Arquitetura de microsserviços

Qual a arquitetura de software seu *microservices* se encaixa?
Ou melhor, seu *microservices* necessita de uma arquitetura?

Abraão Honório - Engenheiro de software sênior da XP Inc.

@AbraaoHonorio

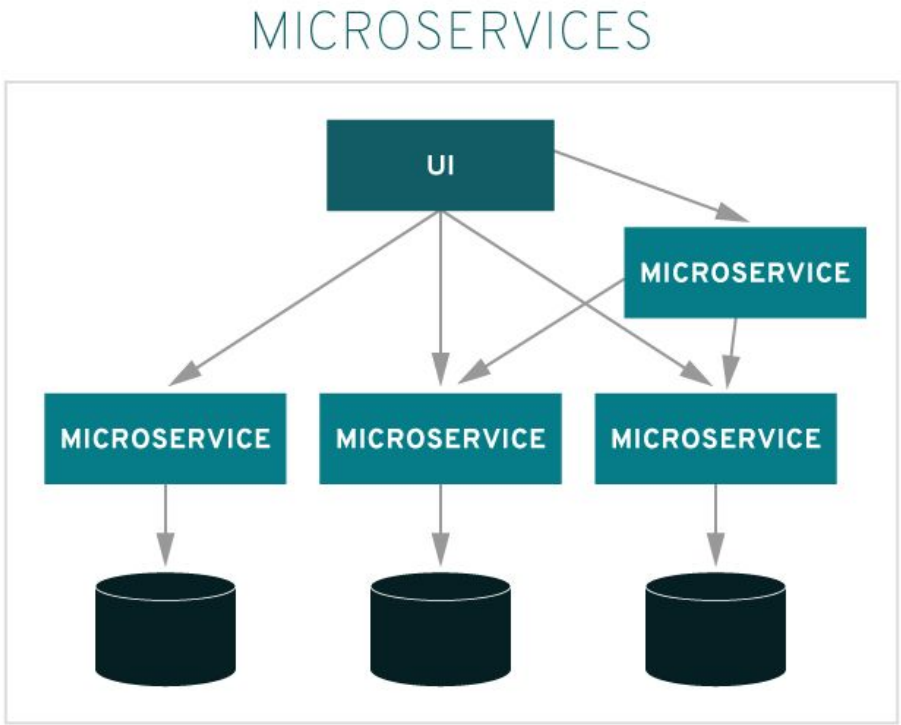
@abraaohonorio

Que sou eu?

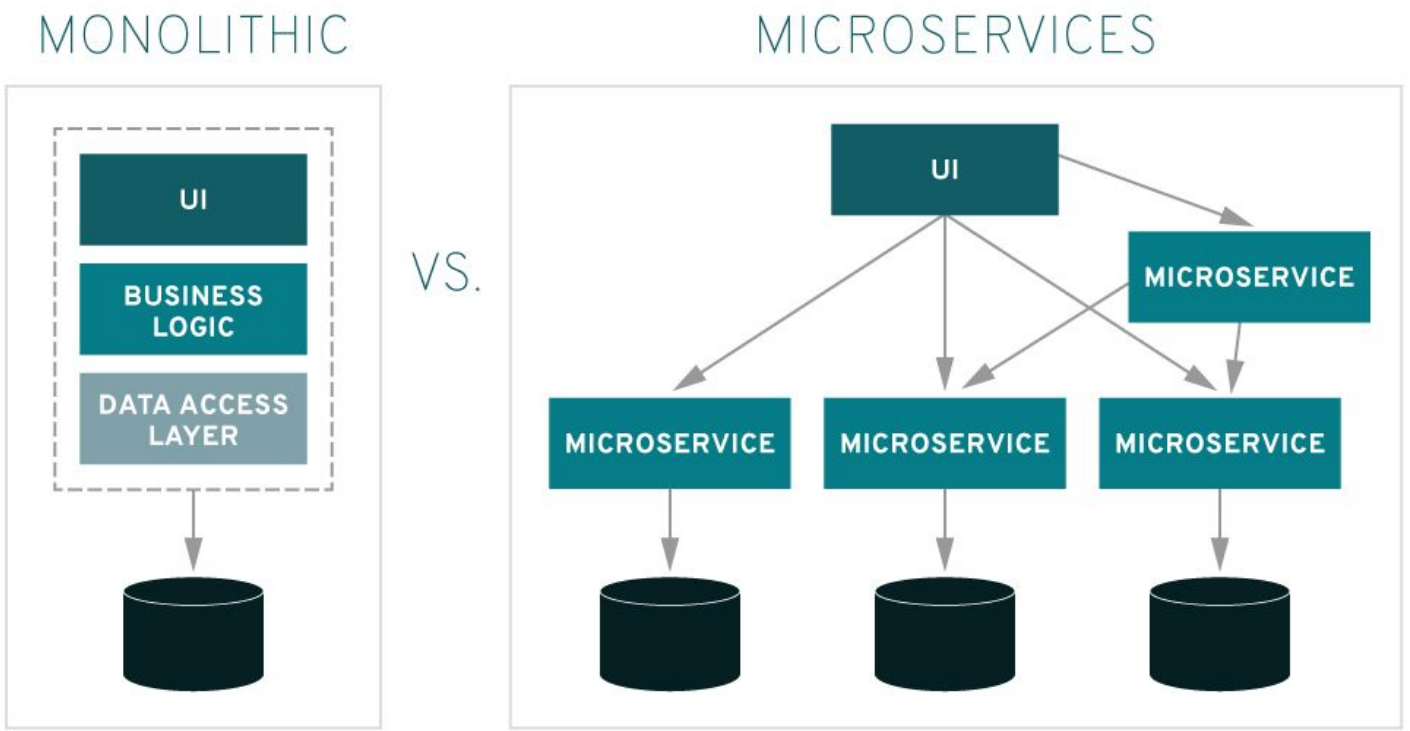
Abraão Honório

Engenheiro de Software com foco em backend, líder de comunidade(.Net_PB), contribuidor de código aberto, entusiasta do trabalho Home Office e formado em Engenharia de Computação pela Universidade Federal da Paraíba (UFPB). Atualmente trabalho como Engenheiro de software sênior na squad de arquitetura de software na XP Inc.

