Azure

Как микросервисы общаются между собой?

https://auth0.com/blog/an-introduction-to-microservices-part-1/

> *No dependencies: Our microservice has no dependencies on other microservices.*

https://auth0.com/blog/an-introduction-to-microservices-part-2-API-gateway/

> *As microservices deal with very specific concerns, some microservice-based architectures tend to become "chatty": to perform useful work, many requests need to be sent to many different services. For convenience and performance reasons, gateways may provide facades ("virtual" endpoints) that internally are routed to many different microservices.*

В этом цикле статей написано следующее: микросервисы не должны зависеть друг от друга. Про микросервисы знает прокси-сервис, он же шлюз, который доступен снаружи. Он распределяет запросы по микросервисам и собирает результат.

Запрещает ли этот подход гексагональную архитектуру проекта? Нет. Рассмотрим на разных уровнях.

Исторически: сначала программа состояла просто из команд от начала и до конца. Затем повторяющийся код завернули в функции и процедуры. Общие функции и процедуры вынесли в библиотеки.

Важный момент: помимо управления сложностью был второй мотив — повторное использование кода.

Когда программы и библиотеки стали слишком большими, их стали разбивать на модули. Сокрытие данных впервые появилось именно в модулях, классов тогда ещё не было. Проблемы: зависимости между модулями, количество модулей.

Когда модулей в программе стало слишком много, их стали разбивать на уровни. Появилась многоуровневая архитектура. В простейшем случае трёхуровневая.

Обычно нижний уровень называют Data Access Layer, но Эрик Эванс, отец DDD, предлагает называть его инфраструктурным. Здесь размещается не только доступ к СУБД, но и доступ к таким сервисам, как Google Maps, PayPal, Twilio.

На верхнем уровне, уровне представления, реализованы механизмы доступа к предметной области: через Web API, через оконное приложение, через scheduled console application. В середине находится уровень предметной области, он же уровень бизнес-логики. Я предпочитаю первое название, потому что его использует Эванс, и потому что я писал игры. Какая может быть бизнес-логика в играх?

Классическое разбиение на модули предполагает упорядочивание зависимостей сверху вниз. Но это подход асимметричный, поскольку действительно незаменимым уровнем в программе является предметная область. Его и нужно сделать независимым, а все остальные уровни делать зависимыми от него.

Теперь мы можем перерисовать архитектуру проекта из кирпичной стены в звёздочку. В таком виде она называется гексагональной архитектурой. Это один из микросервисов? Проблема, как я её вижу, заключается в том, как на рисунке отображается шестиугольная архитектура.

Пчелиные соты.

Но исторически микросервисы появились в таких компаниях, как Amazon и Google, где уже работало монолитное ядро. Они решали проблему расширения функций ядра, таких, которые можно было бы сделать, не затрагивая центрального ствола.

Исторически это не соты, это тюльпан.

Как версионируете? Версионирование внешних и внутренних API.

Взаимодействие между прокси-сервисом и микро-сервисами делаем через слой интерфейсов контроллеров (паттерн MVC).

```
/// <summary>
/// Описывает интерфейс контроллера водителя.
/// </summary>
public interface IDriverController1
{
      /// <summary>
      /// Создает или обновляет водителя компании.
      /// </summary>
      /// <param name="employerId">Идентификатор компании.</param>
      /// <param name="driverId">Идентификатор водителя.</param>
      /// <param name="driverView">Информация о водителе..</param>
      void CreateOrUpdate(long employerId, long driverId,
DriverView driverView);
      .     .     .
}
```

Есть набор жёстких соглашений, как переводить C#-вызовы в REST. Если им следовать, у реализации интерфейса в микро-сервисе и в прокси-сервисе будет однозначная REST-интерпретация. Например:

void CreateOrUpdate(long employerId, long driverId, DriverView driverView);

CreateOrUpdate — из-за Update видно, что операция идемпотентная, значит, PUT, а не POST. Простые параметры будут в строке запроса (Query String-параметров у PUT/POST не бывает).

Значит, PUT /employers/{employerId}/drivers/{driverId}. DriverView будет передаваться в теле запроса.

