****МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ

УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(национальный исследовательский университет)»

**Институт №3\_\_\_\_\_\_Кафедра \_\_\_\_\_304 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Направление подготовки \_09.03.04 «Программная инженерия» Группа\_3О-\_**

**Квалификация (степень) \_Бакалавр \_\_\_\_\_\_\_**

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**

**БАКАЛАВРА**

На тему: «Разработка пилотного проекта веб-сервиса для взаимодействия пользователей портала государственных услуг г. Москвы».

Автор квалификационной работы \_ ( )

(Фамилия, Имя, Отчество)

Руководитель ( )

(Фамилия, Имя, Отчество)

Рецензент ( )

(Фамилия, Имя, Отчество)

**К з а щ и т е д о п у с т и т ь**

Зав. кафедрой 304 Брехов О.М. ( )

(Фамилия, Имя, Отчество)

“\_\_\_”\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_\_\_г.

Москва 2023 г.

[ВВЕДЕНИЕ 3](#_Toc134201691)

[1. ЗАДАЧА ​РАЗРАБОТКИ ПРИЛОЖЕНИЯ-СЕРВЕРА С ИСПОЛЬЗОВАНИЕМ МИКРОСЕРВИСНОЙ АРХИТЕКТУРЫ И СОБЫТИЙНОГО ПОДХОДА К ХРАНЕНИЮ ДАННЫХ. 5](#_Toc134201692)

[1.1. Постановка и анализ задачи 5](#_Toc134201693)

[1.1.1. Назначение приложения и предметная область 5](#_Toc134201694)

[1.1.2. Выбор средств разработки 6](#_Toc134201695)

[1.1.2.1. Выбор целевой операционной системы 6](#_Toc134201696)

[1.1.2.2. Выбор языка программирования 7](#_Toc134201697)

[1.1.2.3. Выбор программной платформы 8](#_Toc134201698)

[1.1.2.4. Выбор клиентской программной платформы 11](#_Toc134201699)

[2. ИЗУЧЕНИЕ АРХИТЕКТУРЫ МИКРОСЕРВИСНЫХ ПРИЛОЖЕНИЙ, СПОСОБОВ КОММУНИКАЦИИ МЕЖДУ НИМИ И ХРАНЕНИЯ ДАННЫХ В ВИДЕ СОБЫТИЙ. 12](#_Toc134201700)

[2.1. Архитектура микросервисных приложений 12](#_Toc134201701)

[2.1.1. Внешний программный интерфейс 14](#_Toc134201702)

[2.1.2. Шина сообщений 14](#_Toc134201703)

[2.1.3. Микросервисы 18](#_Toc134201704)

[2.2. Коммуникация между микросервисами 19](#_Toc134201705)

[2.2.1. Запрос данных между микросервисами 19](#_Toc134201706)

[2.2.1.1. Асинхронная коммуникация 20](#_Toc134201707)

[2.2.1.2. Общие базы данных 20](#_Toc134201708)

[2.2.1.3. Использование кэша 21](#_Toc134201709)

[2.2.2. Транзакции 21](#_Toc134201710)

[2.3. Хранение данных в виде событий 23](#_Toc134201711)

[2.3.1 Event Sourcing 24](#_Toc134201712)

[2.3.2 CQRS 24](#_Toc134201713)

[2.3.2.1 Разделение данных 25](#_Toc134201714)

[3. ПРОЕКТИРОВАНИЕ ПРИЛОЖЕНИЙ ДЛЯ ФУНКЦИЙ РЕСТОРАНА 26](#_Toc134201715)

[3.1. Проектирование системы 26](#_Toc134201716)

[3.2 Проектирование внешнего программного интерфейса 28](#_Toc134201717)

[3.3 Проектирование микросервисов 30](#_Toc134201718)

[4. РАЗРАБОТКА ПРИЛОЖЕНИЙ ДЛЯ ФУНКЦИЙ РЕСТОРАНА 32](#_Toc134201719)

[4.1. Установка и настройка необходимых компонентов 32](#_Toc134201720)

[4.1.1. Установка среды выполнения 32](#_Toc134201721)

[4.1.2. Установка серверной программной платформы 33](#_Toc134201722)

[4.1.3. Установка клиентской программной платформы 34](#_Toc134201723)

[4.2. Разработка приложений 34](#_Toc134201724)

[4.2.2. Разработка доменных сущностей и сообщений 34](#_Toc134201725)

[4.2.3. Разработка сервиса для работы с журналом событий 40](#_Toc134201726)

[4.2.4. Разработка сервиса для отправки сообщений 44](#_Toc134201727)

[4.2.5. Обновление базы данных для чтения 45](#_Toc134201728)

[4.2.6. Разработка микросервиса товаров 46](#_Toc134201729)

[4.2.7. Разработка микросервиса оплаты 50](#_Toc134201730)

# ВВЕДЕНИЕ

Традиционно серверные приложения разрабатывались с использованием монолитного подхода к проектированию архитектуры приложения.

Монолитная архитектура - это подход к разработке приложения, при котором все компоненты приложения находятся в одном исполняемом файле или модуле, при этом приложение реалезуется на одном языке программирования

Это делает процесс разработки, тестирования и развертывания приложения относительно простым и удобным. Все компоненты приложения могут легко общаться между собой, что упрощает обработку запросов и увеличивает производительность.

Однако, монолитная архитектура имеет некоторые недостатки. При увеличении размера приложения становится все сложнее разрабатывать, тестировать и развертывать новые функции.

Другим недостатком монолитной архитектуры является тот факт, что она не является очень гибкой и масштабируемой, что может стать проблемой при увеличении нагрузки на приложение.

В целом, монолитная архитектура является удобным и простым подходом к разработке сервера, но при создании крупных и сложных систем может ограничивать возможности разработчиков и увеличивать затраты на поддержку и развитие приложения.

Альтернативным подходом к разработке приложения является микросервисная архитектура.

Она позволяет создавать сложные системы из небольших, независимых и легко масштабируемых сервисов. Такой подход имеет ряд преимуществ перед монолитной архитектурой, так как позволяет повысить гибкость, масштабируемость и отказоустойчивость системы.

Другим важным аспектом, который будет рассмотрен в данной работе является способ хранения состояния приложения. При традиционном подходе в базе данных сохраняется только последнее состояние приложения, что ведет к потере данных о предыдущих состояниях приложения и невозможности восстановить предыдущее состояние приложения. Для борьбы с этой проблемой существует подход Источники событий, который предполагает хранение истории изменений состояния приложения в виде событий. Каждое изменение состояния приложения записывается в журнал событий в хронологическом порядке. Используя этот журнал, можно воссоздать состояние системы на любой момент времени.

В данной дипломной работе будет рассмотрена микросервисная архитектура вместе с подходом Источники событий на примере разработки сервера для веб-приложения. Будут изучены основные принципы построения микросервисов, архитектурные шаблоны и инструменты, необходимые для создания и управления такой системой. Также будет проведен анализ преимуществ и недостатков данного подхода и рассмотрены сценарии его применения в реальных проектах.

Целью данной работы является изучение и анализ микросервисной архитектуры и хранения состояния приложения в виде событий, а также разработка и реализация сервера с использованием данного подхода.

# ЗАДАЧА ​РАЗРАБОТКИ ПРИЛОЖЕНИЯ-СЕРВЕРА С ИСПОЛЬЗОВАНИЕМ МИКРОСЕРВИСНОЙ АРХИТЕКТУРЫ И СОБЫТИЙНОГО ПОДХОДА К ХРАНЕНИЮ ДАННЫХ.

## Постановка и анализ задачи

### Назначение приложения и предметная область

Разрабатываемое программное обеспечение предназначено для управления рестораном общественного питания и предоставляет функционал необходимый для работы с заказами, их сборки и приготовления.

Предметной областью веб-сервера является сфера услуг общественного питания и отображает ее с точки зрения как работников ресторана, так и его посетителей.

Целью разработки данного приложения является анализ и реализация основных принципов микросервисной архитектуры приложения и так же анализ и реализация событийного подхода к хранению данных, его достоинств и недостатков.

Ставится задача реализации следующего функционала в административной панели управления:

* Создание/редактирование товаров
* Создание/редактирование меню
* Создание, сборка и выдача, отслеживание статусов заказов
* Отслеживание недостающих товаров для сборки заказов
* Отслеживание запасов товаров
* Создание и выполнение запросов на приготовление товаров

Так как пользователи не должны иметь доступ к жизненно важным функциям приложения, то для них нужно реализовать пользовательскую панель управления, к которой предъявляются следующие функциональные требования:

* Просмотр меню
* Создание и отслеживание статусов заказов

Так же для более наглядного отображения событийного подхода к хранению состояния приложения необходимо предусмотреть следующий функционал в административной панели управления:

1. Отображение журнала событий для
   1. Заказов
   2. Товаров
   3. Меню
   4. Запасов товаров
   5. Запросов на приготовление товаров
2. Отображение списка событий, связанных с запрашиваемыми объектами для
   1. Заказов
   2. Товаров
   3. Меню
   4. Запасов товаров
   5. Запросов на приготовление товаров

### Выбор средств разработки

Выбор средств разработки основывается на удобстве реализации выбранных подходов и приспособленности средств к поставленным целям и функционалу.

### Выбор целевой операционной системы

Операционная система (ОС) - это программное обеспечение, которое управляет аппаратными и программными ресурсами компьютера и обеспечивает работу других приложений на этом компьютере.

ОС обеспечивает интерфейс между пользователем и компьютером, позволяет запускать программы, управляет доступом к ресурсам компьютера, обеспечивает безопасность и защиту данных, управляет памятью, обрабатывает ввод и вывод данных и выполняет множество других функций.

Для разработки сервера решено использовать ОС Windows 10, среди ее преимуществ:

1. Широкое распространение: Windows является одной из самых популярных операционных систем в мире, что делает ее доступной для использования и поддержки.
2. Удобство использования: Windows имеет простой и интуитивно понятный интерфейс, что делает ее легкой в освоении даже для новых пользователей.
3. Большой выбор программного обеспечения: Windows поддерживает большое количество программного обеспечения, включая популярные приложения для офисной работы, развлечений и игр.
4. Поддержка различных устройств: Windows поддерживает большой выбор аппаратного обеспечения, включая настольные компьютеры, ноутбуки, планшеты и смартфоны.

### Выбор языка программирования

При разработке сервера стоит обратить внимание на языки программирования, которые наиболее распространены в сфере веб-разработки и обладают большой системой библиотек для уменьшения времени разработки приложения.

Одним из лидеров в сфере вэб-разработки является язык Javascript. Хотя изначально Javascript предназначался для выполнения в браузере, сейчас он может быть выполнен и вне браузера при помощи Node.js.

Node.js - это среда выполнения JavaScript, основанная на движке V8, который используется в браузере Google Chrome. Она позволяет запускать JavaScript-код на сервере, что делает ее популярной для создания масштабируемых и высокопроизводительных веб-приложений. Среди преимуществ Node.js:

1. Масштабируемость: Node.js использует модель событийного цикла, что делает ее масштабируемой и позволяет обрабатывать большое количество запросов одновременно.
2. Удобство разработки: Node.js позволяет использовать JavaScript как на стороне сервера, так и на стороне клиента, что упрощает разработку веб-приложений и уменьшает время разработки.
3. Большое количество модулей: Node.js имеет огромное количество модулей, которые позволяют быстро и удобно добавлять функциональность в приложение.
4. Большое сообщество: Node.js имеет большое и активное сообщество разработчиков, которое предоставляет поддержку и помощь в разработке приложений.