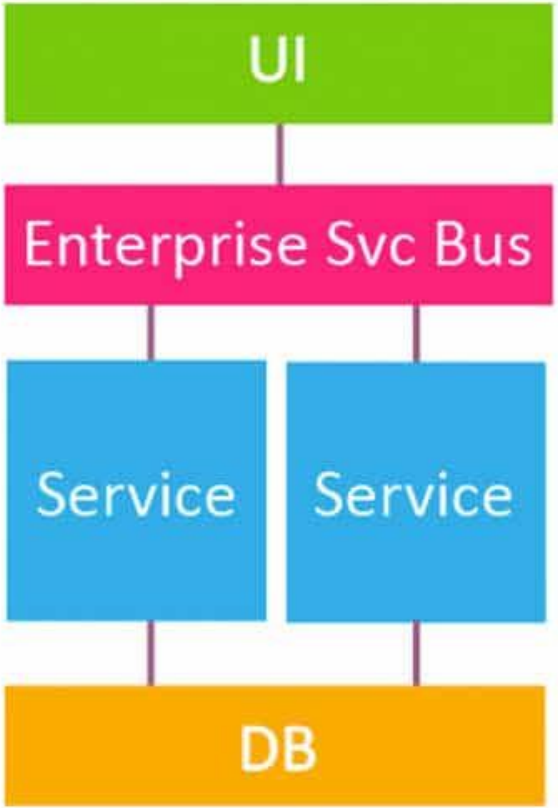
O que é microsserviços?

