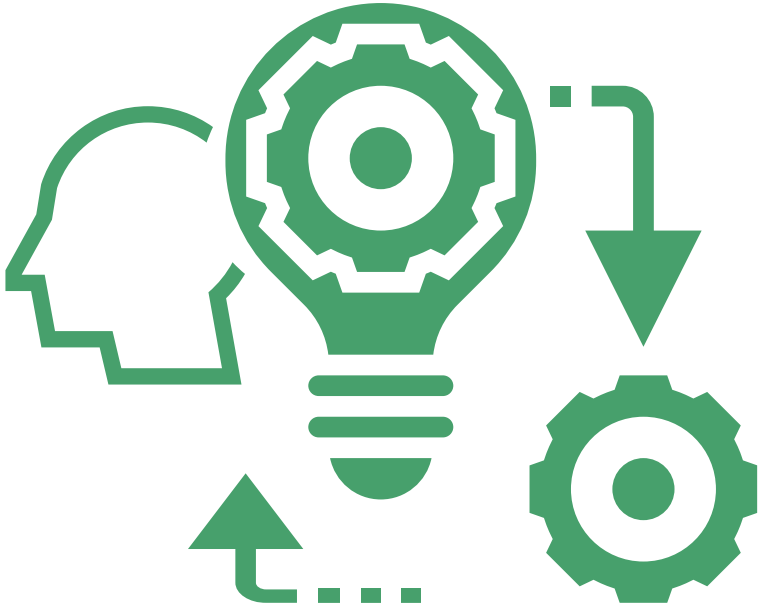


Paradigmas de Programação

Entregáveis

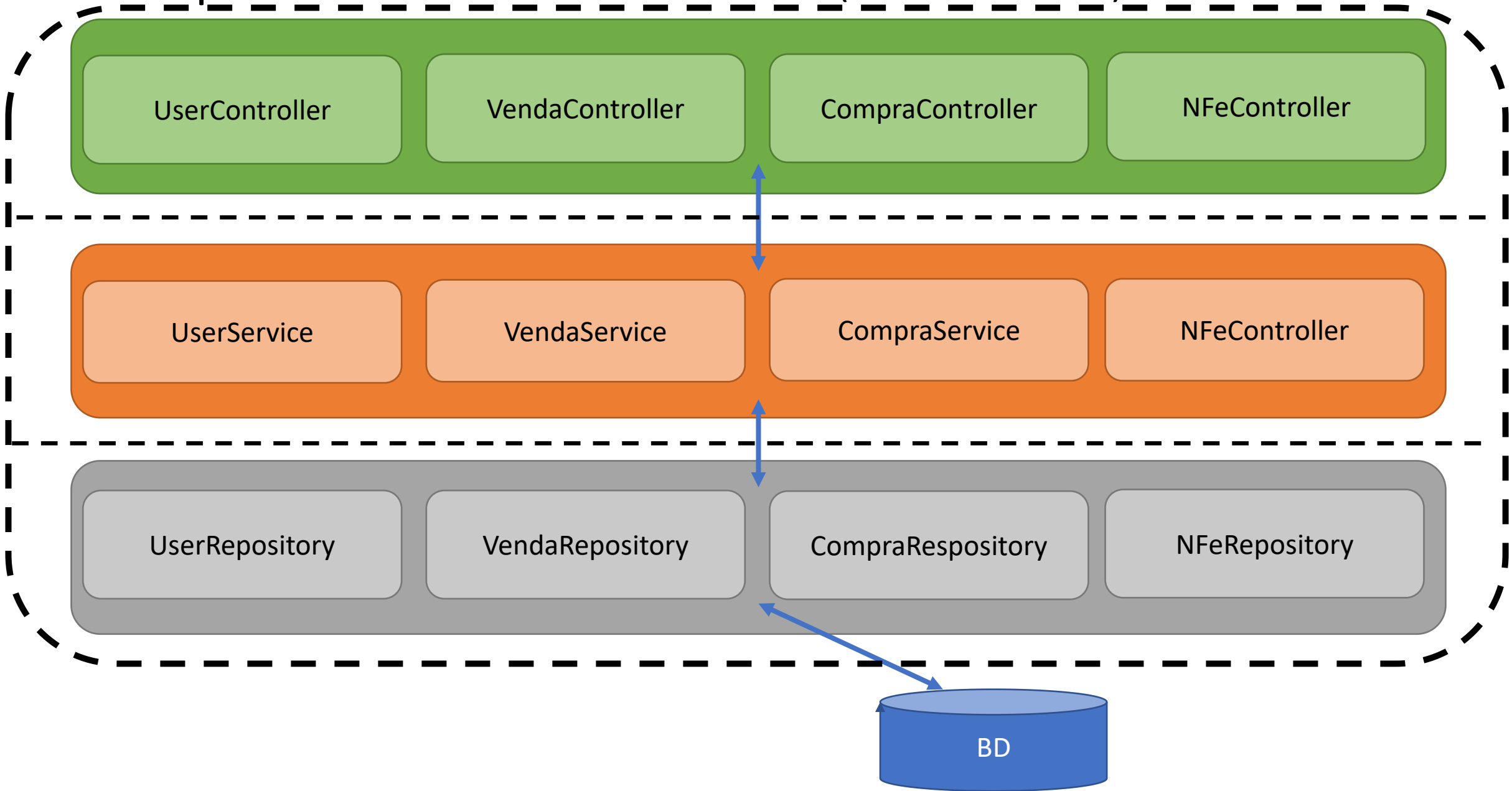
SOLID - Princípio Da Inversão de Dependência

- **Princípio da Inversão de Dependência (Dependency Inversion Principle - DIP)** tem como objetivo fundamental a redução do acoplamento entre os componentes de um sistema.

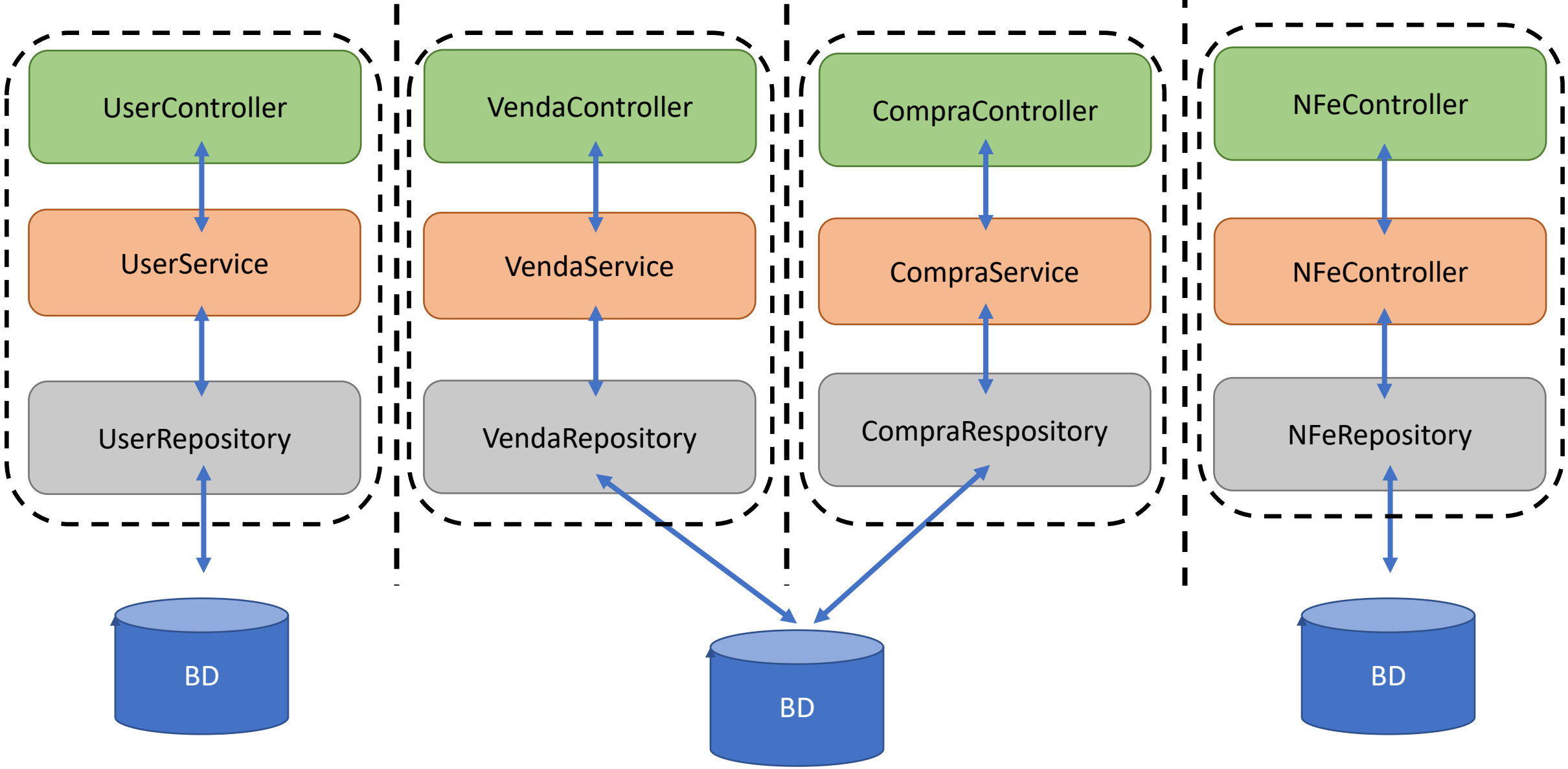
Módulos de alto nível NÃO devem depender de módulos de baixo nível, mas sim de ABSTRAÇÕES.

