы, резервные копии и другие типы больших данных.

Объектное хранилище отличается от традиционных файловых систем и реляционных баз данных тем, что вместо иерархической структуры каталогов или строго заданных таблиц с полями и строками данные организуются в виде объектов. Каждый объект представляет собой единицу данных, включающую сам контент, уникальный идентификатор и метаданные.

Метаданные в объектных хранилищах описывают свойства данных, такие как их тип, дата создания, версия, а также позволяют задавать пользовательские атрибуты с произвольными данными. Подобная гибкость делает объектное хранилище весьма удобным для работы с широким спектром приложений, где требуется не только хранение, но и управление информацией.

Объектное хранилище — это распределенная архитектура: данные хранятся на нескольких узлах или серверах, что обеспечивает высокую доступность, отказоустойчивость и масштабируемость. При этом пользователю не нужно заботиться о том, где физически находятся объекты, т. к. доступ к ним осуществляется через уникальные идентификаторы, часто с использованием стандартных API, таких как REST.

Эта технология хорошо масштабируется, поскольку новая емкость или вычислительные ресурсы могут быть добавлены в систему без сложных изменений в конфигурации.

С точки зрения использования объектное хранилище идеально подходит для приложений, где данные не часто изменяются, но должны быть легкодоступны для чтения. Например, это потоковое видео, резервное копирование данных, публикация контента в Интернете или аналитика больших данных. Кроме того, оно используется в системах с требованиями к долговременному хранению, таких как архивы или юридическая документация.

Развертывание: контейнеризация

Контейнеризация предоставляет гибкое и изолированное окружение для работы программного обеспечения, позволяя избежать конфликтов конфигураций и версий зависимостей. Развертывание нескольких приложений на одном сервере может приводить к сложностям из-за использования приложениями разных версий одного программного обеспечения. Контейнеризация решает эту проблему, изолируя каждое приложение вместе с его собственными зависимостями и настройками.

Для изоляции разных приложений можно использовать виртуальные машины, но это решение может быть неэффективным с точки зрения потребления ресурсов, т. к. для каждого приложения необходимо запустить самостоятельную операционную систему. Контейнеры — это более легковесное решение по сравнению с виртуальными машинами, т. к. не требуют полноценной операционной системы для каждого экземпляра, что делает их более производительными и экономичными с точки зрения использования ресурсов.

С точки зрения разработки и поддержки контейнеризация обеспечивает консистентность окружения на всех этапах — от разработки до развертывания. При

«обычном» развертывании могут возникнуть различия в настройках и зависимостях между окружением, где производилась разработка и тестирование, и продуктовой средой, что увеличивает вероятность ошибок, которые невозможно воспроизвести в тестовом окружении.

Благодаря контейнерам обновление и поддержка приложений становятся проще и безопаснее, т. к. изменения можно вносить только в отдельные контейнеры, не затрагивая остальную систему. Это позволяет внедрять изменения быстрее и уменьшает риск нарушения работы других частей системы.

Технология chroot

chroot — это механизм UNIX-подобных операционных систем, позволяющий изменять корневой каталог для выбранного процесса и его дочерних процессов. Механизм chroot — прародитель современной контейнеризации, появившийся в конце 1970-х годов, задолго до термина «контейнеризация». Применение chroot изолирует выполняемое приложение от основной файловой системы, ограничив его доступ только к заданному каталогу, который станет для него новым корневым каталогом.

С технической точки зрения, когда применяется chroot, программа «видит» указанный каталог как корневой, а всё, что находится выше по иерархии файловой системы, становится недоступным для этого процесса. Ранее механизм chroot применялся в FTP-серверах для изоляции пользователя в его домашнем каталоге.

Механизм chroot не предоставляет контейнеризации в современном понимании этого термина, т. к. он способен только ограничить область видимости в рамках файловой системы, но не способен ограничивать в потреблении вычислительных ресурсов и не изолирует программу и все ее зависимости.

На текущий момент chroot используется преимущественно в администрировании серверов для перехода из «образа восстановления» в операционную систему, установленную на сервере.

Технология Jail

Jail — это механизм виртуализации на уровне операционной системы FreeBSD, который позволяет создавать изолированные окружения. Jail изначально разрабатывался для повышения безопасности серверов и изоляции процессов.

В основе механизма Jail лежит chroot, но «джейлы» работают на базе ядра основной системы и представляют собой «тюрьмы» для процессов, которые являются полностью рабочими, но виртуальными версиями основной операционной системы. Это дает возможность ограничивать не только файловую систему, но и доступ к вычислительным ресурсам сервера и к сведениям из хостовой операционной системы, что позволяет запускать различные программы в изолированных контейнерах на одном физическом сервере без риска, что одно приложение сможет повлиять на другое.

FreeBSD Jail поддерживает возможность разделять права доступа и управлять привилегиями для каждого контейнера. Это делает его идеальным для среды, где тре-

буется высокий уровень безопасности, — например, для веб-хостинга. Каждый Jail можно настроить как полноценное окружение для приложений, а благодаря низким накладным расходам такая виртуализация более производительна по сравнению с виртуальными машинами.

Механизм FreeBSD Jail вдохновил аналогичные проекты для семейства операционных систем Linux: OpenVZ и LXC:

- OpenVZ, вышедшая в 2000-х годах, стала первым серьезным шагом в этом направлении на платформе Linux. OpenVZ предоставляла возможность каждому контейнеру иметь свои файлы, процессы, сеть и даже root-доступ, что приближало ее к оригинальной идее «джейлов» FreeBSD, но с учетом особенностей Linux.
- Чуть позже, в 2008 году, в ответ на растущий спрос на контейнеризацию на базе ядра Linux был создан проект LXC (Linux Containers). Разработка LXC началась как попытка интегрировать контейнеризацию в ядро Linux с использованием его стандартных функций. Если OpenVZ требовала модификации ядра, то LXC стремилась стать частью стандартного ядра Linux и развивалась как более открытая и гибкая платформа для контейнеров.

FreeBSD Jail, OpenVZ и LXC решали задачу контейнеризации, но это сложные инструменты, которые требуют квалификации, а настройка содержимого контейнеров похожа на настройку операционных систем в виртуальных машинах.

Технология Docker

Docker — это платформа для контейнеризации, которая позволяет разработчикам создавать и запускать приложения в изолированных средах. Контейнеры Docker включают всё необходимое для работы программы: исполняемый файл, зависимости и системные инструменты, что делает их переносимыми и независимыми от окружения операционной системы. Основной идеей Docker является стандартизация процесса развертывания: приложение, собранное в Docker-контейнере, будет запускаться одинаково на компьютере автора, на компьютере коллег автора и на сервере, устраняя проблемы совместимости и зависимостей.

Ключевой элемент Docker — *образ* (image), представляющий собой шаблон инструкций для создания контейнеров. Образы собираются по принципу слоев, где каждая команда создает новый слой, добавляя что-то новое. Все слои кешируются, что позволяет переиспользовать общие слои для разных контейнеров. Это делает создание контейнеров быстрым и эффективным процессом.

Например, если вам необходимо контейнеризировать приложение на языке программирования Python, то последовательность команд для создания контейнера будет следующая:

1. Установить интерпретатор Python.
2. Установить все python-зависимости для приложения.
3. Скопировать файлы приложения в контейнер.

При обновлении исходного приложения контейнер необходимо собрать по шаблону заново. Однако кеширование слоев обнаружит, что слой, отвечающий за установку Python, уже существует, равно как и слой с установкой зависимостей приложения, и обновит только последний слой, что позволит сэкономить время и место на накопителе.

Второй важной особенностью технологии Docker является возможность обмениваться образами через репозитории — Docker Registry. На текущий момент существует центральный репозиторий для образов — Docker Hub. Возможность обмениваться готовыми образами существенно упрощает настройку программного обеспечения, а слоистость образов позволяет удобно модифицировать существующие образы.

Шаблоны, последовательность инструкций для создания образов, называются *докер-файлами* (Dockerfile) и имеют такой вид:

```
FROM python:3.9-slim
RUN pip install --no-cache-dir requests
COPY . /app
CMD ["python", "/app/app.py"]
```

Этот шаблон семантически похож на пример контейнеризирования приложения, описанный ранее, но с небольшими различиями:

- вместо установки интерпретатора Python команда `FROM` выполняет поиск образа по имени `python` версии `3.9-slim` в доступных репозиториях образов. Это базовый образ, в котором уже решена задача установки интерпретатора;
- команда `RUN` выполняет указанную команду внутри контейнера. В нашем случае пакетный менеджер `pip` устанавливает пакет под названием `requests`;
- команда `COPY` копирует файлы из текущей операционной системы в контейнер;
- команда `CMD` определяет команду, которая будет выполнена внутри контейнера при его запуске.

Шаблон можно собрать в образ в одну команду:

```
docker build . -t my-new-image:1.0.0
```

После выполнения этой команды будет создан образ с именем `my-new-image` версии 1.0.0. Запустить его можно также в одну команду:

```
docker run my-new-image
```

Команда запуска полностью абстрагирует пользователя от установки и первоначальной настройки программного обеспечения — Docker скачивает шаблон из репозитория и по командам слой за слоем создает образ из пустого контейнера.

Концептуально Docker похож на Jail, т. к. использует ядро хостовой операционной системы и создает контейнеры, которые изолированы друг от друга и от хостовой операционной системы. Но Docker, в отличие от Jail, OpenVZ и LXC, не ограничен семейством операционных систем Linux, а запускается также на Windows, macOS и даже на FreeBSD. Однако для всех остальных операционных систем, кроме Linux,

Docker устанавливает виртуальную машину, в которой запускается Linux, и уже внутри этой виртуальной машины запускаются контейнеры. Несмотря на кажущуюся сложность, Docker полностью абстрагирует пользователя от этого процесса, поэтому контейнеры Docker создаются на других операционных системах так же просто, как и на Linux. Именно эта простота настройки контейнеров стала причиной популярности Docker как средства контейнеризации приложений.

Образы Docker удобны для изоляции отдельных приложений, но синтаксис шаблонов неявно предписывает использовать один контейнер для запускаемой одной программы. А что делать, если разработанное программное обеспечение взаимодействует с другими программами, которые можно запустить в соседних контейнерах?

Docker Compose — это инструмент, который позволяет объединять несколько контейнеров с помощью описания в формате YAML. Файл docker-compose.yml описывает все необходимые параметры для каждого контейнера: имя образа, переменные окружения, каталоги для хранения данных и многое другое. После этого достаточно выполнить одну команду, чтобы развернуть все указанные контейнеры одновременно, при этом все контейнеры в рамках одного Compose могут обращаться друг к другу через TCP/IP, используя имя контейнера как доменное имя. Это значительно упрощает процесс разработки и тестирования, т. к. позволяет быстро запускать и останавливать целые среды с помощью одной команды.

Технология Kubernetes

Kubernetes — это оркестратор, система для автоматизации развертывания, управления и масштабирования контейнеризованных приложений. Разработанный Google и переданный в Cloud Native Computing Foundation, Kubernetes стал популярным инструментом для оркестрации контейнеров в современных IT-инфраструктурах. Система позволяет упростить управление контейнерами и помогает обеспечивать высокую доступность, устойчивость к сбоям и легкость в управлении.

Основной принцип работы Kubernetes — это абстракция над физической или виртуальной инфраструктурой. Вместо того чтобы заботиться о конкретных серверах, на которых запускаются приложения, разработчики взаимодействуют с контейнерами, которые Kubernetes автоматически размещает на подходящих узлах (серверах) в кластере. Кластер Kubernetes состоит из нескольких компонентов: управляющей плоскости, которая отвечает за контроль всей системы, и рабочих узлов, где запускаются контейнеры.

Минимальная единица развертывания в Kubernetes называется «под» — это один или несколько контейнеров, объединенных общей сетью и системой хранения. Поды могут объединять контейнеры, которые тесно связаны и должны работать на одном физическом сервере. Kubernetes следит за состоянием подов и в случае сбоев автоматически восстанавливает их, пересоздавая на доступных узлах.

Kubernetes использует систему декларативного управления. Это означает, что администратор или разработчик задает желаемое состояние кластера, — например,

количество экземпляров приложения, которое должно быть запущено, а Kubernetes поддерживает это состояние независимо от внешних факторов или сбоев. Такой подход делает Kubernetes удобным инструментом для работы с микросервисной архитектурой, поскольку каждый сервис можно развернуть, обновить и масштабировать независимо от других.

Когда приложение развернуто в несколько экземпляров, Kubernetes автоматически распределяет трафик между подами, используя встроенный балансировщик нагрузки. Это гарантирует, что ни один контейнер не будет перегружен, и все пользователи смогут получать быстрый доступ к приложению. Более того, Kubernetes поддерживает автоматическое масштабирование — изменение количества экземпляров приложения в зависимости от нагрузки. Например, если приложение становится более востребованным, Kubernetes может автоматически создать дополнительные его реплики, чтобы обрабатывать возрастающий объем трафика.

Кроме того, Kubernetes активно поддерживает концепцию хранения данных, при которой контейнеры могут использовать постоянное хранилище. Для этого задействуются тома, которые могут быть привязаны к подам, обеспечивая сохранение данных вне зависимости от состояния контейнера. Если под завершает работу или пересоздается, данные не теряются, поскольку привязаны к внешнему хранилищу. Это делает Kubernetes полезным для работы с базами данных и другими приложениями, где данные должны быть доступны даже после перезапуска контейнеров.

Kubernetes обладает системой управления конфигурациями и секретами, что позволяет безопасно хранить данные конфигураций, такие как пароли или API-ключи, отдельно от контейнеров. Это обеспечивает более высокий уровень безопасности, т. к. конфиденциальные данные могут быть зашифрованы и переданы контейнерам по мере необходимости, что особенно актуально для приложений с высокой степенью защищенности.

Kubernetes работает с гибкими стратегиями обновления и отката приложений, что делает возможным их плавное обновление без остановки работы. Например, обновления могут производиться постепенно, по одному поду за раз, что позволяет тестировать новые версии приложений и откатывать изменения в случае проблем. Это минимизирует возможные простои и снижает риски при обновлениях, особенно в критически важных системах.

Технология Nomad

Nomad — это разработанная HashiCorp система оркестрации контейнеров и приложений, которая управляет развертыванием, масштабированием и обслуживанием приложений в различных средах. В отличие от Kubernetes, Nomad ориентирована на простоту, легковесность и универсальность, предоставляя возможность работать не только с контейнерами, но и с другими типами приложений и процессов, включая исполняемые файлы и виртуальные машины. Эта универсальность позволяет использовать Nomad от развертывания микросервисов до управления более традиционными рабочими нагрузками. Nomad, как и Kubernetes, использует декларативный подход.

Кластер Nomad состоит из серверов и агентов (нод). Серверы отвечают за принятие решений по планированию и распределению задач, учитывая требования и ресурсы, поддерживают баланс нагрузки, резервирование и отказоустойчивость, а агенты запускают сами задачи на хостах. Такая архитектура обеспечивает горизонтальную масштабируемость и высокую доступность, что делает Nomad способным поддерживать тысячи узлов.

Nomad придерживается архитектурной простоты, поэтому Nomad — это единый исполняемый файл, который легко развернуть. При этом Nomad обеспечивает только оркестрацию контейнеров. Это отличает Nomad от более сложных платформ, таких как Kubernetes, которые требуют дополнительных компонентов для работы, и развертывание которых занимает существенное время.

Nomad может быть интегрирована с другими инструментами HashiCorp, такими как Consul и Vault, что позволяет расширить ее функциональность: добавить систему обнаружения сервисов или хранилище секретов. Однако это не является обязательными компонентами для работы Nomad.

Развертывание: обнаружение сервисов

Обнаружение сервисов (Service Discovery) — это процесс, который позволяет сервисам находить друг друга и взаимодействовать в распределенных системах. В современных архитектурах при горизонтальном масштабировании сервисы могут динамически создаваться и удаляться, что затрудняет взаимодействие с ними. Обнаружение сервисов помогает решить эту проблему, обеспечивая механизм, который позволяет сервисам автоматически находить друг друга. Существуют два основных подхода к обнаружению сервисов: статическое и динамическое:

- Статическое обнаружение предполагает, что адреса сервисов задаются вручную в конфигурационных файлах, что может быть эффективно для небольших систем, но становится неудобным и подвержено ошибкам при увеличении масштаба.
- При динамическом подходе сервисы регистрируются в специальном реестре при запуске и удаляются из реестра при завершении работы (рис. 7.4). Другие сервисы могут запросить информацию о доступных экземплярах желаемых сервисов, что позволяет им взаимодействовать без необходимости ручной конфигурации. Это особенно полезно при использовании облачных серверов, т. к. масштабирование и управление ресурсами происходят автоматически.

Статическое и динамическое обнаружение может быть совмещено с балансировкой нагрузки, что при наличии нескольких экземпляров одного сервиса позволяет равномерно распределять нагрузку между ними (способы балансировки нагрузки рассматриваются в *главе 8*).

В некоторых случаях вместо реестра сервисов используется рассылка широковещательных сообщений. Сервисы при запуске могут рассылать в локальной сети широковещательные пакеты, выполняющие функцию обнаружения совместимых сервисов. Так, например, система для передачи информации между устройствами KDE

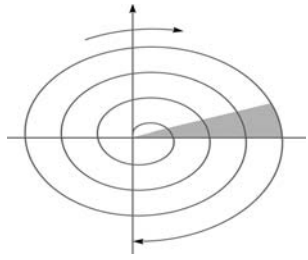
Connect периодически после запуска посылает широковещательный UDP-пакет с информацией о себе, позволяя другим устройствам с KDE Connect обнаружить «новичка» в сети и установить с ним связь. Такой подход применим в случае с одной программой, которая общается с другими экземплярами такой же программы. В облачных системах проще иметь централизованный реестр, который знает все необходимое о каждом сервисе.

Для организации динамического обнаружения сервисов используют программы Apache ZooKeeper или Consul.



Рис. 7.4. Динамическое обнаружение

ГЛАВА 8



Балансировка нагрузки

Балансировка нагрузки — это техника распределения запросов или вычислительных задач между несколькими серверами с целью оптимизировать производительность, снизить задержки и обеспечить устойчивость информационной системы. В высоконагруженных системах, где объем данных или количество запросов превышают возможности одного сервера, нагрузка распределяется на множество серверов, каждый из которых обрабатывает только часть запросов. Это позволяет значительно улучшить время отклика и предотвращает ситуации, когда один сервер перегружен до отказа, а остальные простаивают.

Один из подходов для распределения запросов между серверами — использование *балансировщика нагрузки*, который действует как посредник между пользователями и серверами. Он принимает входящие запросы, определяет, какой сервер наиболее подходит для обработки, и перенаправляет запрос туда. Некоторые балансировщики нагрузки функционируют на уровне сетевых протоколов, а другие — на более высоком уровне, анализируя характеристики запросов, чтобы направить пользователя на сервер, который может выполнить задачу с минимальной задержкой.

Преимущество балансировки в том, что она позволяет оперативно реагировать на сбои и перезагрузку серверов. Если один из серверов выходит из строя, балансировщик автоматически исключает его из балансировки, перенаправляя запросы на оставшиеся серверы. Это делает систему более надежной и помогает обеспечить доступность для пользователей даже при внезапных отказах оборудования.

Алгоритмы балансировки

Алгоритмы балансировки нагрузки определяют, как распределять входящие запросы между серверами или другими ресурсами в системе, чтобы избежать перегрузок. Эти алгоритмы позволяют учитывать множество параметров, таких как текущая нагрузка на серверы, время отклика, количество соединений и другие метрики.

- Round Robin. Циклический алгоритм.

  Самый простой из алгоритмов. Он распределяет запросы по кругу, последовательно направляя каждый новый запрос на следующий сервер в списке, и когда

доходит до последнего сервера, процесс повторяется с первого. Реализация циклического алгоритма тривиальна, но он подходит только для систем, где все серверы имеют одинаковую производительность, да и запросы примерно одинаковые. В случаях, когда серверы имеют разную мощность или запросы имеют разную сложность, Round Robin может привести к неравномерной загрузке.

- Weighted Round Robin. Взвешенный циклический алгоритм.

  Этот алгоритм является усовершенствованной версией предыдущего. Он присваивает каждому серверу «вес», который отражает его относительную производительность, и определяет количество запросов, которые будут направлены серверу. Например, если более мощный сервер имеет вес 5, то балансировщик отправит ему 5 запросов подряд, а затем перейдет к следующему. Это помогает учитывать разницу в производительности и распределять нагрузку более равномерно. Назначение и «калибровка» весов для каждого сервера — это кропотливое и трудозатратное занятие. Поэтому вместо определения веса сервера используется другая метрика — среднее время обработки запроса. Если какой-то сервер обрабатывает запросы в два раза быстрее, то он может обрабатывать в два раза больше запросов. Эта модификация называется *динамическим взвешенным циклическим алгоритмом* (Dynamic Weighted Round Robin). В отличие от оригинала, модификация не требует назначения веса при добавлении нового сервера, а также адаптируется к изменениям производительности сервера.

- Least Connections. Наименьшее количество подключений.

  Так как балансировщик находится между клиентом и сервером, то он знает, сколько клиентов подключено к каждому серверу. Эта информация может быть использована для распределения нагрузки: балансировщик выбирает тот сервер, у которого меньше всего активных соединений. Такой подход особенно полезен в системах, где продолжительность соединений может сильно варьироваться. Если один сервер занят обработкой долгих запросов, новый запрос будет перенаправлен на сервер с меньшей загруженностью. У этого алгоритма также есть модификация с весами, которые влияют на выбор только в случае нескольких серверов с одинаковым минимальным количеством соединений.

- Least Response Time. Наименьшее время ответа.

  Это усложненная модификация алгоритма Least Connections. Балансировщик отправляет свои служебные запросы к каждому из серверов и измеряет время ответа на служебные запросы. Выбор сервера осуществляется исходя из комбинации количества активных соединений и среднего времени ответа на служебные запросы.

- Peak Exponentially Weighted Moving Average. Пиковое экспоненциальное скользящее среднее.

  Идея алгоритма заключается в том, чтобы отслеживать время отклика каждого сервера и вычислять вес на основе скользящего среднего с учетом экспоненциального затухания. Это позволяет алгоритму более чутко реагировать на недавние изменения в производительности и меньше учитывать устаревшие данные. Таким образом, Peak EWMA дает приоритет серверам, которые демонстрируют

наименьшее среднее время отклика за последние запросы, но при этом «старые» задержки быстро теряют значимость по сравнению с новыми значениями.

- Random. Случайное распределение.

  Никаких сложностей — алгоритм случайным образом выбирает сервер, которому передает запрос. Не рекомендуется к использованию.

- Static Hashing. Статическое хеширование.

  В некоторых системах есть потребность в том, чтобы каждый запрос от одного пользователя стабильно попадал на один сервер. Для удовлетворения этой потребности используются алгоритмы на основе хеширования данных из запроса — например, IP-адреса пользователя. В простейшем случае для хеширования используется операция по модулю: результат хеш-функции делится на количество серверов, и остаток определяет, какой сервер обрабатывает запрос. Например, если результат хеш-функции равен 7, а у нас 3 сервера, то остаток от деления на 3 — это 1, значит, запрос отправляется на сервер 1. Статическое хеширование просто в реализации и обеспечивает предсказуемое распределение нагрузки. Однако оно имеет значительные недостатки при изменении числа серверов — при добавлении или удалении сервера большая часть ключей перераспределяется, что нарушает исходное условие и требует дополнительных действий на передачу сессионных данных от сервера, на который указывала хеш-функция раньше, к серверу, на который указывает хеш-функция сейчас.

- Consistent Hashing. Консистентное хеширование.

  Консистентное хеширование организует серверы и данные на виртуальном кольце. Это кольцо представляет собой диапазон значений, образуемых хеш-функцией, и условно замкнуто, — т. е. крайние значения связаны между собой, образуя цикл. Как серверы, так и ключи данных размещаются на этом кольце по их хеш-значениям, причем каждый сервер и каждый ключ данных получают уникальную позицию на кольце с помощью хеш-функции. Например, для хеширования серверов может использоваться их IP-адрес, а для хеширования данных — IP-адрес отправителя или уникальный идентификатор запроса. Запрос передается серверу, хеш которого ближайший по часовой стрелке к хешу запроса на кольце. Это гарантирует, что каждый сервер обслуживает определенный диапазон хешей. Когда новый сервер добавляется на кольцо, он занимает определенное положение и забирает часть ключей от ближайшего к нему сервера, тем самым перераспределяя нагрузку без перераспределения всех данных, которое происходит при статическом хешировании.

DNS

DNS, или Domain Name System, представляет собой систему, которая переводит удобные для человека доменные имена (такие как example.com) в числовые IP-адреса, необходимые для коммуникации в Интернете.

Хотя основная задача DNS — это трансляция человекочитаемых имен в машинное представление, DNS может использоваться для распределения пользовательских

запросов между несколькими серверами, связанными с одним и тем же доменным именем. Основная идея заключается в том, что DNS-сервер может возвращать несколько IP-адресов для одного и того же доменного имени. Когда пользователь отправляет запрос на доступ к веб-сайту, DNS-сервер обрабатывает этот запрос и возвращает список доступных IP-адресов, которые соответствуют серверам, обслуживающим этот домен. Исторически сложилось так, что клиент обычно использует тот IP-адрес, который указан первым, поэтому DNS-сервер может менять порядок IP-адресов в ответе, распределяя нагрузку между всеми серверами.

Достоинство этого способа — отсутствие балансировщика, который находится между клиентом и сервером и вносит дополнительные задержки при обработке запроса. Помимо этого, в случае недоступности сервера по первому адресу, клиенты пытаются подключиться к следующим адресам из списка, что позволяет «DNS-балансировщику» корректно обрабатывать выход из строя отдельных серверов.

Недостатки у балансировки через DNS тоже есть. Во-первых, она не учитывает текущее состояние серверов, что может привести к перегрузке менее мощных узлов. Во-вторых, кеширование DNS-ответов на клиенте может замедлить процесс обновления записей, и изменения могут не применяться мгновенно.

GeoDNS

GeoDNS представляет собой технологию, предназначенную для оптимизации распределения нагрузки на основе географического положения пользователей — клиент обращается к DNS-серверу, а DNS-сервер определяет, откуда поступает запрос, и подбирает наиболее подходящий сервер, находящийся ближе всего к пользователю.

Когда пользователь отправляет первый запрос, операционная система запрашивает трансляцию доменного имени у DNS-сервера. DNS-сервер использует локальную базу данных, чтобы определить приблизительное местоположение этого IP-адреса. На основе этой информации сервер принимает решение о том, какой из доступных серверов будет обслуживать запрос. Если, например, пользователь из Европы обращается к веб-сайту, зеркала которого размещены на серверах, находящихся на разных континентах, GeoDNS может направить его на сервер, расположенный в Европе, вместо того чтобы использовать сервер в США или Азии. Это уменьшает задержки и улучшает время загрузки страниц. Для GeoDNS также доступны все способы балансирования, свойственные обычному DNS.

Тем не менее GeoDNS имеет и некоторые ограничения. Одним из них является потенциальная неточность геолокации, поскольку точность определения местоположения способна варьироваться в зависимости от используемых баз данных геолокации. В некоторых случаях пользователь может получить доступ к серверу, который является ближайшим физически, но из-за особенностей маршрутизации в глобальной сети доступ к нему не является быстрейшим. Еще одной сложностью может стать управление DNS-записями — GeoDNS требует более сложной настройки и администрирования, особенно в крупных и динамических системах. Также необходимо учитывать, что GeoDNS не всегда обеспечивает 100% надежно-

сти, поскольку DNS-записи кешируются, что может привести к задержкам в обновлении информации о доступных серверах.

Обратный прокси-сервер

Обратный прокси-сервер — это посредник между клиентами и внутренними серверами, который принимает запросы от пользователей и направляет их к нужному серверу и сервису в зависимости от его настроек. Взаимодействие с обратным прокси выглядит как общение с самим целевым сервером, однако на самом деле клиент никогда не взаимодействует с сервисами напрямую.

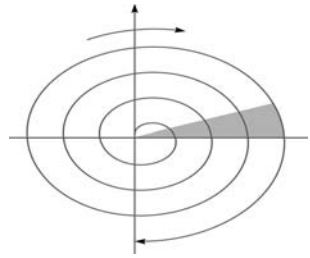
Обратный прокси-сервер может выполнять функцию балансировщика нагрузки, реализуя все описанные ранее алгоритмы балансировки, а также и функции защиты, обеспечивая SSL-терминацию: он принимает зашифрованный трафик, расшифровывает его, передает требуемому сервису в открытом виде, а затем шифрует ответ от сервиса и передает зашифрованные данные обратно клиенту. Это снижает сложность настройки сервисов, которым не приходится реализовывать шифрование, и упрощает управление сертификатами.

Третья полезная функция обратного прокси — кеширование ответов от сервисов. Ответы на популярные запросы кешируются на определенный промежуток времени, и обратный прокси может отдавать ответ без обращения к соответствующему сервису.

Хотя обратный прокси-сервер способен выполнять задачи балансировщика, основная задача обратного прокси — сокрытие внутренней инфраструктуры и контроль доступа к сервисам.

Наиболее популярные обратные прокси-серверы: Apache HTTP Server, Nginx и HAProxy. Каждое из этих решений подходит для разных целей: Nginx и Apache HTTP Server — это универсальные решения для большинства задач, а HAProxy — узкоспециализированный инструмент для высокой нагрузки.

