FullStack.Cafe - Kill Your Tech Interview

Q1: What is .NET Core? ☆

Topics: .NET Core

Answer:

The .NET Core platform is a new .NET stack that is optimized for open source development and agile delivery on NuGet.

.NET Core has two major components. It includes a small runtime that is built from the same codebase as the .NET Framework CLR. The .NET Core runtime includes the same GC and JIT (RyuJIT), but doesn't include features like Application Domains or Code Access Security. The runtime is delivered via NuGet, as part of the ASP.NET Core package.

.NET Core also includes the base class libraries. These libraries are largely the same code as the .NET Framework class libraries, but have been factored (removal of dependencies) to enable to ship a smaller set of libraries. These libraries are shipped as `System.*` NuGet packages on NuGet.org.

Q2: What is the difference between `String` and `string` in C#? ☆

Topics: .NET Core

Answer:

`string` is an *alias* in C# for `System.String`. So technically, there is no difference. It's like `int` vs. `System.Int32`.

As far as guidelines, it's generally recommended to use `string` any time you're referring to an object.

```
string place = "world";
```

Likewise, it's generally recommended to use `String` if you need to refer specifically to the class.

```
string greet = String.Format("Hello {0}!", place);
```

Q3: What is the .NET Framework? ☆

Topics: .NET Core

Answer:

The .NET is a Framework, which is a collection of classes of reusable libraries given by Microsoft to be used in other .NET applications and to develop, build and deploy many types of applications on the Windows platform including the following:

- Console Applications
- Windows Forms Applications

- Windows Presentation Foundation (WPF) Applications
- Web Applications
- Web Services
- Windows Services
- Services-oriented applications using Windows Communications Foundation (WCF)
- Workflow-enabled applications using Windows Workflow Foundation(WF)

Q4: What is .NET Standard? ☆

Topics: .NET Core

Answer:

The **.NET Standard** is a formal specification of .NET APIs that are intended to be available on all .NET implementations.

Q5: What are some characteristics of .NET Core? ☆☆

Topics: .NET Core

Answer:

- **Flexible deployment**: Can be included in your app or installed side-by-side user- or machine-wide.

- **Cross-platform**: Runs on Windows, macOS and Linux; can be ported to other OSes. The supported Operating Systems (OS), CPUs and application scenarios will grow over time, provided by Microsoft, other companies, and individuals.

- **Command-line tools**: All product scenarios can be exercised at the command-line.

- **Compatible**: .NET Core is compatible with .NET Framework, Xamarin and Mono, via the .NET Standard Library.

- **Open source**: The .NET Core platform is open source, using MIT and Apache 2 licenses. Documentation is licensed under CC-BY. .NET Core is a .NET Foundation project.

- **Supported by Microsoft**: .NET Core is supported by Microsoft, per .NET Core Support

Q6: What is the difference between .NET Core and Mono? ☆☆

Topics: .NET Core

Answer:

To be simple:

- Mono is third party implementation of .Net Framework for Linux/Android/iOs
- .Net Core is Microsoft's own implementation for same.

Q7: What's the difference between *SDK* and *Runtime* in .NET Core? ☆☆

Topics: .NET Core

Answer:

- The SDK is all of the stuff that is needed/makes developing a .NET Core application easier, such as the CLI and a compiler.

- The runtime is the "virtual machine" that hosts/runs the application and abstracts all the interaction with the base operating system.

Q8: What is the difference between `decimal`, `float` and `double` in .NET? ☆☆

Topics: .NET Core

Problem:

When would someone use one of these?

Solution:

Precision is the main difference.

- **Float** - 7 digits (32 bit)
- **Double** -15-16 digits (64 bit)
- **Decimal** -28-29 significant digits (128 bit)

As for what to use when:

- For values which are "naturally exact decimals" it's good to use **decimal**. This is usually suitable for any concepts invented by humans: financial values are the most obvious example, but there are others too. Consider the score given to divers or ice skaters, for example.

- For values which are more artefacts of nature which can't really be measured exactly anyway, **float/double** are more appropriate. For example, scientific data would usually be represented in this form. Here, the original values won't be "decimally accurate" to start with, so it's not important for the expected results to maintain the "decimal accuracy". Floating binary point types are much faster to work with than decimals.

Q9: What is an *unmanaged* resource in .NET? ☆☆

Topics: .NET Core

Answer:

Use that rule of thumb:

- If you found it in the Microsoft .NET Framework: it's **managed**.
- If you went poking around MSDN yourself, it's **unmanaged**.

Anything you've used P/Invoke calls to get outside of the nice comfy world of everything available to you in the .NET Framwork is unmanaged – and you're now *responsible* for cleaning it up.

Q10: What is *Boxing* and *Unboxing*? ☆☆

Topics: C# .NET Core

Answer:

Boxing and **Unboxing** both are used for type conversion but have some difference:

- **Boxing** - Boxing is the process of converting a value type data type to the object or to any interface data type which is implemented by this value type. When the CLR boxes a value means when CLR is converting a value type to Object Type, it wraps the value inside a `System.Object` and stores it on the heap area in application domain.

- **Unboxing** - Unboxing is also a process which is used to extract the value type from the object or any implemented interface type. Boxing may be done implicitly, but unboxing have to be explicit by code.

The concept of boxing and unboxing underlines the C# unified view of the type system in which a value of any type can be treated as an object.

Q11: What you understand by *Value types* and *Reference types* in .NET? Provide some comparison. ☆☆

Topics: C# .NET Core

Answer:

In C# data types can be of two types: **Value Types** and **Reference Types**. For a value type, the value is the information itself. For a reference type, the value is a reference which may be null or may be a way of navigating to an object containing the information.

- **Value types** - Holds some value not memory addresses. Example - Struct. A variable's value is stored wherever it is decleared. Local variables live on the stack for example, but when declared inside a class as a member it lives on the heap tightly coupled with the class it is declared in.