- Isso permite uma maior flexibilidade e facilidade de manutenção do código, promovendo baixo acoplamento e alta coesão.

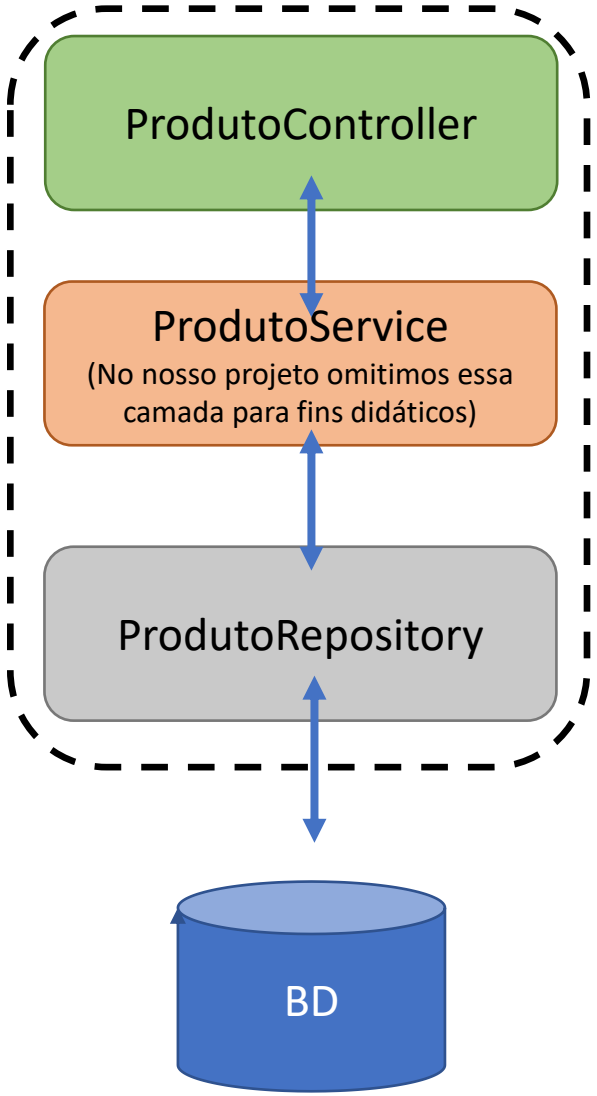
Arquitetura de Camadas (Monolítico)



Microservices com Camadas



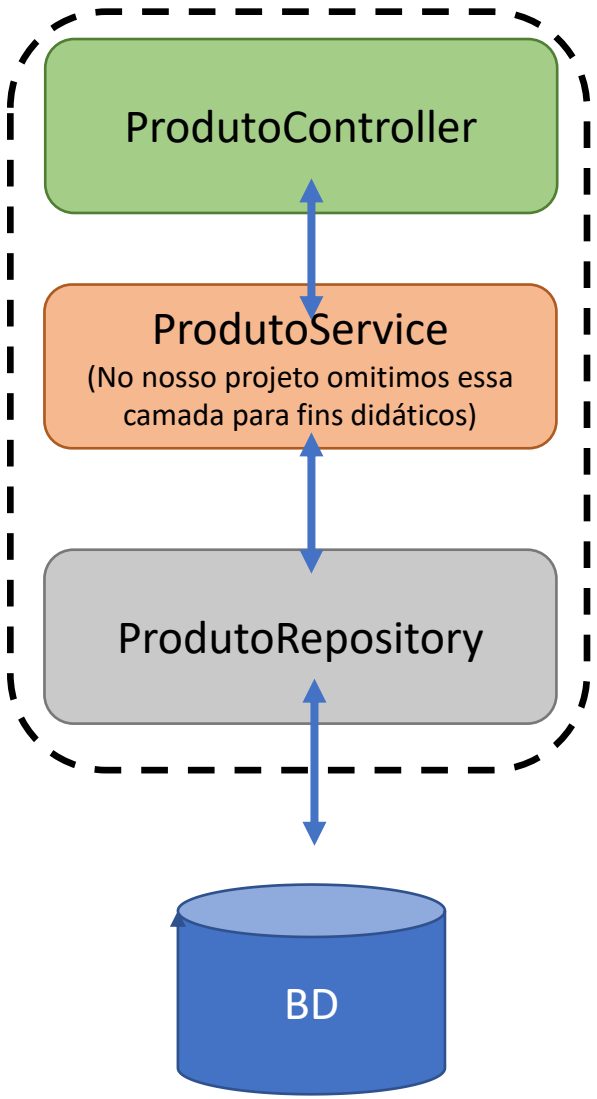
Nosso exemplo



- Em uma arquitetura de camada, as camadas superiores **dependem** das camadas inferiores.

- Nesse exemplo a classe ProdutoController depende da ProdutoService, e a ProdutoService depende da ProdutoRepository

Nosso exemplo

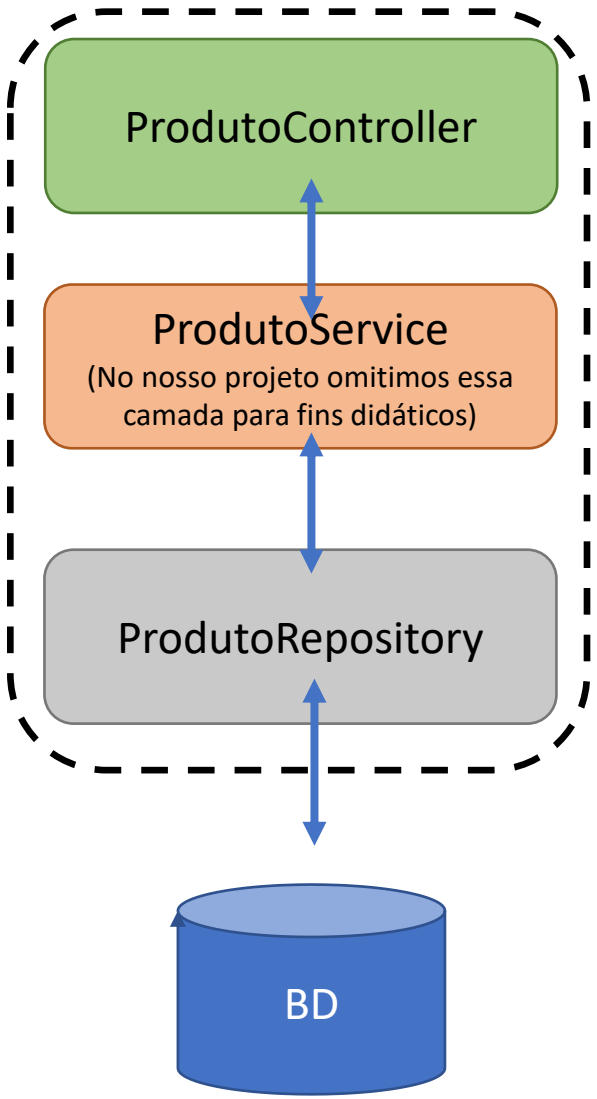


- O exemplo abaixo seria uma violação do Princípio da Inversão de Dependência.

```
public class ProdutoController {
        //Aqui estamos declarando uma dependência,
        // ou seja, para a ProdutoController funcionar ela
        // precisará de uma instância da ProdutoService,
        // nesse caso será um objeto da classe
        // ProdutoServiceImpl que implementa a interface
        // ProdutoService
        private final ProdutoService service;

        public ProdutoController() {
                //Aqui estamos instanciando diretamente
                // a partir da classe, ou seja, a dependência
                // é direta de uma classe concreta
                this.service = new ProdutoServiceImpl();
        }
        ...
}
```

Nosso exemplo



- Respeitando o Princípio da Inversão de Dependência.

