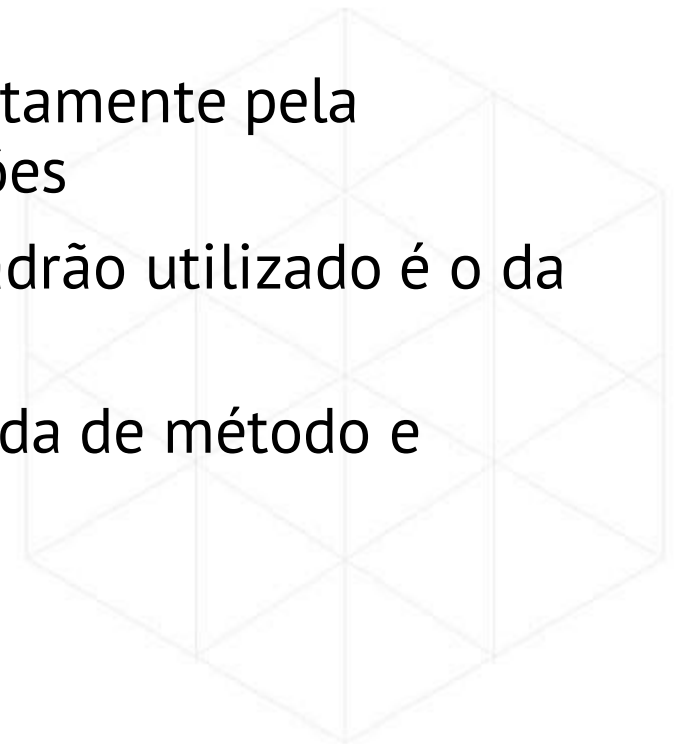




# *Comunicação entre serviços*

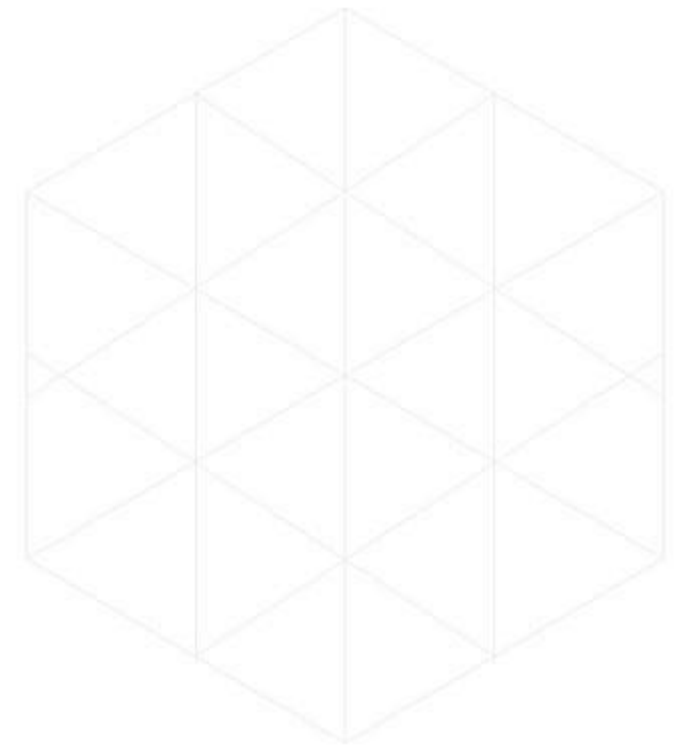
# *Comunicação em aplicações*

- Em uma aplicação tradicional, a comunicação é feita diretamente pela linguagem de programação através de chamada de funções
- Não faz diferença qual módulo está sendo chamado, o padrão utilizado é o da própria linguagem de programação
- Em POO, a função pública dentro de uma classe é chamada de método e serve para mudar o estado deste objeto



# Comunicação em aplicações

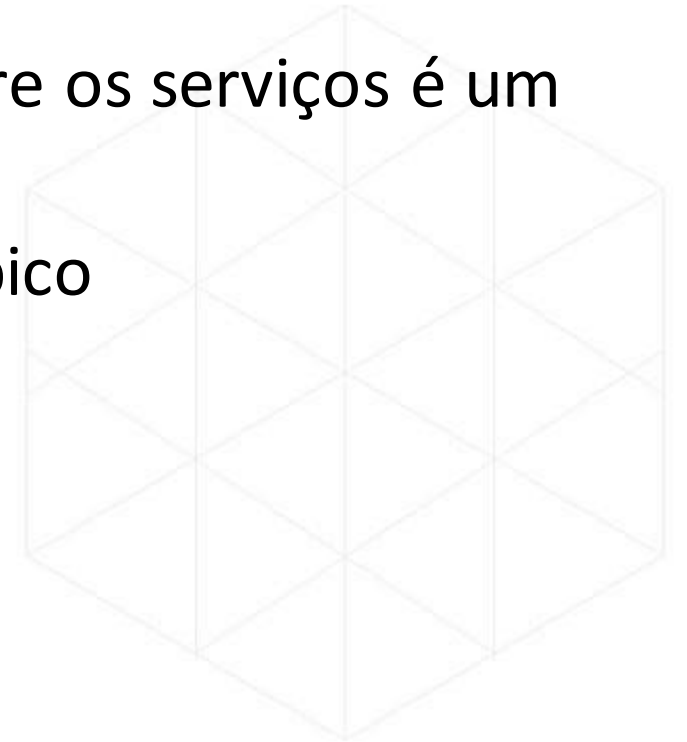
```
1 public class Conta {  
2  
3     private Double saldo;  
4  
5     public void setSaldo(Double saldo) {  
6         this.saldo = saldo;  
7     }  
8  
9     public Double getSaldo() {  
10        return saldo;  
11    }  
12  
13    public void depositar(Double valor){  
14        saldo += valor;  
15    }  
16  
17    public void verificaSaldo(){  
18        System.out.println("Valor do Saldo: "+getSaldo());  
19    }  
20 }
```






# *Comunicação entre aplicações*

- Em uma arquitetura distribuída a comunicação entre os serviços é um tópico importante a ser estudado.
- Não existe uma solução *bala de prata* para este tópico
- A conversa vai muito além do REST...

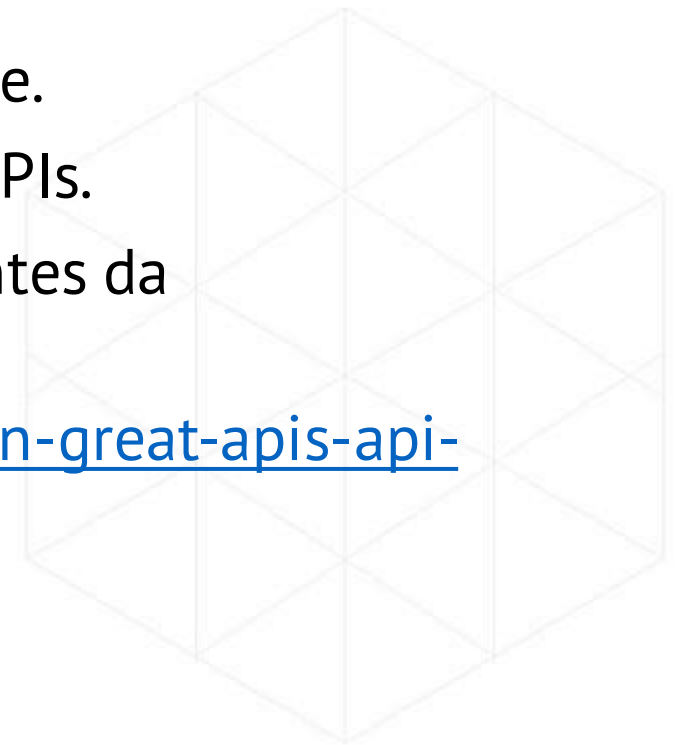




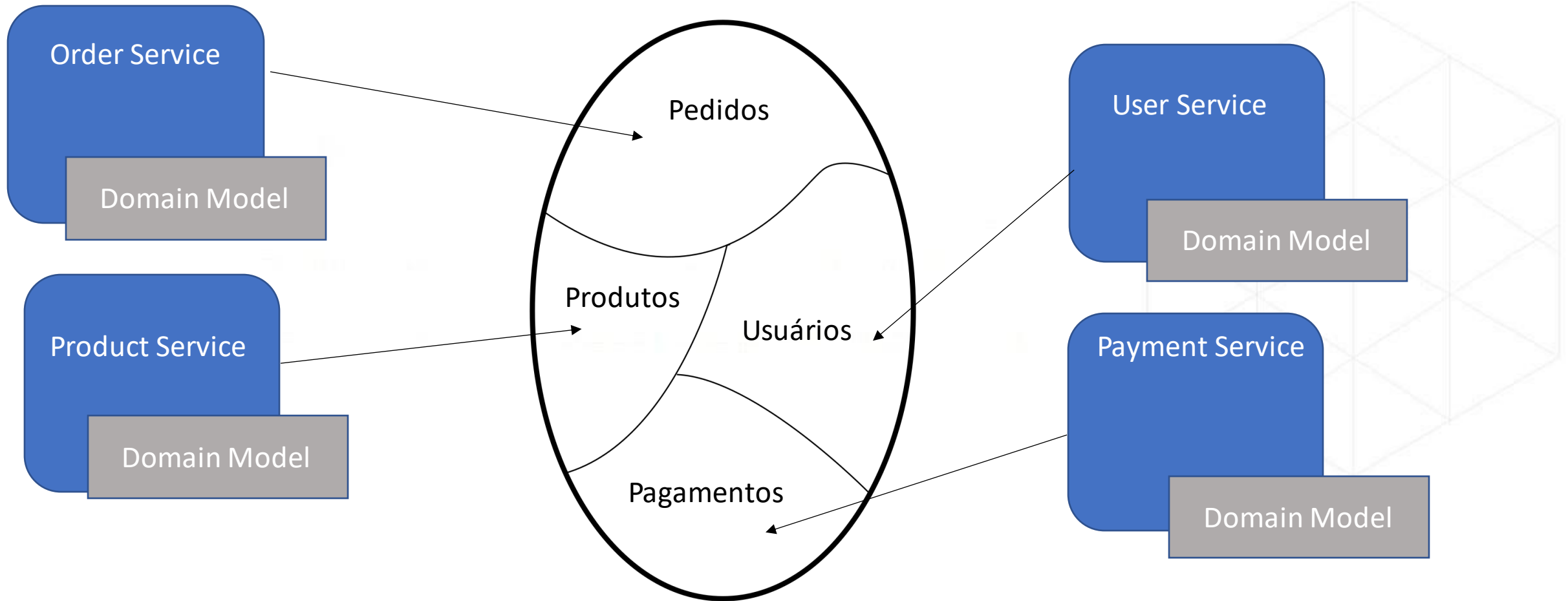
*Uma pequena discussão sobre comunicação  
em uma arquitetura de microsserviços*

# ***API First***

- Interfaces são essenciais em qualquer projeto de software.
- TODA a comunicação com um serviço deve ser feita via APIs.
- É uma boa estratégia fazer o desenho da API primeiro, antes da implementação do código.
- <https://www.programmableweb.com/news/how-to-design-great-apis-api-first-design-and-raml/how-to/2015/07/10>



# *Comunicação em microserviços*





# Comunicação em microserviços

Serviço	Operação	Colaboradores
User Service	<ul style="list-style-type: none"><li>CreateUser();</li><li>UpdateUser();</li><li>Login();</li><li>Logout();</li><li>ValidateUser();</li></ul>	-
OrderService	CreateOrder()	<ul style="list-style-type: none"><li>ProductService.getProduct();</li><li>ProductService.processProduct();</li><li>UserService.validateUser();</li><li>PaymentService.ReceivePayment()</li></ul>
OrderService	ChangeStatus()	ProductService.processProduct()



# Definindo as APIs

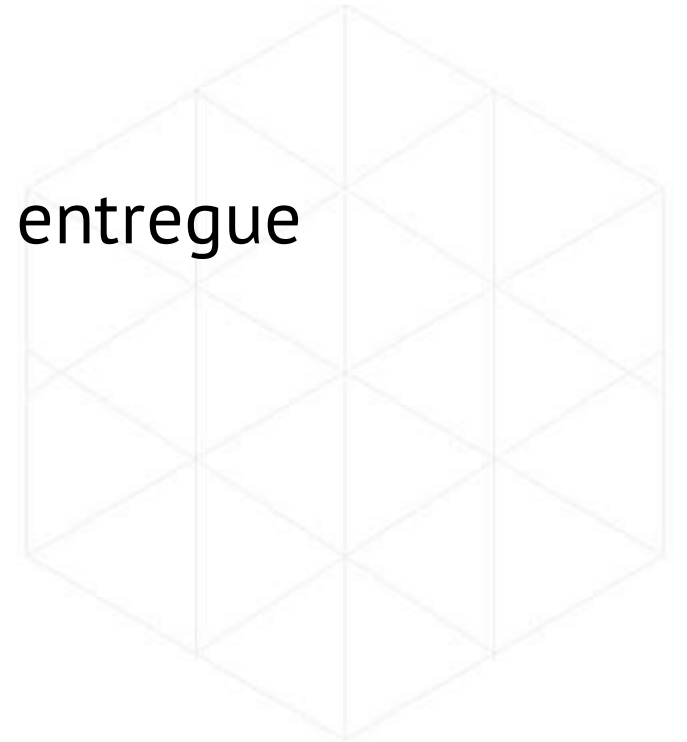
Serviço	Operação	Colaboradores
PaymentService	ReceivePayment()	-
PaymentService	AuthorizePayment()	OrderService.ChangeStatus()
PaymentService	DenyPayment()	OrderService.ChangeStatus()
PaymentService	CancelPayment	OrderService.ChangeStatus()
ProductService	ListProducts()	
ProductService	GetProduct()	
ProductService	ProcessProducts()	

# Serviços e APIs

Serviço	Operação	Colaboradores
PaymentService	ReceivePayment()	-
PaymentService	AuthorizePayment()	OrderService.ChangeStatus()
PaymentService	DenyPayment()	OrderService.ChangeStatus()
PaymentService	CancelPayment	OrderService.ChangeStatus()
ProductService	ListProducts()	
ProductService	GetProduct()	
ProductService	ProcessProducts()	

# ***Comunicação Síncrona***

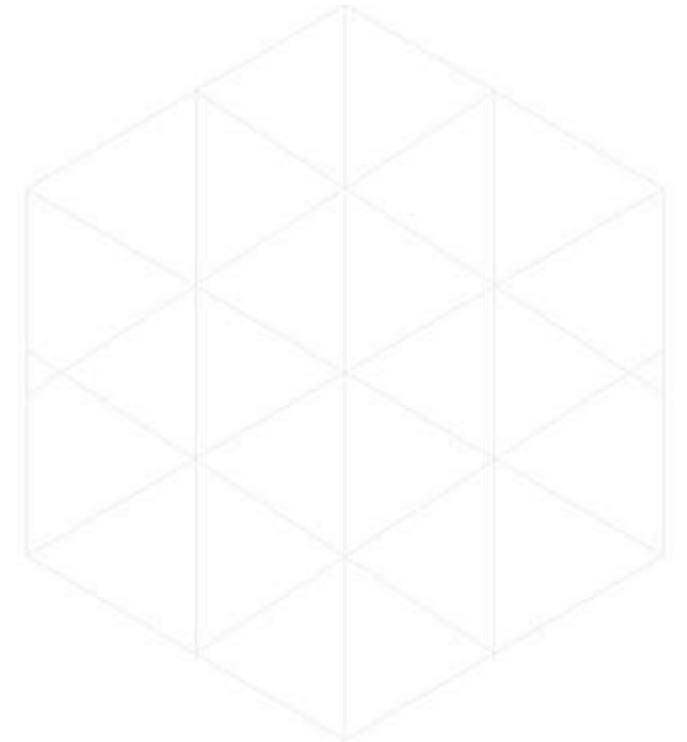
- Contato imediato entre emissor e receptor
- Pode bloquear o emissor até que a resposta seja entregue
- Trabalha no formato Requisição -> Resposta
- Um emissor / Um receptor