- *Advantages:** A value type does not need extra garbage collection. It gets garbage collected together with the instance it lives in. Local variables in methods get cleaned up upon method leave.

- *Drawbacks:**

1. When large set of values are passed to a method the receiving variable actually copies so there are two redundant values in memory.
2. As classes are missed out.it losses all the oop benifits

- **Reference type** - Holds a memory address of a value not value. Example - Class. Stored on heap

- *Advantages:**

1. When you pass a reference variable to a method and it changes it indeed changes the original value whereas in value types a copy of the given variable is taken and that's value is changed.
2. When the size of variable is bigger reference type is good
3. As classes come as a reference type variables, they give reusability, thus benefitting Object-oriented programming

Drawbacks: More work referencing when allocating and dereferences when reading the value.extra overload for garbage collector

Q12: What is CLR? ☆☆

Topics: .NET Core

Answer:

The **CLR** stands for Common Language Runtime and it is an Execution Environment. It works as a layer between Operating Systems and the applications written in .NET languages that conforms to the Common Language Specification (CLS). The main function of Common Language Runtime (CLR) is to convert the Managed Code into native code and then execute the program.

Q13: Name some CLR services? ☆☆

Topics: .NET Core

Answer:

CLR services

- Assembly Resolver
- Assembly Loader
- Type Checker
- COM marshalled
- Debug Manager
- Thread Support
- IL to Native compiler
- Exception Manager
- Garbage Collector

Q14: What is CTS? ☆☆

Topics: .NET Core

Answer:

The **Common Type System (CTS)** standardizes the data types of all programming languages using .NET under the umbrella of .NET to a common data type for easy and smooth communication among these .NET languages.

CTS is designed as a singly rooted object hierarchy with `System.Object` as the base type from which all other types are derived. CTS supports two different kinds of types:

1. **Value Types**: Contain the values that need to be stored directly on the stack or allocated inline in a structure. They can be built-in (standard primitive types), user-defined (defined in source code) or enumerations (sets of enumerated values that are represented by labels but stored as a numeric type).
2. **Reference Types**: Store a reference to the value‘s memory address and are allocated on the heap. Reference types can be any of the pointer types, interface types or self-describing types (arrays and class types such as user-defined classes, boxed value types and delegates).

Q15: What is a .NET application domain? ☆☆

Topics: .NET Core

Answer:

It is an isolation layer provided by the .NET runtime. As such, App domains live with in a process (1 process can have many app domains) and have their own virtual address space.

App domains are useful because:

- They are less expensive than full processes
- They are multithreaded
- You can stop one without killing everything in the process
- Segregation of resources/config/etc
- Each app domain runs on its own security level

Q16: What is MSIL? ☆☆

Topics: .NET Core

Answer:

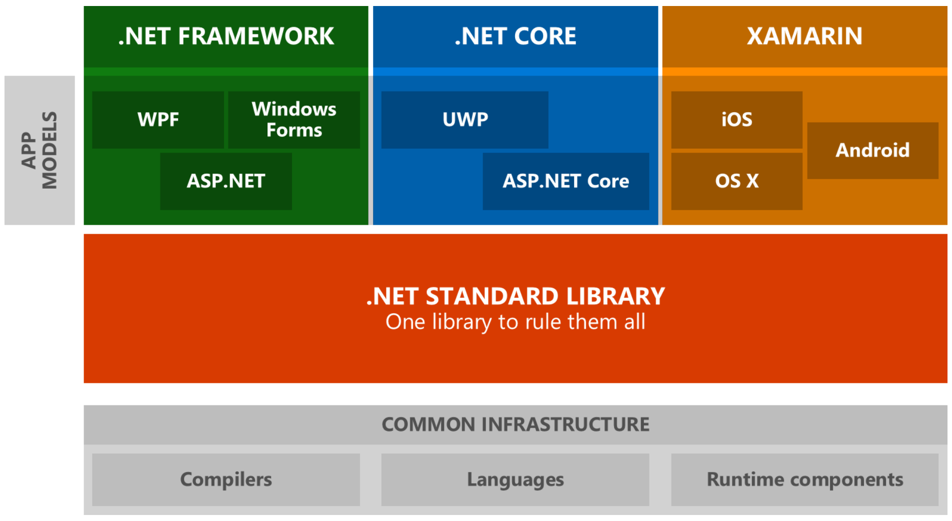
When we compile our .NET code then it is not directly converted to native/binary code; it is first converted into intermediate code known as MSIL code which is then interpreted by the CLR. MSIL is independent of hardware and the operating system. Cross language relationships are possible since MSIL is the same for all .NET languages. MSIL is further converted into native code.

Q17: What is .NET Standard and why we need to consider it? ☆☆

Topics: .NET Core

Answer:

1. **.NET Standard** solves the code sharing problem for .NET developers across all platforms by bringing all the APIs that you expect and love across the environments that you need: desktop applications, mobile apps & games, and cloud services:
2. **.NET Standard** is a **set of APIs** that **all** .NET platforms **have to implement**. This **unifies the .NET platforms** and **prevents future fragmentation**.
3. **.NET Standard 2.0** will be implemented by **.NET Framework**, **.NET Core**, and **Xamarin**. For **.NET Core**, this will add many of the existing APIs that have been requested.
4. **.NET Standard 2.0** includes a compatibility shim for **.NET Framework** binaries, significantly increasing the set of libraries that you can reference from your .NET Standard libraries.
5. **.NET Standard will replace Portable Class Libraries (PCLs)** as the tooling story for building multi-platform .NET libraries.



Q18: What is IoC (DI) Container? ☆☆

Topics: .NET Core Dependency Injection

Answer:

A **Dependency Injection** container, sometimes, referred to as **DI container** or **IoC container**, is a framework that helps with DI. It

- *creates* and
- *injects*

dependencies for us automatically.

