УДК 004.273

ПРИМЕНЕНИЕ МИКРОСЕРВИСНОЙ АРХИТЕКТУРЫ ПРИ РАЗРАБОТКЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Хидиров А.А, студент гр. ИТб-202, III-курс
Научный руководитель: Алексеева Г.А., старший преподаватель
Кузбасский государственный технический университет
имени Т.Ф. Горбачева
г. Кемерово

При создании программного обеспечения необходимо определиться с его архитектуры. Архитектура определяет организацию информационной системы (набор структурных элементов, их интерфейсов и их поведения).

Основными критериями при выборе архитектуры являются:

- отказоустойчивость;
- производительность;
- обновление и поддержка;
- разработка и развертывание;
- интегрирование;
- накладные расходы.

Традиционная модель программного обеспечения представляет собой единый модуль (монолит), работающий автономно и независимо от других приложений (рисунок 1а). Приложение построено как цельный элемент с одной точкой входа. При таком подходе достаточно часто возникают проблемы с масштабированием и отказоустойчивостью программного обеспечения.

Микросервисная архитектура (рисунок 1б) предполагает принципиально иной подход к разработке, при которой приложение собрано из отдельных независимых модулей, составленных из небольших объемов кода. У каждого модуля появляется собственная логика и база данных, а их взаимодействие осуществляется через сеть по протоколонезависимой технологии.

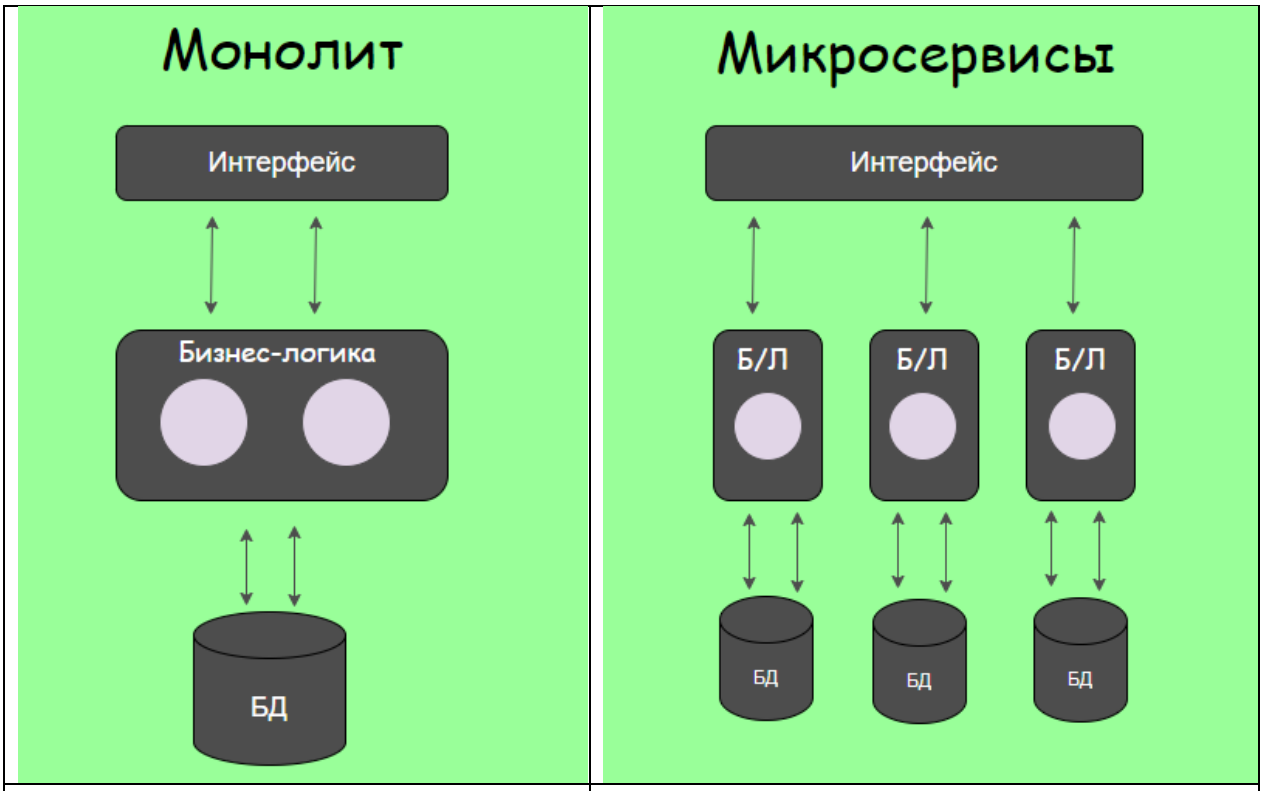
Для демонстрации работы микросервисной архитектуры рассмотрим систему управления задачами (Task Manager). Структура проекта Task Manager приведена на рисунке 2.