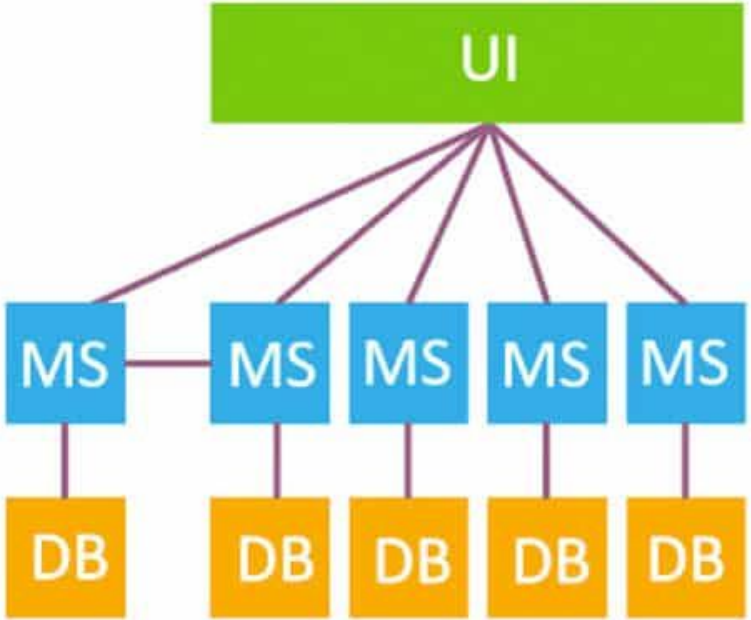


Monolitico x microsserviços

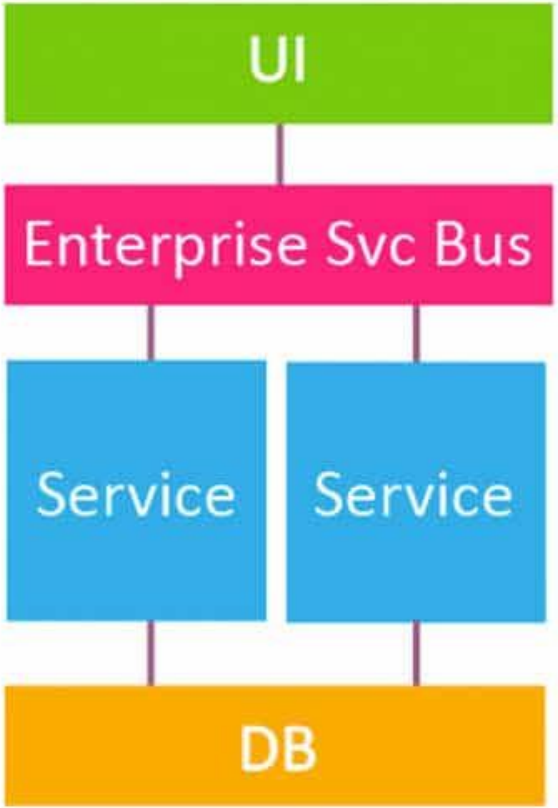


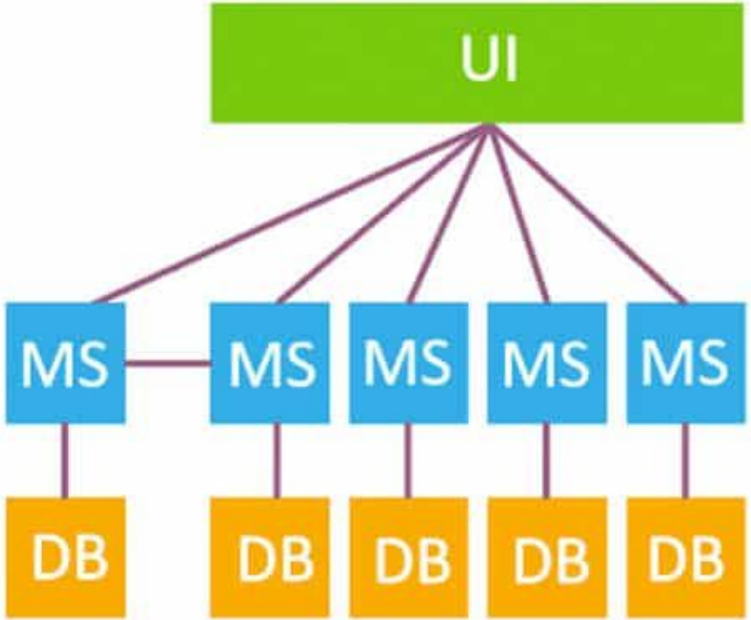
SOA x MICROSSERVIÇOS



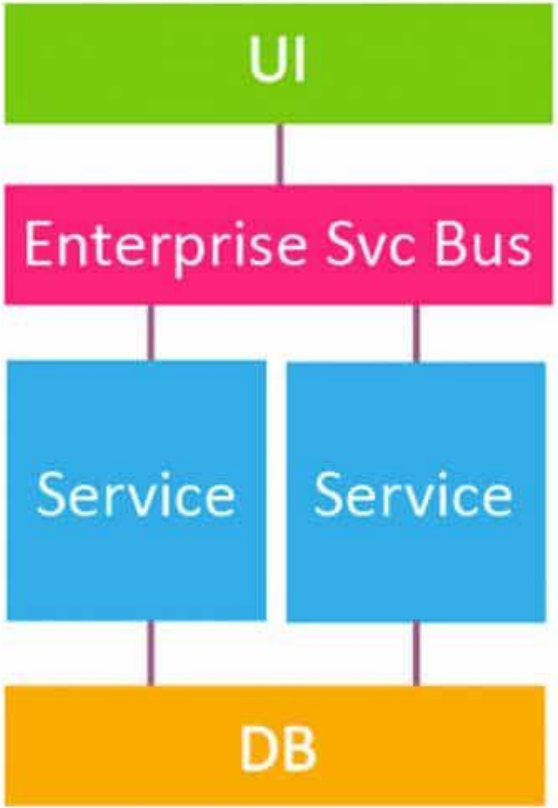


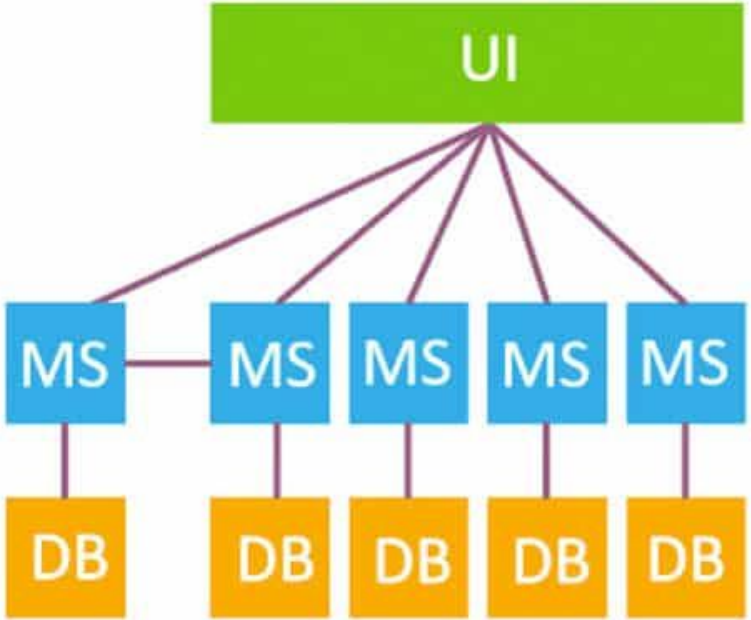
Como os microsserviços e SOA são semelhantes?





Diferenças entre microsserviços e SOA





MICROSSERVIÇOS X SOA

	SOA	Microsserviço
Granularidade	Serviços granulados de curso	Serviços refinados
Facilidade de implantação	Requer a recriação e reimplantação de todo o aplicativo	Cada serviço pode ser construído e implantado de forma independente
Chamada Remota Overhead	Baixa sobrecarga de comunicação	Alta sobrecarga de comunicação devido a um aumento nas chamadas remotas
Velocidade de implantação	Velocidades de implantação lentas	Implantação rápida, contínua e automatizada
Persistência	Todos os serviços em SOA compartilham armazenamento de dados	Cada serviço é livre para escolher seu próprio armazenamento de dados
Facilidade de integração	Semidifícil para integrar novos desenvolvedores, pois o escopo de todo o aplicativo pode precisar ser compreendido	Fácil de integrar novos desenvolvedores, pois não há necessidade de entender o escopo de todo o aplicativo
Programação Poliglota	Pode utilizar diferentes linguagens de programação para cada serviço	Pode utilizar diferentes linguagens de programação para cada serviço
Método de Comunicação	Comunica-se por meio de um barramento de serviço corporativo	Comunica-se via camada API com protocolos leves como REST
Escalabilidade	Pode ser um desafio escalar	Extremamente escalável por meio do uso de contêineres

Arquitetura X *Design* de software

“Atividades relacionadas a arquitetura de software são sempre de design. Entretanto, nem todas atividades de design são sobre arquitetura”

Elemar Junior Fundador e CEO da EximiaCo

Arquitetura de software

“Arquitetura de software tem como objetivo, que o objetivo de negócio seja atendido respeitando os atributos de qualidades e suas restrições de responsabilidades.”