ГЛАВА 9



Обслуживание

В этой главе обсуждаются детали, которые могут показаться незначительными или слишком специфичными при проектировании информационных систем, но с ростом и развитием реализованной системы эти особенности обязательно дадут о себе знать. Лучшие практики проектирования рекомендуют предусмотреть такие риски как можно раньше.

Логирование

Логирование — это процесс записи данных о событиях, происходящих внутри системы, с целью отслеживания ее работы, выявления и анализа ошибок. В процессе работы информационной системы внутри нее происходят события, обрабатывая которые система может формировать информационные сообщения, описывающие текущее состояние процесса и некоторые его промежуточные данные. Эти сообщения записываются в текстовые лог-файлы (логи).

В случае сбоев в информационной системе логи позволяют понять причины проблемы и установить точное место ее возникновения, что существенно упрощает диагностику и исправление. Логирование помогает не только фиксировать сами ошибки, но и собирать дополнительную информацию, которая может быть полезна для понимания контекста: к какому пользователю относится ошибка, в какое время она произошла, какие действия предшествовали сбою и т. д.

Но логирование — это запись в файл, а взаимодействие с файлами — это медленные операции, что ставит перед разработчиком выбор из двух противоположностей: записывать подробные логи, но замедлить программу, или отказаться от логов в угоду производительности. При разработке и отладке, очевидно, используются подробные логи, которые включают информацию обо всех выполняемых операциях и возникающих ошибках. В продуктовой среде выбор более очевиден — создание подробных логов станет замедлять систему, а также излишне подробные логи могут быстро исчерпать объем хранилища, поэтому в продуктовых окружениях используют более краткие логи, которые тем не менее должны быть полезны при

анализе. Оптимальный выбор уровня логирования позволяет поддерживать баланс между полнотой данных и эффективностью использования ресурсов.

В распределенных системах применяется *централизованное логирование*, которое объединяет данные с различных серверов в одном месте, что упрощает их анализ и управление. Для централизованного логирования задействуются различные инструменты, которые помогают собирать, хранить и анализировать логи из разных систем и приложений. К наиболее популярным и функциональным инструментам относятся Elastic Stack (ELK) и Graylog:

- Elastic Stack (ELK).

  Это популярный набор инструментов с открытым исходным кодом для сбора, обработки, анализа и визуализации логов и данных. Название ELK складывается из первых букв трех основных компонентов набора: Elasticsearch, Logstash и Kibana:

  - Elasticsearch — распределенная поисковая и аналитическая система, используемая для хранения и поиска данных. Она отличается высокой скоростью работы и возможностью горизонтального масштабирования, что позволяет обрабатывать огромные объемы данных практически в реальном времени. Elasticsearch хранит логи и метаданные в индексах, которые можно быстро искать, фильтровать и анализировать, применяя сложные запросы;
  - Logstash — инструмент для сбора, обработки и фильтрации данных. Он используется для получения данных из различных источников (файлы логов, базы данных, системы событий и метрик) и их преобразования перед отправкой в Elasticsearch. Logstash поддерживает множество плагинов для интеграции с разными системами и форматами данных, что позволяет гибко настраивать процесс сбора логов. В Logstash данные можно фильтровать, очищать, структурировать и дополнять — это делает его мощным инструментом для подготовки логов к анализу;
  - Kibana — интерфейс для визуализации данных из Elasticsearch. Он позволяет создавать интерактивные визуальные представления, помогающие анализировать логи и отслеживать тренды.

  ELK Stack используется не только для хранения данных, но и для обнаружения аномальной активности и возможных угроз безопасности, поскольку разрешает настраивать оповещения на подозрительные события. ELK Stack предлагает множество удобств для анализа логов, но может потреблять много ресурсов, особенно на этапах обработки и индексирования больших объемов данных.

- Graylog.

  Это менее ресурсозатратная альтернатива Elastic Stack. Graylog проще в установке и управлении, быстрее в обработке данных и предоставляет встроенные уведомления, что делает его удобным для оперативного мониторинга.

Мониторинг

Мониторинг — это процесс непрерывного сбора, анализа и интерпретации данных о состоянии и производительности различных компонентов информационной системы. Он включает в себя наблюдение не только за компонентами системы, но и оборудованием: серверами, сетевыми устройствами и прочими — чтобы обеспечить их стабильную работу и минимизировать время простоя.

Ключевым элементом мониторинга является сбор метрик и логов, которые могут содержать данные об использовании вычислительных ресурсов серверов. Эти данные помогают идентифицировать проблемы, возникающие в системах, и позволяют администраторам принимать обоснованные решения для их устранения. Эффективный мониторинг также дает возможность заранее обнаруживать потенциальные проблемы, что способствует повышению общей надежности системы.

Системы мониторинга могут быть как внутренними, так и внешними. *Внутренние системы* анализируют метрики непосредственно из инфраструктуры, в то время как *внешние системы* могут проверять доступность и производительность сервисов с точки зрения пользователей. Кроме того, мониторинг может быть как активным, так и пассивным. В *активном режиме* система способна периодически проводить тесты на доступность, отправлять запросы и отслеживать ответы, тогда как в *пассивном режиме* она просто собирает данные, которые создаются в процессе работы приложений.

Наиболее распространенные инструменты для внутреннего мониторинга: Prometheus, Zabbix, Nagios, New Relic, Datadog, Sentry. Каждый из них предлагает уникальные функции и возможности настройки, что позволяет выбирать наиболее подходящее решение в зависимости от потребностей конкретной организации.

Некоторые средства мониторинга дают возможность не только наблюдать за изменением метрик системы, но и собирать данные в случае нештатных ситуаций. Например, Sentry можно интегрировать с системой, и в случае необработанного исключения при выполнении система мониторинга сохранит запись об инциденте. Такая запись будет содержать описание исключения, стек вызовов и все доступные входные параметры. В этом случае система мониторинга делает чуть-чуть больше, чем простое логирование, предоставляя максимум информации, когда исключение уже произошло. Системы мониторинга также могут записывать действия пользователей, что, с одной стороны, упрощает воспроизведение дефектов, а с другой — может быть негативно воспринято пользователями.

Оповещение

Механизм *оповещений* информирует разработчиков и сотрудников поддержки о возникновении в информационной системе проблем или событий, требующих внимания. Этот инструмент позволяет реагировать на инциденты в реальном времени, что помогает предотвращать длительные простои и обеспечивать стабильность сервисов.

Система оповещений обычно настраивается с учетом специфики инфраструктуры и бизнес-процессов компании. Ключевая функция здесь — настройка триггеров или условий, при которых отправляется оповещение. Например, это может быть превышение допустимого уровня загрузки процессора, недостаток свободной памяти, длительное время обработки запросов, ошибки в логах или снижение доступности веб-сайта. Такие условия помогают выделить важные события, игнорируя при этом несущественные колебания в данных.

Оповещения могут отправляться через различные каналы — в зависимости от предпочтений команды и характера инцидента. К популярным каналам относятся электронная почта, SMS, мессенджеры (например, Slack, Microsoft Teams), а также специализированные системы управления инцидентами. Некоторые инструменты оповещения поддерживают интеграцию с системами управления задачами и системами контроля версий, что упрощает управление инцидентами и фиксацию процесса их решения.

Интеллектуальная настройка оповещений является важной частью эффективного мониторинга, т. к. избыточные оповещения могут привести к «шуму» и информационной перегрузке. Если каждый инцидент вызывает уведомление, то высок риск того, что критически важные события окажутся незамеченными. Поэтому многие системы предлагают функции для управления частотой оповещений, группировки и подавления событий, а также настройки эскалаций — т. е. передачи оповещения на более высокий уровень, если инцидент не был обработан в заданное время.

Анализаторы исходного кода

Анализаторы исходного кода — это инструменты, предназначенные для автоматического анализа программного кода с целью выявления ошибок, уязвимостей, несоответствия стандартам кода и других проблем. Анализ кода разделяется на две категории: статический анализ и динамический анализ:

- Статический анализ — это проверка исходного кода без его выполнения.

  Такой вид анализа позволяет выявлять проблемы на ранних этапах разработки — до выполнения программы. Статические анализаторы могут находить синтаксические ошибки, распространенные синтаксически корректные опечатки, потенциальные утечки памяти, неправильные и недостижимые условия, а также несоответствие принятому стилю написания кода. Статический анализ может быть неэффективен в языках с динамической типизацией.

- Динамический анализ — это проверка работы программы во время ее выполнения.

  Такой вид анализа позволяет выявлять ошибки, которые могут возникнуть только в процессе выполнения: проблемы с производительностью, происходящие утечки памяти и ошибки при взаимодействии потоков. Динамический анализ также может применяться для построения различных UML-диаграмм по факту выполнения программ.

Современные анализаторы исходного кода могут интегрироваться в процессы CI/CD, что позволяет автоматически проверять код при каждом коммите или перед сборкой. Это ускоряет процесс разработки и облегчает контроль качества, т. к. ошибки и несоответствия обнаруживаются автоматически, без необходимости ручных проверок.

Сканеры уязвимостей

Сканеры уязвимостей — это инструменты, предназначенные для автоматического поиска и анализа уязвимостей в программном обеспечении, сетях и системах. Они помогают выявлять потенциальные угрозы безопасности, позволяя оперативно на них реагировать и устранять проблемы до того, как они могут быть использованы злоумышленниками.

Сканеры уязвимостей работают по принципу обнаружения известных уязвимостей, используя базы данных с информацией о них — такие как National Vulnerability Database (NVD) или Open Source Vulnerability Database (OSVDB). Процесс сканирования обычно начинается с определения серверов и сервисов, которые необходимо проверить. После этого сканер выполняет проверки, собирая доступные данные о конфигурациях, открытых портах и версиях установленного программного обеспечения. В результате сканер формирует отчет, в котором указываются найденные уязвимости, их уровень критичности и рекомендации по устранению.

Уязвимости в базах данных имеют рейтинг, который позволяет разработчикам приоритизировать свои действия и сосредоточиться на устранении наиболее серьезных угроз. Например, уязвимости могут быть оценены по шкале CVSS (Common Vulnerability Scoring System), которая учитывает такие факторы, как возможность эксплуатации, влияние на систему и сложность атаки.

Несмотря на свою полезность, сканеры уязвимостей имеют и определенные ограничения — они не могут обнаруживать уязвимости, которые еще не были добавлены в базы данных, и анализировать логические ошибки, способные привести к сбою программного обеспечения. Тем не менее сканеры уязвимости — хорошее дополнение к статическим и динамическим анализаторам.

К популярным сканерам уязвимостей относятся такие инструменты, как Nessus, OpenVAS, Burp Suite и Acunetix. Каждый из них предлагает различные функции, интерфейсы и методики сканирования, что позволяет выбирать наиболее подходящий инструмент в зависимости от специфики разрабатываемого программного обеспечения.

Аудит-логи

Аудит-логи — это специальные журналы, которые фиксируют события, связанные с доступом пользователей к системе, а также изменения, внесенные в данные или настройки системы. Цель аудит-логов — обеспечить прозрачность действий пользователей и административных операций, что помогает организациям поддержи-

вать безопасность и соблюдать требования законодательства. Такие логи предоставляют детализированную информацию о том, кто, когда и какие действия совершил в системе. Для каждой записи обычно указываются точное время события, идентификатор пользователя, IP-адрес устройства и другие данные.

Аудит-логи являются важным источником информации для расследования инцидентов безопасности и анализа активности. Это означает, что записи должны быть защищены от изменения или удаления. Для этого часто используются специальные механизмы шифрования, контроль целостности и ограничение доступа к самим логам.

Аудит-логи играют важную роль в обеспечении соблюдения локальных и международных нормативных требований, которые обязывают организации контролировать доступ к данным и обеспечивать прозрачность действий пользователей. В случае утечки данных или другого инцидента аудит-логи позволяют восстановить последовательность событий и выявить, кто и когда имел доступ к информации.

Аварийные игры

Аварийные игры, или инженерия хаоса — это практика, направленная на улучшение устойчивости и надежности распределенных систем путем проведения контролируемых экспериментов в реальных условиях. Суть практики заключается в намеренном создании сбоев и нестабильных условий для наблюдения за поведением системы и ее реакцией на эти сбои. Основная цель аварийных игр — выявить слабые места в системе и улучшить ее способность справляться с неожиданными ситуациями.

Сбои в сложных распределенных системах неизбежны. Они могут быть вызваны различными факторами — такими как сбой оборудования, проблемы с сетью или ошибка в программном обеспечении. Аварийные игры позволяют командам выявлять потенциальные слабые места и укреплять архитектуру системы до того, как реальные сбои произойдут. Важно, что такие эксперименты проводятся в контролируемой среде, где можно заранее определить параметры теста и контролировать последствия.

Процесс проведения аварийных игр обычно включает несколько этапов. Сначала определяется, какие системы или компоненты необходимо протестировать, а также устанавливается гипотеза о том, как система должна реагировать на сбой. Затем выполняется эксперимент, в ходе которого создаются сбои — например, отключаются отдельные сервисы или серверы, имитируются задержки в сети. В ходе эксперимента отслеживаются различные метрики производительности, чтобы определить реакцию системы на инцидент.

Результат аварийных игр позволяет выявлять слабые места и потенциальные точки отказа. На основе этих данных могут быть внесены изменения в архитектуру, конфигурации или процессы реагирования на инциденты. Кроме того, аварийные игры помогают развивать культуру надежности и сотрудничества в команде, т. к. они способствуют обсуждению ошибок и недостатков, а не наказанию за них.

Пионер в области аварийных игр, компания Netflix, разработала Chaos Monkey — инструмент для автоматизации процесса создания нештатных ситуаций. Этот инструмент позволяет проверить, как приложения и инфраструктура реагируют на неожиданные сбои и отказы.

Chaos Monkey работает по довольно простому принципу — он случайным образом выбирает виртуальные машины или экземпляры сервисов в облачной среде и отключает их. Это может показаться рискованным, однако цель таких тестов заключается в том, чтобы убедиться, что системы способны выдерживать сбои без значительных последствий для пользователей. Внедрение инструментов типа Chaos Monkey в автоматическом режиме оправдано только после обеспечения резервирования и автоматического восстановления системы, поскольку преждевременное использование инструментов хаоса может привести к негативным последствиям.

Миграция схемы и миграция данных

Миграции схемы и данных необходимы для поддержания системы в актуальном состоянии и обеспечения ее соответствия требованиям, которые могут изменяться со временем. Когда бизнес или функциональные требования к приложению меняются, чтобы поддерживать новую функциональность обычно требуется изменить существующие структуры данных: добавить новые поля, таблицы или связи.

Во время первоначальной разработки программного обеспечения можно обойтись без миграций — пока нет продуктовых данных, допустимо полностью удалить базу данных и создать новую с обновленной структурой. Но после выпуска программного обеспечения база данных заполняется полезными данными, и способ с пересозданием больше не подходит.

Миграция схемы баз данных — это внесение изменений в структуру базы данных. Такая миграция, как правило, управляется с помощью систем контроля версий для баз данных, что делает ее похожей на версионирование кода. Каждый этап миграции представляет собой шаг, в котором описаны изменения, а также порядок их выполнения. Эти изменения обычно фиксируются в виде отдельных файлов, скриптов или SQL-команд, которые сохраняются и отслеживаются. Это помогает разработчикам видеть историю изменений схемы, понимать, какие версии схемы существуют и как миграции влияют на структуру данных. Для управления миграциями используются либо компоненты фреймворков, обеспечивающих взаимодействие с базой данных, такие как Django Migrations или Alembic, либо отдельные приложения — например, Liquibase. Эти инструменты упрощают создание и управление миграциями, автоматически генерируя нужные команды SQL на основе изменений в модели данных. Инструменты для миграции также поддерживают возможность возврата к предыдущей версии схемы, что особенно полезно в случае ошибок.

Миграция способна создать сложности, особенно, в крупных приложениях с большим объемом данных и зависимостей между таблицами. Например, изменение типа данных столбца, который активно используется, может привести к сбоям в работе приложения. Поэтому миграции должны быть тщательно спланированы и протестированы. Для этого, как правило, миграции сначала проводят на тестовых

базах данных — это дает возможность убедиться, что изменения не приведут к ошибкам, потере данных или снижению производительности.

Автоматизация миграций помогает уменьшить количество ошибок, т. к. все изменения фиксируются и выполняются строго в порядке, заложенном в файлах миграций. Каждый миграционный файл или скрипт имеет номер или временну́ю метку, благодаря чему поддерживается последовательность выполнения. Например, при развертывании новой версии приложения система может автоматически применить только те миграции, которые еще не были выполнены, что минимизирует трудозатраты и риск пропустить какой-либо шаг.

Важно отметить, что миграция *схемы базы данных* не выполняет операций над данными, которые содержатся в базе данных, хотя простые операции, такие как создание нового столбца, могут также заполнить ячейки столбца значениями по умолчанию. Однако более сложные изменения не всегда выполнимы средствами миграции схемы. Представим, что в базе данных есть столбец, который содержит несколько строк, разделенных запятой. После очередной итерации обсуждения принимается решение разделить этот столбец на пять столбцов булева типа, а значение «истина» или «ложь» будет зависеть от вхождения имени новых столбцов в значения из старого столбца. Есть несколько способов миграции таких сложных случаев.

Первый вариант — это миграция в несколько шагов. На первом шаге — создание новых столбцов. На втором — изменение системы таким образом, чтобы новые столбцы были предпочтительным источником информации, но в случае отсутствия в них информации должен использоваться резервный источник информации — старые столбцы. На третьем шаге — фоновая конвертация старого формата в новый формат. После завершения конвертации система изменяется таким образом, чтобы новые столбцы были единственным источником информации. Заключительный шаг — вторая миграция, которая удаляет старый столбец. Достоинство этого вида миграций — их прозрачность для пользователей системы. Такой подход позволяет реализовать миграции даже на очень больших базах данных. Однако подобную миграцию нельзя полностью автоматизировать, поскольку процесс состоит из двух миграций схем, между которыми необходимо запустить определенную сборку системы для конвертации данных старого формата в данные нового формата.

Второй вариант — это автономный скрипт миграции, который собирает данные и «перекладывает» их в новых формат. В отличие от первого варианта, скрипт миграции содержит минимальный набор моделей, необходимый для конвертации данных, и этот скрипт может быть запущен отдельно от информационной системы. Такой вариант может быть автоматизирован, но при запуске в продуктовом окружении с большим объемом данных способен вступить в конфликт с основной системой.

Версионирование

Версионирование программного обеспечения — это процесс присвоения уникальных номеров его сборкам. Каждый номер версии помогает отслеживать изменения и обеспечивает четкое представление о стадии разработки, новых функциях, ис-

правлениях и совместимости программного обеспечения. Версионирование используется как разработчиками, так и пользователями, для понимания текущего состояния продукта и его изменений.

- Семантическое версионирование.

  Наиболее распространенной схемой версионирования считается семантическое версионирование. В этой схеме версия представляется в формате MAJOR.MINOR.PATCH, где каждая часть указывает на тип и степень изменений. Значение MAJOR увеличивается при внесении значительных изменений, которые могут нарушить совместимость с предыдущими версиями. MINOR обозначает добавление новых функций, которые сохраняют обратную совместимость в рамках MAJOR-версии, а PATCH используется для исправления ошибок и небольших улучшений, не влияющих на работу уже существующих функций. Семантическое версионирование определяет, что версия 0.MINOR.PATCH используется с начала разработки, и любые интерфейсы таких версий не должны считаться стабильными. При первом публичном выпуске программного обеспечения MAJOR-версия должна быть изменена на единицу. Семантическое версионирование также допускает использование дополнительных идентификаторов для предрелизных версий — например: 1.0.1-beta2. Эта схема удобна тем, что сразу дает понять пользователю, насколько существенны изменения в новой версии.

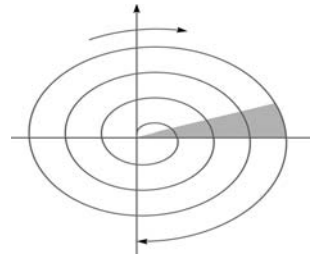
- Версионирование датой.

  Второй популярный подход к версионированию — использование даты для указания версий: ГОД-МЕСЯЦ-ДЕНЬ. Например, версии операционной системы Ubuntu определяются как ГОД.МЕСЯЦ, 22.04 для версии, выпущенной в апреле 2022 года. Такой подход особенно удобен для выпуска планируемых регулярных обновлений, поскольку дает пользователю ориентир по дате выпуска. В отличие от семантического версионирования, подход на основе версионирования датой не дает информации об объеме внесенных изменений, новой функциональности и совместимости между версиями.

- Маркетинговое версионирование.

  В некоторых случаях «маркетинговая» версия продукта при запуске отличается от внутренней версии, составленной по другим правилам. Например, Windows 2000 новее, чем Windows 95, но старее, чем Windows 10. В публичных «версиях» порядок и количество изменений неочевидно, но внутреннее название: Windows NT 5.0, Windows NT 3.95 и Windows NT 10.0 соответственно — вносит ясность. Аналогично произошло с Adobe Photoshop: вместо девятой мажорной версии маркетинговое название изменилось на CS, а затем на CS2. Спустя шесть версий, после CS6 появилась CC, а после — CC 2015 и версионирование по годам. Однако внутренняя версия так и осталась семантической.

ГЛАВА 10



Отказоустойчивость, резервирование и надежность

Отказоустойчивость, резервирование и надежность — это ключевые концепции, связанные с обеспечением устойчивости и непрерывности работы систем, особенно в критически важных приложениях. Рассмотрим их подробно.

Надежность системы

Надежность системы — это ее способность выполнять заданные функции без сбоев в течение определенного времени. Надежность важна в проектировании и эксплуатации как технических, так и программных систем, особенно в областях, где сбои могут иметь серьезные последствия, — например, в авиации, медицине или банковской сфере.

В основе надежности лежит несколько факторов. Во-первых, это устойчивость компонентов системы к отказам — *отказоустойчивость*. Надежные системы проектируются так, чтобы минимизировать вероятность отказов, что достигается выбором качественных материалов, проверенных технологий и тщательной разработкой. Например, в электронике применяются компоненты с высокой устойчивостью к перегрузкам, а в программном обеспечении используются методы статического анализа кода для выявления ошибок до этапа эксплуатации.

Во-вторых, в случае сбоя важна *способность системы к быстрому восстановлению*. Если ошибка произошла, система должна минимизировать ее влияние на конечного пользователя. Это может быть достигнуто за счет резервирования ресурсов, автоматического восстановления и встроенных механизмов мониторинга.

С точки зрения математического описания, надежность системы часто выражается через такие показатели, как *среднее время наработки на отказ* (MTBF, Mean Time Between Failures) и *среднее время восстановления* (MTTR, Mean Time To Recovery). Эти метрики помогают оценить вероятность того, что система будет работать без сбоев в заданный период времени, а также понять, сколько времени потребуется на устранение последствий возможного сбоя. Для высоконадежных систем критически важно, чтобы MTBF был максимально большим, а MTTR — минимальным.

- Среднее время наработки на отказ вычисляется как отношение времени работы ко времени внепланового простоя за выбранный промежуток времени. Например, если система за месяц работала 28 дней, и за это время произошло 2 сбоя, то MTBF = 14 дней. При подсчете простоя принимаются во внимание только внеплановые остановки системы и не учитываются плановые работы.
- Среднее время восстановления за промежуток времени рассчитывается аналогичным образом — это отношение суммы времени всех восстановлений к количеству сбоев. Например, если за полгода произошло 10 сбоев, и суммарно было затрачено 420 минут для восстановления работоспособности, то MTTR = 42 минуты.

Коэффициент готовности показывает вероятность того, что система окажется в рабочем состоянии в определенный момент времени. Этот коэффициент рассчитывается как отношение MTBF к сумме MTBF и MTTR. Например, коэффициент готовности, равный 99.99% (так называемые четыре девятки), означает, что система может быть недоступна не более 52 минут в году.

Отказоустойчивость

Отказоустойчивость — это способность системы продолжать функционировать, даже если некоторые ее компоненты вышли из строя. Такое свойство систем достигается за счет применения методов и технологий, минимизирующих влияние сбоев.

Обеспечение отказоустойчивости заключается в том, чтобы предвидеть возможные точки отказа и спроектировать систему таким образом, чтобы она могла либо полностью продолжать свою работу, либо ограничить последствия сбоя. Одним из основных подходов к обеспечению отказоустойчивости является *резервирование*. Отказоустойчивость связана с понятием «зоны отказа». Эта концепция предполагает, что система должна быть спроектирована таким образом, чтобы сбой в одном компоненте не привел к каскадным отказам других частей.

Для достижения отказоустойчивости также важно включать механизмы автоматического обнаружения и восстановления после сбоев. Это может быть реализовано через системы оркестрации и мониторинга, которые непрерывно отслеживают состояние системы и запускают процедуры восстановления в случае выявления аномалий.

Резервирование

Резервирование — это один из способов обеспечения отказоустойчивости, связанный с созданием дополнительных ресурсов, которые могут быть задействованы при сбоях. Основная идея резервирования заключается в наличии дублирующих элементов, которые могут мгновенно или с минимальной задержкой взять на себя выполнение функций отказавших компонентов. Это может быть как физическое дублирование оборудования, так и использование программных методов — например, создание виртуальных резервных копий.

Резервирование может быть организовано по-разному в зависимости от уровня готовности резервных ресурсов.

- В случае *горячего резервирования* (hot standby) резервные компоненты работают параллельно с основными и могут немедленно взять на себя нагрузку в случае отказа. Этот подход особенно эффективен для систем, требующих высокой доступности.
- *Теплое резервирование* (warm standby) предполагает, что резервный компонент находится в режиме ожидания, и начинает работу только после активации. Например, это может быть сервер, который включается при обнаружении сбоя основного, но требует определенного времени для загрузки и синхронизации данных. Этот подход снижает затраты на энергопотребление и обслуживание по сравнению с горячим резервированием, но добавляет некоторую задержку в процесс восстановления.
- *Холодное резервирование* (cold standby) использует резервные компоненты, которые не активны до тех пор, пока не произойдет сбой, и их запуск может потребовать значительного времени. Это подходит для менее критичных систем, где не требуется мгновенного восстановления, но важно минимизировать последствия полного отказа. Примером может служить резервное оборудование, хранящееся на складе и готовое к установке в случае необходимости.

Резервное копирование

Резервирование применяется не только на уровне оборудования, но и на уровне данных. Для этого используются такие методы, как резервное копирование и репликация. *Резервное копирование* создает копии данных, которые могут быть восстановлены в случае потери информации, а *репликация* обеспечивает их синхронное или асинхронное копирование на резервные устройства. Это позволяет минимизировать риск потери данных и ускорить восстановление после сбоев.

Процесс резервного копирования (бэкап) может быть организован разными способами:

- полный бэкап включает копирование всех данных, что позволяет восстановить данные в исходном состоянии на момент создания резервной копии. Этот подход надежен, но требует значительных ресурсов;
- дифференциальное копирование фиксирует только изменения, произошедшие с последнего полного бэкапа, что сокращает объем резервируемых данных и время, необходимое для копирования;
- третий вариант — инкрементальное копирование — сохраняет лишь те изменения, которые были сделаны с момента последнего резервного копирования (независимо от его типа). Этот метод требует минимальных ресурсов, но усложняет восстановление, т. к. потребуется последовательно применять несколько наборов данных.

Кроме того, резервное копирование может быть горячим или холодным:

- горячее копирование выполняется на работающей системе без ее остановки на обслуживание, что удобно для систем с высокими требованиями к доступности.

Однако это сложнее в реализации и требует механизмов, предотвращающих конфликт между копированием и изменением данных;

- холодное копирование предполагает приостановку работы системы, что обеспечивает целостность копий, но может быть проблематичным для систем, где важна постоянная доступность.

Для обеспечения максимальной эффективности резервного копирования следует использовать правило 3-2-1. Оно предполагает, что необходимо иметь как минимум три копии данных, хранящихся на двух разных типах носителей, причем одна из копий должна находиться в удаленной локации. Это позволяет снизить риск полной утраты данных даже при масштабных сбоях.

Кроме того, резервное копирование требует регулярности. Автоматизация этого процесса с использованием специализированного программного обеспечения помогает снизить вероятность человеческой ошибки и обеспечивает своевременное обновление копий. Современные решения для резервного копирования могут также включать функции шифрования данных, что предполагает их защиту даже в случае компрометации хранилища. Однако в резервном копировании важен не только факт создания резервных копий, но и проверка того, что восстановление из резервной копии возможно.

Схемы резервирования

Резервирование по принципу N+1 или N+M используется для масштабных систем, где вместо полного дублирования всех компонентов добавляются дополнительные компоненты, которые могут заменить любой из вышедших из строя. Например, в дата-центре с десятью серверами может быть добавлен один резервный сервер, готовый заменить любой основной (схема N+1). Это оптимизирует затраты, обеспечивая высокий уровень отказоустойчивости.

Географически распределенное резервирование стало особенно актуальным с развитием облачных технологий. В этом подходе данные и ресурсы дублируются в нескольких физически удаленных локациях. Это защищает систему от масштабных сбоев — таких как природные катаклизмы или перебои в энергоснабжении в одном регионе.