Реализация в микросервисе:

```
[HttpPut]
[Route("employers/{employerId}/drivers/{driverId}")]
public void CreateOrUpdate(long employerId,
                           long driverId,
                           [FromBody] DriverView driverView)
{
      view.ThrowIfNull(nameof(driverView));
      driverRepository.CreateOrUpdate(employerId,
                                      driverId,
                                      driverView.GivenNames,
                                      driverView.FamilyName,
                                      (Gender?)driverView.Gender,

driverView.CitizenshipCountryId,
                                      driverView.ResidenceCityId,
                                      driverView.BirthDate,
                                      driverView.LicenseNumber,
                                      driverView.ExperienceFrom,
                                      driverView.LicenseEndedOn,
                                      driverView.PassportNumber,
                                      driverView.ContactPhone,
                                      driverView.IsEmployer);
```

Реализация в прокси-сервисе:

```
public void CreateOrUpdate(long employerId,
                           long driverId,
                           DriverView driverView)
{
    using (var client = new HttpClient())
    {
        var uri =
Uri($"employers/{employerId}/drivers/{driverId}");
        client.Put(uri, driverView);
    }
}
```

У интерфейса есть номер версии, который растёт. Один и тот же контроллер в микро-сервисе может реализовывать несколько интерфейсов.

Сейчас используем нотацию /api/v**N**/resources, — где **N** — номер версии. Говорят (говорят!), что так нельзя, надо информацию о версии передавать через HTTP Header. Возможно. Кажется, это не очень сложно сделать в ASP.NET Web API, но пока не придумали, как.

О том, почему нельзя номер версии в URI см. http://stackoverflow.com/questions/389169/best-practices-for-api-versioning, ответ №2.

Внутри решили переходить на кодогенерацию, используя Roslyn. Уже придумали, поставили эксперименты, но пока не выделили время.

Насколько рентабельно крутиться на Azure?

Не могу сказать, потому что нет опыта с другими облачными сервисами.. С учётом зарплаты админа выглядит очень неплохо. API App плюс база данных к нему вместе стоят где-то от 1000 до 2000 рублей. HTTPS сразу +2500 руб. Но за всем этим счастьем следит удалённый админ за 20 тысяч рублей, а не встроенный полноценный за 120.

EF Code First? Если да, то с какими проблемами сталкиваетесь?

Основная проблема — слияние миграций. Решили организационными средствами. Миграцию в каждый момент времени делает кто-то один.

Проблемы с old school разработчиками, потому что они old school. Типичная ситуация: эти ваши ORM вообще и EF — ерунда, потому что я запрос переписал, и он стал работать не три часа, а три секунды. Звучит логично, но.

Почему EF не смог построить эффективный запрос? Потому что исходный LINQ-запрос был сложным. Почему он был сложным? Потому что, вероятно, часть бизнес-логики реализована в базе данных. Это то, как люди писали 15–20 лет назад, и с этого надо переучиваться.

По закону Мура в среднем производительность систем возрастает в 2 раза каждые 2 года. Он до сих пор действует. Это значит, что сейчас серверы в 1000 раз производительнее, чем были 20 лет назад. Это значит, что сейчас мы не должны решать те проблемы производительности, которые решали 20 лет назад.

Premature optimization is a root of evil — сначала напишем правильно, потом будем оптимизировать.

Как используете Redis? Кэш и (или) pub-sub?

Redis используем не только как кэш, но и как альтернативу СУБД. Яркий пример: refresh tokens. Время их жизни достаточно большое, и по уму их надо хранить в базе данных. Но база данных это тяжело. А вот Redis — легко. Не надо разрабатывать и поддерживать схему БД. Не надо думать о миграциях. В отличие от memcached, Redis обеспечивает долговременное хранение данных (https://redis.io/topics/persistence).

Redis (как и memcached) «встроен» в Azure, то есть не требует самостоятельного развертывания и настройки, что упрощает администрирование.

В явном виде для реализации Publisher-Subscriber не используем. Возможно, если будут сценарии, попробуем. Но в Azure есть Service Bus — встроенный MQ-сервис.

