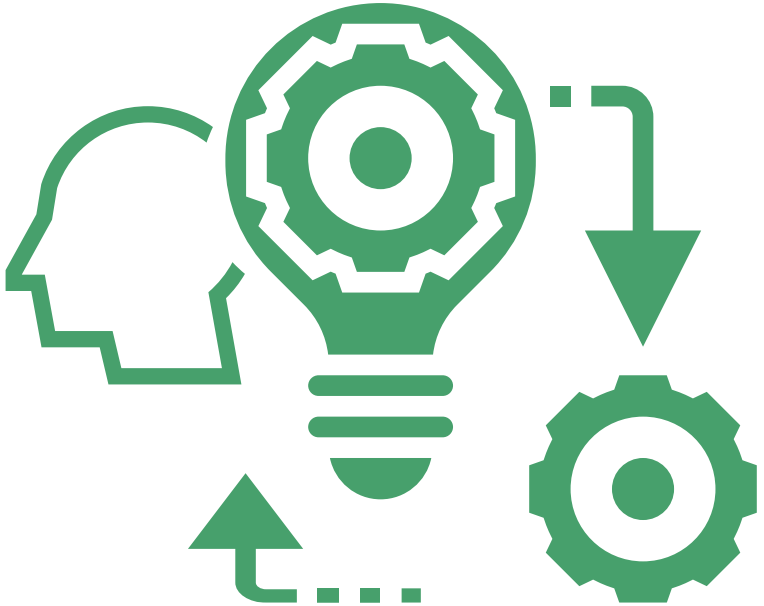


Paradigmas de Programação

Aula 05 – Como vai ser ???

- Entregáveis Aula 04
- Microservices
- Configurações em Microservices

Princípio de Substituição de Liskov

- **Significa dizer que classes derivadas devem poder substituídas por suas classes base e que classes base podem ser substituídas por qualquer uma das suas subclasses.**

- Uma subclasse deve sobrescrever os métodos da superclasse de forma que a funcionalidade do ponto de vista do cliente continue a mesma.

```java
abstract class FormaGeometrica {
    double lado;
    abstract double calculaArea();
    abstract double calculaVolume();
}
```

```java
class Cubo extends FormaGeometrica{
    @Override
    double calculaArea() {
        return 6 * (lado * lado);
    }
    @Override
    double calculaVolume() {
        return 3 * lado;
    }
}
```

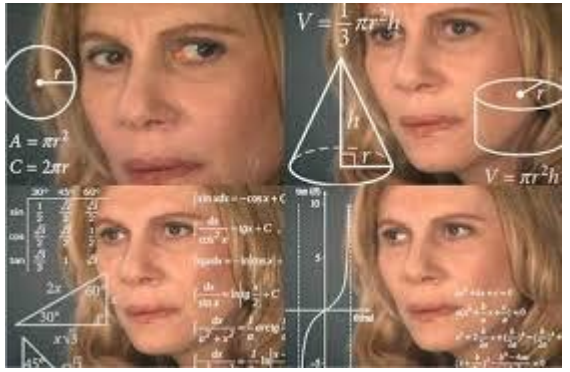
```java
class Retangulo extends FormaGeometrica {
    @Override
    double calculaArea() {
        return lado * lado;
    }
    @Override
    double calculaVolume() {
        throw new UnsupportedOperationException
            ("Unsupported method 'calculaVolume'");
    }
}
```

```java
public class LSP {
    public static void main(String[] args) {
        Cubo cubo = new Cubo(2);
        imprimeArea(cubo);
        imprimeVolume(cubo);
        Quadrado ret = new Quadrado(3);
        imprimeArea(ret);
        imprimeVolume(ret);

    }
    static void imprimeArea(FormaGeometrica forma) {
        System.out.println(forma.calculaArea());
    }
    static void imprimeVolume(FormaGeometrica forma) {
        System.out.println(forma.calculaVolume());
    }
}
```

```
24.0
6.0
9.0
Exception in thread "main" java.lang.UnsupportedOperationException: Unsupported method 'calculaVolume'
        at Quadrado.calculaVolume(LSP.java:53)
        at LSP.imprimeVolume(LSP.java:15)
        at LSP.main(LSP.java:8)
```

Mas porque esse código viola o LSP??



- Esse comportamento viola o LSP porque:
- Ao tentar substituir *FormaGeometrica* por uma instância de *Quadrado*, o método *imprimeVolume()* quebra o programa ao lançar uma exceção.
- Isso sugere que a interface da classe base (FormaGeometrica) define um comportamento que nem todas as subclasses podem cumprir.

E como podemos corrigir???