При реализации данного решения оно было разделено на три подпроекта (рисунок 3):

1. Task.Microservice – проект создан на ASP.NET Core WebAPI. В качестве базы данных использовано PostgreSQL.

2. Users.Microservice – создан аналогично с Task.Microservice.

3. Task Manager.APIGateway – это промежуточный слой между клиентом и микросервисами, который упрощает доступ к API и управление ими.



а) б)

Рисунок 1

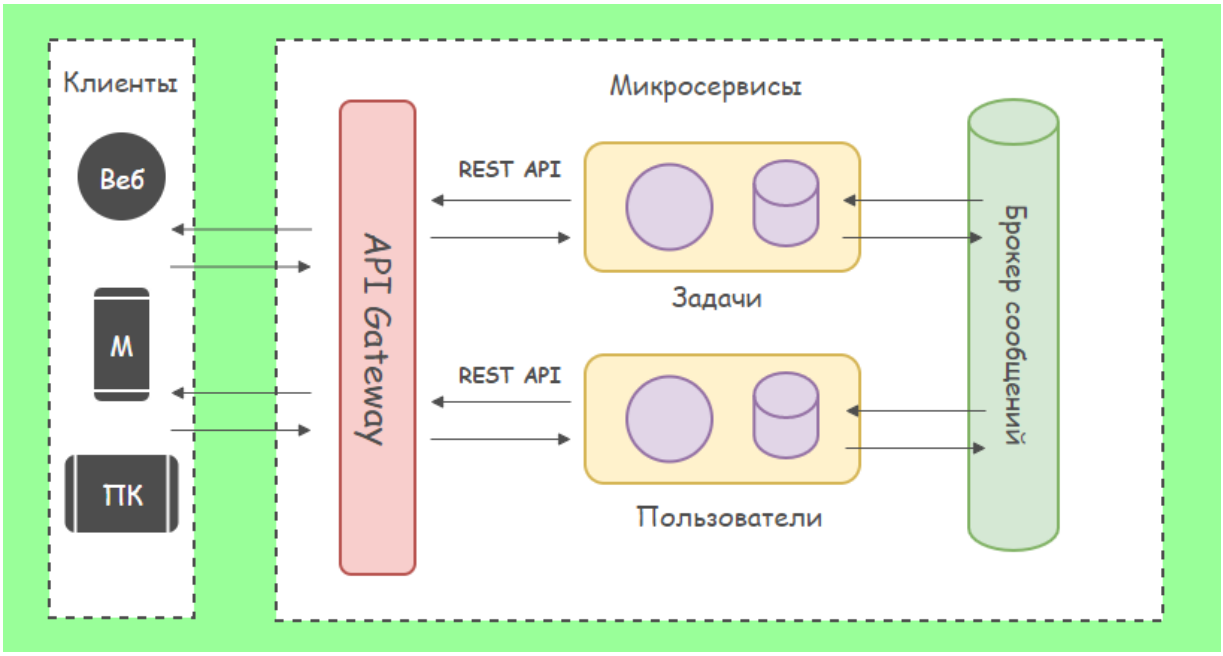


Рисунок 2

Рисунок 3

В Task.Microservice реализованы следующие уровни (рисунок 4):

– Уровень Domain – хранит классы, касающиеся предметной области.
– Уровень Controllers – отвечает за принятие и обработку запросов.
– Уровень Infrastructure – реализует взаимодействие с базой данных.
– Уровень Services – операции (действия).

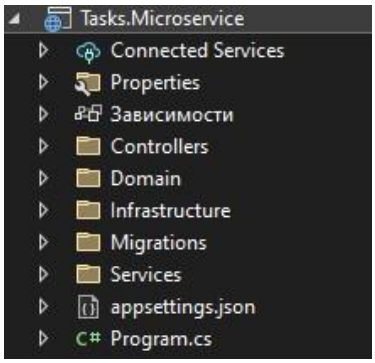


Рисунок 4

Для дальнейшей работы необходимо осуществить проверку конечных точек для Users.Microservice. Проверка и визуализация конечных точек (endpoints) REST API представлена на рисунке 5.

