ОБ АВТОРЕ

Баланов Антон Николаевич имеет большой опыт руководства и консультирования в сфере ИТ-технологий. Работал топ-менеджером в крупных компаниях — таких, как Industrial and Commercial Bank of China (КНР), Caravan portal (ОАЭ), Банк ВТБ, Сбербанк России, VK; руководил разработками сервиса Gosuslugi.ru. Имеет степень MBA IT (CIA) и сертификации Microsoft, CompTIA, ISACA, PMI, SHRM, IIBA, HRCI, ISO, Six Sigma (Master Black Belt). Преподавал в следующих вузах и учебных центрах: Российском университете дружбы народов, СберУниверситете, Институте бизнеса и делового администрирования и Центре подготовки руководителей и команд цифровой трансформации (на базе Высшей школы государственного управления РАНХиГС). Автор десятков книг и научно-практических публикаций в профессиональных изданиях. Является советником Российской академии естественных наук.

Широкая эрудиция и глубокие профессиональные компетенции автора в сфере ИТ-технологий позволили ему создать книжную серию «Айтишный университет», один из выпусков которой находится перед вами.

ОГЛАВЛЕНИЕ

ГЛАВА 1

ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ БЭКЕНД-СИСТЕМЫ

ВВЕДЕНИЕ

Глава 1 посвящена проектированию архитектуры бэкенд-системы, что является важным этапом разработки веб-приложений. Проектирование архитектуры позволяет определить структуру, компоненты и взаимодействие между ними для достижения требуемой функциональности и эффективной работы системы. В данной главе мы рассмотрим различные аспекты проектирования архитектуры бэкенд-системы.

Первым шагом в проектировании архитектуры является анализ требований и определение функциональности системы. Важно понять, какие задачи и функции должна выполнять бэкенд-система, а также какие требования предъявляются к ее производительности, масштабируемости, безопасности и другим аспектам. Анализ требований поможет определить основные компоненты и функциональные модули системы.

В процессе проектирования архитектуры необходимо разработать архитектурные паттерны и принципы проектирования. Это поможет определить общую структуру системы, включая разделение на слои, модули и компоненты. Архитектурные паттерны, такие как клиент-сервер, микросервисы, MVC (Model-View-Controller) и другие, предоставляют проверенные и эффективные решения для проектирования бэкенд-системы.

Выбор подходящей архитектурной модели является важным шагом в проектировании бэкенд-системы. Различные модели, такие как одноуровневая, многоуровневая, SOA (Service-Oriented Architecture), Serverless и другие, имеют свои преимущества и ограничения. Важно выбрать модель, кото-

рая наилучшим образом соответствует требованиям системы и предоставляет необходимые возможности для разработки и масштабирования.

Проектирование модулей, компонентов и взаимодействия между ними является финальным шагом в процессе проектирования архитектуры. Необходимо определить функциональные блоки, которые составляют бэкенд-систему, а также описать взаимодействие и зависимости между ними. Важно обеспечить четкую структуру и хорошую модульность системы, чтобы упростить разработку, тестирование и поддержку кода.

Проектирование архитектуры бэкенд-системы является фундаментальным шагом в разработке веб-приложений. Анализ требований, выбор архитектурных паттернов, моделей и проектирование модулей и компонентов позволяют создать эффективную и масштабируемую систему.

АНАЛИЗ ТРЕБОВАНИЙ И ОПРЕДЕЛЕНИЕ ФУНКЦИОНАЛЬНОСТИ СИСТЕМЫ

Анализ требований и определение функциональности системы являются важными этапами в разработке любой системы, включая бэкенд-системы веб-приложений. Этот процесс позволяет понять, что именно должна делать система и какие требования нужно удовлетворить.