- Uma solução seria separar as classes em duas hierarquias diferentes, por exemplo, criando uma classe base *FormaBidimensional* para formas como *Quadrado* que não possuem volume
- E uma classe *FormaTridimensional* para formas como *Cubo*, que têm volume.
- Isso garantiria que cada subclasse respeite completamente os métodos definidos pela sua classe base.

```
abstract class FormaBidimensional extends FormaGeometrica {
}
abstract class FormaTridimensional extends FormaGeometrica {
    abstract double calculaVolume();
}
```

```
class Cubo extends FormaTridimensional {
```

```
class Quadrado extends FormaBidimensional {
```

```
    public static void main(String[] args) {
        Cubo cubo = new Cubo(2);
        imprimeArea(cubo);
        imprimeVolume(cubo);
        Quadrado ret = new Quadrado(3);
        imprimeArea(ret);
        imprimeVolume(ret);
    }
    static void imprimeArea(FormaGeometrica forma) {
        System.out.println(forma.calculaArea());
    }
    static void imprimeVolume(FormaTridimensional forma) {
        System.out.println(forma.calculaVolume());
    }
```

Princípio de Segregação de Interface

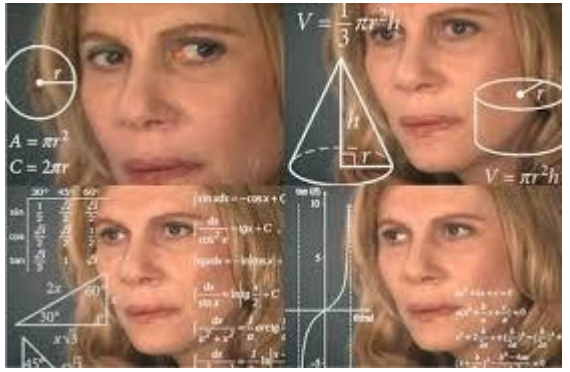
- **Este princípio estabelece que uma classe classe não deve ser forçada a implementar métodos que não usa.**
- De acordo com o Princípio da Segregação da Interface, uma interface deve ser coesa e ter apenas o mínimo necessário para seus clientes.
- As interfaces devem ser segregadas de forma a cada cliente depender apenas dos métodos que precisa utilizar, evitando assim a dependência de funcionalidades desnecessárias.

```java
interface Ave {
    void comer();
    void voar();
}
```

```java
class Canarinho implements Ave {
    @Override
    public void comer() {
        System.out.println("Canarinho comendo");
    }
    @Override
    public void voar() {
        System.out.println("Canarinho voando");
    }
}
```

```java
class Pinguim implements Ave {
    @Override
    public void comer() {
        System.out.println("Pinguim comendo");
    }
    @Override
    public void voar() {
        throw new UnsupportedOperationException("Pinguim não voa");
    }
}
```

Mas porque esse código viola o ISP??



- A interface *Ave* é muito ampla, pois impõe que todas as aves, independentemente de suas habilidades, implementem o método *voar()*.

- Isso viola o ISP, já que *Pinguim* é uma ave que não deveria ser obrigada a implementar algo que não é relevante para ela.

E como podemos corrigir???

- Uma solução seria dividir a interface *Ave* em interfaces mais específicas, como *AveVoadora* e *AveComedora*. Dessa forma, apenas as aves que realmente voam (como *Canarinho*) implementariam a interface *AveVoadora*, enquanto o Pinguim apenas implementaria a interface que contém o comportamento de *comer()*.

```java
interface AveVoadora {
    void voar();
}
interface AveComedora {
    void comer();
}
```

```java
class Canarinho implements AveVoadora, AveComedora {
    @Override
    public void comer() {
        System.out.println("Canarinho comendo");
    }
    @Override
    public void voar() {
        System.out.println("Canarinho voando");
    }
}
```

```java
class Pinguim implements AveComedora {
    @Override
    public void comer() {
        System.out.println("Pinguim comendo");
    }
}
```

```java
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Lista imutável usando List.of()
        List<String> listaImutavel = List.of("A", "B", "C");

        // Lista de tamanho fixo usando Arrays.asList()
        List<String> listaTamanhoFixo = Arrays.asList("X", "Y", "Z");

        // Testando operações de adição e remoção
        listaImutavel.add("D"); // O que acontece aqui?
        listaTamanhoFixo.remove("X"); // E aqui?
    }
}
```

```
Exception in thread "main" java.lang.UnsupportedOperationException
        at java.base/java.util.ImmutableCollections.uoe(ImmutableCollections.java:142)
        at java.base/java.util.ImmutableCollections$AbstractImmutableCollection.add(ImmutableCollections.java:147)
        at Main.main(Main.java:13)
```

```
Exception in thread "main" java.lang.UnsupportedOperationException: remove
        at java.base/java.util.Iterator.remove(Iterator.java:102)
        at java.base/java.util.AbstractCollection.remove(AbstractCollection.java:285)
        at Main.main(Main.java:14)
```

Entregáveis Aula 04 – Questão 01

- Explique se o uso dessas implementações da interface *List* pode ferir o Princípio de Substituição de Liskov (LSP).

