LUISDEV CONSULTORIA E TREINAMENTOS LTDA
CNPJ: 33.656.882/0001-05
R Cel José Eusébio, 95, Casa 13, Higienópolis, São Paulo - SP, CEP: 01239-030
Tel: (84) 9629-5551 | www.luisdev.com.br

Perguntas e Respostas de Entrevistas de Dev .NET Pleno e Sênior

1. O que é mensageria?

Mensageria é uma técnica de comunicação entre sistemas que permite o envio e recebimento de mensagens de forma assíncrona. Em uma arquitetura orientada a eventos, por exemplo, mensageria é usada para desacoplar a comunicação da aplicação com serviços externos e outros componentes do sistema.

Diferença conceitual
- Mensagem => vai ser consumida uma única vez
- Evento => poderá ser consumido por um ou mais inscritos

ENGLISH: Messaging is a communication technique between systems that allows the sending and receiving of messages asynchronously. In an event-driven architecture, for example, messaging is used to decouple the application's communication with external services and other system components.
Conceptual Difference
Message => will be consumed only once
Event => could be consumed by one or more subscribers

2. O que é CQRS?

CQRS é padrão arquitetural que separa as operações de leitura e escrita de um sistema, utilizando diferentes modelos de domínio para cada uma delas. O objetivo é melhorar a escalabilidade e o desempenho da aplicação, permitindo que as operações de leitura sejam otimizadas para consultas e as operações de escrita para atualizações.

ENGLISH: CQRS (Command Query Responsibility Segregation) is an architectural pattern that separates the read and write operations of a system, using different domain models for each of them. The aim is to improve the scalability and performance of the application, allowing read operations to be optimized for queries and write operations for updates.

3. O que é Event Sourcing?

Event Sourcing é uma técnica de arquitetura de software que consiste em armazenar todos os eventos que ocorrem em uma entidade, em vez de armazenar apenas o estado atual. Isso permite a reconstrução do estado atual a partir dos eventos, facilitando a implementação de recursos como desfazer e refazer e histórico de alterações.

ENGLISH: Event Sourcing is a software architecture technique that consists of storing all the events that occur in an entity, instead of storing just the current state. This allows the reconstruction of the current state from the events, facilitating the implementation of features such as undo and redo, and change history.

4. Como escolher entre bancos de dados SQL e NoSQL?

A escolha entre bancos de dados SQL e NoSQL depende das necessidades do projeto e das características dos dados que serão armazenados. Bancos de dados SQL são indicados para projetos que requerem integridade e consistência dos dados, e onde é importante garantir a conformidade com as restrições e regras de negócio. Bancos de dados NoSQL, por outro lado, são indicados para projetos que requerem escalabilidade horizontal, onde é necessário lidar com grandes quantidades de dados e onde a consistência dos dados pode ser eventual em favor da disponibilidade e desempenho.

ENGLISH: The choice between SQL and NoSQL databases depends on the project's needs and the characteristics of the data to be stored. SQL databases are suitable for projects that require data integrity and consistency, and where it is important to ensure compliance with business rules and restrictions. NoSQL databases, on the other hand, are suitable for projects that require horizontal scalability, where it is necessary to handle large amounts of data, and where data consistency can be eventual in favor of availability and performance.

5. O que são os princípios SOLID?

Os princípios SOLID são um conjunto de cinco princípios de desenho de software que ajudam a criar um código mais limpo, fácil de manter e extensível. Eles incluem o Princípio da Responsabilidade Única (SRP), que estabelece que uma classe deve ter apenas uma razão de ser alterada; o Princípio Aberto-Fechado (OCP), que estabelece que uma classe deve estar aberta para extensão, mas fechada para modificação; o Princípio da Substituição de Liskov (LSP), que estabelece que uma classe derivada deve ser substituível por sua classe base; o Princípio da Segregação de Interfaces (ISP), que estabelece que uma interface deve ter apenas métodos que são relevantes para seus clientes; e o Princípio da Inversão de Dependência (DIP), que estabelece que os módulos de alto nível não devem depender dos módulos de baixo nível, mas sim de abstrações. Cada um desses princípios ajuda a criar um código mais flexível, extensível e de fácil manutenção.

ENGLISH: The SOLID principles are a set of five software design principles that help to create cleaner, more maintainable, and extendable code.

They include the Single Responsibility Principle (SRP), which states that a class should have only one reason to change; the Open-Closed Principle (OCP), which states that a class should be open for extension but closed for modification; the Liskov Substitution Principle (LSP), which states that a derived class should be substitutable for its base class; the Interface Segregation Principle (ISP), which states that an interface should only have methods that are relevant to its clients; and the Dependency Inversion Principle (DIP), which states that high-level modules should not depend on low-level modules, but rather on abstractions. Each of these principles helps to create more flexible, extendable, and maintainable code.