Первым шагом в анализе требований является *сбор информации от заинтересованных сторон*. Важно взаимодействовать с заказчиком, конечными пользователями и другими заинтересованными сторонами, чтобы понять их потребности, ожидания и цели относительно системы. Это может быть достигнуто путем проведения интервью, опросов, анализа документации или проведения встреч и рабочих сессий.

После сбора информации необходимо *проанализировать и структурировать полученные данные*. Это включает выделение основных требований и функциональных возможностей системы. Требования могут быть разделены на функциональные (которые определяют, что система должна делать) и не-

функциональные (которые определяют качество и характеристики системы, такие как производительность, безопасность, масштабируемость и т.д.).

Далее необходимо *провести анализ требований* с целью их оценки и приоритизации. Оценка требований может включать оценку их сложности, важности, воздействия на другие компоненты системы и ресурсы, необходимые для их реализации. Это поможет определить, какие требования должны быть реализованы в первую очередь и какие могут быть отложены или изменены.

Важным аспектом анализа требований является их *документирование*. Создание требований в виде спецификаций, диаграмм, пользовательских сценариев и других документов помогает установить общее понимание требований между командой разработки, заказчиком и другими заинтересованными сторонами. Это также служит основой для последующих этапов разработки, таких как проектирование и реализация системы.

Таблица 1.1

Пример с требованиями и определенной функциональностью системы

Требование	***Функциональность***
Регистрация пользователей	Позволяет пользователям создать учетные записи в системе и получить доступ к ее функциональности.
Аутентификация и авторизация	Обеспечивает проверку подлинности пользователей и управление доступом к определенным функциям или ресурсам системы.
Создание, чтение, обновление и удаление данных (CRUD)	Позволяет пользователям создавать, читать, обновлять и удалять данные в системе.

Требование	***Функциональность***
Управление правами доступа	Предоставляет возможность управлять правами доступа пользователей и групп пользователей к определенным функциям или данным.
Отчетность и аналитика	Обеспечивает генерацию отчетов и анализ данных для поддержки принятия решений и мониторинга производительности системы.

Анализ требований является процессом выявления, сбора и анализа требований к системе со стороны заинтересованных сторон. Он позволяет понять, какая функциональность должна быть реализована в системе, чтобы удовлетворить потребности пользователей и бизнес-цели. После этого происходит определение функциональности системы, то есть составление перечня основных возможностей и задач, которые должны быть выполнены системой.

Пример.

Предположим, что мы разрабатываем систему управления проектами. При анализе требований выяснилось, что пользователи должны иметь возможность регистрироваться в системе, чтобы получить доступ к ее функциональности. Таким образом, функциональность системы будет включать регистрацию пользователей. Также требования показали, что система должна обеспечивать аутентификацию и авторизацию пользователей, чтобы проверить их подлинность и управлять доступом к различным функциям системы.

Другое требование состоит в том, чтобы пользователи могли создавать, читать, обновлять и удалять данные в системе. Таким образом, функциональность системы будет включать операции CRUD (Create, Read, Update, Delete) для данных. Кроме того, требуется управление правами доступа, чтобы определить, какие пользователи или группы пользователей имеют доступ к определенным функциям или данным системы. Наконец, система должна обеспечивать возможность генерации

отчетов и анализа данных для поддержки принятия решений и мониторинга производительности системы.

Таким образом, анализ требований и определение функциональности системы позволяют точно определить, какие возможности и задачи должны быть реализованы в системе, чтобы она соответствовала потребностям пользователей и бизнес-целям. Это важный шаг в процессе разработки программного обеспечения, который позволяет создать систему, которая будет эффективно выполнять свои функции и удовлетворять потребности пользователей.

В заключение, анализ требований и определение функциональности системы играют важную роль в успешной разработке бэкенд-системы. Этот процесс позволяет определить, что должна делать система и какие требования нужно удовлетворить. Он также служит основой для последующих этапов разработки, помогая команде разработки создать систему, которая соответствует потребностям и ожиданиям пользователей и заказчика.

