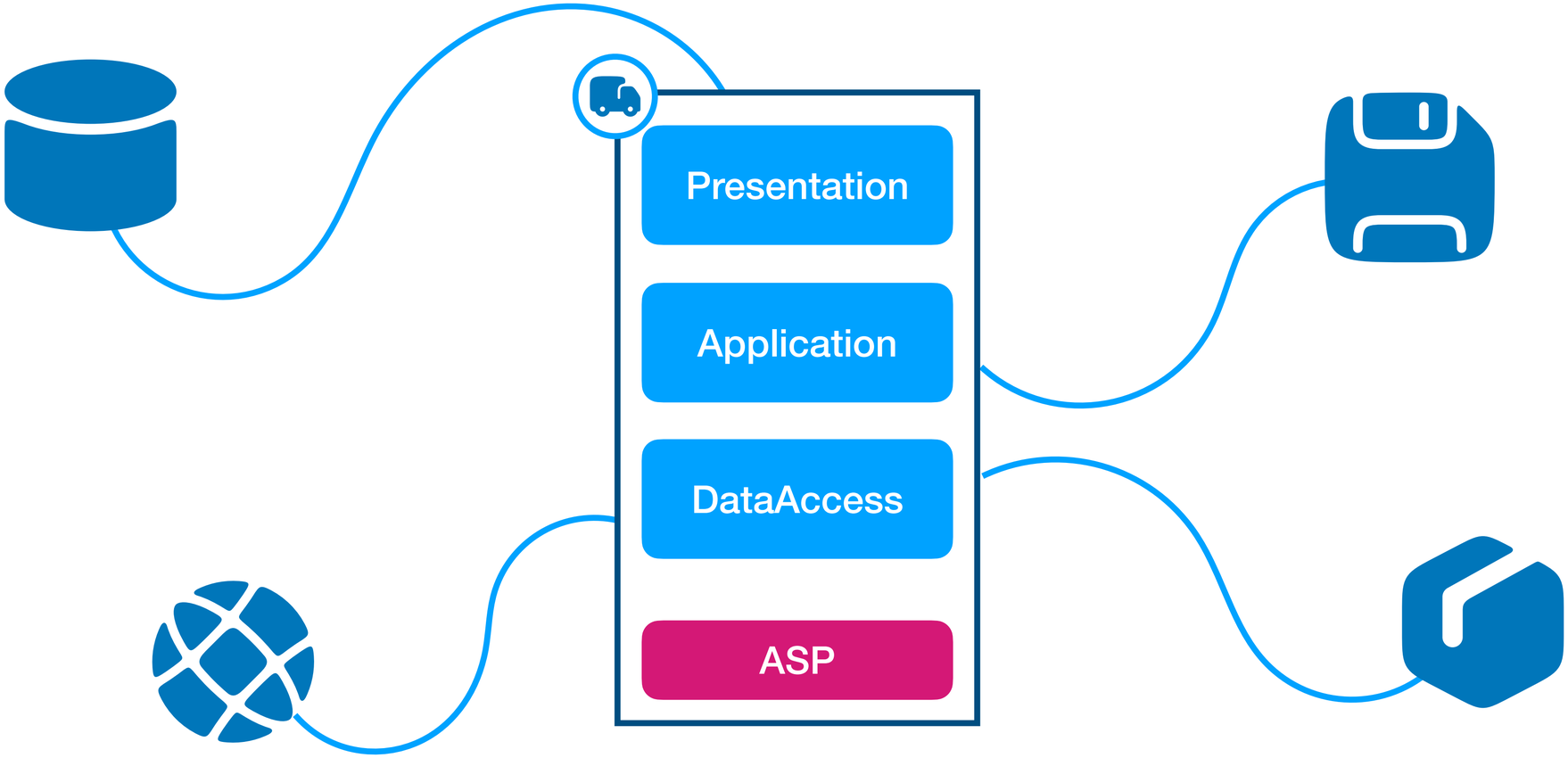
управление данными в микросервисах на C#