### Выбор программной платформы

С развитием языков программирования, разработчики обнаружили повторяющиеся алгоритмы, шаблоны проектирования и методы, которые раз за разом повторялись в разных проектах или вообще превратились в стандарт разработки. Нецелесообразно с точки зрения разработчика тратить время на копирование или переписывание одних и тех же методов. Для решения этой проблемы были созданы программные платформы или же frameworks.

Framework (фреймворк) - это набор инструментов, библиотек и правил, предназначенных для разработки приложений, упрощающих и ускоряющих процесс создания программного обеспечения. Он предоставляет разработчику готовые компоненты, которые можно использовать для создания приложения без необходимости писать весь код с нуля. Framework может содержать готовые решения для обработки запросов пользователя, работы с базами данных, управления сессиями, авторизации и многое другое. Он также может включать в себя стандарты и методологии разработки, что облегчает совместную работу нескольких разработчиков над проектом.

Рассмотрим положительные стороны использования программной платформы:

1. Программная платформа обычно навязывает файловую структуру проекта, что позволяет повысить читаемость проекта и уменьшает время ознакомления с проектом новых разработчиков.
2. Программная платформа содержит протестированный и универсальный набор инструментов и библиотек для разработки приложения
3. Программная платформа предлагает унифицированный способ организации архитектуры приложения
4. Фреймворк может быть расширен при помощи модулей или библиотек созданных сообществом, что еще больше увеличивает его универсальность

Отрицательные стороны использования фреймворков:

1. Ограничения в выборе технологий: Каждый фреймворк имеет свои собственные правила, структуру и набор инструментов, что может ограничить разработчиков в выборе технологий, которые они хотят использовать.
2. Избыточность кода: Иногда фреймворки могут предоставлять слишком много функциональности, которая не используется в конкретном проекте, что приводит к избыточности кода и увеличивает размер приложения.
3. Снижение гибкости: Фреймворки могут быть спроектированы для решения конкретных задач, что может снижать гибкость их использования в других задачах. Это может привести к тому, что разработчики не смогут использовать фреймворк для решения своих специфических задач и придется писать код с нуля.

В качестве программной платформы я выбрал Nest.js. Nest.js – фреймворк для языка JavaScript на платформе Node.js

Он предназначен для создания масштабируемых и эффективных серверных приложений, использующий принципы модульности и многократного использования кода. Он основан на другом фреймворке Express.js и предоставляет множество дополнительных функций и инструментов для обработки HTTP-запросов, работы с базами данных, реализации аутентификации и авторизации, а также других типичных задач серверной разработки. NestJS также поддерживает шаблоны проектирования, такие как Dependency Injection (DI), что упрощает разработку и тестирование приложений. В целом, NestJS позволяет разработчикам создавать высокопроизводительные и модульные серверные приложения с использованием современных инструментов и технологий.

Среди других преимуществ NestJS:

1. Модульность: NestJS поддерживает модульную архитектуру, которая позволяет разбить приложение на небольшие и независимые блоки кода, что упрощает разработку и поддержку приложения.

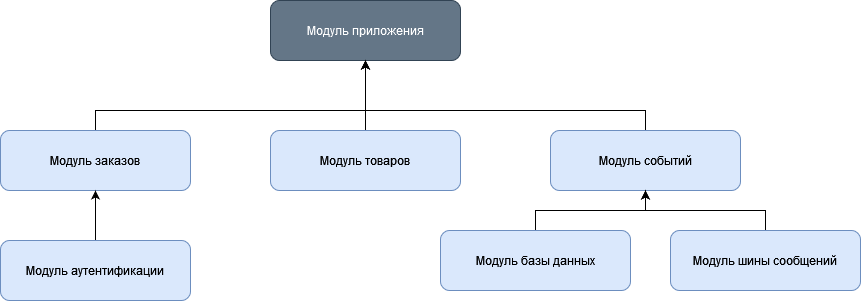


Рисунок Модульная архитектура NestJS

1. Поддержка стандартных протоколов и библиотек: NestJS предоставляет нативную поддержку протоколов и библиотек, таких как Express.js, WebSocket, gRPC, Passport.js и многих других, что упрощает разработку приложений.
2. Шаблоны проектирования: NestJS поддерживает шаблоны проектирования, такие как Dependency Injection (DI), что позволяет создавать гибкие и расширяемые приложения.
3. Автоматическая генерация кода: NestJS предоставляет инструменты для автоматической генерации кода, такие как генератор контроллеров и сервисов, что упрощает и ускоряет разработку приложений.
4. Тестирование: NestJS предоставляет удобные инструменты для тестирования приложений, такие как интеграционные и юнит-тесты, что позволяет создавать надежные и стабильные приложения.
5. Расширяемость: NestJS легко расширяем и настраиваем, что позволяет создавать приложения любой сложности и масштаба.

### Выбор клиентской программной платформы

Для удобства сервер должен обладать графическим интерфейсом, который позволит пользователям взаимодействовать с его функциями.

Реализовывать клиентскую часть было решено в формате вэб-приложения, так как основным преимуществом веб-приложений является их доступность из любого устройства с доступом к интернету, будь то компьютер, смартфон или планшет. Пользователи могут получать доступ к веб-приложениям через браузер без необходимости устанавливать какое-либо дополнительное программное обеспечение. Это делает веб-приложения удобными для использования в различных условиях и местах, что является особенно важным для мобильных устройств.

Кроме того, веб-приложения имеют возможность работать на удаленных серверах, что обеспечивает масштабируемость и отказоустойчивость. Компании могут развернуть веб-приложения в облачной среде и предоставлять доступ к ним множеству пользователей одновременно.

Также, веб-приложения могут легко обновляться без необходимости установки новой версии на каждом устройстве, так как они работают через браузер и могут быть обновлены на сервере. Это удобно для разработчиков и обеспечивает пользователей последними версиями приложения.

Так как ранее мы уже выяснили зачем нужны программные платформы, то можно сразу перейти к выбору.

Для разработки вэб-приложений на языке JavaScript существует несколько лидирующих фреймворков: React, Vue и Angular.

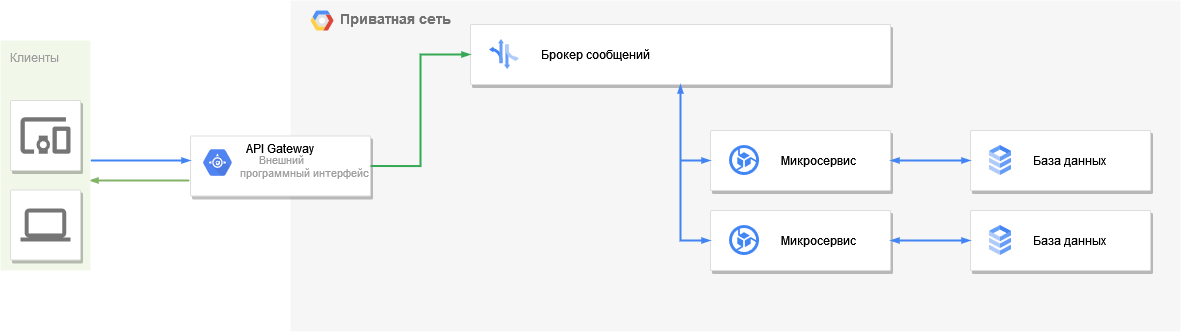
На данный момент они находятся на примерно равном уровне по доступному функционалу и поддержке разработчиков. Но я выбрал React по нескольким причинам:

* Широкое сообщество: React имеет широкое сообщество разработчиков, которые постоянно создают новые компоненты и инструменты для разработки, что делает его очень популярным.
* Простота использования: React очень прост в использовании и не требует большого количества времени на изучение, особенно если вы уже знакомы с JavaScript.
* Поддержка большого количества инструментов для разработки: React имеет большое количество инструментов для разработки, таких как Webpack, Babel и другие, что делает процесс разработки более эффективным и простым в использовании.

# ИЗУЧЕНИЕ АРХИТЕКТУРЫ МИКРОСЕРВИСНЫХ ПРИЛОЖЕНИЙ, СПОСОБОВ КОММУНИКАЦИИ МЕЖДУ НИМИ И ХРАНЕНИЯ ДАННЫХ В ВИДЕ СОБЫТИЙ.

## Архитектура микросервисных приложений

Микросервисная архитектура приложений в своей сути предполагает разбиение большого, монолитного приложения на несколько меньших приложений, каждое из которых выполняет свою функцию в общей системе.



Как правило эта система состоит из следующих компонентов:

* Внешний программный интерфейс: Он служит для взаимодействия внешнего мира с системой. при поступлении внешнего запроса, он отправляет запрос к нужному микросервису и возвращает ответ клиенту. На внешних интерфейсах удобно реализовывать проверку прав пользователя на доступ к системе, чтобы этим не занимался каждый сервер в отдельности.
* Шина сообщений(брокер сообщений): Она служит для коммуникации между всеми компонентами системы. Каждый сервис может отправлять туда сообщения с запросами или ответами и забирать сообщения предназначенные для себя. Брокер сообщений удобно использовать для оповещения сразу нескольких сервисов при помощи одного сообщения, так же некоторые брокеры умеют хранить историю сообщений и повторять отправку сообщений, если она завершилась неудачно.
* Несколько микросервисов с их базами данных: Каждый микросервис является изолированным приложением со своей личной базой данных, которая хранит информацию нужную для его работы. Если другому сервису нужна эта информация, то он должен запросить ее.

Конфигурация может меняться исходя из требований к разрабатываемой системе. Теперь рассмотрим каждый компонент подробнее.

### Внешний программный интерфейс

API Gateway - это слой абстракции между клиентами приложения и микросервисами, которые предоставляют функциональность для приложения. Он предоставляет:

единый точку входа (entry point) для всех клиентов, и

выполняет функции маршрутизации (routing),

балансировки нагрузки (load balancing), обработки запросов (request handling), проверки аутентификации (authentication), авторизации (authorization) и кеширования (caching).

API Gateway помогает упростить архитектуру микросервисов, делая ее более гибкой и масштабируемой. Он также позволяет скрыть детали реализации каждого микросервиса от клиентов приложения, обеспечивая лучшую безопасность и контроль над доступом к функциональности приложения.

API Gateway может использоваться для различных целей, таких как:

* Агрегация данных из нескольких источников (микросервисов) в один ответ для клиентов приложения
* Предоставление единого API-интерфейса для нескольких микросервисов
* Обеспечение безопасности и контроля доступа к функциональности приложения
* Улучшение производительности приложения путем кеширования данных
* Управление и мониторинг микросервисов.

### Шина сообщений

Шина сообщений (Message Bus) - это паттерн архитектуры, который используется для обмена сообщениями между различными компонентами (например, микросервисами) в распределенной системе. Он обеспечивает надежную и асинхронную коммуникацию между компонентами, что делает систему более гибкой, масштабируемой и отказоустойчивой.

