SolarLab>_

Применение микросервисов. Взаимодействие между сервисами

Содержание

1. Для чего мы изобретали свой велосипед?
2. Общая информация о микросервисах (отличия от монолита)
3. Текущие возможности по взаимодействию между микросервисами
4. Почему мы выбрали микросервисы?
5. Почему мы выбрали AMQP?
6. Описание нашей реализации
7. Примеры использования
8. Выводы
9. Ссылки
10. Вопросы и ответы

SolarLab>_

Для чего мы изобретали свой велосипед?

Для чего мы изобретали свой велосипед?

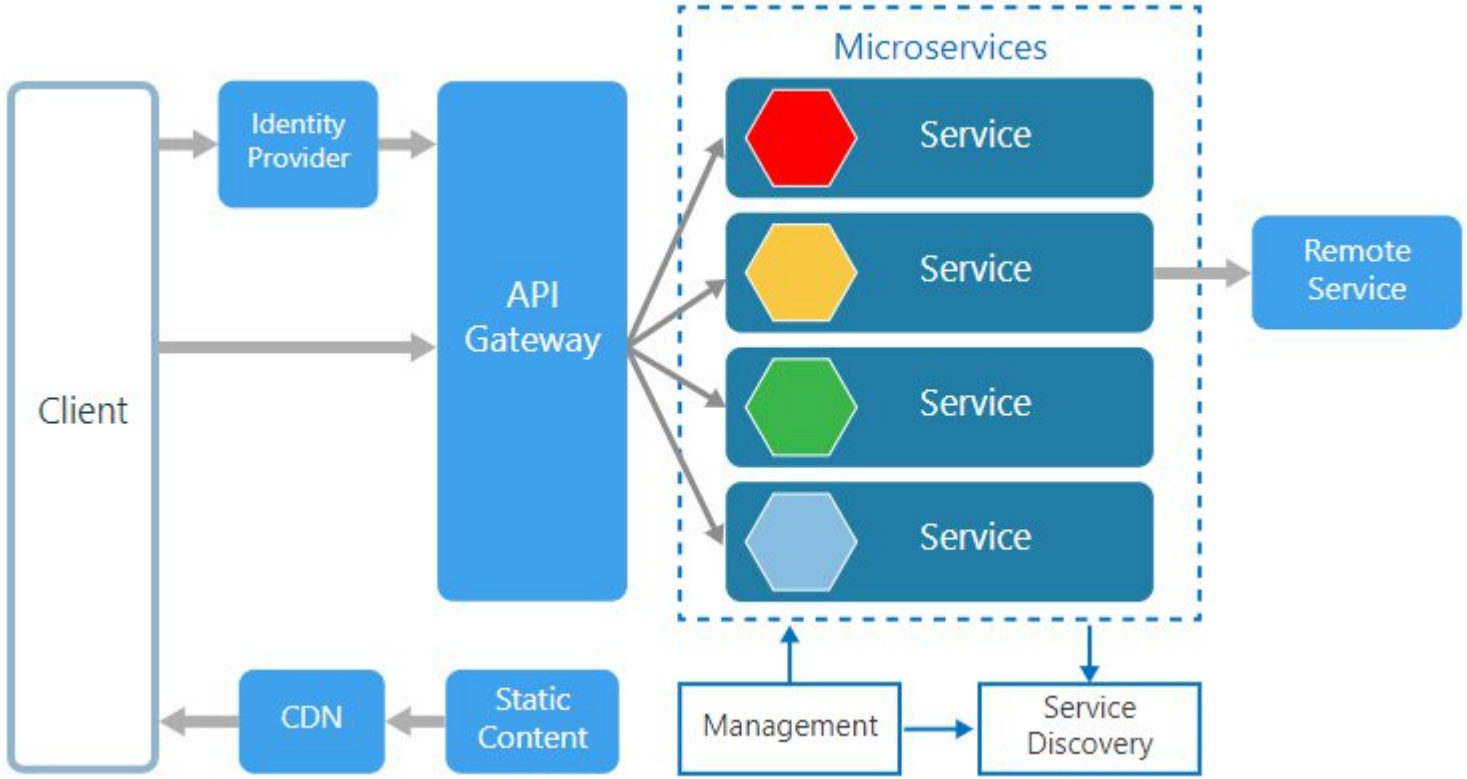
У нас написано приложение для имущественных торгов, которое реализовано на микросервисах.

Все наши микросервисы взаимодействую через шину данных. Наш проект BusManager решает задачу взаимодействия между микросервисами посредством AMQP протокола. В нашем случае реализация идёт на RabbitMq + MassTransit.