явное межсервисное взаимодействие

концепция микросервисов

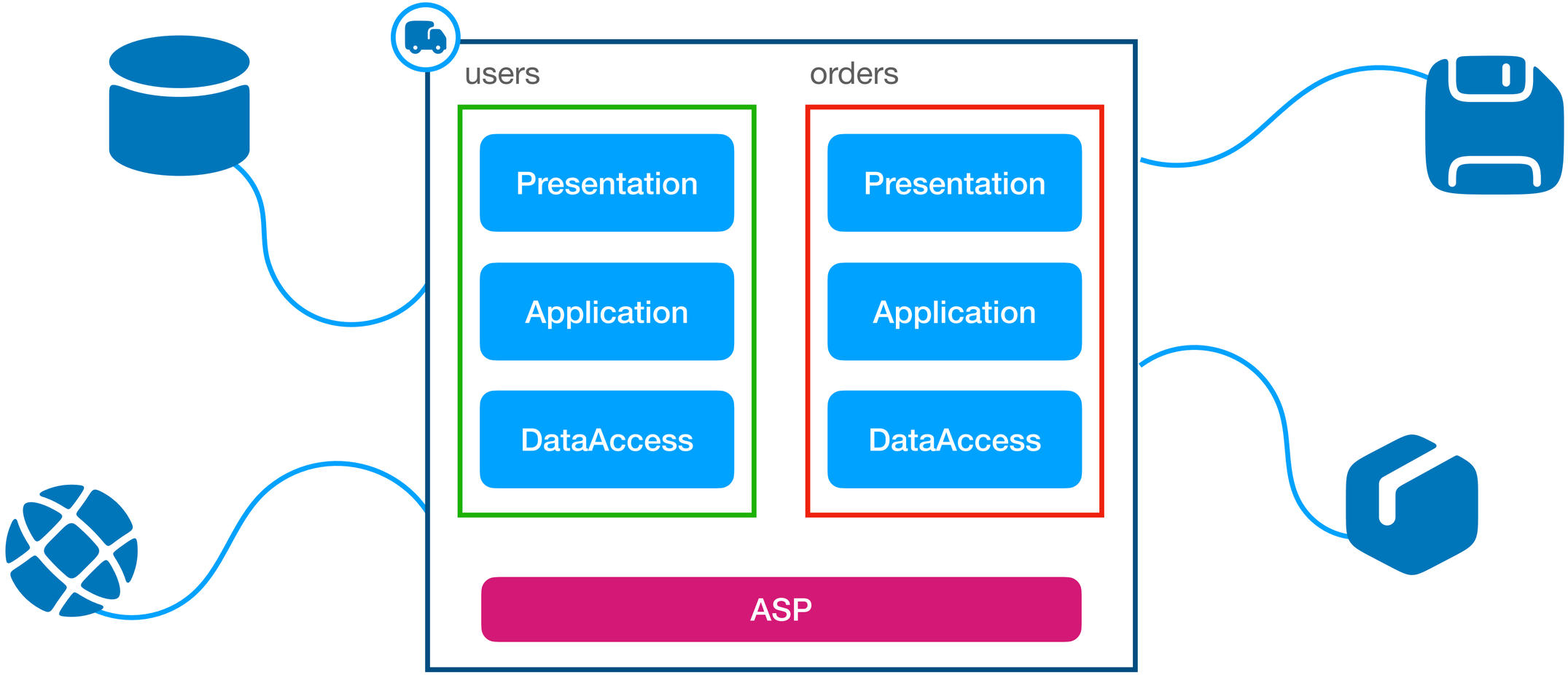
концепция микросервисов

монолиты



концепция микросервисов

модульные монолиты

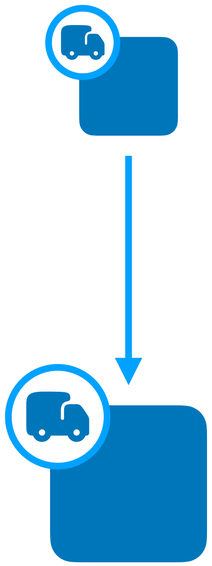


концепция микросервисов

масштабирование

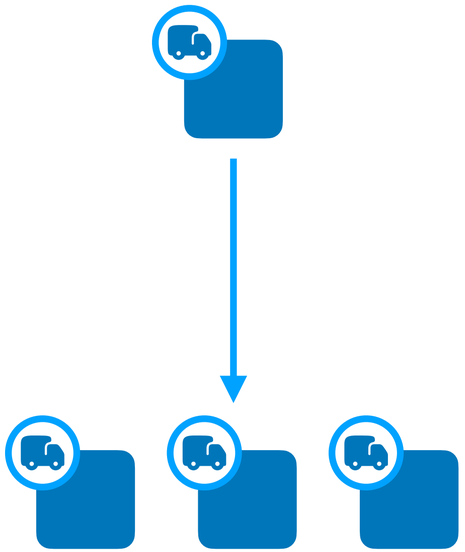
Вертикальное

увеличиваем мощность
одного экземпляра



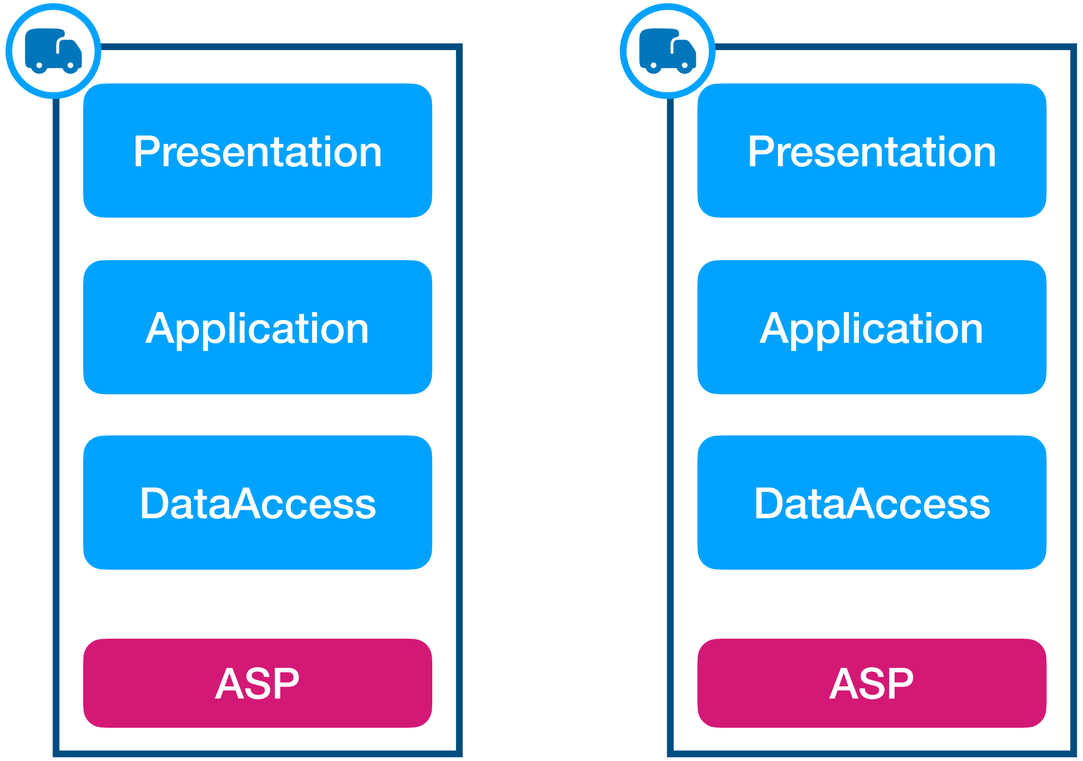
Горизонтальное

увеличиваем количество
экземпляров



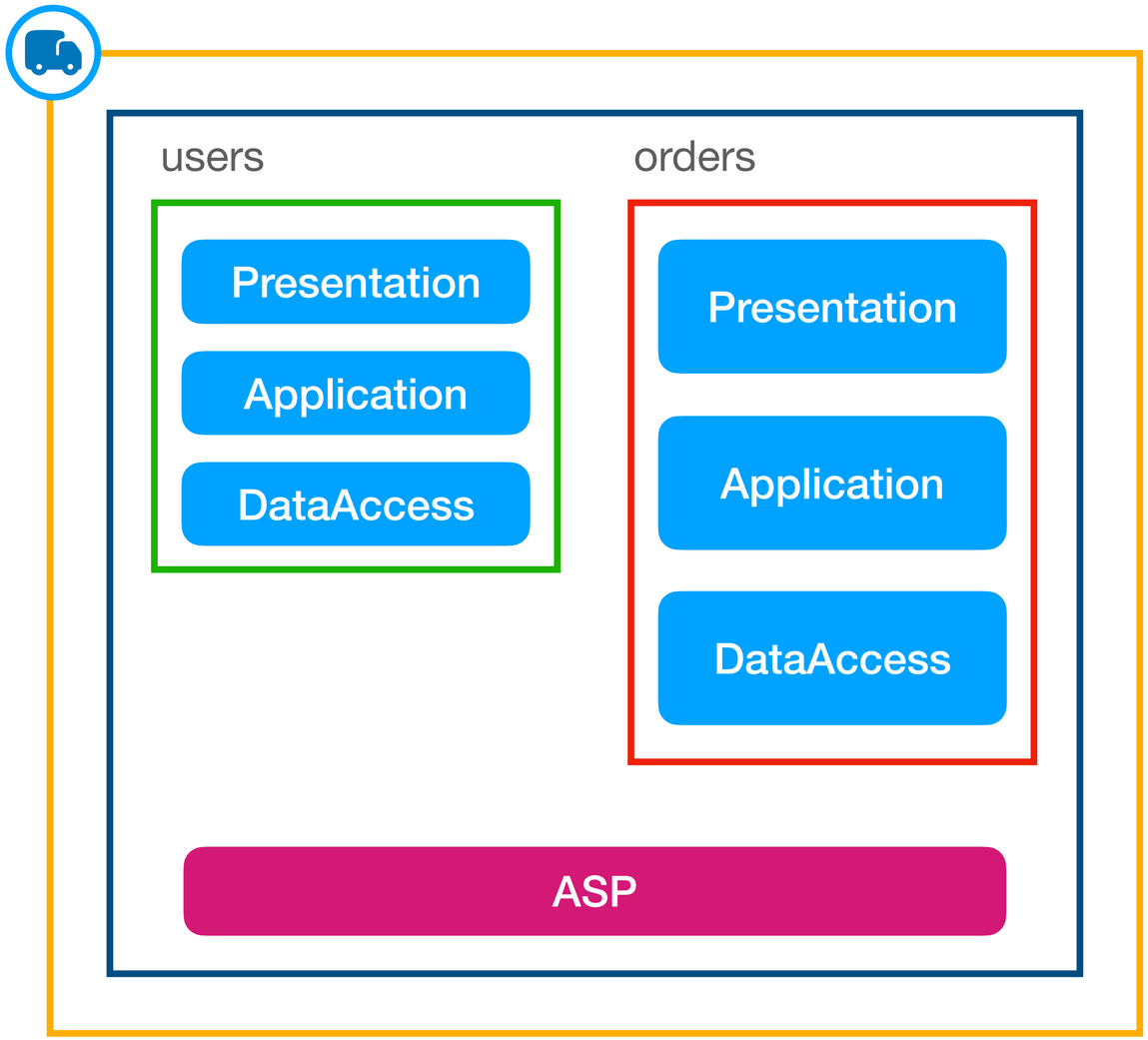
концепция микросервисов