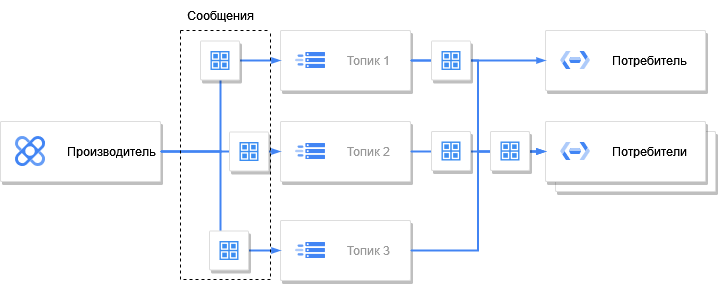


Рисунок Шина сообщений

Шина сообщений состоит из нескольких элементов:

1. Очереди сообщений (Message Queues) - это механизмы, которые позволяют различным компонентам отправлять и получать сообщения друг от друга. Очереди сообщений могут использоваться для обеспечения гарантированной доставки сообщений, а также для обеспечения надежности и отказоустойчивости системы.
2. Топики (Topics) - это механизмы, которые позволяют отправлять сообщения определенной тематики всем компонентам, которые подписались на эту тему. Топики могут использоваться для обмена сообщениями между компонентами, которые не знают друг о друге напрямую.
3. Роутеры (Routers) - это компоненты, которые позволяют маршрутизировать сообщения между различными очередями и топиками, чтобы обеспечить оптимальную коммуникацию между компонентами.
4. Производители(Producers) - это компоненты, которые создают и отправляют сообщения в брокер, используя определенный протокол и интерфейс.
5. Потребители(Consumers) - это компоненты, которые получают и обрабатывают сообщения, которые были отправлены в шину. Потребители могут подписываться на определенные топики или очереди, чтобы получать только определенные типы сообщений.

Шина сообщений имеет множество преимуществ, среди которых:

- Гибкость: различные компоненты могут обмениваться сообщениями независимо друг от друга, что делает систему более гибкой и масштабируемой.

- Надежность: шина сообщений может обеспечить гарантированную доставку сообщений, что делает систему более надежной и отказоустойчивой.

- Отказоустойчивость: если один компонент выходит из строя, другие компоненты могут продолжать работать нормально, что обеспечивает отказоустойчивость всей системы.

- Асинхронность: компоненты могут работать асинхронно, обмениваясь сообщениями в фоновом режиме, что позволяет улучшить производительность и масштабируемость системы.

- Масштабируемость: шина сообщений позволяет легко добавлять новые компоненты и масштабировать систему в соответствии с растущими потребностями

Одними из самых популярных шин сообщений являются Apache Kafka, RabbitMQ и Apache ActiveMQ. Хотя они по сути предлагают схожий функционал, но в них есть некоторые различия:

* Apache Kafka - это высокопроизводительная, масштабируемая и распределенная платформа для обработки, хранения и передачи потоковых данных (стриминговых данных). Она обеспечивает возможность эффективной обработки больших объемов данных в режиме реального времени, используя асинхронную передачу данных между производителями и потребителями. Kafka поддерживает механизмы гарантированной доставки сообщений и репликацию данных на несколько узлов, обеспечивая высокую отказоустойчивость и надежность работы системы. Kafka использует принцип "публикация-подписка" (publish-subscribe) и хранит данные в виде упорядоченных лент (topic partitions). Благодаря своей высокой производительности и масштабируемости, Kafka используется в различных приложениях, включая аналитику данных, обработку событий, мониторинг и управление ресурсами, обработку потоковой медиа-информации и многое другое.
* RabbitMQ является брокером сообщений, который использует протокол AMQP (Advanced Message Queuing Protocol) для обмена сообщениями. Он предоставляет гибкую и масштабируемую платформу для обмена сообщениями между приложениями. RabbitMQ поддерживает различные режимы доставки сообщений, включая точную доставку (exactly-once delivery) и доставку сообщений с подтверждением получения (acknowledgement-based delivery).
* Apache ActiveMQ - это open-source брокер сообщений, который обеспечивает передачу сообщений между различными приложениями и системами. Он реализует широкий спектр протоколов и технологий связи, включая JMS, AMQP, MQTT, STOMP, OpenWire и другие. Apache ActiveMQ поддерживает ряд функций, таких как транзакционность, очереди, темы, маршрутизацию сообщений и т. д. Он имеет расширяемую архитектуру и может быть интегрирован с различными системами.

В дальнейшей разработки приложения будет использоваться Apache Kafka в качестве брокера сообщений из-за его преимуществ:

1. Масштабируемость: Kafka способен обрабатывать огромные объемы данных и обеспечивает высокую производительность.
2. Высокая отказоустойчивость: Kafka предоставляет репликацию данных на несколько узлов, что позволяет сохранять данные в случае отказа какого-либо узла.
3. Гибкость: Kafka позволяет настраивать различные параметры для поддержки специфических потребностей.
4. Надежность: Kafka имеет множество механизмов обеспечения целостности данных и предотвращения потерь сообщений.
5. Расширяемость: Kafka позволяет добавлять новые функции и возможности при необходимости, что обеспечивает гибкость и масштабируемость системы.

### Микросервисы

Теперь рассмотрим основные строительные блоки микросервисной архитектуры – сами микросервисы.

Микросервис - небольшое приложение которое отвечает за отдельную функциональность. Каждый микросервис должен выполнять только одну задачу, и все сервисы должны взаимодействовать между собой через API. Для этого часто используются технологии, такие как REST, GraphQL, gRPC и другие. Каждый микросервис может быть написан на разных языках программирования, использовать разные базы данных и иметь свои API, иметь разную архитектуру и подходы к проектированию приложения.

Давайте рассмотрим примеры микросервисов:

1. Сервис авторизации - обеспечивает аутентификацию и авторизацию пользователей, может включать в себя такие функции, как регистрация, сброс пароля и управление ролями.
2. Сервис управления контентом - предоставляет API для создания, чтения, обновления и удаления контента на сайте, может включать в себя различные типы контента, такие как статьи, фотографии и видео.
3. Сервис обработки платежей - обрабатывает платежи и может включать в себя функции, такие как проверка карты, обработка платежа и хранение истории платежей.
4. Сервис управления заказами - управляет заказами и может включать в себя функции, такие как добавление товаров в корзину, оформление заказа, оплата и отслеживание статуса заказа.
5. Сервис уведомлений - предоставляет API для отправки уведомлений пользователям, таких как электронные письма, SMS-сообщения и уведомления в мобильном приложении.

Эти сервисы могут работать вместе и обмениваться данными через API, образуя распределенную систему, в которой каждый сервис занимается определенным функционалом и может масштабироваться отдельно от других сервисов.

## Коммуникация между микросервисами

Каждый микросервис – изолированное приложение. Он может иметь свою базу данных, что делает их более автономными и изолированными друг от друга. Однако, это может приводить к проблемам согласованности данных и необходимости реализации транзакций на уровне приложения.

### Запрос данных между микросервисами

Так микросервисы являются изолированными приложениями, то если одному микросервису необходимы данные другого сервиса, появляется проблема доступа данных. Для ее решения существует несколько способов:

1. Асинхронная коммуникация: один микросервис отправляет сообщение другому микросервису, используя шину сообщений, и получает ответ позже. Этот подход позволяет микросервисам работать параллельно и не блокировать друг друга, но может быть сложным в реализации и отладке.
2. Базы данных: микросервисы могут использовать общую базу данных для доступа к данным друг друга. Этот подход может быть простым, но может привести к проблемам согласованности данных и зависимости микросервисов друг от друга.
3. Использование кэшей: микросервисы могут использовать кэши для доступа к данным друг друга. Этот подход может повысить производительность и уменьшить нагрузку на сеть, но может привести к проблемам согласованности данных и требует дополнительной конфигурации и управления кэшами.

### Асинхронная коммуникация

При асинхронной коммуникации, сервис, который нуждается в данных посылает запрос через шину сообщений другому сервису и ждет ответ. Из-за асинхронного подхода второй сервис может обработать запрос, когда у него появится возможность, что повышает отказоустойчивость. Затем он выполняет необходимые операции, получает данные из базы данных, форматирует их и возвращает ответ через шину сообщений. Данный способ является наиболее практичным из-за простоты реализации и устойчивости к рассинхронизации данных.

### Общие базы данных

Одним из способов решения проблемы запроса данных, может быть использование одной общей базы данных для нескольких сервисов.

Однако использование общих баз данных у микросервисов является спорным вопросом и может привести к ряду проблем. Одной из таких проблем является нарушение принципа изоляции микросервисов, так как доступ к общей базе данных может привести к тому, что изменения, сделанные одним микросервисом, окажутся неожиданными для другого. Кроме того, использование общей базы данных может усложнить масштабирование системы и привести к увеличению нагрузки на базу данных.

Данный способ опасный и не рекомендуется к использованию из-за проблемы несогласованности данных между сервисами.

### Использование кэша

Вместо того, чтобы запрашивать даннные или иметь доступ к общей базе данных, микросервисы могут кэшировать те данные, которые им необходимы. Например, микросервис А отслеживает изменения на микросервисе Б, и в соответствии с изменением его базы данных, меняет и свою аналогично. В итоге имеются 2 почти идентичные копии базы данных. Конечно, хранить два идентчиных набора данных нецелесообразно, однако стоит учитывать, что копия базы данных на микросервисе А является лишь частичной копией оригинала, так как редко когда микросервису необходим полный доступ к базе данных другого сервиса. А если такой случай возникнет, то стоит задуматься над рациональностью разделения этих сервисов и возможностью объединения их в один.

При использовании кэша очень хорошо себя показывают события, которые публикуются в шину сообщений и содержат изменения, которые произошли с базой данных. Таким образом все заинтересованные компоненты могут отразить эти изменения локально в своих базах данных.

### Транзакции

Следующая проблема, которая встает при выстраивании коммуникации между микросервисами – распределенные транзакции.

Проблема транзакций связана с этой проблемой согласованности. Транзакция - это логически связанный набор операций, который должен быть выполнен атомарно (целиком или не выполнен вообще), чтобы гарантировать целостность данных. В традиционной монолитной архитектуре приложений, транзакции обычно выполняются в рамках одной базы данных, что обеспечивает согласованность данных. Однако в микросервисной архитектуре, где каждый сервис имеет свою базу данных, транзакции могут оказаться распределенными между разными сервисами.

Как решать проблему транзакций в микросервисах? Существует несколько подходов:

1. **Компенсирующие транзакции (Compensating Transactions):** В этом подходе, каждый микросервис выполняет свою транзакцию независимо, и в случае возникновения ошибки, исправляет изменения, сделанные в других сервисах. Например, если один сервис выполняет перевод денег с одного счета на другой, а другой сервис обновляет баланс счета, то при возникновении ошибки в одном из сервисов, другой сервис может исправить баланс счета обратно.
2. **Двухфазные коммиты (Two-Phase Commits)**: В этом подходе, каждый микросервис участвует в распределенной транзакции, и сначала происходит фаза подготовки, где каждый сервис подтверждает, что готов к выполнению транзакции, затем происходит фаза фиксации, где транзакция выполняется. Если один из сервисов не может выполнить транзакцию, то все транзакции отменяются.
3. **Саги (Sagas):** Это подход, в котором транзакция состоит из нескольких шагов (вызовов сервисов), каждый из которых выполняется в рамках своей транзакции. Если какой-то шаг не может быть завершен успешно, то Saga использует компенсирующие действия, чтобы отменить изменения, сделанные в предыдущих шагах, и вернуть систему к согласованному состоянию. Пример сценария Saga может быть следующим: создание заказа на сайте электронной коммерции. Сценарий может включать создание заказа в базе данных, проверку наличия товаров на складе, списание товаров со склада и создание записи о доставке. Если какой-то шаг не проходит успешно, Saga может использовать компенсирующие действия, чтобы отменить изменения, сделанные в предыдущих шагах (например, вернуть товар на склад), и вернуть систему в согласованное состояние.

В данной работе будет использован подход Saga для согласования транзакций между сервисами из-за его отказоустойчивости и прозрачности действий.