SolarLab>_

Общая информация о микросервисах (отличия от монолита)

Архитектура микросервисов



Архитектура микросервисов представляет собой набор небольших автономных служб. Каждая служба является самодостаточной. В каком-то смысле микросервисы являются естественным результатом развития сервисноориентированных архитектур (SOA). Но микросервисы и SOA отличаются. Ниже приведены определяющие характеристики микросервисов.

Характеристики микросервисов

- Архитектура микросервисов состоит из небольших, независимых и слабо связанных между собой служб.
- Каждая служба является отдельной базой кода, которой может управлять небольшая команда разработчиков.
- Службы можно развертывать независимо друг от друга. Разработчики могут обновлять существующую службу без повторной сборки и повторного развертывания всего приложения.
- Службы отвечают за сохранение собственных данных или внешнего состояния. В этом состоит отличие от традиционной модели, в которой сохранение данных обрабатывается на отдельном уровне.
- Службы взаимодействуют между собой с помощью четко определенных API-интерфейсов. Сведения о внутренней реализации каждой службы скрыты от других служб.
- Службы не должны(но могут) совместно использовать один и тот же стек технологий, библиотеки или платформы.

Отличие микросервисов от монолита

Микросервисы:

- Набор небольших сервисов
- Каждый сервис работает в собственном процессе и коммуницирует с остальными используя легковесные механизмы.
- Сервисы построены вокруг бизнес-потребностей
- Развертываются независимо с использованием полностью автоматизированной среды.
- Могут быть написаны на разных языках и использовать разные технологии хранения данных.

Отличие микросервисов от монолита

Монолит:

- Построенное как единое целое.
- Любое изменение - пересборка и развертывание новой версии
- Вся логика по обработке запросов выполняется в единственном процессе
- Масштабировать горизонтально
- Труднее сохранять хорошую модульную структуру, изменения логики одного модуля имеют тенденцию влиять на код других модулей.
- Масштабировать приходится все приложение целиком

SolarLab>_

Текущие способы взаимодействия между микросервисами

Текущие способы взаимодействия между микросервисами

- HTTP
- AMQP
- gRPC
- RPC

- И другие

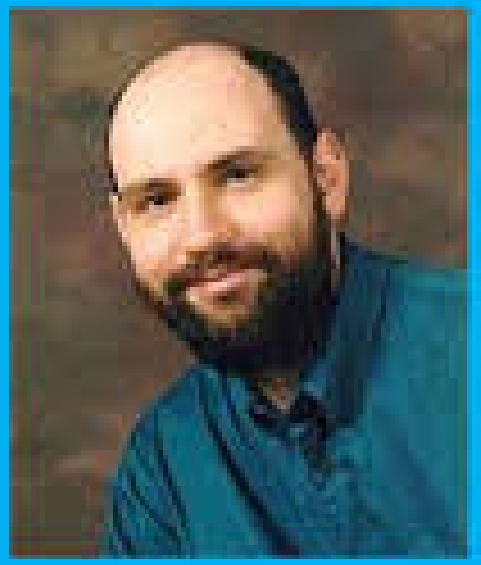
SolarLab>_

Почему мы выбрали микросервисы?

Во многом мне помог вот это человек

Во многом мне помог вот это человек

Мартин Фаулер



Автор большого количества паттернов проектирования, один из соавторов UML и XP. Вероятно, один из крупнейших специалистов практиков в области Software Engineering нашего времени.

SolarLab>_

Почему мы выбрали AMQP?

Почему мы выбрали AMQP?

- Изначально архитектурно мы планировали асинхронное взаимодействие через шину данных
- Слишком много сервисов, развёрнутых как WebApi, только для внутреннего использования
- Нужен отдельный DiscoveryService для WebApi
- Легко масштабировать, при запуске нового инстанса, не нужен LoadBalancer

Преимущества нашего подхода

Вес микросервиса

Сам микросервис не содержит дополнительных нугетов для webapi, которые ему не нужны.

Сохранение дискового пространства

Если вы разворачиваете свой сервис на платных хостингах, где важно дисковое пространство, то ваш микросервис будет «весить» гораздо меньше.

Не нужен Discovery Service

Для работы через http или gRPC нам необходимо знать адрес конечного компьютера. Для этого обычно используется Discovery Service.

Плюс мы можем развернуть несколько сервисов, и они будут обрабатывать сообщения параллельно. Первый кто считал, тот и обрабатывает.

Не нужен LoadBalancer

Для работы через http на боевом стэнде, используя несколько инстансов, необходим LoadBalancer. Что тоже влечёт за собой дополнительную нагрузку на инфраструктуру.

