УДК 543.257.2

ПРОЕКТИРОВАНИЕ СИСТЕМ, УПРАВЛЯЕМЫХ СОБЫТИЯМИ

М.И. Давидовская, Е.С. Мойсейчик

Белорусский государственный университет, пр. Независимости, 4, 220030, г. Минск, Беларусь, davidouskaia@bsu.by

Ключевые характеристики ПО определяются ещё на этапе проектирования программной системы. От выбранной архитектуры и качества проектирования зависит дальнейшее поддержание и эксплуатация системы. Несмотря на то, что реализовать функциональные требования к ПО зачастую возможно при помощи любой архитектуры, далеко не каждый проект становится коммерчески успешным. Данная работа ставит своей целью рассмотреть ключевые проблемы основных подходов к проектированию архитектуры ПО и выделить случаи, когда уже на этапе проектирования стоит отдать предпочтение более сложным, но и более масштабируемым решениям, основанным на событиях.

Ключевые слова: архитектура ПО; распределенные приложения; проектирование; сервисы; события; асинхронное взаимодействие; брокер сообщений.

EVENT-DRIVEN SYSTEM DESIGN

М.И. Давидовская, Е.С. Мойсейчик

Belarusian State University, 4 Niezalieznasci Avenue, Minsk 220030, Belarus,
Corresponding author: davidouskaia@bsu.by

The key characteristics of the software are determined at the design stage of the software system. Further maintenance and operation of the system depends on the chosen architecture and design quality. Despite the fact that it is often possible to implement functional requirements for software using any architecture, not every project becomes commercially successful. This work aims to consider the key problems of the simplest software architectures and highlight cases when, already at the design stage, it is worth giving preference to more complex, but also more scalable, event-based solutions.

Keywrods: software architecture; distributed software; design; services; events; asynchronous communication; message broker.

Введение

Под архитектурой программного обеспечения (ПО) понимают определение основных структурных элементов программы, их интерфейсов и способа взаимодействия. На сегодняшний день разработан и изучен ряд архитектур распределённых приложений. Каждая из них имеет ряд преимуществ и недостатков и в значительной степени опреде-

ляет, насколько быстро и эффективно будет работать приложение, каковы будут затраты на его размещение, какие функции оно сможет реализовать и насколько просто будет внедрять, тестировать и поддерживать их.

Данная работа ставит своей целью проанализировать две основные архитектуры для серверной части распределённого приложения – монолитной и сервисной, – и выделить основные случаи, когда уместно отдать преимущество последнему из подходов на этапе проектирования системы.

Монолитным приложением или монолитом развертывания называют программную систему, которая может быть развернута только как единое целое [1]. Приложение такой архитектуры внутри может иметь логическое разбиение, но логические модули в таком случае не выступают в качестве самостоятельного приложения и должны разворачиваться как единый проект.

Простейшей распределённой архитектурой является архитектура «клиент – сервер». Абстрагируясь от количества и структуры клиентов такого приложения, можно утверждать, что с точки зрения архитектуры ПО серверной части в общем случае оно также представляет собой монолит развёртывания. Приложение на стороне сервера в такой системе разворачивается и исполняется как один исполняемый файл.

Такой подход имеет как ряд преимуществ, так и ряд очевидных недостатков:

1. Необходимость централизованно вносить изменения во все файлы монолитного приложения, чтобы не нарушить целостность и согласованность. Сложность логики приложения возрастает с ростом кодовой базы. Внесение любого даже незначительного изменения в приложение требует инициации всех этапов конвейера непрерывной доставки. Скорость разработки и доставки новых функций замедляется.
2. Сложное масштабирование системы и падение производительности при выполнении трудоёмких операций. При необходимости увеличить вычислительную мощность узла сервера для одной функции необходимо увеличивать количество вычислительных ресурсов для всего приложения. Вынужденная синхронность операций: в случае, когда операция может выполняться асинхронно, то есть клиент может не ожидать результата, монолитное серверное приложение не может обработать следующий запрос до тех пор, пока не завершится предыдущая операция.
3. Затруднены обновление и поддержка системы. Замена или обновление стека технологий требует модификации всей кодовой базы

приложения. Неисправности не локализованы в приложении и также требуют анализа всего приложения.