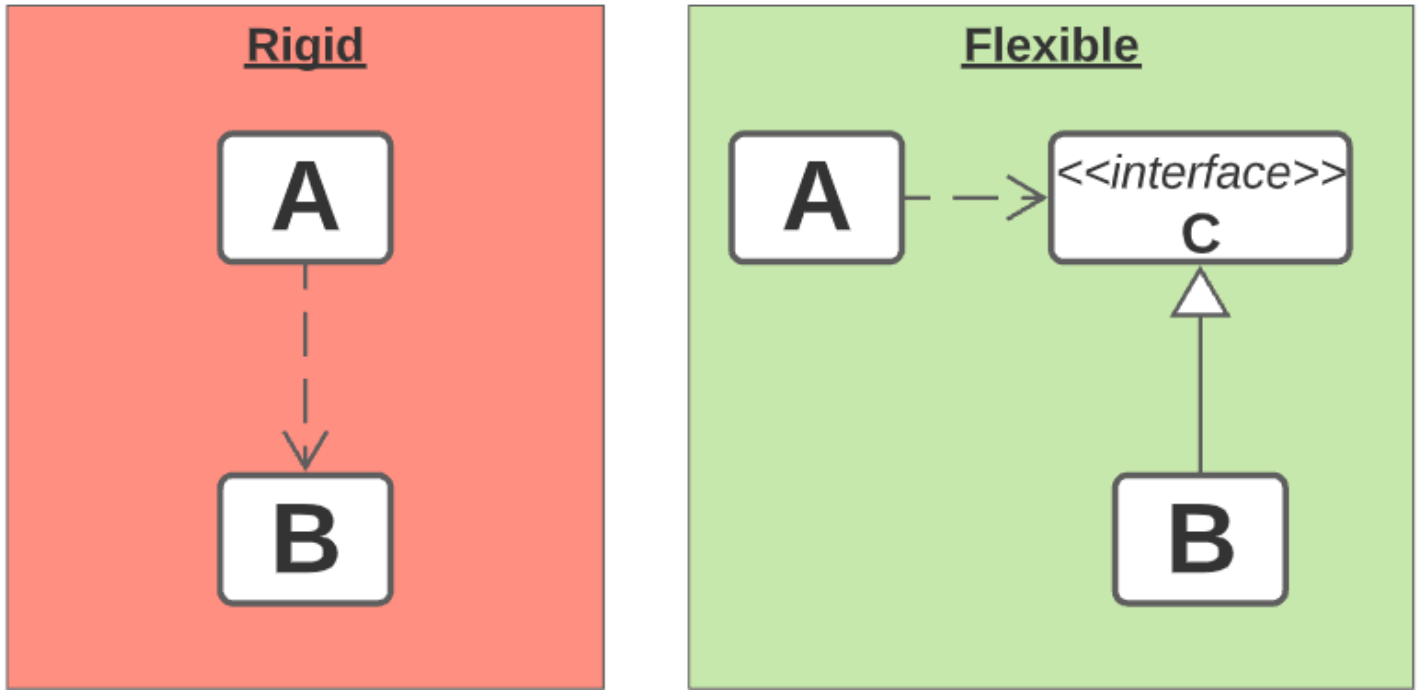
```
public class ProdutoController {
        //Aqui continuamos declarando uma dependência,
        // ou seja, para a ProdutoController funcionar ela
        // precisará de uma instância da ProdutoService
        // porém, aqui apenas referenciamos uma Interface e
        // e não uma classe concreta
        private final ProdutoService service;

        public ProdutoController(ProdutoService service) {
                //No método construtor então recebemos via
                // INJEÇÃO DE DEPENDÊNCIA uma implementação da
                // Interface
                this. service = service;
        }
        ...
}
```

Porque “Inversão”???

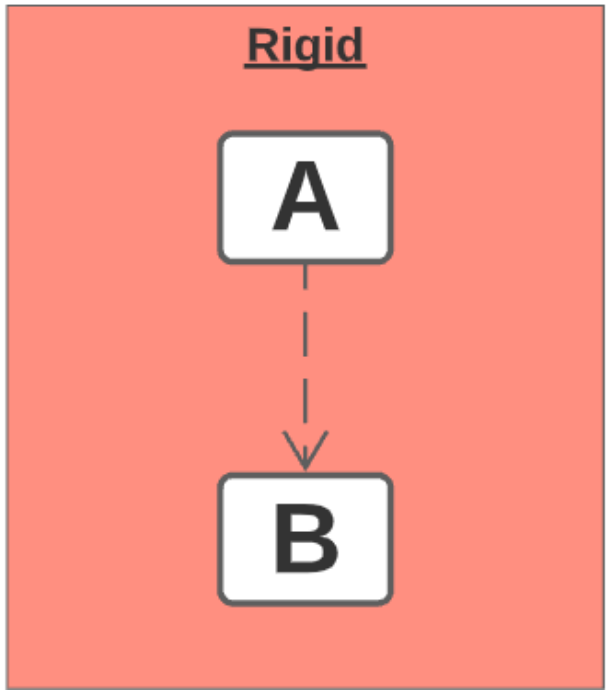
- Sem o DIP a classe A depende diretamente da classe B
- Com o DIP a classe A depende de uma Interface (abstração) C, e a classe B implementa a Interface C
- Sem o DIP, qualquer mudança em B, pode gerar impactos em A
- Com o DIP, mudanças em B