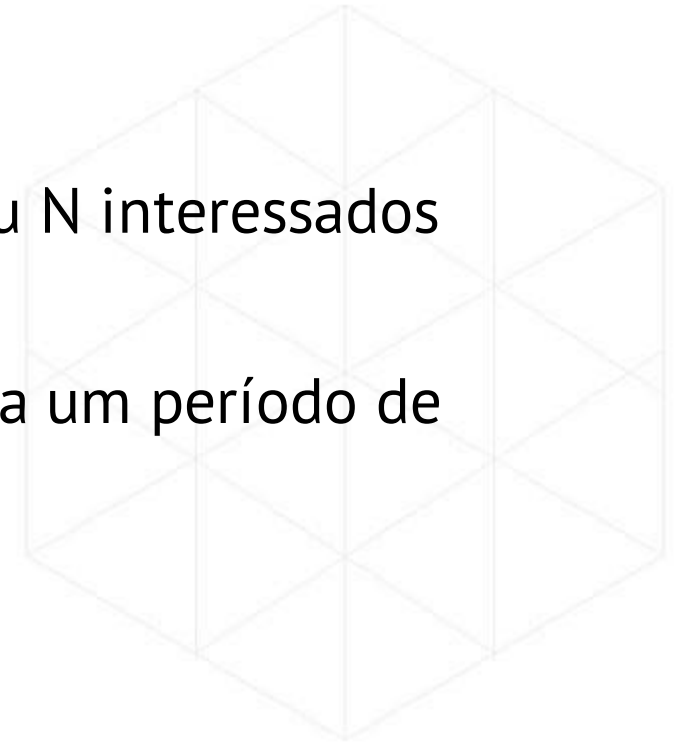
# ***Comunicação Assíncrona***

- Não bloqueia nenhuma das partes
- Não é esperada uma resposta imediata
- one-to-one
- one-to-many



# ***One-to-many***

- Publish/Subscribe
  - Mensagem publicada, pode ser consumida por zero ou N interessados
- Publish/async response
  - Uma mensagem é publicada e quem publicou aguarda um período de tempo até que as respostas cheguem



# *Comunicação em microserviços*

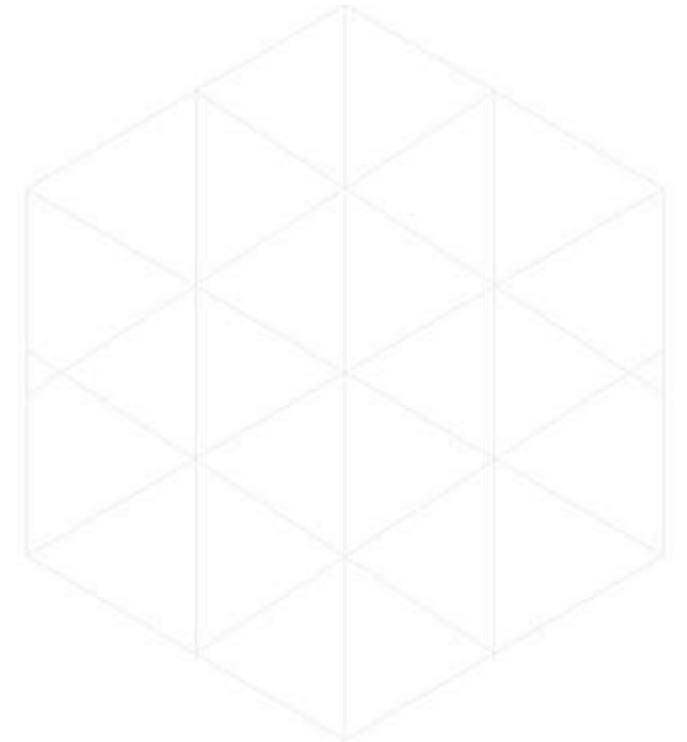
- Importante nesta etapa manter o desenho agnóstico

	One-to-one	One-to-many
Síncrona	Request/Response	-
Assíncrona	Request/Response Assíncrona	Publish/subscribe Publish/async responses



# *Comunicação em microserviços*

- Mecanismos síncronos
  - REST
  - GRPC
  - ...
- Assíncronos
  - Kafka
  - AMPQ
  - STOMP



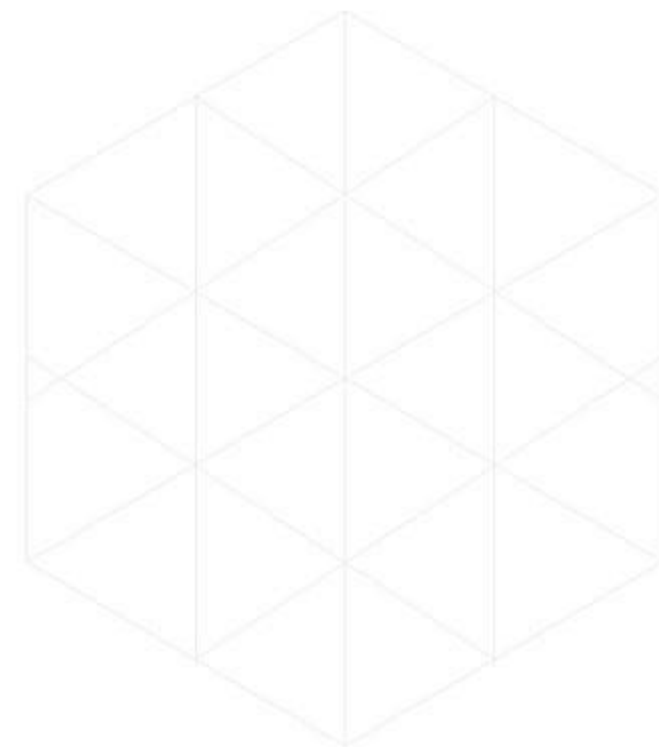
# ***Formatos de mensagens***

- Uma arquitetura de *microserviços* é um ambiente heterogêneo.
- Portanto, a comunicação entre eles deve ser ***agnóstica***.
- O formato deve ser idealmente **independente de tecnologia**.
- Dito isso, temos duas opções de formatos: **texto** e **binário**.



# ***Formatos de Texto***

- Amplamente utilizado através de **JSON** ou **XML**.
- A principal vantagem desse modelo é ser *auto descritivo*.
- Também é inteligível pra pessoas.
- Pode ser alterado sem nenhuma *breaking change*.





# ***Formatos de Texto***

- Pode ser extremamente verboso, especialmente XML.
- Caso seja um objeto complexo, o parse dele na linguagem de programação pode levar um precioso tempo.



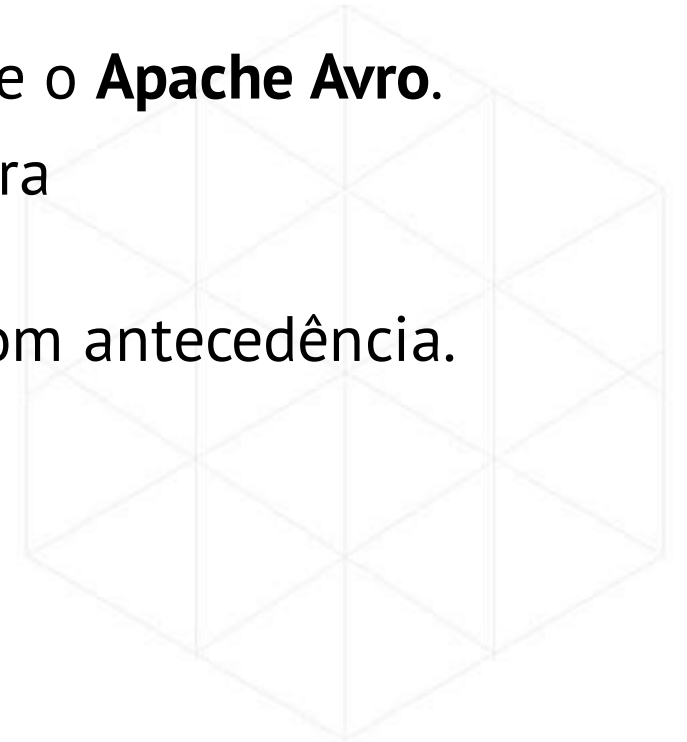
# Formatos de Texto

```
"identification":{  
  "type":"AGENCIA",  
  "code":"0001",  
  "checkDigit":"9",  
  "name":"Marília",  
  "relatedBranch":"0001",  
  "openingDate":"2010-01-02"  
},  
"postalAddress":{  
  "address":"Av Naburo Ykesaki, 1270",  
  "additionalInfo":"Loja B",  
  "districtName":"Centro",  
  "townName":"Marília",  
  "ibgeCode":"3515890",  
  "countrySubDivision":"SP",  
  "postCode":"17500001",  
  "country":"Brasil",  
  "countryCode":"BRA",  
  "geograficCoordinates":{  
    "latitude":"-90.009876543",  
    "longitude":"-180.00986543"  
  }  
},
```

- 503 caracteres
- 376 caracteres de definições
- 127 caracteres de informação útil
- **25%** de informação útil

# ***Formatos Binários***

- Uns dos formatos mais populares são o **Protocol Buffers** e o **Apache Avro**.
- É necessário uma dependência instalada na aplicação para serialização/deserialização das mensagens.
- No caso do Avro, deve-se informar o *schema* dos dados com antecedência.





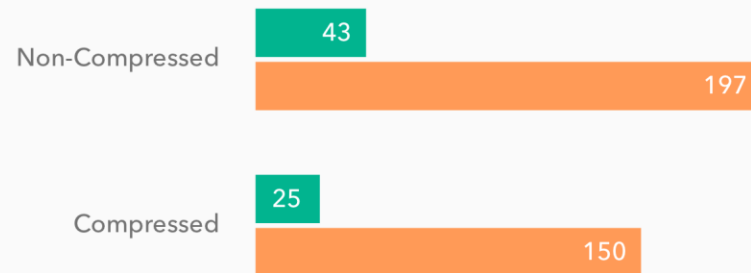
# ***Proto Buff***

- Desenvolvido pelo Google
- Prover uma maneira mais eficiente de serializar/desserializar dados
- Open source
- Possui vários tipos de dados, como enum e métodos
- <https://auth0.com/blog/beating-json-performance-with-protobuf/>



# Proto Buff

## Java to Java Communication



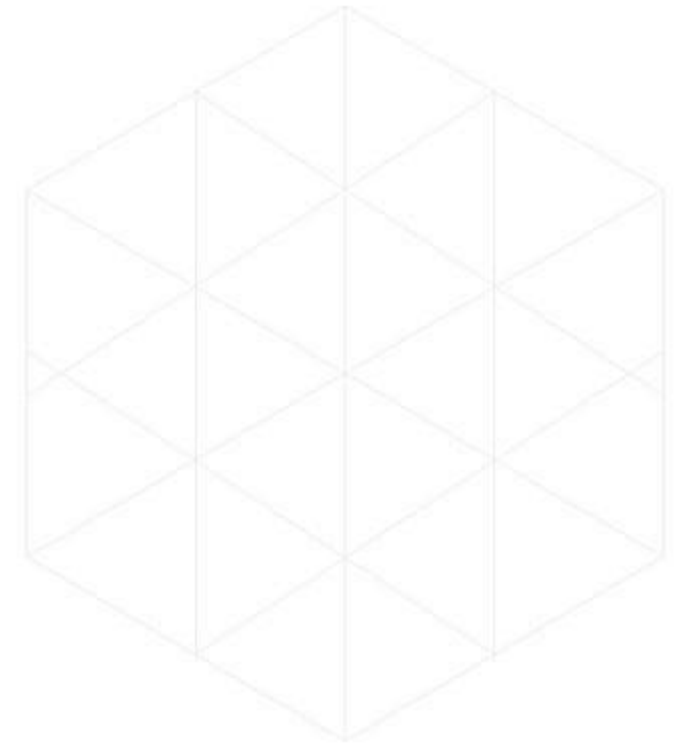
### References

- Protobuf Time (ms)
- JSON Time (ms)



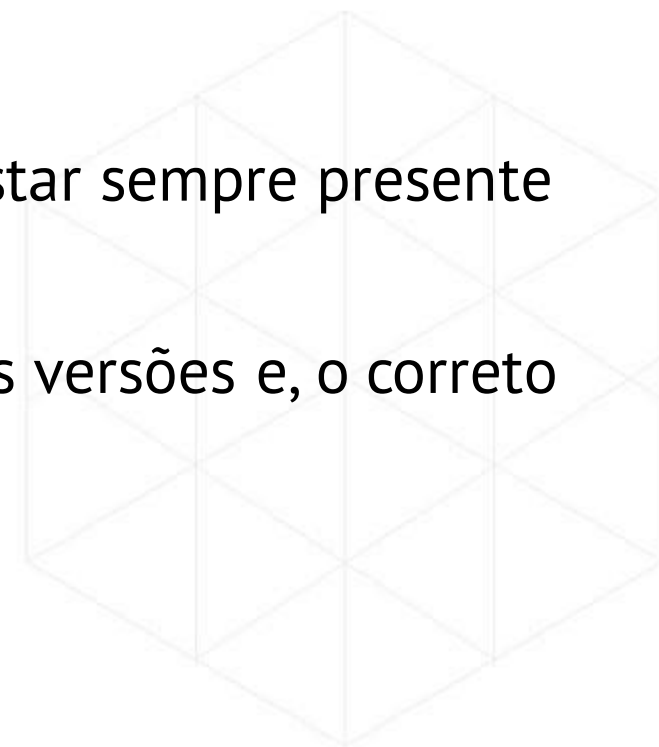
# ***Apache Avro***

- Desenvolvido pela Apache
- Estrutura de dados ricas
- Não precisa, necessariamente, de um gerador de código
- Este pode, opcionalmente, ser utilizado para otimizações



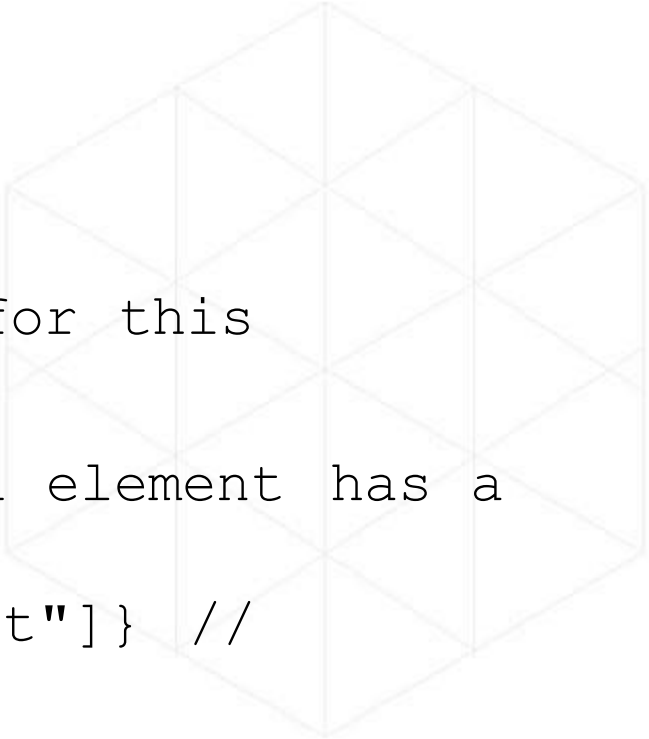
# Apache Avro

- Depende da definição de *schemas*
- No processo de leitura de informações, o schema deve estar sempre presente
- Tipagem dinâmica
- Quando um schema é atualizado, pode-se utilizar as duas versões e, o correto ser resolvido em tempo de execução



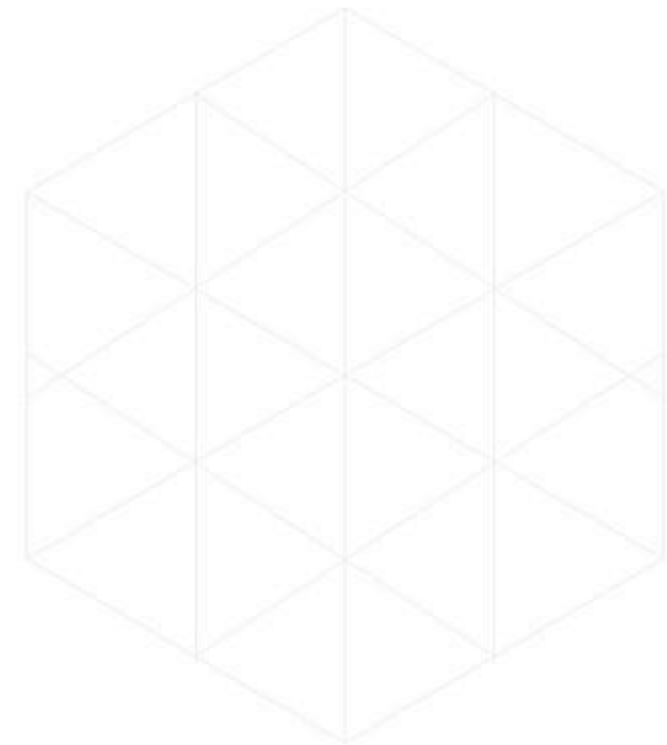
# Apache Avro

```
{  
  "type": "record",  
  "name": "LongList",  
  "aliases": ["LinkedLongs"],      // old name for this  
  "fields" : [  
    {"name": "value", "type": "long"}, // each element has a  
long  
    {"name": "next", "type": ["null", "LongList"]} //  
optional next element  
  ]  
}
```



# ***Formato da comunicação***

- Inteligível para humanos
  - JSON
  - XML
  - ...
- Binário
  - Avro
  - Buffer Protocols
  - ...