6. O que são Design Patterns? Cite alguns que você utiliza/utilizou em seus projetos.

Design Patterns são soluções recorrentes para problemas de desenho de software, que foram desenvolvidas e documentadas ao longo dos anos por programadores experientes. Eles são soluções comprovadas que podem ser aplicadas em diferentes contextos para resolver problemas comuns. Exemplos de Design Patterns incluem Singleton, Factory Method, Strategy, Façade, Builder, entre outros.

ENGLISH: Design Patterns are recurring solutions to software design problems that have been developed and documented over the years by experienced programmers. They are proven solutions that can be applied in different contexts to solve common problems. Examples of Design Patterns include Singleton, Factory Method, Strategy, Facade, Builder, among others.

7. O que é POO? Quais os seus pilares?

POO (Programação Orientada a Objetos) é um paradigma de programação que se concentra na abstração de objetos e suas interações. Os pilares da POO são: encapsulamento, herança, polimorfismo e abstração. O encapsulamento é o princípio de ocultar a complexidade interna de um objeto, expondo apenas uma interface pública. A herança permite a criação de novas classes a partir de classes existentes, compartilhando comportamentos e propriedades. O polimorfismo permite que objetos de diferentes classes sejam tratados como se fossem do mesmo tipo, permitindo a execução de implementações de mesmo método de cada uma dessas classes. A abstração é o processo de abstrair um conceito, comportamento ou entidade em uma classe ou objeto no código.

ENGLISH: OOP (Object-Oriented Programming) is a programming paradigm that focuses on the abstraction of objects and their interactions. The pillars of OOP are: encapsulation, inheritance, polymorphism, and abstraction. Encapsulation is the principle of hiding an object's internal complexity, exposing only a public interface. Inheritance allows the creation of new classes from existing ones, sharing behaviors and properties. Polymorphism allows objects of different classes to be treated as if they were of the same type, allowing the execution of each of these class's method implementations.

Abstraction is the process of abstracting a concept, behavior, or entity into a class or object in code.

8. O que é DRY e YAGNI?

DRY (Don't Repeat Yourself) é um princípio de programação que enfatiza a importância de evitar a repetição de código. YAGNI (You Ain't Gonna Need It) é outro princípio que enfatiza a importância de não adicionar funcionalidades ou recursos que não são estritamente necessários no momento. Ambos os princípios visam a simplificar e reduzir a complexidade do código, promovendo a manutenibilidade e a flexibilidade.

ENGLISH: DRY (Don't Repeat Yourself) is a programming principle that emphasizes the importance of avoiding code repetition. YAGNI (You Ain't Gonna Need It) is another principle that emphasizes the importance of not adding features or resources that are not strictly necessary at the moment. Both principles aim to simplify and reduce the complexity of the code, promoting maintainability and flexibility.

9. Qual diferença entre composição e herança?

A composição é um princípio de programação que consiste em criar objetos complexos a partir de objetos mais simples, através de associações entre eles. A herança, por outro lado, é um princípio que permite que as classes sejam criadas a partir de outras classes existentes, compartilhando seus comportamentos e propriedades. A principal diferença entre os dois é que a herança é uma relação "é um", enquanto a composição é uma relação "tem um". A herança pode tornar o código mais fácil de entender e manter, mas também pode levar a problemas de acoplamento e complexidade excessiva.

ENGLISH: Composition is a programming principle that consists of creating complex objects from simpler ones, through associations between them. Inheritance, on the other hand, is a principle that allows classes to be created from other existing classes, sharing their behaviors and properties. The main difference between the two is that inheritance is an "is a" relationship, while composition is a "has a" relationship. Inheritance can make the code easier to understand and maintain, but it can also lead to coupling problems and excessive complexity.

10. O que é um API Gateway?

Um API Gateway é um componente de software que funciona como ponto de entrada para um conjunto de microserviços ou APIs. Ele fornece uma interface unificada para os clientes, permitindo que eles acessem múltiplas APIs ou serviços através de um único ponto de entrada. O API Gateway também pode ser usado para gerenciar autenticação, autorização, balanceamento de carga, roteamento, cache e monitoramento.

ENGLISH: An API Gateway is a software component that serves as an entry point for a set of microservices or APIs. It provides a unified interface for clients, allowing them to access multiple APIs or services through a single entry point. The API Gateway can also

be used to manage authentication, authorization, load balancing, routing, caching, and monitoring.

11. Como funciona caching em uma aplicação distribuída?

O caching em uma aplicação distribuída funciona armazenando dados frequentemente acessados em memória ou em um cache externo, como o Redis. Dessa forma, quando um usuário solicita um recurso, o cache é verificado primeiro para ver se ele já foi armazenado. Se o recurso estiver presente no cache, ele é retornado imediatamente. Se não, esse recurso precisará ser buscado em sua origem, e guardado na cache para as próximas requisições. Isso reduz o número de solicitações que são enviadas para o servidor e melhora a performance da aplicação.

ENGLISH: Caching in a distributed application works by storing frequently accessed data in memory or in an external cache, like Redis. This way, when a user requests a resource, the cache is first checked to see if it has already been stored. If the resource is present in the cache, it is returned immediately. If not, that resource needs to be fetched from its origin, and stored in the cache for subsequent requests.
This reduces the number of requests that are sent to the server and improves the performance of the application.

