Questões relacionadas a C#

1. Orientação a Objetos:

# • Explique o conceito de herança múltipla e como C# aborda esse cenário.

**A herança múltipla é um conceito da programação orientada a objetos em que uma classe pode herdar características e comportamentos de mais de uma classe base. C# não suporta herança múltipla direta de classes devido a problemas de complexidade e ambiguidade que ela pode introduzir. Em vez disso, C# permite herança múltipla através de interfaces. Uma classe em C# pode implementar múltiplas interfaces, permitindo que ela adote comportamentos de várias fontes sem herdar implementações de classes concretas.**

A screenshot of a computer

Description automatically generated

# • Explique o polimorfismo em C# e forneça um exemplo prático de como ele pode ser implementado.

**É um princípio da orientação a objetos que permite que objetos de diferentes classes sejam tratados como objetos de uma classe base comum. O polimorfismo permite que métodos sejam usados de forma genérica, mesmo que sua implementação específica varie entre diferentes classes. Exemplo, em uma classe de ContaPessoaFisica e ContaPessoaJuridica, podemos implementar o método Sacar(). Ambas as classes possuem o método com a mesma assinatura, porem com regras e logicas diferentes.**

A screenshot of a computer

Description automatically generated

**Ao aplicar o SRP, separamos a lógica de negócios do pedido da responsabilidade de persistência, tornando o código mais modular e fácil de manter.**

2. SOLID:

# • Descreva o princípio da Responsabilidade Única (SRP) e como ele se aplica em um contexto de desenvolvimento C#.

**O Princípio da Responsabilidade Única estabelece que uma classe deve ter apenas uma única responsabilidade ou razão para mudar. Isso significa que cada classe deve ser responsável por uma única parte da funcionalidade de um sistema e deve encapsular apenas um conjunto de comportamentos coesos.**

A screenshot of a computer

Description automatically generated

**Exemplo, se criarmos a classe ContaCorrente, teremos um repositório, onde fará todo o crud específico da ContaCorrente**

# • Como o princípio da inversão de dependência (DIP) pode ser aplicado em um projeto C# e como isso beneficia a manutenção do código?

**O Princípio da Inversão de Dependência sugere que módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações (por exemplo, interfaces). Além disso, abstrações não devem depender de detalhes, mas detalhes devem depender de abstrações**.

A screenshot of a computer

Description automatically generated

**Com o DIP, a classe NotificationService depende de uma abstração INotifier, permitindo a troca de implementações (e-mail, SMS, etc.) sem modificar o código da classe, promovendo uma fácil manutenção e expansão.**

3. Entity Framework (EF):

# • Como o Entity Framework gerencia o mapeamento de objetos para o banco de dados e vice-versa?

**O Entity Framework (EF) é uma ferramenta de mapeamento objeto-relacional (ORM) que automatiza a conversão entre objetos em C# e tabelas no banco de dados. Ele usa o conceito de modelos de dados que podem ser definidos usando classes POCO (Plain Old CLR Objects) ou através de um Designer de Modelos. O EF suporta dois tipos principais de mapeamento:**

1. **Code-First: Define o modelo de dados em código C# e, em seguida, gera o banco de dados.**
2. **Database-First: Usa um banco de dados existente para gerar classes de entidade no código C#.**

**Exemplo de Code-First:**

A screenshot of a computer

Description automatically generated

**No exemplo acima, o EF irá mapear a classe Product para uma tabela Products no banco de dados, onde cada propriedade da classe corresponde a uma coluna na tabela.**

# • Como otimizar consultas no Entity Framework para garantir um desempenho eficiente em grandes conjuntos de dados?

**Carregamento Sob Demanda (Lazy Loading): Carregar dados apenas quando necessário.**

A screenshot of a computer

Description automatically generated

**Carregamento Antecipado (Eager Loading): Carregar dados relacionados imediatamente para evitar consultas adicionais ao banco.**

A screenshot of a computer

Description automatically generated

**Carregamento Explícito (Explicit Loading): Carregar dados manualmente conforme necessário.**

A screenshot of a computer

Description automatically generated

**Filtragem e Projeção: Use consultas LINQ para filtrar e projetar apenas os dados necessários.**

A screenshot of a computer

Description automatically generated

**No-Tracking Queries: Para leituras rápidas que não exigem rastreamento de alterações.**

A screenshot of a computer

Description automatically generated

4. WebSockets:

# • Explique o papel dos WebSockets em uma aplicação C# e como eles se comparam às solicitações HTTP tradicionais.

**Os WebSockets são uma tecnologia que permite comunicação bidirecional em tempo real entre um cliente (como um navegador da web) e um servidor. Eles mantêm uma única conexão persistente, permitindo a troca de mensagens contínua sem a necessidade de reestabelecer a conexão para cada transmissão de dados.**

