УДК 004.942

НЕДОСТАТКИ МОНОЛИТНОЙ АРХИТЕКТУРЫ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Руденя А.С.
Научный руководитель - Попова Ю.Б., доцент, к.т.н.

Цель работы – исследовать особенности монолитной архитектуры программного обеспечения.

В настоящее время большое количество веб-приложений реализовано при помощи, так называемого, «Результата действия» и имеет монолитную архитектуру (Рисунок 1).

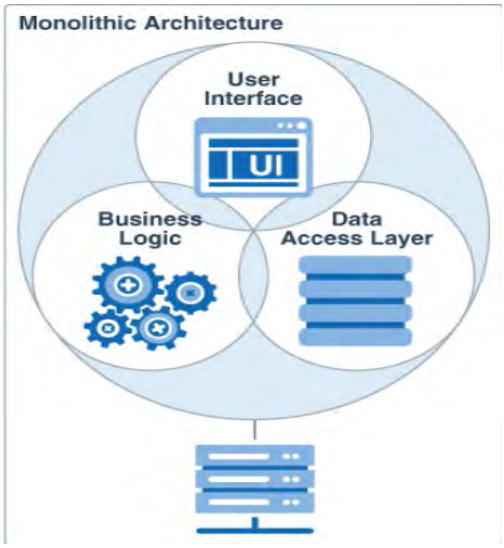

Рисунок 1 – Схема монолитной архитектуры [1]

Таким образом, когда пользователь обращается к ресурсу, он ожидает получить определенный ответ, например, в виде веб-страницы с некоторыми данными. На стороне сервера метод контроллера, получая параметры, обрабатывает их и формирует некоторый ответ в виде результата действия – HTML-страницы. Однако на сегодняшний день, когда созданы все условия для разделения логики серверной и клиентской частей, такой подход не является больше актуальным, т.к. сервер не должен тратить ресурсы в виде времени и памяти на то, чтобы сформировать HTML-страницу и отправить её пользователю [2]. Также такой подход делает систему негибкой, вследствие того, что нельзя использовать другой модуль с клиентской частью, поскольку все крепко связано с текущей реализацией.

Выделим основные недостатки применения монолитной архитектуры программных продуктов [3]:

1. Сильное зацепление. Когда серверная и клиентская части физически расположены рядом, существует большой соблазн зацепить их вместе, например, вставить какой-нибудь CSRF-токен прямо в HTML-код, который запускает клиентское приложение. Пока проект состоит из одной клиентской части и одной серверной — это не очень страшно. Однако при необходимости добавления ещё одной клиентской части, например, мобильного приложения, придётся решать все отложенные проблемы интеграции: придумывать протоколы взаимодействия, учиться делать прокси-запросы или работать с CORS (англ., Cross-OriginResourceSharing). Если бы проекты были разъединены с самого начала, то сами собой появились бы удобные для работы соглашения и протоколы — REST (англ.,RepresentationalStateTransfer) или GraphQL, авторизация через JWT (англ., JSON WebToken). Также было бы проще привлекать новых членов команды разработки, поскольку не приходилось бы изучать код серверной части и дорабатывать его под каждую новую клиентскую часть.

2. Ненужная коммуникация. В проекте с раздельными репозиториями серверной и клиентской частей разработчики никогда не будут спорить, где разместить файл Index.htmlи в какой момент лучше запускать Webpack, поскольку каждый будет заниматься своей частью проекта, используя любимые подходы. Разработчики серверной части не будут думать об инвалидации кэша CSS-файлов, а разработчики клиентской части не будут знать ничего о маршрутизации запросов. Если у серверной части сломалась публикация, разработчики клиентской частимогут не ждать, пока её починят, поскольку у них есть отдельный процесс, который обновляет сервер независимо.

3. Сложное развертывание приложения. Используя большие репозитории, происходит невозможность использования целого семейства сервисов, которые облегчают работу по публикации приложения. К примеру, Netlify без единой настройки может опубликовать клиентскую часть на своих серверах с CDN (англ., Content Delivery Network), HTTP2 и всеми актуальными технологиями. Проекты Zappa и Claudia делают тоже самое с клиентскими частями на Pyton и JavaScript. К сожалению, если такой сервис не поймёт, что именно находится в репозитории, то он не сможет помочь. Например, если Netlify увидит репозиторий с Django, в котором клиентская часть запрятана куда-то глубоко и собирается средствами фреймворка, то не выполнит никаких действий по публикации приложения.

4. Монолиты, как правило, перерождаются из своего чистого состояния в, так называемый, «большой шарик грязи»— состояние, возникшее вследствие

нарушения архитектурных правил, и имеющее сросшиеся со временем компоненты.

5. Это перерождение замедляет процесс разработки: каждую будущую функцию будет сложнее развивать из-за того, что компоненты растут вместе, их также необходимо изменять вместе. Создание новой функции может означать прикосновение, например, к 5 различным местам: 5 мест, в которых нужно написать тесты; 5 мест, которые могут иметь нежелательные побочные эффекты для существующих функций.

6. Потенциальные сложности масштабирования. Монолит легко масштабировать до тех пор, пока он не перерастёт в «большой шарик грязи», как упоминалось ранее. Масштабирование может быть проблематичным, когда только одной части системы требуются дополнительные ресурсы, ведь в монолитной архитектуре нельзя масштабировать отдельные части системы.

7. В монолите практически нет изоляции. Проблема или ошибка в модуле может замедлить или разрушить все приложение.

8. Строительство монолита часто протекает с помощью выбора основы. Отключение или обновление первоначального выбора может быть затруднительным, поскольку это должно быть сделано сразу и для всех частей системы.

Исследование монолитной архитектуры позволило выявить ряд ее недостатков и сделать выбор в пользу современной микросервиснойархитектуры [1]. Такой подход позволит разделить логику клиентской и серверной частей, система станет не такой завязанной и сможет работать по отдельности, т.е. в любой момент можно перейти на новую клиентскую часть, не касаясь при этом серверной части. Что касается производительности, то теперь серверу надо будет выполнить только бизнес-логику, не формируя при этом еще HTML-страницу, что значительно облегчит и ускорит работу сервера, а объем загружаемых ресурсов уменьшится в несколько раз.

Литература

1. Монолитная vsМикросервисная архитектура [Электронный ресурс]. – Режим доступа: https://dailycoding.io/article/4BABLqr0nR9E0gUU7Ko6 – Дата доступа: 19.05.2020.

2. C#/.NET. Контроллеры. Результаты действий [Электронный ресурс]. – Режим доступа: https://metanit.com/sharp/mvc/3.4.php – Дата доступа: 19.05.2020.

3. Почему лучше разделить фронтенд и бэкенд [Электронный ресурс]. – Режим доступа: https://bureau.ru/soviet/20200116/ – Дата доступа: 19.05.2020.