12. Quais as principais diferenças entre uma arquitetura monolítica e de microsserviços?

A arquitetura monolítica é uma abordagem de desenvolvimento de software em que todas as funcionalidades de uma aplicação são agrupadas em um único monólito. Essa abordagem torna mais fácil o desenvolvimento de pequenas aplicações que não possuem muitas complexidades e não precisam de escalabilidade, mas dificulta o processo de manutenção e atualização de aplicações grandes. Além disso, alterações em uma parte da aplicação podem ter impacto em outras partes. Já a arquitetura de microsserviços consiste na divisão de uma aplicação em componentes menores e independentes, cada um executando um conjunto de funcionalidades específicos e relacionados (comumente através de Agregados). Esses componentes são desenvolvidos e implantados separadamente, o que torna a manutenção e atualização mais fácil.

Além disso, o uso de microsserviços permite a escalabilidade individual de cada componente, melhorando a performance e a disponibilidade da aplicação. Entretanto, essa abordagem traz desafios adicionais em termos de comunicação, orquestração e segurança

ENGLISH: Monolithic architecture is a software development approach in which all the features of an application are bundled into a single monolith. This approach makes it easier to develop small applications that don't have many complexities and don't need

scalability, but it makes the process of maintaining and updating large applications difficult. Moreover, changes in one part of the application can impact other parts.
On the other hand, microservices architecture consists of dividing an application into smaller, independent components, each performing a set of specific, related functionalities (commonly through Aggregates). These components are developed and deployed separately, which makes maintenance and updating easier.

Also, the use of microservices allows the individual scalability of each component, improving the performance and availability of the application. However, this approach brings additional challenges in terms of communication, orchestration, and security.

13. O que é uma expressão lambda?

Expressão lambda é um recurso da linguagem que permite criar funções anônimas de maneira concisa. As expressões lambda são usadas para passar código como parâmetros de outras funções ou métodos. Elas são frequentemente usadas para definir predicados, comparadores e transformações de coleções em uma sintaxe mais concisa e legível.

ENGLISH: Lambda expression is a language feature that allows the creation of anonymous functions concisely. Lambda expressions are used to pass code as parameters to other functions or methods. They are often used to define predicates, comparators, and transformations of collections in a more concise and readable syntax.

14. Qual a diferença entre uma classe abstrata e uma interface?

Tanto a classe abstrata quanto a interface são mecanismos de abstração em C#. A diferença principal entre elas é que uma classe abstrata pode conter implementações de métodos, enquanto uma interface só pode definir as assinaturas de métodos. Além disso, uma classe pode implementar várias interfaces, mas só pode herdar de uma classe abstrata. Já uma classe abstrata pode ter construtores, campos e propriedades, enquanto uma interface só pode ter propriedades, métodos e eventos.

ENGLISH: Both abstract class and interface are mechanisms of abstraction in C#. The main difference between them is that an abstract class can contain method implementations, while an interface can only define method signatures. Moreover, a class can implement multiple interfaces, but it can only inherit from one abstract class. An abstract class can have constructors, fields, and properties, while an interface can only have properties, methods, and events.

15. O que é e como criar um método de extensão?

Métodos de extensão permitem adicionar novos métodos a tipos existentes sem precisar criar uma nova classe derivada. Eles são criados criando-se uma classe estática e um

método estático, que aceita o tipo que se deseja estender como primeiro parâmetro e utiliza a palavra-chave "this" na definição do parâmetro para indicar que se trata de um método de extensão. Métodos de extensão são úteis para adicionar comportamentos específicos a tipos de bibliotecas externas ou para estender tipos base da linguagem.

ENGLISH: Extension methods allow you to add new methods to existing types without needing to create a new derived class. They are created by creating a static class and a static method, which accepts the type you want to extend as the first parameter and uses the "this" keyword in the parameter definition to indicate that it is an extension method. Extension methods are useful for adding specific behaviors to types from external libraries or to extend base types of the language.

16. Qual o propósito da instrução using?

A instrução using é usada para gerenciar recursos não gerenciados em C#. Ela permite que um objeto que implemente a interface IDisposable seja usado dentro de um bloco de código, e ao final do bloco, o método Dispose do objeto é chamado automaticamente. Isso garante que os recursos utilizados pelo objeto sejam liberados, mesmo se ocorrerem exceções durante a execução do bloco de código.

ENGLISH: The using statement is used to manage unmanaged resources in C#. It allows an object that implements the IDisposable interface to be used within a code block, and at the end of the block, the Dispose method of the object is automatically called. This ensures that the resources used by the object are released, even if exceptions occur during the execution of the code block.

17. Qual a diferença entre overload e override?

Overload e override são conceitos relacionados a polimorfismo. Overload se refere à criação de vários métodos com o mesmo nome, mas com diferentes parâmetros, o que permite que diferentes sobrecargas do método sejam chamadas dependendo dos argumentos passados. Override se refere à criação de um método em uma classe derivada que substitui a implementação de um método com o mesmo nome na classe base. O método substituto deve ter a mesma assinatura que o método original e deve usar a palavra-chave "override".