Восстановление после катастроф

Восстановление после катастроф (Disaster Recovery) — это процесс и набор стратегий, направленных на обеспечение непрерывности работы и восстановление системы, данных и инфраструктуры после серьезных инцидентов, таких как природные катастрофы, кибератаки, аварии оборудования или человеческие ошибки. Оно представляет собой критический аспект управления рисками, особенно для организаций, где прерывание работы может повлечь за собой значительные финансовые потери, нарушение обязательств перед клиентами или ущерб репутации.

Основная цель Disaster Recovery заключается в минимизации времени простоя и потери данных. Для этого разрабатывается и внедряется план восстановления,

который детализирует действия, необходимые для возобновления работы после катастрофы. Такой план учитывает не только технические аспекты, но и организационные меры, включая координацию команд, информирование заинтересованных сторон и соблюдение нормативных требований.

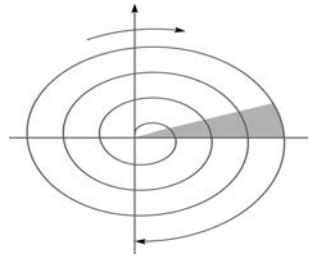
Процесс восстановления начинается с анализа рисков и уязвимостей системы. Это позволяет определить возможные сценарии катастроф, такие как пожар, наводнение, кибератака или массовый сбой оборудования. На основе этих сценариев определяются критически важные ресурсы и функции, которые должны быть восстановлены в первую очередь.

Методы Disaster Recovery включают использование резервного копирования, репликации данных и географически распределенных ресурсов. Резервное копирование позволяет сохранить копии данных на внешних носителях или в облаке, что защищает их от локальных повреждений. Репликация данных в режиме реального времени, напротив, обеспечивает почти мгновенное восстановление, т. к. данные синхронизируются между основным и резервным центрами обработки. Географическое распределение ресурсов предотвращает риски, связанные с региональными инцидентами, такими как землетрясения или ураганы.

Облачные технологии играют ключевую роль в современных подходах к Disaster Recovery. Облачные платформы позволяют быстро развернуть виртуальные ресурсы в случае сбоя, обеспечивая гибкость и масштабируемость. Например, организации могут использовать модели Disaster Recovery as a Service (DRaaS), где облачные провайдеры предлагают готовые решения для автоматического восстановления систем и данных.

Тестирование и регулярное обновление плана Disaster Recovery имеют критическое значение. Они позволяют убедиться, что все процессы работают эффективно, а сотрудники знают свои обязанности в случае катастрофы. В тестировании используются как симуляции инцидентов, так и контрольные проверки — например, восстановление из резервных копий. Кроме того, планы должны регулярно пересматриваться, чтобы учитывать изменения в инфраструктуре, приход новых сотрудников, а также изменения в бизнес-процессах.

ЧАСТЬ II



Примеры проектирования систем

Проектирование информационных систем — это не только выбор подходящих подходов, практик и технологий, но и множество компромиссов между скоростью, качеством и стоимостью разработки. У любой проектируемой программы есть цели и задачи, которые решаются созданием этой программы. Поэтому при проектировании важно периодически задаваться вопросом: «Зачем я это делаю?» и продолжать при наличии ответа. Можно выделить две категории ответов на этот вопрос:

- Академический интерес.

  Во-первых, информационные системы можно реализовывать исключительно с академическим интересом. Это может быть как желание изучить те или иные технологии, так и забавы из категории «ненормальное программирование», когда инструменты используются не по назначению. Проектирование ради обучения дает максимальную свободу. Когда изучаешь материал сам, то сроки могут быть максимально гибкими. Проектируемая система может быть игрушечной и, следовательно, не требовать поддержки после разработки. Неудачные решения при разработке можно учесть, но они не будут создавать сложности в будущем. Ограничения в академической версии в основном «синтетические» и продиктованы желанием что-то изучить или бросить себе вызов.

- Коммерческий интерес.

  Во-вторых, информационные системы проектируют и реализуют для извлечения прибыли путем решения чьих-либо проблем. В этом варианте появляется множество ограничений, которые требуют внимания. Например, написание идеального кода, конечно, сильно упростит развитие системы. Но если конкурент выпустит информационную систему раньше вас, то пользователи пойдут к нему. Получается, что лучше хоть как-то, но быстро, чем идеально, но поздно. Этот принцип подтверждают социальные сети. Каждый может запустить свою социальную сеть, но сделать ее конкурентоспособной существующим соцсетям будет невероятно сложно, если не невозможно. Спешить и делать все некачественно, но очень быстро — тоже не лучшая тактика. Хотя подтверждения этому прин-

ципу найти сложно, ведь «некачественно» — это субъективная метрика, шутки про рефакторинг в среде программистов — не шутки. При развитии информационных систем рано или поздно возникнет ситуация, когда что-то нужно переделать.

Проектирование систем — это не просто выбор технологий и связывание их в единое целое, это также умение грамотно распределять ресурсы и даже немного заглядывать в будущее. Архитектура может быть технически безупречна, но какой в этом толк, если пользователь предпочел продукт конкурента, который вышел раньше. Или идеальная архитектура может перестать быть идеальной завтра, когда из-за бизнес-требований придется переделывать всю информационную систему или, того хуже, делать «неочевидные решения», ухудшая качество кода.

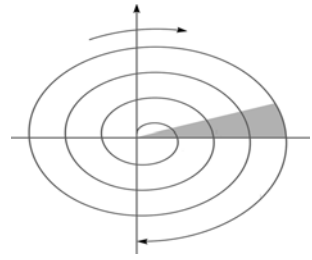
Эта часть книги посвящена проектированию, исходным данным, размышлениям и примерам того, как можно создать нужную информационную систему.

Здесь нет универсального решения, но есть ход мысли, который поможет при решении собственных задач.

Примечание

Далее рассматриваются примеры проектирования систем безотносительно языков программирования. Для каждой задачи необходимо выбирать тот инструмент, который лучше всего решает задачу. Хотя в тексте вы найдете упоминания про языки программирования, для консистентности все представленные диаграммы классов используют синтаксис, близкий к синтаксису языка Java. Это обосновано статической типизацией языка и педантичной реализацией его объектно-ориентированной парадигмы, что позволяет более наглядно объяснить строение информационной системы.

ГЛАВА 11



Проектирование неинтерактивного редактора изображений

В этой главе проектируется неинтерактивный редактор изображений, выполняемый в командной строке на компьютере пользователя. Мы рассмотрим здесь сначала монолитную архитектуру, а затем модификацию приложения при появлении новых требований.

Исходные требования

Оригинальное задание — это практическая работа из книги Игоря Жиркова «Low-Level Programming»[1]. Нам предлагается реализовать программу на языке C, которая должна поворачивать изображение формата BMP любого разрешения на 90 градусов против часовой стрелки. Также существуют дополнительные задания на оценку «хорошо» и «отлично»:

- поворот на произвольный угол;
- применение эффекта размытия по Гауссу;
- применение эффекта сепии.

Хотя это задание имеет академический интерес, его можно свести и к коммерческому. Оценка за работу — это «прибыль», которую можно получить, преподаватель — это заказчик, который будет проводить приемочное тестирование, а срок выполнения практической работы — это «конкуренты», которые могут выпустить свой продукт раньше и из-за которых можно получить меньше выгоды.

Рассмотрим возможные способы организации программы.

Монолитный подход

Тот факт, что проектируемая программа — это практическая работа в учебном заведении, позволяет использовать наиболее простые и быстрые подходы при ее

[1] См. **https://goo.su/qN6Nh**.

проектировании и реализации. При этом программа реализует один сценарий использования: поворот изображения на 90 градусов (рис. 11.1).

Декомпозируем этот сценарий на шаги, которые должны быть выполнены для поворота изображения (рис. 11.2).



Рис. 11.1. Сценарий: поворот изображения на 90 градусов

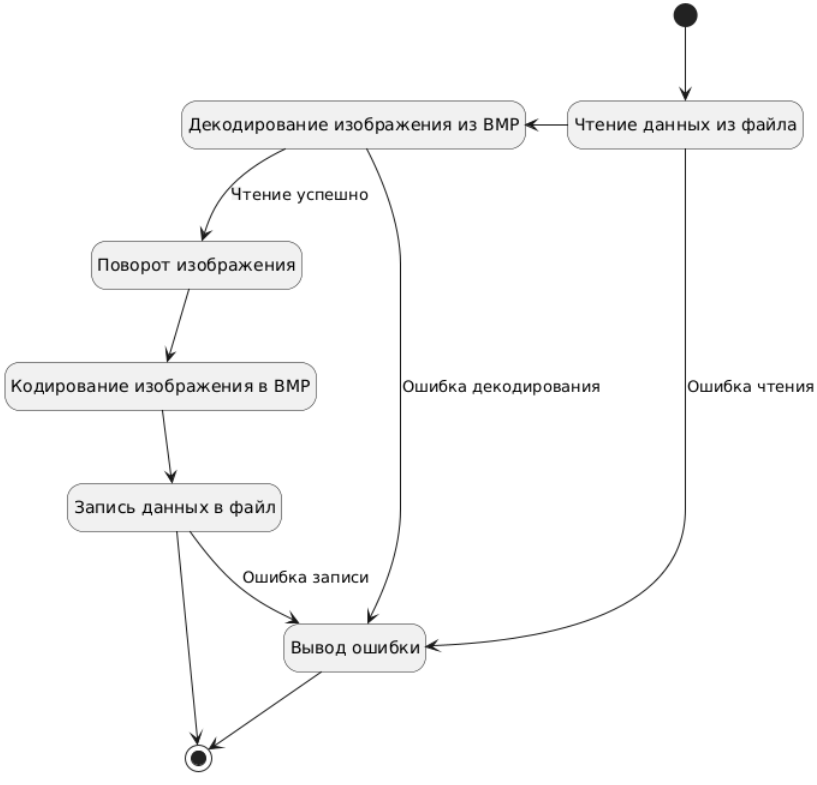


Рис. 11.2. Декомпозиция сценария на шаги

Получается пять обязательных шагов: чтение, декодирование, поворот, кодирование и запись данных. В трех шагах возможны различные ошибки — например, отсутствие файла, файл формата не BMP или недостаток места на накопителе для записи результата. Довольно простая последовательность. Представим ее в виде классов в диаграмме состояний (рис. 11.3).

Каждый шаг в этой диаграмме состояний можно реализовать отдельной функцией в программе. Сначала представим «сырое» изображение в памяти программы в виде структуры `RawImage`, состоящей из ширины, высоты и одномерного массива пик-

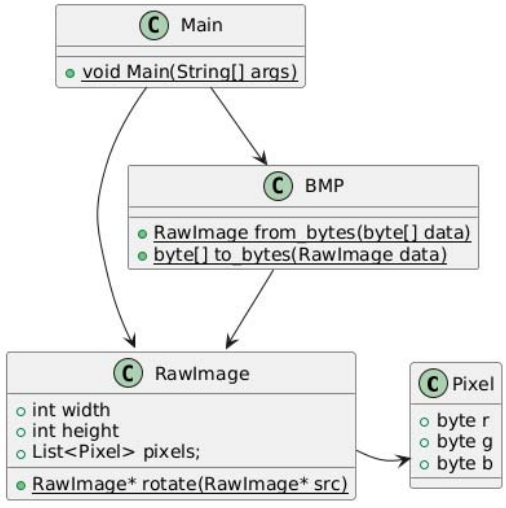


Рис. 11.3. Представление сценария в виде диаграммы состояний

селов, где каждый пиксел состоит из трех компонентов. В пространстве имен рядом со структурой `RawImage` реализуем функцию `rotate()`, которая принимает на вход одну структуру и возвращает другую — результат.

Затем добавляем пространство имен `BMP` и реализуем две функции: одну для декодирования считанных байтов формата BMP с конвертацией в структуру `RawImage`, а вторую — для конвертации `RawImage` в байты согласно формату BMP.

Завершающий штрих — точка входа в программу `main()`, которая получает первым и единственным аргументом имя файла. Эта функция выполняет все проверки и последовательно вызывает функции `from_bytes()`, `rotate()` и `to_bytes()`. С точки зрения практической работы программа завершена. Критически оценим получившееся:

- Простая структура.

  Получившаяся структура классов довольно простая и может быть реализована на большинстве языков программирования, в том числе без поддержки объектно-ориентированной парадигмы.

- Расширяемость фильтров.

  В предложенном варианте описан один фильтр — поворот изображения. Добавить другие фильтры достаточно просто: в пространстве имен `RawImage` реализовать новые функции, а в точке входа — параметр для выбора нужного фильтра.

Этого достаточно, чтобы доказать работоспособность приложения и свои навыки в рамках практической работы. Но давайте взглянем на него с точки зрения пользователя, который будет пользоваться этим редактором.

- Ограничения на формат файла.

  Хотя это является условием оригинального задания, создаваемое приложение работает только с изображениями формата BMP, потому что это наиболее про-

стой формат для самостоятельной реализации. Но в современном мире это давно не самый популярный формат изображений, и чаще используются PNG, JPEG, WEBP и HEIC.

- Отсутствие опций фильтров.

  Поворот на 90 градусов — это один из немногих фильтров, которые могут обойтись без дополнительных параметров. Для всего остального было бы неплохо уметь задавать параметры — например: радиус для размытия по Гауссу, размеры для изменения разрешения изображения и прочие.

- Один шаг за выполнение.

  В текущей реализации редактор изображений может применять только один фильтр за раз. Если пользователю нужно повернуть изображение и наложить эффект сепии, то понадобится программу вызвать дважды, при этом будут сгенерированы два изображения: промежуточное (повернутое) и итоговое (повернутое с эффектом сепии). И поскольку у нас неинтерактивный редактор, то он должен быть удобен для работы в скриптах и прочих инструментах автоматизации. В текущей версии это не соблюдается, т. к. редактор не принимает последовательность шагов и оставляет после себя промежуточные файлы.

Обработаем недостатки и сделаем точки расширения, которые помогут добавлять в программу функциональность.

Модульный подход

Рассмотрим редактирование изображения подробнее — процесс редактирования состоит из трех шагов (рис. 11.4):

1. Открыть изображение — декодировать файл в несжатое представление в памяти.
2. Применить ноль или более преобразований.
3. Сохранить изображение — кодировать представление в один из известных форматов.

Действия «Открыть изображение» и «Сохранить изображение» — это общие случаи, у которых есть частные случаи — по одному на каждый поддерживаемый формат. Для сохранения читаемости диаграммы прецедентов выбраны два формата: PNG и BMP, однако их может быть и больше.

Действие «Применить фильтр» также имеет несколько частных случаев. Но это действие отличается от открытия или сохранения изображения своей опциональностью. Иными словами, пользователь волен открыть BMP-изображение, применить ровно 0 фильтров и сохранить изображение в формате PNG. Хотя может показаться странным, что *редактор* изображений не редактирует изображение, возможность конвертирования из этого формата в другой — это полезная функциональность для пользователя, которая практически не требует времени на реализацию.

Три указанные действия можно разложить на три интерфейса: для чтения, обработки и записи (рис. 11.5). Каждый из этих интерфейсов является наследником базо-

вого интерфейса, определяющего имя аргумента командной строки, который соответствует реализации интерфейса. Например, представим команду, которая открывает BMP-изображение, поворачивает его на 45 градусов по часовой стрелке и сохраняет в PNG:

```
./editor -ibmp test.bmp -rotate clockwise 45 -opng result.png
```

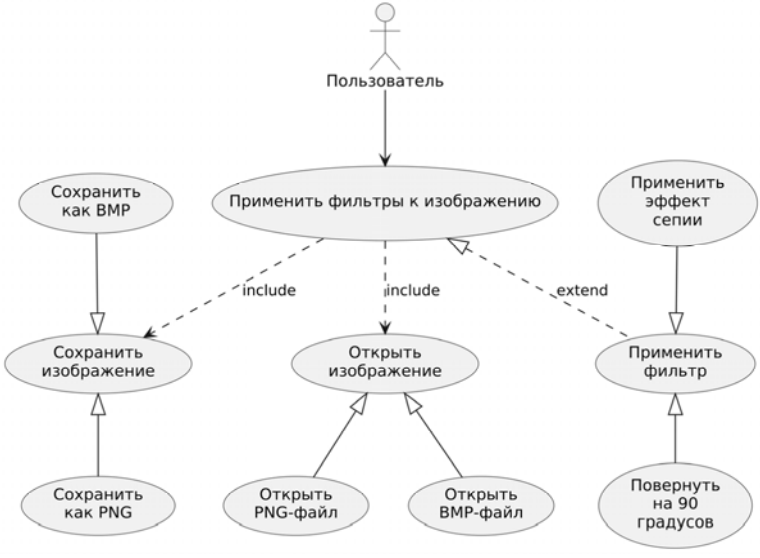


Рис. 11.4. Процесс редактирования изображения

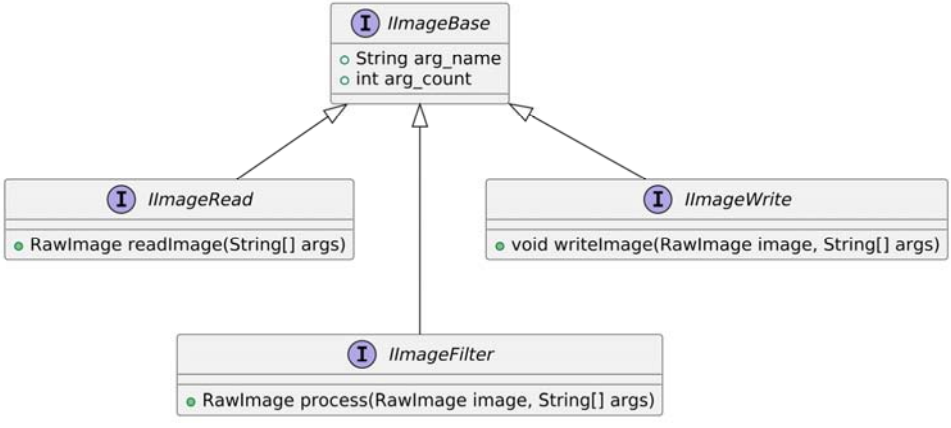


Рис. 11.5. Разложение действий на три интерфейса

Тогда для реализации интерфейса `IImageRead` в классе, который читает BMP-изображение, придется определить `arg_name=-ibmp`, а `arg_count=1`. Аналогично для поворота изображения: `arg_name=-rotate`, `arg_count=2` соответственно. Таким образом «ядро» программы сможет определить, какой класс, ответствен за этот аргу-

мент, и передать ему необходимое количество следующих аргументов. Но почему именно такой подход?

Этот подход похож на микроядерную архитектуру с тем исключением, что программа — не ядро операционной системы, а реализации классов выполняются в том же потоке. Однако основная идея соблюдается: «ядро» редактора маршрутизирует команды модулям, которые выполняют работы (рис. 11.6). При этом каждый «модуль» может разрабатываться независимо.

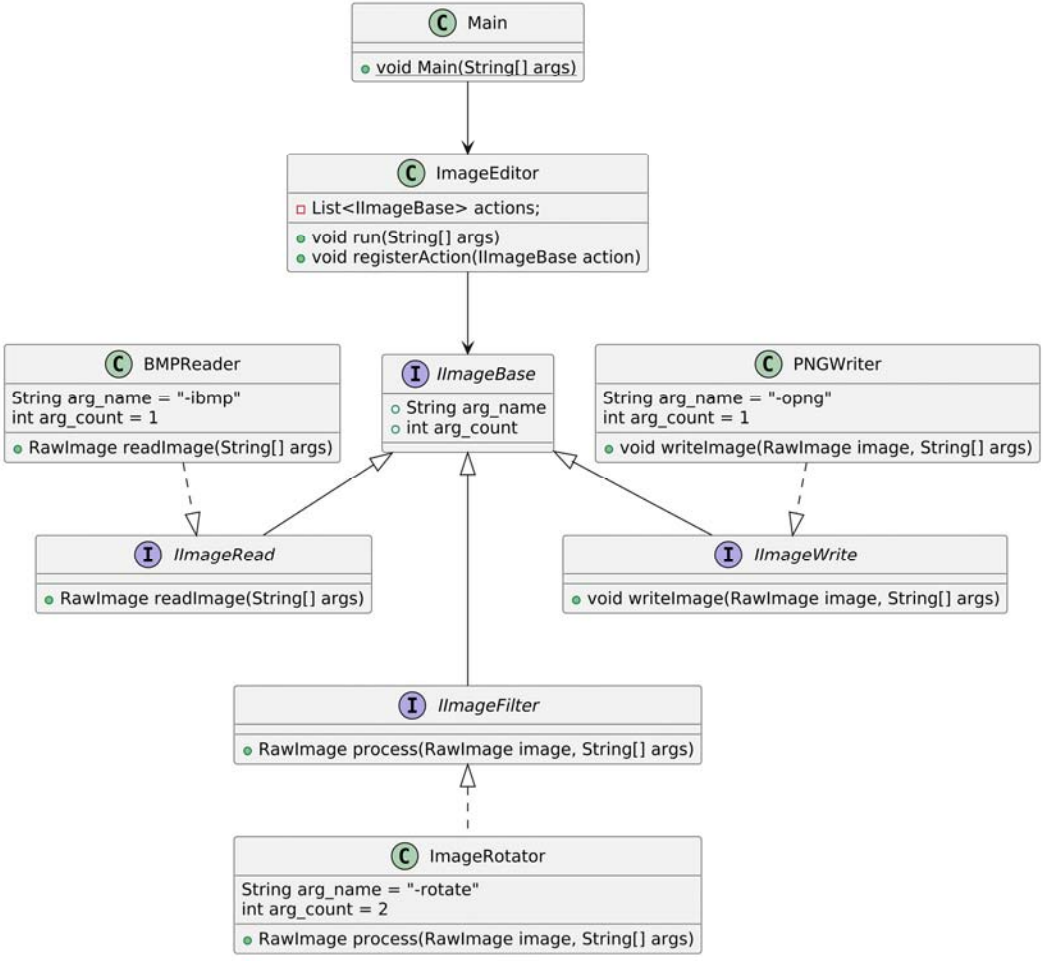


Рис. 11.6. Модульная структура приложения

«Ядро» приложения — класс `ImageEditor`, который регистрирует внутри себя обработчики различных аргументов командной строки, а метод `run()` получает все аргументы командной строки, разбирает их и передает соответствующим обработчикам. Отдельный метод регистрации необходим для определения конфликтов, когда два обработчика претендуют на одно имя ключа. Рассмотрим положительные и отрицательные стороны этого решения:

- Модульность.

  Интерфейсы для каждого типа операций позволяют разрабатывать компоненты, которые обеспечивают требуемую функциональность. Появилась потребность декодировать TIFF? Достаточно создать модуль и зарегистрировать его в ядре. В зависимости от используемого языка программирования компоненты могут быть как частью приложения, так внешними файлами — например, разделяемыми библиотеками.

- Логичность управления.

  Ядро программы разбирает аргументы командной строки последовательно и применяет обработчики последовательно. Таким образом, пользователь задает порядок действий, и эти действия будут применены в строго указанном порядке.

- Неудобство управления.

  Каждый обработчик принимает фиксированное количество аргументов, которое указывается в `arg_count`. Это подходит для ситуаций, когда у обработчика все параметры обязательные. Например, для открытия изображения требуется только путь до файла. Дело усложняется, если параметров несколько и часть из них опциональные. Например, при записи в PNG-файл можно указать качество сжатия, а также выбрать — использовать чересстрочную развертку или нет. Для пользователя это может оказаться излишним, но текущая реализация требует от него указания всех параметров обработчика. Можно отказаться от «сложных» настроек, но тогда будут недовольны те, кому они нужны. Есть несколько вариантов решения этой проблемы: использовать переменные окружения или передавать параметры в строке.

Переменные окружения — наименее времязатратный вариант, который тем не менее «обходит» созданные ядром интерфейсы взаимодействия с пользователем. И поскольку ядро не обрабатывает переменные окружения, то несколько разных модулей могут использовать одно имя переменной окружения, что может привести к неожиданному поведению.

Другое решение — передача параметров в одной строке. Например, так:

```
filename=/home/voldemar/1.png&quality=2&interlance=true
```

Обработчик будет разбирать строку по принципу «ключ=значение, разделенные знаком амперсанда», и это может сработать. Но в операционных системах семейства Linux знак `&`, равно как и знак `=`, являются разрешенными символами в именах файлов. Корректная обработка таких ситуаций может быть весьма сложной задачей. Для пользователя такая строка тоже станет представлять неудобства, особенно если разные модули, написанные разными разработчиками, будут иметь разный формат. Например, если второй разработчик применит символы `;` и `|` для разделения параметров и ключа от значения соответственно.

Критична ли проблема разнородности аргументов у разных модулей? Разумеется, ответ на этот вопрос нужно искать в обратной связи конечных пользователей, но в рамках этого объяснения можно сказать волевое: «Нет, не критична». Инструмен-

ты со сложными параметрами существуют — например, набор библиотек FFmpeg использует специальную строку для видеофильтров, которая неподготовленному пользователю может показаться сложной:

```
zscale=t=linear:npl=100,format=gbrpf32le,zscale=p=bt709,tonemap=tonemap=mobius:
desat=0,zscale=t=bt709:m=bt709:r=tv,format=yuv420p
```

Помимо этого, для чтения и записи файлов необходимо явно указывать обработчики корректного формата. С точки зрения удобства пользователя, можно сделать общую точку входа для чтения изображений, а программа выберет подходящий по расширению файл.

Выявленные замечания можно устранять в рамках поддержки разработанного программного обеспечения, в том числе административно. Например, создать документ с рекомендацией по обработке аргументов командной строки.

Кажется, что всё сделано хорошо и можно переходить к рассмотрению следующей архитектуры. Но на самом деле нет.

Внезапное требование

Если вы ранее сталкивались с редакторами изображений в командной строке, то могли уже подумать об этой недоработке в описанной архитектуре. Обратите внимание, что интерфейс `IImageFilter` предлагает функцию, которая принимает на вход одно изображение и возвращает его преобразованным. Это значит, что в предложенной архитектуре невозможны операции, которые требуют более одного изображения, — например, объединение трех изображений в одно или применение фильтра по маске.

В текущей архитектуре компонент может самостоятельно реализовать чтение другого изображения из файла, но это плохое решение. Если компонент принесет собственную реализацию чтения изображения, то получится дублирование кода, а если явно использовать компоненты, реализующие `IImageRead`, то взаимосвязи между классами станут менее очевидными.

Как и всегда, у нас есть несколько вариантов решения проблемы с внезапным требованием, которое не вписывается в ранее созданную архитектуру. Внесем в нее некоторые модификации, которые исправляют указанные ранее проблемы и добавляют новую функциональность (рис. 11.7):

- добавим в интерфейсы `IImageRead` и `IImageWrite` поле `extensions`, которое хранит список поддерживаемых расширений. Например, у класса `BMPReader` в этом массиве будет одно значение — `bmp`;
- добавим интерфейсы `IImageOpenUtils` и `IImageSaveUtils` с их реализациями для открытия и сохранения файлов. Эти классы по имени файла выбирают подходящий обработчик.

  Первые два изменения унифицируют чтение и запись файлов на основе одного аргумента;

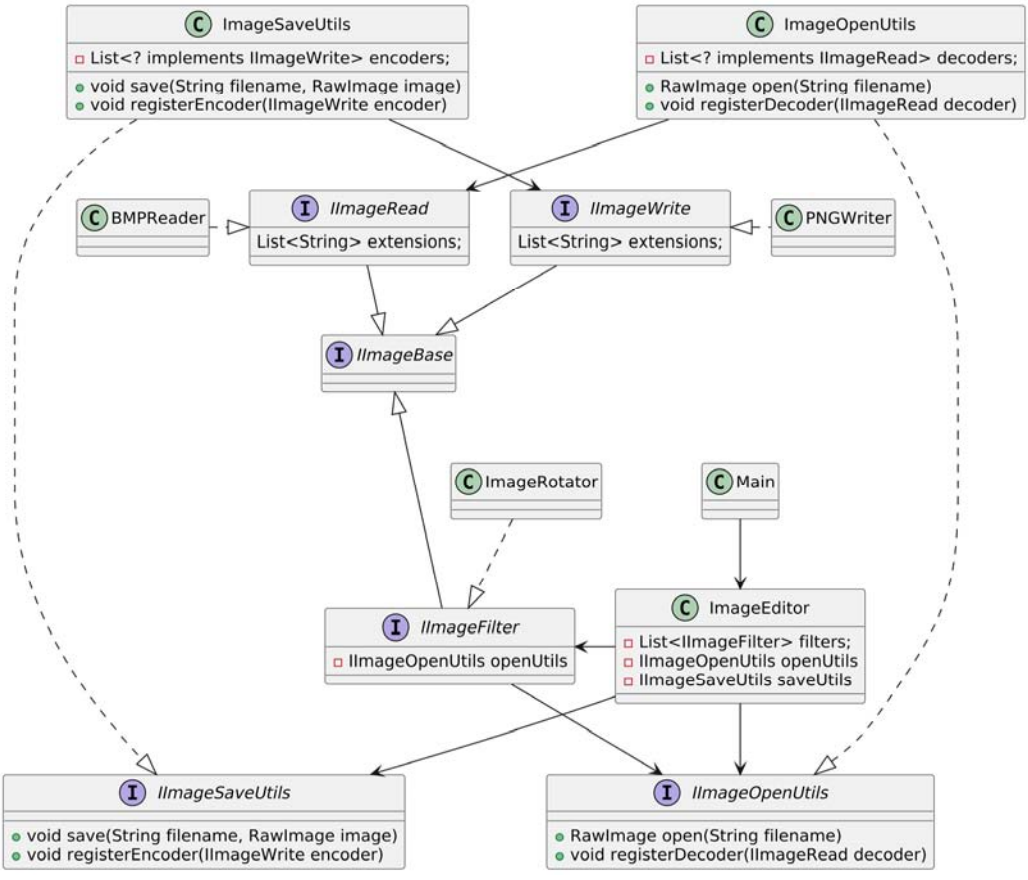


Рис. 11.7. Модифицированная структура приложения (для краткости указаны только добавленные методы)

- в интерфейс `IImageFilter` добавим поле для реализации интерфейса `IimageOpenUtils` — это позволит фильтрам использовать функции чтения изображений из файла.