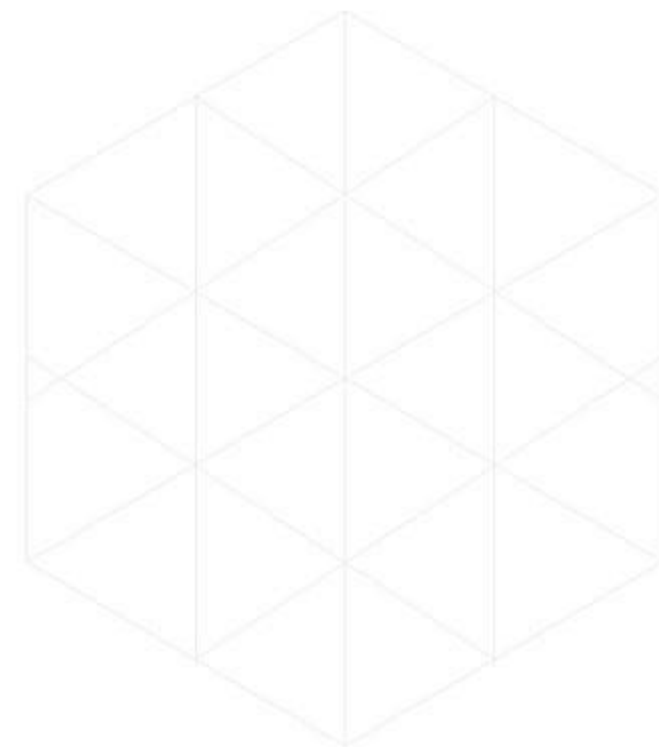
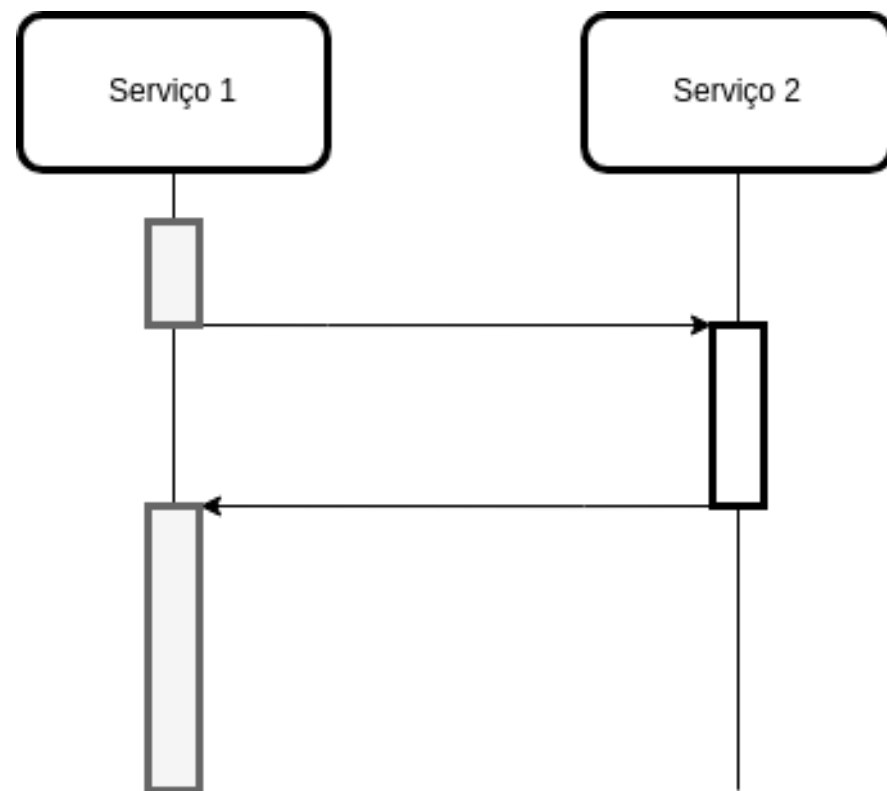
# *Comunicação síncrona*

# ***Comunicação Síncrona***

- Request/Response.
- Requisições podem ser bloqueadas até a resposta.
- Podem ser não bloqueantes, como as reativas.



# *Comunicação Síncrona*



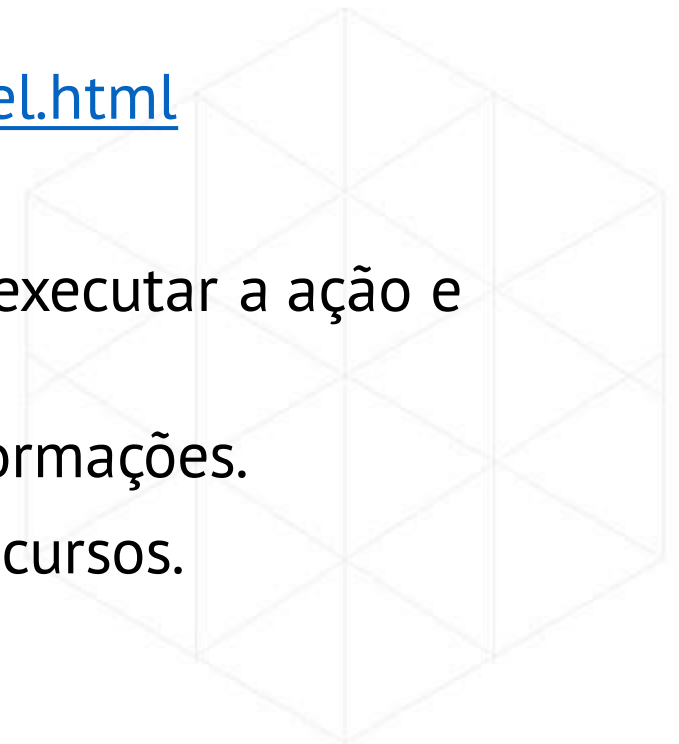
# REST

- Protocolo de comunicação que *realmente* utiliza HTTP.
- Normalmente representa somente um objeto de negócio por recurso.
- Utiliza verbos HTTP para referenciar os recursos.
- Pode ser utilizado com quaisquer formatos, não somente JSON.
- Poucas aplicações realmente implementam RESTFUL.
- <https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>



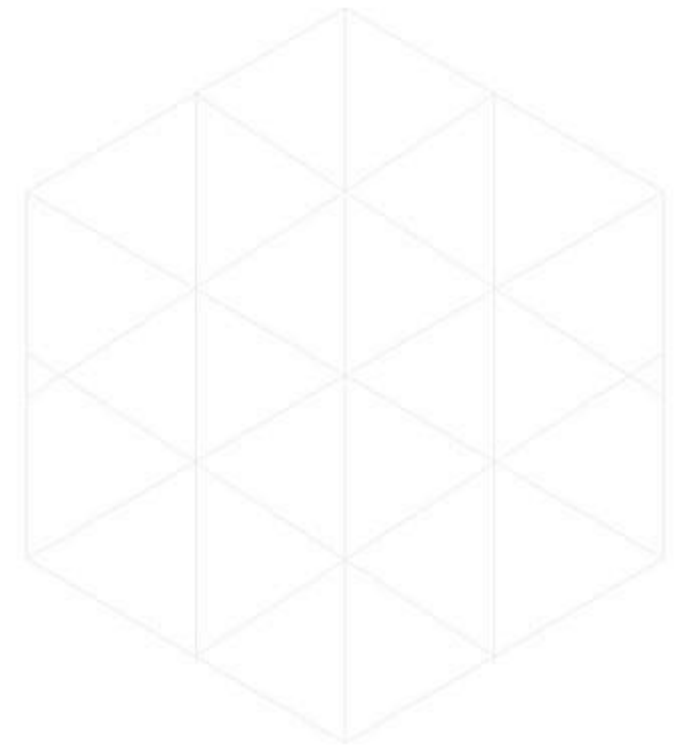
# ***Richardson Maturity Model***

- <http://martinfowler.com/articles/richardsonMaturityModel.html>
- Level 0: invoca os serviços via POST.
- Level 1: Existe a utilização de recursos. Utiliza post para executar a ação e passa vários parâmetros.
- Level 2: Verbos HTTP de acordo. Permitem Cache das informações.
- Level 3: HATEOAS. Links autocontidos para os próprios recursos.



# Richardson Maturity Model

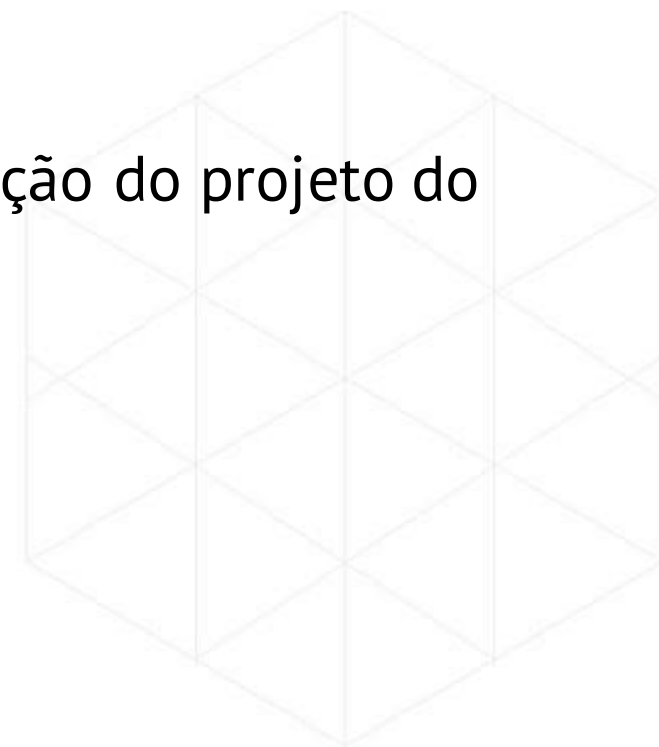
```
{
  "FirstName": "Razvan",
  "LastName": "Stetcu",
  "BirthDate": "07-08-1994",
  "Links": [
    {
      "Href": "http://localhost:62782/Employee/4",
      "Rel": "get_employee",
      "Method": "GET"
    },
    {
      "Href": "http://localhost:62782/Employee/4",
      "Rel": "delete_employee",
      "Method": "DELETE"
    },
    {
      "Href": "http://localhost:62782/Employee",
      "Rel": "edit_employee",
      "Method": "PUT"
    }
  ]
}
```





# ***Especificando uma API REST***

- Não possui linguagem de definição única.
- Uma das mais populares é o *Open API Specification*, evolução do projeto do Swagger.
- <http://www.openapis.org>



# ***Desafios no desenho de uma API REST***

- Muitos Recursos em um único Request.

```
GET /orders/order-id-1345?expand=consumer
```



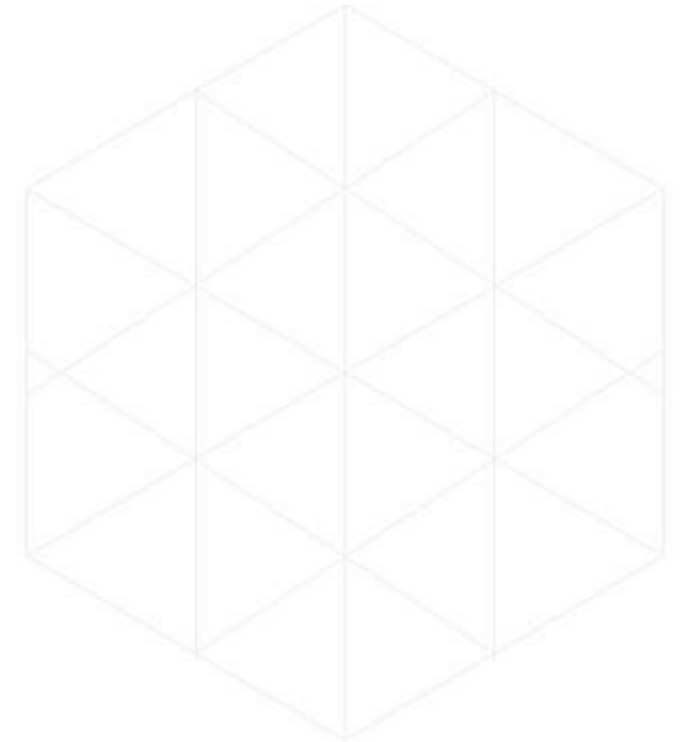
# ***Desafios no desenho de uma API REST***

- Mapear operações em verbos HTTP.

PUT /order/cancel

POST /user/activate

POST /user/deactivate



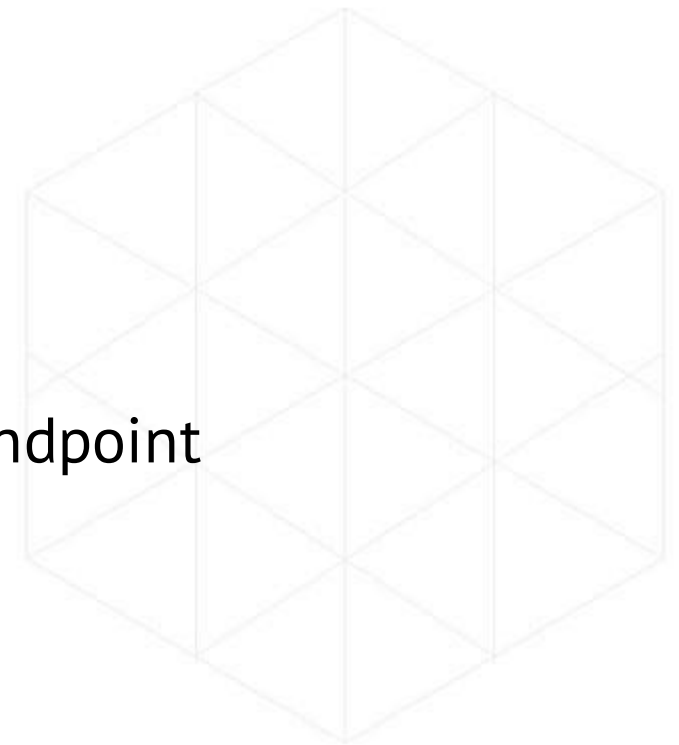
# ***Benefícios do REST***

- Simples e familiar.
- Possível de testar de *quase* qualquer lugar.
- *Firewall Friendly*
- Não requer nenhum tipo de intermediário entre a API e o Client.