- Avalie se a Interface Segregation Principle (ISP) está sendo violado ao utilizar essas listas, considerando as operações permitidas e não permitidas nas mesmas.

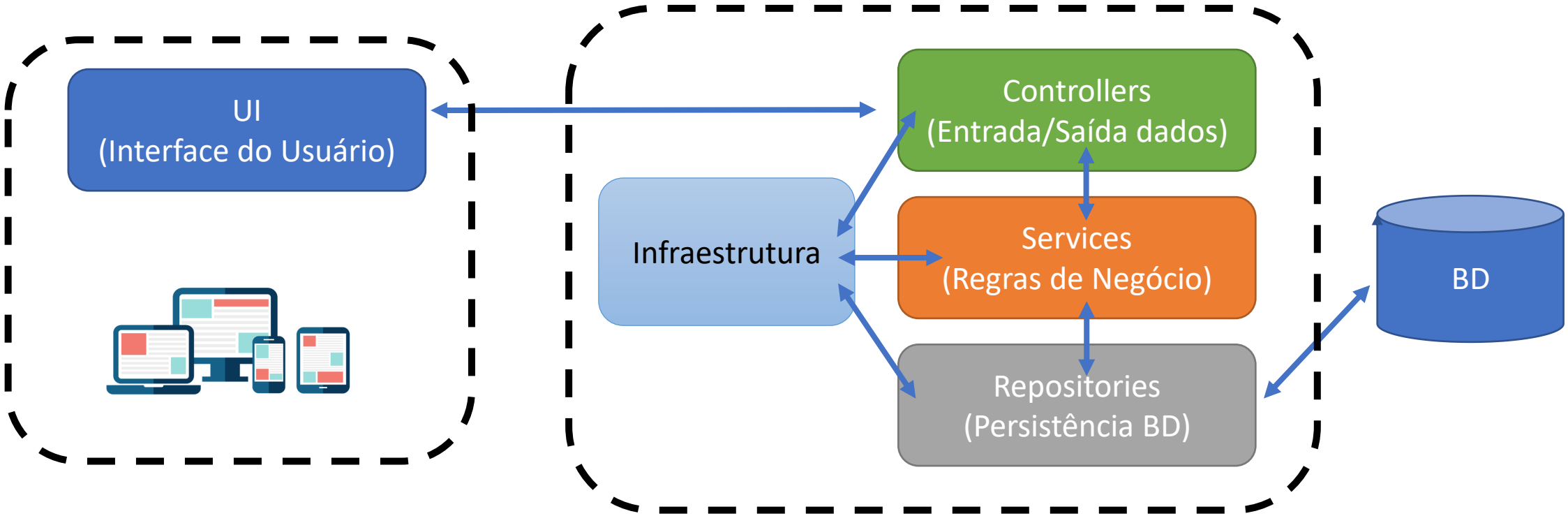
Entregáveis Aula 04 – Questão 02

No Spring Boot, a ordem de prioridade das fontes de configuração, da mais alta para a mais baixa, é a seguinte:

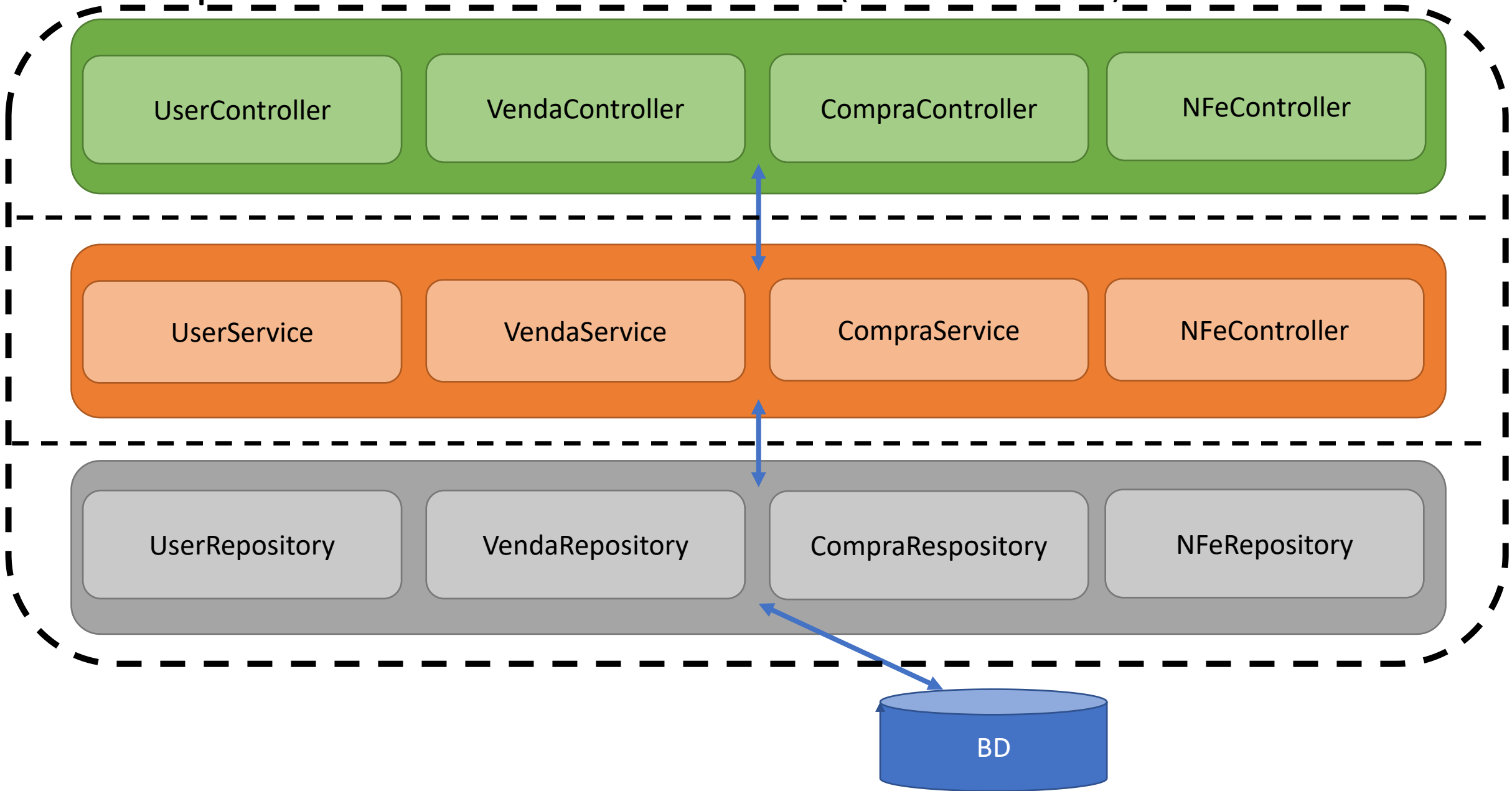
1. **Argumentos da Linha de Comando**: Configurações passadas como argumentos ao iniciar a aplicação, por exemplo, `--my.custom.property=value`. Esses argumentos têm a maior prioridade e sobrescrevem todos os outros métodos de configuração.
2. **Propriedades do Sistema**: Definidas com `System.setProperty("my.custom.property", "value")`, essas propriedades têm uma alta prioridade e podem ser usadas para configurar valores diretamente no ambiente do sistema.
3. **Variáveis de Ambiente**: Configurações provenientes de variáveis de ambiente, como `MY_CUSTOM_PROPERTY=value`. Essas variáveis têm uma prioridade significativa.
4. **Servidor de Configuração (Spring Cloud Config Server)**: Quando configurado, o Spring Cloud Config Server fornece configurações centralizadas e externas. Essas configurações geralmente têm prioridade sobre arquivos de configuração no classpath e em `/config`, mas são substituídas por configurações definidas nas fontes de maior prioridade listadas acima.
5. **Arquivos de Configuração Externos**: Arquivos de configuração que estão fora do classpath, como `/config/application.properties` ou `/config/application.yml`. Esses arquivos têm prioridade sobre os arquivos no classpath.
6. **Arquivos de Configuração no Classpath**: Arquivos como `application.properties` e `application.yml` localizados no classpath da aplicação. Eles são uma fonte padrão de configuração.

Microservices (Microserviços)