Теперь основное ядро `ImageEditor` хранит только фильтры, а чтение и запись делегирует соответствующим классам.

Разберемся, какие проблемы решают эти изменения, а какие создают:

- Управление.

  Наличие модулей, которые занимаются открытием и сохранением изображений, улучшает пользовательский опыт. Модуль открытия узнает формат файла из имени файла и — если догадка оказалась неверной — перебирает все известные реализации до тех пор, пока одна из них не прочитает файл. Это может быть полезно в случае, когда файл PNG-изображения имеет имя типа test.jpg. Многие просмотрщики изображений и веб-браузеры опираются именно на содержимое файла, а не на его имя.

- Совместимость.

  Так как это модификация уже существующей программы, то важно убедиться, что не нарушается совместимость с компонентами, которые были разработаны для старой версии. В новой модификации поле `arg_name` остается актуальным только для фильтров, но не для открытия или сохранения изображений. Компоненты открытия изображений могут быть перенесены без модификаций: если массив `extensions` пуст, то вызвать метод `readImage`, и если будет выброшено исключение, этот компонент не сможет прочитать указанный файл. С компонентами для сохранения изображений так не получится — придется вносить правки во все существующие компоненты сохранения изображений.

- Спорные решения, принятые ранее.

  При первом проектировании было сделано разделение на интерфейс для чтения и интерфейс для сохранения изображений. В рамках одного формата данных реализации этих двух интерфейсов имеют множество общих структур, поэтому решение о разделении на два интерфейса представляется теперь сомнительным, поскольку можно всё сделать в одном.

Попытаемся решить и эти проблемы (рис. 11.8):

- Сначала объединим интерфейс для открытия и записи изображений в один общий — `IImageIO`. Этот интерфейс имеет четыре метода: первая пара методов — проверка возможности чтения и непосредственно чтение, а вторая пара — аналогичные методы, но для записи. Метод `isSuitableForRead()` может проводить первичный анализ файла — например, прочесть заголовок и убедиться, что текущий компонент знает, как обрабатывать файлы такого типа. Аналогично метод `isSuitableForWrite()` убеждается, что имя файла содержит расширение, соответствующее этому обработчику. Если в обработчике по каким-то причинам отсутствует реализация, например записи в этом формате, то достаточно переопределить метод-проверку и при любых входных данных возвращать логический нуль.

- Далее сделаем интерфейс и реализацию обработчика `ImageIOHandler`. Этот обработчик предоставляет нам два простых интерфейса: для открытия и сохранения изображения. На этом уровне абстракции достаточно передавать имя файла. Отдельно для операций записи используется строковый параметр `options`, который позволяет передавать параметры для реализации сохранения. Передача параметра отдельно от имени файла позволяет не беспокоиться за специальные символы в строке параметров.

- Затем модифицируем интерфейс, который отвечает за фильтры. Имя аргумента и количество последующих обязательных аргументов остаются неизменными. Зато в этом интерфейсе добавляется интерфейс `IImageIOHandler`. Таким образом, любой фильтр может использовать все доступные инструменты для чтения и записи изображений. По образу и подобию делаем интерфейс и реализацию, которая занимается обработкой команд на применение фильтров.

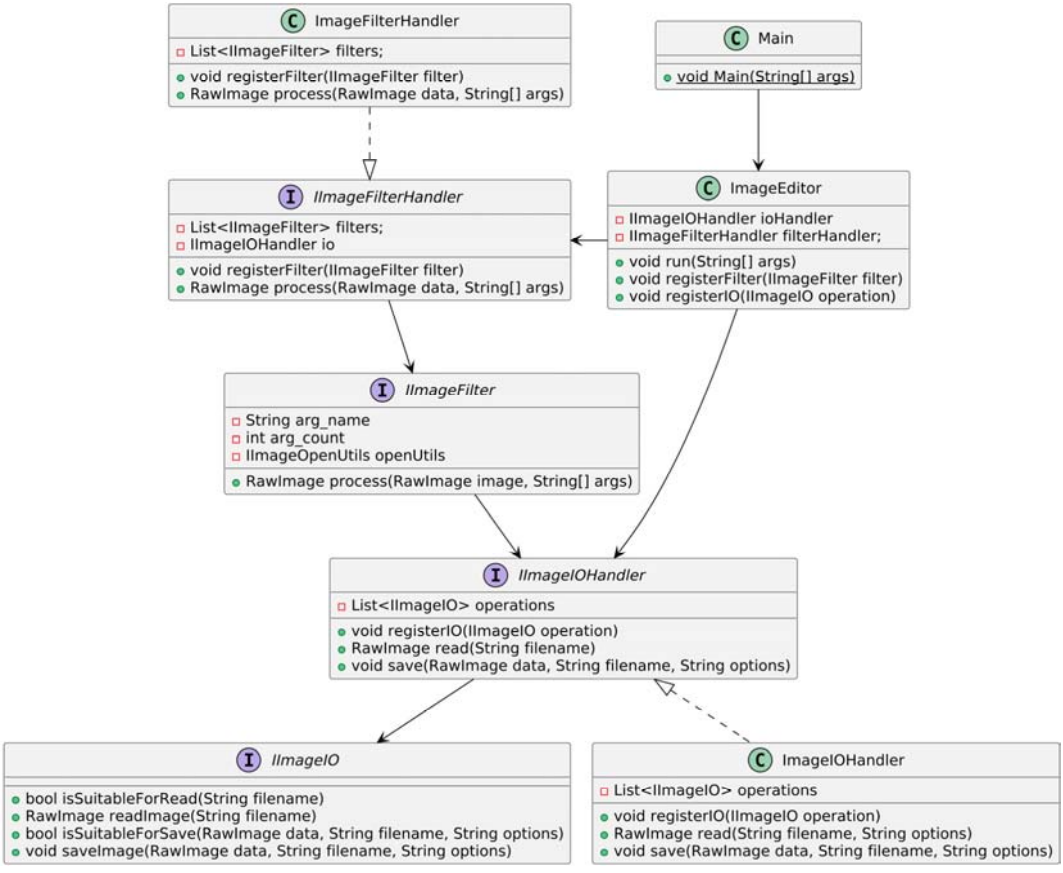


Рис. 11.8. Окончательный вариант структуры приложения

После всех примененных модификаций ядро `ImageEditor` разбирает аргументы командной строки так:

- если аргумент знаком, значит, это открытие или сохранение изображения. Извлекаем путь, если есть — дополнительные опции, после чего передаем в `IImageIOHandler`;
- если аргумент незнаком — передаем его обработку в `IImageFilterHandler`.

Вновь проведем анализ получившейся архитектуры:

- Совместимость.

  Этот пункт актуален только при обновлении программы с архитектуры, описанной ранее на текущую. Множественные изменения в интерфейсах «ломают» совместимость с существующими модулями и требуют адаптации. Это наглядный пример изменения мажорной версии в семантическом версионировании. Устранение несовместимости модулей займет дополнительное время — на обновление модулей или на создание документации по миграции между версиями архитектуры. В идеальном случае для всех внешних модулей следует определить по-

ле «версия» для однозначного разделения модулей, созданных для предыдущих версий.

- Производительность.

  При считывании изображения в память обработчик `ImageIOHandler` перебирает все доступные декодеры изображений. В лучшем случае имя файла имеет корректное расширение, и нужный декодер будет обнаружен мгновенно. В худшем случае, если файл не является изображением, обработчику придется перебрать все декодеры, и каждый из них будет считывать файл и производить вычисления. Это потенциально может замедлить процесс.

Рекомендуемые технологии

Оригинальное задание предполагает реализацию на языке программирования C, однако размышления о коммерческом продукте привели нас к диаграммам классов и объектно-ориентированному подходу.

С академической точки зрения наш редактор изображений можно реализовать на любом языке программирования — как на низкоуровневых: C, C++, Go, так и на высокоуровневых: Java, Python и JavaScript. Однако в случае академической мотивации проекта интерес заключается не только в проектировании удачной архитектуры, но и в самостоятельной технической реализации открытия и сохранения файлов в выбранные форматы.

С коммерческой же точки зрения лучше выбирать «нативные», т. е. компилируемые в машинный код языки программирования: C, C++ и Go. Эти языки больше подходят для работы на байтовом уровне, а также используют меньше абстракций и, следовательно, вносят меньше накладных расходов. При этом для работы с форматами изображений можно использовать официальные библиотеки: Libpng, Libjpeg и подобные.

Распространение

В современных операционных системах для учета установленных программ используют пакетные менеджеры. Каждый пакетный менеджер определяет свой формат, в котором распространяется программное обеспечение. Наличие пакетов для менеджеров популярных операционных систем станет для программного продукта дополнительным преимуществом. Вот наиболее популярные пакетные менеджеры с указанием, в каких ОС они используются:

- dpkg — Ubuntu, Debian;
- RPM — Fedora, CentOS;
- apk-tools — Alpine Linux (легковесный Linux, используется в Docker-образах в качестве основы);
- homebrew — сторонний менеджер пакетов для macOS;
- pacman — Arch Linux;

- pkg — FreeBSD и прочие BSD-подобные ОС;
- установщик Windows.

Сборка пакетов — это не только формирование «установщика» в соответствии с правилами, но и сборка программного обеспечения под целевое семейство операционных систем и архитектуру компьютера.

Выбор архитектур и операционных систем лежит скорее в менеджерской области ответственности, т. к. адаптация программного обеспечения для всех существующих архитектур и всех существующих семейств операционных систем будет процессом долгим и, возможно, нецелесообразным.

Заключение

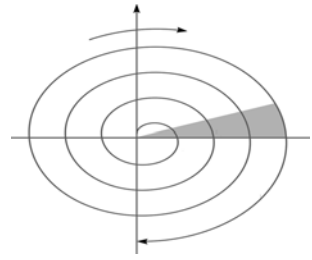
В этой главе мы рассмотрели путь проб и ошибок при проектировании приложения, появившегося в результате развития академической идеи. Хотя не все из принятых решений были удачны, финальная версия имеет достаточную гибкость для реализации различных методов редактирования изображений.

«Ретроспективно» любые требования кажутся очевидными. Но в момент принятия решений эти требования могут еще не существовать, быть неактуальными или считаться невозможными.

Всегда можно «доработать» систему для удовлетворения новых требований, но это может плохо сочетаться с примененными архитектурными стилями, что снизит качество кода.

Совместимость новых версий — это важное удобство, но иногда приходится отказываться от старых подходов.

ГЛАВА 12



Проектирование текстового блога

В этой главе проектируется веб-сервис, реализующий функциональность корпоративного блога. Начинаем с наиболее бюджетного варианта с планом масштабирования при возрастании популярности. Основное требование — выпустить минимальную версию как можно скорее.

Исходные требования

Как следует из названия, текстовый блог будет размещать текстовые заметки, доступные другим пользователям. Каждая заметка должна включать в себя следующую информацию:

- заголовок;
- время публикации;
- время последнего редактирования (опционально);
- текст, который может содержать медиа: изображения, анимации, аудио, видео и гиперссылки на ресурсы в Интернете;
- имя и изображение автора, редакторов и экспертов, принимавших участие в создании публикации (опционально);
- ключевые слова.

Определим действия и роли, которые доступны в системе (рис. 12.1).

В «максимальной комплектации» проектируемая система должна реализовывать три роли: автор, редактор и пользователь (посетитель). Автор должен иметь полный контроль над своими заметками, редактор может изменять текст заметок, к которым он допущен, а посетители должны иметь возможность читать текст, комментировать и уведомлять об ошибках и опечатках.

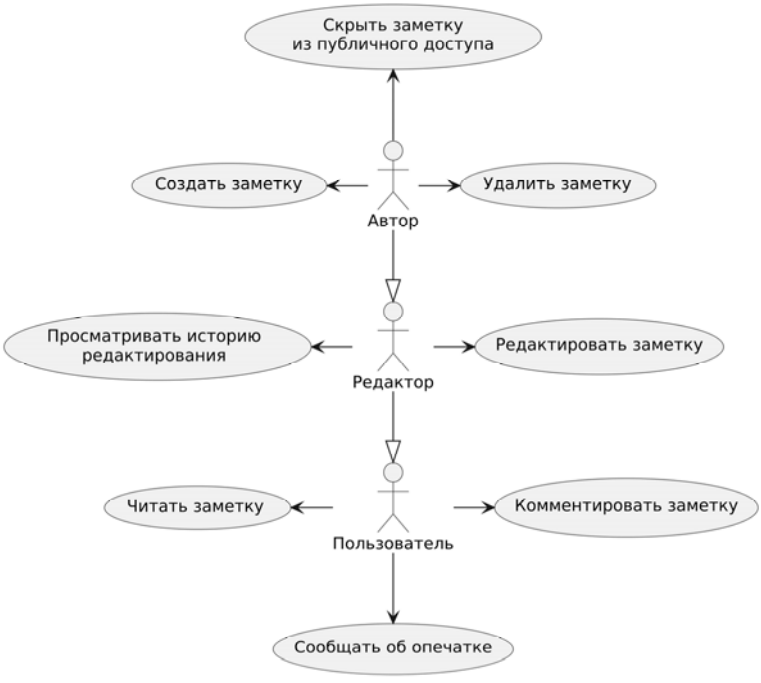


Рис. 12.1. Действия и роли, доступные в системе

Быстрый минимальный вариант

Когда в приоритете скорость разработки, следует обратиться к готовым инструментам, которые решают поставленную задачу. Существует множество сервисов, позволяющих создавать сайты и блоги без навыков программирования и верстки. Однако главный минус этих сервисов — отсутствие гибкости. Чаще всего они предлагают набор шаблонов и блоков без возможности самостоятельного расширения. Кроме того, в некоторых задачах — например, для внутренних корпоративных страниц или баз знаний, может быть неприемлемо размещение данных где-либо, кроме подконтрольной компании инфраструктуры.

Другой возможный вариант — использование системы управления контентом WordPress, мощного инструмента, позволяющего создавать собственные темы, стили оформления и расширения. Но от обращения к WordPress можно отказаться по нескольким причинам. Во-первых, это большой инструмент, не отличающийся высокой производительностью. Во-вторых, WordPress использует PHP, так что при необходимости разработки собственных расширений придется искать разработчика на PHP, если такого нет в штате. В третьих, в WordPress или его расширениях нередко находят уязвимости. Например, запись CVE-2024-10924 от ноября 2024 года в базе данных CVE[1] описывает уязвимость в одном из расширений WordPress, бла-

[1] Common Vulnerabilities and Exposures (CVE) — база данных общеизвестных уязвимостей информационной безопасности.

годаря которой злоумышленник может войти в систему под любым пользователем, в том числе и под администратором.

Дискуссию про удобства и риски использования готовых систем можно оставить открытой и в рамках этой главы отказаться от готовых систем в пользу изучения проектирования архитектур.

Для реализации минимального варианта выделим необходимую функциональность:

- создание и удаление заметок авторами;
- просмотр заметок пользователями.

Если нет времени на разработку, то часть действий можно оставить в «ручном» режиме (рис. 12.2). В нашем случае — сделать менее удобный интерфейс для создания записей в блоге. Это будет неудобством для авторов, но не затронет читателей, которых явно больше и которые мотивируют авторов к творчеству.

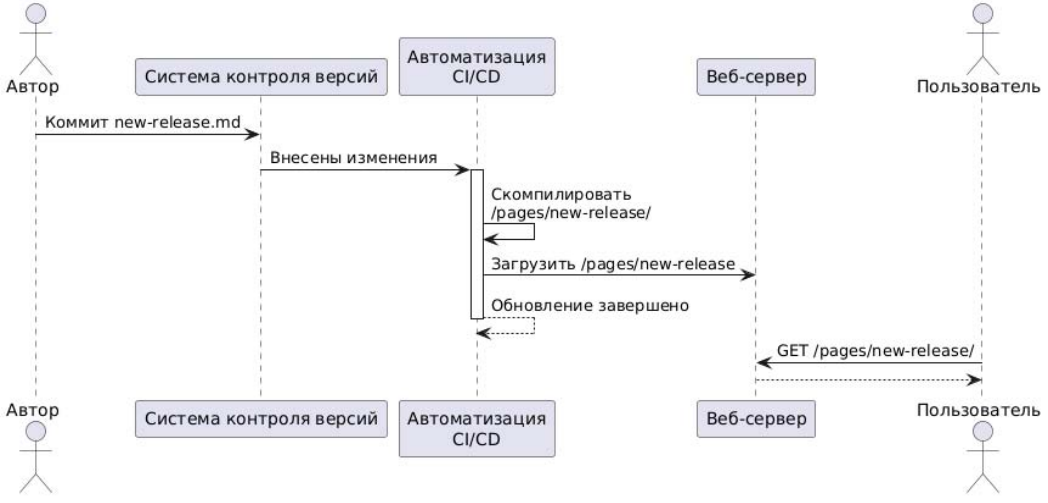


Рис. 12.2. Исходная схема работы блога

Используем для разработки следующие инструменты:

- Markdown — язык разметки для форматирования текста, использующий только печатные символы. Все материалы должны быть написаны и сохранены в нем.
- систему контроля версий — хранит тексты и позволяет определять кто, когда и какие изменения внес;
- веб-сервер — предоставляет пользователям доступ к материалам;
- систему CI/CD — при изменениях в системе контроля версией конвертирует markdown-материалы в формат, который понятен браузеру посетителя, и загружает на сервер.

Как отмечалось ранее, для уменьшения объема разработки придется пожертвовать удобствами для авторов в пользу функциональности для читателей. Возможный, но нерациональный вариант — делегировать авторам верстку HTML-страниц, которые

можно сразу отображать. Но у этого подхода сразу несколько минусов. Во-первых, не каждый автор знаком с HTML. Во-вторых, если создавать полноценные страницы, то все стили — т. е. навигация блога и оформление текста — «вшиваются» в страницу, и в случае исправлений в дизайне сайта придется редактировать все записи. Вместо этого можно использовать язык разметки Markdown, в котором разметка: заголовки, стили, ссылки — обозначаются специальными символами. В этом случае автору или редактору придется иметь дело с обычным текстом, что гораздо проще, чем HTML. Вот пример markdown-разметки:

```
# Заголовок первого уровня

## Заголовок второго уровня

Обычный текст, **полужирный** текст или __курсивный__ текст.

![Описание иллюстрации](pages/images/example.png)
```

После текстовых процессоров вроде Microsoft Word и LibreOffice Writer работа с Markdown может показаться непривычной, поскольку ее результат не виден в том же окне, но такова цена быстрого запуска. Тем не менее с момента создания в 2004 году Markdown становится все более известным — он частично поддерживается в мессенджерах и коллективных блогах, таких как Хабрахабр.

Впрочем, текстовый формат Markdown раскрывается при взаимодействии с системой контроля версий, которая эффективно работает с исходным кодом, а тексты — это тоже своеобразный исходный код. Таким образом система контроля версий решает множество задач.

Каждая фиксация изменений (коммит) в системе контроля версий показывает, кто, когда и какие изменения внес. Например, можно увидеть все исправления редактора. Использование Markdown позволяет отслеживать изменения непосредственно в тексте, что невозможно с форматами файлов текстовых процессоров.

Система контроля версий разграничивает пользователей и их права. Основная ветка: `master` или `main` — это «продуктовая» версия блога — всё, что попало в нее, становится публичным. Система контроля версий позволяет определить круг лиц, которые имеют право принимать изменения в основную ветку, следовательно, без разрешения ответственных лиц новый материал в блог не попадет.

Автоматизация действий (CI/CD) позволяет автоматически собирать новую версию сайта и загружать ее на сервер.

Хотя тексты, написанные на Markdown, не обрабатываются веб-браузерами напрямую, существуют инструменты, которые конвертируют текст на Markdown в HTML-страницы, — например, фреймворк Hugo, одна из функций которого — создание статических HTML-страниц из markdown-страниц, или его аналог Docusaurus.

Рассмотрим достоинства и недостатки этого варианта:

- Универсальность.

  Наш вариант архитектуры максимально гибок к используемым решениям. Компании, которым важно иметь полный контроль над своими системами, могут на-

строить систему контроля версий, систему автоматизации и разместить сервер, где будет расположена рассматриваемая проектируемая система, в своей инфраструктуре. А студент с очень ограниченным бюджетом сможет использовать облачные решения с бесплатным тарифным планом. При этом нет привязки к конкретным продуктам — можно применить любую систему контроля версий, любую систему автоматизации, в том числе самодельную, и любой фреймворк, конвертирующий текст в HTML, не обязательно из упомянутых здесь.

Выбор технологий настолько свободен, что вместо виртуального сервера блог в виде статических страниц может быть размещен в публичном объектном хранилище с веб-интерфейсом. Объектное хранилище тарифицируется исключительно по объему хранимых данных, а также по количеству скачиваний хранимых файлов. Это позволит оптимизировать затраты на ранних этапах жизни блога, когда посетителей еще мало.

- Популярные инструменты.

  В предлагаемом подходе используются инструменты, которые часто применяются в разных задачах в информационных технологиях, в том числе при разработке. Поэтому для компаний, связанных с IT-областью, вероятно, придется только подобрать нужный фреймворк для конвертации текста в HTML.

Проектирование архитектуры

Прежде чем приступать к проектированию архитектуры текстового блога, необходимо определиться со способом форматирования текста. Ранее был выбран язык разметки Markdown, и в целях совместимости следует оставить его, но тем не менее стоит оценить все возможные варианты:

- Markdown — уже известный формат. Вполне возможный вариант;
- HTML — обладает наибольшей функциональностью, но может быть сложен для людей, не связанных с IT-технологиями;
- BBcode — схожий с HTML упрощенный язык разметки, используемый на форумах. Предлагает ограниченные возможности, но может оказаться сложным для неспециалистов;
- ReStructuredText — схожий с Markdown язык форматирования, задействуемый преимущественно для документирования исходного кода;
- вики-разметка — применяется на различных веб-сервисах энциклопедической направленности. Общего универсального синтаксиса у нее нет, и хотя язык разметки Creole пытался унифицировать вики-разметку, стать стандартом у нее не получилось;
- создать свой язык разметки. Это даст возможность реализовать любые запросы, но потребует дополнительного времени, могут также возникнуть проблемы с совместимостью.

Из этого списка можно выделить как простой и функциональный язык разметки Markdown, так и понятный браузеру язык разметки HTML — более сложный, но

обладающий практически неограниченными возможностями по сравнению с Markdown.

Спроектируем диаграмму классов, которая представляет сущности ORM-модели и связи между ними (рис. 12.3). Для начала достаточно пользователя (`User`), заметки (`Post`), а также вспомогательных сущностей: ролей (`Role`) для определения прав пользователя и метки (`Tag`) для заметки, чтобы назначать ключевые слова. У заметки есть состояние, описанное перечислением `PostState`. На начальном этапе определены три состояния:

- не опубликована (`NOT_PUBLISHED`) — заметка существует в системе в качестве черновика. Она доступна пользователям, которые непосредственно задействованы в работе над заметкой: авторам и редакторам;
- опубликована (`PUBLISHED`) — заметка доступна всем;
- скрыта (`HIDDEN`) — заметка скрыта по какой-либо причине: желание автора, потеря актуальности и т. п.

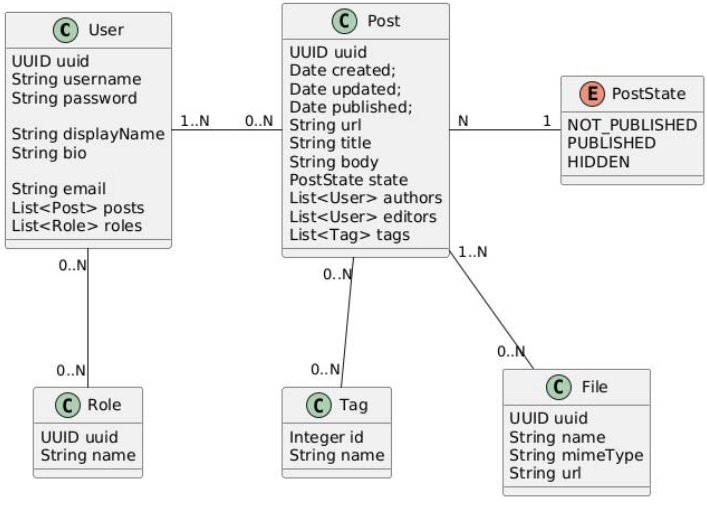


Рис. 12.3. Диаграмма классов проектируемого блога

Отдельно необходимо отметить, что в качестве уникального идентификатора и первичного ключа выбран 128-битный универсальный идентификатор (`UUID`), а не целочисленный идентификатор. Этот выбор обоснован уязвимостью сквозной нумерации к перебору. Например, если существует заметка с номером 17 и используется сквозная нумерация, то, очевидно, существуют заметки с номерами от 1 до 16. Предсказуемость идентификаторов может плохо сказаться на материалах, которые доступны публично, но только «по ссылке», — т. е. не упоминаются в навигации блога и не индексируются поисковыми системами. 128-битный идентификатор, генерирующийся случайным образом, решает эту проблему, делая перебор нецелесообразным.

Второй важный момент — хранение медиаматериалов, упоминаемых в заметках. С точки зрения минимизации усилий на разработку и стоимость инфраструктуры,

задача хранения материалов к статье переносится на плечи автора. Языки разметки позволяют ссылаться на материалы на других серверах, поэтому, куда автор загрузит — там пусть и лежит. Это очень *выгодное*, но плохое решение в долгосрочной перспективе. Авторы могут загружать материалы на разные сервисы, у которых разные подходы к хранению загруженного. Из-за этого возможна ситуация, когда заметка доступна, а все изображения уже удалены либо хостингом, либо автором.

Для решения проблемы с исчезновением файлов на сторонних сервисах необходимо реализовать загрузку файлов на «свое» хранилище и вести учет используемых в заметке файлов. Это улучшит пользовательский опыт в долгосрочной перспективе. Хранение файлов можно организовать тремя способами:

1. Хранение бинарных данных в базе данных.

   Многие СУБД имеют тип данных BLOB (Binary Large Object, бинарный большой объект) для сохранения файлов непосредственно в базе данных. При использовании этого варианта нужно быть внимательным с ORM-фреймворками, т. к. по умолчанию в модель отображаются все данные, включая BLOB-столбец.

2. Хранение файлов рядом с сервисом — самый обычный вариант, в котором файлы загружаются в специальный каталог на сервере, где развернут сервис.

3. Хранение в объектном облачном хранилище.

С точки зрения гибкости и расширяемости лучшим вариантом является третий — облачное хранилище. Хранение файлов рядом с запущенными сервисами, будь то база данных или непосредственный сервис, потребляет ресурсы сервера. Файлы могут быть очень большими и быстро исчерпать хранилище. Более того, базы данных имеют ограничение на объем файла, который может храниться в типе данных BLOB. Объектное хранилище в общем случае лишено этих недостатков.

Итак, с форматом заметок и хранением файлов определились, следующий этап — определение интерфейса взаимодействия с пользователем. Здесь есть выбор между разделением на API и фронтенд или организацией формирования HTML-страниц на стороне сервера.

Разделить приложение на бэкенд (бизнес-логику и API) и фронтенд (интерфейс пользователя) — это более долгое, но более гибкое решение (рис. 12.4). Универсальное API позволяет создавать разные представления: от странички в браузере до приложений для смартфона, планшета или компьютера. Этот подход также удобен, если разработкой интерфейса пользователя и бизнес-логикой занимаются разные специалисты. Но есть у этого решения и недостаток, связанный с производительностью веб-страниц. При загрузке страницы дополнительно загружается весь фреймворк, который занимается формированием графического интерфейса, затем происходит инициализация интерфейса пользователя, и только во время инициализации интерфейса отправляются запросы на получение данных. Минимизация и упаковка кода фронтенда позволяют ускорить загрузку, но дополнительные накладные расходы остаются.