ENGLISH: Overload and override are concepts related to polymorphism. Overload refers to the creation of multiple methods with the same name but different parameters, which allows different overloads of the method to be called depending on the arguments passed. Override refers to creating a method in a derived class that replaces the implementation of a method with the same name in the base class. The substitute method must have the same signature as the original method and must use the "override" keyword.

18. Qual a diferença entre IEnumerable e IQueryable?

IEnumerable e IQueryable são interfaces usadas para consultar coleções de dados em C#. IEnumerable é usada para coleções em memória, como listas, e fornece métodos para enumerar a coleção. IQueryable é usada para consultar coleções de dados que podem estar armazenadas fora da memória, como em um banco de dados. IQueryable permite a construção de consultas de forma mais eficiente, enviando as consultas para o servidor de banco de dados de forma mais otimizada.

ENGLISH: IEnumerable and IQueryable are interfaces used to query data collections in C#. IEnumerable is used for in-memory collections, like lists, and provides methods for enumerating the collection. IQueryable is used for querying data collections that might be stored outside of memory, such as in a database. IQueryable allows for constructing queries more efficiently, sending the queries to the database server in a more optimized manner.

19. O que acontece quando se chama ToList em uma coleção?

Quando você chama o método ToList() em um objeto do tipo IEnumerable, todos os dados na coleção são carregados na memória do sistema e são manipulados pela aplicação. O resultado é que todas as operações de consulta são executadas em memória. Isso pode ser problemático para grandes conjuntos de dados, pois pode levar a problemas de desempenho e limitações de memória. Já ao chamar o método ToList() em um objeto do tipo IQueryable, o LINQ gera uma consulta SQL a partir das expressões de consulta definidas no código e envia essa consulta para o banco de dados. O banco de dados, então, retorna apenas o subconjunto de dados solicitado pela consulta. A consulta só é executada quando você chama o método ToList(), que traz os dados do banco de dados para a memória. Esse processo deixa a execução muito mais rápida, especialmente em conjuntos de dados grandes.

ENGLISH: When you call the ToList() method on an IEnumerable object, all the data in the collection are loaded into the system memory and manipulated by the application. The result is that all query operations are performed in memory. This can be problematic for large data sets as it can lead to performance issues and memory limitations.
When calling the ToList() method on an IQueryable object, LINQ generates a SQL query from the query expressions defined in the code and sends that query to the database. The database then returns only the subset of data requested by the query. The query is only executed when you call the ToList() method, which fetches the data from the database into memory. This process makes the execution much faster, especially for large data sets.

20. O que é LINQ?

LINQ é um componente da Microsoft que permite a manipulação de dados de forma consistente, com duas abordagens: query-based e method-based. O query-based utiliza uma sintaxe de consulta semelhante ao SQL, enquanto o method-based utiliza chamadas de métodos que podem ser encadeadas. Com o LINQ, é possível realizar consultas em coleções de objetos, bancos de dados e serviços web, permitindo que as operações sejam executadas de forma mais rápida e simples.

ENGLISH: LINQ is a Microsoft component that allows for consistent data manipulation, with two approaches: query-based and method-based.
The query-based approach uses a SQL-like query syntax, while the method-based approach uses method calls that can be chained.
With LINQ, it is possible to perform queries on collections of objects, databases, and web services, allowing operations to be performed more quickly and simply.

21. O que é e como funciona generics?

Generics permitem que tipos de dados sejam definidos em um momento posterior à criação do código, oferecendo maior flexibilidade e reutilização de código. A principal vantagem é que as classes e métodos genéricos podem ser reutilizados com tipos diferentes, sem a necessidade de reescrever o código para cada tipo de dado.

ENGLISH: Generics allow data types to be defined at a later time than the code creation, offering more flexibility and code reusability. The main advantage is that generic classes and methods can be reused with different types, without the need to rewrite the code for each data type.

22. O que é async e await?

Async e Await são recursos do C# que permitem que as operações assíncronas sejam executadas de forma mais eficiente, sem a necessidade de bloquear a thread que está executando aquele código. Quando uma operação é marcada como "await", a execução do código é pausada até que a operação assíncrona seja concluída, permitindo que outras operações sejam executadas em paralelo. Isso resulta em um aplicativo mais responsivo e com melhor desempenho.

ENGLISH: Async and Await are features of C# that allow asynchronous operations to be performed more efficiently, without needing to block the thread that is running that code. When an operation is marked as "await", the execution of the code is paused until the asynchronous operation is completed, allowing other operations to be performed in parallel. This results in a more responsive application with better performance.