Arquitetura de software

Exemplos:

1. **Arquitetura de microsserviços**
2. **Arquitetura serverless**
3. **Arquitetura orientada a eventos**
4. **...**

Design de software

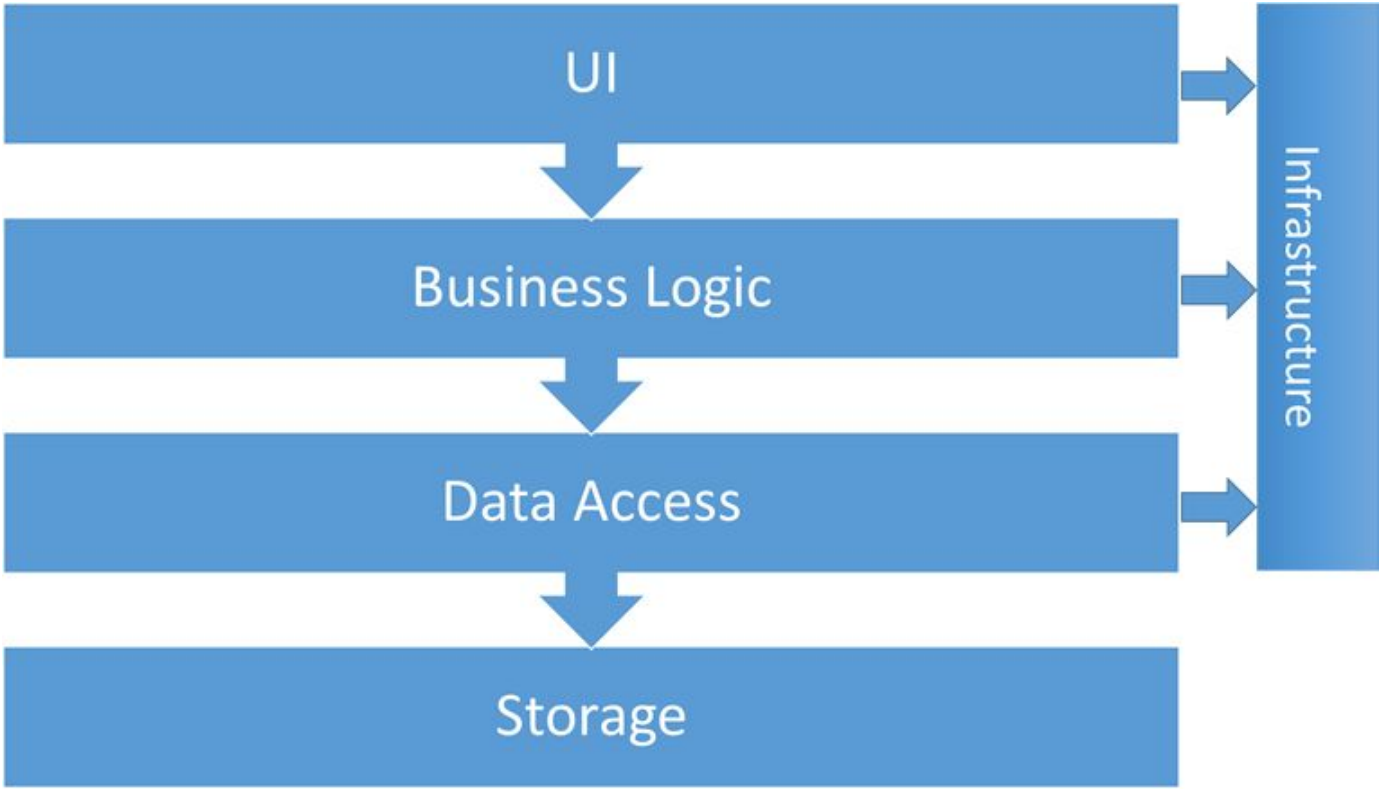
“O design de software é um exercício no gerenciamento da complexidade e trata no nível do código.”

Design de software

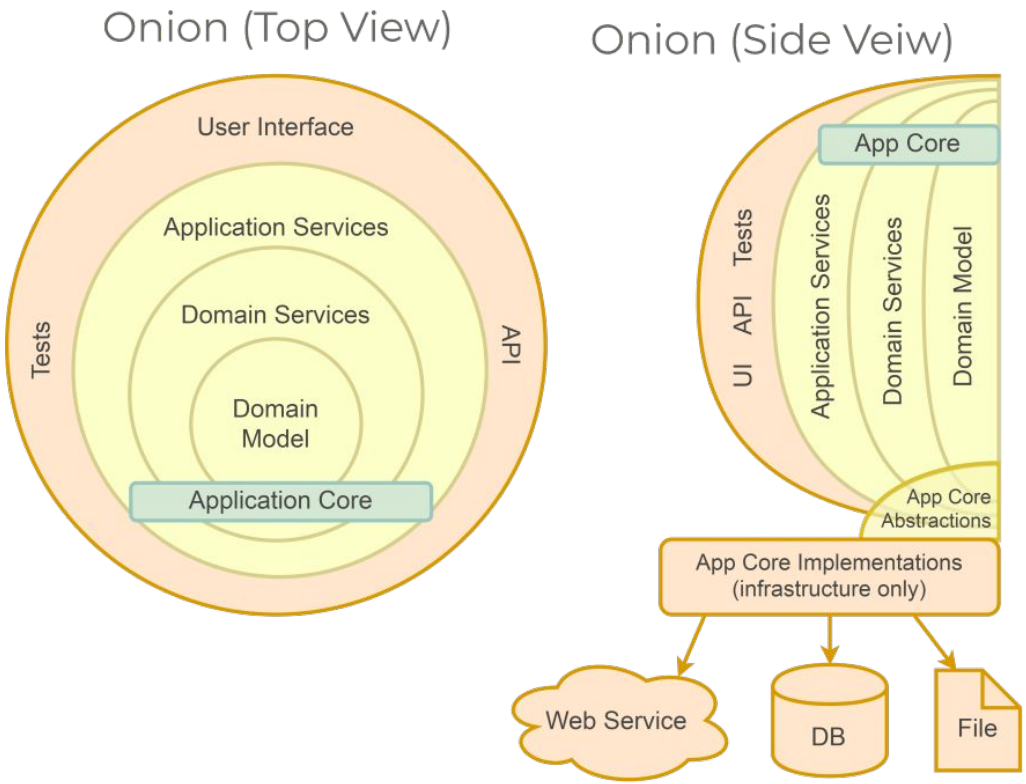
Exemplos:

1. **SOLID**
2. **Design patterns**
3. **Clean code**
4. **...**

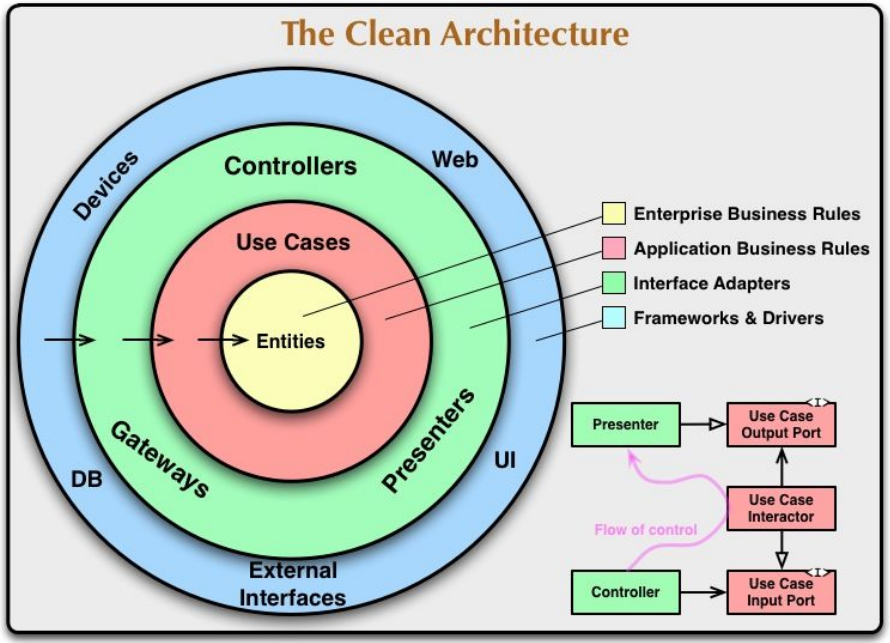
Padrão Arquitetural: Traditional Layered Architecture