Q19: What is *Generic Host* in .NET Core? ☆☆

Topics: ASP.NET .NET Core

Answer:

The .NET Generic Host is a feature which sets up some convenient patterns for an application including those for dependency injection (DI), logging, and configuration. It was originally named Web Host and intended for Web scenarios like ASP.NET Core applications but has since been generalized (hence the rename to *Generic* Host) to support other scenarios, such as Windows services, Linux daemon services, or even a console app.

A *host* is an object that encapsulates an app’s resources, such as:

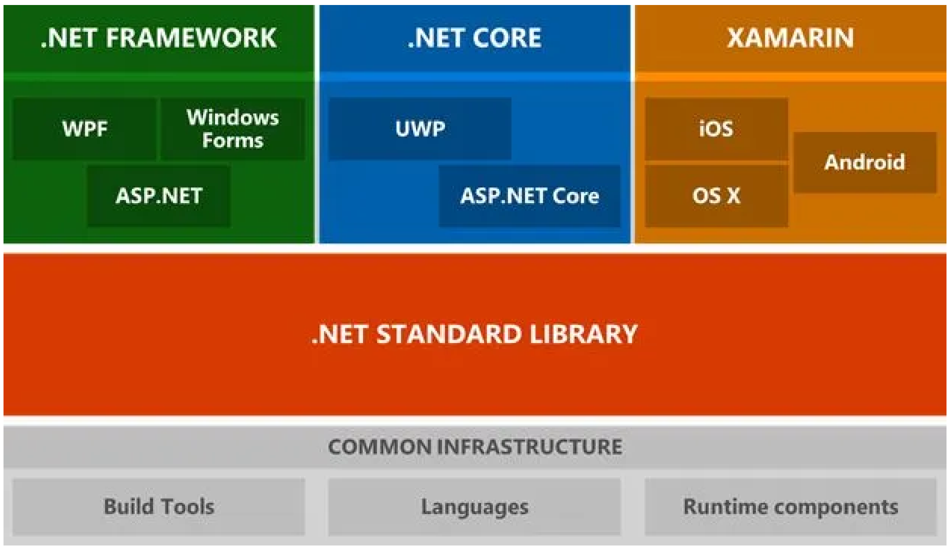
- Dependency injection (DI)
- Logging
- Configuration
- `IHostedService` implementations

Q20: What's the difference between .NET Core, .NET Framework, and Xamarin? ☆☆☆

Topics: .NET Core

Answer:

- **.NET Framework** is the "full" or "traditional" flavor of .NET that's distributed with Windows. Use this when you are building a desktop Windows or UWP app, or working with older ASP.NET 4.6+.
- **.NET Core** is cross-platform .NET that runs on Windows, Mac, and Linux. Use this when you want to build console or web apps that can run on any platform, including inside Docker containers. This does not include UWP/desktop apps currently.
- **Xamarin** is used for building mobile apps that can run on iOS, Android, or Windows Phone devices.



Xamarin usually runs on top of Mono, which is a version of .NET that was built for cross-platform support before Microsoft decided to officially go cross-platform with .NET Core. Like Xamarin, the Unity platform also runs on top of Mono.

FullStack.Cafe - Kill Your Tech Interview

Q1: What is Kestrel? ☆☆☆

Topics: .NET Core

Answer:

- Kestrel is a cross-platform web server built for ASP.NET Core based on libuv – a cross-platform asynchronous I/O library.
- It is a default web server pick since it is used in all ASP.NET Core templates.
- It is really fast.
- It is secure and good enough to use it without a reverse proxy server. However, it is still recommended that you use IIS, Nginx or Apache or something else.

Q2: What is difference between .NET Core and .NET Framework? ☆☆☆

Topics: .NET Core

Answer:

.NET as whole now has 2 flavors:

- .NET Framework
- .NET Core

.NET Core and the *.NET Framework* have (for the most part) a subset-superset relationship. .NET Core is named "Core" since it contains the core features from the .NET Framework, for both the runtime and framework libraries. For example, .NET Core and the .NET Framework share the GC, the JIT and types such as `String` and `List`.

.NET Core was created so that .NET could be open source, cross platform and be used in more resource-constrained environments.

Q3: Explain what is included in .NET Core? ☆☆☆

Topics: .NET Core

Answer:

- A .NET runtime, which provides a type system, assembly loading, a garbage collector, native interop and other basic services.

- A set of framework libraries, which provide primitive data types, app composition types and fundamental utilities.

- A set of SDK tools and language compilers that enable the base developer experience, available in the .NET Core SDK.

- The 'dotnet' app host, which is used to launch .NET Core apps. It selects the runtime and hosts the runtime, provides an assembly loading policy and launches the app. The same host is also used to launch SDK tools in much the same way.

Q4: Explain two types of deployment for .NET Core applications ☆☆☆

Topics: .NET Core

Answer:

- **Framework-dependent deployment (FDD)** - it relies on the presence of a shared system-wide version of .NET Core on the target system. The app contains only its own code and any third-party dependencies that are outside of the .NET Core libraries. FDDs contain .dll files that can be launched by using the dotnet utility from the command line.

  ```
  dotnet app.dll
  ```

- **Self-contained deployment** - Unlike FDD, a self-contained deployment (SCD) doesn't rely on the presence of shared components on the target system. All components, including both the .NET Core libraries and the .NET Core runtime, are included with the application and are isolated from other .NET Core applications. SCDs include an executable (such as app.exe on Windows platforms for an application named app), which is a renamed version of the platform-specific .NET Core host, and a .dll file (such as app.dll), which is the actual application.