23. Como é possível aguardar a execução de múltiplas Tasks?

É possível aguardar a execução de múltiplas tasks usando o método Task.WhenAll. Esse método permite que várias tasks sejam executadas em paralelo e aguarda até que todas sejam concluídas. Um exemplo de código seria:

```
var task1 = ApiCallOne();
var task2 = ApiCallTwo();
var task3 = ApiCallThree();
await Task.WhenAll(task1, task2, task3);
```

ENGLISH: It is possible to await the execution of multiple tasks using the Task.WhenAll method. This method allows multiple tasks to be run in parallel and waits until they are all completed. An example code would be:

```
var task1 = ApiCallOne();
var task2 = ApiCallTwo();
var task3 = ApiCallThree();
await Task.WhenAll(task1, task2, task3);
```

24. O que é ConfigureAwait?

ConfigureAwait é um método em C# que pode ser usado para configurar o contexto de sincronização de uma operação assíncrona. Ao chamar ConfigureAwait com o argumento false, você está especificando que o código após a operação assíncrona não precisa ser executado no mesmo contexto que a operação original. Isso pode melhorar a performance ao evitar o retorno ao contexto original desnecessariamente. É particularmente útil em bibliotecas de código, onde o contexto de sincronização pode não ser relevante.

ENGLISH*:* ConfigureAwait is a method in C# that can be used to configure the synchronization context of an asynchronous operation. By calling ConfigureAwait with the argument false, you're specifying that the code following the asynchronous operation does not need to be executed in the same context as the original operation. This can improve performance by avoiding returning to the original context unnecessarily. It's particularly useful in code libraries, where the synchronization context may not be relevant.

25. O que é um delegate?

Um delegate em C# é um tipo que representa referências a métodos com um tipo de parâmetro e retorno específicos. Quando você instancia um delegate, você pode associá-lo a qualquer método com uma assinatura compatível. Delegates são usados para passar métodos como parâmetros para outros métodos, que podem chamar o método referenciado pelo delegate.

ENGLISH: A delegate in C# is a type that represents references to methods with a specific parameter and return type. When you instantiate a delegate, you can associate it with any method with a compatible signature. Delegates are used to pass methods as parameters to other methods, which can call the method referenced by the delegate.

26. Qual a diferença entre Func, Action, e Predicate?

Func, Action e Predicate são tipos de delegates. Func é um delegate que representa um método que recebe um ou mais parâmetros e retorna um valor. Action é um delegate que representa um método que recebe um ou mais parâmetros e não retorna um valor (void). Predicate é um tipo especial de Func que sempre recebe um parâmetro e retorna um booleano.

ENGLISH: Func, Action, and Predicate are types of delegates. Func is a delegate that represents a method that takes one or more parameters and returns a value. Action is a delegate that represents a method that takes one or more parameters and doesn't return a value (void). Predicate is a special type of Func that always takes one parameter and returns a boolean.

27. O que são Concurrent Collections?

Concurrent Collections em .NET são classes que permitem o acesso concorrente seguro a coleções de dados de vários threads. Exemplos incluem ConcurrentQueue, ConcurrentStack, ConcurrentDictionary, etc. Estas coleções são projetadas para serem thread-safe, o que significa que elas gerenciam o acesso de múltiplas threads a seus dados de forma a prevenir condições de corrida.

ENGLISH: Concurrent Collections in .NET are classes that allow safe concurrent access to collections of data from multiple threads. Examples include ConcurrentQueue, ConcurrentStack, ConcurrentDictionary, etc. These collections are designed to be thread-safe, which means they manage the access of multiple threads to their data in a way to prevent race conditions.

28. Como funciona try-catch-finally?

A estrutura try-catch-finally é usada para manipular exceções no código. O bloco "try" contém o código que pode potencialmente gerar uma exceção. Se uma exceção ocorrer, o fluxo de execução é transferido para o bloco "catch", que é usado para tratar ou lidar com a exceção que ocorreu. Independentemente de uma exceção ocorrer ou não, o bloco "finally" é executado depois do bloco "try" e "catch". Este bloco é geralmente usado para executar qualquer tarefa de limpeza necessária, como fechar conexões de arquivo ou banco de dados.

ENGLISH: The try-catch-finally structure is used to handle exceptions in the code. The "try" block contains the code that could potentially throw an exception. If an exception occurs, the execution flow is transferred to the "catch" block, which is used to handle or deal with the exception that occurred. Regardless of whether an exception occurs or not, the "finally" block is executed after the "try" and "catch" blocks. This block is usually used to perform any necessary cleanup tasks, such as closing file or database connections.

29. Como funciona a palavra-chave yield?

A palavra-chave "yield" é usada para criar um iterador em C#. Em vez de retornar uma coleção completa de itens, a instrução "yield return" retorna um item de cada vez, à medida que o iterador é chamado, o que permite economizar memória e melhorar a performance, pois os itens são gerados conforme necessário.

ENGLISH: The "yield" keyword is used to create an iterator in C#. Instead of returning a full collection of items, the "yield return" statement returns one item at a time as the iterator is called, which saves memory and improves performance because items are generated as needed.

30. O que são Threads e Tasks?

Threads são fluxos de execução de código que permitem que múltiplas tarefas sejam executadas simultaneamente. As tasks são um tipo de thread que foram introduzidas no .NET Framework 4.0 e são uma maneira mais fácil e eficiente de executar operações em segundo plano. As tasks usam menos recursos do sistema e são mais flexíveis do que as threads, permitindo que o código seja executado em paralelo e com melhor desempenho.