Работа ставит перед собой следующие задачи для достижения цели:

- Рассмотреть теоретические основы сервисных архитектур ПО.
- Показать, что сервисные архитектуры могут выступать в качестве систем, управляемых событиями.
- Обосновать, почему сервисные архитектуры решают упомянутые проблемы монолитных приложений.
- Выделить случаи, когда уместно использование средств сервисной архитектуры, на примере практической части.

Методы исследования, применяемые в работе:

- Теоретические: изучение литературы и других открытых источников, посвященных архитектуре ПО, тестированию производительности распределённых приложений, а также платформам, предоставляющим возможность написания серверного ПО для распределённых систем, таких как .NET и Java EE.
- Практические: обобщение и систематизация опыта путем проектирования веб-приложения для задач healthcare-области с применением сервисной архитектуры.

1. Методология исследования

Сервис – это независимое приложение, реализующее узкоспециализированные функции. Сервисная архитектура обобщенно определяется как функциональная декомпозиция приложения на отдельные сервисы. Ключевым аспектом здесь является изолированность сервисов, как единиц модульности приложения, и их независимое развертывание, а не размер сервисов [2].

Для взаимодействия с системой сервис предоставляет удаленное API – интерфейс, инкапсулирующий его реализацию. Вызов какого-либо кода сервиса вне API невозможен. Для достижения слабой связанности модулей системы, кроме сокрытия функций сервиса за API, сервисная архитектура часто предлагает раздельное хранение данных для каждого сервиса или части сервисов. В условиях большого количества модулей архитектура предлагает использование примитивных каналов сообщения между модулями и легковесных протоколов взаимодействия.

Таблица 1– Сравнение архитектурных стилей, основанных на сервисах

Критерий	**СОА**	**Микросервисы**
Хранение данных	Единая модель данных, глобальное хранение	Собственная модель данных, отдельное хранение
Взаимодействие сервисов	Сервисная шина, сложные протоколы удаленного вызова (SOAP)	Легковесные протоколы удаленного вызова (REST, gRPC), асинхронный обмен сообщениями
Внутренняя организация сервиса	Крупное приложение, монолитная архитектура	Небольшое приложение, предметно-ориентированная архитектура

Перечисленные черты позволяют выделить ряд преимуществ такой архитектуры в сравнении с другими подходами:

1. Гибкость и расширяемость системы. Легко добавлять и разворачивать новые сервисы и функции. Поддержка системы упрощена, контекст ошибки ограничен сервисом. Небольшие модули с чёткими функциями проще поддерживать командам разработки. Возможно заменять сервисы и использовать различные языки программирования (их версии) как для реализации сервисов, так и для хранения данных, если оно раздельное.
2. Баланс производительности и затрат на размещение. Возможность размещать сервисы на различных вычислительных узлах по необходимости. Возможность масштабироваться в рамках одного сервиса.

Из приведённого определения также очевидно, что при наличии в системе многих исполняемых файлов сервисов, развёрнутых раздельно, становится возможным реализовать асинхронный тип взаимодействия с клиентом. Для этого необходимо делегировать вычислительную работу одному сервису, в то время как остальные компоненты системы смогут по-прежнему обрабатывать синхронные запросы клиента.

Для реализации такой функциональности в сервисной архитектуре используется механизм обмена сообщениями между сервисами напрямую или через брокера, основанный на архитектурном шаблоне «издатель – подписчик». Клиент посылает запрос в виде сообщения и продолжает выполнение. Сервис обрабатывает сообщение клиента в период своей доступности и посылает ответ, если он должен это сделать. Это также делает возможным рассылать сообщение одного клиента множеству серви-

сов. Таким образом сервисная архитектура может выступать в качестве системы, управляемой событиями.