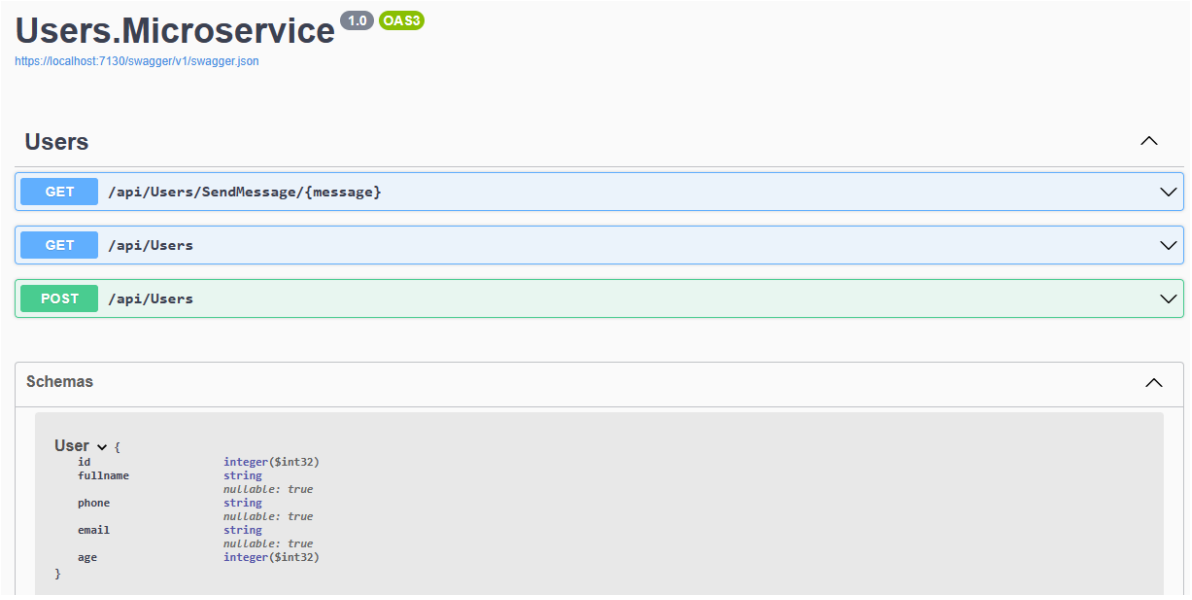


Рисунок 5

Одним из наиболее важных аспектов разработки микросервисов является межсервисная связь. В монолитном приложении выполнение вызовов одного процесса между компонентами реализуется при вызовах методов на уровне языка. Если во время разработки следует шаблон проектирования MVC, обычно имеются классы моделей, которые сопоставляют реляционные базы данных с объектной моделью. Затем создаются компоненты, предоставляющие

методы, помогающие выполнять стандартные операции с таблицами базы данных, такие как создание, чтение, обновление и удаление.

В архитектуре, основанной на микросервисах, важно разделить эту сложную структуру на независимо разработанные и развернутые сервисы, которые также образуют сетку со многими каналами связи. Если речь идет о стилях общения, то можно классифицировать их по двум осям. Первым шагом является определение того, является ли протокол синхронным или асинхронным:

1. Синхронный протокол. Для связи с веб-приложениями протокол HTTP был стандартом в течение многих лет. Это синхронный протокол без сохранения состояния, который имеет свои недостатки. При синхронной связи клиент отправляет запрос и ожидает ответа.

2. Асинхронный. Ключевым моментом здесь является то, что клиент не должен блокировать поток во время ожидания ответа. В большинстве случаев такое общение осуществляется с брокерами обмена сообщениями. Производитель сообщения обычно не ждет ответа. Он просто ждет подтверждения того, что сообщение было получено брокером. Наиболее популярным протоколом для этого типа связи является AMQP (Advanced Message QueuingProtocol), который поддерживается многими операционными системами и облачными провайдерами. Асинхронная система обмена сообщениями может быть реализована в режиме «один к одному» (очередь) или «один ко многим». Самыми популярными брокерами сообщений являются RabbitMQ и Apache Kafka.

При реализации рассматриваемого проекта использовался протокол RabbitMQ. Работа данного сервиса во многом похож на паттерн "Издатель/ Подписчик". Для запуска RabbitMQ был создал docker контейнер.

В нашем случае Users.Microservice является издателем, который при добавление нового пользователя отправляет данные через брокер RabbitMQ на Task.Microservice (рисунок 6). В Task.Microservice реализован слушатель (рисунок 7).

API Gateway – это промежуточный слой между клиентом и микросервисами, который упрощает доступ к API и управление ими.

В микросервисной архитектуре API Gateway выполняет следующие функции:

– Агрегация. API Gateway может объединять несколько микросервисов в один единый API, что упрощает доступ к ним клиентам и уменьшает количество запросов.

– Аутентификация и авторизация. API Gateway может проверять учетные данные клиента и разрешать или запрещать доступ к различным микросервисам в зависимости от прав доступа.