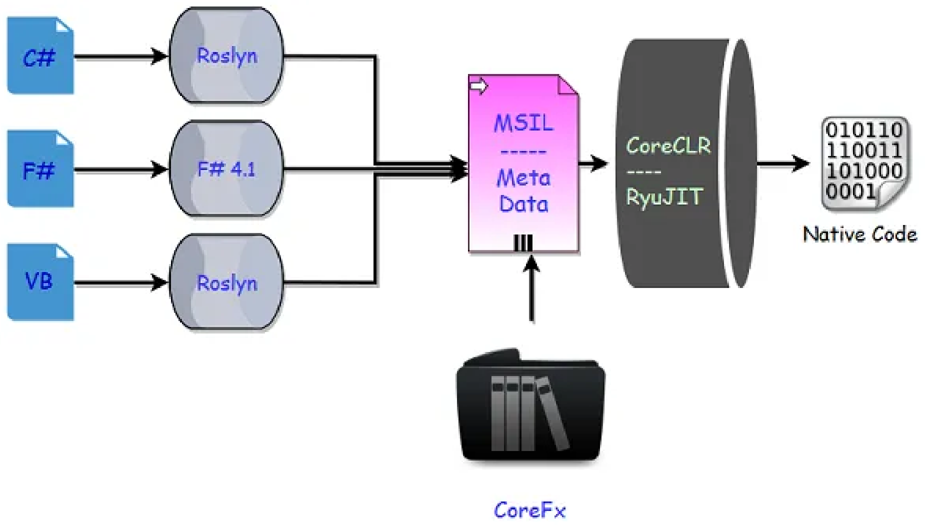
Q5: What is CoreCLR? ☆☆☆

Topics: .NET Core

Answer:

CoreCLR is the .NET execution engine in .NET Core, performing functions such as garbage collection and compilation to machine code.

Consider:



Q6: Is there a way to catch *multiple* exceptions at once and without code duplication? ☆☆☆

Topics: .NET Core C#

Problem:

Consider:

```
try
{
    WebId = new Guid(queryString["web"]);
}
catch (FormatException)
{
    WebId = Guid.Empty;
}
catch (OverflowException)
{
    WebId = Guid.Empty;
}
```

Is there a way to catch both exceptions and only call the `WebId = Guid.Empty` call once?

Solution:

Catch `System.Exception` and switch on the types:

```
catch (Exception ex)
{
    if (ex is FormatException || ex is OverflowException)
    {
        WebId = Guid.Empty;
        return;
    }

    throw;
}
```

Q7: Why to use of the `IDisposable` interface? ☆☆☆

Topics: .NET Core C#

Answer:

The "primary" use of the `IDisposable` interface is **to clean up unmanaged resource**s. Note the purpose of the Dispose pattern is to provide a mechanism to clean up both *managed* and *unmanaged* resources and when that occurs depends on how the Dispose method is being called.

Q8: Explain the difference between `Task` and `Thread` in .NET ☆☆☆

Topics: .NET Core C#

Answer:

- **Thread** represents an actual OS-level thread, with its own stack and kernel resources. Thread allows the highest degree of control; you can `Abort()` or `Suspend()` or `Resume()` a thread, you can observe its state, and you can set thread-level properties like the stack size, apartment state, or culture. `ThreadPool` is a wrapper around a pool of threads maintained by the CLR.

- The **Task class** from the Task Parallel Library offers the best of both worlds. Like the `ThreadPool`, a task does not create its own OS thread. Instead, tasks are executed by a `TaskScheduler`; the default scheduler simply runs on the ThreadPool. Unlike the ThreadPool, Task also allows you to find out when it finishes, and (via the generic Task) to return a result.

Q9: What does Common Language Specification (CLS) mean? ☆☆☆

Topics: .NET Core

Answer:

The **Common Language Specification (CLS)** is a fundamental set of language features supported by the Common Language Runtime (CLR) of the .NET Framework. CLS is a part of the specifications of the .NET Framework. CLS was designed to support language constructs commonly used by developers and to produce verifiable code, which allows all CLS-compliant languages to ensure the type safety of code. CLS includes features common to many object-oriented programming languages. It forms a subset of the functionality of common type system (CTS) and has more rules than defined in CTS.

Q10: Explain the difference between *Managed* and *Unmanaged* code in .NET? ☆☆☆

Topics: .NET Core

Answer:

- **Managed** code is not compiled to machine code but to an intermediate language which is interpreted and executed by some service on a machine and is therefore operating within a (hopefully!) secure framework which handles dangerous things like memory and threads for you. It runs on the CLR (Common Language Runtime), which, among other things, offers services like garbage collection, run-time type checking, and reference checking. So, think of it as, "My code is *managed* by the CLR."

- **Unmanaged code** is compiled to machine code and therefore executed by the OS directly. It therefore has the ability to do damaging/powerful things Managed code does not. This is how everything used to work, so typically it's associated with old stuff like .dlls

Q11: What is *Explicit Compilation*? ☆☆☆

Topics: .NET Core

Answer:

Explicit compilation converts the upper level language into object code prior to program execution. **Ahead of time (AOT)** compilers are designed to ensure that, the CPU can understand every line in the code before any interaction takes place.

Q12: What are the benefits of *Explicit Compilation (AOT)*? ☆☆☆

Topics: .NET Core

Answer:

Ahead of time (AOT) delivers **faster start-up time**, especially in large applications where much code executes on startup. But it requires more disk space and more memory/virtual address space to keep both the IL and precompiled images. In this case the JIT Compiler has to do a lot of disk I/O actions, which are quite expensive.

Q13: What is JIT compiler? ☆☆☆

Topics: .NET Core

Answer:

Before a computer can execute the source code, special programs called compilers must rewrite it into machine instructions, also known as object code. This process (commonly referred to simply as "compilation") can be done explicitly or implicitly.

Implicit compilation is a two-step process:

- The first step is converting the source code to intermediate language (IL) by a language-specific compiler.
- The second step is converting the IL to machine instructions. The main difference with the explicit compilers is that only executed fragments of IL code are compiled into machine instructions, at runtime. The .NET framework calls this compiler the **JIT (Just-In-Time) compiler**.