ENGLISH: Threads are code execution flows that allow multiple tasks to be performed simultaneously. Tasks are a type of thread that was introduced in .NET Framework 4.0 and are an easier and more efficient way to perform background operations. Tasks use fewer system resources and are more flexible than threads, allowing code to run in parallel and with better performance.

31. Qual a diferença entre .NET Framework, .NET Core, e .NET?

O .NET Framework é uma implementação do .NET original que roda apenas em sistemas operacionais Windows, enquanto o .NET Core é uma versão multiplataforma do .NET que roda em várias plataformas, incluindo Windows, macOS e Linux. .NET é uma família de tecnologias e ferramentas que inclui o .NET Framework e o .NET Core, bem como outras implementações, como o .NET 5 e o .NET Standard.

O .NET Standard é uma especificação que define as APIs que devem estar presentes em todas as implementações do .NET, independentemente da plataforma de destino. O

objetivo do .NET Standard é permitir que os desenvolvedores criem bibliotecas .NET que possam ser consumidas por diferentes implementações do .NET, incluindo o .NET Framework, o .NET Core e o Xamarin.

ENGLISH: NET Framework is an implementation of the original .NET that runs only on Windows operating systems, while .NET Core is a cross-platform version of .NET that runs on several platforms, including Windows, macOS, and Linux. .NET is a family of technologies and tools that includes .NET Framework and .NET Core, as well as other implementations such as .NET 5 and .NET Standard.
.NET Standard is a specification that defines the APIs that must be present in all .NET implementations, regardless of the target platform. The purpose of .NET Standard is to allow developers to create .NET libraries that can be consumed by different .NET implementations, including .NET Framework, .NET Core, and Xamarin.

32. O que é Intermediate Language (IL)?

Intermediate Language (IL) é uma linguagem de baixo nível que é gerada pelo compilador do .NET a partir do código-fonte. O IL é executado pelo CLR e pode ser considerado uma forma de código executável intermediário entre o código-fonte e o código de máquina.

ENGLISH: Intermediate Language (IL) is a low-level language that is generated by the .NET compiler from the source code. The IL is executed by the CLR and can be considered a form of intermediate executable code between the source code and the machine code.

33. Como a compilação Just-in-Time (JIT) funciona?

A compilação Just-in-Time (JIT) é um processo de compilação que o CLR usa para converter o código IL em código de máquina nativo durante o tempo de execução. O CLR compila cada método em um assembly .NET em tempo de execução, à medida que o método é executado pela primeira vez. A compilação JIT pode melhorar o desempenho do tempo de execução, pois permite que o CLR otimize o código para a plataforma específica em que está sendo executado.

ENGLISH: Just-In-Time (JIT) compilation is a compilation process that the CLR uses to convert IL code into native machine code during runtime. The CLR compiles each method in a .NET assembly at runtime, as the method is executed for the first time. JIT compilation can improve runtime performance as it allows the CLR to optimize the code for the specific platform on which it is being run.

34. O que é Common Language Runtime (CLR)?

O Common Language Runtime (CLR) é a máquina virtual responsável por executar o código gerenciado, independentemente da implementação específica do .NET. Ele fornece recursos como gerenciamento de memória, segurança, coleta de lixo e gerenciamento de exceções.

O Common Language Runtime (CLR) é o componente do .NET Framework (e outras implementações do .NET) que executa o Intermediate Language (IL). Quando um programa é compilado em C# ou outra linguagem .NET, o compilador gera um arquivo executável (como um .exe ou .dll) que contém código IL em vez de código de máquina nativo. Durante a execução do programa, o CLR carrega o arquivo executável e o traduz em tempo de execução para o código de máquina nativo do sistema em que o programa está sendo executado. Isso é conhecido como compilação Just-in-Time (JIT) e é uma das maneiras pelas quais o CLR ajuda a garantir a segurança e a interoperabilidade do código .NET.

Então, em resumo, o CLR executa o Intermediate Language (IL), que é traduzido em tempo de execução para código de máquina nativo.

ENGLISH: The Common Language Runtime (CLR) is the virtual machine responsible for running managed code, regardless of the specific .NET implementation. It provides features like memory management, security, garbage collection, and exception handling.

The Common Language Runtime (CLR) is the component of the .NET Framework (and other .NET implementations) that runs the Intermediate Language (IL). When a program is compiled in C# or another .NET language, the compiler generates an executable file (such as a .exe or .dll) that contains IL code rather than native machine code. During the execution of the program, the CLR loads the executable file and translates it at runtime into the native machine code of the system on which the program is being run. This is known as Just-In-Time (JIT) compilation and is one of the ways in which the CLR helps to ensure the security and interoperability of .NET code.

So, in summary, the CLR runs the Intermediate Language (IL), which is translated at runtime into native machine code.