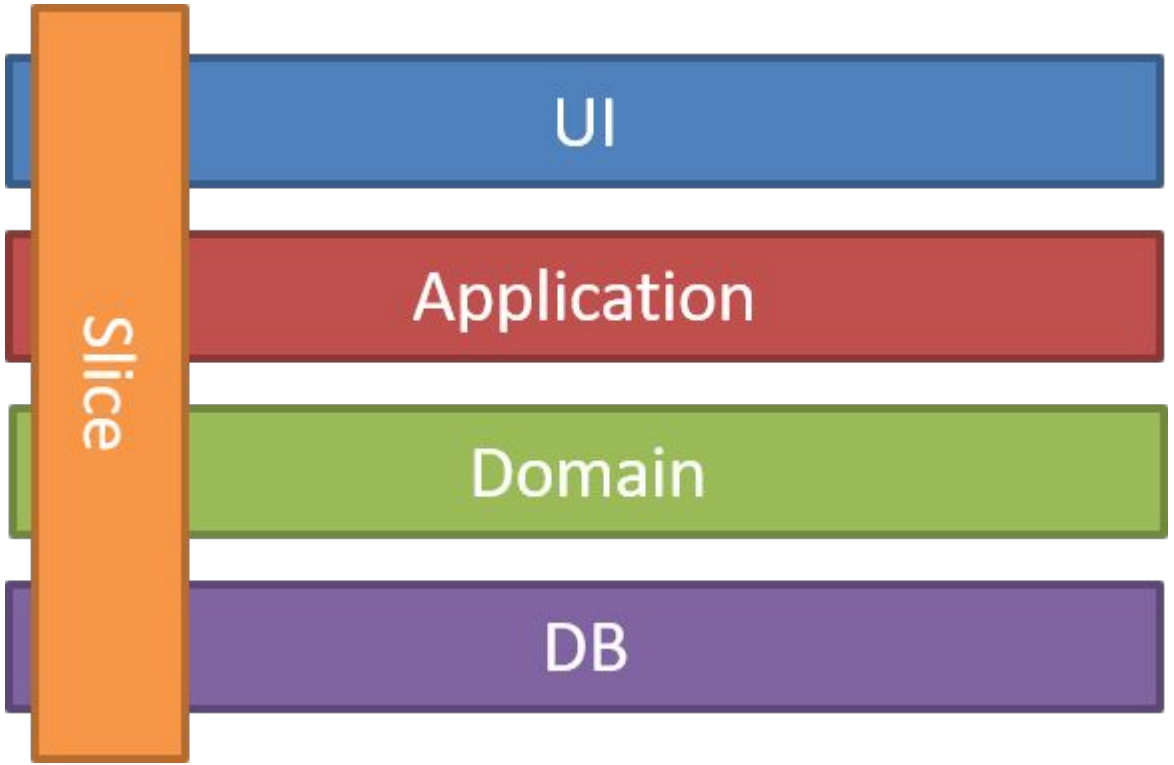
Padrão Arquitetural: Onion Architecture



Padrão Arquitetural: Clean architecture



Padrão Arquitetural: Clean architecture

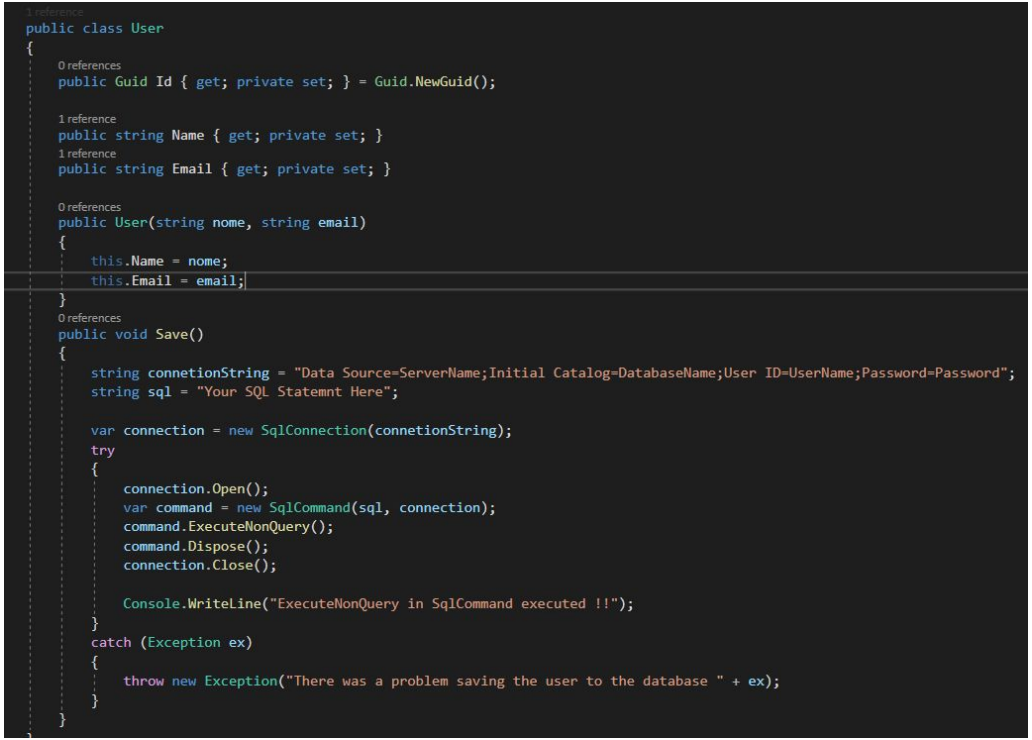


Base de tudo: S.O.L.I.D

1. Single Responsibility Principle
2. Open/Closed Principle
3. Liskov Substitution Principle
4. Interface Segregation Principle
5. Dependency Inversion Principle

Todos os códigos a seguir são ilustrativos =D.

1. Single Responsibility Principle



```
1 reference
public class User
{
    0 references
    public Guid Id { get; private set; } = Guid.NewGuid();

    1 reference
    public string Name { get; private set; }
    1 reference
    public string Email { get; private set; }

    0 references
    public User(string nome, string email)
    {
        this.Name = nome;
        this.Email = email;
    }
    0 references
    public void Save()
    {
        string connetionString = "Data Source=ServerName;Initial Catalog=DatabaseName;User ID=UserName;Password=Password";
        string sql = "Your SQL Statemnt Here";

        var connection = new SqlConnection(connetionString);
        try
        {
            connection.Open();
            var command = new SqlCommand(sql, connection);
            command.ExecuteNonQuery();
            command.Dispose();
            connection.Close();

            Console.WriteLine("ExecuteNonQuery in SqlCommand executed !!");
        }
        catch (Exception ex)
        {
            throw new Exception("There was a problem saving the user to the database " + ex);
        }
    }
}
```

1. Single Responsibility Principle

```
1 reference
public class User
{
    0 references
    public Guid Id { get; private set; } = Guid.NewGuid();

    1 reference
    public string Name { get; private set; }
    1 reference
    public string Email { get; private set; }

    0 references
    public User(string nome, string email)
    {
        this.Name = nome;
        this.Email = email;
    }
    0 references
    public void Save()
    {
        string connetionString = "Data Source=ServerName;Initial Catalog=DatabaseName;User ID=UserName;Password=Password";
        string sql = "Your SQL Statemnt Here";

        var connection = new SqlConnection(connetionString);
        try
        {
            connection.Open();
            var command = new SqlCommand(sql, connection);
            command.ExecuteNonQuery();
            command.Dispose();
            connection.Close();

            Console.WriteLine("ExecuteNonQuery in SqlCommand executed !!");
        }
        catch (Exception ex)
        {
            throw new Exception("There was a problem saving the user to the database " + ex);
        }
    }
}
```

“Uma classe deve ter somente uma razão para mudar”