** HTTP Tradicional:**

* **Natureza: Requisição/resposta.**
* **Conexão: Cada requisição abre e fecha uma conexão.**
* **Overhead: Maior, devido à negociação repetida da conexão.**
* **Tempo Real: Menos adequado para comunicação em tempo real.**
* **Uso: Ideal para operações CRUD simples e páginas estáticas.**

** WebSockets:**

* **Natureza: Bidirecional.**
* **Conexão: Uma conexão persistente que permanece aberta.**
* **Overhead: Reduzido, já que não há negociação repetida de conexão.**
* **Tempo Real: Muito adequado para comunicação em tempo real.**
* **Uso: Ideal para aplicações de chat, jogos, atualizações de mercado em tempo real, etc.**

# • Quais são as principais considerações de segurança ao implementar uma comunicação baseada em WebSockets em uma aplicação C#?

** Autenticação e Autorização:**

* **Autenticação: Assegure-se de que os clientes sejam autenticados antes de estabelecer a conexão WebSocket. Use tokens de autenticação como JWT (JSON Web Tokens) para identificar usuários.**
* **Autorização: Verifique se o cliente tem permissão para acessar o recurso específico.**

** Proteção Contra Cross-Site WebSocket Hijacking:**

* **Verificação de Origem: Implemente verificações de origem (origin checks) para assegurar que os WebSockets só aceitem conexões de domínios confiáveis.**

** Limitação de Taxa e Controle de Conexões:**

* **Limitação de Taxa: Implemente mecanismos de limitação de taxa para prevenir que um cliente inicie muitas conexões rapidamente, o que pode levar a ataques DoS (Denial of Service).**
* **Controle de Conexões: Monitore e limite o número de conexões abertas por cliente ou IP para evitar sobrecarga no servidor.**

** Validação de Dados:**

* **Sanitização de Dados: Sempre valide e sanitize dados recebidos para proteger contra injeções de código ou outros ataques baseados em payload.**
* **Tamanho do Payload: Imponha limites no tamanho dos dados que podem ser enviados ou recebidos para evitar ataques de buffer overflow.**

** Mecanismo de Heartbeat:**

* **Detecção de Conexões Inativas: Implemente um mecanismo de heartbeat para detectar e encerrar conexões inativas ou quebradas, liberando recursos do servidor.**

** Logging e Monitoramento:**

* **Registro de Atividades: Registre atividades de WebSockets para auditoria e monitoramento de segurança.**
* **Detecção de Anomalias: Use ferramentas de monitoramento para detectar comportamentos anômalos ou ataques em potencial.**

5. Arquitetura:

# • Descreva a diferença entre arquitetura monolítica e arquitetura de microsserviços. Qual seria sua escolha ao projetar uma aplicação C#?

Arquitetura Monolítica:

* Estrutura: Toda a aplicação é desenvolvida como uma única unidade indivisível.
* Desenvolvimento: Mais simples para iniciar, com uma abordagem centralizada.
* Manutenção: Pode se tornar complexa com o aumento do tamanho da base de código.
* Deploy: Toda a aplicação deve ser implantada como um todo, mesmo para pequenas alterações.
* Escalabilidade: Escalada horizontal é difícil; geralmente, a aplicação é escalada verticalmente.
* Uso: Adequado para pequenas aplicações com requisitos de escalabilidade simples.

Arquitetura de Microsserviços:

* Estrutura: A aplicação é dividida em vários serviços independentes, cada um com seu próprio contexto.
* Desenvolvimento: Mais complexo, pois cada serviço pode ser desenvolvido, implantado e escalado independentemente.
* Manutenção: Mais fácil de gerenciar, pois cada serviço é independente e pode ser atualizado sem impactar outros serviços.
* Deploy: Serviços podem ser implantados individualmente, permitindo uma entrega contínua.
* Escalabilidade: Fácil de escalar horizontalmente, permitindo a escala individual de serviços conforme necessário.
* Uso: Ideal para aplicações grandes e complexas que exigem alta escalabilidade e flexibilidade.

**Depende da aplicação, da complexidade. Caso for uma aplicação simples com apenas um único processo, escolho a monolítica. Caso houver um processo complexo, onde exija diversas soluções, e projetos com alta demanda, escolheria uma aplicação de microsserviços**

# • Como você escolheria entre a arquitetura de microsserviços e a arquitetura monolítica ao projetar uma aplicação C# que precisa ser altamente escalável?

**Ao escolher entre monolítico e microsserviços, consideraria as necessidades específicas do projeto, a experiência da equipe e os objetivos a longo prazo da aplicação, validando os prós e contras de cada arquitetura.**