Но и сам подход Saga бывает двух видов, которые отличаются по способу организации логики транзакций:

* Choreography-based Saga (сага на основе хореографии) - каждый сервис определяет свою роль в транзакции, и они совместно согласовывают логику транзакции. Каждый сервис публикует события, которые сигнализируют о завершении своей части транзакции, и другие сервисы реагируют на эти события соответствующим образом.
* Orchestration-based Saga (сага на основе оркестрации) - один сервис, называемый оркестратором, управляет выполнением всей транзакции и координирует работу других сервисов. Оркестратор определяет порядок выполнения каждой операции транзакции и управляет ее состоянием.

Я решил использовать сага на основе оркестрации из-за прозрачности действий и удобства хранения всей логики транзакции на одном сервисе.

## Хранение данных в виде событий

При традиционном подходе к хранению данных, в базе данных хранится только самое последнее состояние приложения и при каждом изменении, старые данные заменяются на новые. У этого подхода есть несколько недостатков:

1. Потеря информации: перезаписывание данных и замена старой информации влечет за собой ее потерю. Мы не можем вернуть систему к состоянию, в котором она находилась в прошлом.
2. Отсутствие прослеживаемости: прямым следствием из предыдущего пункта является потеря причинно-следственной связи между действиями произведенными над системой и ее состоянием. Из-за того, что мы не отслеживаем все состояния системы, мы не можем знать по каким причинам она приняли свое текущее состояние. Например, если обнаружилась критическая ошибка, что заказ может перейти из статуса “Создан” в статус “Выдан”, не проходя статуса “Оплачен”, мы не сможем это отследить только исходя из последнего состояния заказа.
3. Потеря статистики: так как мы не знаем как и почему изменялась система, то и не можем собрать статистику ее изменений, хотя это может быть очень полезным для разработки приложения. К примеру, невозможно собрать информацию о том сколько в среднем времени занимала сборка заказа за март, что согласитесь было бы очень полезным для владельца бизнеса.

Для решения этих проблем существует такой подход к проектированию приложений как Event Sourcing(отслеживание событий или событийный подход к хранению данных).

### Event Sourcing

**Event Sourcing** - это подход проектирования, который позволяет сохранять все изменения состояния системы в виде событий, происходящих в системе. Каждое событие описывает изменение состояния системы в конкретный момент времени. Вся история событий может быть использована для восстановления состояния системы в определенный момент времени.

В Event Sourcing, изменения состояния системы не сохраняются напрямую в базе данных, а записываются в виде событий в журнале событий. Каждое событие представляет собой объект, содержащий информацию о произошедшем изменении состояния системы. После того, как событие было сохранено в журнале, оно не может быть изменено или удалено.

При чтении данных система восстанавливает текущее состояние, пересчитывая все события, связанные с этим состоянием, в соответствии с логикой бизнес-процесса. Таким образом, Event Sourcing позволяет сохранять полную историю изменений состояния системы, что упрощает анализ данных и позволяет лучше понимать, как система работает.

Говоря об Event Sourcing, нельзя не упомянуть такой подход как **Command Query Responsibility Segregation**(разделение ответственности между командами и запросами) или сокращенно **CQRS**.

### CQRS

CQRS - это архитектурный подход, который предлагает разделить операции чтения и записи данных в системе на отдельные модели и обработчики запросов. Давайте рассмотрим основные компоненты:

Команды(commands) – операции, которые используются для записи данных в систему. Результатом команды является изменение системы, а значит и генерация нового события(event).

Запросы(queries) – операции, которые используются для чтения данных из системы. Результатом запроса являются данные, которые были запрошены из системы. В отличии от команды, запрос не изменяет состояние системы и не генерирует событие.

События(events) – объекты, которые содержат информацию об изменении состояния системы. Событие является единственным источником истины о состоянии приложения. Они не являются обязательным участником подхода CQRS, но так как мы используем этот подход вместе с Event Sourcing, то их можно использовать вместе.

### Разделение данных

Так же CQRS предполагает наличие двух баз данных: базы данных для записи (write database) и базы данных для чтения (read database).

База данных для записи содержит все изменения, сделанные в системе, с учетом всех команд, поступивших в систему. Она используется только для записи новых данных и обновления существующих записей.

База данных для чтения содержит только те данные, которые нужны для выполнения запросов. Она используется только для чтения и оптимизирована для этой операции. Также может содержать данные, полученные из других источников.

В контексте событийного подхода к хранению данных, базой данных для записи будет служить журнал событий.

А база данных для чтения будет следить за событиями, которые попадают в журнал и исходя из них делать изменения данных.

Одной из главных причин наличия базы данных для чтения является трудность в поиске данных в журнале событий. Каждый раз, когда нужно будет совершить поиск данных, будет необходимо заново восстанавливать текущее состояние системы, что очень ресурсоемко. Поэтому гораздо лучше осуществлять поиск данных и их отображение при помощи отдельной базы данных.

Важно отметить, что команды созданные на основе данных из базы данных для чтения, все равно при обработке должны согласовываться с хранилищем событий и база данных для чтения не является источником истины о состоянии приложения.

# ПРОЕКТИРОВАНИЕ ПРИЛОЖЕНИЙ ДЛЯ ФУНКЦИЙ РЕСТОРАНА

## Проектирование системы

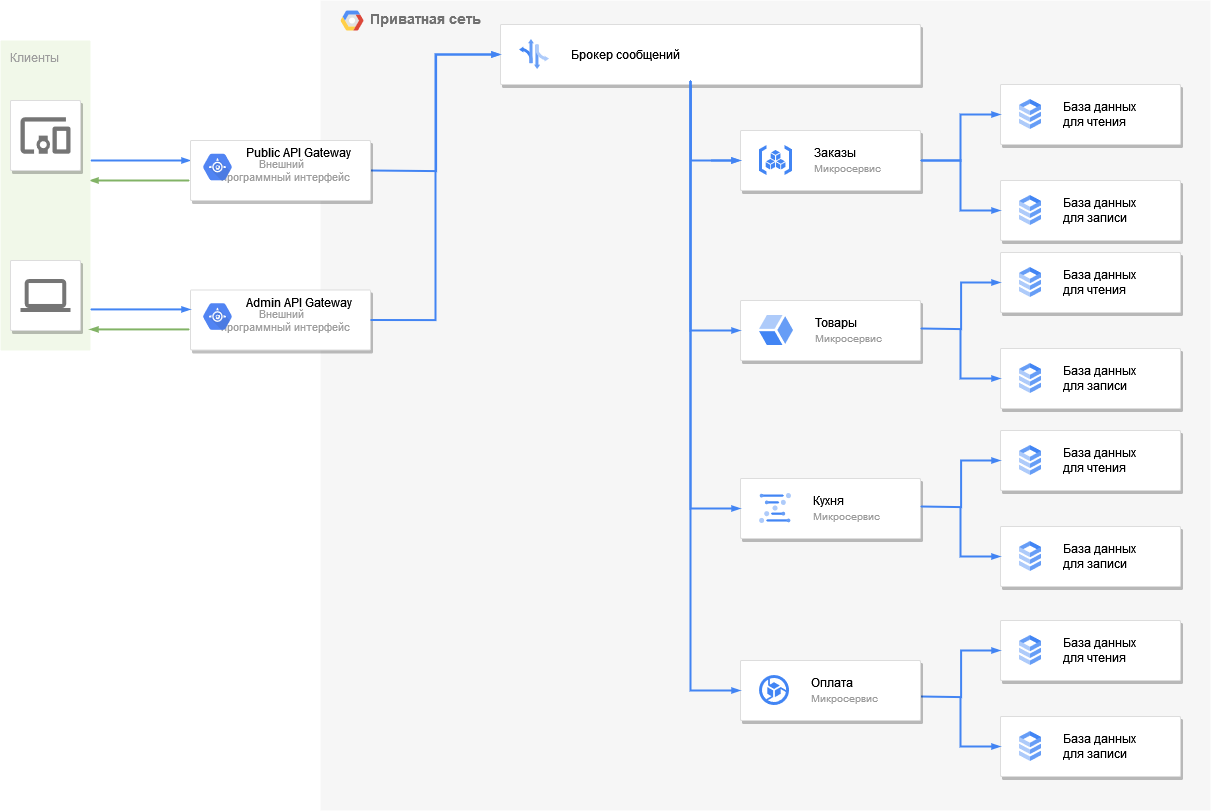
Ранее в составлении задачи на разработку приложения, мы определились с функционалом, который нужен в нашем приложении. Давайте его приведем еще раз:

Функциональные требования административной панели управления:

* Создание/редактирование товаров
* Создание/редактирование меню
* Создание, сборка и выдача, отслеживание статусов заказов
* Отслеживание недостающих товаров для сборки заказов
* Отслеживание запасов товаров
* Создание и выполнение запросов на приготовление товаров
* Отображение журнала событий для
  + Заказов
  + Товаров
  + Меню
  + Запасов товаров
  + Запросов на приготовление товаров
* Отображение списка событий, связанных с запрашиваемыми объектами для
  + Заказов
  + Товаров
  + Меню
  + Запасов товаров
  + Запросов на приготовление товаров

Функциональные требования клиентской панели:

* Просмотр меню
* Создание и отслеживание статусов заказов

Исходя из данного функционала можно спроектировать следующую систему микросервисов:

Рассмотрим ее компоненты:

* Public API Gateway – внешний программный интерфейс, который обрабатывает запросы пришедшие от клиентов ресторана. Он имеет доступ к просмотру меню, созданию, оплате и отслеживанию статуса заказа.
* Admin API Gateway – внешний программный интерфейс, который обрабатывает запросы пришедшие от административной панели управления. Он имеет доступ ко всем функциям приложения, а так же просмотру журнала событий.
* Микросервис заказов – сервис, который занимается хранением, созданием, сборкой, выдачей и отслеживанием статусов заказов.
* Микросервис товаров – сервис, который занимается хранением, добавлением и изменением доступных товаров на складе.
* Микросервис кухни – сервис, который занимается отслеживанием запасов товаров и приготовлением продуктов.
* Микросервис оплаты – сервис, который занимается оплатой заказов.
* Брокер сообщений – шина сообщений Apache Kafka, которая отвечает за коммуникацию между сервисами и интерфейсами.

## 3.2 Проектирование внешнего программного интерфейса

В основу обоих программных интерфейсов будет положен принцип REST с использованием модульной архитектуры приложения характерной для фреймворка Nest.js.

REST (Representational State Transfer) - это стиль архитектуры программного обеспечения для распределенных систем, который основывается на использовании протокола HTTP. Он определяет набор ограничений и рекомендаций для создания веб-сервисов, которые обеспечивают максимально возможную универсальность и масштабируемость.

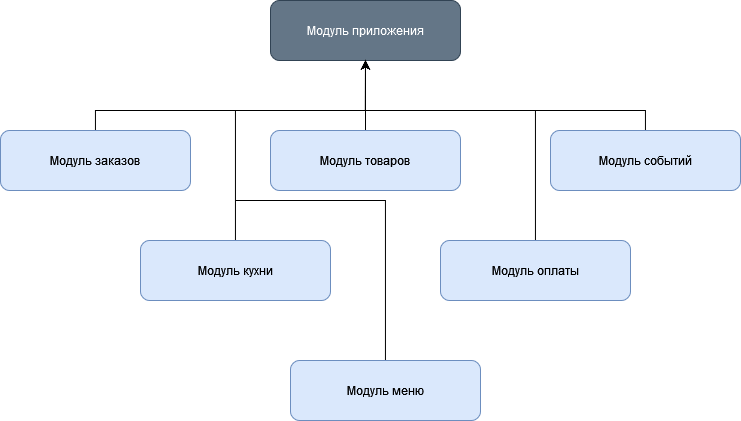
Основные принципы REST:

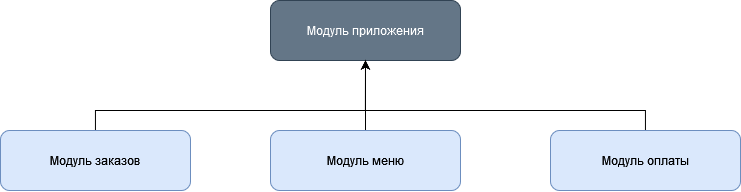
1. Клиент-серверная архитектура: клиенты и серверы должны быть независимыми и взаимодействовать друг с другом через установленный интерфейс.
2. Отсутствие состояния: каждый запрос клиента к серверу должен содержать всю необходимую информацию для выполнения запроса, а сервер не должен сохранять состояние клиента между запросами.
3. Кэширование: клиенты должны иметь возможность кэшировать ответы сервера, чтобы избежать повторных запросов к серверу.
4. Единообразие интерфейса: интерфейс должен быть унифицированным для всех клиентов, чтобы обеспечить максимальную масштабируемость.
5. Слои: клиенты не должны знать о сложности работы серверов и не должны иметь прямого доступа к источнику данных.