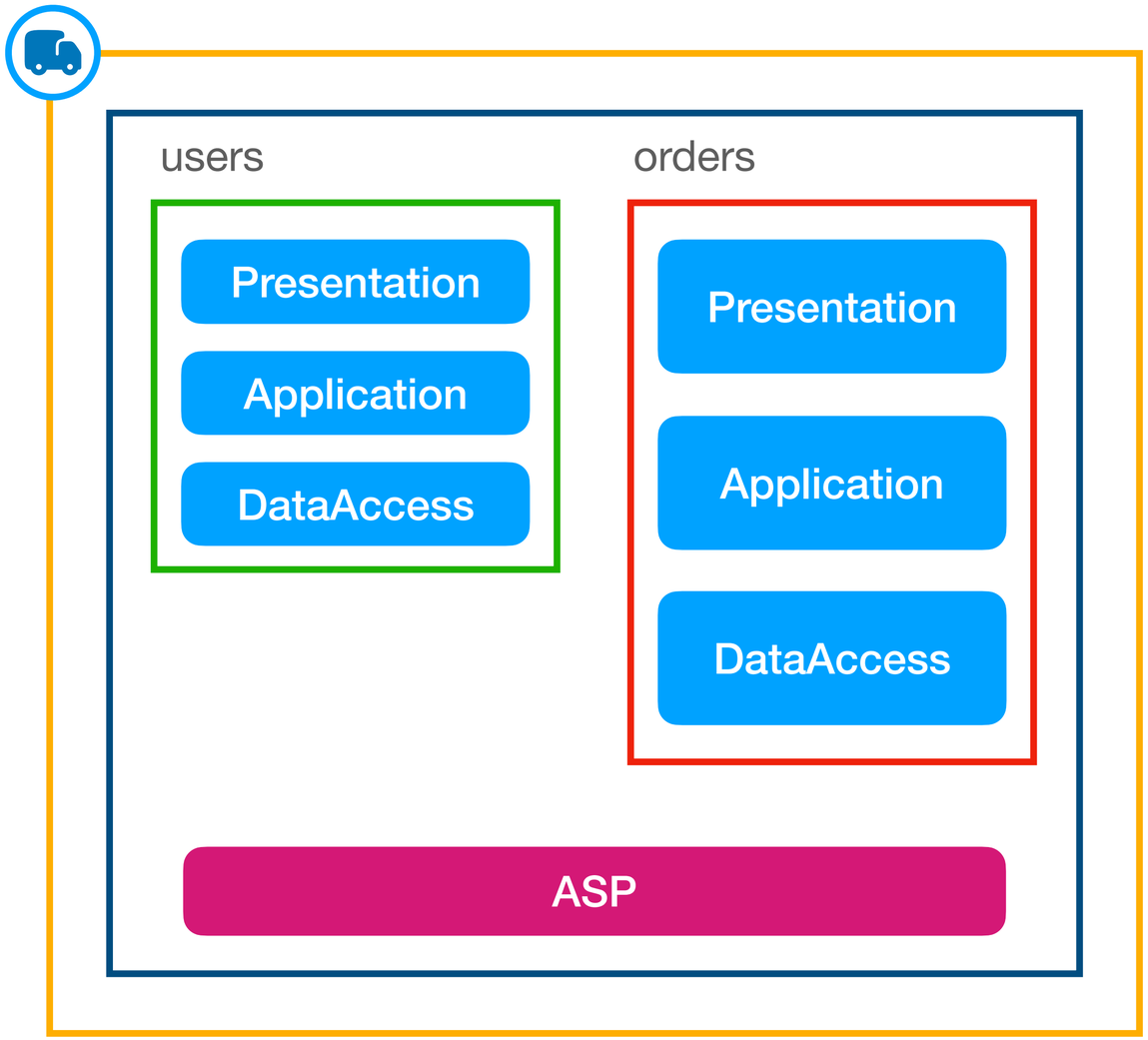
распределённые монолиты



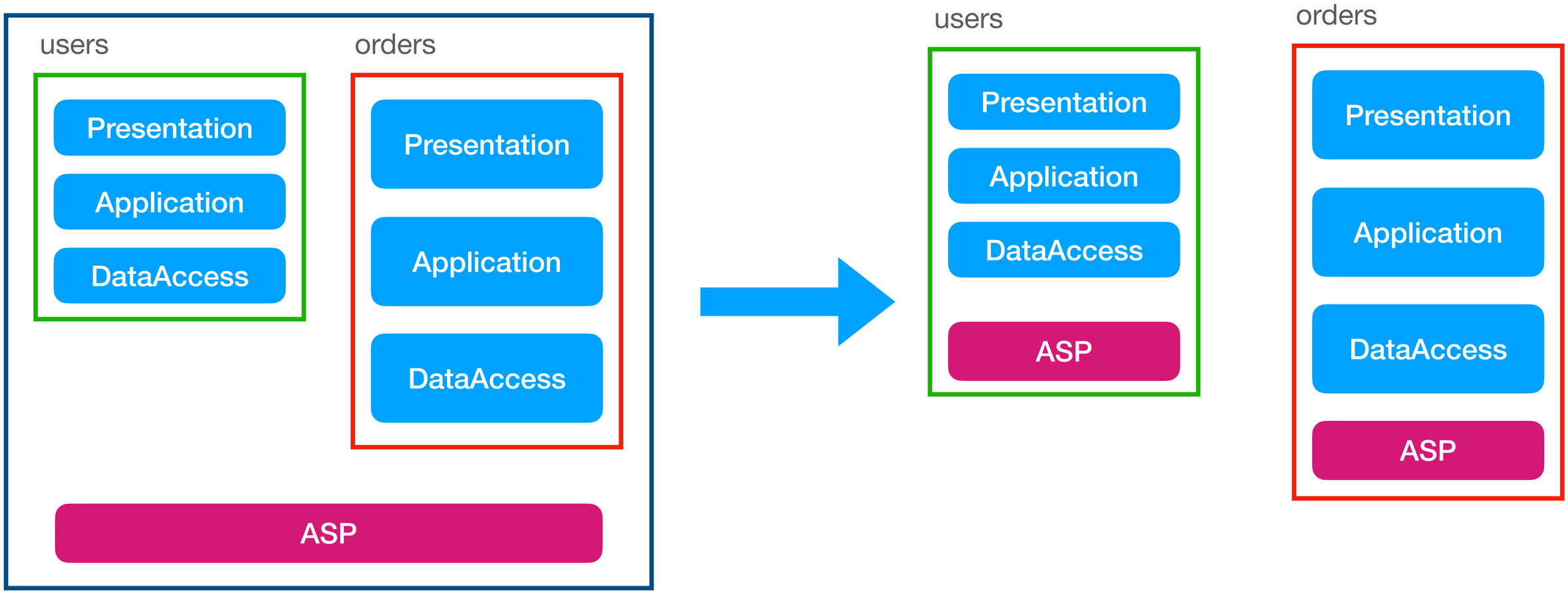
концепция микросервисов

распределённые монолиты



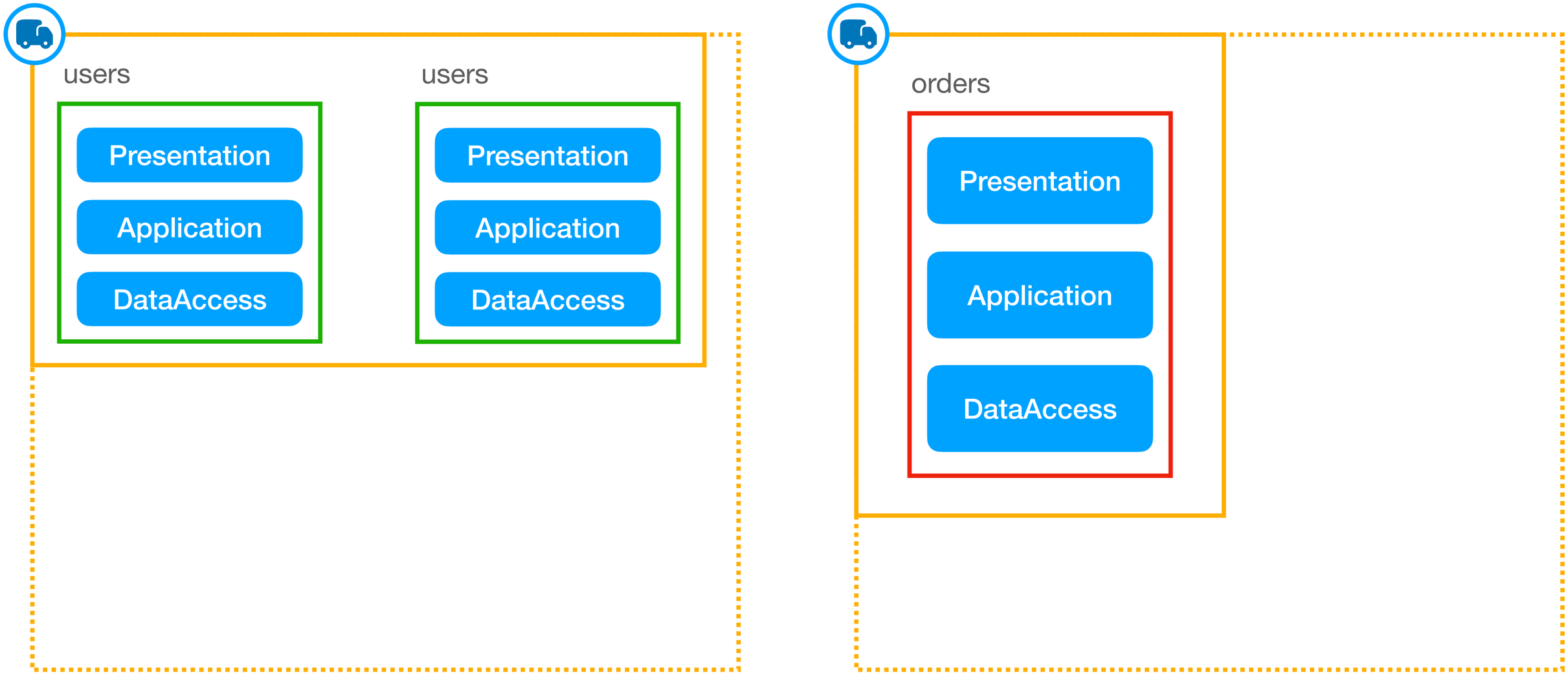


концепция микросервисов



концепция микросервисов

масштабирование



концепция микросервисов

проблемы архитектуры

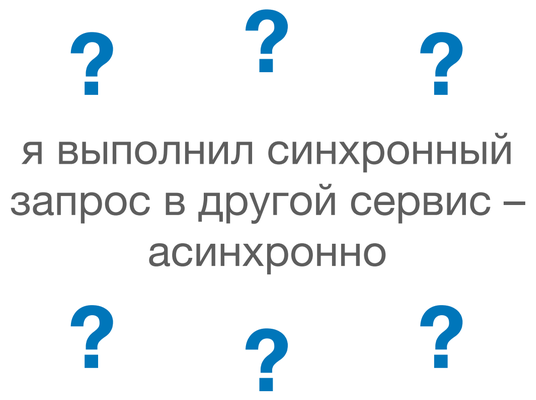
- б**о**льшая сложность при реализации
- б**о**льшая сложность при развертывании
- сложность при взаимодействии модулей