35. Como funciona o gerenciamento de memória no .NET? E o Garbage Collector (GC)?

O gerenciamento de memória no .NET é realizado por meio de um coletor de lixo (garbage collector). O garbage collector é responsável por identificar e coletar automaticamente os objetos não utilizados na memória. Isso permite que os desenvolvedores não precisem gerenciar manualmente a alocação e a liberação de memória. O garbage collector opera de maneira assíncrona, periodicamente, e tem como objetivo manter a alocação de memória em níveis eficientes.

O Garbage Collector (GC) é um recurso da Common Language Runtime (CLR) responsável por gerenciar a memória utilizada por um programa .NET. Ele é responsável por alocar e desalocar memória conforme necessário, evitando que o programa sofra de problemas como memory leaks e memory fragmentation.

O GC funciona em três camadas: a camada de geração 0, a camada de geração 1 e a camada de geração 2.

Na camada de geração 0, os objetos recém-criados são alocados. A maioria dos objetos é desalocada rapidamente nessa camada, quando eles saem do escopo, o que a torna a camada com menor tempo de vida para objetos na memória. Já a camada de geração 1 é a área intermediária para objetos que sobrevivem à coleta da geração 0. E, por fim, a camada de geração 2 é a área para objetos de longa duração que sobrevivem às duas primeiras coletas.

O GC opera de forma inteligente e periódica em cada camada para garantir que a memória seja gerenciada da melhor forma possível.

ENGLISH: Memory management in .NET is carried out through a garbage collector (GC). The garbage collector is responsible for automatically identifying and collecting unused objects in memory. This allows developers not to have to manually manage the allocation and release of memory. The garbage collector operates asynchronously, and periodically, aiming to keep memory allocation at efficient levels.

The Garbage Collector (GC) is a feature of the Common Language Runtime (CLR) responsible for managing the memory used by a .NET program.

It is responsible for allocating and deallocating memory as needed, preventing the program from suffering from problems such as memory leaks and memory fragmentation.

The GC operates on three layers: generation 0 layer, generation 1 layer, and generation 2 layer. In the generation 0 layer, newly created objects are allocated. Most objects are quickly deallocated at this layer when they go out of scope, making it the layer with the shortest lifespan for objects in memory. The generation 1 layer is the intermediate area for objects that survive the generation 0 collection. Finally, the generation 2 layer is the area for long-lived objects that survive the first two collections.

The GC operates intelligently and periodically on each layer to ensure that memory is managed in the best possible way.

36. Qual a diferença entre os tipos de valor e tipos de referência?

Os tipos de valor armazenam diretamente o valor na memória, enquanto os tipos de referência armazenam um ponteiro para a memória em que o valor está localizado. Os tipos de valor são armazenados na pilha (stack) e são gerenciados pelo sistema operacional, enquanto os tipos de referência são armazenados no heap, que é uma região de memória gerenciada pelo Garbage Collector. A alocação de tipos de valor é mais rápida, mas o uso de tipos de referência é mais flexível e eficiente para estruturas de dados complexas.

ENGLISH: Value types directly store the value in memory, while reference types store a pointer to the memory where the value is located. Value types are stored on the stack and are managed by the operating system, while reference types are stored on the heap, which is a region of memory managed by the Garbage Collector.
The allocation of value types is faster, but the use of reference types is more flexible and efficient for complex data structures.

37. O que são REST APIs?

REST (Representational State Transfer) é um estilo arquitetural de desenvolvimento de APIs, que utiliza o protocolo HTTP para expor operações que podem ser executadas em recursos específicos. O objetivo é fornecer uma interface uniforme para acesso aos recursos, independente de plataforma ou linguagem de programação utilizada.

ENGLISH: REST (Representational State Transfer) is an architectural style of API development, which uses the HTTP protocol to expose operations that can be performed on specific resources. The goal is to provide a uniform interface for accessing resources, regardless of the platform or programming language used.

38. Quais os tipos de caching e como funcionam?

Existem dois tipos principais de caching: server-side e client-side. O server-side caching é feito pelo servidor que hospeda a API, e consiste em armazenar o resultado de uma requisição para fornecer a mesma resposta quando a mesma requisição é feita novamente. O client-side caching é realizado pelo cliente, e consiste em armazenar a resposta recebida para a mesma requisição, para que não precise ser feita novamente.

Existem dois tipos de caching server-side: cache de memória e cache distribuído. O cache de memória armazena os dados em memória RAM e é acessado rapidamente, permitindo uma melhoria significativa de performance em consultas repetidas. Já o cache distribuído armazena os dados em uma infraestrutura distribuída, como um banco de dados em memória ou um servidor de cache dedicado, permitindo compartilhar os dados entre várias instâncias de um aplicativo e melhorando a escalabilidade horizontal.

O ASP.NET Core oferece suporte nativo a ambos os tipos de caching e, dependendo do cenário, pode ser necessário escolher uma ou outra opção.

ENGLISH: There are two main types of caching: server-side and client-side. Server-side caching is done by the server hosting the API and consists of storing the result of a request to provide the same response when the same request is made again. Client-side caching is carried out by the client and consists of storing the received response for the same request, so that it does not need to be made again.