# ***Algumas Desvantagens***

- Suporta *somente* o estilo **request/response**
- Disponibilidade do ecossistema pode ser prejudicada
- Deve-se saber diretamente as URLs dos serviços
- Dificuldade em mapear mais de um recurso no mesmo endpoint
- Verbos HTTP não permitem muita flexibilidade



# ***RPC***

- *Remote Procedure Call*
- Maneira *genérica* de executar métodos em aplicações remotas sem especificar os detalhes de comunicação
- Focado em *Funcionalidades*
- Não necessariamente utiliza HTTP





# ***RPC vs REST***

```
POST /enviarMensagem HTTP/1.1
Host: api.example.com
Content-Type:
application/json

{"userId": 1, "msg": "Olá!"}
```

```
POST /users/1/mensagens
HTTP/1.1
Host: api.example.com
Content-Type:
application/json

{"msg": "Olá!"}
```

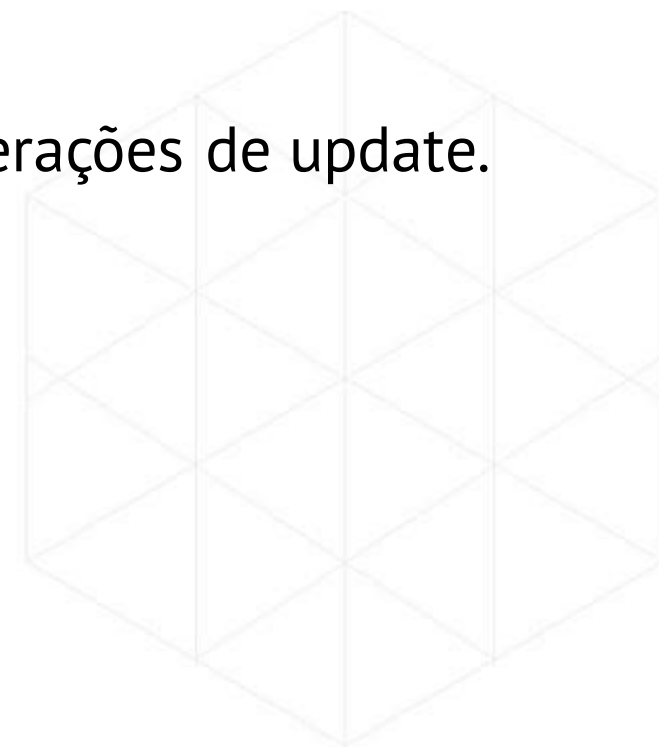
# *gRPC*

- [www.grpc.io](http://www.grpc.io)
- Protocolo baseado em mensagens binárias através de *stubs* e *skeletons*.
- HTTP/2.
- Muito eficiente, especialmente em objetos grandes.
- Também pode ser utilizado com Streaming de dados.



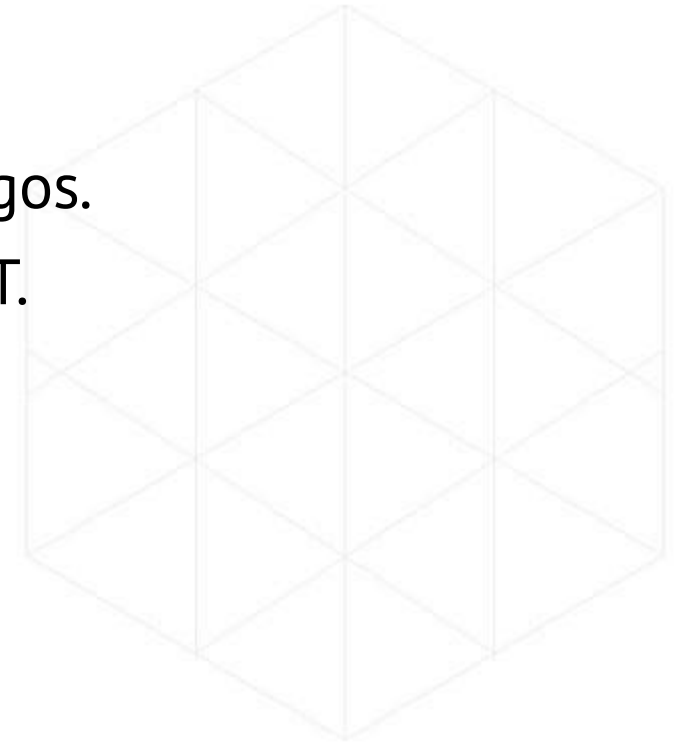
# ***Benefícios gRPC***

- Concebido para uma API com um grande conjunto de operações de update.
- Multilinguagem.
- Streaming bidirecional.
- Eficiente com objetos grandes.

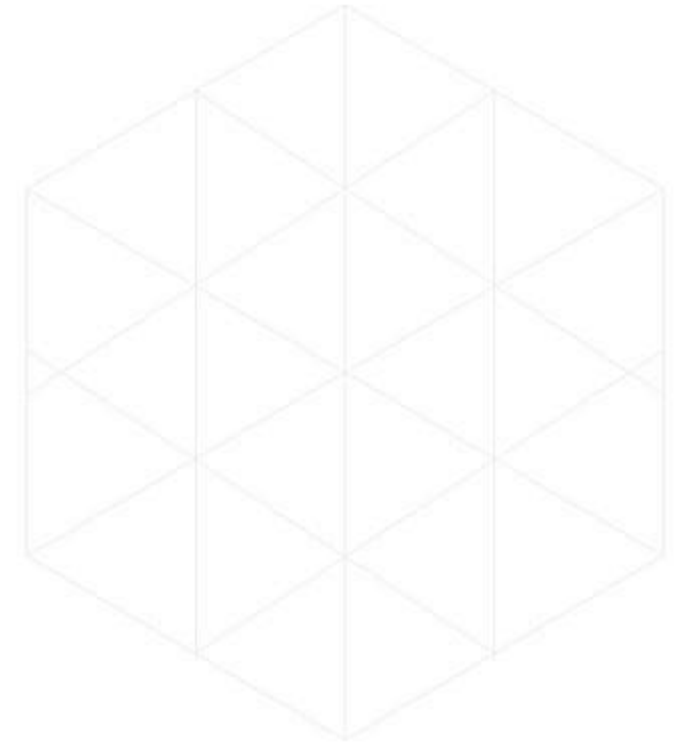
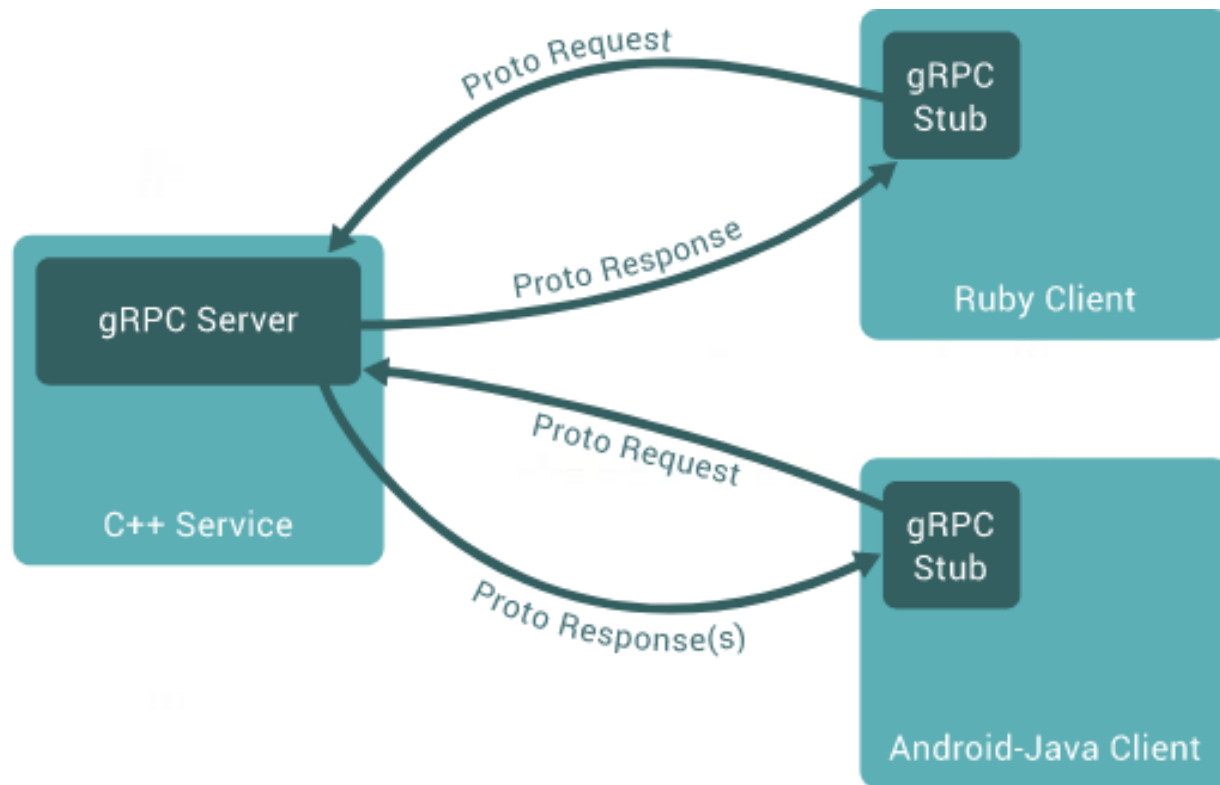


# ***Desvantagens gRPC***

- Curva de aprendizado maior que REST/JSON.
- HTTP2 pode não ser suportado pelos firewalls mais antigos.
- Sofre do mesmo problema de disponibilidade que o REST.
- <https://grpc.io/docs/languages/java/basics/>



# *gRPC*

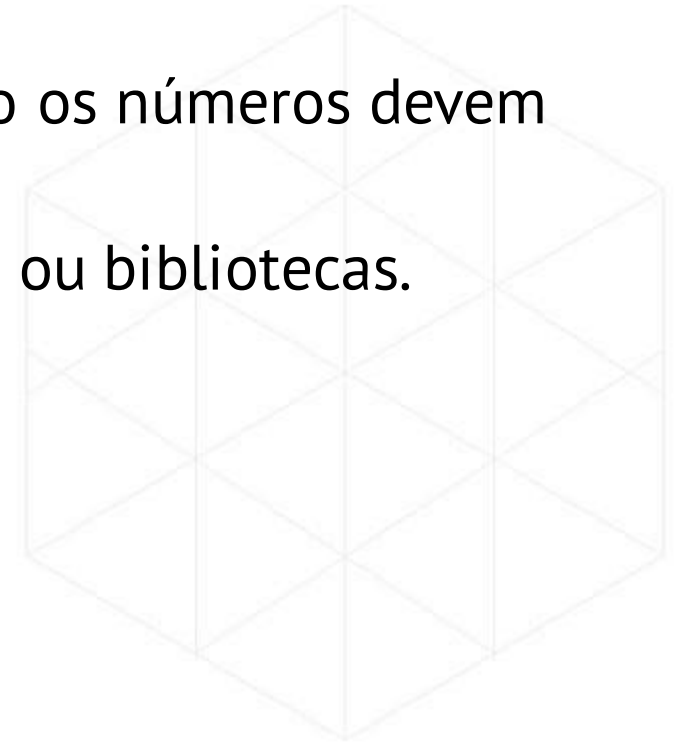


# ***Evoluindo as APIs***

- Novas features/mudanças de funcionalidades que não fazem mais sentido.
- Em *microserviços*, o custo aumenta quanto mais heterogêneo sejam os times de desenvolvimento.
- Se a API for exposta pra fora da organização, esse custo aumenta mais ainda.
- É comum uma API ter duas versões ao mesmo tempo em produção.

# ***Semantic Versioning***

- Guia para versionamento de APIs. Regras que dizem como os números devem ser utilizados para incrementar.
- Inicialmente foi utilizado para versionamento de pacotes ou bibliotecas.
- Major.minor.patch
  - Major: breaking changes
  - Minor: features, sem breaking changes
  - Patches: bugfix sem breaking changes





# *Semantic Versioning*

- Em uma API Rest, o major pode ficar na própria URL
- **http://localhost:8080/api/v1/version**
- Ou em um header http

```
GET /orders/xyz HTTP/1.1  
Accept: application/vnd.example.resource+json; version=1  
...
```

- Em mensageria, pode-se colocar em um parâmetro da mensagem publicada

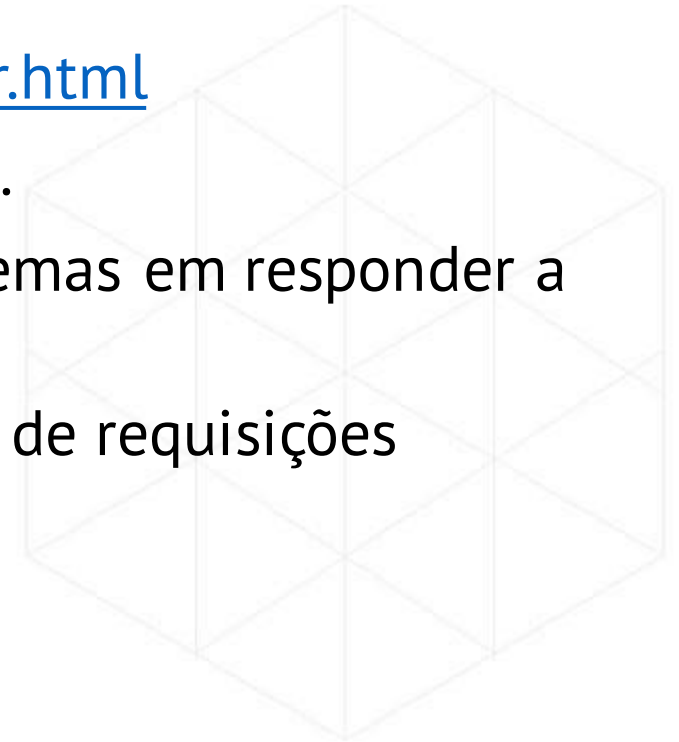
```
{ Version: 1, LastName: "Doe", Name: "John", Age: 0 }
```