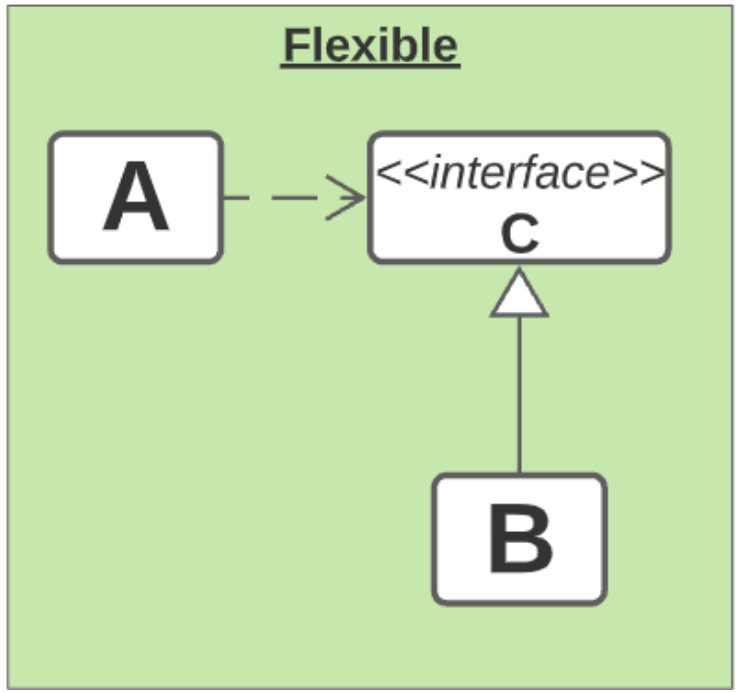
não geram impactos em A



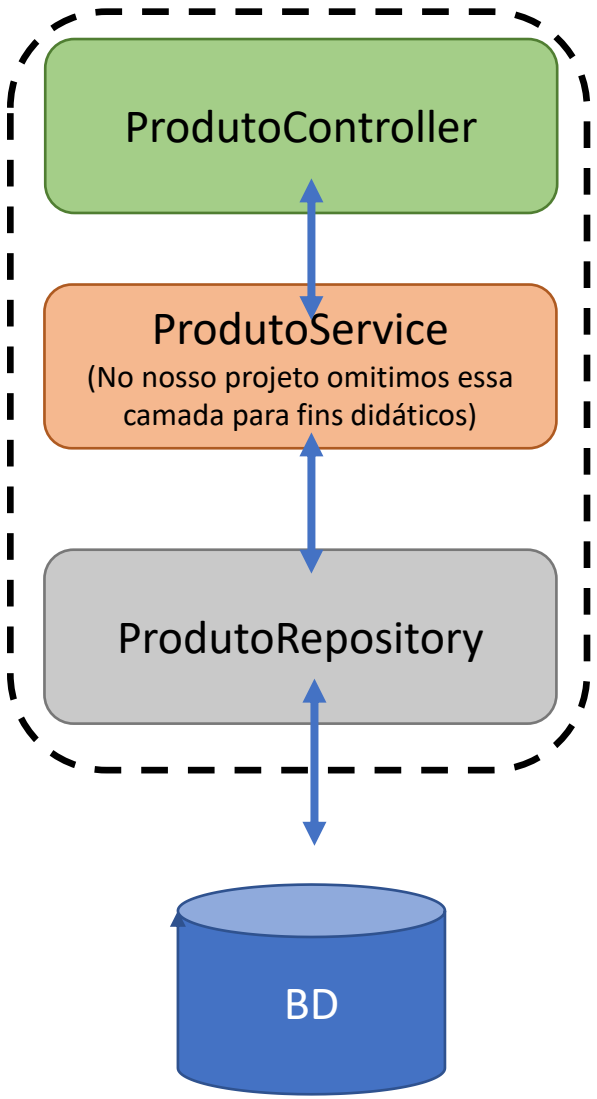
Então A depende de uma **“abstração”**???

- SIM.
- Mas se é uma abstração, como iremos utilizar???
- Aqui entra a Injeção de dependência





Nosso exemplo



- Injeção de Dependência

```
public class ProdutoController {
        //Aqui temos a dependência apontando para uma
        //  Interface
        private final ProdutoService service;

        public ProdutoController(ProdutoService service) {
                //No método construtor então recebemos via
                // INJEÇÃO DE DEPENDÊNCIA uma implementação da
                // Interface, ou seja, pode haver mais de uma
                // implementação, e ambas podem ser utilizadas
                this. service = service;
        }
        ...
}
```

Entregável

Pesquise e compare as duas formas de Injeção de Dependências disponíveis no Spring Boot:

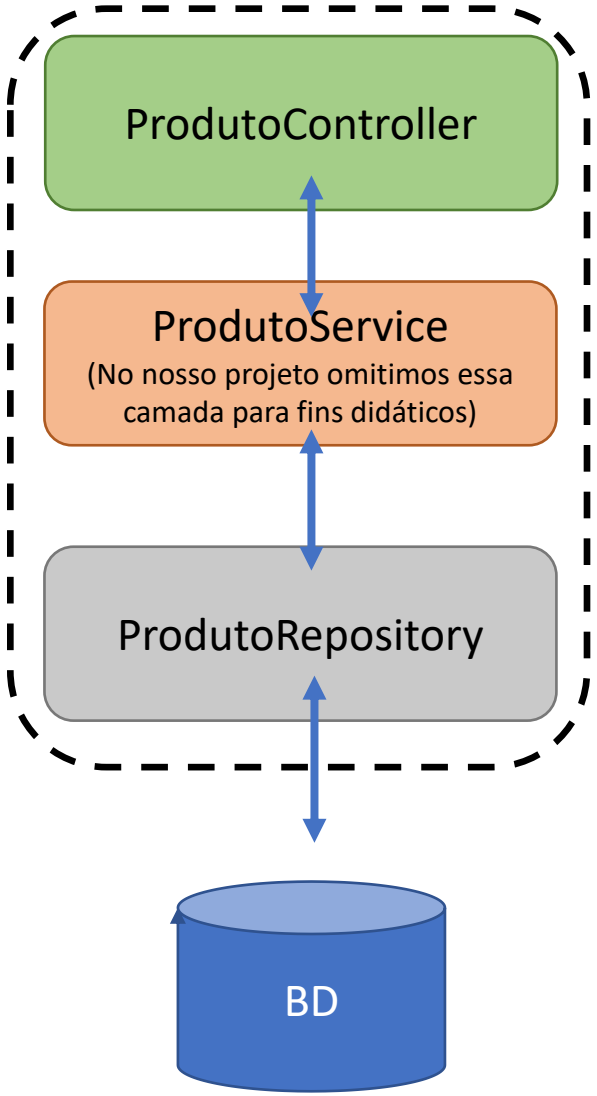
1. Injeção de dependência via `@Autowired`.
2. Injeção de dependência via método construtor.

Sua resposta deve incluir as seguintes análises:

- Explique como cada abordagem funciona no Spring Boot.
- Aponte as vantagens e desvantagens de cada método.
- Dê exemplos de situações em que cada abordagem seria mais adequada.

Lembre-se de ser claro e objetivo, demonstrando entendimento sobre o impacto dessas práticas na estrutura e manutenibilidade do código.

Nosso exemplo



- Injeção de Dependência

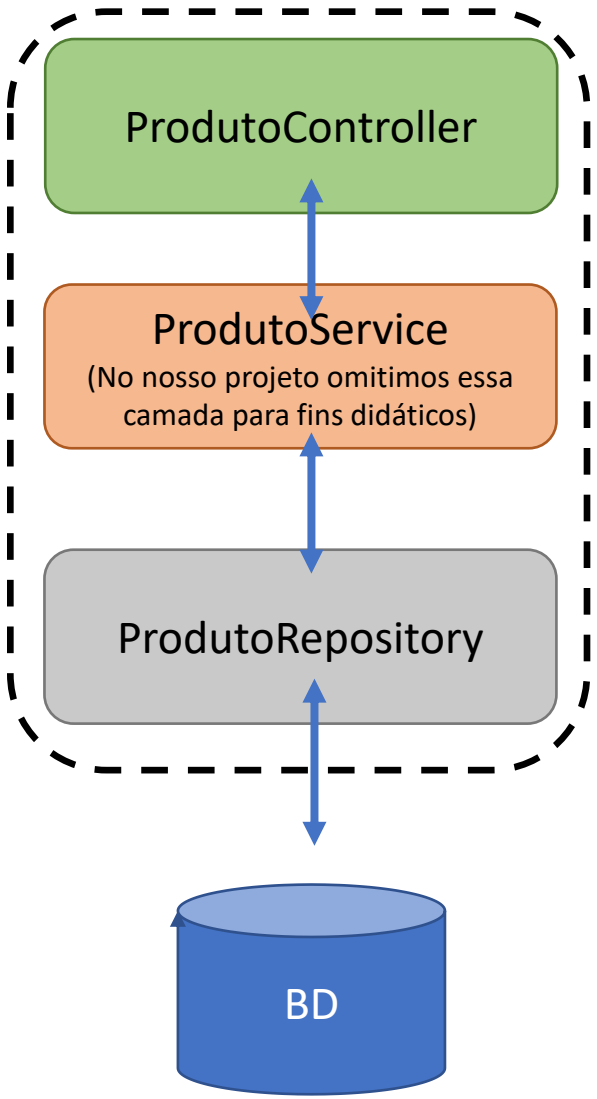
```
public class ProdutoController {
        //Aqui temos a injeção via Autowired
        @Autowired
        private ProdutoService service;

        //Nesse caso não precisamos do método construtor
        }
        ...
}
```

Injeção de Dependência via `@Autowired`

- **Como funciona:** A anotação `@Autowired` indica ao Spring que o campo ou método anotado deve ser automaticamente preenchido com um bean correspondente do contexto do Spring. O Spring procura por um bean do tipo correto e o injeta na propriedade marcada com `@Autowired`.
- **Vantagens:**
  - **Flexibilidade:** Permite injetar dependências em campos, métodos ou construtores.
  - **Simplicidade:** Sintaxe concisa e fácil de entender.
- **Desvantagens:**
  - **Menos explícita:** A relação entre as dependências pode não ser tão clara quanto na injeção por construtor.
  - **Possibilidade de nulos:** Se o Spring não encontrar um bean correspondente, o campo será nulo, podendo levar a NullPointerExceptions se não forem tratadas adequadamente.