Код по требованию: сервер может предоставлять код, который выполняется на стороне клиента для дополнительной функциональности.

REST поддерживает различные форматы данных, включая XML и JSON, а также множество протоколов, включая HTTP, HTTPS, SMTP и FTP. Он используется для разработки веб-сервисов, мобильных приложений и других приложений, которые должны обмениваться данными с другими системами.

Модульная архитектура интерфейсов будет выглядеть следующим образом:

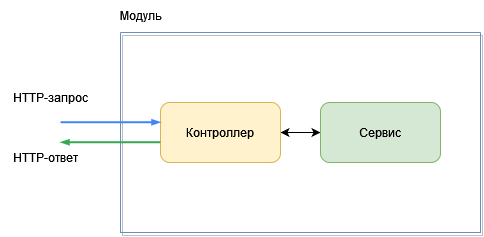




Типичный модуль будет состоять из контроллера и сервиса.

**Контроллер** – объект, который отвечает за обработку и валидацию входящих HTTP-запросов и вызов нужной функции сервиса.

**Сервис** – объект, в котором мы указываем набор методов при помощи которых мы обрабатываем входящие запросы и выполняем необходимые действия.



В конкретном случае нашего проекта, сервис будет посылать запросы к другим приложениям и возвращать данные в виде HTTP-ответа.

## 3.3 Проектирование микросервисов

Все четыре микросервиса будут использовать подход CQRS, поэтому их архитектура будет примерно одинаковой с поправкой лишь на функциональные отличия.

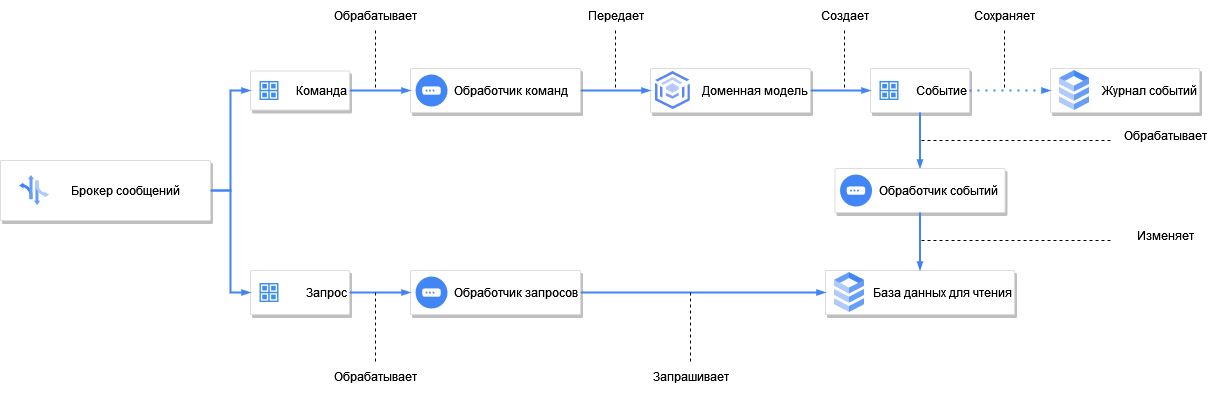
Так же в приложениях будет использоваться такое понятие как доменная модель.

**Доменная модель (Domain Model)** - это понятийная модель, которая описывает объекты, понятия, связи и операции, характерные для предметной области, которую моделируют. Она является представлением предметной области в коде и служит основой для проектирования приложения.

Доменная модель описывает основные сущности (entities) в системе и их атрибуты (properties), а также связи (relationships) между сущностями и поведение (behavior) системы. Она обычно разрабатывается в тесном сотрудничестве с экспертами в предметной области и помогает лучше понять требования к системе.

В своей сути вся логика работы приложения, связанная с предметной областью деятельности, для которой проектируется приложение, должна осуществляться через доменную модель.

В нашем случае это означает, что при создании заказа мы обращаемся к доменной модели, к сущности Заказ(которая выражена классом), передаем параметры и именно в ней реализуется логика обработки заказа, валидация параметров и другие предметные задачи.



Компоненты микросервиса c использованием подхода CQRS:

**Обработчик команд** – объект, который обрабатывает входящие команды, достает из них данные и передает доменной модели для дальнейшей обработки.

**Доменная модель** – представление предметной области в приложении. Она получает данные от обработчика команд и исходя из логики предметной области создает или изменяет доменную сущность, затем генерирует событие исходя из действий, произошедших с сущностью. Например, событие “Заказ №2 создан”.

**Журнал событий** – база данных, которая сохраняет события в хронологическом порядке. Является единственным источником истины о состоянии приложения.

**Обработчик событий** – объект, который обрабатывает входящие события, достает из них данные и изменяет базу данных для чтения.

**База данных для чтения** – база данных, которая содержит в себе данные приложения. В отличии от журнала событий может хранить информацию в любом виде, удобном для операций чтения. Может хранить неполную информацию или дополнять ее самостоятельно.

**Обработчик запросов** – объект, который обрабатывает входящие запросы и возвращает соответствующую им информацию из базы данных для чтения.

# РАЗРАБОТКА ПРИЛОЖЕНИЙ ДЛЯ ФУНКЦИЙ РЕСТОРАНА

## Установка и настройка необходимых компонентов

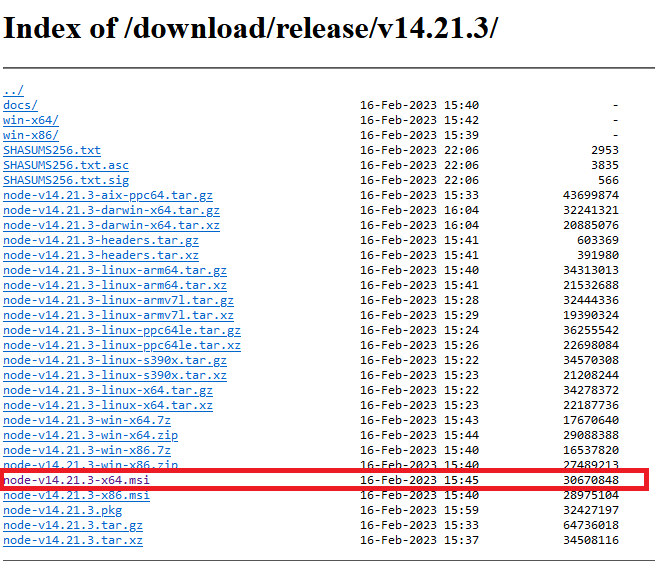
### Установка среды выполнения

В главе 1 данной объяснительной записки было указано, что для выполнения языка JavaScript вне браузера необходима среда выполнения Node.JS.

Версия Node.JS 14 является текущий стабильной версией и самой популярной при разработке коммерческих приложений. Давайте установим среду:

Перейти по ссылке https://nodejs.org/download/release/v14.21.3

Выбрать файл node-v14.21.3-x64.msi



Пройти по всем шагам инсталлятора

Убедиться, что Node.JS установлена:

Microsoft Windows [Version 10.0.19045.2846]

(c) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.

C:\Users\Example>node --version

v14.21.3

C:\Users\Example>

### Установка серверной программной платформы

Серверной программной платформой моего выбора была Nest.JS версии 9.4.0.

Вместе со средой выполнения так же идет пакетный менеджер NPM.

NPM (Node Package Manager) - это менеджер пакетов для языка программирования JavaScript, который позволяет легко устанавливать, обновлять, удалять и использовать пакеты, написанные на JavaScript. NPM поставляется в комплекте с Node.js и является стандартным менеджером пакетов для этой платформы.

Он используется для управления зависимостями в проектах JavaScript, которые могут содержать сотни или тысячи библиотек и модулей, которые могут быть установлены и управляемы автоматически.

Давайте установим Nest.JS с помощью NPM:

1. npm i -g @nestjs/cli
2. Проверим, что фреймворк установлен:

|  |
| --- |
| Microsoft Windows [Version 10.0.19045.2846]  (c) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.  C:\Users\Example>nest --version  8.2.2 |

Приложения было решено хранить в формате монорепозитория, чтобы они могли разделять между собой часто используемые компоненты и библиотеки. Для управления монорепозиториями существует система сборки приложений NX. Установка не требуется.

### Установка клиентской программной платформы

Ранее в качестве клиентской программной платформы был выбран фреймворк React. Установка не требуется.

## Разработка приложений

Добавить сюда инфу про установку библиотек

### Разработка доменных сущностей и сообщений

Перед началом разработки приложений, для каждого микросервиса необходимо выделить из функционала доменные сущности, через которые осуществляется работа с предметной областью, и сообщения.

**Микросервис товаров :**

1. Доменные сущности:
   1. Товар – сущность, состоящая из названия, статуса активности, ссылки на изображение.
   2. Позиция в меню – сущность, состоящая из товара, статуса активности, цены и товара.
   3. Меню – сущность, состоящая из позиций.
2. Команды:
   1. Создание меню
   2. Обновление меню
   3. Создание товара
   4. Обновление товара
3. Запросы:
   1. Получить меню
   2. Получить товар
   3. Найти товары
   4. Получить позицию в меню
4. События:
   1. Меню создано
   2. Меню обновлено
   3. Товар создан
   4. Товар обновлен

**Микросервис заказов:**

1. Доменные сущности:
   1. Позиция в заказе – сущность, состоящая из позиции в меню, количества товаров.
   2. Заказ – сущность, состоящая из номера, статуса заказа, номера платежного документа, позиций заказа.
2. Команды:
   1. Создание заказа
   2. Обновление заказа
3. Запросы:
   1. Получить заказ
   2. Найти заказы
4. События:
   1. Заказ создан
   2. Заказ оплачен
   3. Заказ готов к выдаче
   4. Заказ завершен
   5. Заказ отменен

**Микросервис оплаты:**

1. Доменные сущности:
   1. Платежный документ – сущность, состоящая номера заказа, статуса оплаты, типа оплаты и суммы оплаты.
2. Команды:
   1. Создание платежного документа
   2. Обновление платежного документа
3. Запросы:
   1. Получить платежный документ
4. События:
   1. Платежный документ создан
   2. Платежный документ завершен
   3. Платежный документ отклонен

**Микросервис кухни:**

1. Доменные сущности:
   1. Заявка на приготовление блюда – сущность, из статуса приготовления блюда и номера товара.
   2. Запас блюда – сущность, состоящая из номера товара и количества в запасе.
2. Команды:
   1. Создание заявки на приготовление блюда
   2. Обновление заявки на приготовление блюда
   3. Изменение запаса блюда
3. Запросы:
   1. Получить заявку на приготовление блюда
   2. Найти заявки на приготовление блюда
   3. Получить запас блюда
4. События:
   1. Заявка на приготовление блюда создана
   2. Заявка на приготовление блюда готова
   3. Заявка на приготовление блюда отклонена
   4. Запас блюда создан
   5. Запас блюда изменен

Для каждой доменной сущности создадим по классу отражающему требования к сущности.