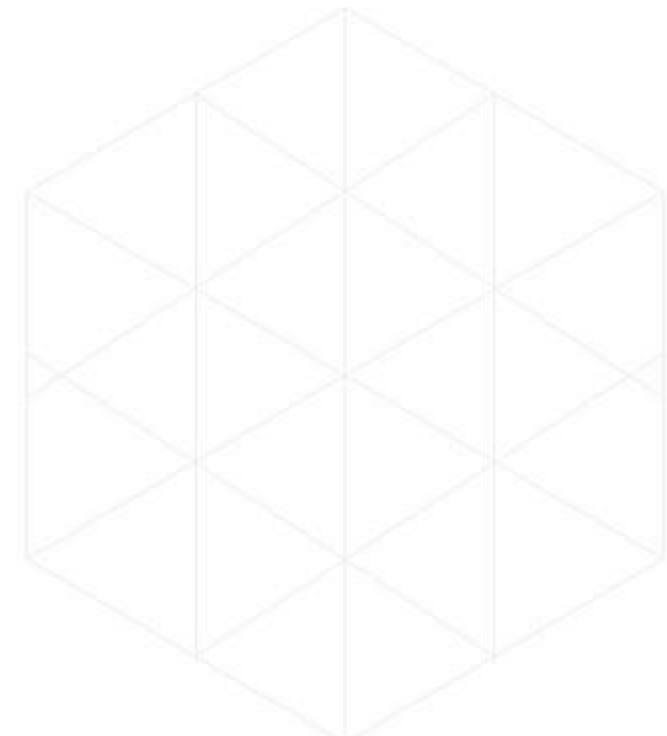
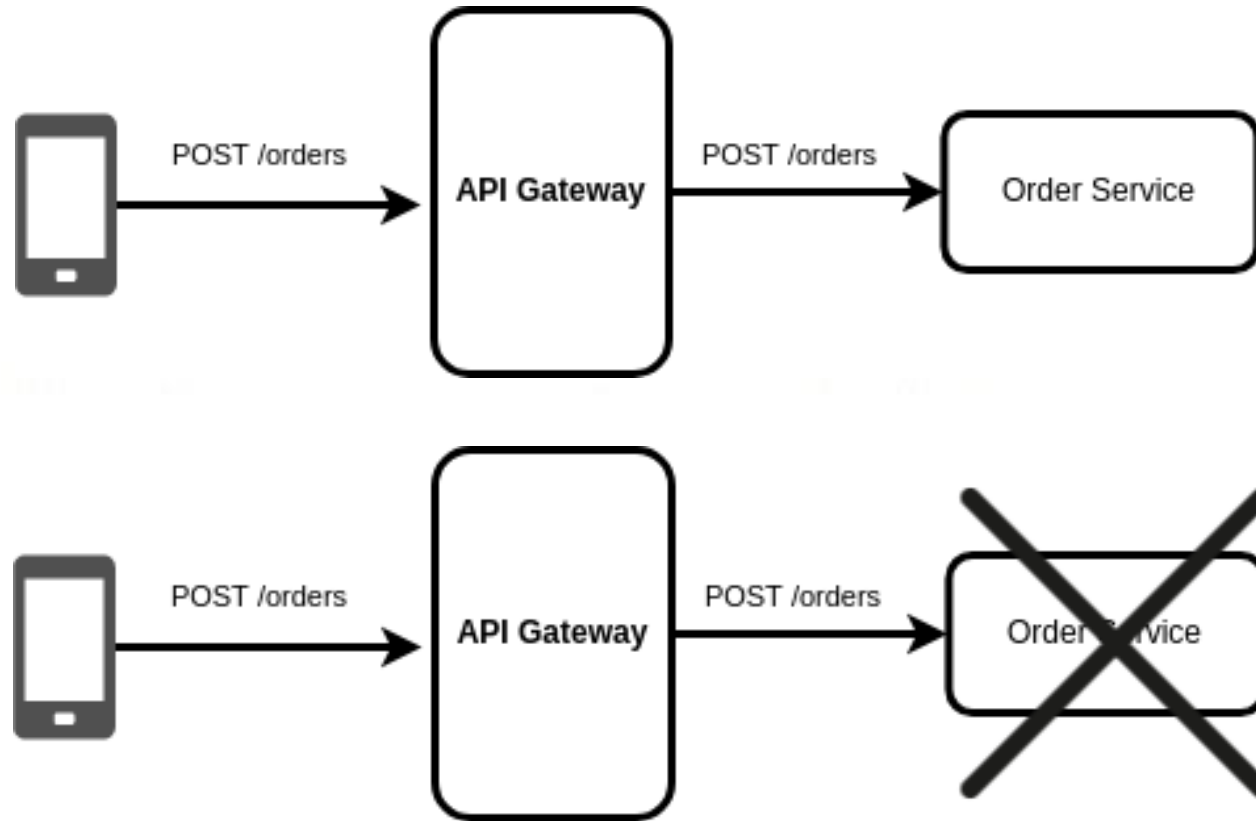
# *Circuit Breaker e Service Discovery*

# Circuit Breaker

- <http://microservices.io/patterns/reliability/circuit-breaker.html>
- Pattern que ajuda a tratar falhas em *requisições síncronas*.
- Como são aplicações separadas, o serviço pode ter problemas em responder a requisição em tempo hábil.
- Caso não seja tratado, pode bloquear um grande número de requisições



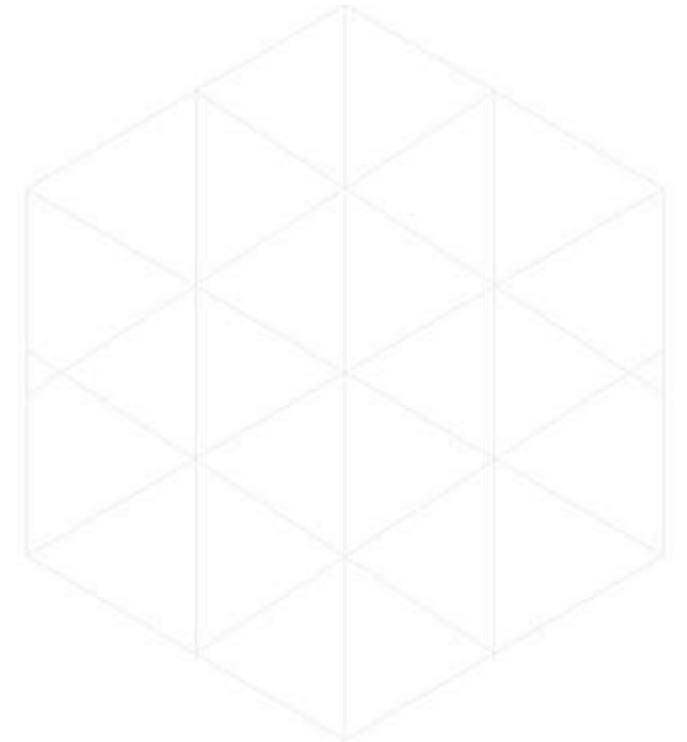
# Circuit Breaker





# *Circuit Breaker*

- Deve-se sempre tratar falhas parciais.
- Para isso, temos dois passos:
  - Proxies
  - Estratégia de recuperação de falhas

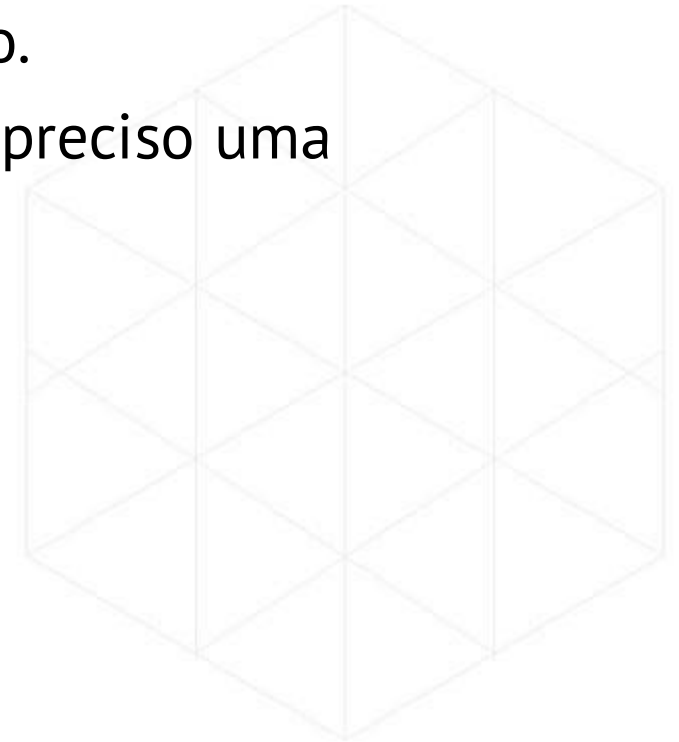
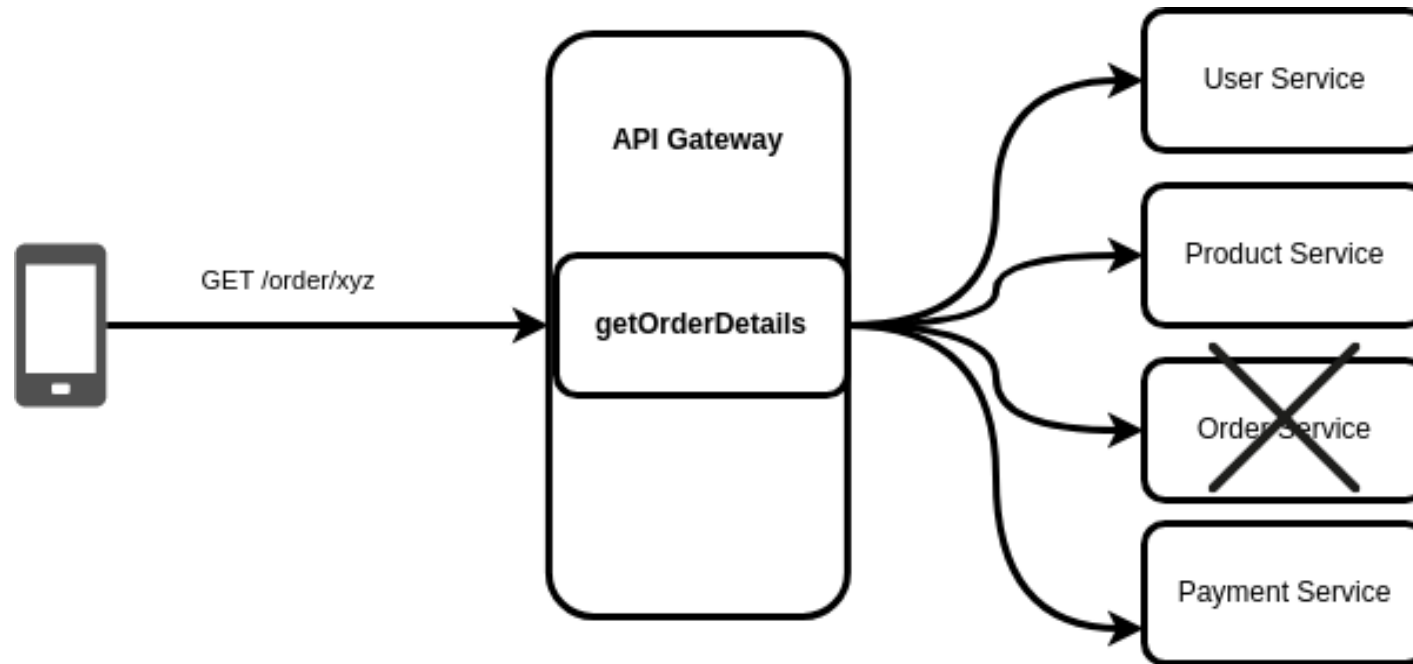


# *Proxies*

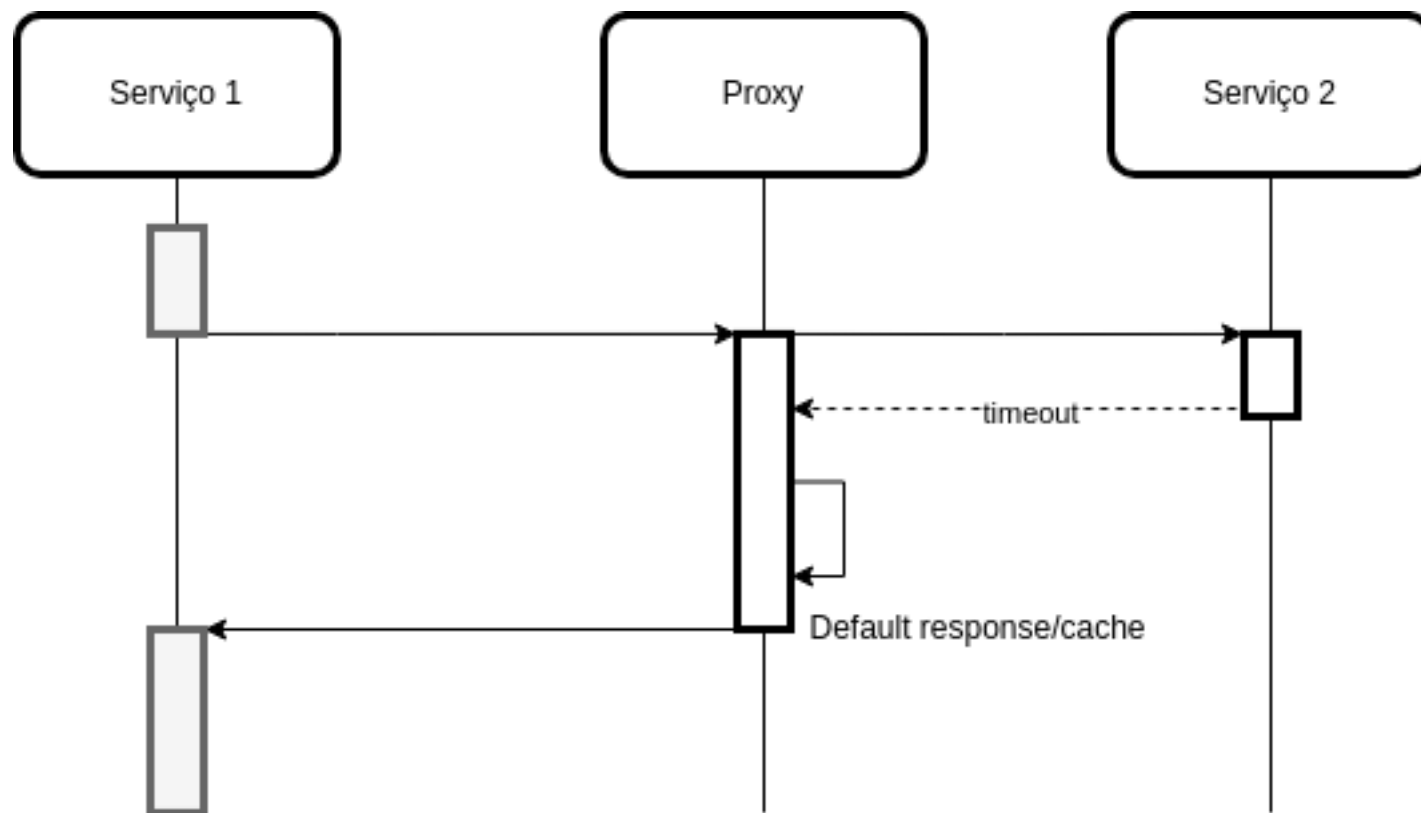
- Em uma arquitetura de microsserviços elementos que possam representar falhas, como timeouts ou excesso de requisições devem ser tratados com muita importância
- Existem soluções prontas para esse tipo de abordagem, como o **Hystrix** e **Resilience4j**
- <http://techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html>
- <https://netflixtechblog.com/making-the-netflix-api-more-resilient-a8ec62159c2d>
- <https://www.youtube.com/watch?v=kR2sm1zell4>

# Recuperando de um serviço indisponível

- O uso de uma biblioteca é somente uma parte da solução.
- Muitas vezes basta retornar erro ao usuário. Em outras, é preciso uma estratégia de fallback ou mesmo um valor default



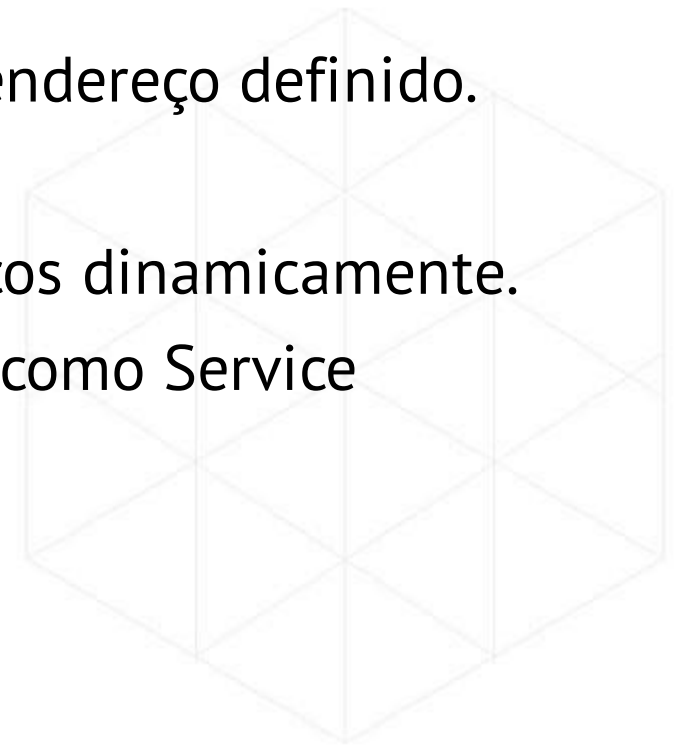
# *Recuperando de um serviço indisponível*