Недостатки нашего подхода

Взаимодействие для других платформ

Большинство платформ поддерживает RabbitMq, но его реализация довольно сложна (в плане Request / Response)

Связь через контракты

При ошибке в написании контрактов, могут возникнуть сложность для новичков на проекте

Отказоустойчивость

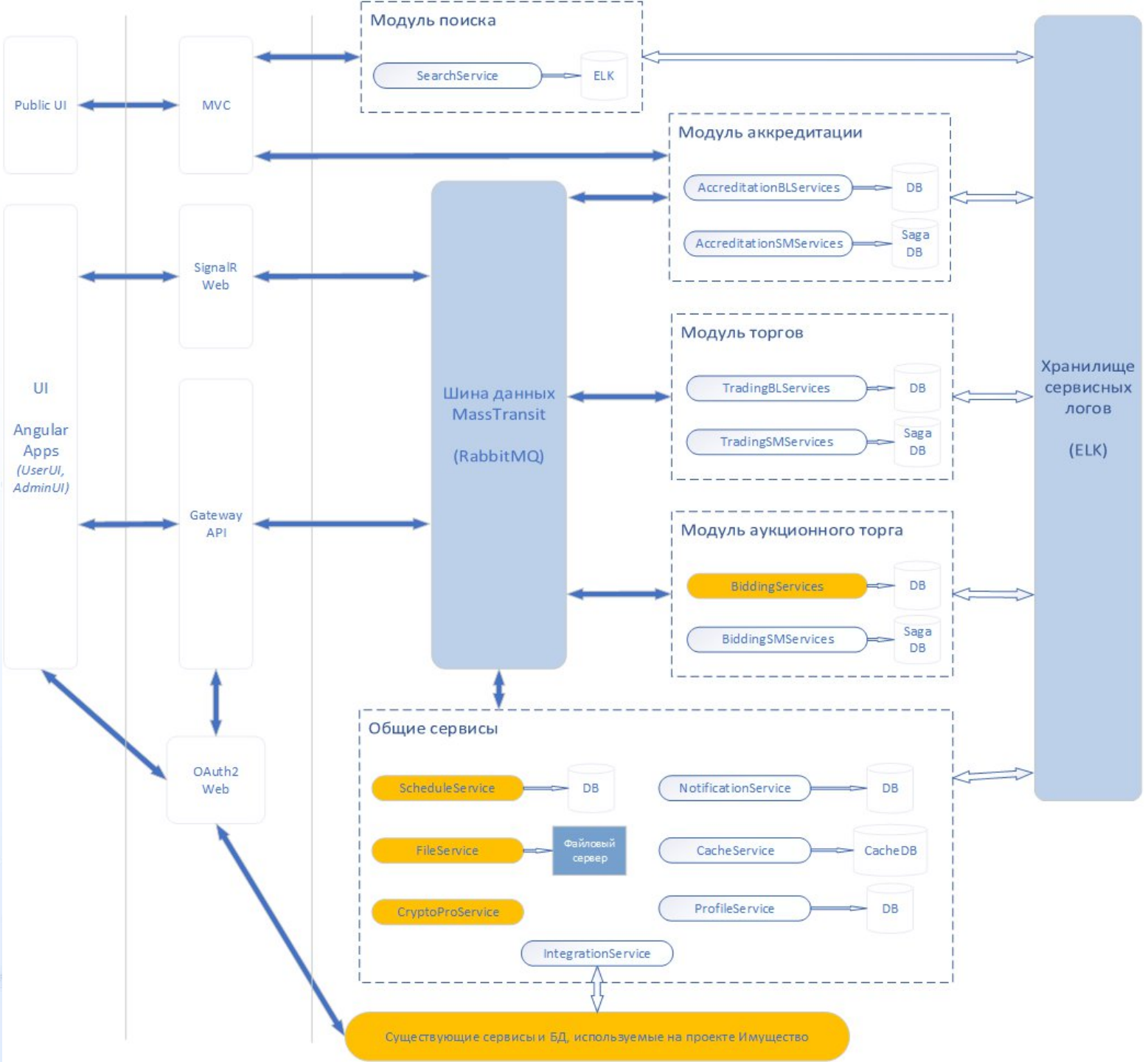
При выходе из строя Rabbit все запросы полностью не проходят. Так же и при очень больших объёмах сообщений через шину могут быть задержки

Сериализация данных через шину

Потенциально могут быть проблемы при сериализации данных через шину, например для decimal.MaxValue.