Противоположный метод — это генерация HTML-страниц непосредственно на сервере (рис. 12.5). Называется такой подход server-side rendering — «отрисовка на серверной стороне». В этом способе веб-браузер делает запрос к веб-серверу, а веб-

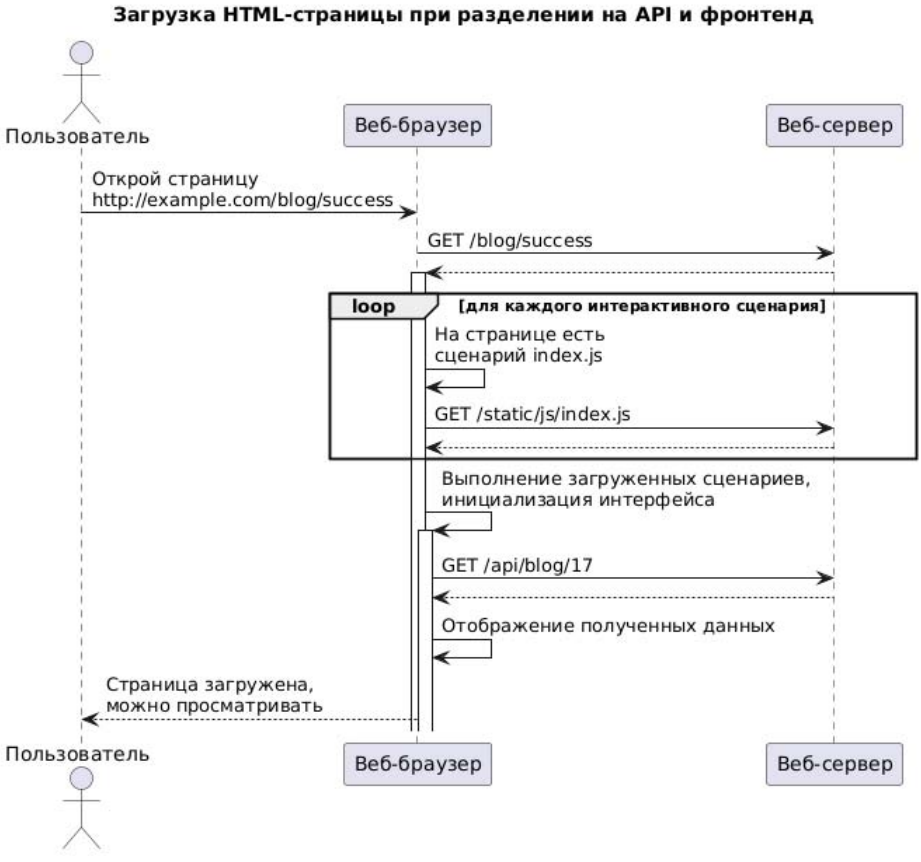


Рис. 12.4. Разделение приложения на бэкенд и фронтенд

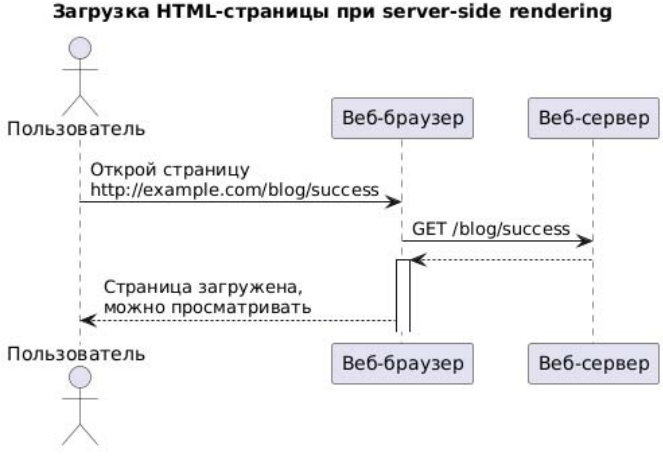


Рис. 12.5. Генерация HTML-страниц непосредственно на сервере

сервер сразу генерирует ответ, который понятен браузеру. Веб-сервер, безусловно, тоже выполняет работу по формированию представления страницы, но сервер, который этим занимается, скорее всего, использует более производительное оборудование и благодаря кешированию сделает всё быстрее. К тому же в ответе передается только то, что необходимо для отображения запрошенной страницы. Отрисовка на стороне сервера хорошо пригодна для приложений с минимальной интерактивностью внутри страницы. Текстовый блог идеально подходит под описание: каждая заметка — это отдельная страница, которая не предлагает вводить информацию, нажимать на кнопки, подтверждать свой выбор и пр. Поэтому остановимся именно на этом варианте.

Далее можно выбрать архитектурный подход, который объединит все части в одно приложение. И выбор здесь минимален — приложение должно быть монолитным. В системе всего две основные сущности: пользователь и заметка. Разделять это на микросервисы нецелесообразно — у каждой заметки есть автор, информация о котором должна показываться непосредственно в начале заметки. Аналогично информация об авторе может включать в себя список опубликованных заметок. Разделение на сервис пользователей и сервис заметок исключительно усложнит систему.

Диаграмма развертывания блога тривиальна: на одном сервере размещается приложение, приложение взаимодействует с базой данных и может записывать файлы в облачное хранилище (рис. 12.6). Пользователь взаимодействует с приложением и опционально загружает медиа из объектного хранилища, если они упомянуты в заметке.

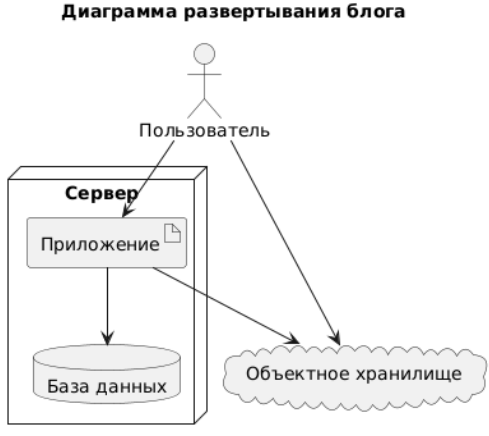


Рис. 12.6. Диаграмма развертывания блога

Для полноты описания архитектуры перечислим эндпоинты, которые должны поддерживаться для корректного выполнения функциональных требований:

- доступ к заметкам:
  - `GET /` — главная страница;
  - `GET /post/{uuid}` — страница, соответствующая заметке;

- доступ к карточкам пользователей:
  - `GET /user/{uuid}` — карточка пользователя;
  - `PATCH /user/{uuid}` — редактировать пользователя;
- управление заметками:
  - `GET /{uuid}/edit` — страница редактирования указанной заметки;
  - `PATCH /{uuid}` — применяет отправленные изменения;
  - `POST /{uuid}/report` — сообщить об опечатке в указанной заметке;
  - `GET /new` — страница создания новой заметки;
  - `POST /new` — создать новую заметку;
- управление файлами:
  - `GET /file/{uuid}` — получить метаинформацию о файле;
  - `POST /file` — загрузить новый файл;
  - `PATCH /file/{uuid}` — обновить существующий файл;
  - `DELETE /file/{uuid}` — удалить файл;
- поиск:
  - `GET /search` — страница поиска по заданным параметрам;
- аутентификация:
  - `GET /login` — страница входа;
  - `POST /login` — аутентификация по заданным параметрам;
  - `GET /register` — страница регистрации;
  - `POST /register` — регистрация.

Для высокоуровневого описания архитектуры этого достаточно.

Рекомендуемые технологии

Для реализации веб-сервисов хорошо подходят высокоуровневые языки программирования — такие как Python, Java, JavaScript (TypeScript). Большинство этих языков уже имеют наборы библиотек и фреймворков, решающих нужные задачи:

- веб-сервер: FastAPI, Flask, Spring, Node.js;
- отрисовка на серверной стороне: Jinja2, JavaServer Pages, Next.js;
- ORM: SQLAlchemy, Hibernate, Sequelize;
- клиент для объектного хранилища: HTTP-библиотека или клиент для протокола, поддерживаемого выбранным объектным хранилищем;
- конвертер Markdown в HTML.

Диаграмма классов, показанная на рис. 12.3, построена исходя из предложения использовать реляционную СУБД — например, PostgreSQL. Этот проект использует

базу данных для обычного хранения данных, поэтому большинство реализаций будут одинаково хороши. Тем не менее допустимо использовать и документоориентированную базу данных — например, MongoDB.

Безопасность

Проектирование системы на «бумаге» учитывает потребности, которым система должна удовлетворять, но применение конкретных технологий и конкретных реализаций создает риски неверного или небезопасного использования инструментов.

Ранее рекомендовалось задействовать реляционную базу данных, а еще чуть ранее — возможность сохранения текста в Markdown и в HTML. Каждая из этих рекомендаций, обоснованная в момент проектирования, таит в себе угрозу информационной безопасности.

При работе с реляционными базами данных надо быть максимально осторожным и не допускать данные, которые ввел пользователь, непосредственно к запросу. Вот пример уязвимости на языке Python:

```
post_uuid = "17"
engine.execute(f"select * from posts where uuid = {post_uuid}")
```

Здесь имя таблицы используется из переменной, при этом никаких ограничений на содержимое переменной `table` нет. В случаях, когда значение переменной приходит напрямую от пользователя, может случиться непоправимое. Например, злоумышленник подставит корректное имя таблицы, затем поставит точку с запятой и напишет новое выражение:

```
post_uuid = "posts; drop table users"
engine.execute(f"select * from posts where uuid = {post_uuid}")
```

Язык запросов SQL позволяет указывать в одной строке несколько выражений, разделенных специальным символом — точкой с запятой. В результате СУБД выполнит обе команды:

```
select * from posts where uuid = 17 ;
drop table users
```

При этом вторая команда уничтожит таблицу с пользовательскими данными, что, скорее всего, приведет к полной неработоспособности приложения. Использование ORM-фреймворков защищает от этой ошибки. Однако если почему-то использование ORM-фреймворка нецелесообразно, то для защиты от таких ошибок следует использовать «подготовленные выражения» (prepared statements), которые есть в каждой реляционной СУБД.

Вторая возможная проблема — хранение текста в «чистом» HTML. Хотя к созданию текстов допускаются только авторизованные пользователи, тем не менее автор с плохими намерениями может вставить в текст тег `<script>`, и при появлении такого тега в HTML-странице браузер исполнит указанный в нем JS-код. Запуск произвольного кода в браузере пользователя — это серьезная уязвимость. Универсального решения нет. Можно либо отказаться от поддержки HTML-текстов и ограни-

читься функциональностью, которая есть в Markdown, либо составить список запрещенных HTML-тегов и удалять эти теги при сохранении текста. Первый способ ограничивает возможности, а второй не является надежным на 100%, т. к. злоумышленник может использовать разные теги, в том числе и появившиеся какие-либо новые, а также применять различные механики, позволяющие обойти удаление вредоносного тега.

Аутентификация

Учитывая, что проектируется корпоративный сервис, следует использовать корпоративные средства для управления пользователями и аутентификации. Если в организации применяется технология единого входа (Single Sign-On, SSO) — например, OpenID Connect, Keycloak или OAuth, то надо задействовать именно ее. Веб-фреймворки нередко поддерживают такой вид аутентификации по умолчанию, что уменьшает сложность разработки.

Если SSO не используется, но есть система управления пользователями (служба каталогов) — например, LDAP, то следует использовать ее. В отличие от SSO, пользователь здесь должен передать логин и пароль в разрабатываемую систему, чтобы она могла проверить корректность введенных данных и получить информацию о пользователе. В этом случае в системе придется реализовать хранение пользователей и их прав доступа, а также создание сессионных токенов для доступа к страницам. Security-модули популярных веб-фреймворков в разной степени могут помочь при решении этих задач.

В наиболее сложном случае, когда нет никаких средств корпоративной аутентификации, необходимо реализовать все описанное ранее и дополнительно регистрацию и, вероятно, верификацию новых пользователей. Диаграмма классов, представленная на рис. 12.3, рассчитана на вариант с регистрацией, но без подтверждения. Для сотрудников компании выделяются почтовые ящики на корпоративном домене — например, vlad@example.com. В этом случае можно ввести белый список доменов, для которых разрешена регистрация.

> ***Примечание***
>
> Хотя в диаграмме классов, представленной на рис. 12.3, есть поле `password`, хранить пароли в информационных системах — плохая практика. В это поле необходимо сохранить хеш пароля, который указал пользователь. Хеш — функция необратимая, и в случае, если пользователь ввел верный пароль, сохраненный кеш и вычисленный хеш при аутентификации сойдутся. Для хеширования пароля не следует использовать MD5 и прочие устаревшие алгоритмы.

Поиск

У каждой заметки есть автор, ключевые слова (теги) и время публикации. Поиск по этим критериям делается средствами СУБД. Поиск по словам (полнотекстовый поиск) также может быть реализован средствами СУБД, PostgreSQL имеет для этого соответствующие функции.

Если блог публичный, то поисковые системы проиндексируют содержимое и, помимо встроенного поиска, появится альтернатива в виде виджета внешних поисковых систем.

Масштабирование

- Горизонтальное масштабирование.

  Спроектированная система совместима с горизонтальным масштабированием. Использование REST-подобного интерфейса позволяет развернуть множество экземпляров приложений и настроить балансировку между ними. Клиент при обращении передает полный набор информации, необходимый для выполнения запроса, поэтому нет необходимости в «закреплении» клиента за определенным экземпляром приложения. Стратегия балансирования Least Connections кажется хорошим решением, т. к. «закрепление» при этом не нужно, а сервис возвращает HTML-страницы, которые могут быть разного объема.

- Оптимизация подключений к базе данных.

  Увеличение количества пользователей требует увеличения количества экземпляров приложения, а это, в свою очередь, увеличивает количество выполняемых подключений к базе данных. В зависимости от используемых фреймворков обработка каждого запроса от клиента может приводить к созданию нового подключения к базе данных, а это операция медленная, т. к. база данных создает новый процесс для обработки нового соединения. Для решения этой проблемы используют «пулер» (pooler) — программу, которая поддерживает открытые соединения с базой данных и «сдает их в аренду» приложениям, которые подключаются к пулеру. Это позволяет переиспользовать соединения, а приложение получает данные быстрее. Наиболее известным пулером является pg_bouncer, работающий с PostgreSQL, однако есть реализации и для других СУБД.

- Сеть доставки контента.

  Текстовый блог — это статический контент, который не зависит от пользователя и его состояния, что позволяет для всех публичных материалов подключить сеть доставки контента — CDN. Для работы с CDN потребуются доработки — при редактировании заметки необходимо отправить запрос к CDN-провайдеру, чтобы URL, соответствующий заметке, был инвалидирован и кешировался заново. Автоматизация этого процесса была бы хорошим удобством, но эти действия могут быть выполнены и вручную.

Локализация контента

В процессе развития спроектированной системы может возникнуть вероятность появления требования, чтобы размещаемый контент был бы доступен на нескольких языках. В текущей реализации поддержка нескольких языков для одной заметки не предусмотрена. Однако есть несколько «обходных» решений:

- Публикация переводов отдельно.

  Самый простой, но не самый красивый вариант — это создавать переводы как отдельные заметки в блоге. Недостатки этого способа — у оригинальной заметки и перевода не будет никакой явной связи, и ссылку на заметку на другом языке придется указывать непосредственно в тексте заметки. А если статья уже переведена на 42 языка, то перевод на 43-й с обновлением всех предыдущих ее версий потребует массу лишнего времени.

- Машинный перевод.

  В некоторых случаях допустимо использовать автоматический переводчик в виде виджета на странице, который по нажатию кнопки переведет всё содержимое страницы на выбранный язык. Качество автоматического перевода хуже, чем ручного, но это лучше, чем ничего.

- Самостоятельный сервис.

  В этом случае для каждого языка разворачивается самостоятельный сервис со своей базой данных (рис. 12.7). Этот подход позволяет «прозрачно» выполнять переводы статей, используя идентичные идентификаторы. Например, если оригинал доступен на русском языке по адресу **russia.example.com/blog/success-story**, то перевод на китайский размещается на другом домене, но по тому же пути на сервере: **china.example.com/blog/success-story**. Вместе с тем такие национальные экземпляры приложения могут использовать локальный провайдер объектного хранилища и не использовать CDN, если в этом нет необходимости. Тогда при обращении к «главному» сервису по адресу `example.com` с помощью GeoDNS подбирается ближайший сервис и выполняется переадресация на национальный.

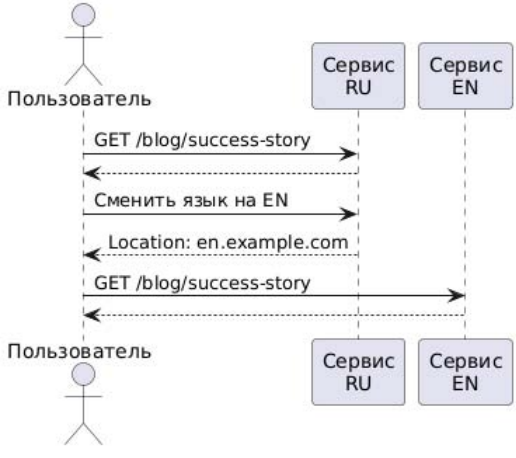


Рис. 12.7. Самостоятельный сервис-переводчик

Логирование и мониторинг

Проектируемый сервис не содержит сложной бизнес-логики, поэтому надо просто сохранять логи доступа к веб-серверу и реализовать аудит-логирование действий зарегистрированных пользователей.

В мониторинге достаточно настроить проверку доступности по HTTP для каждого экземпляра приложения и проверку состояния базы данных.

Отслеживание популярности материалов

Любой блог — это не только создание контента, но и подсчет маркетинговых метрик — например, просмотров. Впрочем, в спроектированном блоге не подразумевается обязательного аккаунта для читателей, и подсчет уникальных просмотров не будет точным.

- Сессии.

  Несмотря на отсутствие аккаунтов читателей, веб-сервер может присваивать каждому клиенту уникальный идентификатор сессии и считывать просмотры как запросы к странице заметки. При этом необходимо записывать время последнего запроса к каждой странице от каждой сессии, что позволит засчитывать уникальный просмотр, например, раз в день. Этот метод, однако, подвержен уязвимости: клиент может не принимать или не отправлять идентификатор сессии, и при каждом запросе сервер станет выдавать новый идентификатор сессии и засчитывать новый уникальный просмотр. Возможный вариант решения этой проблемы — не учитывать просмотры, если идентификатор сессии не определен.

- Информация о клиенте.

  При запросах к веб-серверам все веб-браузеры и прочие веб-клиенты оставляют информацию о себе: IP-адрес и параметр `User-Agent`, который описывает клиента. Уникальный просмотр можно определить по кортежу из этих значений. Аналогично предыдущему варианту этот способ тоже подвержен фальсификации. Параметр `User-Agent` — это произвольная строка, которую задает клиент. Даже в браузерах ее можно менять по желанию пользователя. Так что и это ненадежный вариант определения уникальности посетителя.

  Более того, определение пользователя через сессии или информацию о клиенте не будет работать, если весь сервис «спрятан» за CDN: клиент обращается к кешированной копии, а «просмотры» — это запросы кеширующих серверов к сервису.

- Отслеживание на фронтенде.

  Интерактивные сценарии на фронтенде (JavaScript) имеют в своем распоряжении больше данных, что позволяет более точно определять клиента.

  Также существуют аналитические продукты, которые могут отслеживать не только факт посещения, но и продолжительность и последовательность посеще-

ния сайта, — например, Google Analytics и «Яндекс.Метрика». Эти инструменты дают много информации и работают совместно с рекламной площадкой. Возможно, они собирают избыточно много информации, поэтому многие пользователи используют блокировщики рекламы, которые в том числе борются с трекерами.

Выбор способа реализации лежит вне плоскости проектирования архитектуры. Если осуществляется рекламное продвижение разработанной системы, то разумно использовать средства аналитики, которые предоставляет рекламная площадка. Если не используется CDN и есть необходимость выводить количество просмотров пользователям, то стоит реализовать свой счетчик просмотров. Также ничего не мешает использовать оба варианта одновременно.

Рекомендательная система

Рекомендательная система показывает пользователю похожие интересные материалы и задерживает его в блоге на большее время. Но, учитывая, что в проектируемой системе читатели анонимные, реализовывать персональные рекомендации смысла нет. Рекомендации можно создавать на основе ключевых слов заметки, времени публикации (более новые имеют больший вес) и просмотров. Также можно ограничиться блоками «Больше всего просмотров» и «Самое свежее».

Соблюдение локальных норм

Рост блога неизбежно привлечет внимание не только читателей со всего света, но и, возможно, нарушит юридические кондиции определенных регионов. В случае с текстовыми заметками можно предусмотреть два варианта развития событий:

- Запрос на удаление.

  Например, если заметка нарушает авторские права, правообладатель может потребовать удаления заметки. Эта ситуация корректно обрабатывается текущей системой — заметка может быть скрыта или полностью удалена.

- Запрос на скрытие для пользователей определенного региона.

  Это могут быть санкционные требования или требования цензуры. В этом случае всё еще возможно полное удаление заметки для всех пользователей, но с точки зрения маркетинговых показателей это неэффективно. Лучше добавить в сервис базу GeoIP для определения региона пользователя, отправившего запрос. Многие CDN-провайдеры поддерживают эту функцию по умолчанию.

Резервное копирование

Наибольшую ценность в этом проекте представляет содержимое базы данных и материалы, хранящиеся в объектном хранилище. Содержимое базы данных можно копировать каждую ночь на объектное хранилище другого провайдера инфраструк-

туры. Ввиду малого объема можно использовать полное резервное копирование, а не инкрементальное.

Заключение

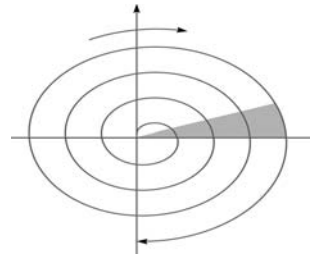
В этой главе мы рассмотрели архитектуру корпоративного текстового блога, который может быть легко масштабирован.

Первую версию текстового блога можно сделать феноменально быстро с использованием инструментов, которые обычно применяются для разработки программного обеспечения.

Разработка приложения, которое запускается вдали от пользователя, требует особой внимательности к безопасности, поскольку приложение обслуживает множество посетителей.

Облачные технологии и сети доставки контента упрощают масштабирование из «локального» блога во всемирный.

ГЛАВА 13



Проектирование сервиса конвертации файлов

В этой главе проектируется сервис онлайн-конвертации файлов: пользователь загружает файл, выбирает желаемый формат и получает результат.

Исходные требования

Определимся с функциональными требованиями — сервис онлайн-конвертации данных должен:

- позволять загружать файлы, выбирать, в какой формат конвертировать загруженный файл, и скачивать конвертированный результат;
- хранить конвертированные файлы не менее 14 дней;
- предоставлять два уровня взаимодействия: бесплатный с ограничениями и платный без ограничений;
- использовать сторонние инструменты для выполнения конвертации;
- быть масштабируемым горизонтально;
- поддерживать обновления без остановки сервиса.

Есть к нему также и нефункциональные требования:

- сервис может быть развернут в географически разных точках, при этом в случае отсутствия связи между этими точками должен работать в полном объеме;
- должен сохранять работоспособность при аварийном завершении некоторых компонентов. Коэффициент готовности — 99,99%;
- должен поддерживать несколько клиентов: веб-браузер и мобильные/компьютерные приложения.

Проектирование архитектуры

Для такой задачи отлично подходит архитектура, близкая к микросервисной, только не каждый сервис в ней представлен веб-сервером (рис. 13.1). Проектируемая

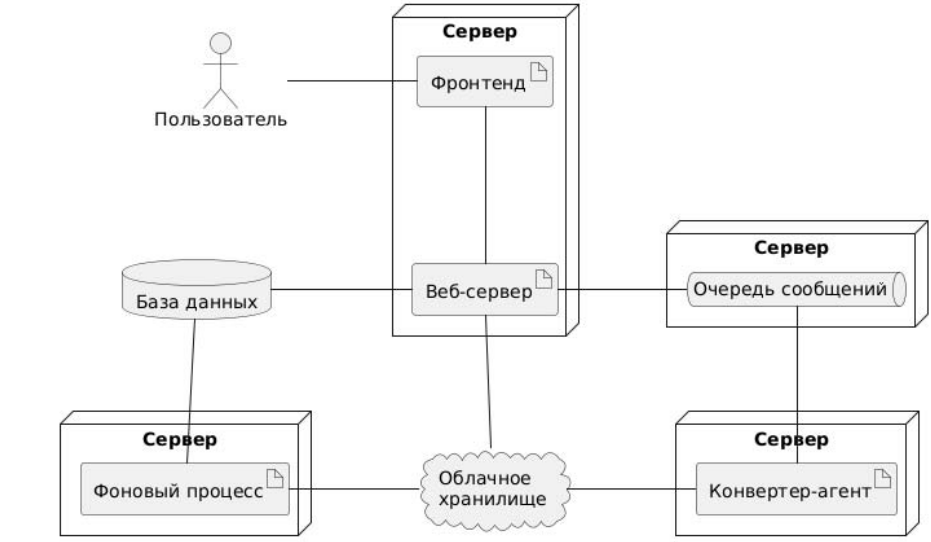


Рис. 13.1. Архитектура приложения

система логически разделяется на две части: веб-часть для взаимодействия с пользователем и часть для выполнения работы, включающие:

- конвертер-агент — его задача получать файл, получать параметры конвертации и выполнять заданную конвертацию. При этом результат конвертации должен быть доступен пользователю;
- веб-сервер — обеспечивает взаимодействие с пользователем, принимает файлы от пользователя, предоставляет настройки для конвертации файлов и позволяет следить за состоянием конвертации;
- фоновый процесс — удаляет файлы, которые хранятся дольше положенного срока, и деактивирует платные функции пользователей по истечении срока подписки;
- облачное хранилище — объектное хранилище, в которое помещаются пользовательские файлы.

Конвертация файла — это многоступенчатый процесс (рис. 13.2):

1. Сначала пользователь отправляет файл на бэкенд с запросом — допустим, он хочет конвертировать файл cake.png в формат JPEG.
2. Бэкенд регистрирует задание в базе данных и доставляет пользователю идентификатор задания.
3. Затем происходит загрузка файла от пользователя в объектное хранилище системы.
4. Когда файл загружен, запись об этом сохраняется в базе данных, а в очередь сообщений помещается сообщение о задании, которое необходимо выполнить.
5. Агент, который может выполнить это задание, забирает его из очереди сообщений и приступает к выполнению.

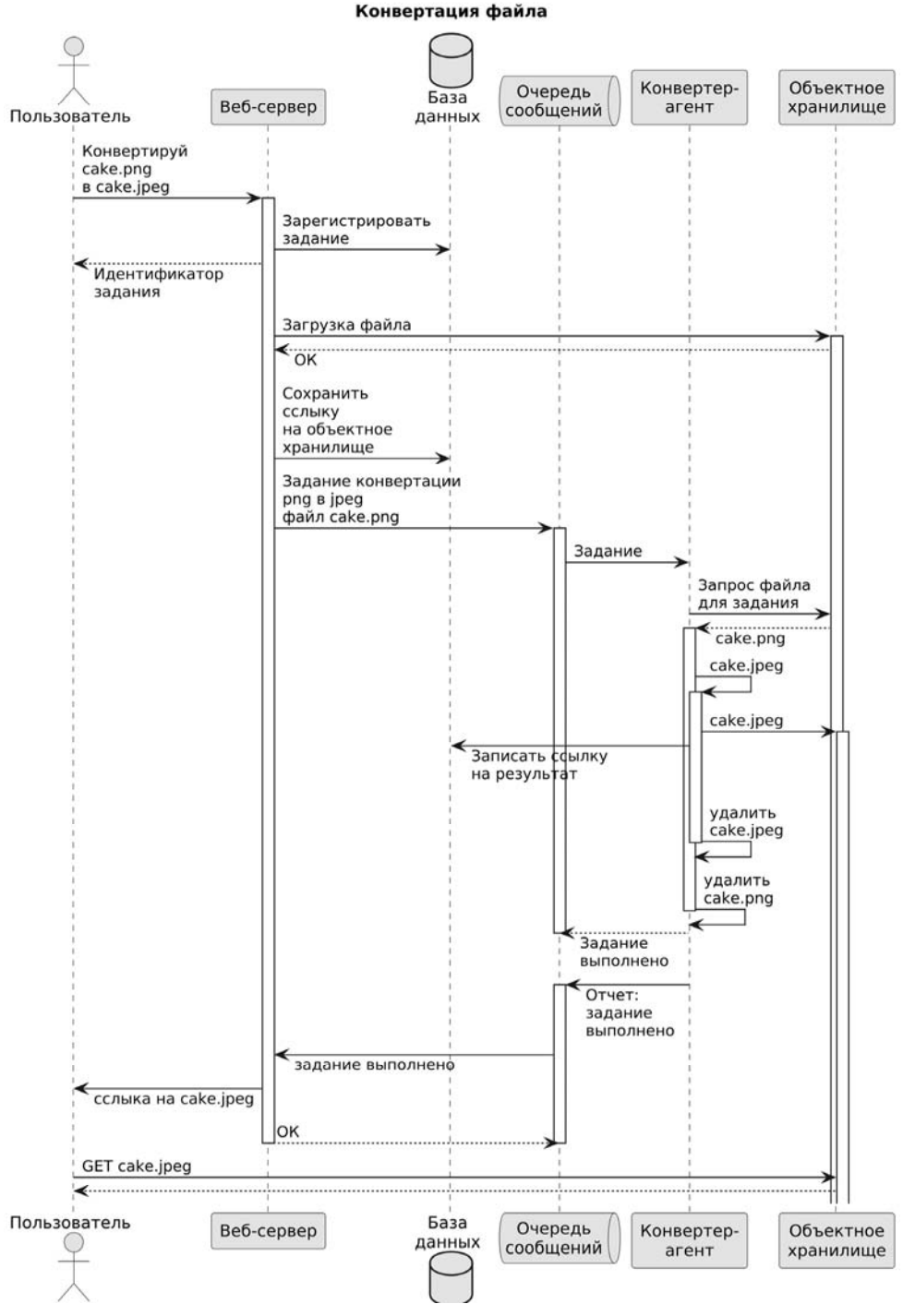


Рис. 13.2. Процесс конвертации файла

Собственно конвертация выполняется в несколько простых шагов:

1. В сообщении (задании) передается ссылка на объектное хранилище.
2. По этой ссылке скачивается файл.
3. Файл конвертируется в указанный формат.
4. Сконвертированный файл загружается в объектное хранилище с тем же именем, но с новым расширением.
5. Удаляются локальные файлы.
6. В очередь сообщений отправляется сообщение, что задание выполнено.
7. В очередь сообщений отправляется уведомление для бэкенда (веб-сервера), что задание выполнено.