Nosso exemplo



- Injeção de Dependência

```
public class ProdutoController {
        //Com a injeção via Construtor podemos declara-la como
        // "final", o que garante os benefícios da
        //  Imutabilidade
        private final ProdutoService service;

        public ProdutoController(ProdutoService service) {
                //No método construtor então recebemos via
                // INJEÇÃO DE DEPENDÊNCIA uma implementação da
                // Interface, ou seja, pode haver mais de uma
                // implementação, e ambas podem ser utilizadas
                this. service = service;
        }
        ...
}
```

Injeção de Dependência via Construtor

- **Como funciona:** O construtor da classe é utilizado para definir as dependências necessárias. O Spring instancia a classe e injeta as dependências correspondentes nos argumentos do construtor.
- **Vantagens:**
  - **Mais explícita:** As dependências da classe ficam claramente visíveis no construtor.
  - **Impossibilidade de nulos:** As dependências são obrigatórias, evitando NullPointerExceptions.
  - **Facilita testes:** Ao criar instâncias da classe em testes unitários, é obrigatório fornecer todas as dependências necessárias.
- **Desvantagens:**
  - **Menos flexível:** A ordem dos argumentos do construtor deve ser respeitada.
  - **Mais verboso:** Pode gerar construtores com muitos argumentos, especialmente para classes com muitas dependências.

Quando usar cada abordagem?

- `@Autowired`:
  - **Opcional:** Quando a dependência não é obrigatória para o funcionamento da classe.
  - **Configuração flexível:** Quando a configuração da dependência pode variar em diferentes ambientes.
- **Construtor:**
  - **Obrigatória:** Quando a dependência é essencial para o funcionamento da classe.
  - **Clareza:** Quando se deseja tornar explícitas as dependências da classe.
  - **Testes:** Para facilitar a criação de testes unitários.

Recomendação:

Recomenda-se utilizar a injeção por construtor como padrão, reservando a `@Autowired` para casos específicos em que a flexibilidade seja essencial. Ao seguir essa prática, você estará construindo aplicações mais robustas, testáveis e fáceis de manter.

Docker Desktop

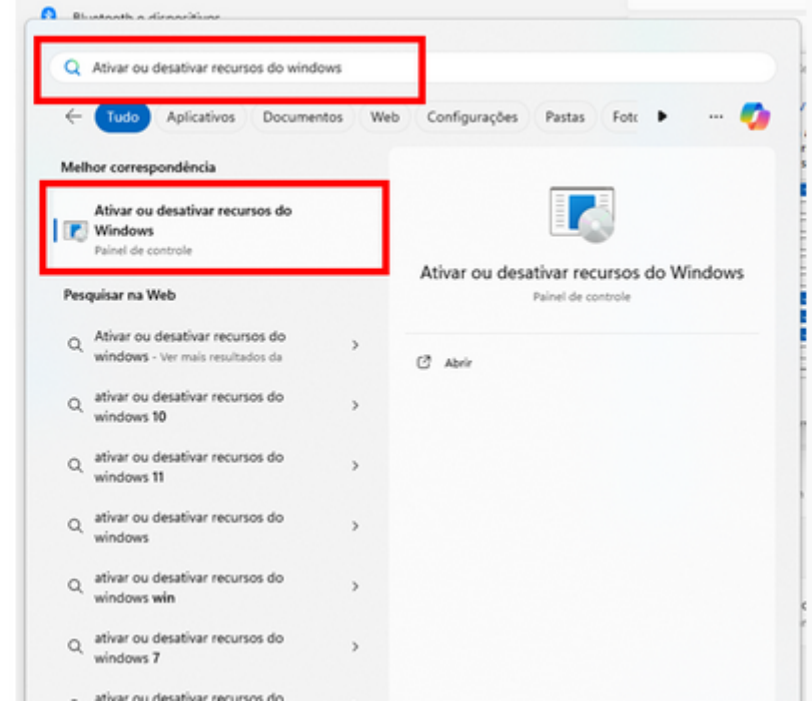
Docker Desktop é um aplicativo de instalação com um clique para seu ambiente Mac, Linux ou Windows que permite criar, compartilhar e executar aplicativos e microsserviços em contêineres. Ele fornece uma GUI (interface gráfica do usuário) simples que permite gerenciar seus contêineres, aplicativos e imagens diretamente de sua máquina.

ATENÇÃO: Antes de instalar o Docker Desktop, é necessário habilitar o Subsistema do Windows para Linux (WSL – Windows Subsystem for Linux)

Site: https://www.docker.com/
Link para download: https://www.docker.com/products/docker-desktop/

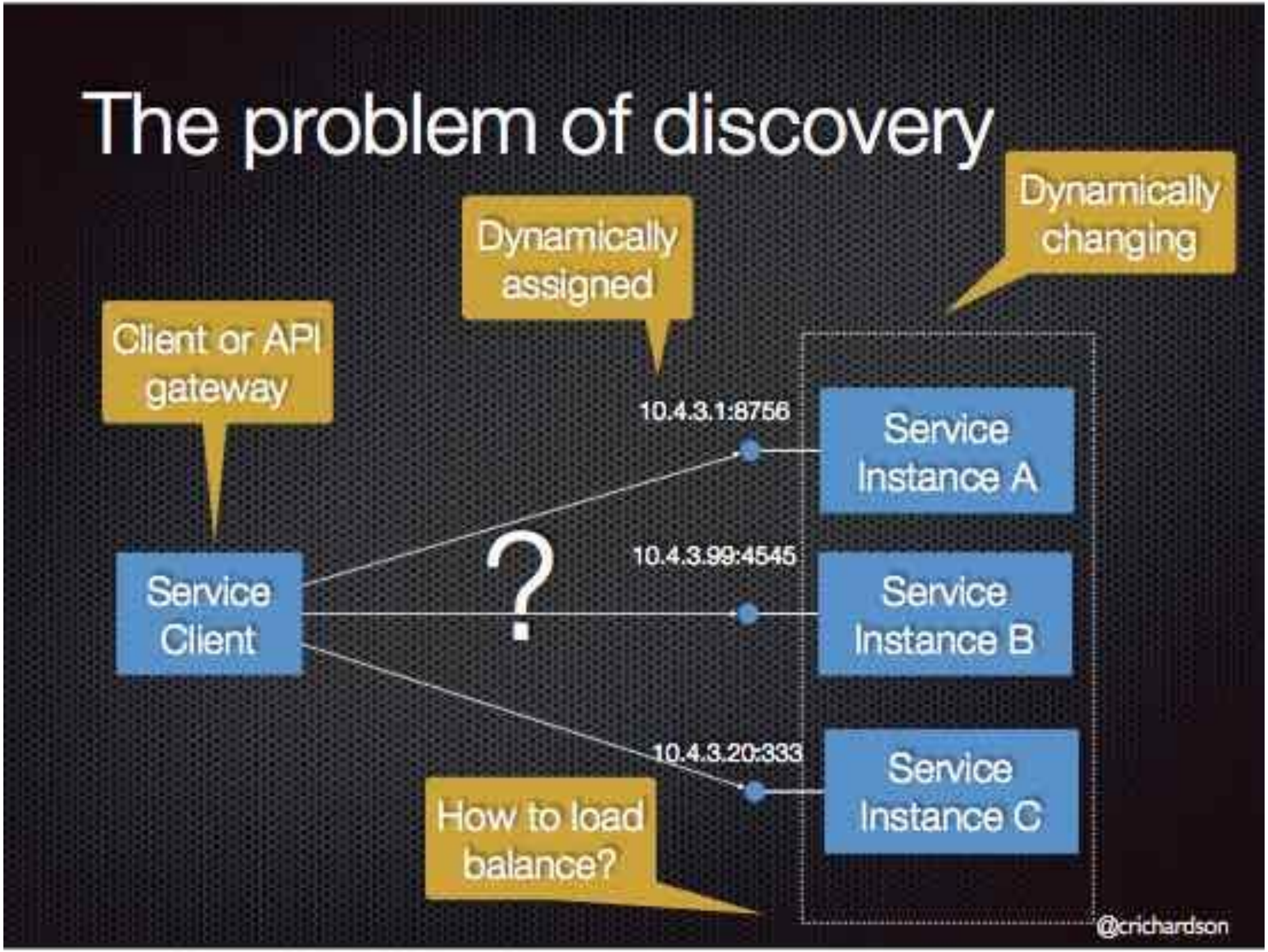
Habilitando o WSL:

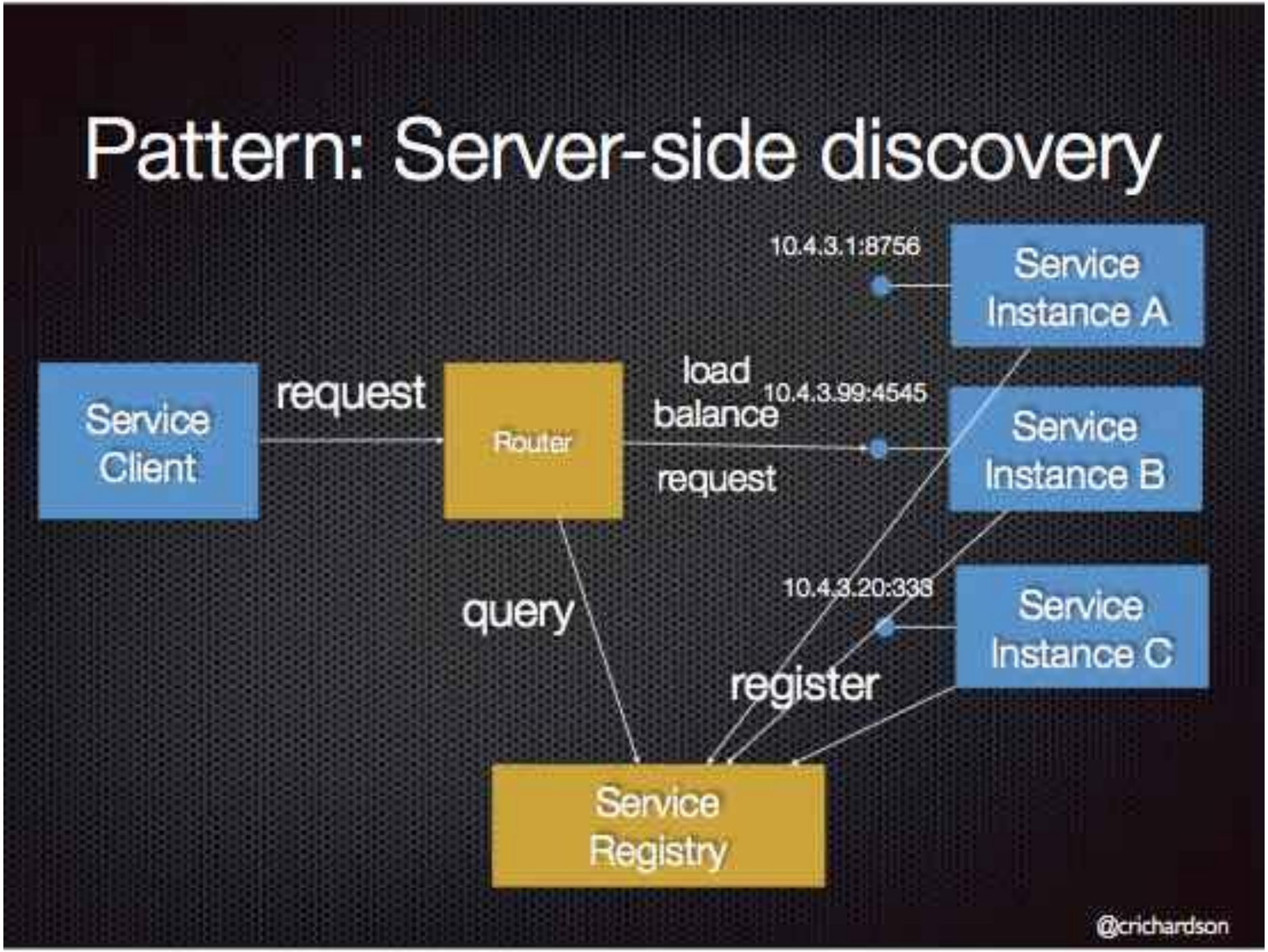
Através da Barra de pesquisa no Menu Iniciar, procure por