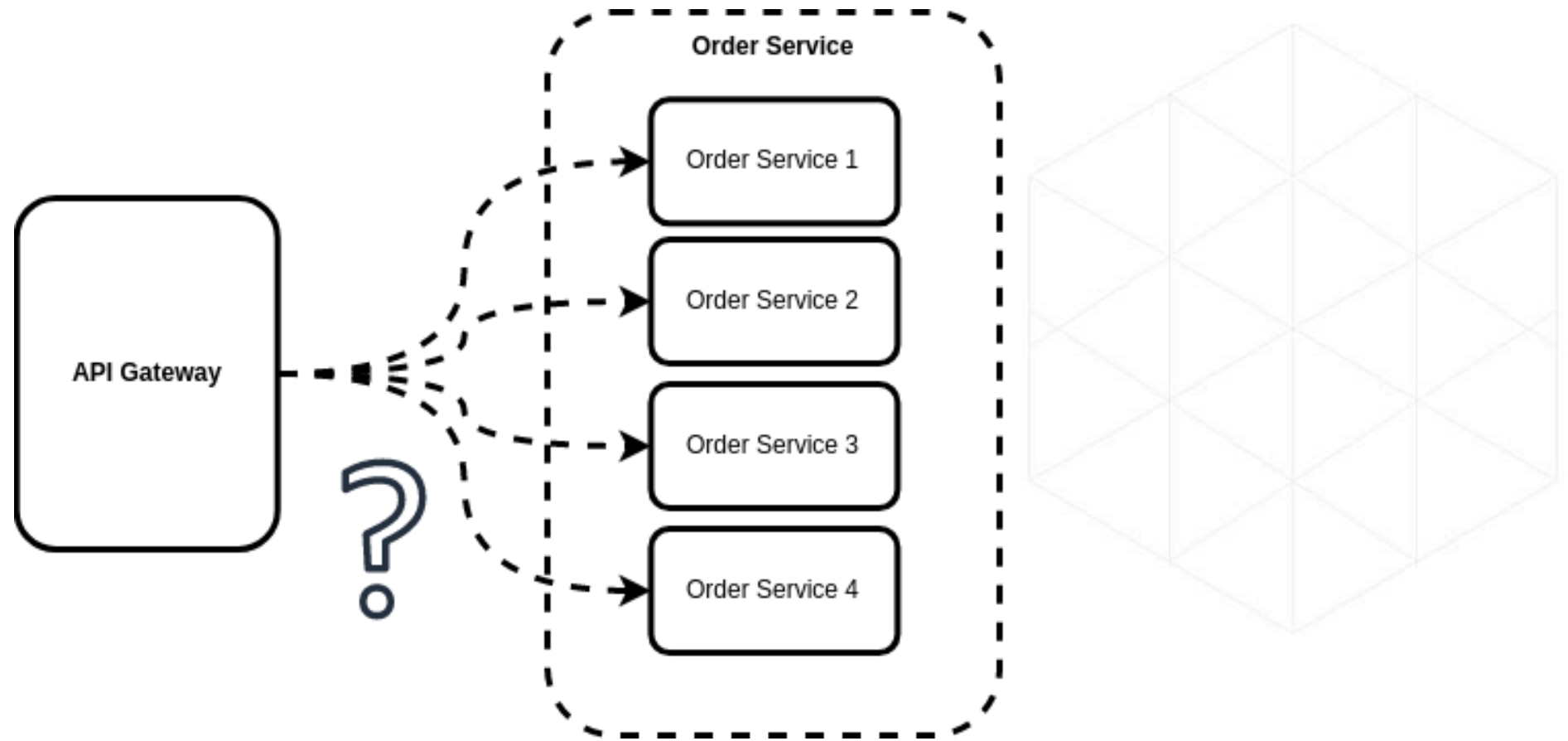


# ***Service Discovery***

- É comum mais de uma instancia de um serviço sem um endereço definido.
- É comum estes serviços estarem em redes distintas.
- Nesse contexto, a aplicação deve descobrir estes endereços dinamicamente.
- Essa espécie de banco de dados de serviços é conhecida como Service discovery.

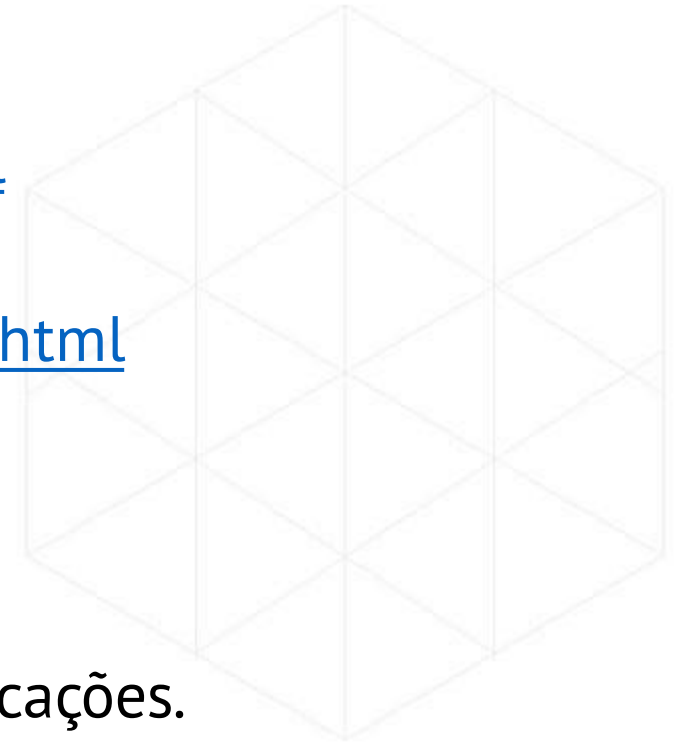


# *Service Discovery*

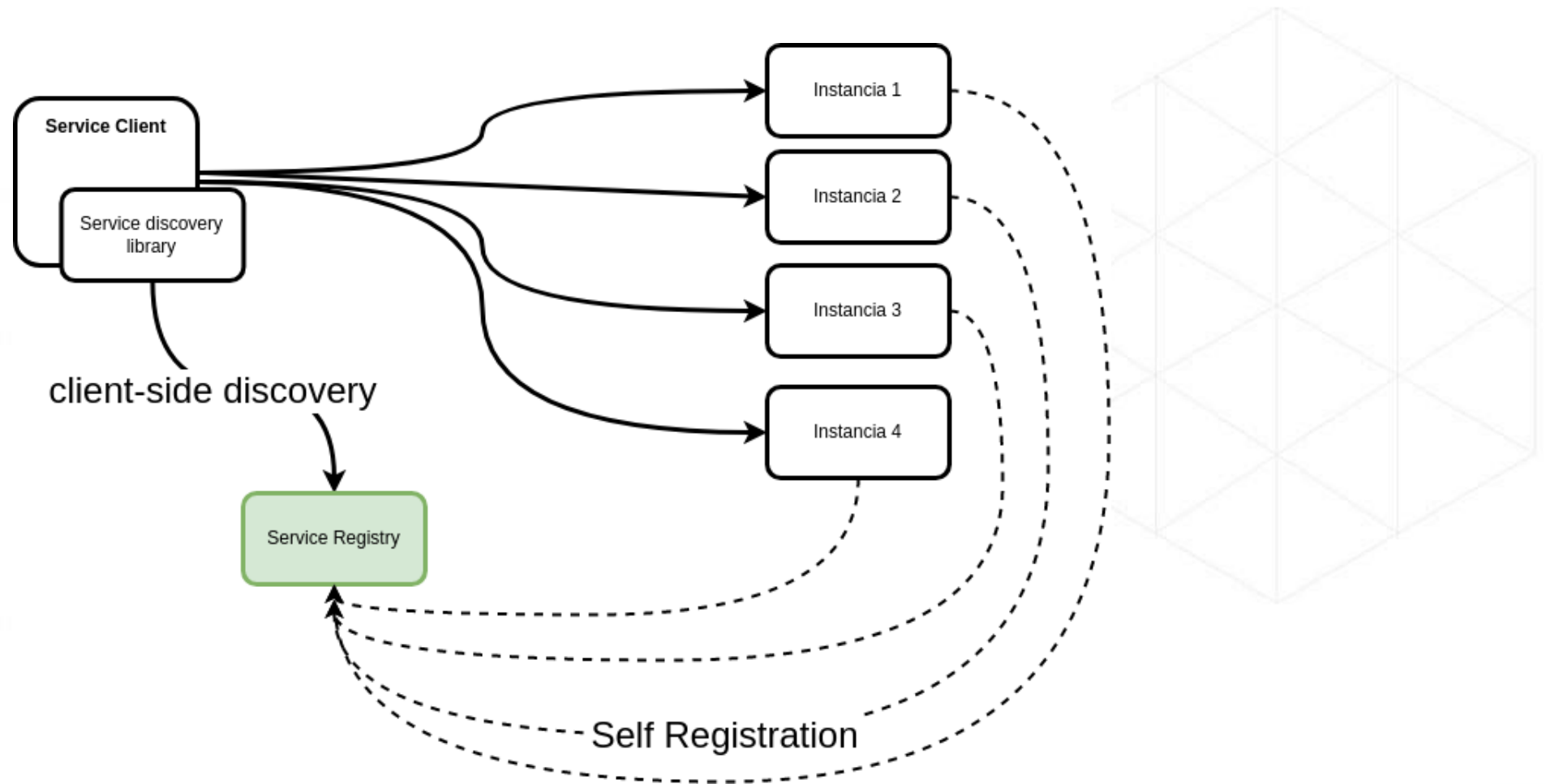


# *Patterns de Aplicação*

- Self Registration
  - <http://microservices.io/patterns/self-registration.html>
- Client-side-discovery
  - <http://microservices.io/patterns/client-side-discovery.html>
- Popularizados pela Pivotal e Netflix
  - Eureka
  - Spring Cloud
- São necessárias bibliotecas de Service Discovery nas aplicações.

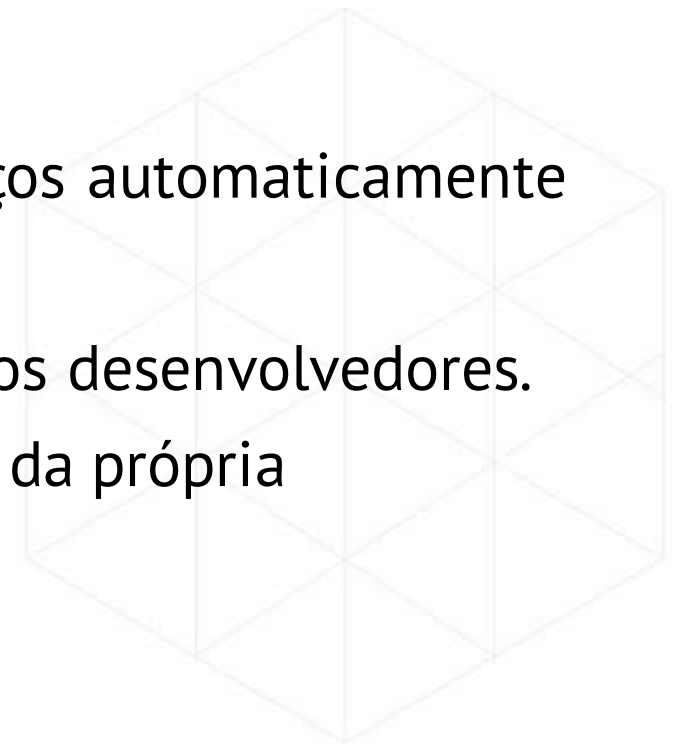


# Patterns de Aplicação

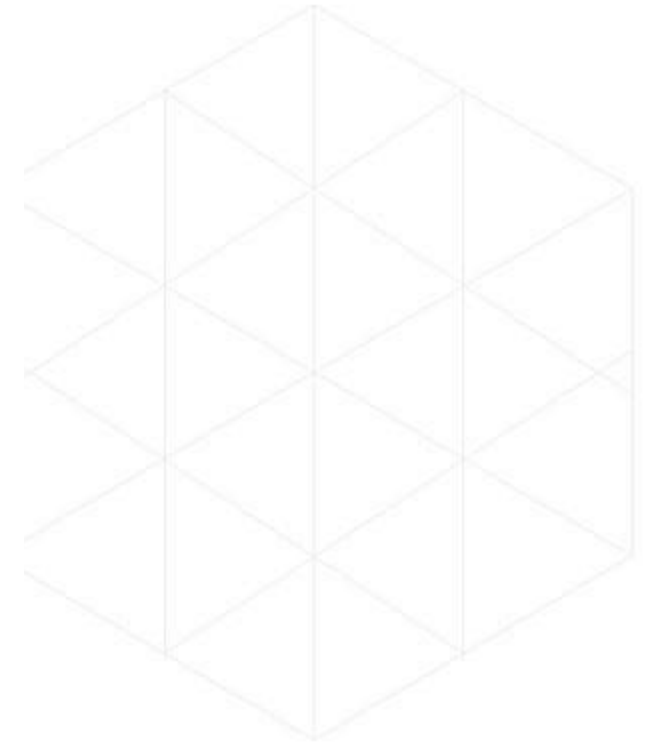
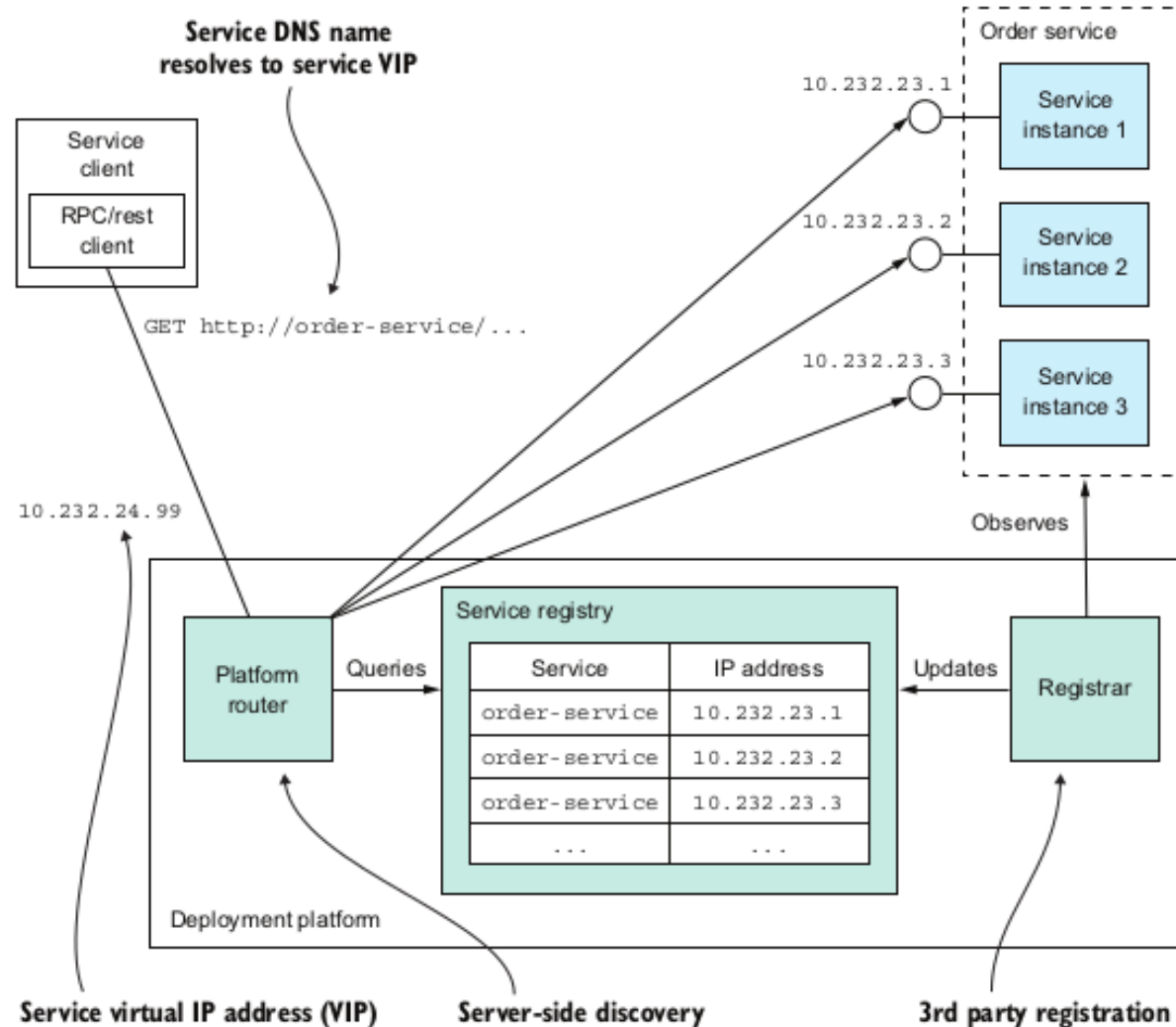


# ***Patterns de infraestrutura***

- Infra de deploy já fornece IP virtual a cada serviço.
- Fornece DNS que resolve no IP virtual, roteando os serviços automaticamente
- Kubernetes já utiliza este tipo de pattern.
- A grande vantagem é que não é necessário a interação dos desenvolvedores.
- Já a desvantagem é que só suporta discovery de serviços da própria plataforma.