В завершение бэкенд получает уведомление, что задание выполнено, и показывает результат пользователю. У пользователя есть некоторое время, чтобы скачать результат, — в функциональных требованиях на это отводится минимум 14 дней.

Благодаря очереди сообщений и разделению системы на компоненты система способна переживать аварийное завершение экземпляров различных своих компонентов. Но отказ *всех* экземпляров любого компонента приведет к простою системы. При этом компоненты системы будут вести себя следующим образом:

- Веб-сервер.

  Если веб-сервер завершится до регистрации задания, то фронтенд выведет ошибку и пользователю придется еще раз создать задание.

  Если же веб-сервер завершится после регистрации задания — например, во время загрузки файла, то можно попытаться установить соединение с другим экземпляром веб-сервера и продолжить — ведь уникальный идентификатор задания уже известен.

- База данных, очередь сообщений и объектное хранилище.

  Предполагается, что база данных и очередь сообщений зарезервированы собственными средствами и в случае аварийного завершения мастера будет выбран новый мастер, который продолжит работу.

- Конвертер-агент.

  Если конвертер-агент аварийно завершится во время своей работы, то в очередь сообщений придет отказ (`reject`), и очередь сообщений отдаст это задание другому агенту.

- Фоновый процесс.

  Отказ фонового процесса никак не повлияет на выполнение заданий в краткосрочной перспективе. В долгосрочной перспективе объектное хранилище может быть переполнено.

Рассмотрим каждый компонент по отдельности.

Проектирование веб-сервера

Веб-сервер (бэкенд) предоставляет интерфейс, который используется фронтендом и любыми мобильными/компьютерными приложениями. Для классических приложений необходимо сделать REST-подобное API:

- `GET /format/{format}` — получить информацию, в какие форматы может быть конвертирован указанный формат, и получить список настроек для каждого формата;
- `POST /file` — создать задание на конвертацию;
- `POST /file/{uuid}` — загрузить файл в задание на конвертацию;
- `GET /file/{uuid}` — узнать состояние задания.

Потребуются также интерфейсы для работы с пользователями:

- `GET /user` — получить информацию о текущем пользователе;
- `POST /login` — аутентификация;
- `POST /register` — регистрация.

> ***Важный момент относительно загрузки файлов...***
>
> Имя файла — это пользовательские данные, которые могут быть любыми, в том числе пересекающимися. Чтобы избежать пересечений, веб-сервер загружает файл в объектное хранилище с именем, которое состоит исключительно из уникального *идентификатора задания* и оригинального расширения файла. При этом оригинальное имя файла сохраняется в базе данных и в атрибутах файла в объектном хранилище. Такое именование не только решает проблему пересечения имен, но и однозначно позволяет определить принадлежность файла.

Для интерактивного веб-интерфейса нужен отдельный интерфейс, который устанавливает WebSocket-соединение. Через WebSocket-соединение можно получать информацию о выполнении задания без лишних запросов (рис. 13.3). При создании задания на конвертацию по WebSocket веб-сервер создает отдельный поток, который подписывается на события, связанные с идентификатором задания.

В отличие от создания задания на конвертацию по WebSocket, при создании задания через «классический» интерфейс события (рис. 13.4) эти сообщения остаются невостребованными и попадают к фоновому процессу.

При создании задания для агента веб-сервер помещает в сообщение исчерпывающий набор данных:

- идентификатор задания;
- ссылку на объектное хранилище;
- желаемый формат;
- настройки, специфичные для желаемого формата.

Это сообщение отправляется агенту, и начинается конвертация.



Рис. 13.3. Создание задания на конвертацию по WebSocket



Рис. 13.4. Создание задания через «классический» интерфейс события

Проектирование агента

Агент — это программа, которая, с одной стороны, подключена к очереди сообщений, а с другой стороны, управляет процессами (рис. 13.5). Предполагается, что

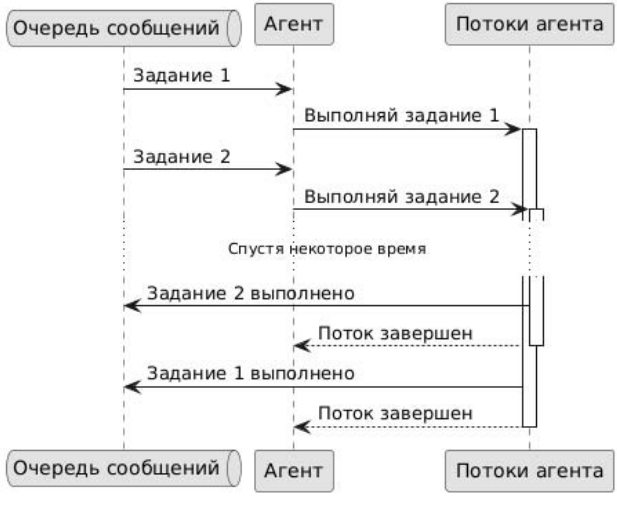


Рис. 13.5. Основной процесс агента

агент запускается один на виртуальную машину и получает все вычислительные ресурсы этой виртуальной машины.

Основной процесс агента конфигурируется в зависимости от доступных обработчиков и вычислительных ресурсов, доступных в текущем окружении. Например, если в виртуальной машине доступны инструменты для конвертирования изображений, то агент подпишется на все темы, которые называются по известным форматам изображений. Если в окружении возможно конвертировать видео, то агент подпишется на темы, содержащие названия видеоформатов. В процессе конфигурации вычислительных ресурсов агент вычисляет количество доступных ядер и распределяет задания строго по ядрам. Например, если конвертация изображения занимает одно ядро, а виртуальная машина восьмиядерная, то одновременно может быть запущено до восьми потоков конвертации. Если конвертация видео занимает все доступные ядра, то одновременно может быть запущена только одна такая задача.

> ***Примечание***
>
> В тексте используется термин «потоки», однако при реализации это могут быть как потоки (thread), так и отдельные дочерние процессы. Потоки — более легкие с точки зрения операционной системы, но могут «уронить» основной поток и всех «соседей», а процессы — более безопасны, но более сложны в управлении.

Обработчики для конвертации могут быть подключены аналогично модулям-обработчикам изображений, как это было сделано в *главе 11* при проектировании неинтерактивного редактора изображений.

Каждый поток обработчика остается «подключенным» к брокеру сообщений и при успешном выполнении задания отправляет ACK — т. е. подтверждение доставки, что трактуется брокером сообщений как успех. В случае ошибки отправляется отказ, и брокер сообщений возвращает сообщение в очередь. Если дочерний процесс обработчика аварийно завершился, то отказ отправляет основной процесс — тем самым обеспечивается бесперебойная работа конвертора.

Проектирование взаимодействия с инструментами конвертации

Агент конвертации должен вызывать программы, которые выполняют полезную работу. В зависимости от используемого программного обеспечения может быть несколько вариантов взаимодействия, к которым нужно быть готовым:

- Неинтерактивное взаимодействие.

  Управление программой происходит через командную строку или при помощи файла с командами. Это лучший способ взаимодействия для автоматизации, т. к. последовательность команд может быть сгенерирована агентом и передана конвертирующему программному обеспечению. Максимально стабильный интерфейс взаимодействия: программа-программа. Пример неинтерактивного интерфейса — редактор изображений, спроектированный ранее (см. *главу 11*).

В неинтерактивное взаимодействие также входят скриптовые языки внутри программного обеспечения. Например, Blender, программное обеспечение для создания 3D-графики, имеет встроенный интерпретатор Python, который управляет внутренним состоянием программы. Таким образом можно определить последовательность действий в Python-скрипте и передать этот скрипт на выполнение в Blender.

- Интерактивное текстовое взаимодействие.

  Это менее стабильный интерфейс взаимодействия, предполагающий, что агент конвертации симулирует поведение человека. В таком взаимодействии программное обеспечение может задавать уточняющие вопросы в процессе работы, а также выводить на экран текст, который удобен для чтения человеком, но не для программы. Например, при подключении по SSH к технике умного дома (IoT, Интернет вещей) можно встретить диалоговое меню вместо командного интерпретатора. В этом случае остается только притворяться человеком.

  В семействе операционных систем UNIX для этого создали программу expect, которая выполняет определенную последовательность действий над приложениями, которые хотят интерактивного текстового взаимодействия. Хотя изначально expect создавалась для тестирования, эта утилита применяется для автоматизации интерактивного взаимодействия.

- Интерактивное графическое взаимодействие.

  Возможно, это самый нестабильный способ для автоматизации приложений с графическим интерфейсом. Такой способ нужен для программ, которые предоставляют исключительно графический интерфейс. Принцип его работы аналогичен expect — имитировать действия пользователя. Различие в том, что инструменты для графических приложений имеют больше функций: управлять окнами, обнаруживать элементы интерфейса, манипулировать мышью. Однако приложения с эмуляцией графического интерфейса может быть трудно автоматизировать параллельно из-за ограничений операционной системы или самого приложения.

  Для управления графическими приложениями используется утилита expect или язык AutoIt.

Проектирование фонового процесса

Задача фонового процесса — поддерживать порядок в системе и периодически удалять устаревшие данные. Хотя технически можно принять на веру, что объектное хранилище у провайдера инфраструктуры бесконечное, тем не менее бюджет на оплату бесконечного хранилища также приближается к бесконечности. Поэтому удалять неактуальные файлы — это необходимость. Фоновый процесс решает следующие задачи:

- очищает объектное хранилище от файлов-результатов, которые были созданы более чем 14 дней назад;

- очищает объектное хранилище от исходных файлов, для которых уже существует файл-результат;
- обрабатывает сообщения о завершении, если задание было запущено из «классического» интерфейса. Обработка сообщения включает в себя удаление оригинального файла.

Масштабирование фонового процесса возможно только частично. Однотипные плановые задачи, такие как очистка объектного хранилища, не должны запускаться параллельно, чтобы не было конфликта за общий ресурс. Обработчик сообщений завершения может быть масштабирован, т. к. он является потребителем сообщений.

Выбор технологий

Для разработки веб-сервера и фонового процесса рекомендуется использовать высокоуровневые языки программирования с поддержкой ORM и наличием фреймворков для очередей сообщений и непосредственно веб-сервера.

Выбор языка программирования для разработки агента — более сложная задача. В отличие от веб-компонентов, агент должен запускаться на разных операционных системах. Это обусловлено тем, что не все инструменты для конвертирования файлов существуют для всех операционных систем. Например, 3ds Max, программное обеспечение для 3D-моделирования, поддерживает исключительно Windows. Разумеется, найдутся программы, которые поддерживают только Linux или только macOS.

Для кросс-платформенного агента подойдут языки программирования Python и Java. Допустимо также разрабатывать специфичные решения под отдельные семейства операционных систем — тогда следует использовать C++, Swift, Go, C#. Разработка отдельных агентов потребует больше времени, но это позволит выиграть в производительности.

Виртуальные машины с агентами — это место, которое должно быть масштабируемым. Здесь стоит использовать оркестраторы вроде Kubernetes или Nomad с автоматическим масштабированием в зависимости от потребляемых вычислительных ресурсов.

Ограничение анонимных пользователей

Основной способ монетизации для проектируемой системы — «премиум»-подписки для пользователей, снимающие ограничения, наложенные на пользователей, подписки не имеющих. Исходя из этого, необходимо продумать детали, которые помешают хитрым пользователям использовать ресурсы сервиса бесплатно.

Запрет использования сервиса в качестве файлового хранилища

Это сейчас маловероятная ситуация: пользователь загружает файл на конвертацию и отдает ссылку на оригинальный файл другому пользователю. В текущей архитек-

туре оригинальный файл удаляется сразу после завершения конвертации, поэтому такой сценарий маловероятен, но это не дает гарантий, что в будущем не будут внесены изменения, которые заставят файлы задержаться подольше.

Чтобы пресечь возможность использования сервиса как файлового хранилища, необходимо в объектном (облачном) хранилище сделать два контейнера (рис. 13.6): публичный и частный (приватный). Все входящие файлы загружаются в частный контейнер, а доступ туда имеют только компоненты системы. Публичный контейнер доступен без аутентификации, и в нем содержатся исключительно результаты конвертации.



Рис. 13.6. Публичный и приватный контейнеры в облачном хранилище

Ограничение количества действий с одного адреса

Ограничение бесплатных действий — это мотивация пользователей как минимум зарегистрироваться. При каждой конвертации следует записывать информацию о времени конвертации, имени пользователя или его IP-адресе, если пользователь без аккаунта. Это позволит вести учет количества доступных для пользователя действий.

Хотя на первый взгляд проблема кажется несущественной, здесь есть важный момент. До сих пор в Интернете используется адресация IPv4, в которой всего 2^{32} адресов. Всем адресов не хватает, поэтому применяются различные технологии — например, NAT, которые позволяют нескольким пользователям делить один адрес.

Такой подход может привести к недовольству анонимного пользователя, если его «сосед по адресу» исчерпал лимит бесплатных действий, но это меньшая проблема. На смену IPv4 уже много лет медленно внедряют IPv6, в котором 2^{128} адресов. Подвох, связанный с IPv6, заключается в том, что на текущий момент многие провайдеры связи очень щедро раздают адреса подсетями /64, позволяя одному клиенту занимать до 2^{64} адресов. В результате при подключении через IPv6 один пользователь может использовать бесплатные действия для адреса, переходить на следующий адрес и продолжать использовать сервис без ограничений.

Решения здесь получаются не самые красивые: не использовать IPv6, не выполнять «анонимных» бесплатных действий для клиентов с IPv6, при подсчете использовать адрес подсети, предполагая, что маска подсети /64. Компромиссным вариантом представляется запрет на бесплатные действия с адресов IPv6. Все эти ограничения мотивируют пользователей зарегистрироваться, чтобы бесплатные действия считались для аккаунта, а не для IP-адреса.

Запрет регистрации с фальшивых почтовых адресов

В Интернете существуют сервисы, которые предоставляют кратковременный виртуальный почтовый ящик, на который можно зарегистрировать аккаунт и, самое важное, подтвердить владение почтой. На самом же деле через некоторое время почтовый ящик исчезнет, и пользователь никогда не получит рассылки от сервиса, где он зарегистрировался. Тем не менее это средство минимизации нежелательных рассылок может быть использовано для получения бесплатных действий в проектируемой системе.

К сожалению, гарантированных способов борьбы с регистрацией ненастоящих аккаунтов пока нет. Вы можете, например, для предоставления бесплатного тарифного плана предложить пользователю привязать к сервису банковскую карту. Впрочем, скорее всего, этот способ отпугнет пользователя, т. к. сервис еще не показал свои возможности, а уже «покушается» на деньги. Можно также ввести «белый список» доменов, на которых нужно иметь почту, чтобы зарегистрировать аккаунт. Это менее отпугивающее решение, но популярные почтовые сервисы не запрещают регистрировать несколько аккаунтов, так что это лишь немного усложнит жизнь любителям бесплатного, но не более.

Ограничение по объему и форматам

Вместо ограничения регистрации новых пользователей следует ограничить объем работ, которые могут выполнять пользователи на бесплатном тарифном плане. Конвертация PNG-изображения в JPEG-изображение — быстрая операция, которая потребляет несколько секунд времени и пару мегабайт объектного хранилища. Конвертация трехчасового фильма в разрешении 4K — вот это уже более сложная задача, которая не должна быть доступна на бесплатном тарифном плане.

Безопасность

Ранее, при проектировании текстового блога (см. *главу 12*), обсуждалась безопасность при работе с данными, которые поступают снаружи системы. SQL-инъекции остаются актуальными, но, как и ранее, использование ORM-фреймворка предотвращает реализацию этого риска.

Настоящей проблемой для проектируемой системы являются пользовательские файлы, которые необходимо открывать соответствующими программами. При этом существует несколько рисков, связанных с некоторыми типами файлов:

- Исполняемый код.

  Некоторые форматы файлов сохраняют в себе исполняемый код, который может быть случайно запущен при открытии файла. Например, текстовый процессор Microsoft Word предоставляет формат файла *.docm, в котором могут быть записаны макросы — скрипты на языке VBA. Исполнение произвольного кода обычно считается наиболее критической уязвимостью, т. к. это позволяет сделать буквально всё что угодно. Есть два решения ситуации: запрет таких файлов или «безопасный режим» в программе. Например, в MS Word при открытии файла, скачанного из Интернета, макросы автоматически отключаются.

- Уязвимости в ПО.

  Агент конвертации запускает программы, в которых могут быть дефекты. Следовательно, может произойти ситуация, когда специально сформированный файл может привести в лучшем случае к аварийному завершению программы, а в худшем — к выполнению произвольного кода. Этот риск смягчается административно: необходимо постоянно обновлять используемое программное обеспечение и при необходимости запрещать определенные форматы файлов до устранения уязвимости.

- Штатное поведение.

  Наиболее редкий, но все еще неприятный случай — это штатное поведение, которое потребляет невероятное количество ресурсов. Обычно это связано со сжатыми файлами, и наиболее известным примером является файл 42.zip. Это архив, который занимает 42 килобайта, но при распаковке создает файл объемом 4,5 петабайта. Проблема здесь в том, что процесс распаковки происходит совершенно ожидаемым способом, но, скорее всего, ресурсы на накопителе закончатся быстрее, чем файл будет распакован. Защититься от таких файлов достаточно сложно. Можно составить список наиболее известных деструктивных файлов, а также следить за потреблением ресурсов и аварийно завершать приложение, если потребление ресурсов превышает заданное максимальное значение.

Взаимодействие с платежными системами

Проектируемый сервис предполагает предоставление платных услуг, поэтому необходимо использовать платежные сервисы. Оставим за рамками юридическую и бюрократическую подготовку к внедрению платежной системы. Вместо этого сосредоточимся на разработке и тестировании.

Платежные системы предоставляют набор разработчика (SDK, Software Development Kit), что позволяет с минимальными усилиями интегрировать взаимодействие с платежной системой. Помимо SDK, у платежных систем существует тестовое окружение, в котором работают все функции, но без использования настоящих денег. В этом окружении предоставляются виртуальные карты с фиксированным поведением: от успешной оплаты до различных отказов из-за недостатка средств или технических проблем.

Автоматизировать тестирование на этом окружении может быть затруднительно, однако ручного тестирования будет вполне достаточно.

Приватность данных

Пользователи загружают данные для конвертации и надеются, что, кроме них, никто не сможет скачать результат. Однако с технической точки зрения итоговый файл доступен всем, кто знает ссылку. Использование случайных идентификаторов UUIDv4 или аналогичных значительно усложняет перебор, но не защищает файл от несанкционированного доступа.

Пароль доступа

Защита доступа по ссылке решается установкой пароля. Но в объектном хранилище нет концепции пароля, есть только подписанные ссылки (pre-signed URL), которые определяют временны́е права доступа при переходе по этой ссылке даже к частным хранилищам. Веб-сервер может генерировать подписанную ссылку с очень ограниченным временем действия, что не позволит воспользоваться ссылкой при случайном ее обнаружении спустя время.

Альтернативный вариант — записывать хеш пароля в атрибуты файла в объектном хранилище, а вместо прямого обращения к объектному хранилищу использовать собственный прокси (рис. 13.7). Задача этого прокси — проверять наличие пароля на файле и отдавать файл только при правильном пароле.

Шифрование данных

Спроектированная система сохраняет результат конвертации в объектное хранилище в открытом виде. Пользователи «изолированы» друг от друга незнанием ссылок, но администратор системы имеет более широкие права и может запросить у объектного хранилища список всех файлов в контейнере. Соответственно, администратор имеет доступ ко всем файлам пользователей. В идеальном мире так быть не должно — файлы должны быть недоступны даже администратору.

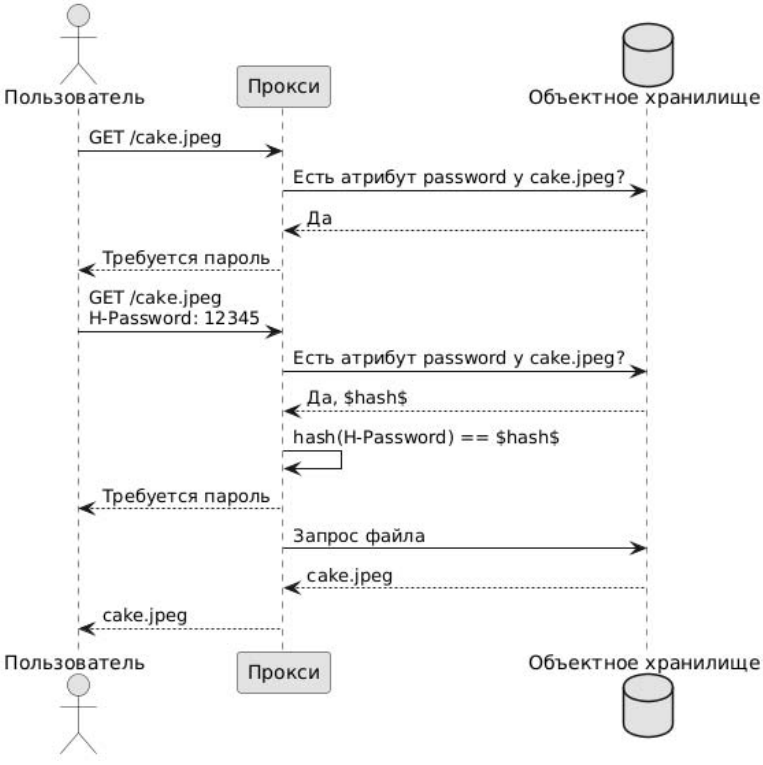


Рис. 13.7. Использование хеша пароля, записанного в атрибуты файла

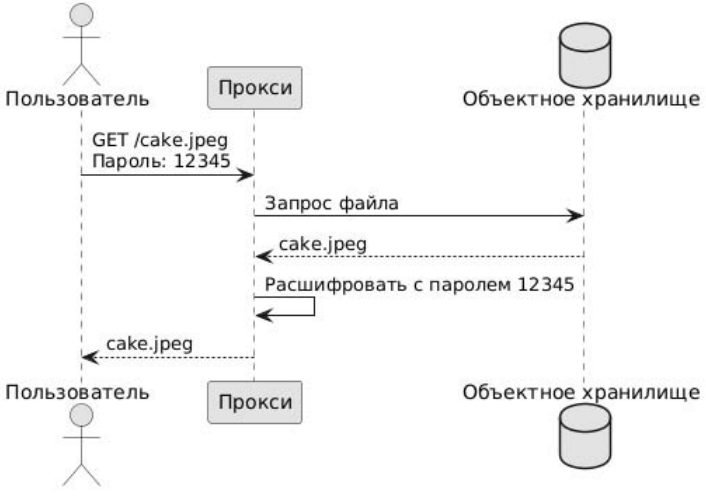


Рис. 13.8. Схема с шифрованием входного файла

При загрузке файла на конвертацию пользователь должен указать пароль, который не попадает в базу данных, но используется для шифрования входного файла и передается агенту через очередь сообщений (рис. 13.8). Агент дешифрует оригинальный файл и затем, перед загрузкой в объектное хранилище, шифрует результат переданным паролем. После этого сообщение удаляется из очереди сообщений, а оригинальные файлы удаляются у агента. С этих пор результат доступен только тому, кто знает ссылку и пароль. В этом случае прокси не проверяет пароль, а пытается им расшифровать файл. Если расшифровка происходит успешно, то файл отдается пользователю. Иначе — ошибка.

Этот способ шифрования обеспечивает максимальную безопасность, т. к. после завершения всех действий пароль не существует в системе. Более того, если пользователи придерживаются лучших практик безопасности и не используют простые и повторяющиеся пароли, то все файлы в объектном хранилище будут зашифрованы разными паролями, и даже максимальный уровень доступа к хранилищу не позволит посмотреть пользовательские файлы.

Логирование и мониторинг

Микросервисный подход, используемый в спроектированной системе, требует внимательного наблюдения.

- Идентификатор задания.

  Ранее при проектировании веб-сервера задействовался уникальный идентификатор задания, обеспечивающий отсутствие пересечений с именами файлов.

  Идентификатор задания — это очень полезный параметр, который позволяет отследить обработку запроса на протяжении всей системы. При разработке компонентов системы необходимо, чтобы каждая строчка, попадающая в логи, содержала идентификатор задания. Тогда при настройке агрегации логов с веб-сервера, агентов и фонового процесса можно получить полную картину выполнения задания. Использование идентификатора задания — это распространенная практика во многих системах. В англоязычных источниках используют идентификатор трассировки — trace id.

- Мониторинг ресурсов.

  Вычислительные ресурсы в системе с динамической масштабируемостью должны всегда контролироваться. Вот список параметров, которые необходимо отслеживать:

  - количество выделенных виртуальных машин для агентов;
  - процент загрузки процессора;
  - объем свободной оперативной памяти;
  - объем свободного дискового места для виртуальных машин с агентами;
  - количество выполняемых заданий агентами;

- количество сообщений в очередях сообщений;
- суммарный объем данных в каждом контейнере объектного хранилища;
- HTTP-коды ответов веб-сервера.

Критерии срабатывания оповещений придется выставлять опытным путем. Однако если количество сообщений в какой-то из очередей не убывает, а только возрастает со временем, то это критерий того, что в системе что-то пошло не так.

Распределенная архитектура

Теперь, когда система полностью функциональна, есть возможность развернуть ее в других географических регионах. К сожалению, сеть доставки контента (Content Delivery Network, CDN) не поможет, т. к. у нас нет статического контента, а каждый конвертированный файл, скорее всего, скачивается один раз. Это делает CDN в нашей задаче бесполезным инструментом и мотивирует к созданию распределенной архитектуры.

Несколько экземпляров системы, развернутые в разных регионах, будут отлично работать независимо друг от друга, и при настройке GeoDNS с маршрутизацией на ближайшую из систем большинство пользователей не заметят отсутствия связности между системами. Однако отсутствие связности скажется на пользователях, которые перемещаются и из-за GeoDNS могут попасть в другой экземпляр системы. Здесь-то и выяснится, что второй регион не знает о пользователях первого региона. Это особенно плохо, если пользователь в первом регионе купил платную подписку. Следовательно, нужна синхронизация пользователей между системами.

Синхронизация данных между распределенными системами — это сам по себе сложный процесс, поэтому не нужно его излишне усложнять. Назначим каждому экземпляру системы уникальный число-буквенный идентификатор, чтобы при регистрации пользователю выдавался идентификатор вида `{логинПользователя}_{идентификаторСистемы}`. Это менее красиво, чем исключительно логин пользователя, но функционально, т. к. при этом в разных экземплярах системы не будет идентичных идентификаторов.

При регистрации нового пользователя система рассылает всем остальным системам уведомление о том, что зарегистрирован новый пользователь, и системы в других регионах создают в своих базах данных соответствующую запись (рис. 13.9). Аналогично в рассылке могут быть отмечены события покупки платной функциональности и запуск конвертаций, чтобы в удаленных регионах для пользователя также имелась история его действий. В случае недоступности экземпляра в другом регионе система должна накапливать сообщения до восстановления связи.

Этот подход позволяет системам существовать практически независимо, периодически обмениваясь событиями. Однако есть и побочные эффекты. Например, если откроется экземпляр системы в регионе MSK, а затем закроется экземпляр в регио-

не LED, то в системе будет существовать множество пользователей с суффиксом экземпляра, которого больше нет в распределенной системе. Это не повлияет на работоспособность системы, но пользователи могут похвастаться уникальным суффиксом как пятизначным номером в ICQ. Впрочем, если вместо пользовательского ника выдавать числовой идентификатор, то так и будет.

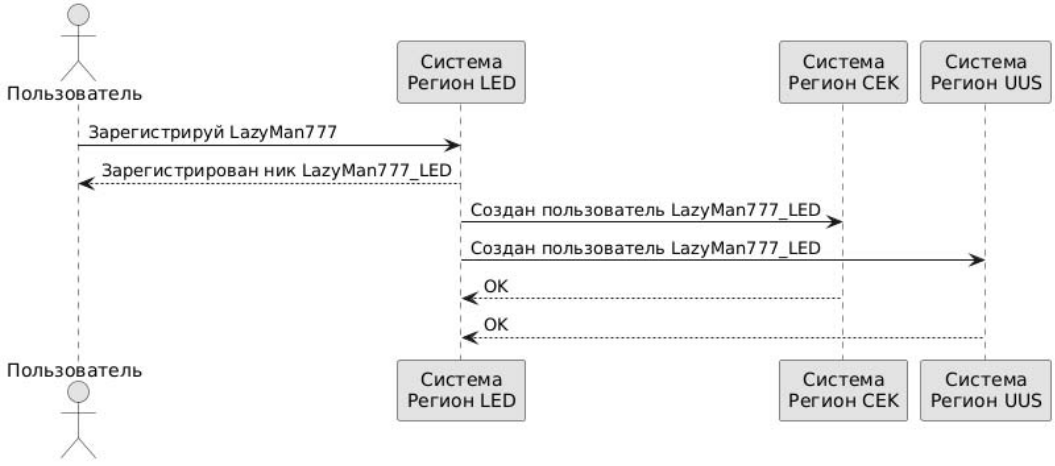


Рис. 13.9. Распределенная архитектура на основе использования уникальных число-буквенных идентификаторов

Улучшение пользовательского опыта

Предложенная архитектура нацелена на функциональность и во многом представляет «спартанские» по современным меркам интерфейсы. Но можно внести и улучшения, которые сделают опыт общения с системой более приятным:

Подробные статусы

Предлагаемая система полностью лишена статусов процесса: пользователь способен отслеживать только выгрузку оригинального файла и скачивание результирующего. Но между этими двумя состояниями — тишина. Вот это место и можно улучшить. Например, модернизировать агента, чтобы о каждом шаге создавалось оповещение для пользователя. Диаграмма последовательности (рис. 13.10) показывает примерные статусы, которые могут быть отправлены пользователю.