1. Single Responsibility Principle

```
1 reference
public class User
{
    0 references
    public Guid Id { get; private set; } = Guid.NewGuid();

    1 reference
    public string Name { get; private set; }
    1 reference
    public string Email { get; private set; }

    0 references
    public User(string nome, string email)
    {
        this.Name = nome;
        this.Email = email;
    }
}
```

```
1 reference
public class UsuarioRepository
{
    private readonly ConnectionSQL _connection;
    0 references
    public UsuarioRepository(ConnectionSQL conexaoSQL)
    {
        _connection = conexaoSQL;
    }

    0 references
    public void Save(User user)
    {
        if(user != null)
        _connection.Save(user);
    }
}
```

```
3 references
public class ConnectionSQL
{
    private readonly SqlConnection _connection;
    0 references
    public ConnectionSQL(SqlConnection conexaoSQL, string connetionString)
    {
        _connection = new SqlConnection(connetionString);
    }


    1 reference
    public void Save(User user)
    {
        string sql = "INSERT INTO klant(id,anme,email) VALUES(@id,@name,@email)";

        try
        {
            using (SqlCommand cmd = new SqlCommand(sql, _connection))
            {
                cmd.Parameters.Add("@id", SqlDbType.Int).Value = user.Id;
                cmd.Parameters.Add("@name", SqlDbType.NText, 50).Value = user.Name;
                cmd.Parameters.Add("@email", SqlDbType.NText, 50).Value = user.Email;
                cmd.CommandType = CommandType.Text;
                cmd.ExecuteNonQuery();
            }

            Console.WriteLine("ExecuteNonQuery in SqlCommand executed !!");
        }
        catch (Exception ex)
        {
            throw new Exception("There was a problem saving the user to the database " + ex);
        }
    }
}
```

1. Single Responsibility Principle

```
1 reference
public class User
{
    0 references
    public Guid Id { get; private set; } = Guid.NewGuid();

    1 reference
    public string Name { get; private set; }
    1 reference
    public string Email { get; private set; }

    0 references
    public User(string nome, string email)
    {
        this.Name = nome;
        this.Email = email;
    }
}
```

```
1 reference
public class UsuarioRepository
{
    private readonly ConnectionSQL _connection;
    0 references
    public UsuarioRepository(ConnectionSQL conexaoSQL)
    {
        _connection = conexaoSQL;
    }

    0 references
    public void Save(User user)
    {
        if(user != null)
        _connection.Save(user);
    }
}
```

```
3 references
public class ConnectionSQL
{
    private readonly SqlConnection _connection;
    0 references
    public ConnectionSQL(SqlConnection conexaoSQL, string connetionString)
    {
        _connection = new SqlConnection(connetionString);
    }


    1 reference
    public void Save(User user)
    {
        string sql = "INSERT INTO klant(id,anme,email) VALUES(@id,@name,@email)";

        try
        {
            using (SqlCommand cmd = new SqlCommand(sql, _connection))
            {
                cmd.Parameters.Add("@id", SqlDbType.Int).Value = user.Id;
                cmd.Parameters.Add("@name", SqlDbType.NText, 50).Value = user.Name;
                cmd.Parameters.Add("@email", SqlDbType.NText, 50).Value = user.Email;
                cmd.CommandType = CommandType.Text;
                cmd.ExecuteNonQuery();
            }

            Console.WriteLine("ExecuteNonQuery in SqlCommand executed !!");
        }
        catch (Exception ex)
        {
            throw new Exception("There was a problem saving the user to the database " + ex);
        }
    }
}
```

“Não se deve depender de uma implementação e sim de uma abstração”. Mas isso é foco da parte 5

1. Single Responsibility Principle

```
1 reference
public class User
{
    0 references
    public Guid Id { get; private set; } = Guid.NewGuid();

    1 reference
    public string Name { get; private set; }
    1 reference
    public string Email { get; private set; }

    0 references
    public User(string nome, string email)
    {
        this.Name = nome;
        this.Email = email;
    }
}
```

```
1 reference
public class UsuarioRepository
{
    private readonly ConnectionSQL _connection;
    0 references
    public UsuarioRepository(ConnectionSQL conexaoSQL)
    {
        _connection = conexaoSQL;
    }

    0 references
    public void Save(User user)
    {
        if(user != null)
        _connection.Save(user);
    }
}
```

```
3 references
public class ConnectionSQL
{
    private readonly SqlConnection _connection;
    0 references
    public ConnectionSQL(SqlConnection conexaoSQL, string connetionString)
    {
        _connection = new SqlConnection(connetionString);
    }


    1 reference
    public void Save(User user)
    {
        string sql = "INSERT INTO klant(id,anme,email) VALUES(@id,@name,@email)";

        try
        {
            using (SqlCommand cmd = new SqlCommand(sql, _connection))
            {
                cmd.Parameters.Add("@id", SqlDbType.Int).Value = user.Id;
                cmd.Parameters.Add("@name", SqlDbType.NText, 50).Value = user.Name;
                cmd.Parameters.Add("@email", SqlDbType.NText, 50).Value = user.Email;
                cmd.CommandType = CommandType.Text;
                cmd.ExecuteNonQuery();
            }

            Console.WriteLine("ExecuteNonQuery in SqlCommand executed !!");
        }
        catch (Exception ex)
        {
            throw new Exception("There was a problem saving the user to the database " + ex);
        }
    }
}
```

“Não se deve depender de uma implementação e sim de uma abstração”. Mas isso é foco da parte 5

Todos os códigos a seguir são ilustrativos =D.