  - вызываем реализации не из того же процесса
  - на операции в других сервисах не действуют те же транзакции
- основная сложность – проектирование межсервисного взаимодействия

явное межсервисное взаимодействие

межсервисное взаимодействие

терминология

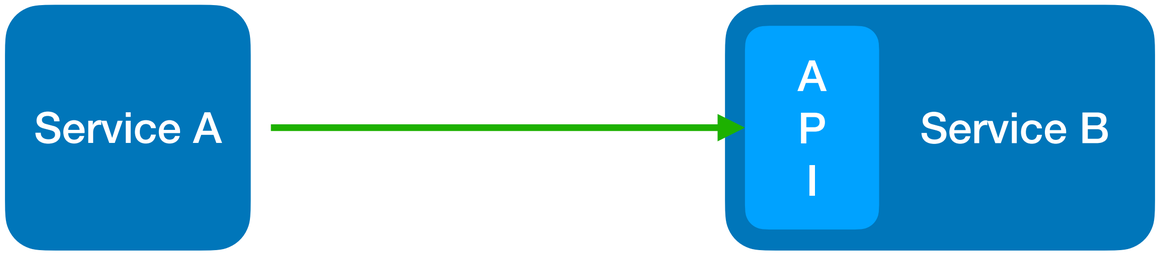
- общие понятия – синхронное и асинхронное
- эти же термины используются для описания выполнения кода
- чтобы избежать двусмысленности в рамках курса будут использоваться другие термины
- явное – синхронное
- реактивное – асинхронное