РАЗРАБОТКА АРХИТЕКТУРНЫХ ПАТТЕРНОВ И ПРИНЦИПОВ ПРОЕКТИРОВАНИЯ

Разработка архитектурных паттернов и принципов проектирования является важной задачей при разработке программных систем. Правильное применение этих паттернов и принципов позволяет создавать гибкие, масштабируемые и легко поддерживаемые приложения. Давайте рассмотрим несколько популярных архитектурных паттернов и принципов проектирования, а также примеры и таблицу для каждого из них.

1. MVC (Model-View-Controller) — это паттерн, разделяющий приложение на три основных компонента: модель, представление и контроллер.

- Модель представляет данные и бизнес-логику приложения.
- Представление отвечает за отображение данных и взаимодействие с пользователем.

- Контроллер обрабатывает пользовательский ввод, обновляет модель и управляет представлением.

Таблица 1.2

Пример таблицы для паттерна MVC

Компонент	***Описание***
Модель	Хранит данные и реализует бизнес-логику приложения.
Представление	Отображает данные пользователю и обрабатывает ввод.
Контроллер	Обрабатывает пользовательский ввод и управляет моделью.

2. Слоистая архитектура (Layered Architecture) — это архитектурный паттерн, в котором система разделена на слои с определенными функциональными задачами.

- Каждый слой выполняет определенные операции и обрабатывает определенный уровень абстракции.
- Взаимодействие между слоями происходит только через строго определенные интерфейсы.

Таблица 1.3

Пример для слоистой архитектуры

Слой	***Описание***
Представление	Отображение данных и обработка пользовательского ввода.
Бизнес-логика	Обработка бизнес-логики и управление бизнес-процессами.
Доступ к данным	Взаимодействие с базой данных или внешними источниками данных.

3. SOLID — это набор принципов проектирования, которые помогают создавать гибкие и расширяемые системы.

- Принцип единственной ответственности (Single Responsibility Principle) — класс должен иметь только одну причину для изменения.
- Принцип открытости/закрытости (Open/Closed Principle) — классы должны быть открыты для расширения, но закрыты для модификации.
- Принцип подстановки Барбары Лисков (Liskov Substitution Principle) — объекты должны быть заменяемыми своими подтипами без изменения корректности программы.
- Принцип разделения интерфейса (Interface Segregation Principle) — клиенты не должны зависеть от интерфейсов, которые они не используют.
- Принцип инверсии зависимостей (Dependency Inversion Principle) — зависимости должны строиться на абстракциях, а не на конкретных реализациях.

Таблица 1.4

Пример для принципов SOLID

Принцип	***Описание***
Принцип единственной ответственности	Класс должен иметь только одну причину для изменения.
Принцип открытости/закрытости	Классы должны быть открыты для расширения, но закрыты для модификации.
Принцип подстановки Барбары Лисков	Объекты должны быть заменяемыми своими подтипами без изменения корректности программы.
Принцип разделения интерфейса	Клиенты не должны зависеть от интерфейсов, которые они не используют.
Принцип инверсии зависимостей	Зависимости должны строиться на абстракциях, а не на конкретных реализациях.

Эти архитектурные паттерны и принципы проектирования являются основой для создания хорошо структурированных и гибких программных систем. Их правильное применение помогает обеспечить масштабируемость, легкость поддержки и удобство разработки ваших приложений.

ВЫБОР ПОДХОДЯЩЕЙ АРХИТЕКТУРНОЙ МОДЕЛИ ДЛЯ БЭКЕНД-СИСТЕМЫ

Выбор подходящей архитектурной модели для бэкенд-системы играет важную роль в обеспечении эффективности, масштабируемости и надежности системы. При выборе архитектурной модели для бэкенд-системы необходимо учитывать требования проекта, ожидаемую производительность, масштабируемость, доступность и другие факторы. Вот некоторые популярные архитектурные модели для бэкенд-систем.