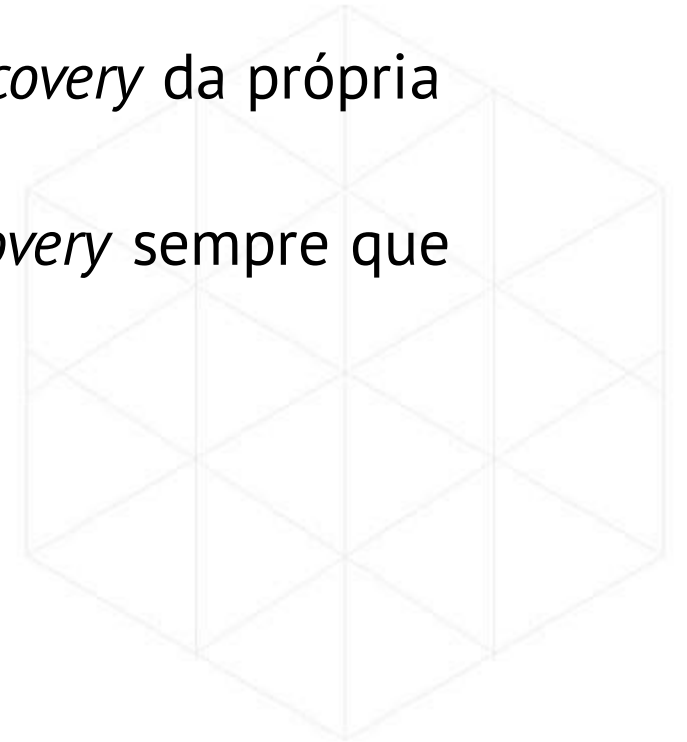


# Patterns de infraestrutura



# ***Patterns de infraestrutura***

- Como dito antes, uma desvantagem é que só suporta *discovery* da própria plataforma.
- Apesar dessa limitação, é recomendável usar *service discovery* sempre que possível.





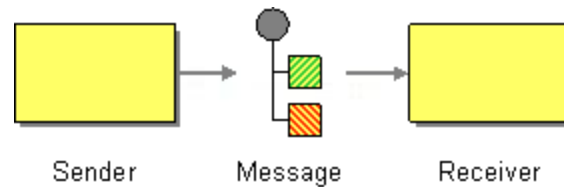


# *Comunicação assíncrona*



# Mensageria

- <http://microservices.io/patterns/communication-style/messaging.html>
- Utiliza canais (*message channels*)
- Consiste em um header e um body



# ***Tipos de mensagem***

- Documento
- Comando
- Evento



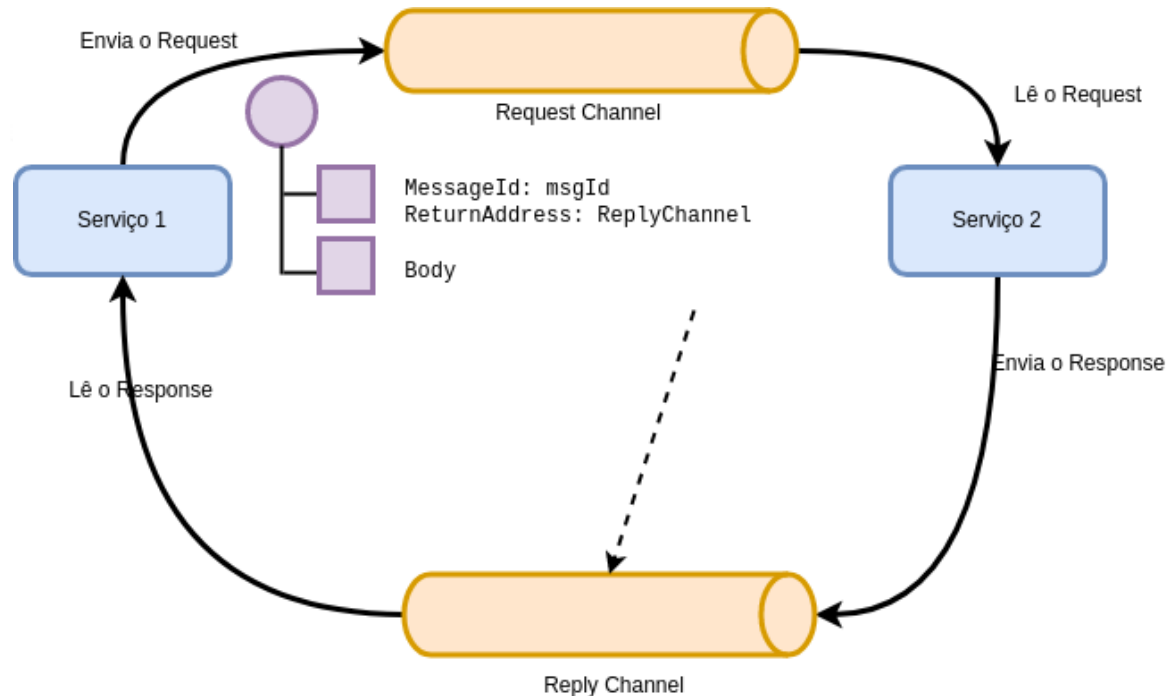
# ***Message Channel***

- Abstração da infraestrutura da mensagem.
- End-to-end.
- Publish-subscribe.
- <http://www.enterpriseintegrationpatterns.com/PointToPointChannel.html>
- <http://www.enterpriseintegrationpatterns.com/PublishSubscribeChannel.html>



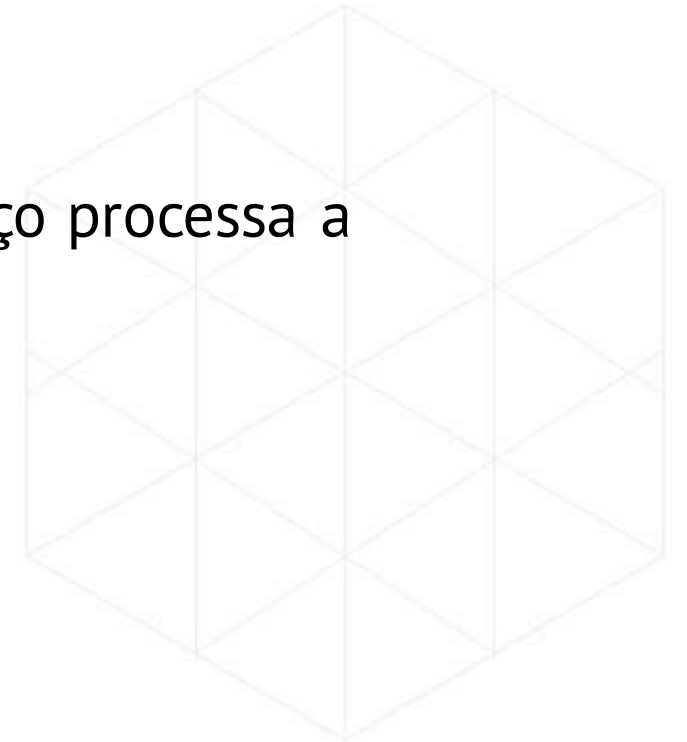
# ***Request/Response assíncrono***

- É possível utilizar request/response mesmo sendo mensageria.
- Nesse processo deve ser especificado para onde a resposta deve ir.



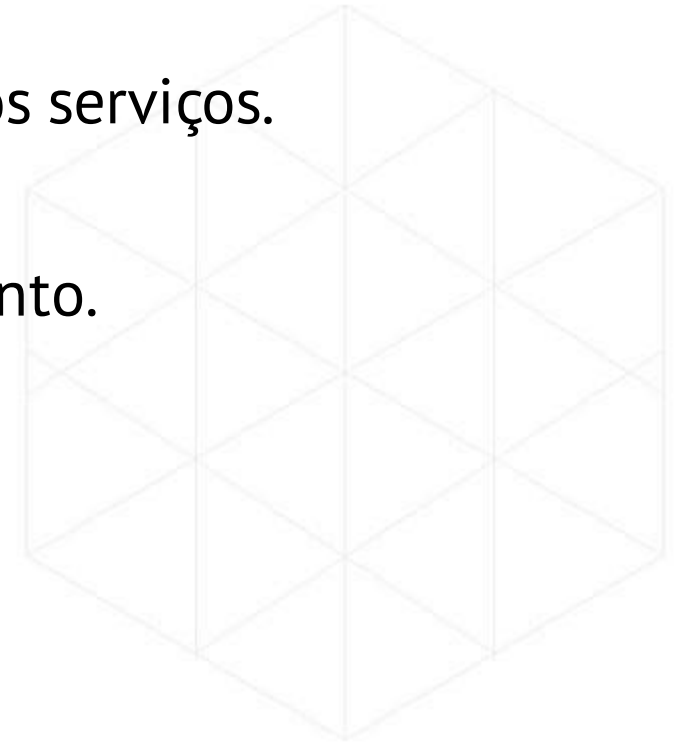
# ***One-way notifications***

- Notificações unilaterais.
- Cliente envia mensagem para o canal do serviço. O serviço processa a mensagem.



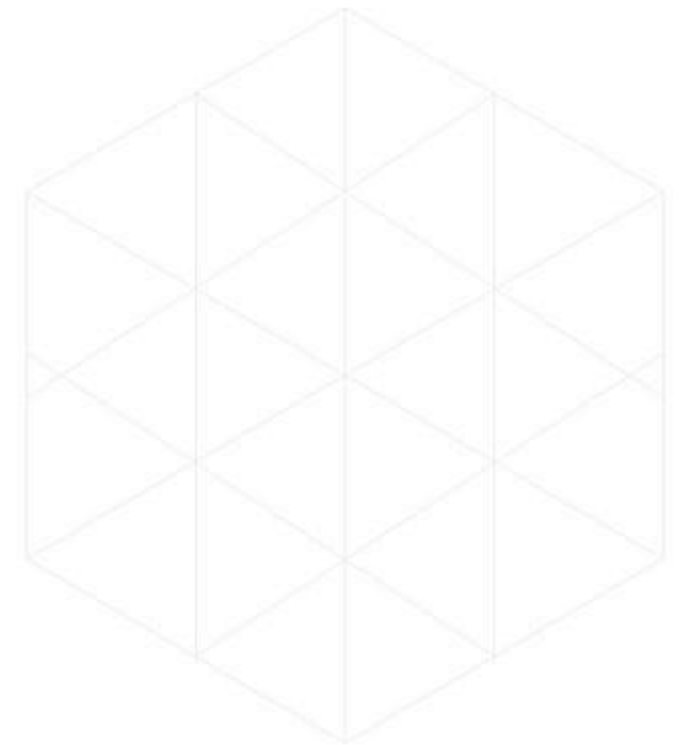
# ***Publish/Subscribe***

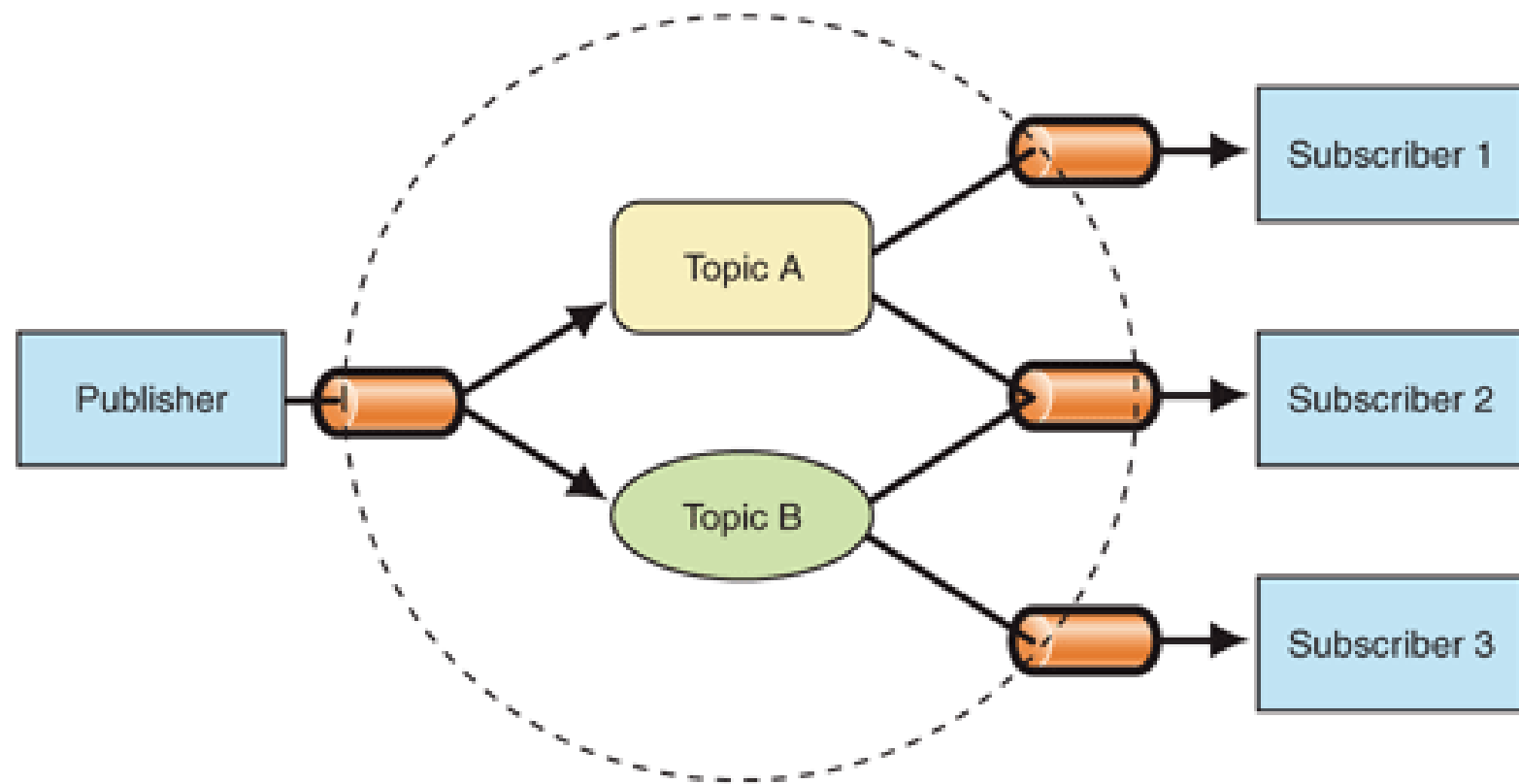
- Publica mensagem em canal que pode ser lido por muitos serviços.
- Cada consumidor faz seu próprio processamento.
- Normalmente esse tipo de mensagem é chamado de evento.