Доменные сущности выделим в отдельную общую библиотеку domain-entity.

Пример доменной сущности:

Меню в файле *menu.domain-entity.ts:*

|  |
| --- |
| import { Types } from 'mongoose';  import { MenuItemDomainEntity } from './menu-item.domain-entity';  export class MenuDomainEntity {    private \_id: string;    private \_items: MenuItemDomainEntity[];    private \_active: boolean;    constructor(items: MenuItemDomainEntity[], id?: string, active?: boolean) {      this.id = id ?? new Types.ObjectId(id).toString();      this.items = items;      this.active = active ?? false;    }    public get id(): string {      return this.\_id;    }    public set id(value: string) {      this.\_id = value;    }    public get items(): MenuItemDomainEntity[] {      return this.\_items;    }    public set items(value: MenuItemDomainEntity[]) {      this.\_items = value;    }    public get active(): boolean {      return this.\_active;    }    public set active(value: boolean) {      this.\_active = value;    }    public async addItem(item: MenuItemDomainEntity) {      this.items.push(item);    }    public async removeItem(id: number) {      this.items = this.items.filter((item) => item.id !== id);    }  } |

Каждому из этих сообщений необходимо присвоить:

1. Название топика: имя очереди сообщений в шине сообщений Apache Kafka, куда будут отправлять сообщения
2. Класс, описывающий данные, содержащиеся в сообщении
3. Класс, описывающий данные, содержащиеся в ответном сообщении(в случае команды или запроса)

Классы описывающие данные в сообщениях выделим в общую библиотеку *contracts*.

Пример классов для запроса на поиск заявок на приготовления блюд *cooking-request.find.ts*:

|  |
| --- |
| import { CookingRequestStatus } from '@burger-shop/interfaces';  import { ApiProperty } from '@nestjs/swagger';  import { IsEnum } from 'class-validator';  import { CookingRequestGetQueryResponse } from './cooking-request.get';  export class CookingRequestFindQueryRequest {    @IsEnum(CookingRequestStatus)    @ApiProperty({      name: 'filter[status]',      enum: CookingRequestStatus,    })    public readonly status: CookingRequestStatus;  }  export class CookingRequestFindQueryResponse {    @ApiProperty({ type: CookingRequestGetQueryResponse, isArray: true })    public readonly requests: CookingRequestGetQueryResponse[];  } |

Топики событий выделим в объекты и поместим в файл *event-store.const.ts*.

Пример топиков событий для товаров:

|  |
| --- |
| export enum ProductEventNames {    productCreated = 'product.created.event',    productUpdated = 'product.updated.event',    productSnapshot = 'product.snapshot.event',  } |

Топики команд и запросов поместим в *kafka.topics.enum.ts.*

Пример топиков команд и запросов:

|  |
| --- |
| export enum CommandTopics {    orderCreate = 'order.create.command',    orderUpdate = 'order.update.command',    paymentCreate = 'payment.create.command',    paymentUpdate = 'payment.update.command',    orderPay = 'order.pay.command',    productCreate = 'product.create.command',    productDelete = 'product.delete.command',    productUpdate = 'product.update.command',  …    menuClearRead = 'menu.clear-read.command',  }  export enum QueryTopics {    orderGet = 'order.get.query',    orderFind = 'order.find.query',    paymentGet = 'payment.get.query',    menuGet = 'menu.get.query',  …    cookingStockGet = 'cooking-stock.get.query',  } |

### Разработка сервиса для работы с журналом событий

Для удобства с журналом событий разработаем для каждого приложения сервис для работы с ним.

Сервис должен выполнять следующие задачи:

* Сохранять события в журнал
* Создавать снимки состояния событий
* Искать события по дате создания и по идентификатору события
* Получать состояние сущности из событий по ее идентификатору

Все сервисы для работы с журналом будут вести себя примерно аналогично, поэтому рассмотрим их работу на примере сервиса для приложения оплаты в файле *event-store.payment.service.ts*.

**Сохранение события.**

Сервис обращается к журналу событий и сохраняет объект события.

|  |
| --- |
| public async savePaymentEvent(dto: ISaveEvent) {      return this.paymentModel.create(dto);    } |

**Поиск событий.**

Сервис формирует фильтр и обращается к базе данных, получая список событий.

|  |
| --- |
| public async getPaymentEvents(filter: {      from?: Date;      to?: Date;      id?: string;    }) {      const { from, to, id } = filter;      const mongoFilter: any = {};      if (from) {        mongoFilter['updatedAt'] = {};        mongoFilter['updatedAt']['$gte'] = from;      }      if (to) {        mongoFilter['updatedAt'] = mongoFilter['updatedAt'] ?? {};        mongoFilter['updatedAt']['$lt'] = to;      }      if (id) {        mongoFilter['objectId'] = id;      }      return this.paymentModel.find(mongoFilter).sort({ createdAt: 'asc' });    } |

**Получить платежный документ из событий.**

Сервис находит последний снимок состояния событий и затем выбирает события, которые были созданы после снимка. Это необходимо для оптимизации поиска событий.

Затем происходит восстановление доменной сущности при помощи класса *PaymentDomainTransformer* и его метода *hydrate.*

|  |
| --- |
| public async getPayment(id: string) {      const snapshot = await this.paymentSnapshotModel.findOne({ objectId: id });      const date = snapshot?.createdAt ?? new Date(Date.parse('2023-02-24'));      const events = await this.getPaymentEvents({ from: date, id });      if (snapshot) {        events.push(snapshot);      }      return PaymentDomainTransformer.hydrate(events);    } |

**Создать снимок состояния событий.**

Сервис проверяет сколько прошло событий с момента последнего снимка состояния событий объекта. Если прошло некоторое N, то он создает снимок состояния событий и затем при создании доменной сущности отсчет идет от этого снимка.

|  |
| --- |
| public async makeSnapshotPaymens() {      const latestSnapshot = await this.paymentSnapshotModel        .findOne()        .sort({ createdAt: 'desc' });      const date =        latestSnapshot?.createdAt ?? new Date(Date.parse('2023-02-24'));      const snapshots = await this.getPaymentSnapshots(date);      const events = await this.getPaymentEvents({ from: date });      if (events.length < SNAPSHOT\_FREQUENCY) return;      const eventStreamMap = new Map<string, EventDocument[]>();      snapshots.forEach((event) => {        const events = eventStreamMap.get(event.objectId);        if (events) {          events.push(event);          eventStreamMap.set(event.objectId, events);        } else {          eventStreamMap.set(event.objectId, [event]);        }      });      events.forEach((event) => {        const events = eventStreamMap.get(event.objectId);        if (events) {          events.push(event);          eventStreamMap.set(event.objectId, events);        } else {          eventStreamMap.set(event.objectId, [event]);        }      });      for (const [productId, data] of eventStreamMap.entries()) {        const product = PaymentDomainTransformer.hydrate(data);        const event = PaymentDomainTransformer.snapshot(product);        await this.paymentSnapshotModel.findOneAndUpdate(          { objectId: event.objectId },          event,          {            upsert: true,          }        );      }      Logger.log('Snapshotted payments');    } |

**Преобразователь событий.**

Для преобразования событий в доменные сущности нам необходимо определить класс с методами:

* Создать сущность из массива событий
* Создать снимок состояния из сущности

Для сервиса оплаты это класс *PaymentDomainTransformer* определенныйв файле *payment.domain-transformer.ts.*

|  |
| --- |
| export default class PaymentDomainTransformer {    public static hydrate(events: ISaveEvent[]) {      const domain = new PaymentDomainEntity({        sum: 0,        type: PaymentType.CARD,        orderId: '',      });      events.forEach((event) => {        this.applyEvent(domain, event);      });      return domain;    }    private static applyEvent(domain: PaymentDomainEntity, event: ISaveEvent) {      if (event.name === PaymentEventNames.paymentSnapshot) {        const { payment } = JSON.parse(          event.payload        ) as PaymentCreatedEventPayload;        domain.id = payment.id;        domain.link = payment.link ?? '';        domain.status = payment.status;        domain.sum = payment.sum;        domain.type = payment.type;      }      if (event.name === PaymentEventNames.paymentCreated) {        const { payment } = JSON.parse(          event.payload        ) as PaymentCreatedEventPayload;        domain.id = payment.id;        domain.link = payment.link ?? '';        domain.status = payment.status;        domain.sum = payment.sum;        domain.type = payment.type;      }      if (event.name === PaymentEventNames.paymentFulfilled) {        const { status } = JSON.parse(          event.payload        ) as PaymentStatusUpdatedEventPayload;        domain.status = status;      }      if (event.name === PaymentEventNames.paymentRejected) {        const { status } = JSON.parse(          event.payload        ) as PaymentStatusUpdatedEventPayload;        domain.status = status;      }      if (event.name === PaymentEventNames.paymentRefunded) {        const { status } = JSON.parse(          event.payload        ) as PaymentStatusUpdatedEventPayload;        domain.status = status;      }    }    public static snapshot(domain: PaymentDomainEntity): ISaveEvent {      const payload: PaymentCreatedEventPayload = {        payment: {          id: domain.id,          link: domain.link,          status: domain.status,          sum: domain.sum,          type: domain.type,          orderId: domain.orderId,        },      };      return {        objectId: domain.id,        name: PaymentEventNames.paymentSnapshot,        payload: JSON.stringify(payload),      };    }  } |

Метод **hydrate** – создает сущность из массива событий.

Метод **snapshot** – создает снимок состояния событий из сущности.

**Библиотека для работы с журналом событий.**

Выделим сервисы для работы с журналом событий в библиотеку *event-store*.

### Разработка сервиса для отправки сообщений

Ранее мы определились с топиками сообщений, которые будут участвовать в нашей системе. Для отправки сообщения в соответствующий ему топик создадим сервис *KafkaProducerService* в файле *kafka-producer.service.ts*.

Определим у него методы:

* **send** – метод отправляющий команду или запрос в топик
* **emit** - метод отправляющий событие в топик

|  |
| --- |
| public async emit<TResult, TInput>(      topic: string,      value: TInput    ): Promise<void> {      this.logger.verbose(`Emit ${topic}, value:${JSON.stringify(value)}`);      await this.kafka.connect();      await this.kafka.emit<TResult, TInput>(topic, value);    }    public async send<TResult, TInput>(      topic: string,      value: TInput    ): Promise<TResult> {      this.logger.verbose(`Send ${topic}, value:${JSON.stringify(value)}`);      await this.kafka.connect();      const result = await lastValueFrom(        this.kafka.send<TResult, TInput>(topic, value)      );      return result;    } |

На основании этих методов определим упрощенные методы для отправки сообщений. Для примера рассмотрим два из них.

Метод для отправки команды создания товара

|  |
| --- |
| public async sendProductCreate(      payload: ProductCreateRequest    ): Promise<ProductCreateResponse> {      return this.send<ProductCreateResponse, ProductCreateRequest>(        CommandTopics.productCreate,        payload      );    } |

При помощи метода *send*, он отправляет данные в топик *CommandTopics.productCreate*.

Метод для отправки события создания товара

|  |
| --- |
| public async emitProductCreated(      payload: ProductCreatedEventPayload    ): Promise<void> {      await this.kafka.emit<void, ProductCreatedEventPayload>(        EventTopics.productCreated,        payload      );    } |

При помощи метода *emit*, он отправляет данные в топик       *EventTopics.productCreated*.