Основным стандартом, регламентирующим работу брокеров сообщений, является Advanced Message Queueing Protocol (AMQP) – открытый протокол для передачи сообщений. Он устанавливает наличие в брокере сообщений следующих компонент:

- Сообщения со структурированными заголовками, как единицы передаваемых данных.
- Очереди сообщений, хранящей сообщение клиента до тех пор, пока сервис не обработает его.
- Точек обмена – механизмов разного типа, распределяющих пришедшие в брокер сообщения по соответствующим очередям.

2. Результаты

Таким образом, основными теоретическими результатами работы являются:

1. Сравнительный анализ монолитной и сервисной архитектуры распределённых приложений.
2. Обоснование, почему сервисная архитектура ПО подходит для проектирования систем, управляемых событиями.
3. Обоснование, почему сервисная архитектура решает рассмотренные во введении проблемы монолитных приложений.
4. Определение задач, для которых предпочтительно использовать сервисную архитектуру ПО.

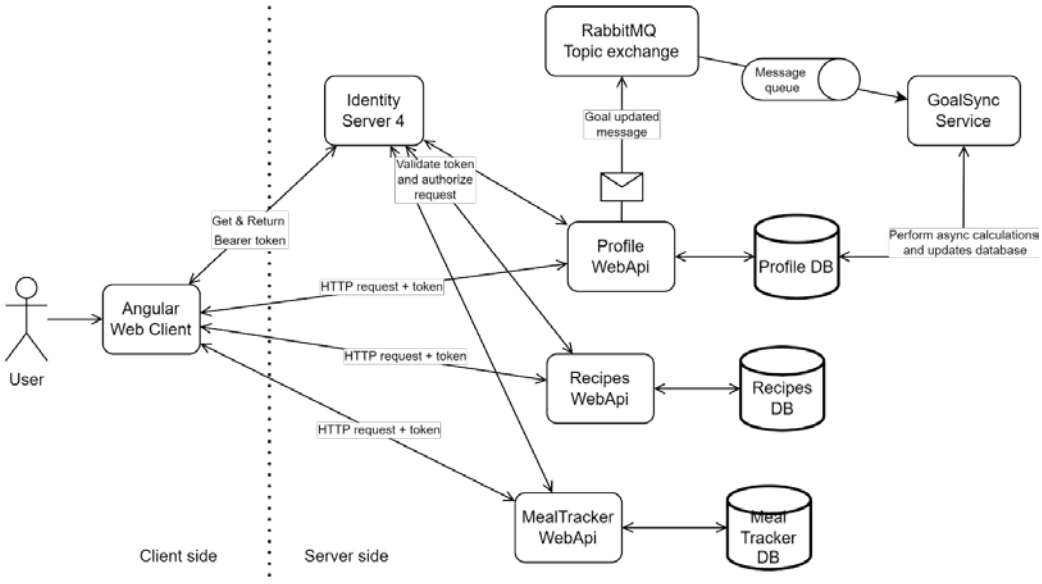

Рисунок – Структурная схема спроектированного приложения

Результатом практической части работы является веб-приложение для решения задач healthcare-задач пользователей, спроектированное в микросервисной архитектуре, реализованное в технологиях стека .NET [3] и готовое для размещения в сети [4]. В качестве брокера сообщений в работе использовался RabbitMQ, реализующий стандарт AMQP [5].

Библиографические ссылки

1. Wolff E. Microservices. Flexible software architectures. London: Pearson Education, 2016. 368 p.
2. Ричардсон К. Микросервисы. Паттерны разработки и рефакторинга. СПб. : Питер, 2019. 544 с.
3. Хордсал К. Микросервисы на платформе. СПб. : Питер, 2018. 352 с.
4. Мойсейчик Е.С., Давидовская М.И. Проектирование модульной архитектуры веб-приложения на примере Heathcare-приложения на основе микросервисов. // Научные исследования XXI века. 2022. № 1(15). С. 71–75.
5. RabbitMQ Features. Documentation [Электронный ресурс]. URL: https://www.rabbitmq.com/documentation.html. (дата обращения: 28.08.2022.)