– Безопасность. API Gateway может обеспечивать безопасность передаваемых данных с помощью шифрования и других методов.

– Кеширование. API Gateway может кэшировать ответы микросервисов, чтобы уменьшить нагрузку на них и ускорить доступ клиентов.

– Мониторинг. API Gateway может отслеживать запросы и ответы на них для мониторинга производительности и обнаружения проблем.

– Трансформация данных. API Gateway может преобразовывать данные между форматами, что упрощает доступ клиентов к различным микросервисам, которые могут использовать разные форматы данных.

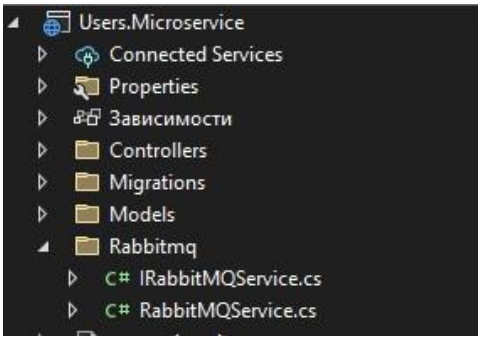


Рисунок 6

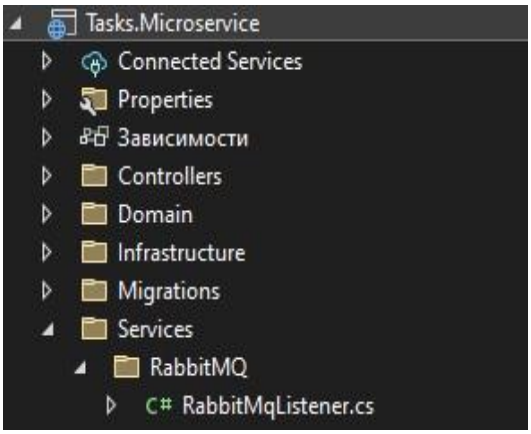


Рисунок 7

API Gateway является важным компонентом микросервисной архитектуры, так как позволяет упростить доступ к микросервисам и управлять ими централизованно, что снижает сложность и повышает надежность системы.

Для реализации API-шлюза в ASP.NET Core был применен пакет Ocelot (Nuget). В файле ocelot.json описывается конфигурация шлюза (рисунок 8).

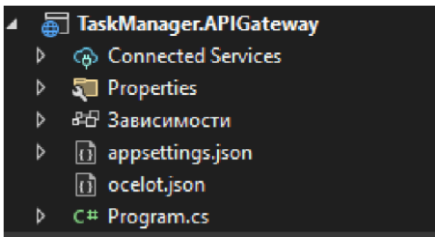


Рисунок 8

В ходе реализации проекта были выявлены определенные недостатки: значительные проблемы могут возникать при сбое связи, отката распределенных транзакций. Данные проблемы можно решить на этапе проектировании, уделив им особое внимание, но это в свою очередь может затянуть процесс и

повлечь увеличение накладных расходов. С другой стороны применение данной архитектуры позволяет достичь следующих положительных моментов: архитектура независимо масштабируется с использованием метода, называемого горизонтальным масштабированием, то есть масштабированием только тех сервисов, которые имеют более высокую нагрузку, кроме того, эти службы являются отказоустойчивыми и не имеют единой точки отказа.

Список литературы:

1. Роберт, Мартин. Чистая архитектура. Искусство разработки программного обеспечения. – Питер, 2018. – 352 с.
2. Ричардсон, Крис. Микросервисы. Паттерны разработки и рефакторинга. – Питер, 2019. – 544 с.
3. Мехди, Меджуи. Непрерывное развитие API. Правильные решения в изменчивом технологическом ландшафте / Мехди Меджуи, Эрик Уайлд, Ронни Митра, Майк Амундсен. – СПб. Питер, 2020. – 272 с.
4. Microservices Asynchronous Message-Based Communication. [Электронный ресурс]. – Режим доступа: https://medium.com/design-microservices-architecture-with-patterns/microservices-asynchronous-message-based-communication-6643bee06123, свободный. (Дата обращения 09.03.2023)
5. How to Build an Event-Driven ASP.NET Core Microservice Architecture. [Электронный ресурс]. – Режим доступа: https://itnext.io/how-to-build-an-event-driven-asp-net-core-microservice-architecture-e0ef2976f33f, свободный. (Дата обращения 09.03.2023)
6. A pattern language for microservices. [Электронный ресурс]. – Режим доступа: https://microservices.io/patterns/index.html, свободный. (Дата обращения 19.03.2023)