1. **Монолитная архитектура**

- *Описание*. Все компоненты системы объединены в одном приложении или сервисе.
- *Преимущества*. Простота разработки и развертывания, легкая коммуникация между компонентами, возможность быстрого масштабирования.
- *Недостатки*. Ограниченная масштабируемость, сложность поддержки и обновления, отсутствие изоляции компонентов.

2. **Микросервисная архитектура**

- *Описание*. Система состоит из набора независимых сервисов, каждый из которых выполняет свою узкую функцию.
- *Преимущества*. Легкая масштабируемость, изоляция компонентов, независимая разработка и развертывание, улучшенная гибкость и масштабируемость.
- *Недостатки*. Увеличенная сложность управления множеством сервисов, необходимость обеспечения коммуникации и согласованности между сервисами.

3. **Серверно-ориентированная архитектура**

- *Описание*. Большая часть бизнес-логики находится на сервере, а клиентские приложения служат только для отображения данных и взаимодействия с сервером.
- *Преимущества*. Упрощенное развертывание клиентских приложений, централизованное управление бизнес-логикой и данными, повышенная безопасность.

- *Недостатки*. Ограниченные возможности работы без подключения к серверу, возможные проблемы с производительностью и откликом.

Таблица 1.5

Примеры архитектурных моделей и их особенностями

Архитектурная модель	***Описание***	***Преимущества***	***Недостатки***
Монолитная архитектура	Все компоненты системы объединены в одном приложении или сервисе	Простота разработки и развертывания	Ограниченная масштабируемость
Микросервисная архитектура	Система состоит из набора независимых сервисов, каждый из которых выполняет свою узкую функцию	Легкая масштабируемость, изоляция компонентов	Увеличенная сложность управления множеством сервисов
Серверно-ориентированная архитектура	Большая часть бизнес-логики находится на сервере, а клиентские приложения служат только для отображения данных и взаимодействия с сервером	Упрощенное развертывание клиентских приложений, централизованное управление бизнес-логикой и данными	Ограниченные возможности работы без подключения к серверу

Примеры.

1. Монолитная архитектура.

- *Пример*. Традиционное веб-приложение, в котором все компоненты, такие как фронтенд, бэкенд, база данных, находятся в одном монолитном приложении.

- *Описание*. Весь код и функциональность приложения находятся внутри одного проекта или приложения, которое разворачивается на сервере. Все запросы и обработка данных выполняются внутри этого приложения.

2. *Микросервисная архитектура.*

- *Пример*. Интернет-магазин, где каждая функциональность, такая как управление продуктами, обработка заказов, оплаты, доставка, реализована в виде отдельного сервиса.
- *Описание*. Каждый сервис выполняет свою узкую функцию и может быть независимо разработан, развернут и масштабирован. Сервисы могут взаимодействовать друг с другом через API.

3. *Серверно-ориентированная архитектура.*

- *Пример*. Приложение для управления задачами, где весь бизнес-логика находится на сервере, а клиентское приложение просто отображает данные и отправляет запросы на сервер.
- *Описание*. Клиентские приложения выполняют минимальную функциональность и полагаются на сервер для обработки данных и бизнес-логики. Все обновления и изменения происходят на сервере, а клиентское приложение отображает только обновленные данные.

Выбор подходящей архитектурной модели зависит от требований проекта и его особенностей. Важно учитывать факторы, такие как масштабируемость, разделение ответственности, гибкость разработки и обновления, а также сопровождаемость системы.

ПРОЕКТИРОВАНИЕ МОДУЛЕЙ, КОМПОНЕНТОВ И ВЗАИМОДЕЙСТВИЯ МЕЖДУ НИМИ

Проектирование модулей, компонентов и взаимодействия между ними является важной частью создания эффективной и масштабируемой бэкенд-системы. Этот процесс позволяет определить структуру системы, разделить ее на логические