Примечание

Для улучшения читаемости показанной на рис. 13.10 схемы, агент уведомляет пользователя напрямую. На самом деле агент отправляет статусы в очередь сообщений, и уже откуда их извлекает поток, обрабатывающий WebSocket-подключение с владельцем файла. В случае отсутствия WebSocket-подключения уведомления о смене статуса обрабатываются в фоновом процессе и могут быть высланы по электронной почте или отправлены в мобильное приложение, если таковое существует.

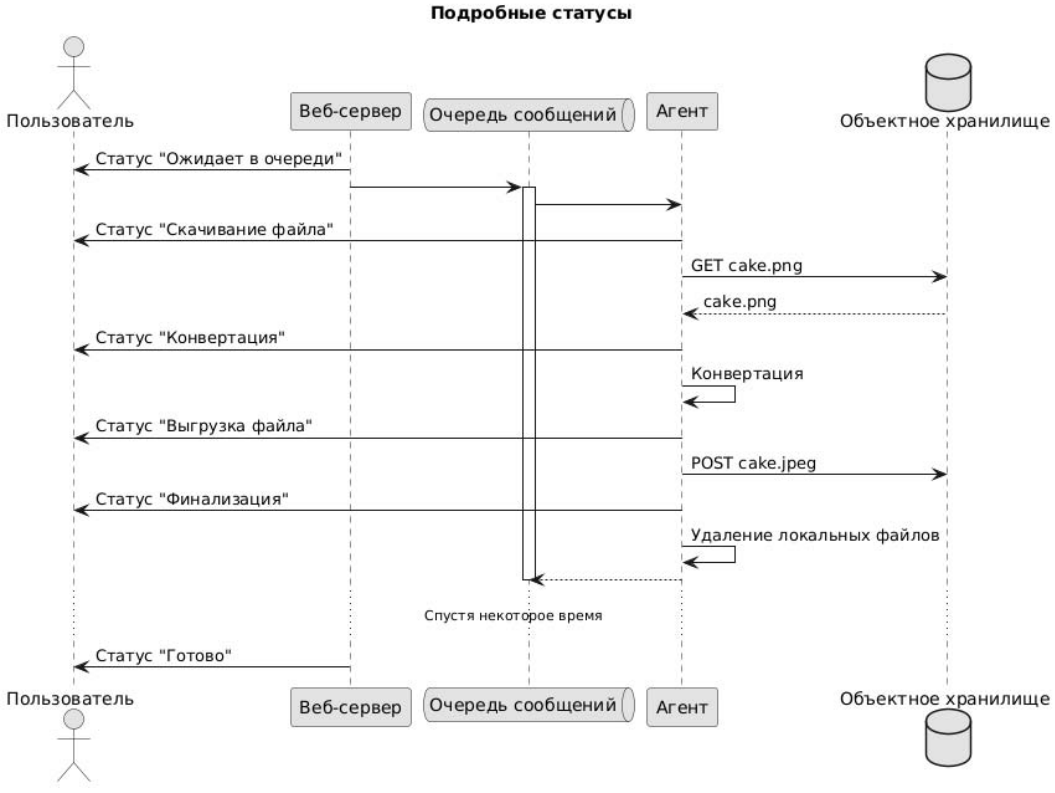


Рис. 13.10. Примеры статусов, которые могут быть отправлены пользователю

Хеширование входных файлов

В некоторых случаях результат конвертации может пользователя не устроить — например, фильм в новом формате не воспроизводится на его плеере. Для новой попытки придется вновь загружать файл на сервис, чтобы конвертировать его в приемлемый формат, и снова скачать результат. Для пользователя в такой ситуации можно реализовать удобство: считать хеши файлов, и если пользователь пытается загрузить уже известный по хешу файл, то использовать уже загруженный файл.

Однако это удобство влияет на объем занимаемого пространства в объектном хранилище, т. к. исходные файлы должны будут храниться там некоторое время. Решение о внедрении этого улучшения и определении временно́го промежутка, в течение которого должен храниться оригинальный файл, должно приниматься на основе исследования и статистики.

Определение некорректных файлов

Система определяет исходный формат по расширению файла. Однако в некоторых случаях расширение не соответствует действительности, но узнать об этом можно только после начала выполнения конвертации. Избежать такой коллизии можно, если на стороне клиента делать базовые проверки файла. Большинство файлов имеют заголовок, первые байты которого указывают на принадлежность к определенному формату, а также контрольные суммы, которые позволяют убедиться в корректности данных. Это не определит все ошибки, но, по крайней мере, не позволит загрузить видеофайл с расширением png.

Добавление новой функциональности

Микросервисная архитектура предполагает возможность независимой разработки сервисов, однако развертывание новой версии без простоя может оказаться непростой задачей.

Добавление нового формата файла должно производиться в несколько этапов:

1. Развертывание новой версии агента, который поддерживает новый тип файлов.
2. Вежливое выключение старой версии агентов: агенты доделывают уже полученные задания, но новых не берут. После завершения всех заданий агент выключается.
3. Обновление веб-версии для включения поддержки нового поддерживаемого формата.

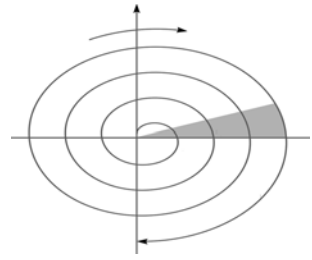
Такой порядок обновления сначала обновляет агенты до поддержки нового формата, а затем допускает пользователей к новой функциональности.

Заключение

В этой главе мы рассмотрели сервис конвертации файлов. Вот его основные особенности:

- ситуация, в которой микросервисный подход к организации такого сервиса действительно подходит, — это географически распределенная система с минимальными дополнительными сложностями;
- когда часть функциональности системы платная, то пользователи будут максимизировать выгоду от ее бесплатной части. Этому не всегда можно эффективно противодействовать;
- по умолчанию нельзя доверять входным данным, а входным файлам нельзя доверять вовсе.
- пользовательские данные должны быть зашифрованы.

ГЛАВА 14



Проектирование интернет-магазина в Telegram

В этой главе рассматривается проектирование интернет-магазина в мессенджере Telegram. В отличие от обычного веб-приложения, мессенджер предлагает свои интерфейсы взаимодействия с пользователем.

Исходные требования

Представим, что компания, занимающаяся общепитом, хочет создать сервис для онлайн-заказа еды. Однако предполагает расположить его на полях мессенджера — вместо «классического» интернет-магазина в браузере или приложения. У этого решения есть своя мотивация:

- мессенджеры уверенно вошли в жизнь людей, поэтому это уже не «новые технологии»;
- после первого обращения к магазину в мессенджере будет создан диалог, который станет напоминанием о магазине, — если принять на веру предположение, что люди удаляют диалоги реже, чем закрывают вкладки в браузере;
- мессенджер — это место, где люди обмениваются сообщениями, поэтому уведомления от мессенджера воспринимаются более положительно, чем от сайтов или сторонних приложений.

Пользовательская история, которую решает этот магазин, — заблаговременный заказ обеда в рабочие дни: пользователь выбирает ресторан, предпочитаемые блюда, бронирует столик и оформляет заказ. Затем он отправляется в выбранный ресторан, и к моменту, когда прибывает в заведение, заказ уже готов, что минимизирует для пользователя время ожидания подачи блюд.

Составим список функциональных требований для магазина — приложение должно:

- работать в мессенджере Telegram;
- предоставлять пользователю (посетителю) возможность просматривать ассортимент товаров;
- позволять ему добавлять позиции в корзину;

- давать возможность выбрать место (столик), куда будет подан заказ;
- позволять оплатить заказ онлайн или на месте;
- обеспечивать наличие роли администратора;
- позволять администратору добавлять, редактировать и удалять позиции в магазине, в том числе временно отключать их с текстом: «будет позже».

Анализ возможностей

Магазин внутри мессенджера задает ограничения на взаимодействие с пользователем, что необходимо учитывать при проектировании системы. В Telegram есть два вида пользователей: люди и боты:

- Пользователи-люди — аккаунты, принадлежащие человеку.

  Могут взаимодействовать с любыми пользователями в мессенджере, инициировать диалоги, отправлять текст, изображения, видео и файлы. Пользователь «привязывается» к номеру телефона. Официально не может быть автоматизирован.

- Боты — служебные пользователи, ограниченные в правах, но созданные для автоматизации.

  Могут отправлять все виды сообщений. В диалоге с ботом есть команды — заготовленные сообщения, боты также способны создавать кнопки для взаимодействия, но не могут первыми начать диалог и общаться с другими ботами. Аккаунт бота создается через BotFather и не требует регистрации по номеру телефона.

Начиная с апреля 2024 года Telegram ввел для пользователей *бизнес-аккаунты*, придав им дополнительные возможности: в его профиле указывается рабочее время, в аккаунте предусмотрены автоматическое приветствие и шаблонные ответы, к бизнес-аккаунтам можно привязать боты, чтобы автоматизировать коммуникацию с другими пользователями, — например, с помощью искусственного интеллекта.

Бизнес-аккаунты удобны для продаж, в которых нужна человеческая консультация и согласование уникальных для каждого случая требований, которые сложно формализовать. Впрочем, при заказе еды дополнительная консультация не нужна, т. к. блюда должны быть описаны в карточке позиции. Исключение ингредиентов из блюда может реализовываться как отдельными параметрами заказа, так и комментарием при заказе, поэтому выбор в пользу бота неоспорим. При необходимости в будущем можно связать бота с бизнес-аккаунтом, но автоматизация в любом случае останется на боте.

Рассмотрим способы коммуникации программы и человека в интерфейсе мессенджера:

- Команды.

  Самый первый способ активации функциональности в боте. Команда — это сообщение, начинающаяся с символа `/`. Telegram определяет самое первое сооб-

щение, которое начинает диалог человека и бота, — человек всегда отправляет команду `/start`. Создатель бота может заявить список поддерживаемых команд, и тогда команды и краткая справка по командам станут доступны в интерфейсе мессенджера. Однако этот шаг не является обязательным и не влияет на функциональность приложения — только на пользовательский опыт.

- Текстовое взаимодействие.

  В личных сообщениях бот получает все сообщения и может реагировать на них. Вести человеческие диалоги могут только дорогие большие языковые модели, поэтому «классическое» человеческое общение с ботом реализуется редко. Вместо этого ботам предоставляется интерфейс для создания кнопок с выбором вариантов ответа. Например, бот отправляет сообщение: «Положить ли двойную порцию лука?» и формирует две кнопки: «Да, пожалуйста» и «Нет, спасибо». При нажатии на кнопку заготовленного ответа в чат отправляется сообщение, эквивалентное тексту кнопки. В процессе это похоже на игру, где диалог поддерживается через выбор вариантов ответа, но в истории мессенджера выглядит как настоящий диалог.

- Интерактивное сообщение.

  Боты могут создавать сообщения с кнопками под сообщением, что позволяет сделать сообщение интерактивным. При нажатии на кнопку под сообщением бот может изменить содержимое сообщения. Например, до появления в Telegram официальной функциональности опросов бот Vote создавал сообщение с кнопками по каждому варианту голосования. Нажатие на кнопку засчитывалось как голосование за соответствующий вариант, а сообщение обновлялось с учетом новых данных. Этот бот работает до сих пор. Интерактивные сообщения позволяют реализовывать многоступенчатые меню, но в истории мессенджера остается только последнее состояние интерактивного сообщения.

- Ссылка на внешний ресурс.

  Сообщение может содержать ссылку на внешний ресурс, который откроется в браузере. При использовании Telegram Login внешний ресурс с разрешения пользователя может получить базовую информацию о пользователе: имя, фамилию, ник, аватарку. Это позволяет внешним сервисам использовать Telegram в качестве сервера аутентификации. Такой вариант взаимодействия полезен для классических интернет-магазинов.

- Мини-приложение.

  Особый вид веб-интерфейса в Telegram — веб-приложение, которое может коммуницировать с клиентом мессенджера, что позволяет модифицировать интерфейс веб-приложения под цветовую гамму мессенджера, использовать виброотклик при нажатиях и многое другое.

Для создания автоматизации в Telegram выбор не большой, поэтому выбираем бота. При необходимости бот может быть совмещен с бизнес-аккаунтом.

Ограничения

При работе с Telegram нужно учитывать, что разрабатываемая система строится практически полностью на основе другого продукта. Это серьезно влияет на разрабатываемую систему, которая приобретает следующие ограничения:

- Секретный токен.

  Бот аутентифицируется в Telegram по секретному токену. Тот, кто владеет токеном, может выполнять любые действия от имени бота. Токен всегда обладает максимальными правами, и нет никакого механизма, чтобы сделать какие-то разграничения. В том случае, если токен по каким-либо причинам стал доступен публично, можно инвалидировать старый токен и создать новый.

- Бот принадлежит пользователю.

  В Telegram бот принадлежит пользователю, который его создал. Это чревато риском, что человек, который изначально создал бота, может уволиться из компании, и тогда она может потерять доступ к настройкам бота, а также к возможности перевыпустить токен. Зато эти возможности останутся у бывшего сотрудника. Указанный риск смягчается созданием отдельного «служебного» пользователя, который не будет принадлежать конкретному человеку.

- Читать может только один.

  В Telegram интерфейс для ботов сделан на основе механизма Long Polling — длинных опросов. Бот отправляет запрос на сервер и ждет, когда сервер отправит обновление (update), — информацию о событии, произошедшем в мессенджере. Событие всегда описывает одно действие: отправку сообщения, редактирование сообщения, удаление сообщения, изменение настроек чата. Обновление, прочитанное ботом, считается доставленным, даже если бот аварийно завершился во время обработки этого обновления.

  При использовании длинных опросов горизонтальное масштабирование становится невозможным — сервер Telegram запрещает множественные подключения для получения обновлений. Таким образом обработчик событий в системе может быть только один.

- Отправлять могут многие.

  Тем не менее интерфейсы для создания событий — например, отправки сообщений, не имеют таких ограничений, что позволяет владеть несколькими экземплярами приложений, которые могут отправлять сообщения. Единственное накладываемое ограничение — все экземпляры, представляющие одного бота, не могут отправлять более нескольких сообщений в один чат в секунду и не могут в одну секунду отправлять сообщения более чем в несколько десятков чатов. Точные значения ограничений доступны в правилах использования интерфейсов для ботов.

- Некоторые функции могут быть платными.

  Например, ограничение по отправке сообщений в чаты может быть смягчено за отдельную плату.

- Боты не хранят историю.

  В отличие от пользователей, у которых есть список диалогов, а в диалогах присутствует история, Bot API не предоставляет функций для получения диалогов или истории сообщений в диалоге. Если необходимо, разработчик бота должен сам сохранять список пользователей и сообщения от них.

- Необходимость абстрагироваться от мессенджера.

  При проектировании систем такого рода возникают экзистенциальные вопросы. Всегда ли мы будем использовать Telegram? Вдруг придется переходить на другой мессенджер? Стоит ли тут «подстелить соломки» и спроектировать абстрактный интерфейс, который при необходимости позволит быстро адаптировать существующую систему к другому мессенджеру? Ответ на этот вопрос требует серьезного анализа рисков.

  С технической точки зрения абстрагирование от функций мессенджера может заключаться в создании дополнительного слоя, который станет усложнять взаимодействие с мессенджером и при этом может никогда не пригодиться. Возможный компромисс в этой ситуации — построение архитектуры, подобной микросервисной, где микросервис-«ядро» решает бизнес-задачи на основе каких-то входящих сообщений, а отдельные микросервисы реализуют взаимодействие с выбранным мессенджером и конвертируют внутренние унифицированные сообщения в интерфейс в диалоге.

Проектирование интерфейса бота

Для взаимодействия с пользователем выбираем интерактивное сообщение, потому что этот вариант не засоряет диалог техническими сообщениями, которых может быть очень много, когда пользователь листает меню заведения.

Во избежание большой вложенности разделим размещение заказа на пять этапов — четыре исходных: выбор ресторана, выбор блюд, выбор времени, к которому всё должно быть готово, и выбор места (столика), за которым будет сидеть посетитель, и, после заполнения всех исходных параметров, пятый — возможность подтвердить корректность заказа. Диаграмма состояний, приведенная на рис. 14.1, приближенно показывает доступные пункты меню интерактивного сообщения и переходы между ними. При этом выбор каждого этапа доступен из «главного меню» интерактивного сообщения. Это добавляет пользователем дополнительные нажатия на кнопку «назад» для возврата в главное меню и придает структуре меню большую стройность.

Каждый заказ (рис. 14.2) начинается с выполнения команды `/start`. В этом состоянии пользователю доступны выбор ресторана, блюд и времени посещения. Система отдельно различает интерактивное сообщение, в котором не заполнено ничего, от сообщений, в которых заполнено хотя бы одно поле. После заполнения всех трех доступных полей открывается возможность выбора места в ресторане.



Рис. 14.1. Состояния интерактивного сообщения



Рис. 14.2. Состояния заказа

Как только пользователь указывает выбранное в ресторане место, заказ переходит в состояние «забронирован» — т. е. место, выбранное пользователем, становится недоступным для выбора другим пользователям. Это предварительное состояние — далее пользователю нужно подтвердить свои намерения. В зависимости от бизнес-модели это могут быть текстовое подтверждение: «Да, я точно буду» или внесение предоплаты.

Выбор ресторана и времени — это простой выбор варианта из множества возможных. Иначе обстоят дела с выбором блюд. Даже в заведениях быстрого питания

имеется несколько категорий блюд, и каждая из них содержит свой набор позиций. Соответственно, можно составить навигацию именно по этому принципу: категория блюда — блюдо. При этом на «странице» категорий и на «странице» блюд сделать кнопки: «следующая страница» и «предыдущая страница» — чтобы в каждый момент времени количество кнопок под сообщением не превышало 10–15 (рис. 14.3). Хотя Telegram позволяет размещать до 100 кнопок под одним сообщением, большое количество информации на экране требует дополнительного действия — пролистывания списка. А кнопки навигации по страницам позволяют держать интерактивное меню в рамках одного экрана.

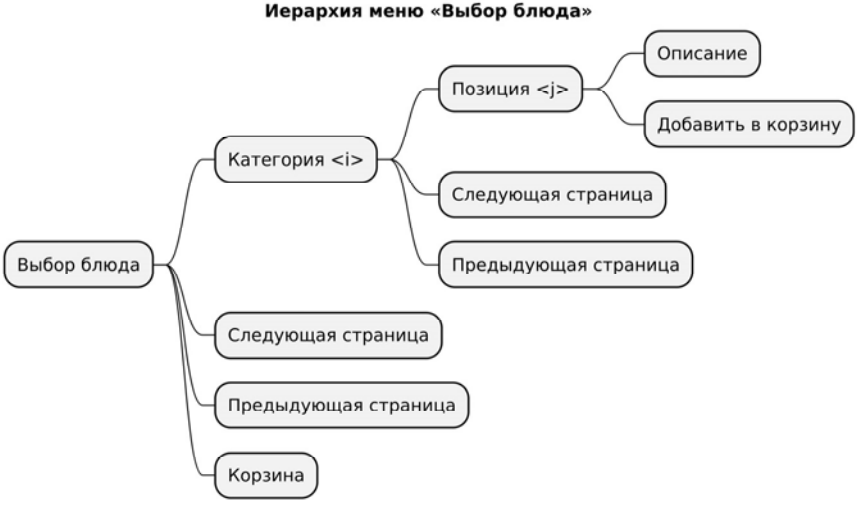


Рис. 14.3. Иерархия меню «Выбор блюда»



Рис. 14.4. Пример итоговой навигации

Несмотря на громоздкость в описании, итоговая иерархия может быть достаточно лаконичной, если блюда будут разделяться не более чем на десять категорий, в каждой из которых будет не более десяти позиций. Пример итоговой навигации при выборе ресторана, времени посещения, заказываемых блюд и места (столика) приведен на рис. 14.4.

Проектирование зоны администратора

Хотя исходное требование при проектировании системы — что «приложение должно работать в мессенджере Telegram» — имеет свою мотивацию с точки зрения пользователя, но оно создает неудобства для управления приложением. Создание, редактирование и упорядочение блюд в мессенджере возможно, но функциональности в чатах Telegram недостаточно для отображения большого количества информации, таблиц, графиков и карт заведений.

В этом случае следует воспользоваться возможностью создавать веб-интерфейсы, «привязанные» к Telegram. В анализе возможностей было определено, что Telegram позволяет использовать два вида веб-интерфейсов: отдельный веб-сервис, в котором аутентификация происходит через Telegram, и мини-приложения, имеющие связь с клиентом мессенджера. Мини-приложения адаптированы для вертикальной ориентации экрана, т. е. для мобильных устройств, что может быть неудобно. Поэтому остается только внешний веб-сервис (рис. 14.5).

Веб-сервис — это дополнительный расширенный интерфейс для администраторов, который тем не менее должен иметь связь с Telegram. Эту связь можно реализовать через систему аутентификации Telegram Login. При вводе команды `/admin` бот отправляет пользователю сообщение, которое содержит ссылку на веб-сервис с типом `login`. В клиенте Telegram это отображается как обычная интерактивная кнопка, но при нажатии на нее появляется диалоговое окно, подтверждающие желание пользователя аутентифицироваться на указанном веб-ресурсе. Если пользователь подтверждает свое желание, то в ссылку вставляются параметры запроса (query parameters), которые содержат основную информацию о пользователе:

- уникальный идентификатор;
- ник пользователя;
- имя и фамилию;
- аватарку;
- время нажатия на кнопку;
- хеш-подпись ссылки.

Хеш-подпись позволяет проверить, что ссылка содержит достоверную информацию и что ссылка на самом деле создана ботом. Если упорядочить все данные в запросе и посчитать HMAC-хеш с использованием секретного токена, который используется для аутентификации бота в Telegram, то вычисленный хеш должен совпадать с переданным. Безопасность этого процесса полагается на тот факт, что секретный токен известен только владельцу бота, по совместительству являюще-

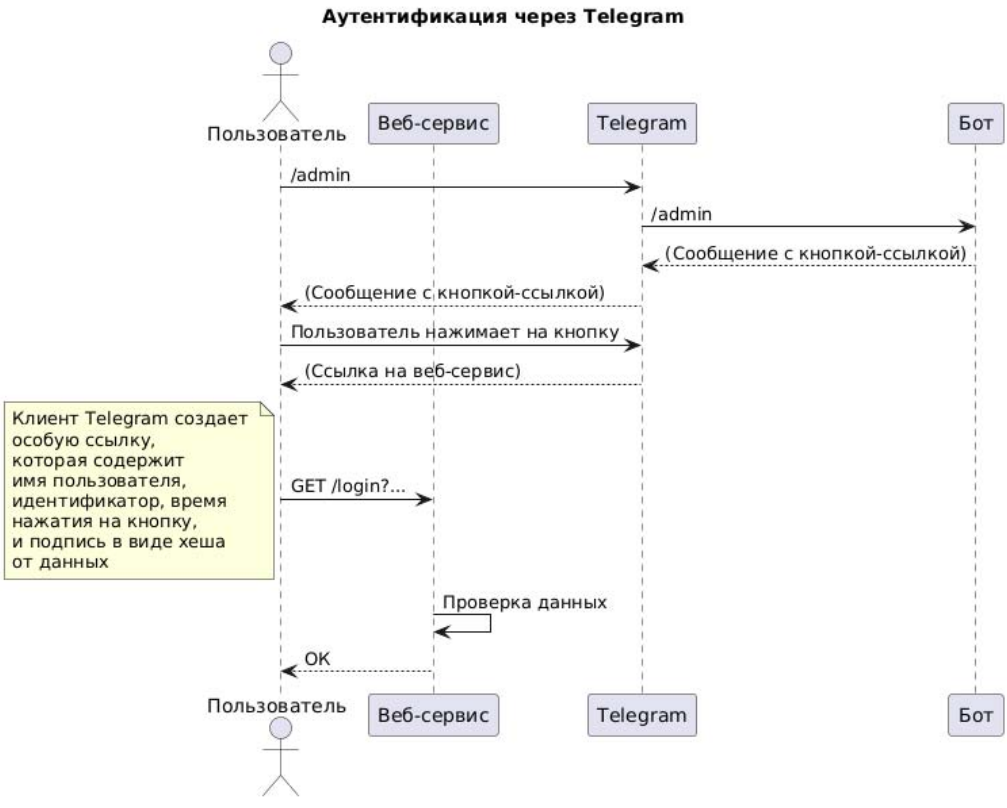


Рис. 14.5. Внешний веб-сервис аутентификации через Telegram

муся владельцем веб-сервиса. При таком подходе веб-сервис может хранить свое представление о пользователях в Telegram.

Хотя Telegram позволяет запрашивать информацию о пользователях, которые начали диалог с ботом, никакого механизма для разграничения прав не существует. Следовательно, веб-сервис может ввести роли и права двумя способами:

- Статический способ.

  Отдельно вести список всех администраторов и их аккаунтов в Telegram. Вручную назначать права.

- Динамический способ.

  Так как веб-сервис может делать запросы к Telegram, то можно воспользоваться необычным решением. Если в компании есть официальный чат сотрудников, то можно добавить в него бот и определять права в системе, исходя из прав пользователей в этом чате. Например, наличие пользователя в чате сотрудников дает право на выполнение команды `/admin`, которая открывает ссылку для доступа в веб-интерфейс, кроме того, наличие пользователя в чате проверяется при аутентификации в веб-сервисе. Далее можно использовать существующие права или строку «должность» для разделения ролей.

Для работы зоны администратора нужно реализовать следующие страницы:

- `GET /place` — список всех ресторанов;
- `POST /place` — добавить новый ресторан;
- `PATCH /place/{place_uuid}` — изменить конфигурацию ресторана;
- `DELETE /place/{place_uuid}` — удалить ресторан;
- `GET /place/{place_uuid}/{date}` — просмотр занятости ресторана с идентификатором place_uuid в день date;
- `PATCH /place/{place_uuid}/{date}` — внесение изменений;
- `GET /menu` — посмотреть меню полностью;
- `POST /menu` — добавить новое блюдо;
- `PATCH /menu/{id}` — редактировать информацию о блюде;
- `DELETE /menu/{id}` — удалить из меню.
- `POST /send` — сделать рекламную рассылку по пользователям в Telegram.

Проектирование архитектуры

Исходя из ограничений Telegram, а также для гипотетической поддержки других мессенджеров, систему можно разделить на три сервиса. На диаграмме развертывания (рис. 14.6) стрелками обозначено направление сообщений от Telegram и к нему:

- сервис-ядро обрабатывает HTTP-запросы от администраторов, запросы из очереди сообщений, а также управляет всей бизнес-логикой приложения;
- «Приемник сообщений» получает обновления от Telegram и конвертирует их в формат, понятный ядру;
- «Отправитель сообщений» извлекает сообщения из очереди и конвертирует их в формат, понятный мессенджеру.

Теоретически приемник сообщений может отвечать на простые запросы самостоятельно, ускоряя время ответа для пользователя. Кроме того, веб-сервис способен сам отправлять сообщения в мессенджер, используя соответствующий компонент. Однако это создает ситуацию, в которой два отправителя ничего не знают друг о друге и могут превысить лимит сообщений, устанавливаемый мессенджером.

Это пример «идеальной» расширяемой и резервируемой архитектуры, которая хорошо смотрится на диаграмме, но может ухудшить пользовательский опыт, — обработка даже самого простого запроса потребует работы трех компонентов и создания двух сообщений в брокере.

Альтернативный вариант — «размазать» отправителя сообщений по приемнику и веб-серверу (рис. 14.7). В этом случае приемник сообщений может быстро отвечать на обновления, которые не требуют сложной бизнес-логики. Примером может служить навигация по меню, описанная ранее.