явное межсервисное взаимодействие

принцип

- один сервис отправляет запросы напрямую к другому
- вызывающий сервис **явно** зависит от отвечающего



явное межсервисное взаимодействие

REST

- **RE**presentational **S**tate **T**ransfer
- конвенция по проектированию HTTP API
- пути должны именоваться на основе **ресурсов** системы
- действия должны отражаться HTTP методами

явное межсервисное взаимодействие

REST

- **`GET`** – получение данных
- **`POST`** – создание ресурса
- **`PUT`** – частичное обновление ресурса
- **`DELETE`** – удаление ресурса

явное межсервисное взаимодействие

REST

```
GET /users
GET /users/{user_id}
GET /users/{user_id}/photos
POST /users/{user_id}/photos
PUT /users/{user_id}/photos/{photo_id}/caption
DELETE /users/{user_id}/photos/{photo_id}
```

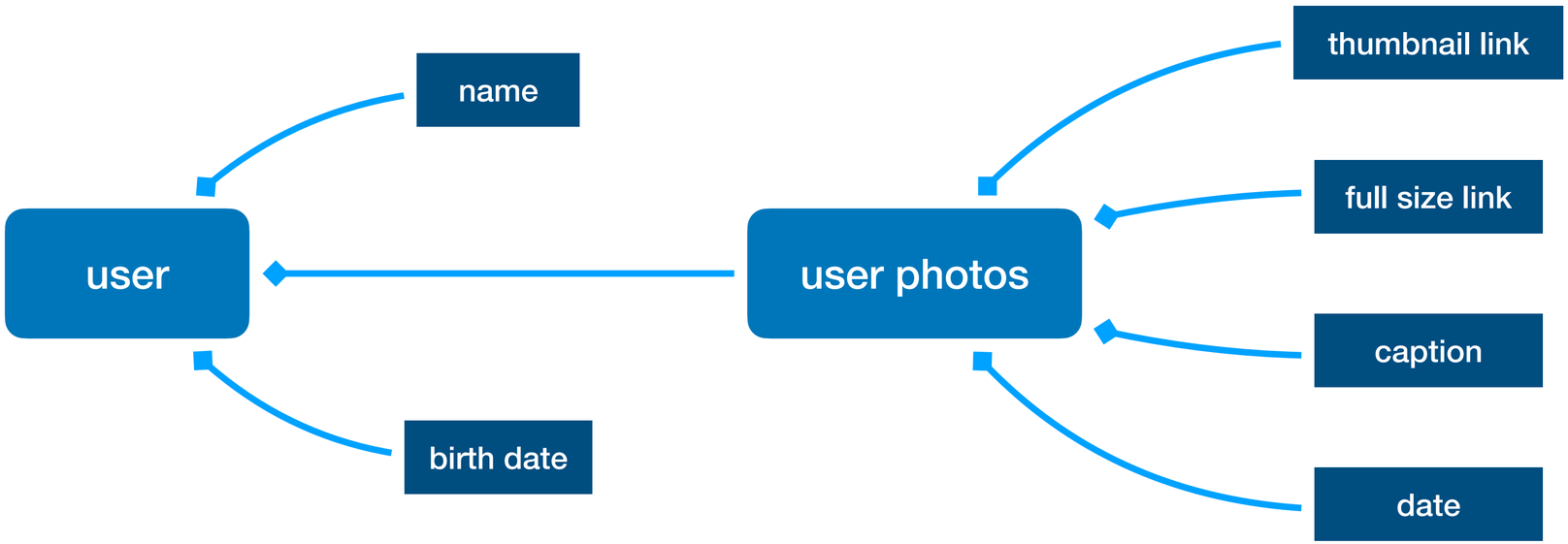
явное межсервисное взаимодействие

GraphQL

- Overfetching – когда мы получаем из запроса больше данных, чем нам нужно
- Underfetching – когда мы получаем из запроса меньше данных, чем нам нужно, соответственно нужно делать несколько запросов