There are two types of server-side caching: memory cache and distributed cache. Memory cache stores data in RAM and is accessed quickly, allowing significant performance improvement for repeated queries. On the other hand, distributed cache stores data in a distributed infrastructure, such as an in-memory database or a dedicated cache server, allowing data sharing among multiple instances of an application and improving horizontal scalability. ASP.NET Core offers native support for both types of caching and, depending on the scenario, one or the other option may need to be chosen.

39. Quais ações de melhoria em performance podemos fazer em uma API?

Para melhorar o desempenho de uma API ASP.NET Core, algumas ações incluem: otimização de banco de dados, redução de latência de rede, diminuir quantidade de requisições externas, melhoria na performance de acesso a serviços externos, utilização de caching server-side (como memory cache e cache distribuído), e otimização de código com multi-threading e/ou paralelismo, e algoritmos.

Porém, vale lembrar que a otimização da performance não é uma ação única, mas sim um processo contínuo, que deve ser avaliado e aprimorado constantemente. É importante analisar e entender os gargalos de performance da API e aplicar as ações que melhor atendam às necessidades do projeto.

ENGLISH: To improve the performance of an ASP.NET Core API, some actions include: database optimization, reducing network latency, reducing the number of external requests, improving access performance to external services, using server-side caching (like memory cache and distributed cache), and code optimization with multi-threading and/or parallelism, and algorithms.

However, it is important to remember that performance optimization is not a one-off action, but rather a continuous process, which should be assessed and improved constantly. It is important to analyze and understand the performance bottlenecks of the API and apply the actions that best meet the project's needs.

40. O que é injeção de dependência e quais os tempos de ciclo de vida disponíveis?

Injeção de dependência é um padrão de projeto que permite o gerenciamento de dependências de uma aplicação. No ASP.NET Core, ele é utilizado para injetar instâncias de classes em outras classes, sem que seja necessário criar as instâncias manualmente. Existem três tipos de ciclo de vida de injeção de dependência: Transient (a cada instância), Scoped (por solicitação) e Singleton (uma única instância).

ENGLISH: Dependency injection is a design pattern that allows the management of an application's dependencies. In ASP.NET Core, it is used to inject instances of classes into other classes, without the need to manually create the instances. There are three types of dependency injection lifecycle: Transient (per instance), Scoped (per request), and Singleton (a single instance).

41. Quais as diferenças entre EF Core e Dapper?

EF Core e Dapper são duas bibliotecas para acesso a banco de dados em .NET. EF Core é um ORM (Object Relational Mapper) completo, que abstrai o acesso ao banco de dados por meio de classes. Dapper é uma biblioteca mais simples, que utiliza consultas SQL para realizar operações no banco de dados. EF Core é mais abrangente e pode ser mais fácil de usar em projetos maiores, enquanto o Dapper é mais rápido e pode ser mais adequado para projetos menores, mas também tem seus usos em projetos maiores principalmente por ser performático.

ENGLISH: EF Core and Dapper are two libraries for database access in .NET. EF Core is a complete ORM (Object-Relational Mapper) that abstracts access to the database through classes. Dapper is a simpler library that uses SQL queries to perform operations on the database. EF Core is more comprehensive and may be easier to use in larger projects, while Dapper is faster and may be more suitable for smaller projects. However, Dapper also has its uses in larger projects, mainly because it is performant.

42. Como funciona a pipeline de requisição no ASP.NET Core?

A pipeline de requisição no ASP.NET Core é composta por uma série de middleware, que são componentes que podem manipular solicitações recebidas e gerar respostas. Cada middleware na pipeline pode decidir se passa a solicitação para o próximo middleware ou se interrompe o processamento da solicitação e retorna uma resposta diretamente. Isso permite uma grande flexibilidade na maneira como as solicitações são processadas.

ENGLISH: The ASP.NET Core request pipeline is composed of a series of middleware, which are components that can handle incoming requests and generate responses. Each middleware in the pipeline can decide whether to pass the request on to the next middleware or stop processing the request and return a response directly. This allows great flexibility in the way requests are processed.

43. O que é um token JWT e qual a sua estrutura?

Um token JWT (JSON Web Token) é uma maneira compacta e segura de representar informações entre duas partes. Ele é estruturado em três partes: o cabeçalho, o payload e a assinatura.

- O cabeçalho contém metadados sobre o token, incluindo o tipo do token e o algoritmo de assinatura usado.
- O payload contém as reivindicações ou dados que estão sendo transmitidos.
- A assinatura é usada para verificar se o token não foi alterado no trânsito.

Cada parte é codificada em Base64 e separada por pontos.

ENGLISH: A JWT (JSON Web Token) is a compact and secure way to represent information between two parties. It is structured in three parts: the header, the payload, and the signature.

- The header contains metadata about the token, including the type of the token and the signature algorithm used.
- The payload contains the claims or data that are being transmitted.
- The signature is used to verify that the token has not been altered in transit.

Each part is encoded in Base64 and separated by dots.