# ***API baseada em mensagens***

- Mesmos formatos de uma API síncrona.
- Não existe uma linguagem de documentação formal.



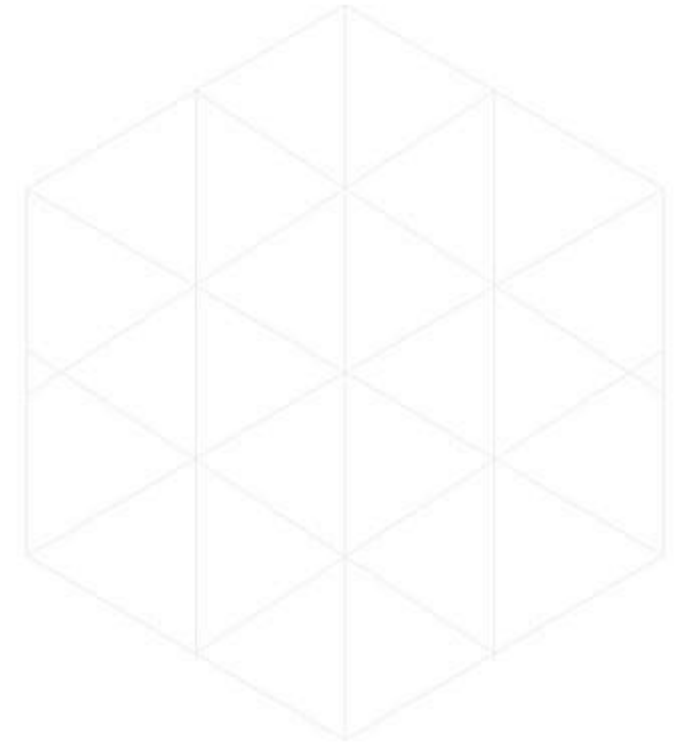
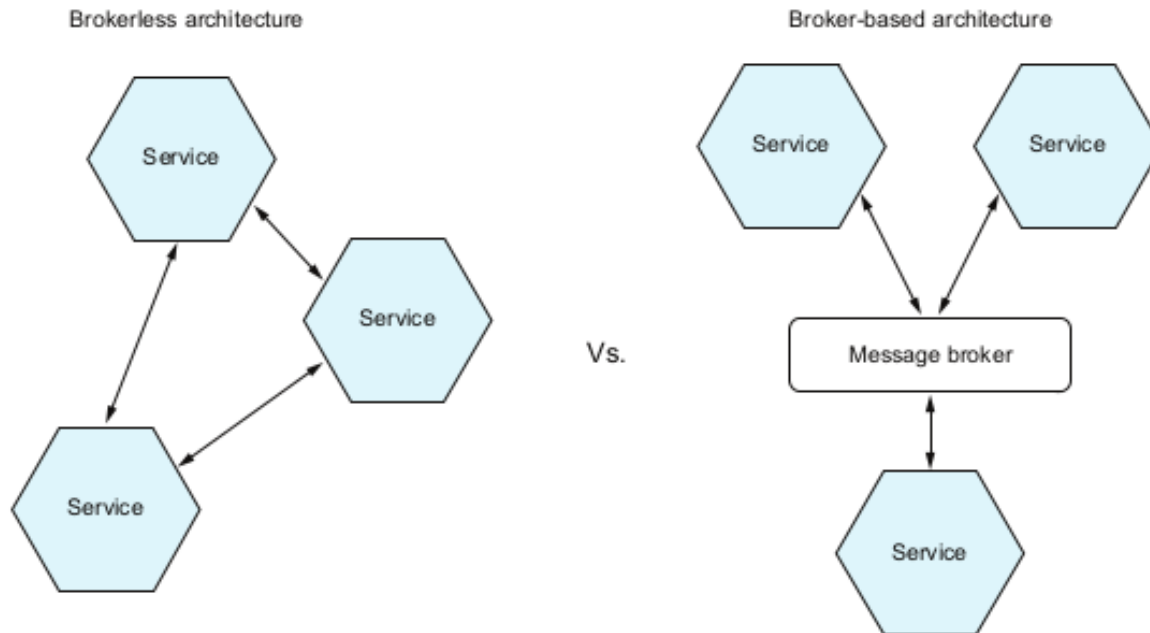


Communication Infrastructure



# *Message broker*

- Serviço que centraliza o envio/recepção da mensagem



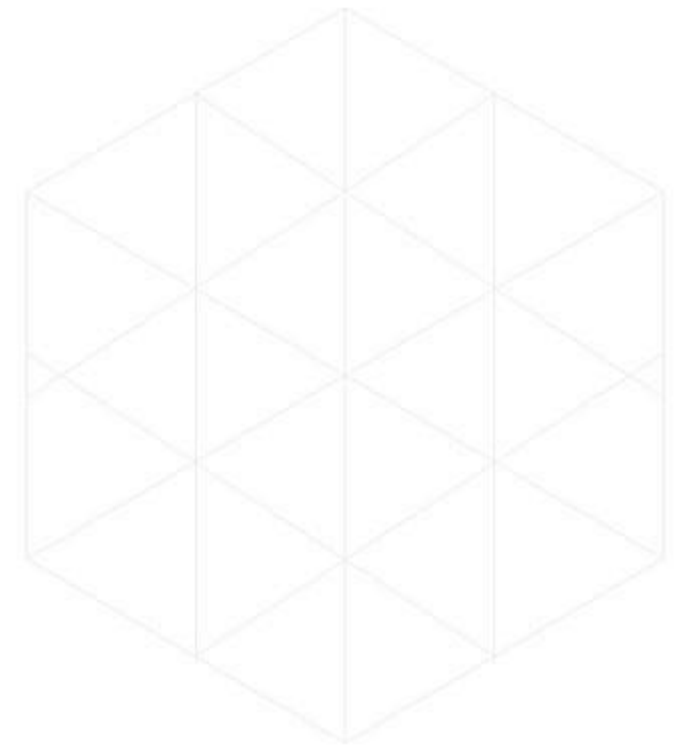
# *Message Broker*

- ActiveMQ ([<http://activemq.apache.org>](<http://activemq.apache.org/>))
- RabbitMQ ([<https://www.rabbitmq.com>](<https://www.rabbitmq.com/>))
- Apache Kafka ([<http://kafka.apache.org>](<http://kafka.apache.org/>))



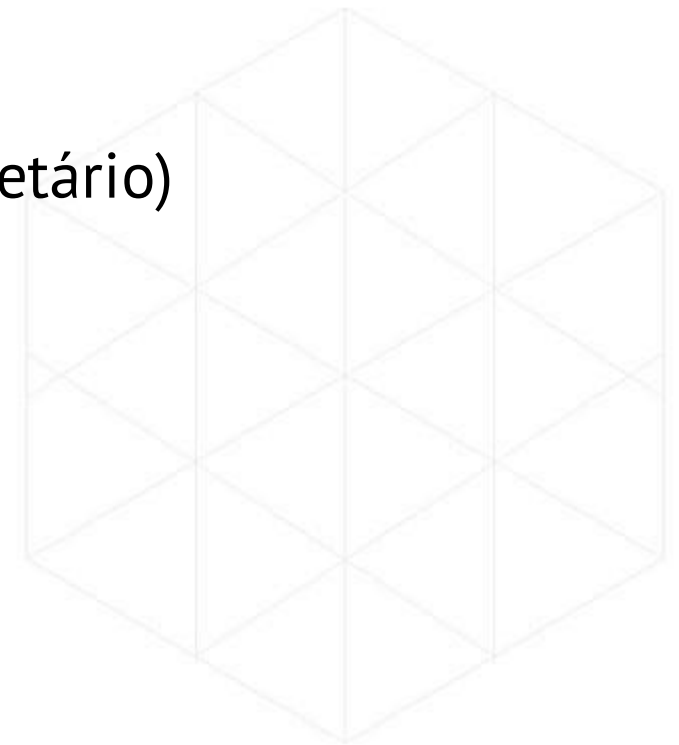
# ***Message Broker baseado em nuvem***

- AWS Kinesis
- AWS SQS
- PubSub
- ...



# ***Escolha de um Message Broker***

- Linguagens suportadas
- Padrões de mensagens suportadas (AMQP, STOMP, proprietário)
- Ordenação de mensagens
- Garantias de entrega
- Durabilidade
- Escalabilidade
- ...



# ***Message Brokers e seus channels***

Message Broker	Point-to-pont	Publish-subscribe
JMS	Queue	Topic
Apache Kafka	Topic	Topic
AMPQ (RabbitMQ)	Exchange + Queue	Queue por consumidor
AWS SQS	Queue	-

# Mensageria e disponibilidade

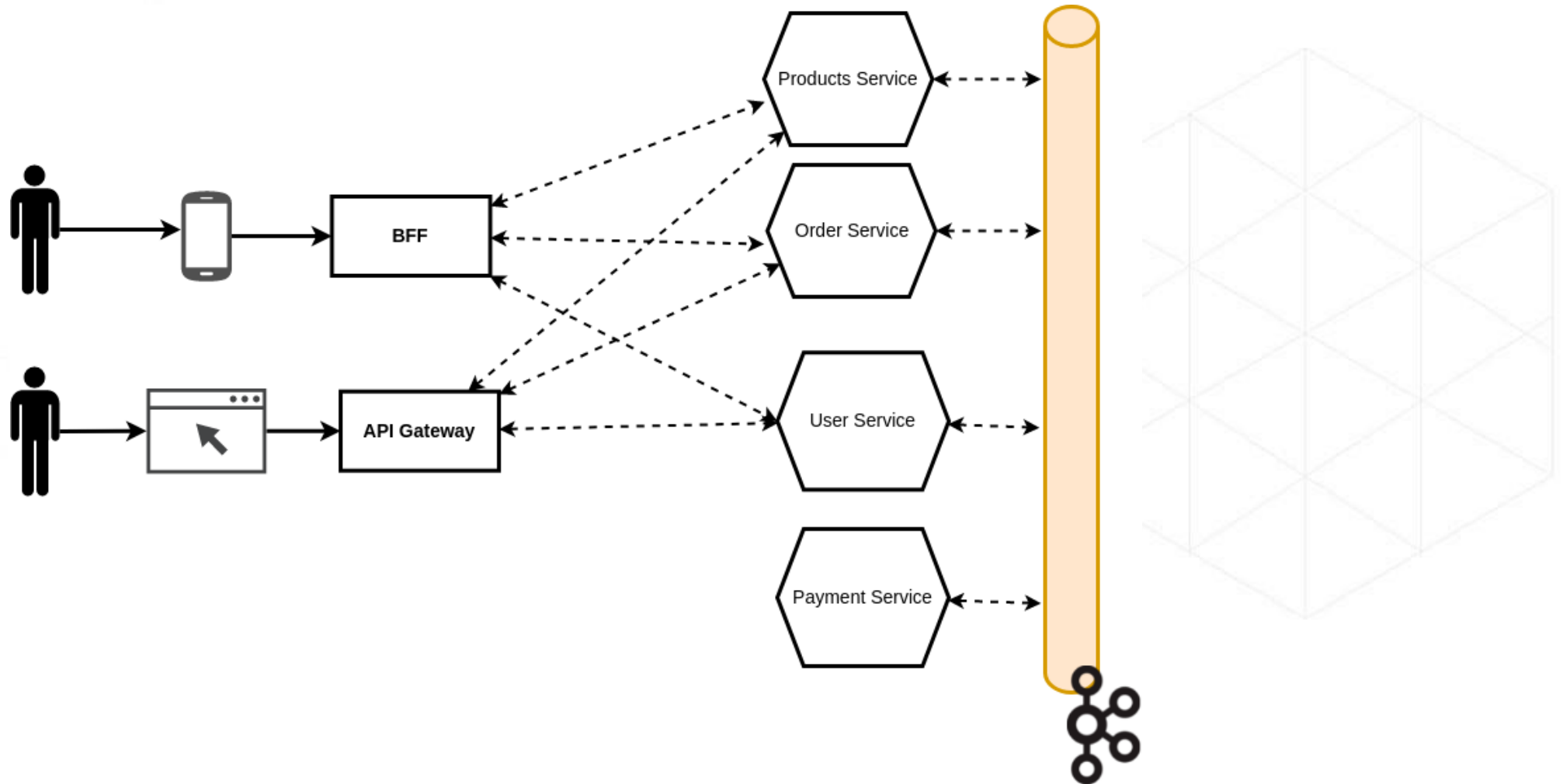
- Sempre que possível e viável, **comunicação assíncrona deve ser utilizada.**
- Todos os serviços síncronos devem estar disponíveis ao mesmo tempo.
- *Se um serviço tem 99.5% de disponibilidade e ele depende outros dois serviços também com 99.5% de disponibilidade, a disponibilidade do primeiro cai para 98.5% (99.5 elevado a 3).*





*Comunicando nossos serviços*

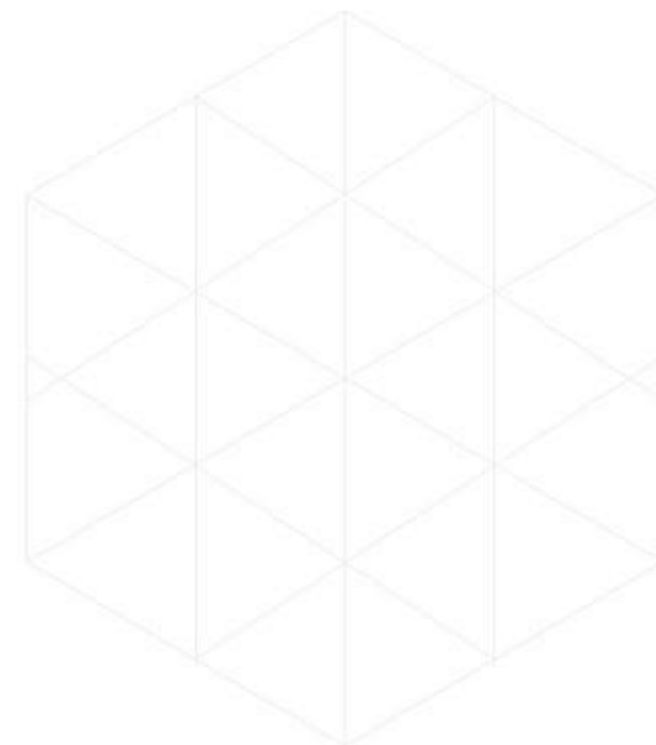
# Comunicação de serviços





# ***Próximos passos***

- Transações



## ***Para saber mais...***

- <https://www.programmableweb.com/news/how-to-design-great-apis-api-first-design-and-raml/how-to/2015/07/10>
- <https://auth0.com/blog/beating-json-performance-with-protobuf/>
- <https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>
- <http://martinfowler.com/articles/richardsonMaturityModel.html>
- <http://www.openapis.org>
- [www.grpc.io](http://www.grpc.io)
- <http://microservices.io/patterns/reliability/circuit-breaker.html>

## ***Para saber mais...***

- <http://techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html>
- <https://netflixtechblog.com/making-the-netflix-api-more-resilient-a8ec62159c2d>
- <https://www.youtube.com/watch?v=kR2sm1zell4>
- <http://microservices.io/patterns/self-registration.html>
- <http://microservices.io/patterns/client-side-discovery.html>
- <http://microservices.io/patterns/communication-style/messaging.html>

## ***Para saber mais...***

- <https://grpc.io/docs/languages/java/basics/>
- <http://www.enterpriseintegrationpatterns.com/PointToPointChannel.html>
- <http://www.enterpriseintegrationpatterns.com/PublishSubscribeChannel.html>

# OBRIGADO!

## **Centro**

Rua Formosa, 367 - 29º andar Centro, São Paulo - SP, 01049-000

## **Alphaville**

Avenida Ipanema, 165 - Conj. 113/114 Alphaville, São Paulo - SP, 06472-002

**+55 (11) 3358-7700**

[contact@7comm.com.br](mailto:contact@7comm.com.br)

**7comm**  
Serviços e Soluções em TI