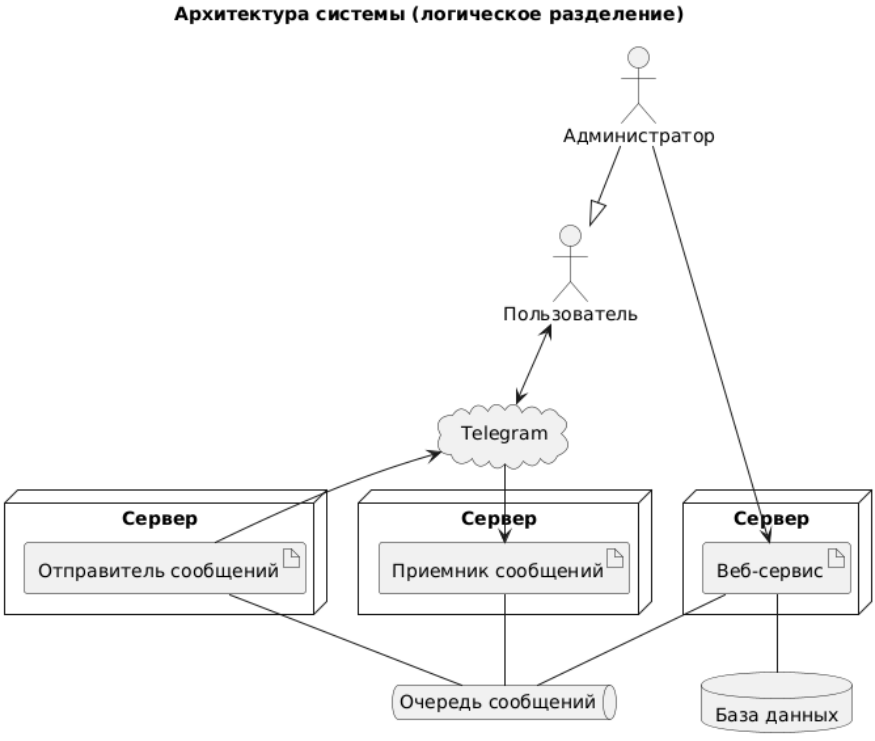


Рис. 14.6. Диаграмма развертывания архитектуры системы: логическое разделение

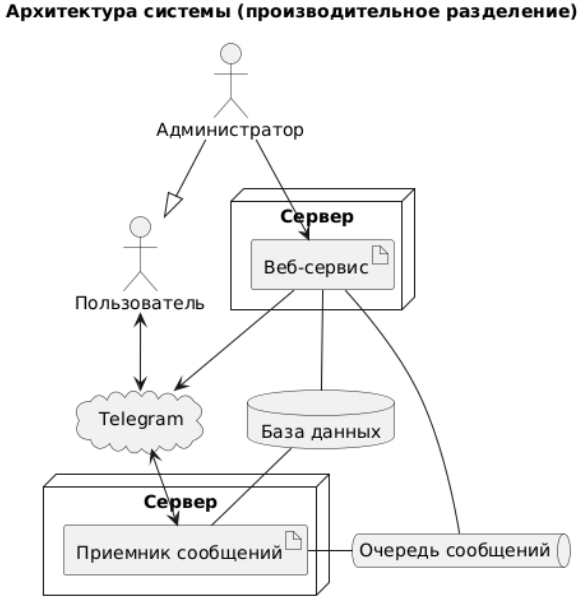


Рис. 14.7. Альтернативный вариант архитектуры: производительное разделение

Недостаток этого способа заключается в том, что отправление сообщений в мессенджер разнесено по нескольким сервисам и нужна синхронизация, чтобы не нарушить ограничение. Однако при ближайшем рассмотрении оказывается, что ограничение распространяется только на массовую отправку уведомлений (без запроса пользователя), в то время как на ответы на сообщения пользователей ограничения не распространяются.

Поскольку приемник сообщений не будет заниматься массовой рассылкой, ограничение необходимо поставить только на веб-сервер. Отправка до 30 уведомлений в секунду разным пользователям — это продолжительный процесс, но за час таким образом будет разослано до 108 тысяч сообщений.

Выбор технологий

Для реализации рассматриваемой задачи выбор языков программирования и библиотек — максимально свободный. Лучшая рекомендация: использовать высокоуровневые языки программирования, такие как Java или Python, однако реализация клиентов для Telegram Bot API существует на множестве языков программирования, включая Pascal, Ocaml и Perl.

В качестве базы данных подойдет реляционная, особых требований к брокеру сообщений нет.

Рассылка уведомлений

Массовая рассылка сообщений в мессенджере — это задача, выделяющаяся на фоне CRUD-операций, которые выполняются в системе. Внешняя система, т. е. Telegram, как уже было отмечено ранее, накладывает ограничение на отправку сообщений — до 30 сообщений в секунду в разные диалоги.

Для массовой отправки сообщений нужно зафиксировать список получателей и сообщение. Сделать это можно в базе данных через две сущности: `Message` и `MessageEntry` (рис. 14.8).

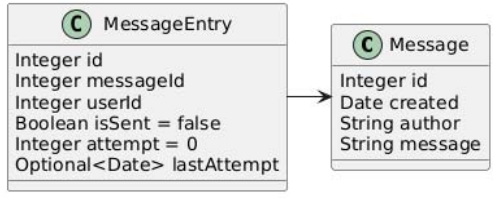


Рис. 14.8. Получатель сообщения (`MessageEntry`) и сообщение (`Message`)

Основная сущность (`Message`) хранит текст, который нужно отправить, а также автора рассылки и время запуска рассылки. Сущность `MessageEntry` соответствует получателю сообщений. При запуске рассылки происходит следующее:

1. Создается сущность `Message` с текстом, текущим временем и автором рассылки для аудита.
2. Из базы выбираются все известные пользователи, и для каждого пользователя создается сущность `MessageEntry` с `messageId`, равным идентификатору сущности `Message` из п. 1, и `userId`, равным идентификатору текущего пользователя.
3. После чего происходит рассылка.
4. Из базы выбираются случайные двадцать сущностей `MessageEntry` с `isSent = false` и `attempt < 3`.
5. Если выборка пуста, то система бездействует одну секунду и повторяет действия, начиная с пункта 1 заново.
6. Для каждой сущности с интервалом $^1/_{20}$ секунды производится отправка текста указанному пользователю и установка значения `lastAttempt`, равного текущему времени.
7. Если отправка успешна, то `isSent = true`.
8. Если неуспешна, то значение поля `attempt` увеличивается на единицу.
9. Повторение, начиная с пункта 1.

Этот способ будет работать, но нужно предусмотреть несколько крайних случаев:

- Пользователь запретил присылать сообщения.

  В мессенджере пользователь имеет полное право отказаться от взаимодействия с ботом, выбрав в меню клиента позицию «Остановить бота», что запретит боту отправлять сообщения этому пользователю. Это штатная ситуация, и она должна быть обработана. В нашем случае попытку можно считать успешной, т. к. выбранное пользователем действие — не техническая ошибка, из-за которой нужно попробовать повторить отправку.

- Частые рассылки.

  Каждая рассылка создает сущности `MessageEntry` по количеству пользователей, известных системе. При этом после завершения рассылки большинство этих сущностей бесполезно. Для оптимизации занимаемого места `MessageEntry` следует удалять — например, по истечении месяца с момента успешной отправки.

- Случайная выборка.

  По умолчанию при выборке `MessageEntry` из базы данных можно не использовать фильтры — тогда порядок строк в реляционной базе данных будет явно не задан и, скорее всего, в выборку попадут те строки, которые недавно редактировались. Если действительно нужна случайная выборка, надо явно добавить элемент случайности — например, выбирать не первые 20 строк, а со случайного смещения оператором `LIMIT`.

- Рассылка в рабочее время.

  В маленьком заведении в одном городе, скорее всего, будет меньше двухсот тысяч клиентов, поэтому рассылка справится со всеми клиентами менее чем за два

часа. Однако система спроектирована так, чтобы в случае ошибки повторять отправку сообщений, поэтому есть теоретическая возможность отправить пользователю рекламное сообщение в ночное время. Чтобы этого избежать, необходимо отдельно предусмотреть запрет рассылки с 19.00 до 10.00.

Примечание

В случае, если система работает для ресторанов, которые находятся в нескольких городах из разных часовых поясов, задача усложняется. Telegram не предоставляет информации о часовом поясе пользователя, так что системе необходимо угадывать его на основе совершенных заказов. Например, если пользователь в своем заказе выбрал ресторан в Новосибирске, то можно предположить, что он находится в часовом поясе этого города.

Горизонтальное масштабирование

В системе можно горизонтально масштабировать веб-сервер, т. к., с одной стороны, он поддерживает HTTP и REST API, а с другой — подключается к брокеру сообщений, который умеет распределять сообщения между потребителями. Горизонтальному масштабированию не поддается только приемник сообщений из Telegram, поскольку метод Long Polling не разрешает несколько параллельных подключений.

Однако при необходимости можно переключить бота с режима Long Polling на режим WebHook, в котором серверы Telegram BotAPI присылают обновления на заявленный адрес по HTTP самостоятельно. В этом режиме боту не нужно запускать соединения, которые ожидают ответа, а реакция на обновление может быть отправлена в ответном пакете, что сокращает количество запросов и увеличивает производительность бота (рис. 14.9).

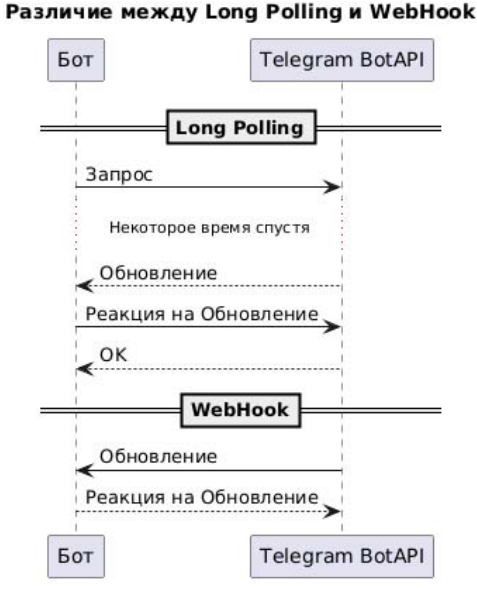


Рис. 14.9. Различие между режимами Long Polling и WebHook

В случае, если этого недостаточно, у режима WebHook есть возможность запустить несколько экземпляров бота, перед которыми стоит балансировщик нагрузки. Это позволяет использовать горизонтальное масштабирование для приемника сообщений.

Мониторинг

При достижении определенного количества ежедневных пользователей Telegram начинает отслеживать ответы бота на события в мессенджере. Если время ответов или их количество снизится относительно ежедневно наблюдаемых значений, то Telegram отправит предупреждение о том, что необходимо проверить бота, — не произошел ли сбой. Хотя этот механизм полезен, он недоступен по умолчанию, и нет возможности проверить, стал ли Telegram отслеживать спроектированного бота. Поэтому не стоит полагаться на этот мониторинг, но знать о нем полезно.

- Очередь сообщений.

  Как и в рассмотренном ранее сервисе конвертации файлов (см. *главу 13*), необходимо отслеживать количество сообщений в очередях. Если количество сообщений растет, то случился сбой или всплеск нагрузки.

- Исключения в программе.

  Чтобы отслеживать ошибки отправки сообщений в Telegram, необходимо отслеживать количество возникающих исключений в коде отправки. Следовательно, необходимо корректно обрабатывать ошибки от фреймворка или библиотеки, которая реализует взаимодействие с BotAPI.

- Проверка состояния.

  Для проверки приемника сообщений можно сделать команду `/check`, которая отвечает OK, и средствами мониторинга обращаться к этой команде с определенным периодом. Это звучит логично и просто, однако, как отмечалось ранее, к ботам могут обращаться только пользователи.

  Для реализации такой проверки состояния необходим служебный пользователь, а также фреймворк, который реализует пользовательское взаимодействие с Telegram.

Дальнейшие улучшения

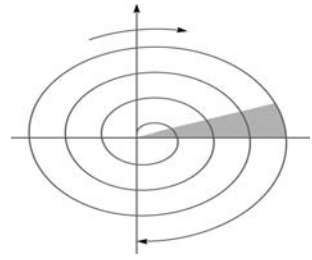
Так как проектируемая система основана на другом программном продукте — мессенджере, то необходимо следить за обновлениями мессенджера и оперативно на них реагировать. Во-первых, это могут быть изменения в API, которые в будущем способны нарушить совместимость. Во-вторых, появляются функции, которые могут повышать удобство взаимодействия с пользователем, а также различные косметические обновления. Например, в BotAPI 8.0 появились методы, позволяющие пользователям выбирать «статус» в имени из списка бота. Это может использоваться в рекламных целях.

Заключение

В этой главе мы рассмотрели сервис, включающий в себя бота в Telegram, и веб-сервис для более удобного администрирования системы. Вот его основные особенности:

- использование стороннего программного продукта в качестве основы накладывает ограничения на возможные способы взаимодействия с пользователем;
- слой абстракции, отделяющий разрабатываемую систему от основополагающего продукта, может быть избыточным и ненужным.

ГЛАВА 15



Проектирование плагина для отправки проекта из программы 3D-моделирования

В этой главе рассматривается проектирование расширения для существующего программного продукта, установленного на компьютере пользователя.

Исходные требования

Необходимо спроектировать решение, которое в минимальное количество действий должно отправлять проект и все его составные файлы из программы для 3D-моделирования во внешнее хранилище.

Составим список требований:

- решение должно получать список всех файлов, используемых в текущем открытом проекте;
- должно архивировать все файлы в ZIP-архив и передавать на удаленный сервер;
- работа решения не должна влиять на работу программы для 3D-моделирования;
- интерфейс решения должен быть интегрирован в интерфейс программы 3D-моделирования.
- решение должно иметь возможность выбора внешнего хранилища.

Ограничения

Как и в случае с интернет-магазином, мы имеем дело с проектированием программного продукта, работающего в связке с другим продуктом, что накладывает некоторые ограничения на возможности и функциональность проектируемого решения. Однако во взаимодействии нашего продукта с программой 3D-моделирования есть ряд отличий от взаимодействия интернет-магазина с мессенджером, речь о котором шла в *главе 14*:

- Запуск на стороне пользователя.

  Расширение для программы 3D-моделирования запускается на компьютере пользователя — т. е. у нас нет возможности полноценного мониторинга и инструментов для отладки в продуктовой среде;

- Целевая платформа.

  У каждой социальной сети и мессенджера есть своя целевая аудитория со своими интересами и особенностями. Это позволяет ограничить список программных решений, с которыми приходится взаимодействовать. Например, в Telegram имеются средства для автоматизации, и ими будут пользоваться. В других мессенджерах боты не так распространены, что позволяет пренебречь абстрагированием от интерфейса конкретного мессенджера в пользу быстродействия системы и скорости разработки. Однако в программах для моделирования нельзя ограничиться каким-то одним решением и игнорировать все остальные. Возможность нажатием одной кнопки сохранить большой проект и загрузить его во внешнее хранилище будет полезна во многих инструментах — как для моделирования, так и для обработки видео. Именно поэтому в этой главе не называются конкретные продукты — решение должно быть универсальным и применимо как можно к большему количеству программных продуктов.

- Фиксированная архитектура.

  Программы для моделирования — это программы с графическим интерфейсом пользователя, так что, скорее всего, в них используется событийно-ориентированная архитектура: действия в графическом интерфейсе генерируют события, которые передаются соответствующим обработчикам. События графического интерфейса выполняются в основном потоке, поэтому наше расширение должно выполнять ресурсоемкие задачи в фоновых потоках, поскольку выполнение сложных операций — например, сетевого взаимодействия, в основном потоке приведет к «зависанию» основного приложения.

Проектирование простой архитектуры

Наиболее очевидное решение — спроектировать расширение, которое обращается непосредственно к внешнему хранилищу (рис. 15.1). В этом случае расширение может иметь скриптовый формат и выполнять единственную функцию: загружать файл в хранилище.

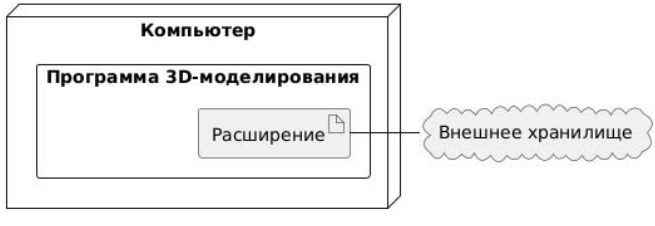


Рис. 15.1. Простая архитектура решения

Для реализации соответствия нашего расширения заявленным требованиям нужно:

- разработать графический интерфейс;
- выполнить необходимые настройки;
- определить адрес внешнего хранилища;
- сформировать данные для доступа, если они будут необходимы;
- предусмотреть шкалу процесса загрузки;
- создать кнопку «Сохранить и отправить».

Этот минимальный интерфейс и функция загрузки обеспечивают выполнение всех имеющихся требований. Для удобства пользователя кнопку «Сохранить и отправить» после начала загрузки можно превратить в кнопку «Отмена». Загрузка обязательно должна производиться в фоновом потоке.

Рассмотрим достоинства и недостатки предложенного подхода:

- Простота.

  Расширения чаще всего представляются в виде одного файла, что, с точки зрения обычного пользователя, выглядит привлекательно: для работы нужно положить файл в указанный каталог и найти новый пункт меню известной программы.

- Отсутствие гибкости.

  Исходные требования предполагают загрузку файлов проекта во внешнее хранилище. Но по какому протоколу должна происходить передача: SSH, FTP, S3? Выбор в пользу одного формата уменьшает потенциальную аудиторию проектируемого решения и создает трудности при смене хранилища, а выбор в пользу универсальности требует включения в состав расширения множества библиотек, которые реализуют максимальное количество протоколов.

- Привязка к языку программирования.

  Большинство программных продуктов, которые предоставляют возможность разработки расширений, имеют ограничение на один или два языка программирования. Например, для создания расширений в Adobe After Effects можно использовать C++ или JSX, а в Blender — Python. Это значит, что, помимо специфичных функций для сбора информации о проекте, придется реализовать логику загрузки во внешнее хранилище на доступном языке программирования, а это может быть неподходящий для нас инструмент. Более того, при разработке расширений для более чем одного программного продукта, возникает проблема, связанная с реализацией на разных языках общей логики отправки на удаленный сервер. Это существенно усложняет исправление дефектов в «общей части».

Предложенная архитектура целесообразна только при выполнении всех указанных далее условий:

- протокол загрузки на удаленный сервер — фиксированный и не будет изменен в будущем;
- расширение разрабатывается строго для одного программного продукта, и это не будет изменено в будущем;

- после загрузки проекта на удаленный сервер не планируется дополнительных действий.

В случае, если хотя бы одно из этих условий может быть нарушено, стоит отказаться от простой архитектуры в пользу более сложного варианта.

Проектирование составной архитектуры

Как показано в предыдущем разделе, использование одного расширения для удовлетворения всех поставленных требований возможно, но лишь при соблюдении определенных условий. В том случае, когда параметры внешнего хранилища могут изменяться, перечень возможных действий с внешним хранилищем расширяться, а количество поддерживаемых программных продуктов расти, создание для каждого возможного варианта взаимодействий отдельных расширений по схеме простой архитектуры вскоре приведет к сложностям при их сопровождении.

Возможный вариант — создание отдельной вспомогательной программы, которая запускается на компьютере пользователя и выполняет функцию прослойки между расширением в программе для моделирования и внешним хранилищем (рис. 15.2).



Рис. 15.2. Дополнение простой архитектуры вспомогательной программой

В составной архитектуре (рис. 15.3) расширение берет на себя исключительно специфичные для программного продукта задачи: сохранение проекта перед отправкой, поиск всех используемых файлов и предоставление графического интерфейса для пользователя. А вспомогательная программа решает «общие» между расширениями задачи: коммуникация расширения со вспомогательной программой, архивация файлов из переданного списка, передача архива во внешнее хранилище.

Вспомогательная программа может реализовывать в программе моделирования несколько интерфейсов для взаимодействия с расширением. Например, для программ, в которых для разработки расширений используется JavaScript, поддержка WebSocket наверняка доступна по умолчанию. А если для разработки используются C++ или Python, то поддержки WebSocket, скорее всего, нет, зато есть возможность использовать TCP-соединение или UDP-пакеты.

Вспомогательная программа для управления действиями должна иметь свой графический интерфейс, в котором пользователю важны следующие компоненты:

- отслеживание процесса загрузки архива;
- возможность отменить загрузку;
- наличие настроек для подключения к внешнему хранилищу.

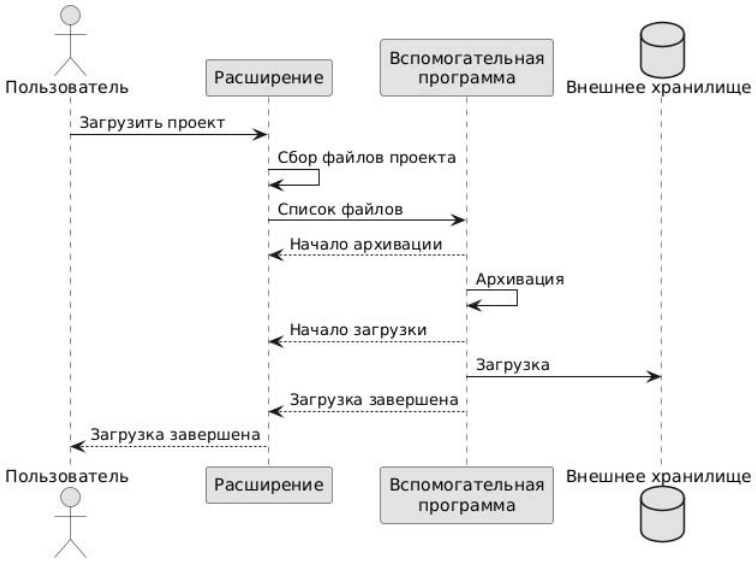


Рис. 15.3. Составная архитектура решения

Необходимо обратить внимание на то, что теоретически возможна ситуация, в которой с одной вспомогательной программой будут взаимодействовать несколько расширений, в том числе из разных программ, а также и несколько экземпляров одной программы, в которой открыты разные проекты.

Выбор технологий

Как отмечалось ранее, язык программирования для написания расширения фиксирован и определяется программным продуктом, для которого разрабатывается расширение.

Вспомогательная программа может быть написана на любом языке программирования, но предпочтительно использовать высокоуровневые языки: Java, Python. Однако если вспомогательная программа создается для внешних пользователей, то предпочтительно задействовать компилируемые языки программирования — например, C++, чтобы усложнить реверс-инжиниринг программы конкурентами.

Логирование

Выполнение расширения и вспомогательной программы происходит на компьютере пользователя, поэтому логи также остаются на этом же компьютере. У пользователя могут возникать проблемы с поиском файлов логов, поэтому во вспомогательную программу необходимо добавить кнопку «Показать логи» или, возможно, предусмотреть возможность загрузки логов в отдельный контейнер хранилища.

При реализации логирования необходимо ограничить максимальный размер логфайлов, т. к. нельзя допускать переполнение накопителя в компьютере пользователя избыточными логами.

Потенциальные технические улучшения

- Экономия дискового пространства у пользователя.

  Проекты, создаваемые в программах для 3D-моделирования, могут занимать гигабайты дискового пространства. Для экономии сетевого трафика и уменьшения времени, затраченного на передачу файлов, неплохо было бы предусмотреть их предварительное архивирование. Однако создание архива также требует дискового пространства, которого тоже может не хватить. Для предотвращения таких случаев следует использовать потоковую архивацию, которая позволяет архивировать данные частями и тут же отправлять заархивированные данные во внешнее хранилище. В этом случае на компьютере пользователя не создается занимающего место промежуточного архива.

  Другой вариант экономии пространства — синхронизация файлов в отдельный каталог на внешнем хранилище. Такая оптимизация полезна, когда нужно обновить состояние проекта во внешнем хранилище до актуального. В этом случае необходимо определить файлы, которые на сервере отличаются от имеющихся на компьютере пользователя, и загрузить только их без использования архивации.

- Расширение функциональности вспомогательной программы.

  Параллельно с задачей копирования проектов нередко существует задача создания результата (рендера) на основе файлов проекта. Рендер результата занимает долгое время и может потреблять большое количество вычислительных ресурсов. Поэтому рендер зачастую производится не на компьютере пользователя, а в удаленных кластерах.

  Это позволяет предположить, что, помимо определения списка файлов проекта, можно также определять настройки рендера и передавать их не просто во внешнее хранилище, а в отдельную систему, которая по заданным настройкам и пути до файлов проекта запускает рендер. В этом случае вспомогательная программа становится не просто прослойкой между расширениями и хранилищем, а полноценным клиентом сервиса рендера и может отслеживать выполнение заданий.

Заключение

В этой главе мы рассмотрели пример архитектуры расширений для семейств программных продуктов. Вот его основные особенности:

- расширения работают по правилам основного программного продукта, поэтому доступные интерфейсы, библиотеки и языки программирования могут сильно различаться;
- при разработке расширений, которые могут быть актуальны для множества схожих программных продуктов, добавление дополнительного слоя абстракции — это необходимость, которая позволяет не реализовывать одну логику на разных языках программирования.

Заключение

В этой книге мы наметили путь через ключевые концепции, принципы и лучшие практики, помогающие создавать продукты, которые решают проблемы людей.

В условиях стремительного развития IT-индустрии становится очевидным, что хорошее проектирование информационных систем требует не только глубоких технических знаний, но и умения адаптироваться к изменениям, мыслить стратегически и эффективно работать в команде. Эффективное проектирование — это не только процесс разработки архитектур и алгоритмов, но и искусство создания систем, которые соответствуют реальным потребностям бизнеса и пользователей. Информационные системы становятся неотъемлемой частью современного общества, и от качества их проектирования зависит устойчивость и эффективность нашей цифровой среды.

Мы надеемся, что материалы этой книги помогут вам углубить свои знания, расширить профессиональные горизонты и вдохновят вас на новые проекты.

Предметный указатель

F

Feature-Driven Development 26, 27

G

GeoDNS 174

H

HMAC-хеш 186

K

Kubernetes 99, 100

R

Rational Unified Process (RUP) 16

W

WebSocket-соединение 163

А

Аварийные игры 114
Алгоритмы балансировки нагрузки 103
Анализ кода 112
Архивирование файлов 200
Аудит-логи 113
Аутентификация 152, 168

Б

База данных 57
Балансировщик нагрузки 103, 107
Бизнес-аккаунты 180
Боты 180
Брокер сообщений 62, 63

В

Версионирование 116, 117
Вертикальное масштабирование 91
Внедрение зависимостей 54
Водопадная модель 13, 14
Восстановление после катастроф 122
Выделенный сервер 93

Г

Гибкие методологии 23, 24
Горизонтальное масштабирование 92, 153

Д

Двухфазный протокол согласования 40
Диаграмма
◊ классов 20
◊ последовательности 22, 23
◊ прецедентов 22
◊ развертывания 21, 22
◊ состояний 21
Динамические библиотеки 53

З

Заглушки 83, 86
Закон Амдала 64, 66

И

Инверсия управления 53
Инкременты Scrum 25
Интеграционное тестирование 84

К

Каскадная модель 13, 15
Категории MoSCoW 30
Кеширование 71, 73
Классификация FURPS+ 30
Кластер Nomad 101
Комментирующие сущности UML 19
Контейнеризация 95
Контейнеры Docker 97
Концепция резолверов 71
Коэффициент готовности 120

Л

Логирование 109–111

М

Межпроцессное взаимодействие 68
Механизм
◊ chroot 96
◊ Jail 96
Миграция схемы баз данных 115
Микросервисная архитектура 34, 35, 159
Микроядерная архитектура 35, 36, 132
Многопоточность 65
Многоуровневая архитектура 37
Модель DBaaS 94
Модульное тестирование 83, 84
Мониторинг 110–112, 155, 193
Монолитная архитектура 33, 34
Мьютексы 65

Н

Набор библиотек FFmpeg 134
Надежность 119
Непрерывная доставка 87–89
Непрерывная интеграция 88
Непрерывное развертывание 89
Нефункциональные требования 31

О

Облачный сервер 93
Обнаружение сервисов 101
Объектное хранилище 94, 95
Оповещения 112
Отказоустойчивость 119
Отношения UML 19

П

Пайплайны 90
Пакетные менеджеры 138
Парадигма MapReduce 69
Параллельные вычисления 64
Пирамида потребностей 29
Поведенческие и дополнительные сущности UML 19
Подход
◊ OLAP 57, 60
◊ OLTP 57–61
◊ REST 38, 39
◊ SAGA 42, 43
◊ User Experience Design (UX) 31
Политики инвалидации 73, 76
Потоки (thread) 165
Потоковая архивация 200
Потребности 29
Принцип
◊ KISS 45–47
◊ SOLID 47, 54
◊ YAGNI 47
Принципы тестирования 80
Псевдопараллелизм 66, 67

Р

Развертывание 87
Распределенная транзакция 39
Распределенные вычисления 68, 70
Реверс-инжиниринг 199
Резервирование 119–122
Резервное копирование 121, 123, 157
Репликация 121

С

Семантическое версионирование 137
Сеть доставки контента (Content Delivery Network, CDN 153, 174

Система управления контентом WordPress 142
Системы управления базами данных 57
Сканеры уязвимостей 113
Сквозное тестирование 84
Событийно-ориентированная архитектура 36
События 36
Сопрограммы 67
Спиральная модель 15, 16
Спринты Scrum 25
Среднее время
◊ восстановления 119
◊ наработки на отказ 119
Статические библиотеки 53
Структурные сущности UML 18

Т

Тестирование 79
Технология
◊ NAT 168
◊ единого входа 152
Требования к программному обеспечению 30
Трехфазный протокол согласования 41

У

Утилита expect 166

Ф

Фреймворк Scrum 24, 25
Функциональные требования 30, 31

Х

Хореография 42

Ю

Юнит-тестирование 83

Я

Язык
◊ AutoIt 166
◊ моделирования UML 18
◊ разметки
▫ BBcode 145
▫ Markdown 143–146, 150, 151