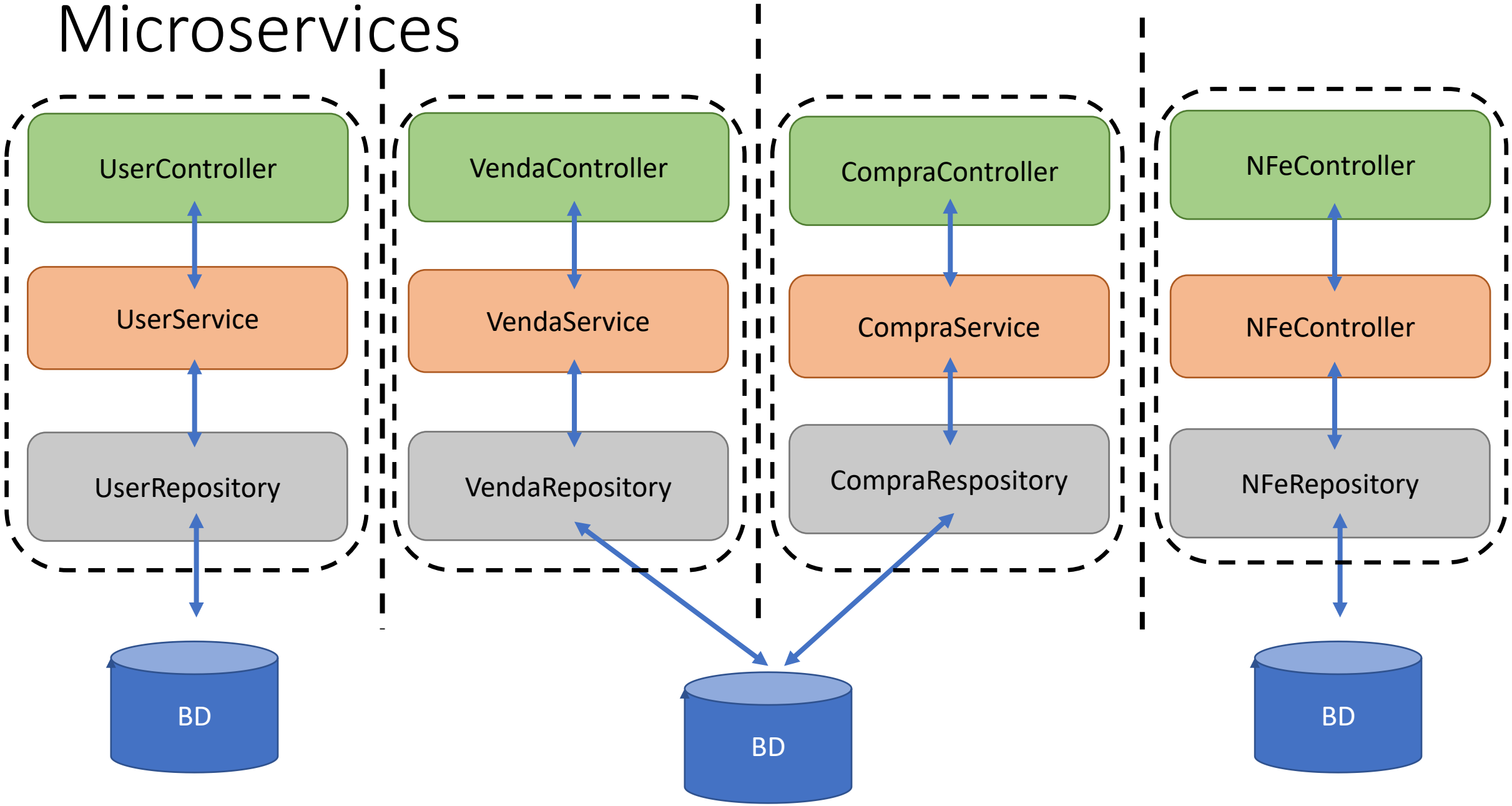
Monolítico (Cliente/Servidor e Camadas)