В неявном виде используем для поддержки масштабирования SignalR. Но это делается очень прозрачно, почти ничего не приходится писать руками.

DDD применяете?

Да. Независимо от Azure-не Azure — DDD категорически прочищает голову и упрощает проектные решения.

Проблема: DDD — это непросто. Должен быть опыт в ОО, плюс к нему, должны быть неудачи, чтобы DDD стал доступен. Из собственного опыта: читаю Эванса и восклицаю: «ага, вот как это надо было делать в том проекте!»

Важный момент сейчас, если приходится писать сложный код, первая мысль: не понимаю предметную область, надо разговаривать с заказчиком. Это результат принятия DDD: если программист понимает предметную область, его код чистый и понятный.

Что нравится: есть готовая карта, как раскладывать предметную область на части. Сущности, объекты-значения, агрегаты, сервисы — отнеси класс к такой-то категории.

Что ещё нравится: Эванс помнит про дырявые абстракции и утверждает, что реализация может влиять на модель.

Микросервисная архитектура.

Как конкретно организовано: микросервис на самом деле не совсем микро. Видел в Node.js простые реализации, но C#, как и Java накладывает способы проектирования.

Микросервис — это 4 проекта: предметная область, Web API, доступ к данным, тесты. Иногда больше, если несколько инфраструктурных проектов, например, Maps.Google.

Entity Framework, Code First, миграции, Autofac.

Прокси-сервис реализует общение с внешним миром. SignalR, отправка SMS, аутентификация, кеширование.

Микросервисы не знают друг о друге и о прокси-сервисе. Есть центральный микрос-сервис, который на самом деле совсем не микро. Вокруг него несколько мелких, которые про него «знают» на уровне уникальных идентификаторов, но реальной зависимости на уровне кода нет, то есть никаких references между проектами не протянуто.

Прокси знает о микросервисах и умеет собирать информацию из них. Но в нём логика очень простая: 1-2-3 вызова, сборка данных в один большой объект (паттерн *Удалённый фасад*), и всё.

Как выкладываем в Azure?

Используем TFS Online, который сейчас называется Visual Studio Team Services. Там есть встроенный Build Server, где поддерживает публикация в Azure. Сейчас три контура:

- Dev. Выкладываем из ветки master каждый раз, когда в неё поступают обновления. Прогоняем набор быстрых тестов. В случае ошибок шлём письма. Контур работает постоянно и используется для ручного тестирования и разработки внешних программ.
- Stage. Выкладываем вручную, когда закончен спринт.
- QA (quality assurance). Выкладываем из ветки master в три часа ночи каждый рабочий день. Прогоняем набор медленных и нагрузочных тестов. Чистим базу. Контур работает только на время тестов, стоит дёшево. Фактически, обеспечивает continuous integration.

Набор интеграционных тестов написали на JavaScript. Но можно и на Python: всё зависит от того, какие спецы есть. На Build Server есть вот это:

https://www.visualstudio.com/en-us/docs/build/concepts/agents/hosted#software

SoapUI там нет, но есть Java, поэтому можно загрузить SoapUI во время билда и прогнать SoapUI-тесты. Мы поставили эксперимент, работает. Но для нас сложно, нет никого, кто мог бы писать интеграционные тесты в SoapUI.

Чем пользуемся для гарантированной доставки?

Гарантировать доставку невозможно. Можно знать, доставлен пакет или не доставлен. Мы используем SignalR для быстрого уведомления, потому что push notification не гарантирует короткого времени доставки.

Про что надо знать: по умолчанию, в Azure закрыты Web Sockets. Надо зайти в Application Settings и поднять, после этого SignalR перестанет использовать long polling.

Если внешний прокси-сервис масшабируем (то есть можно развернуть несколько инстансов), нужно обеспечить взаимодействие между ними через Redis для отправки информации правильному получателю. Подробности: https://docs.microsoft.com/en-us/aspnet/signalr/overview/performance/scaleout-with-redis

Всё очень прозрачно.

Сталкивались с чем: если у вас REST, то не может быть никакого SignalR. Позиция кажется странной. В целом: внешние разработчики боятся SignalR. Магия.