SolarLab>_

Архитектура нашего приложения


Модуль поиска
SearchService
ELK
Public UI
MVC
Модуль аккредитации
AccreditationBLServices
DB
AccreditationSMServices
Saga DB
SignalR Web
UI
Angular Apps
(UserUI, AdminUI)
Шина данных MassTransit
(RabbitMQ)
Модуль торгов
TradingBLServices
DB
TradingSMServices
Saga DB
Хранилище сервисных логов
(ELK)
Gateway API
Модуль аукционного торга
BiddingServices
DB
BiddingSMServices
Saga DB
Общие сервисы
ScheduleService
DB
NotificationService
DB
FileService
Файловый сервер
CacheService
CacheDB
CryptoProService
ProfileService
DB
IntegrationService
OAuth2 Web
Существующие сервисы и БД, используемые на проекте Имущество

SolarLab>_

Описание нашей реализации

Как мы сделали общение между микросервисами

- Все общие контракты между микросервисами находятся с отдельном нугет пакете
- Каждый микросервис свои контракты «наружу» выставляет через отдельный нугет пакет
- Для обращения к микросервису через Request / Response используются специальные контракты. Для контракта Request – разработчик обязан реализовать интерфейс **IWithQueueName**. Для контракта Response наследование идёт от **BaseResponse<T>**
- Проект BusManager размещён с открытым исходным кодов на github - https://github.com/SolarLabRU/BusManager
- Контракты и реализация для BusManager добавлены в открытый nuget репозиторий на nuget.org
- Мы используем для общения MassTransit + RabbitMq

SolarLab>_

Примеры использования

Пример использования Request / Response

- Разработчик создаёт новый контакт Request

```
public class MyRequest : IWithQueueName
{
    public string QueueName =>
      "MyService.QueueForMyRequest"
    // сюда можно добавить любые
    // необходимые поля для запроса
}
```

```
public class MyResponse : BaseResponse<bool>
{
    // результат Response может быть любым
    // в том числе DTO
}
```

- Создаём отдельный нугет пакет для этих классов и пушим его в нугет репозиторий(у нас свой приватный)
- Всё, ваши контракты готовы

Пример использования Request / Response

- Далее там, где нам нужны данные из нашего сервиса(например Gateway) подключаем IBusManager и его имплементацию MassTransitBusManager. Делаем inject(IBusManager busManager) где нужен вызов и вызываем:

```
var request = new MyRequest();

var response = await busManager
.Request<MyRequest, MyResponse>(request);
```

Дальше можно обрабатывать response как вам удобно

Пример использования BusManager в микросервисе

- В конкретном микросервисе мы «подвязываемся» на прослушивание очереди и ждём сообщения. Имя очереди должно совпадать с именем очереди в контракте:

```
var busConfig
= new Dictionary<string,
Action<IRabbitMqReceiveEndPointConfiguration>>
{
    {
        "MyService.QueueForMyRequest",
        e => e.Handler<MyRequest>(async ctx =>
        {
            // может добавить serviceProvider
            // может добавить serviceProvider.CreateScope()
            // можем добавить обработку ошибок
            var resultFromBL = await someBL.SomeMethodAsync()
            var result = mapper.Map<ResultFromBusinessLogic,
            await ctx.RespondAsync<MyResponse>(result);
        })
    }
};
```

Пример использования BusManager в микросервисе

- Так же наш BusManager позволяет инициализировать начало работы с очередью через настройки

```
var busManager = serviceProvider
  .GetRequiredService<IBusManager>();
busManager.StartBus(settings);
```

- И в конце busManager.StopBus()

SolarLab>_

Выводы

Выводы

Приложение можно писать как монолитное, так и используя сервисы, так и микросервисное или любым другим удобным для вас способом.

Микросервисы — хорошо. Монолит — хорошо. Хороший код — хорошо. Рабочий код — хорошо.

"Отказоустойчивость", "распределенность", "масштабируемость" это безусловно хорошо, но не это главное преимущество микросервисной архитектуры. Самое важное, что дает такой подход — удешевление стоимости внесения изменений.

Для нашего проекта отлично подошли микросервисы. И мы организовали удобную для нас работу через шину Rabbit используя

SolarLab>_

Ссылки

Ссылки

- **Стиль архитектуры микросервисов от Майкрософт** https://docs.microsoft.com/ru-ru/azure/architecture/guide/architecture-styles/microservices
- **Microservices от Фаулера** https://martinfowler.com/articles/microservices.html
- Microservices. Как правильно делать и когда применять? https://www.dataart.ru/news/microservices-kak-pravil-no-delat-i-kogda-primenyat/
- Переход от монолита к микросервисам https://habr.com/ru/post/305826/
- Микросервисы: пожалуйста, не нужно https://habr.com/ru/post/311208/
- Удалённый вызов процедур на Вики https://ru.wikipedia.org/wiki/%D0%A3%D0%B4%D0%B0%D0%BB%D1%91%D0%BD%D0%BD%D1%8B%D0%B9_%D0%B2%D1%8B%D0%B7%D0%BE%D0%B2_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D0%B4%D1%83%D1%80
- Исходники BusManager’a https://github.com/SolarLabRU/BusManager

Вопросы?

SolarLab>_

Спасибо за внимание