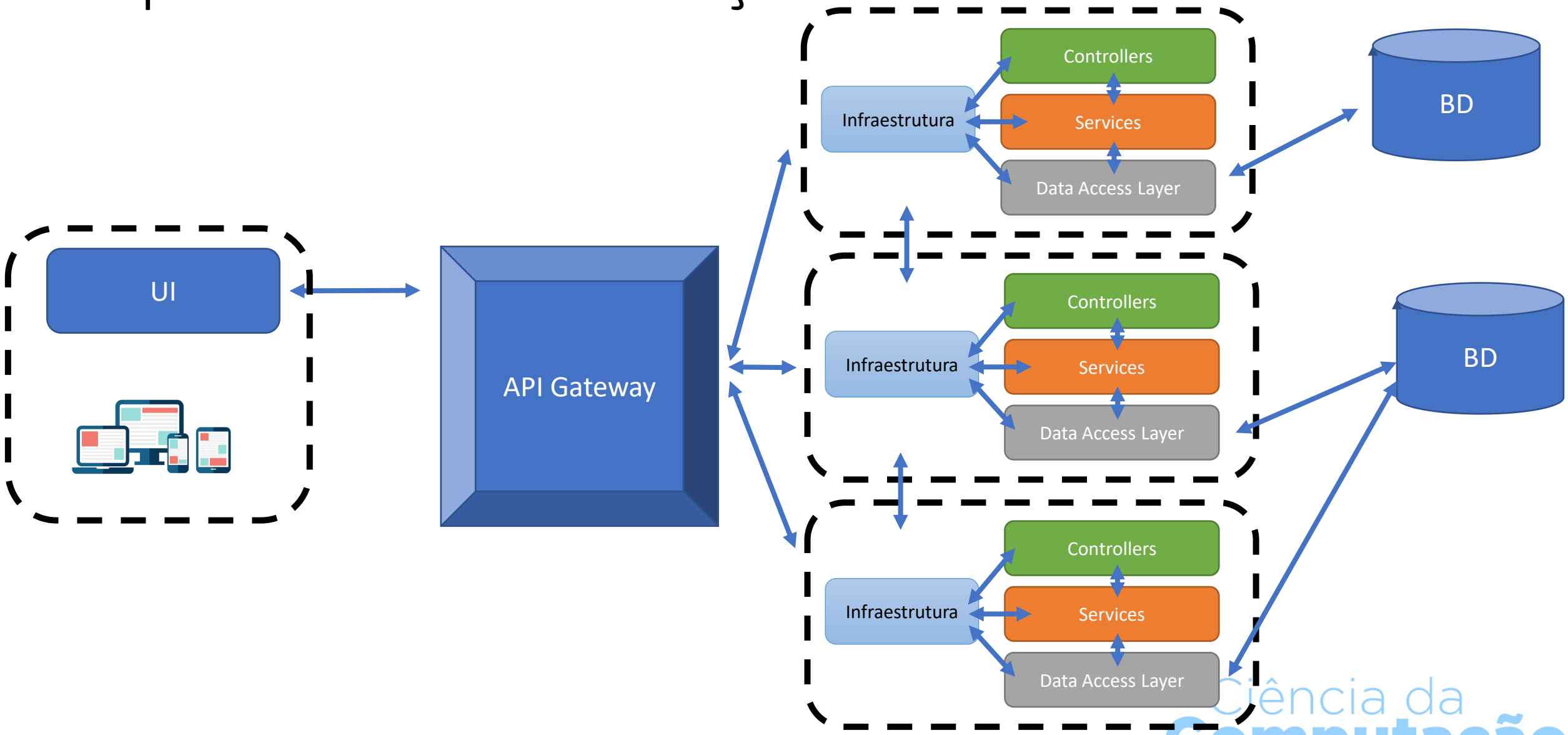
Arquitetura de Camadas (Monolítico)