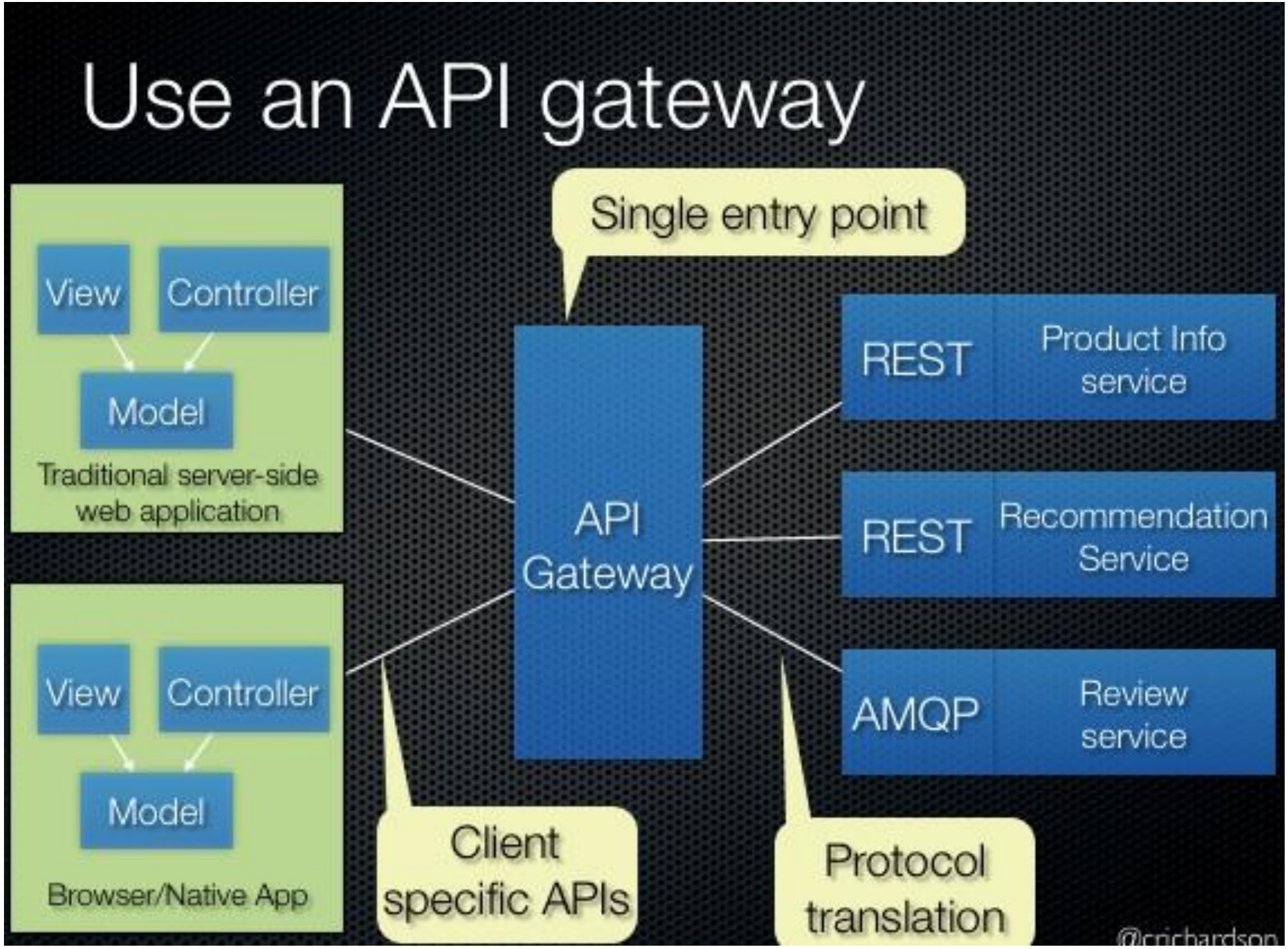


API Gateway

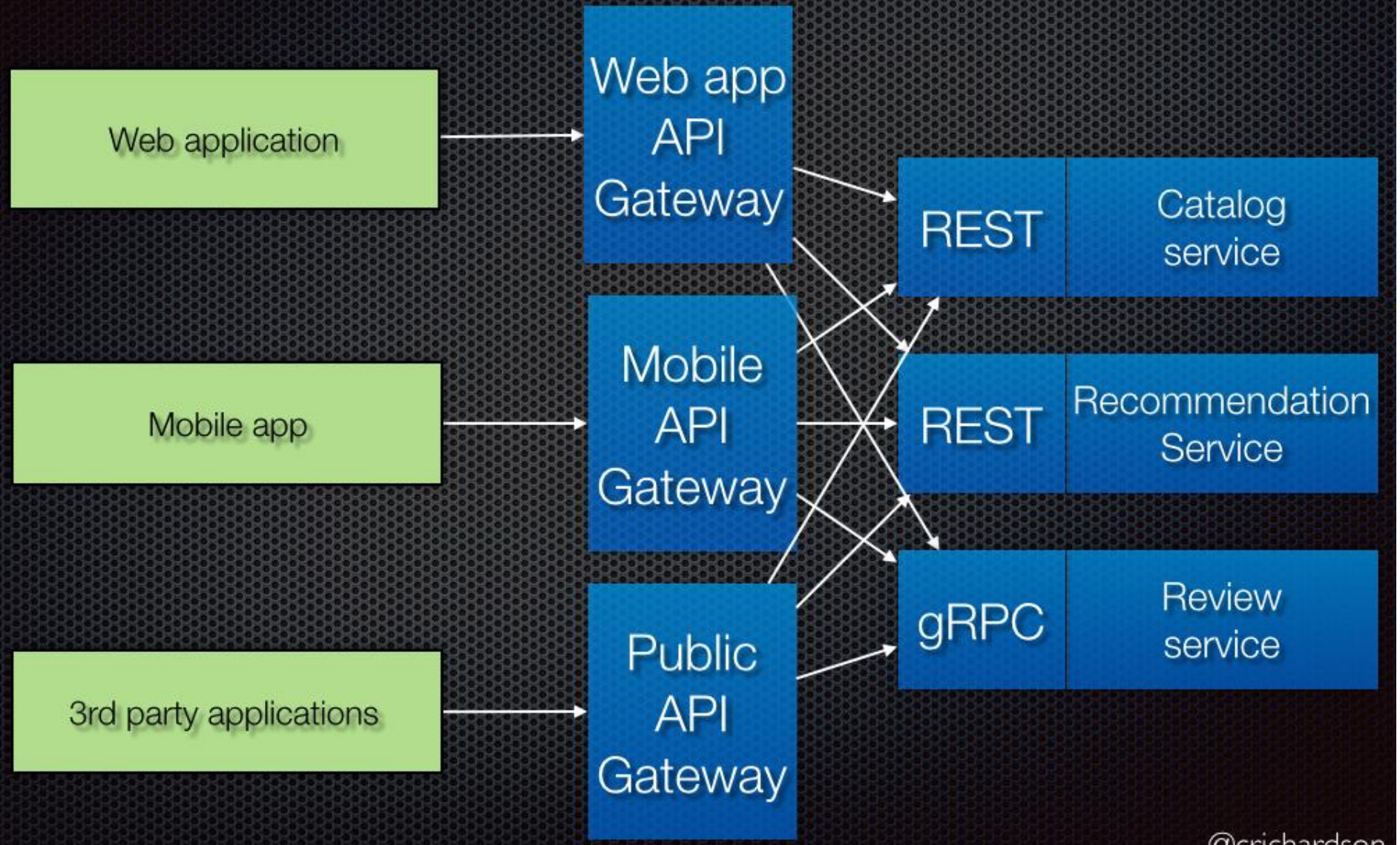
API Gateway Pattern
Server-side Discovery Pattern

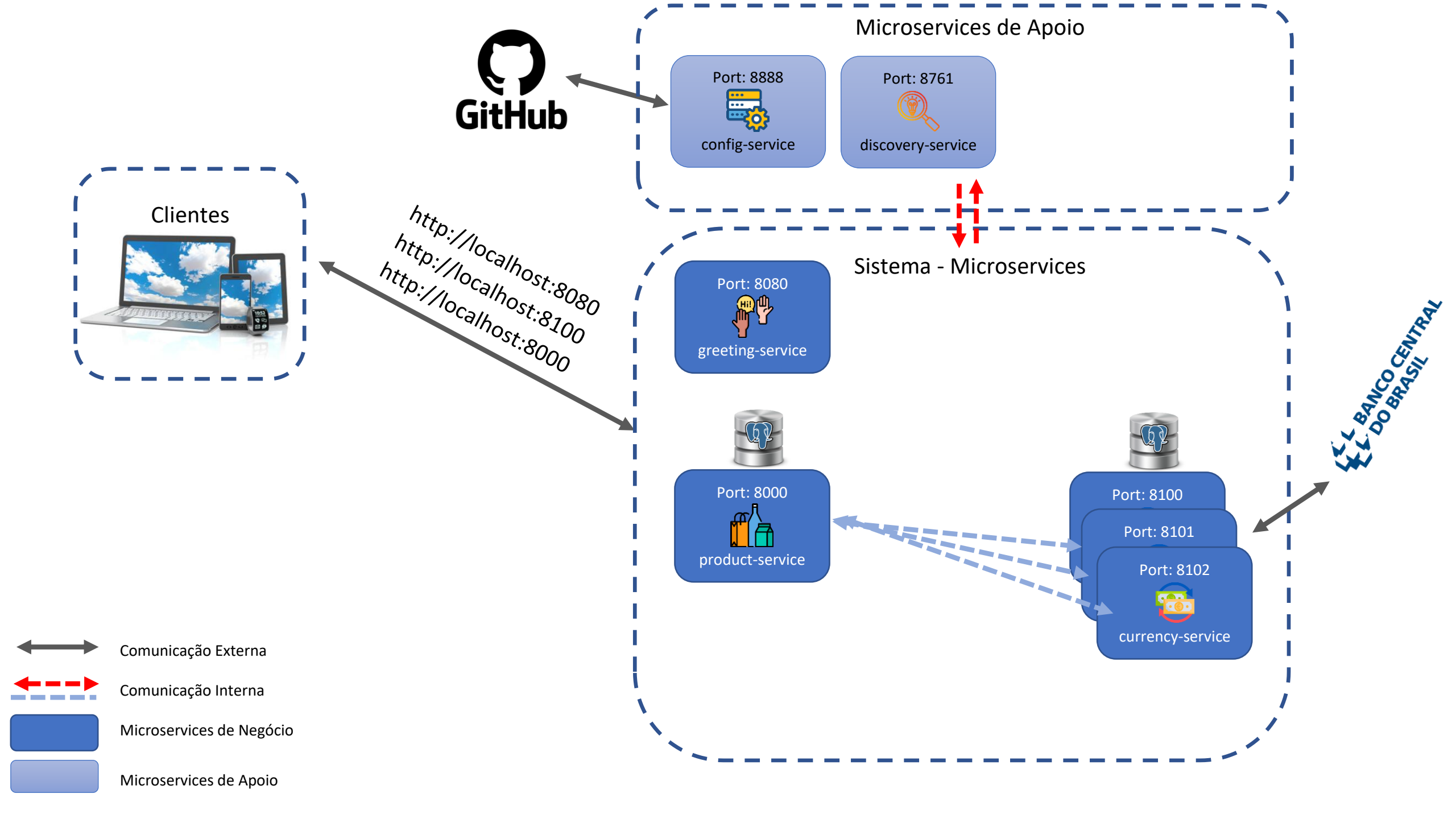

The problem of discovery
Dynamically changing
Dynamically assigned
Client or API gateway
10.4.3.1:8756
Service Instance A
10.4.3.99:4545
Service Client
?
Service Instance B
10.4.3.20:333
Service Instance C
How to load balance?
@crichardson