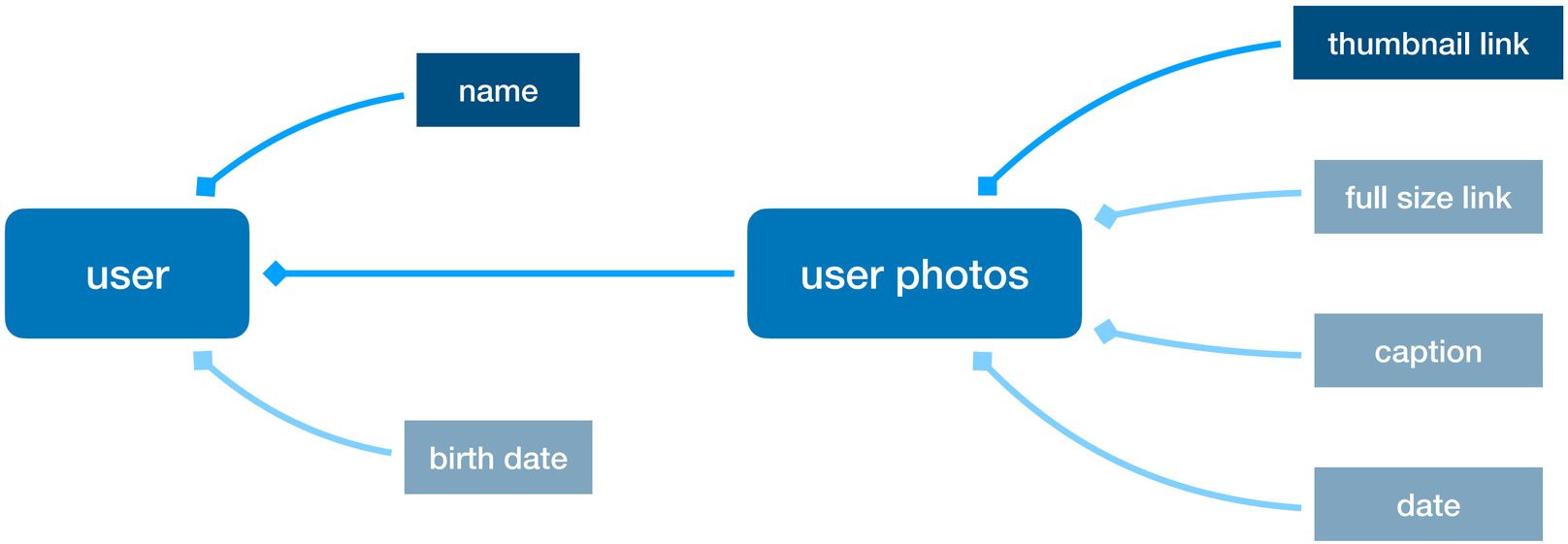
явное межсервисное взаимодействие

GraphQL



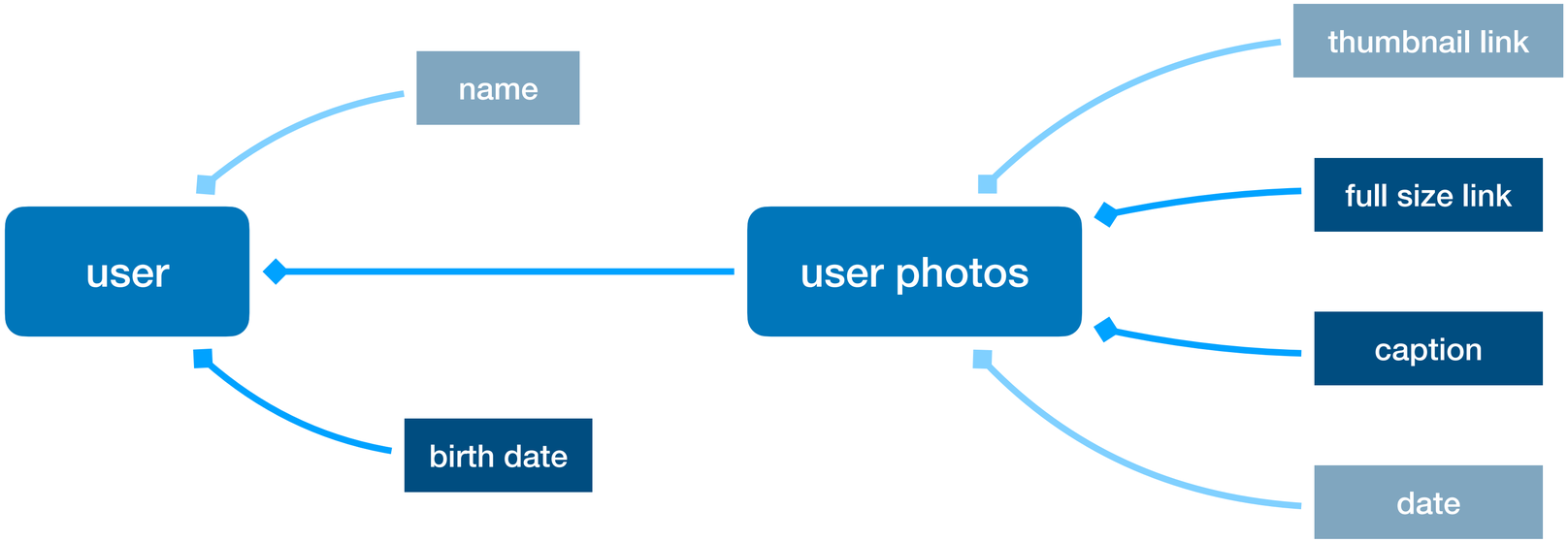
явное межсервисное взаимодействие

GraphQL



явное межсервисное взаимодействие

GraphQL



явное межсервисное взаимодействие

RPC

- **R**emote **P**rocedure **C**alling
- сервис определяет своё API в виде набора методов
- интеграция выглядит будто вы просто вызываете метод
- `JSON-RPC`, `SignalR`, `gRPC`

gRPC

gRPC

proto

- для определения контрактов gRPC использует proto
- protobuf – платформа-независимый язык для определения структуры Protocol Buffers
- Protocol Buffers – протокол бинарной сериализации и стуктуризации данных

```
syntax = “proto3”;
package sample.api.contracts;

option csharp_namespace = “Sample.Api.Contracts”;

...
```

protobuf

сообщения

- определяют типы контрактов
- объявляются ключевым словом `message`
- каждый атрибут сообщения должен иметь уникальный индекс
- в сериализованном виде хранятся в виде словаря
- proto предоставляет множество стандартных примитивов – `int32`, `int64`, `float`, `double`, `string` ...
- атрибуты-коллекции помечаются модификатором `repeated`

protobuf

сообщения

```
message User {
  int64 user_id = 1;
  string user_name = 2;
  repeated int64 user_favourite_numbers = 3;
}
```

protobuf

сообщения

- файлы с сообщениями могут иметь зависимости
- другие файлы с сообщениями подключаются с использованием ключевого слова `import`
- после этого в файле будут доступны все типы из подключённого
- путь до proto файла указывается от ProtoRoot

```
import "models/user.proto";
```

protobuf

нуллабельность

- protobuf синтаксис не имеет встроенных nullable аннотаций
- по умолчанию null значения поддерживаются для объектов сообщений, так как сериализатор не будет писать ключи для null значений
- используйте модификатор optional для пометки значений где возможен null
- для нуллабельных примитивов есть сообщения-обёртки из `google/protobuf/wrappers`
  - `google.protobuf.Int32Value`
  - `google.protobuf.StringValue`
  - `...`
- `repeated` атрибуты не могут иметь null значений

protobuf

енамы

- объявляются с помощью ключевого слова `enum`
- каждый кейс должен иметь индекс
- имеют особые конвенции нейминга
  - кейсы енама должны определяться заглавными буквами
  - кейсы енама должны содержать его название как префикс
- енамы всегда должны иметь кейс с индексом `0`, именуемый `UNSPECIFIED`

protobuf

енамы

```
enum UserState {
  USER_STATE_UNSPECIFIED = 0;
  USER_STATE_ACTIVE = 1;
  USER_STATE_DELETED = 2;
}
```

protobuf

сервисы

- конструкция service определяет своего рода контроллер в gRPC
- сервис состоит из набора операций, определяемых ключевым словом `rpc`
- каждая операция имеет request и response
- всегда стоит выделять отдельные сообщения для request и response

protobuf

сервисы

```
service UserService {
  rpc CreateUser(CreateUserRequest) returns (CreateUserResponse);
}
```

gRPC

виды запросов

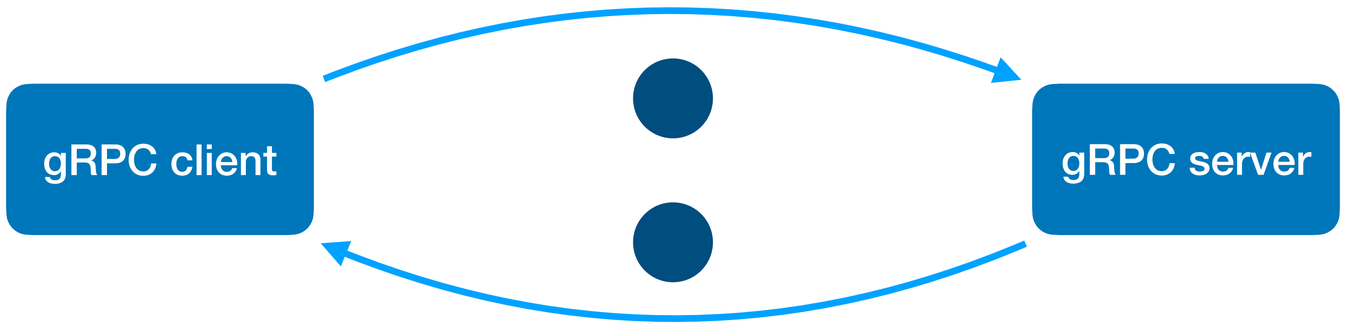
- gRPC предоставляет функционал потоковой обработки сообщений
- поток сообщений может быть как в request, так и в response
- операции в gRPC делятся на 4 типа

gRPC

UnaryRequest

- одно сообщение в request
- одно сообщение в response

```
rpc CreateUser(CreateUserRequest) returns (CreateUserResponse);
```