Q14: When should we use .NET Core and .NET Standard Class Library project types? ☆☆☆

Topics: .NET Core

Answer:

- Use a **.NET Standard** library when you want to increase the number of apps that will be compatible with your library, and you are okay with a decrease in the .NET API surface area your library can access.

- Use a **.NET Core** library when you want to increase the .NET API surface area your library can access, and you are okay with allowing only .NET Core apps to be compatible with your library.

Q15: What is the difference between Class Library (.NET Standard) and Class Library (.NET Core)? ☆☆☆

Topics: .NET Core

Answer:

- **Compatibility**: Libraries that target .NET Standard will run on any .NET Standard compliant runtime, such as .NET Core, .NET Framework, Mono/Xamarin. On the other hand, libraries that target .NET Core can only run on the .NET Core runtime.

- **API Surface Area**: .NET Standard libraries come with everything in `NETStandard.Library` whereas .NET Core libraries come with everything in `Microsoft.NETCore.App`. The latter includes approximately 20 additional libraries, some of which we can add manually to our .NET Standard library (such as `System.Threading.Thread`) and some of which are not compatible with the .NET Standard (such as `Microsoft.NETCore.CoreCLR`).

Q16: What is the difference between .NET Standard and PCL (Portable Class Libraries)? ☆☆☆

Topics: .NET Core

Answer:

.NET Standard and PCL profiles were created for similar purposes but also differ in key ways.

Similarities:

- Define APIs that can be used for binary code sharing.

Differences:

- .NET Standard is a curated set of APIs, while PCL profiles are defined by intersections of existing platforms.
- .NET Standard linearly versions, while PCL profiles do not. PCL profiles represents Microsoft platforms while the .NET Standard is platform-agnostic.

Q17: What's is BCL? ☆☆☆

Topics: .NET Core

Answer:

A .NET Framework library, BCL is the standard for the C# runtime library and one of the Common Language Infrastructure (CLI) standard libraries. BCL provides types representing the built-in CLI data types, basic file access, collections, custom attributes, formatting, security attributes, I/O streams, string manipulation, and more.

The Base Class Library (BCL) is literally that, the base. It contains basic, fundamental types like System.String and System.DateTime.

Q18: What is FCL? ☆☆☆

Topics: .NET Core

Answer:

The .NET Framework class library is exactly what its name suggests: a library of classes and other types that developers can use to make their lives easier. While these classes are themselves written in C#, they can be used from any CLR based language.

The **Framework Class Library (FCL)** is the wider library that contains the totality: ASP.NET, WinForms, the XML stack, ADO.NET and more. You could say that the FCL includes the BCL.

On a simple level, .NET Framework = libraries (FCL, BCL), language compilers (C#, VB.NET) and Common Language Runtime (CLR).

Q19: What about MVC in .NET Core? ☆☆☆

Topics: ASP.NET MVC .NET Core

Answer:

The ASP.NET Core MVC framework is a lightweight, open source, highly testable presentation framework optimized for use with ASP.NET Core.

ASP.NET Core MVC includes the following:

- Routing
- Model binding
- Model validation
- Dependency injection
- Filters
- Areas
- Web APIs
- Testability
- Razor view engine
- Strongly typed views
- Tag Helpers
- View Components