Остальные методы выглядят почти одинаково, полное описание сервиса можно найти в приложении.

### Обновление базы данных для чтения

Контроллер *product.query.controller.ts* отвечает за обработку событий и запросов. Через него происходит обращение к базе данных для чтения и ее обновление.

Давайте на примере меню посмотрим как работает обновление.

|  |
| --- |
| @MessagePattern(EventTopics.menuUpdated)    public async onMenuUpdated(      @Payload() payload: MenuUpdatedEventPayload    ): Promise<void> {      return this.productQueryService.onMenuUpdated(payload);  } |

Контроллер считывает сообщение обновления меню, достает данные и вызывает метод onMenuUpdated класса ProductQueryService.

|  |
| --- |
| public async onMenuUpdated(dto: MenuUpdatedEventPayload): Promise<void> {      const { menu } = dto;      if (dto.eventName === MenuEventNames.menuActivated) {        await this.menuRepository.update(menu.id, {          active: true,        });      }      if (dto.eventName === MenuEventNames.menuDeactivated) {        await this.menuRepository.update(menu.id, {          active: false,        });      }      if (dto.eventName === MenuEventNames.menuItemsUpdated) {        await this.menuRepository.update(menu.id, {          items: menu.items.map((item, idx) => {            const { productId } = item;            return { ...item, id: idx, product: productId };          }),        });      }    } |

В методе происходит обращение к базе данных через репозиторий и обновляются данные.

Обновление базы данных для чтения во всех сервисах происходит аналогично с поправкой на различия в хранимой информации.

### Разработка микросервиса товаров

Исходя из задачи можем выделить следующий функционал микросервиса по работе с товарами:

* Создание/редактирование товаров
* Создание/редактирование меню
* Просмотр меню

Для обработки команд необходим контроллер *product.command.controller.ts*:

|  |
| --- |
| import {    MenuCreateCommandRequest,    MenuCreateCommandResponse,    MenuUpdateCommandRequest,    MenuUpdateCommandResponse,    ProductCreateRequest,    ProductCreateResponse,    ProductDeleteRequest,    ProductDeleteResponse,    ProductUpdateRequest,    ProductUpdateResponse,  } from '@burger-shop/contracts';  import {    CommandTopics,    KafkaLoggerInterceptor,  } from '@burger-shop/kafka-module';  import { UseInterceptors } from '@nestjs/common';  import { Controller } from '@nestjs/common';  import { MessagePattern, Payload } from '@nestjs/microservices';  import ProductCommandService from './product.command.service';  @UseInterceptors(KafkaLoggerInterceptor)  @Controller()  export default class ProductCommandController {    constructor(private readonly productCommandService: ProductCommandService) {}    @MessagePattern(CommandTopics.productCreate)    public async create(      @Payload() payload: ProductCreateRequest    ): Promise<ProductCreateResponse> {      return this.productCommandService.create(payload);    }    @MessagePattern(CommandTopics.productUpdate)    public async update(      @Payload() payload: ProductUpdateRequest    ): Promise<ProductUpdateResponse> {      return this.productCommandService.update(payload);    }    @MessagePattern(CommandTopics.productDelete)    public async delete(      @Payload() payload: ProductDeleteRequest    ): Promise<ProductDeleteResponse> {      return this.productCommandService.delete(payload);    }    @MessagePattern(CommandTopics.menuCreate)    public async createMenu(      @Payload() payload: MenuCreateCommandRequest    ): Promise<MenuCreateCommandResponse> {      return this.productCommandService.createMenu(payload);    }    @MessagePattern(CommandTopics.menuUpdate)    public async updateMenu(      @Payload() payload: MenuUpdateCommandRequest    ): Promise<MenuUpdateCommandResponse> {      return this.productCommandService.updateMenu(payload);    }    @MessagePattern(CommandTopics.menuClearRead)    public async menuClearRead(): Promise<void> {      return this.productCommandService.menuClearRead();    }    @MessagePattern(CommandTopics.productClearRead)    public async productClearRead(): Promise<void> {      return this.productCommandService.productClearRead();    }  } |

Функция-обработчик команды состоит из нескольких компонентов:

* Декоратор @MessagePattern – функция-декоратор, которая вызывает закрепленную за ней функцию, когда появляется новое сообщение в топике, указанном в качестве параметра функции.
* Декоратор @Payload – функция-декоратор, которая позволяет достать данные из сообщения и поместить их в переменную, закрепленную за декоратором.
* Декоратор @Controller() – функция-декоратор, которая объявляет, что этот класс нужно использовать в качестве контроллера.

В контроллере мы имеем набор функций-обработчиков команд для всех команд определенных в приложении.

После обработки команды, функции вызывают соответствующие методы объекта класса *ProductCommandService*, в котором осуществляется работа с доменной моделью, создание событий и вызов других команд.

#### Создание товара

Рассмотрим функцию *create*, которая отвечает за создание товара.

|  |
| --- |
| public async create(      dto: ProductCreateRequest    ): Promise<ProductCreateResponse> {      const domain = new ProductDomainEntity(dto.name, null, dto.imgLink);      const payload: ProductCreatedEventPayload = {        product: {          name: domain.name,          id: domain.id,          imgLink: domain.imgLink,        },        eventName: ProductEventNames.productCreated,      };      await this.kafkaProducerService.emitProductCreated(payload);      await this.eventStoreService.saveProductEvent({        objectId: domain.id,        name: ProductEventNames.productCreated,        payload: JSON.stringify(payload),      });      return {        succes: true,      };    } |

Алгоритм создания товара следующий:

* + 1. Создается доменная сущность ProductDomainEntity используя данные из команды.
    2. Создаются данные для события создания товара payload
    3. Событие отправляется в шину сообщений при помощи сервиса kafkaProducerService
    4. Событие сохраняется в журнал событий при помощи сервиса eventStoreService

#### Обновление товара

Рассмотрим функцию *create*, которая отвечает за создание товара.

|  |
| --- |
| public async update(      dto: ProductUpdateRequest    ): Promise<ProductUpdateResponse> {      const { id, name, imgLink } = dto;      const product = await this.eventStoreService.getProduct(id);      if (!product) return { success: false };      const payload: ProductUpdatedEventPayload = {        product: {          id,          name,          imgLink,        },        eventName: ProductEventNames.productUpdated,      };      await this.kafkaProducerService.emitProductUpdated(payload);      await this.eventStoreService.saveProductEvent({        objectId: id,        name: ProductEventNames.productUpdated,        payload: JSON.stringify(payload),      });      return { success: true };    } |

Алгоритм обновления товара следующий:

* + 1. Из журнала событий достается товар с идентификатором id.
    2. При неуспешном нахождении товара функция завершается, иначе у доменной сущности товара обновляются данные и создаются данные для события обновления товара payload.
    3. Событие отправляется в шину сообщений при помощи сервиса kafkaProducerService
    4. Событие сохраняется в журнал событий при помощи сервиса eventStoreService

#### Создание меню

|  |
| --- |
| public async createMenu(      dto: MenuCreateCommandRequest    ): Promise<MenuCreateCommandResponse> {      const { items } = dto;      const productsMap = new Map<string, ProductDocument>();      const menu = await this.menuRepository.findOne({ active: true });      if (menu) return null;      for (const item of items) {        const product = await this.productRepository.find(item.productId);        if (!product) {          return { success: false };        }        productsMap.set(item.productId, product);      }      const menuItems = items.map((item, idx) => {        const product = productsMap.get(item.productId);        return new MenuItemDomainEntity(          product.id,          item.available,          item.price,          idx        );      });      const menuDomain = new MenuDomainEntity(menuItems, null, true);      const payload = {        menu: {          items: menuDomain.items.map((item, idx) => ({            available: item.available,            price: item.price,            productId: item.productId,            id: idx,          })),          id: menuDomain.id,          active: menuDomain.active,        },        eventName: MenuEventNames.menuCreated,      };      await this.kafkaProducerService.emitMenuCreated(payload);      await this.eventStoreService.saveMenuEvent({        objectId: menuDomain.id,        name: MenuEventNames.menuCreated,        payload: JSON.stringify(payload),      });      return {        success: true,        id: menuDomain.id,      };    } |

Алгоритм создания меню:

1. Проверить, что другое меню не существует
2. Если существует, то выйти из функции
3. Найти все товары, указанные в данных *items*
4. Если хоть 1 товар не найден, то выйти из функции
5. Создать доменные сущности для товаров MenuItemDomainEntity
6. Создать доменную сущность для меню MenuDomainEntity
7. Создать событие создания меню
8. Отправить событие в шину сообщений
9. Сохранить событие в журнал событий

#### Обновление меню

|  |
| --- |
| public async updateMenu(      dto: MenuUpdateCommandRequest    ): Promise<MenuUpdateCommandResponse> {      const { id, data } = dto;      const { items, active } = data;      const menu = await this.menuRepository.get(id);      if (!menu) return { success: false };      if (menu.active !== active) {        if (menu.active === false) {          await this.deactivateMenu(dto);        } else {          await this.activateMenu(dto);        }      }      const productsMap = new Map<string, ProductDocument>();      for (const item of items) {        const product = await this.productRepository.find(item.productId);        if (!product) {          return { success: false };        }        productsMap.set(item.productId, product);      }      const menuItems = items.map((item, idx) => {        return new MenuItemDomainEntity(          item.productId,          item.available,          item.price,          idx        );      });      const menuDomain = new MenuDomainEntity(menuItems, menu.id, active);      const payload = {        menu: {          items: menuDomain.items.map((item, idx) => ({            available: item.available,            price: item.price,            productId: item.productId,            id: idx,          })),          id: menuDomain.id,        },        eventName: MenuEventNames.menuItemsUpdated,      };      await this.kafkaProducerService.emitMenuUpdated(payload);      await this.eventStoreService.saveMenuEvent({        objectId: menuDomain.id,        name: MenuEventNames.menuItemsUpdated,        payload: JSON.stringify(payload),      });      return {        success: true,      };    } |

Алгоритм обновления меню:

1. Проверить, что меню не существует
2. Если не существует, то выйти из функции
3. Найти все товары, указанные в данных *items*
4. Если хоть 1 товар не найден, то выйти из функции
5. Создать доменные сущности для товаров MenuItemDomainEntity
6. Создать доменную сущность для меню MenuDomainEntity
7. Создать событие обновления меню
8. Отправить событие в шину сообщений
9. Сохранить событие в журнал событий

#### Получение меню

Контроллер *product.query.controller.ts* отвечает за обработку событий и запросов. Через него происходит обращение к базе данных для чтения и ее обновление.

На примере меню можно убедиться, как работает запрос на чтение данных.

|  |
| --- |
| @MessagePattern(QueryTopics.menuGet)    public async getProductMenuById(      @Payload() payload: MenuGetQueryRequest    ): Promise<MenuGetQueryResponse> {      return this.productQueryService.getMenu(payload);    } |

Контроллер считывает запрос на получение меню и вызывает метод getMenu класса ProductQueryService.