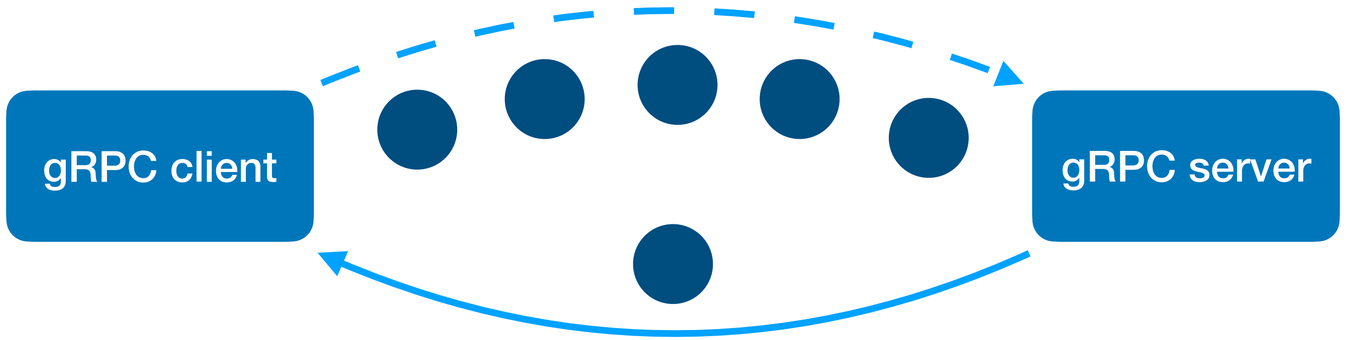


gRPC

ClientStream

- поток сообщений в request
- одно сообщение в response

```
rpc CreateUsers(stream CreateUserRequest) returns (CreateUsersResponse);
```

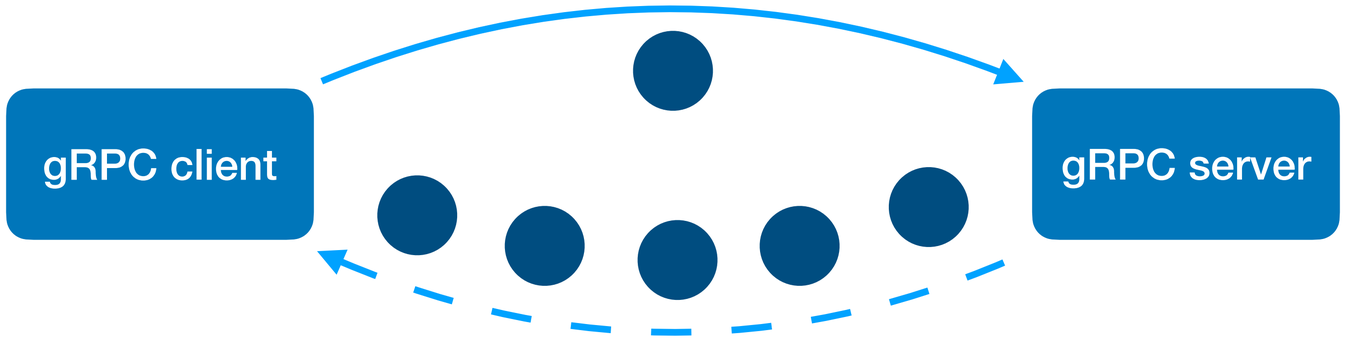


gRPC

ServerStream

- одно сообщение в request
- поток сообщений в response

```
rpc GetUserEvents(GetUserEventsRequest) returns (stream GetUserEventResponse);
```

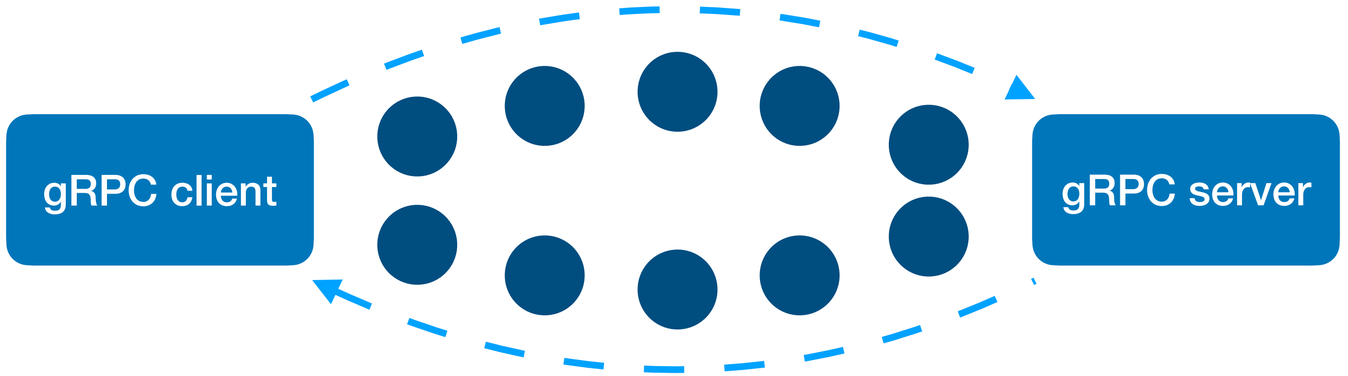


gRPC

DuplexStream

- поток сообщений в request
- поток сообщений в response

```
rpc MonitorEvents(stream EventSubscribeRequest) returns (stream EventOccuredResponse);
```



gRPC и .NET

gRPC и .NET

генерация C# кода по proto

- для генерации используется библиотека Grpc.Tools
- для генерации нужно настроить csproj
- такая конфигурация сгенерирует типы по сообщениям и енамам

```
<ItemGroup>
    <Protobuf ProtoRoot="protos" Include="protos\**\*.proto"/>
</ItemGroup>
```

gRPC и .NET

генерация C# кода по proto

- для генерации кода по сервисам, необходимо задать атрибут GrpcServices
  - None
  - Client
  - Server
  - Both

```
<ItemGroup>
    <Protobuf ProtoRoot="protos" Include="protos\**\*.proto" GrpcServices="Server"/>
</ItemGroup>
```

gRPC и .NET

реализация сервисов

```
service UserService {
  rpc CreateUser(CreateUserRequest) returns (CreateUserResponse);
}
```

```
public class UserController : UserService.UserServiceBase
{
    public override Task<CreateUserResponse> CreateUser(
        CreateUserRequest request,
        ServerCallContext context)
    {
        // create user logic
    }
}
```