Q20: Explain the IoC (DI) Container service lifetimes? ☆☆☆

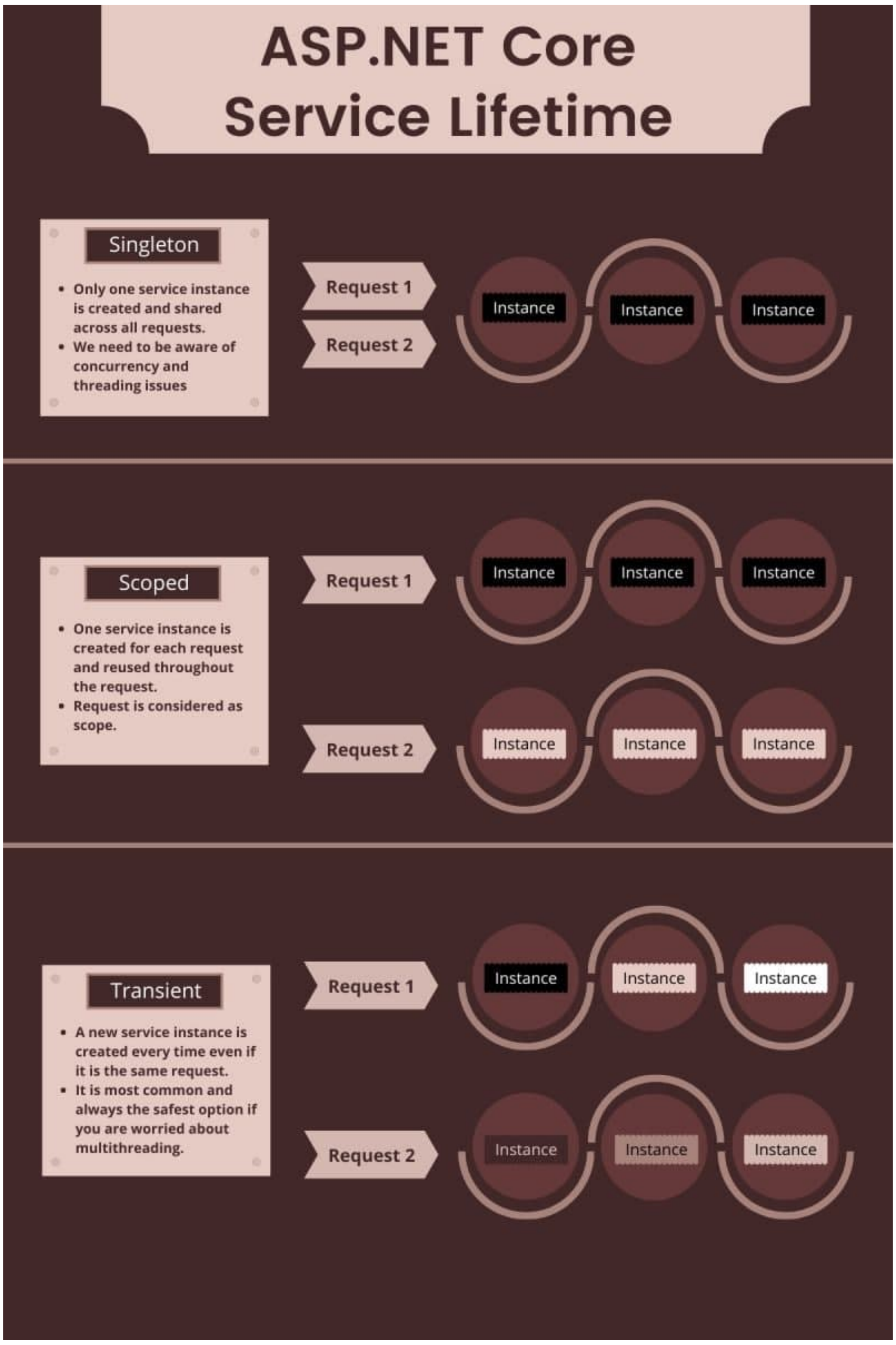
Topics: .NET Core Dependency Injection

Answer:

When we register services in a container, we need to set the lifetime that we want to use. The service lifetime controls how long a result object will live after it has been created by the container.

There are three lifetimes that can be used with Microsoft Dependency Injection Container, they are:

- **Transient** — Services are created **each time they are requested**. It gets a new instance of the injected object, on each request of this object. For each time you inject this object is injected in the class, it will create a new instance.
- **Scoped** — Services are created **on each request** (once per request). This is most recommended for WEB applications. So for example, if during a request you use the same dependency injection, in many places, you will use the same instance of that object, it will make reference to the same memory allocation.
- **Singleton** — Services are created **once for the lifetime of the application**. It uses the same instance for the whole application.


ASP.NET Core Service Lifetime
Singleton
• Only one service instance is created and shared across all requests.
• We need to be aware of concurrency and threading issues
Request 1
Request 2
Instance
Instance
Instance
Scoped
• One service instance is created for each request and reused throughout the request.
• Request is considered as scope.
Request 1
Instance
Instance
Instance
Request 2
Instance
Instance
Instance
Transient
• A new service instance is created every time even if it is the same request.
• It is most common and always the safest option if you are worried about multithreading.
Request 1
Instance
Instance
Instance
Request 2
Instance
Instance
Instance

FullStack.Cafe - Kill Your Tech Interview

Q1: How can you create your own *Scope* for a *Scoped* object in .NET? ☆☆☆

Topics: .NET Core ASP.NET Dependency Injection

Answer:

Scoped lifetime actually means that within a created “scope” objects will be the same instance. It just so happens that within .Net Core, it wraps a request within a “scope”, but you can actually create scopes manually. For example :

Consider:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped<IMyScopedService, MyScopedService>();
    var serviceProvider = services.BuildServiceProvider();
    var serviceScopeFactory = serviceProvider.GetRequiredService<IServiceScopeFactory>();
    IMyScopedService scopedOne;
    IMyScopedService scopedTwo;
    using (var scope = serviceScopeFactory.CreateScope())
    {
        scopedOne = scope.ServiceProvider.GetService<IMyScopedService>();
    }
    using (var scope = serviceScopeFactory.CreateScope())
    {
        scopedTwo = scope.ServiceProvider.GetService<IMyScopedService>();
    }