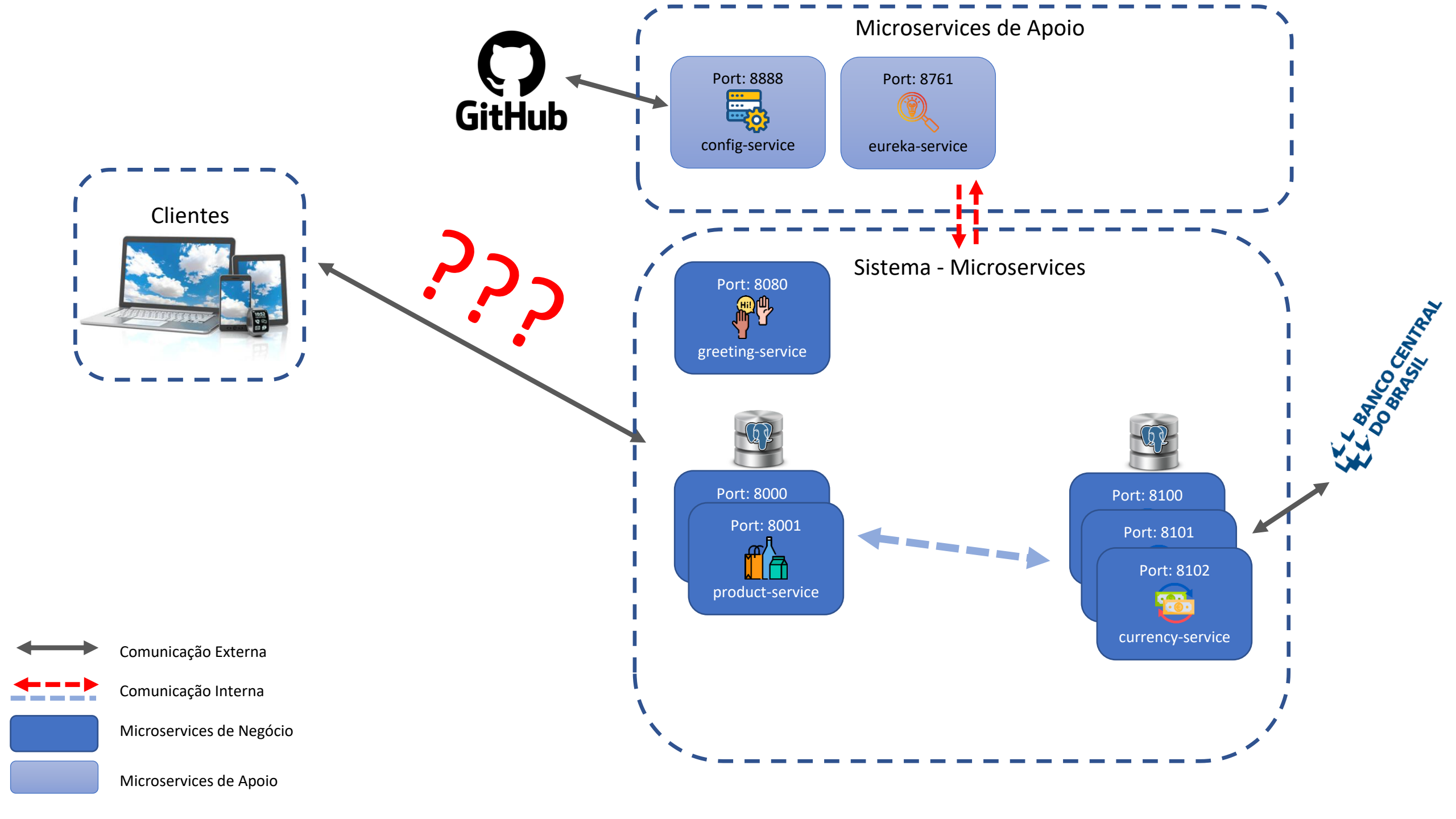

Pattern: Server-side discovery
10.4.3.1:8756
Service Instance A
Service Client
request
Router
load balance
10.4.3.99:4545
request
Service Instance B
10.4.3.20:338
Service Instance C
query
register
Service Registry
@crichardson

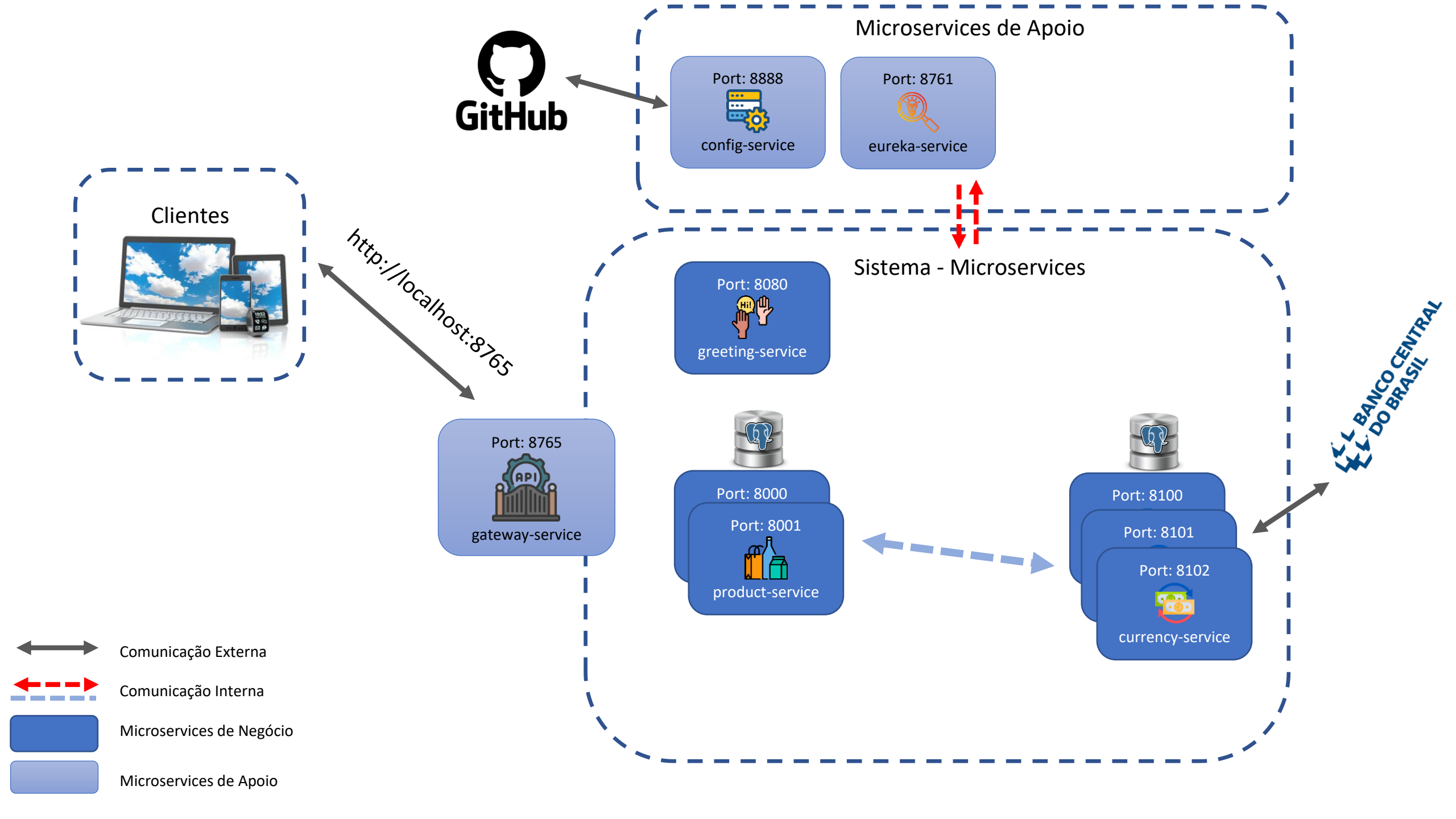

Use an API gateway
Single entry point
View
Controller
Model
Traditional server-side web application
View
Controller
Model
Browser/Native App
API Gateway
REST
Product Info service
REST
Recommendation Service
AMQP
Review service
Client specific APIs
Protocol translation
@crichardson

Variation: Backends for frontends




Microservices de Apoio
GitHub
Port: 8888
config-service
Port: 8761
discovery-service
Clientes
http://localhost:8080
http://localhost:8100
http://localhost:8000
Sistema - Microservices
Port: 8080
greeting-service
Port: 8000
product-service
Port: 8100
Port: 8101
Port: 8102
currency-service
BANCO CENTRAL DO BRASIL
Comunicação Externa
Comunicação Interna
Microservices de Negócio
Microservices de Apoio


Microservices de Apoio
Port: 8888
config-service
Port: 8761
eureka-service
GitHub
Clientes
???
Sistema - Microservices
Port: 8080
greeting-service
Port: 8000
Port: 8001
product-service
Port: 8100
Port: 8101
Port: 8102
currency-service
BANCO CENTRAL DO BRASIL
Comunicação Externa
Comunicação Interna
Microservices de Negócio
Microservices de Apoio


Microservices de Apoio
GitHub
Port: 8888
config-service
Port: 8761
eureka-service
Clientes
http://localhost:8765
Sistema - Microservices
Port: 8080
greeting-service
Port: 8765
gateway-service
Port: 8000
Port: 8001
product-service
Port: 8100
Port: 8101
Port: 8102
currency-service
BANCO CENTRAL DO BRASIL
Comunicação Externa
Comunicação Interna
Microservices de Negócio
Microservices de Apoio

Agora... Vamos codar!!!!

Ciência da
Computação