gRPC и .NET

реализация сервисов

```
builder.Services.AddGrpc(); // Grpc.AspNetCore
builder.Services.AddGrpcReflection(); // Grpc.AspNetCore.Server.Reflection
```

```
app.UseEndpoints(e ⇒
{
    e.MapGrpcService<UserController>();
    e.MapGrpcReflectionService();
});
```

gRPC и .NET

HTTP2 эндпоинты

```
"Kestrel": {
  "EndpointDefaults": {
    "Protocols": "Http2"
  }
}
```

```
"Kestrel": {
  "Endpoints": {
    "gRPC": {
      "Url": "http://*:8020",
      "Protocols": "Http2"
    },
    "Http": {
      "Url": "http://*:8022",
      "Protocols": "Http1"
    }
  }
}
```

gRPC и .NET

серверные интерцепторы

```
public class SampleServerInterceptor : Interceptor
{
    public override Task<TResponse> UnaryServerHandler<TRequest, TResponse>(...);

    public override Task<TResponse> ClientStreamingServerHandler<TRequest, TResponse>(...);

    public override Task ServerStreamingServerHandler<TRequest, TResponse>(...);

    public override Task DuplexStreamingServerHandler<TRequest, TResponse>(...);
}
```

gRPC и .NET

серверные интерцепторы

```
builder.Services.AddGrpc(grpc ⇒ grpc.Interceptors.Add<SampleServerInterceptor>());
```

gRPC и .NET

клиенты

- генерируются при `GrpcServices = Client | Both`
- добавляются в DI контейнер
- имеют синхронные и асинхронные перегрузки под каждую операцию

```
builder.Services.AddGrpcClient<UserService.UserServiceClient>((sp, o) ⇒
{
    var options = sp.GetRequiredService<IOptions<UserServiceOptions>>();
    o.Address = options.Value.Address;
});
```

gRPC и .NET

клиентские интерцепторы

```
public class SampleClientInterceptor : Interceptor
{
    public override TResponse BlockingUnaryCall<TRequest, TResponse>(...);

    public override AsyncUnaryCall<TResponse> AsyncUnaryCall<TRequest, TResponse>(...);

    public override AsyncServerStreamingCall<TResponse> AsyncServerStreamingCall<TRequest, TResponse>(...);

    public override AsyncClientStreamingCall<TRequest, TResponse> AsyncClientStreamingCall<TRequest, TResponse>(...);

    public override AsyncDuplexStreamingCall<TRequest, TResponse> AsyncDuplexStreamingCall<TRequest, TResponse>(...);
}
```

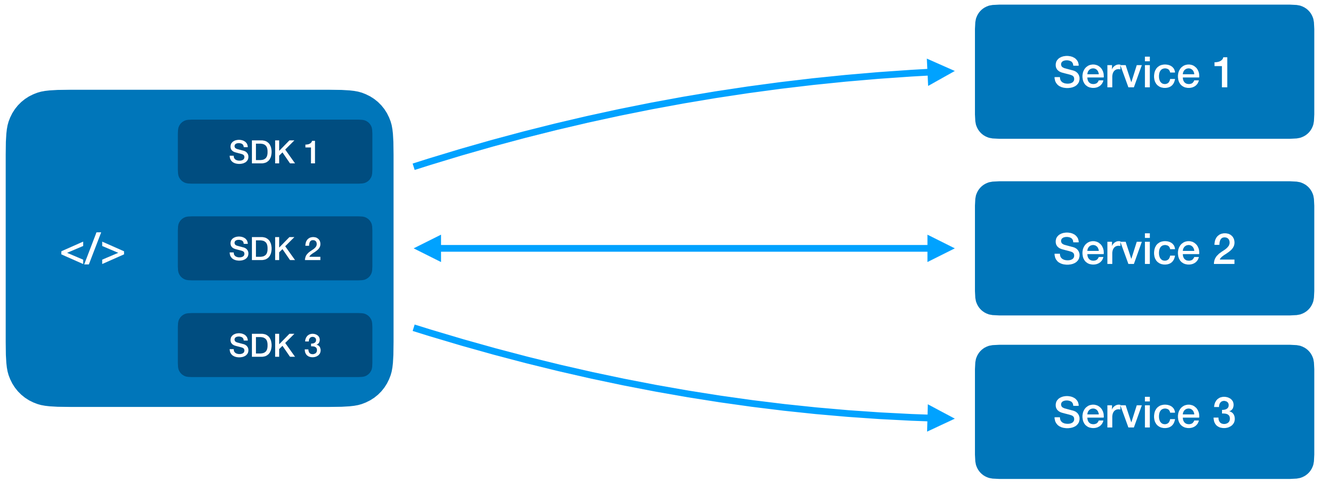
gRPC и .NET

клиентские интерцепторы

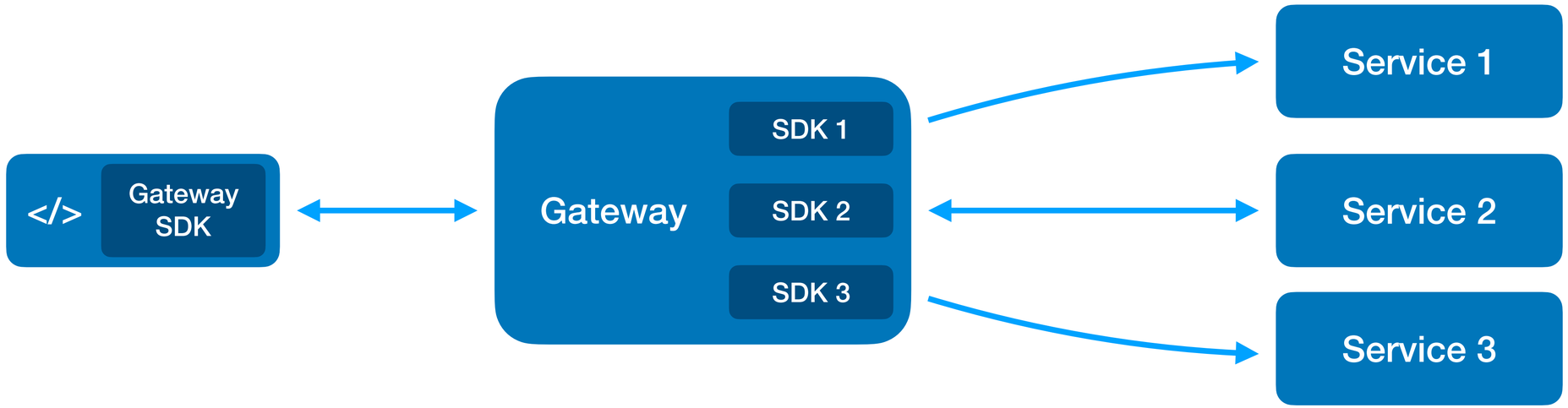
```
builder.Services
    .AddGrpcClient<UserService.UserServiceClient>()
    .AddInterceptor<SampleClientInterceptor>();
```

гейтвей

гейтвей



гейтвей


Gateway
SDK
Gateway
SDK 1
SDK 2
SDK 3
Service 1
Service 2
Service 3