Microservices



Arquitetura MicroServiços

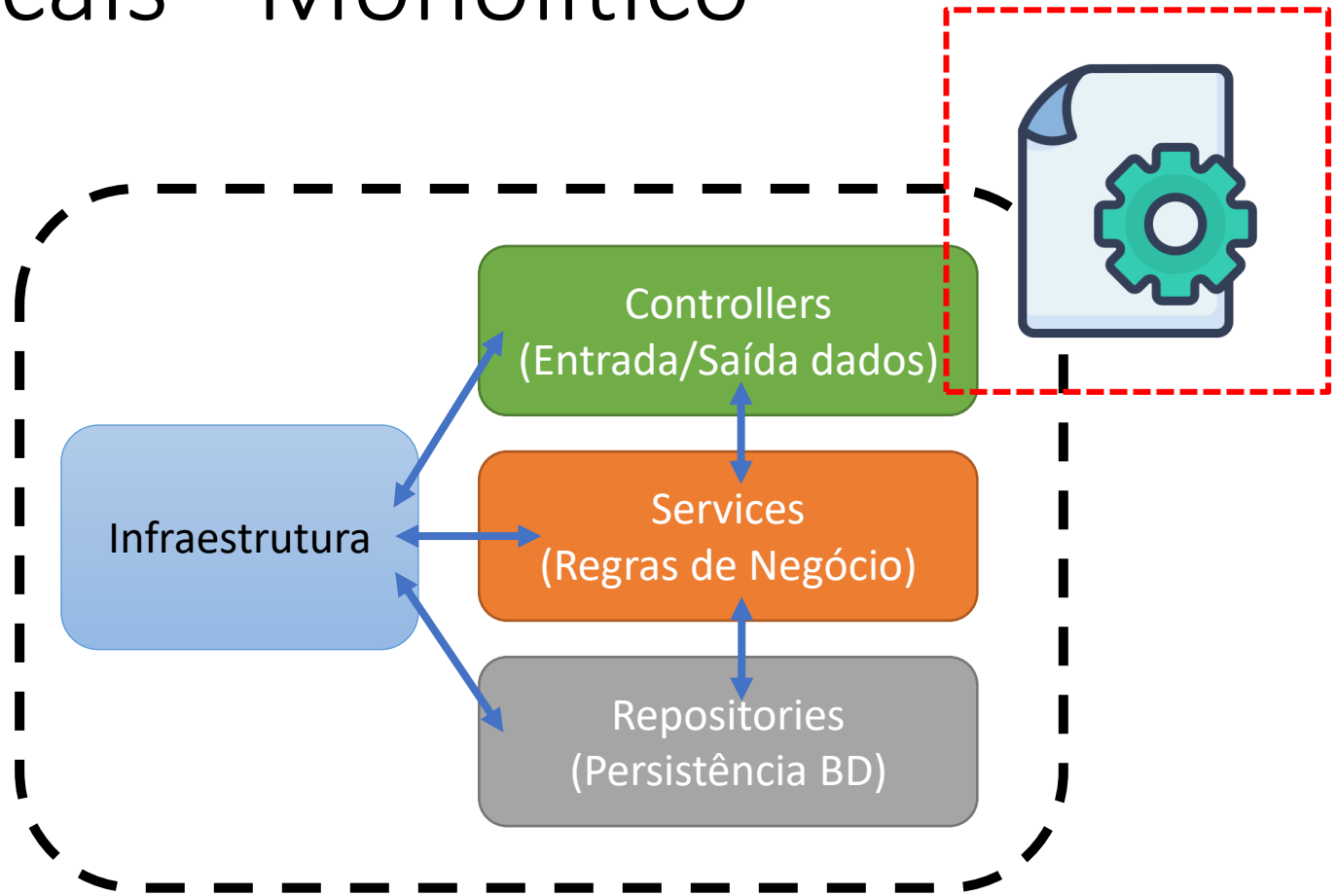


Configurações em Microservices

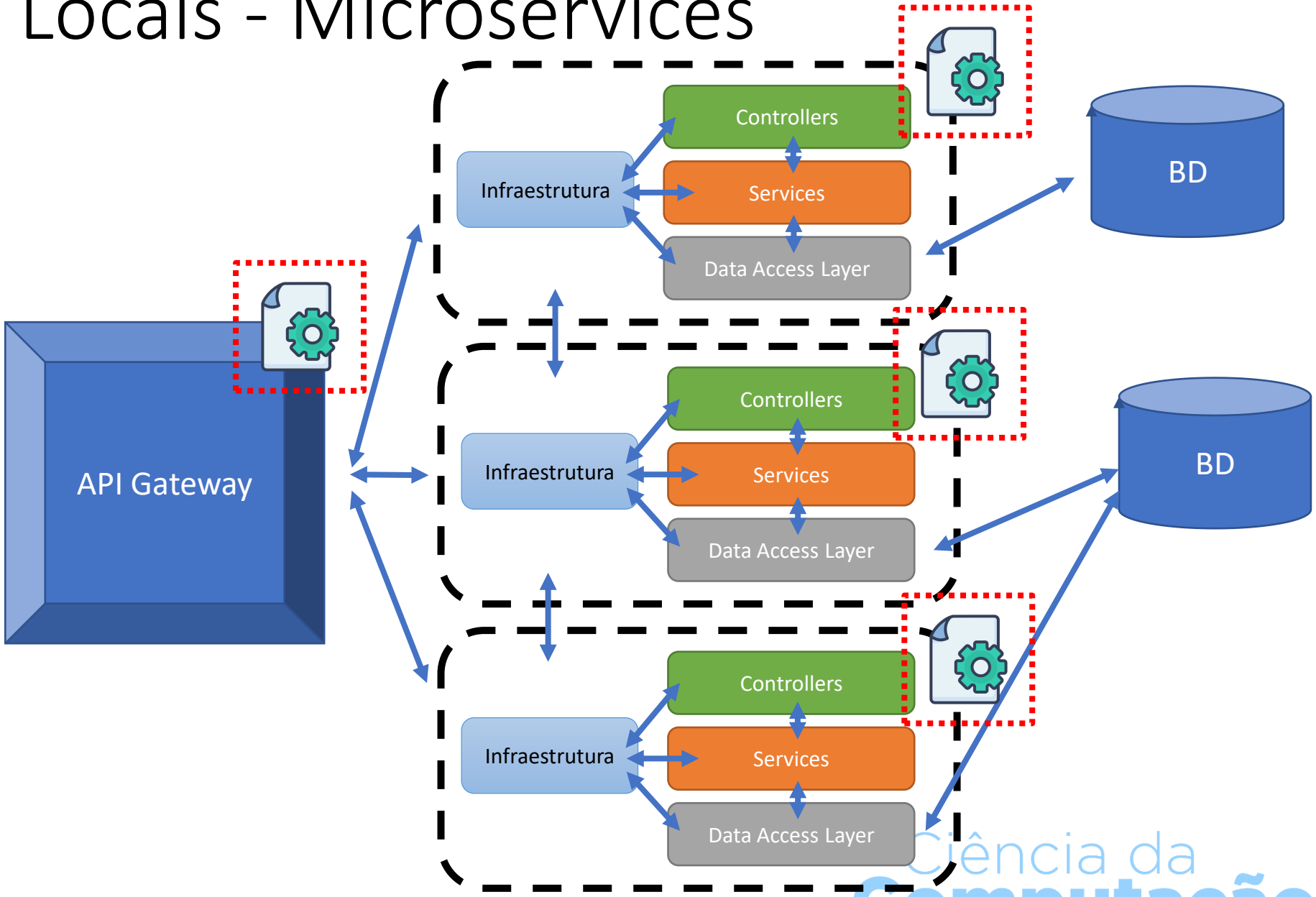
Configuração

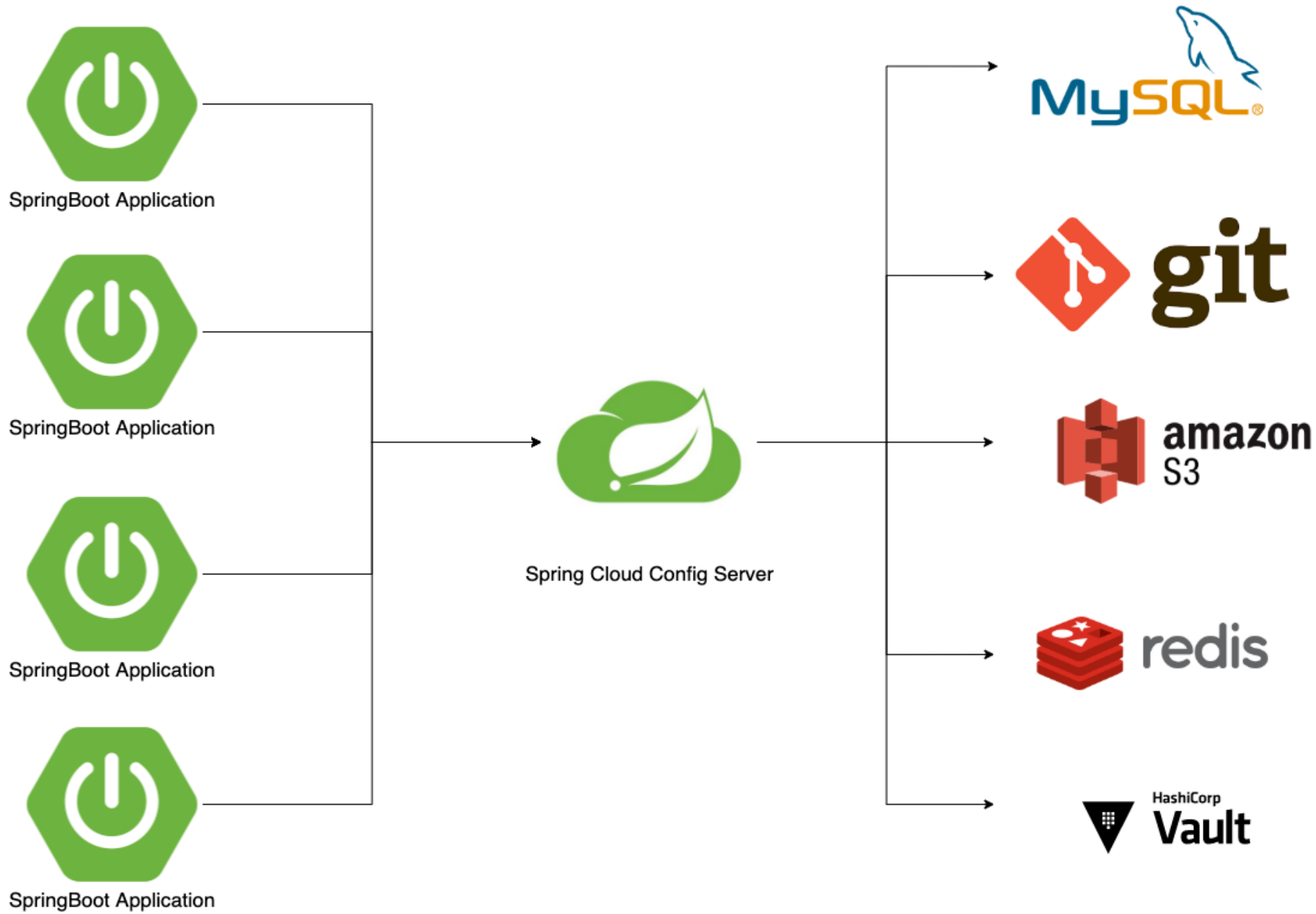
- Configurações em aplicações são essenciais para definir aspectos como URLs de banco de dados, credenciais, portas e outros detalhes que podem variar entre ambientes (desenvolvimento, teste, produção).
- Em uma aplicação monolítica, centralizar essas configurações em arquivos como `application.properties` ou `application.yml` é relativamente simples.
- Porém, em um ambiente de **Microservices**, essa questão se torna muito mais complexa. Com dezenas ou centenas de microservices rodando em diferentes ambientes, a gestão das configurações precisa ser eficiente e centralizada.

Configurações Locais - Monolítico



Configurações Locais - Microservices




SpringBoot Application
SpringBoot Application
SpringBoot Application
SpringBoot Application
Spring Cloud Config Server
MySQL
git
amazon
S3
redis
HashiCorp
Vault

Ciência da
Computação