В методе происходит обращение к базе данных и возвращается меню, изменяя некоторые поля при помощи метода mapMenu для более удобного представления объекта клиенту.

|  |
| --- |
| public async getMenu(      dto: MenuGetQueryRequest    ): Promise<MenuGetQueryResponse> {      const result = await this.menuRepository.get(dto.id);      const menu = ProductQueryService.mapMenu(result);      return { menu };    }    private static mapMenu(menu: MenuModel): MenuResponseDto {      return {        id: menu.id,        items: menu.items.map(ProductQueryService.mapMenuItem),        active: menu.active,      };    }    private static mapMenuItem(item: MenuItemModel): MenuItemResponseDto {      return {        id: item.id,        available: item.available,        price: item.price,        product: ProductQueryService.mapProduct(item.product),      };    }    private static mapProduct(product: Product): ProductResponseDto {      return { id: product.id, name: product.name, imgLink: product.imgLink };    } |

### Разработка микросервиса оплаты

Функционал микросервиса для оплаты:

* Создание платежного документа
* Обновление платежного документа
* Просмотр платежного документа

Команды будут обрабатываться в контроллере *PaymentCommandController* и передаваться сервису *PaymentCommandService.*

#### Создание платежного документа

|  |
| --- |
| public async createPayment(      data: PaymentCreateCommandRequest    ): Promise<PaymentCreateCommandResponse> {      const { sum, type } = data;      const payment = new PaymentDomainEntity({        sum,        type,        status: PaymentStatus.PENDING,        orderId: data.orderId,      });      const payload: PaymentCreatedEventPayload = {        payment: {          id: payment.id,          status: payment.status,          sum: payment.sum,          type: payment.type,          orderId: data.orderId,        },      };      await this.kafkaManagerService.emitPaymentCreated(payload);      await this.eventStoreService.savePaymentEvent({        name: PaymentEventNames.paymentCreated,        payload: JSON.stringify(payload),        objectId: payment.id,      });      return { id: payment.id };    } |

Алгоритм создания платежного документа:

1. Создать доменную сущность с переданными данными и в статусе ожидает оплаты.
2. Создать данные для события *payload*
3. Сохранить событие в журнал событий
4. Отправить событие в шину сообщений

#### Обновление платежного документа

|  |
| --- |
| public async updatePayment(      data: PaymentUpdateCommandRequest    ): Promise<PaymentUpdateCommandResponse> {      if (data.status === PaymentStatus.FULFILLED) {        return this.fulfillPayment(data);      }      if (data.status === PaymentStatus.REJECTED) {        return this.rejectPayment(data);      }    }    public async fulfillPayment(      data: PaymentUpdateCommandRequest    ): Promise<PaymentUpdateCommandResponse> {      const { id } = data;      const payment = await this.paymentRepository.find(id);      if (!payment) return { success: false };      if (payment.status !== PaymentStatus.PENDING) return { success: false };      const payload: PaymentStatusUpdatedEventPayload = {        id,        status: data.status,        orderId: payment.orderId,        eventName: PaymentEventNames.paymentFulfilled,      };      await this.kafkaManagerService.emitPaymentStatusUpdated(payload);      await this.eventStoreService.savePaymentEvent({        name: PaymentEventNames.paymentFulfilled,        payload: JSON.stringify(payload),        objectId: id,      });      return { success: true };    }    public async rejectPayment(      data: PaymentUpdateCommandRequest    ): Promise<PaymentUpdateCommandResponse> {      const { id } = data;      const payment = await this.paymentRepository.find(id);      if (!payment) return { success: false };      if (payment.status !== PaymentStatus.PENDING) return { success: false };      const payload: PaymentStatusUpdatedEventPayload = {        id,        status: data.status,        orderId: payment.orderId,        eventName: PaymentEventNames.paymentRejected,      };      await this.kafkaManagerService.emitPaymentStatusUpdated(payload);      await this.eventStoreService.savePaymentEvent({        name: PaymentEventNames.paymentRejected,        payload: JSON.stringify(payload),        objectId: id,      });      return { success: true };    } |

Алгоритм обновления платежного документа:

1. В зависимости от переданного статуса вызвать метод *fulfillPayment* или *rejectPayment*.
2. Создать данные для события *payload*
3. Сохранить событие в журнал событий
4. Отправить событие в шину сообщений

#### Просмотр платежного документа

|  |
| --- |
| public async getPayment(      data: PaymentGetQueryRequest    ): Promise<PaymentGetQueryResponse> {      if (isObjectIdOrHexString(data.id) === false) return null;      const payment = await this.paymentRepository.find(data.id);      return { payment: payment as any };    } |

Алгоритм просмотра платежного документа:

1. Получить доменную сущность по id
2. Вернуть данные

### Разработка микросервиса заказов

Функционал микросервиса для заказов:

* Создание заказа
* Получение заказа
* Оплата заказа
* Отметить заказ готовым к выдаче
* Завершить заказ

Команды будут обрабатываться в контроллере *OrderCommandController* и передаваться сервису *OrderCommandService.*

Здесь впервые вводится подход Saga. Для распределенной транзакции “Создание заказа” создадим Saga.

Определим абстрактный класс *CreateOrderSagaState,* в котором будут определены возможные действия с заказом в рамках транзакции.

|  |
| --- |
| export default abstract class CreateOrderSagaState {    public saga: CreateOrderSaga;    public setContext(saga: CreateOrderSaga) {      this.saga = saga;    }    public abstract create(dto: OrderCreateCommandRequest);    public abstract pay(data: PaymentStatusUpdatedEventPayload);    public abstract cancel(orderId: string); // Возвращаем инветарь    public abstract ready(dto: OrderUpdateCommandRequest);    public abstract complete(dto: OrderUpdateCommandRequest);  } |

Действия:

* Создать
* Оплатить
* Отменить
* Отметить готовым к выдаче
* Завершить

На основании этого класса будем создавать шаги транзакции, то есть классы, которые подразумевают, что заказ находится в одном из состояний.

Пример для неоплаченного заказа:

|  |
| --- |
| export class CreateOrderPayStep extends CreateOrderSagaState {    public saga: CreateOrderSaga;    public async create(dto: OrderCreateCommandRequest) {      Logger.error('Заказ уже создан');    }    public async pay(data: PaymentStatusUpdatedEventPayload): Promise<void> {      const order = await this.saga.eventStoreService.getOrder(data.orderId);      if (!order) return;      if (data.status === PaymentStatus.FULFILLED) {        order.status = OrderStatus.PAYED;        const payload: OrderUpdatedEventPayload = {          order: {            id: order.id,            status: order.status,          },          eventName: OrderEventNames.orderPayed,        };        await this.saga.eventStoreService.saveEvent({          objectId: order.id,          payload: JSON.stringify(payload),          name: OrderEventNames.orderPayed,        });        await this.saga.kafkaProducerService.emitOrderUpdated({          order: {            id: order.id,            status: order.status,          },          eventName: OrderEventNames.orderPayed,        });      }    }    public async cancel(orderId: string) {      const order = await this.saga.eventStoreService.getOrder(orderId);      if (!order) return;      order.status = OrderStatus.CANCELED;      const payload: OrderUpdatedEventPayload = {        order: {          id: order.id,          status: order.status,        },        eventName: OrderEventNames.orderCanceled,      };      await this.saga.eventStoreService.saveEvent({        objectId: order.id,        payload: JSON.stringify(payload),        name: OrderEventNames.orderCanceled,      });      await this.saga.kafkaProducerService.emitOrderUpdated({        order: {          id: order.id,          status: order.status,        },        eventName: OrderEventNames.orderCanceled,      });    }    public ready() {      Logger.error('Заказ не оплачен');    }    public complete() {      Logger.error('Заказ не оплачен');    }  } |

Логика действий в этом случае:

* Создать – ошибка, так как заказ уже создан
* Оплатить – удачно, создаем соответствующее событие и сохраняем его
* Отменить – удачно, создаем событие и сохраняем
* Отметить готовым к выдаче – ошибка, заказ нельзя выдать до оплаты
* Завершить – ошибка, нельзя завершить заказ до оплаты

Инициализация шага будет происходить при помощи класса *CreateOrderSaga.*

|  |
| --- |
| export default class CreateOrderSaga {    private state: CreateOrderSagaState;    constructor(      public readonly kafkaProducerService: KafkaProducerService,      public readonly eventStoreService: EventStoreOrderService    ) {}    getState() {      return this.state;    }    setState(status: OrderStatus) {      if (status === OrderStatus.NEW) {        this.state = new CreateOrderNewStep();      }      if (status === OrderStatus.CREATED) {        this.state = new CreateOrderPayStep();      }      if (status === OrderStatus.PAYED) {        this.state = new CreateOrderMarkReadyStep();      }      if (status === OrderStatus.WAITING\_FOR\_PICKUP) {        this.state = new CreateOrderMarkCompleteStep();      }      if (status === OrderStatus.CANCELED) {        this.state = new CreateOrderCanceledStep();      }      this.state.setContext(this);    }  } |

Исходя из статуса сущности заказа, создается соответствующий ему шаг транзакции.

#### Создание заказа

|  |
| --- |
| public async createOrder(      dto: OrderCreateCommandRequest    ): Promise<OrderCreateCommandResponse> {      const saga = new CreateOrderSaga(        this.kafkaProducerService,        this.eventStoreService      );      saga.setState(OrderStatus.NEW);      const result = await saga.getState().create(dto);      return result;    } |

Алгоритм создания заказа:

1. Создаем новую распределенную транзакцию Saga
2. Инициализируем ее текущий шаг
3. Вызываем метод создания заказа

#### Оплата заказа

|  |
| --- |
| public async onPaymentStatusUpdated(data: PaymentStatusUpdatedEventPayload) {      const order = await this.eventStoreService.getOrder(data.orderId);      if (!order || order.status !== OrderStatus.CREATED) return;      const saga = new CreateOrderSaga(        this.kafkaProducerService,        this.eventStoreService      );      saga.setState(order.status);      if (data.eventName === PaymentEventNames.paymentFulfilled) {        return saga.getState().pay(data);      }      if (data.eventName === PaymentEventNames.paymentRejected) {        return saga.getState().cancel(data.orderId);      }    } |

Алгоритм оплаты заказа:

1. Получаем заказ из журнала событий
2. Если он не существует, то выходим из функции
3. Создаем новую распределенную транзакцию Saga
4. Инициализируем ее текущий шаг
5. Если оплата успешна, то вызываем метод оплаты заказа, иначе вызываем метод отмены заказа

#### Отметить заказ готовым к выдаче

|  |
| --- |
| private async markOrderReady(dto: OrderUpdateCommandRequest) {      const { id } = dto;      const order = await this.eventStoreService.getOrder(id);      if (!order) return null;      const saga = new CreateOrderSaga(        this.kafkaProducerService,        this.eventStoreService      );      saga.setState(order.status);      return saga.getState().ready(dto);    } |

Алгоритм готовности заказа к выдаче:

1. Получаем заказ из журнала событий
2. Если он не существует, то выходим из функции
3. Создаем новую распределенную транзакцию Saga
4. Инициализируем ее текущий шаг
5. Вызываем метод готовности заказа к выдаче

#### Отметить заказ завершенным

|  |
| --- |
| public async makrOrderComplete(      dto: OrderUpdateCommandRequest    ): Promise<OrderUpdateCommandResponse> {      const { id } = dto;      const order = await this.eventStoreService.getOrder(id);      if (!order) return null;      const saga = new CreateOrderSaga(        this.kafkaProducerService,        this.eventStoreService      );      saga.setState(order.status);      return saga.getState().complete(dto);    } |

Алгоритм готовности заказа к выдаче:

1. Получаем заказ из журнала событий
2. Если он не существует, то выходим из функции
3. Создаем новую распределенную транзакцию Saga
4. Инициализируем ее текущий шаг
5. Вызываем метод завершения заказа

#### Получение информации о заказе

|  |
| --- |
| async getOrder(data: OrderGetQueryRequest): Promise<OrderGetQueryResponse> {      const result = await this.repository.find({ \_id: data.id });      if (!result) return null;      return {        id: result.id,        status: result.status,      };    } |

Алгоритм получения информации о заказе:

1. Получить доменную сущность по id
2. Вернуть данные