2. Open/Closed Principle

```
public class Salary
{
    0 references
    public decimal Calculate(decimal salary, int goal, string type)
    {
        if (type == "CTO")
            return salary * (goal/100)*10;
        else if (type == "head")
            return salary * (goal / 100) * 6;
        else if (type == "senior software enginner")
            return salary * (goal / 100) * 3;
        else if (type == "intern")
            throw new Exception("doesn't get salary hehehehe");
        else
            return -1;
    }
}
```

“Entidades de software (classes, módulos, funções etc) devem ser abertas para extensão mas fechadas para modificação

2. Open/Closed Principle

```
3 references
public abstract class Employee
{
    2 references
    public decimal Salary { get; set; }
    2 references
    public decimal Goal { get; set; }
    3 references
    public abstract decimal CalculateSalary();
}
```

```
/// don't repeat that
0 references
public class CTO : Employee
{
    3 references
    public override decimal CalculateSalary()
    {
        return Salary * (Goal / 100) * 10;
    }
}

0 references
public class Head : Employee
{
    3 references
    public override decimal CalculateSalary()
    {
        return Salary * (Goal / 100) * 10;
    }
}

0 references
public class Intern : Employee
{
    3 references
    public override decimal CalculateSalary()
    {
        throw new Exception("doesn't get salary hehehehe");
    }
}
```

3. Liskov Substitution Principle

```
public class Product
{
    0 references
    public Guid Id { get; private set; } = Guid.NewGuid();
    1 reference
    public string Name { get; set; }
    1 reference
    public Decimal Price { get; set; }

    2 references
    public Product(string nomeProduto, decimal precoProduto)
    {
        Name = nomeProduto;
        Price = precoProduto;
    }

    0 references
    public string Information(Product product)
    {
        if (product is Fridge)
            return InformationFridge(product as Fridge);
        if (product is Stove)
            return InformationStove(product as Stove);
        return null;
    }
```

“As classes base devem ser substituíveis por suas classes derivadas.”

3. Liskov Substitution Principle

```
3 references
public abstract class Product
{
    0 references
    public Guid Id { get; private set; } = Guid.NewGuid();
    1 reference
    public string Name { get; set; }
    1 reference
    public Decimal Price { get; set; }

    2 references
    public Product(string name, decimal price)
    {
        this.Name = name;
        this.Price = price;
    }

    2 references
    public abstract string Information();

}
1 reference
```

```
1 reference
public class Fridge : Product
{
    0 references
    public Fridge(string nomeProduto, decimal precoProduto) : base(nomeProduto, precoProduto)
    {
    }
    2 references
    public override string Information()
    {
        return "Fridge Frost Free";
    }
}
1 reference
public class Stove : Product
{
    0 references
    public Stove(string nomeProduto, decimal precoProduto) : base(nomeProduto, precoProduto)
    {
    }

    2 references
    public override string Information()
    {
        return "Stove four burner";
    }
}
```

4. Interface Segregation Principle

```
3 references
public interface IEmployee
{
    3 references
    public decimal CalculateSalaryCTO();
    3 references
    public decimal CalculateSalaryHEAD();
    3 references
    public decimal CalculateSalaryIntern();
}
```

```
0 references
public class CTO : IEmployee
{
    1 reference
    public decimal Salary { get; set; }
    1 reference
    public decimal Goal { get; set; }

    3 references
    public decimal CalculateSalaryCTO()
    {
        return Salary * (Goal / 100) * 10;
    }

    3 references
    public decimal CalculateSalaryHEAD()
    {
        throw new NotImplementedException();
    }

    3 references
    public decimal CalculateSalaryIntern()
    {
        throw new NotImplementedException();
    }
}

0 references
public class Head : IEmployee
{
    1 reference
    public decimal Salary { get; set; }
    1 reference
    public decimal Goal { get; set; }

    3 references
    public decimal CalculateSalaryCTO()
    {
        throw new NotImplementedException();
    }

    3 references
    public decimal CalculateSalaryHEAD()
    {
        return Salary * (Goal / 100) * 10;
    }

    3 references
    public decimal CalculateSalaryIntern()
    {
        throw new NotImplementedException();
    }
}
```

4. Interface Segregation Principle

```
public interface IEmployee
{
    2 references
    public decimal CalculateSalary();
}
```

```
0 references
public class CTO : IEmployee
{
    1 reference
    public decimal Salary { get; set; }
    1 reference
    public decimal Goal { get; set; }

    2 references
    public decimal CalculateSalary()
    {
        return Salary * (Goal / 100) * 10;
    }
}

0 references
public class Head : IEmployee
{
    1 reference
    public decimal Salary { get; set; }
    1 reference
    public decimal Goal { get; set; }

    2 references
    public decimal CalculateSalary()
    {
        return Salary * (Goal / 100) * 10;
    }
}
```

5. Dependency Inversion Principle

```
1 reference
public class ProductRepository
{
    2 references
    private SqlConnection Connection { get; set; }
    0 references
    public ProductRepository(SqlConnection connection)
    {
        Connection = connection;
    }
    0 references
    public void Save()
    {
        //code..
        Connection.Close();
    }
    0 references
    public Product GetProduct()
    {
        //code..
        return new Product();
    }
```

5. Dependency Inversion Principle

```
1 reference
public class ProductRepository
{
    2 references
    private IDbConnection Connection { get; set; }
    0 references
    public ProductRepository(IDbConnection connection)
    {
        Connection = connection;
    }
    0 references
    public void Save()
    {
        //code..
        Connection.Close();
    }
    0 references
    public Product GetProduct()
    {
        //code..
        return new Product();
    }
```

5. Dependency Inversion Principle

microservices necessita de uma arquitetura(ou melhor padrão arquitetural)?

Depende....

microservices necessita de uma arquitetura(ou melhor padrão arquitetural)?

Arquitetura é algo único, é feito de forma artesanal para resolver um problema específico. Então estude bem seu problema e estude sobre arquitetura, padrões arquiteturais, faça PoC e acima de tudo documente sua decisão através de ADR(architecture decision record).


KEEP
CALM
AND
Estude,Estude
Estude...

ATENÇÃO

É minha visão sobre o assunto.

Não tomar por verdade absoluta