    Debug.Assert(scopedOne != scopedTwo);
}
```

In this example, the two scoped objects aren’t the same because created each object within their own *scope*.

Q2: What are some benefits of using *Options Pattern* in ASP.NET Core? ☆☆☆

Topics: .NET Core ASP.NET

Answer:

The options pattern uses classes to provide strongly typed access to groups of related settings. When configuration settings are isolated by scenario into separate classes, the app adheres to two important software engineering principles:

- The Interface Segregation Principle (ISP) or Encapsulation: Scenarios (classes) that depend on configuration settings depend only on the configuration settings that they use.
- Separation of Concerns: Settings for different parts of the app aren't dependent or coupled to one another.

Q3: What officially replaces WCF in .Net Core? ☆☆☆

Topics: .NET Core WCF

Answer:

You can use **gRPC** for hosting web services inside .NET core application.

1. gRPC is a high performance, open-source RPC framework initially developed by Google.
2. The framework is based on a client-server model of remote procedure calls. A client application can directly call methods on a server application as if it was a local object.

Q4: What is the correct pattern to implement long running background work in Asp.Net Core? ☆☆☆

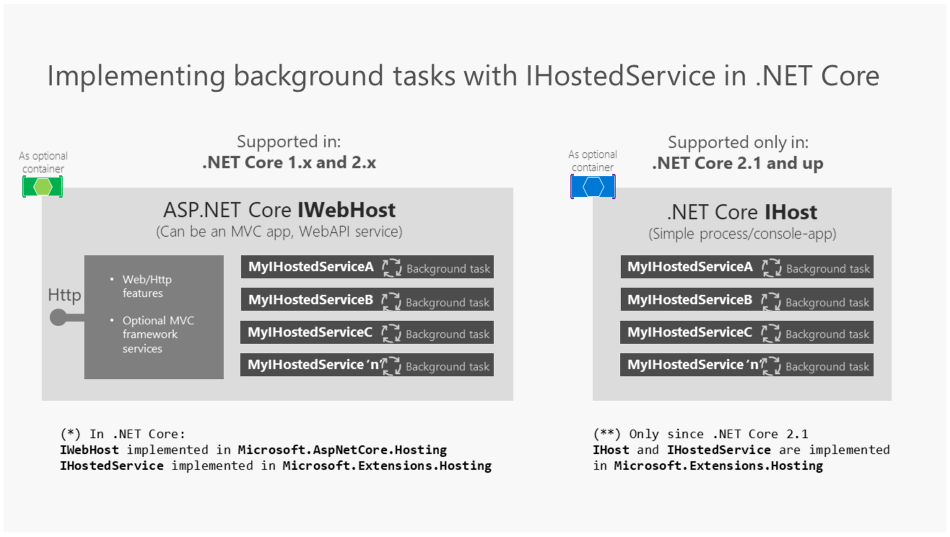
Topics: .NET Core ASP.NET

Answer:

You should certainly *not* use `Task.Run` / `StartNew` / `LongRunning` - that approach has *always* been wrong.

- Note that your long-running work may be shut down at any time, and that's normal. If you need a more reliable solution, then you should have a separate background system outside of ASP.NET (e.g., Azure functions / AWS lambdas).

- Since .NET Core 2.0, the framework provides a new interface named `IHostedService` helping you to easily implement hosted services. Hosted Services are services/logic that you host *within* your host/application/microservice. The `IHostedService` background task execution is coordinated with the lifetime of the application. When you register an `IHostedService`, .NET will call the `StartAsync()` and `StopAsync()` methods of your `IHostedService` type during application start and stop respectively.


Implementing background tasks with IHostedService in .NET Core
As optional container
Supported in: .NET Core 1.x and 2.x
ASP.NET Core IWebHost
(Can be an MVC app, WebAPI service)
Http
Web/Http features
Optional MVC framework services
MyIHostedServiceA Background task
MyIHostedServiceB Background task
MyIHostedServiceC Background task
MyIHostedService 'n' Background task
(*) In .NET Core:
IWebHost implemented in Microsoft.AspNetCore.Hosting
IHostedService implemented in Microsoft.Extensions.Hosting
As optional container
Supported only in: .NET Core 2.1 and up
.NET Core IHost
(Simple process/console-app)
MyIHostedServiceA Background task
MyIHostedServiceB Background task
MyIHostedServiceC Background task
MyIHostedService 'n' Background task
(**) Only since .NET Core 2.1
IHost and IHostedService are implemented in Microsoft.Extensions.Hosting

Q5: Explain the use of `BackgroundService` class in Asp.Net Core? ☆☆☆

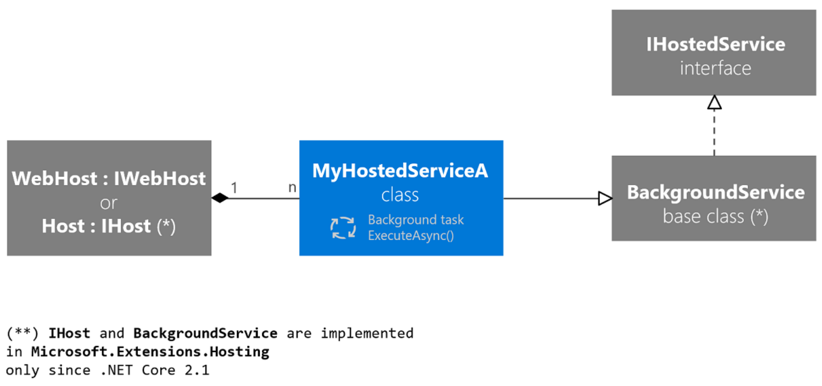
Topics: .NET Core ASP.NET

Answer:

Since .NET Core 2.0, the framework provides a new interface named IHostedService helping you to easily implement hosted services. The basic idea is that you can register multiple background tasks (hosted services) that run in the background while your web host or host is running.

Since most background tasks will have similar needs in regard to the cancellation tokens management and other typical operations, there is a convenient abstract base class you can derive from, named `BackgroundService` (available since .NET Core 2.1).

Class diagram with a custom IHostedService and related classes and interfaces



Consider the following simplified code which is polling a database and publishing integration events into the Event Bus when needed.

```
public class GracePeriodManagerService : BackgroundService
{
    private readonly ILogger<GracePeriodManagerService> _logger;
    private readonly OrderingBackgroundSettings _settings;

    private readonly IEventBus _eventBus;

    public GracePeriodManagerService(IOptions<OrderingBackgroundSettings> settings,
                                     IEventBus eventBus,
                                     ILogger<GracePeriodManagerService> logger)
    {
        // Constructor's parameters validations...
    }

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        _logger.LogDebug($"GracePeriodManagerService is starting.");

        stoppingToken.Register(() =>
            _logger.LogDebug($" GracePeriod background task is stopping."));

        while (!stoppingToken.IsCancellationRequested)
        {
            _logger.LogDebug($"GracePeriod task doing background work.");

            // This eShopOnContainers method is querying a database table
            // and publishing events into the Event Bus (RabbitMQ / ServiceBus)
            CheckConfirmedGracePeriodOrders();

            await Task.Delay(_settings.CheckUpdateTime, stoppingToken);
        }

        _logger.LogDebug($"GracePeriod background task is stopping.");
    }

    .../...
}
```

Q6: What is the difference between .NET Framework/Core and .NET Standard Class Library project types? ☆☆☆☆

Topics: .NET Core

Answer:

Use a .NET Standard library when you want to increase the number of apps that will be compatible with your library, and you are okay with a decrease in the .NET API surface area your library can access.

Implementing a .Net Standard Library allows code sharing across all different flavours of .NET applications like .NET Framework, .NET Core and Xamarin.

Q7: What's the difference between RyuJIT and Roslyn? ☆☆☆☆

Topics: .NET Core

Answer:

- **Roslyn** is the compiler that compile your code (C# or VB) to IL.
- **RyuJIT** is a Just In Time compiler that compile your IL to a native code.

Both of them are now open source.

Q8: Explain how does Asynchronous tasks `Async/Await` work in .NET? ☆☆☆☆

Topics: .NET Core C#

Problem:

Consider:

```
private async Task<bool> TestFunction()
{
  var x = await DoesSomethingExists();
  var y = await DoesSomethingElseExists();
  return y;
}
```

Does the second `await` statement get executed right away or after the first `await` returns?

Solution:

`await` pauses *the method* until the operation completes. So the second `await` would get executed after the first `await` returns.

The purpose of `await` is that it will return the current thread to the thread pool while the awaited operation runs off and does whatever.

This is particularly useful in high-performance environments, say a web server, where a given request is processed on a given thread from the overall thread pool. If we don't await, then the given thread processing the request (and all it's resources) remains "in use" while the db / service call completes. This might take a couple of seconds or more especially for external service calls.

Q9: What is the difference between `AppDomain`, `Assembly`, `Process` and a `Thread`? ☆☆☆☆

Topics: .NET Core

Answer:

- An **AppDomain** is an isolation unit within a process. AppDomains can be created at runtime, loaded with code, and unloaded. Its an isolation boundary designed to make .NET apps more reliable.
- An **assembly** holds one or more modules, which hold compiled chunks of code. You will typically see an assembly as an .EXE or a .DLL.
- A **process** is an executing application (waaaay oversimplified).
- A **thread** is an execution context. The operating system executes code within a thread. The operating system switches between threads, allowing each to execute in turn, thus giving the impression that multiple applications are running at the same time.

To put it all together (very simplified):

- A program is executed.
- A process is created by the operating system, and within its single thread it starts loading code to execute.
- In a .NET application, a single AppDomain is created by the CLR.
- The application's executing assembly (the .EXE) is loaded into this AppDomain and begins execution.
- The application can spawn new processes, create AppDomains, load other assemblies into these domains, and then create new Threads to execute code in any of these AppDomains.

Q10: What is the difference between CIL and MSIL (IL)? ☆☆☆☆

Topics: .NET Core

Answer:

CIL is the term used in the CLI Standard. MSIL is (I suppose) CIL created by MS tools. Effectively they are synonymous.

- CIL – Common Intermediate Language – is the term used in the International Standard.
- MSIL – Microsoft Intermediate Language – is the product term for the Microsoft implementation of that standard.

Q11: Why does .NET use a JIT compiler instead of just compiling the code once on the target machine? ☆☆☆☆

Topics: .NET Core

Answer:

There are two things to be gained by using an intermediate format like .NET or Java:

1. You can run the program on any platform, exactly because the code is represented in an intermediate format instead of native code. You just need to write an interpreter for the intermediate format.
2. It allows for some run-time optimizations which are not (easily) possible at compile-time: for example, you can take advantage of special features on new CPUs, even if those CPUs didn't exist when you wrote your program - only the JIT compiler needs to know about that.

Q12: Explain *Implicit Compilation* process ☆☆☆☆

Topics: .NET Core

Answer:

Implicit Compilation is a two-steps process, and it requires a Virtual Machine to be able to execute your code.

- The first step of the process is converting your program to a bytecode understandable by Virtual Machine. .NET bytecode is called Common Intermediate Language or CIL. It is also known as Microsoft Intermediate Language (MSIL) or just Intermediate Language (IL).
- The second step is converting CIL code to a machine code running on metal. This is Virtual Machine's task. Common Language Runtime (.NET Virtual Machine) converts only executed CIL fragments into CPU instructions at runtime. The .NET framework calls this compiler the JIT (Just-In-Time) compiler.

Q13: What are benefits of using JIT? ☆☆☆☆

Topics: .NET Core

Answer:

- JIT can generate faster code, because it targets the current platform of execution. AOT compilation must target the lowest common denominator among all possible execution platforms.
- JIT can profile the application while it runs, and dynamically re-compile the code to deliver better performance in the hot path (the most used functions).

Q14: Why does .NET Standard library exist? ☆☆☆☆

Topics: .NET Core

Answer:

The reason that .NET Standard exists is for portability; it defines a set of APIs that .NET platforms agree to implement. Any platform that implements a .NET Standard is compatible with libraries that target that .NET Standard. One of those compatible platforms is .NET Core.

The .NET Standard library templates exist to run on multiple runtimes (at the expense of API surface area). Obversely, the .NET Core library templates exist to access more API surface area (at the expense of compatibility) and to specify a platform against which to build an executable.

Q15: How to choose the target version of .NET Standard library? ☆☆☆☆

Topics: .NET Core

Answer:

When choosing a .NET Standard version, you should consider this trade-off:

1. The higher the version, the more APIs are available to you.
2. The lower the version, the more platforms implement it.

It's recommended to target the lowest version of .NET Standard possible. So, after you find the highest .NET Standard version you can target, follow these steps:

1. Target the next lower version of .NET Standard and build your project.
2. If your project builds successfully, repeat step 1. Otherwise, retarget to the next higher version and that's the version you should use.

Q16: Does .NET support *Multiple Inheritance*? ☆☆☆☆

Topics: OOP .NET Core

Answer:

.NET does not support multiple inheritance directly because in .NET, a class cannot inherit from more than one class. .NET supports multiple inheritance through interfaces.

Q17: When to use *Transient* vs *Scoped* vs *Singleton* DI service lifetimes? ☆☆☆☆

Topics: .NET Core ASP.NET Dependency Injection

Answer:

Transient

- they are created every time *per code request*
- they will use **more memory** & Resources and can have a **negative** impact on performance
- a new instance is provided every time a service instance is requested whether it is in the scope of the same HTTP request or across different HTTP requests,
- use this for the **lightweight** service with little or **no state**
- if you're only doing computation, for instance, that can be transient scoped because you're not exhausting anything by having multiple copies
- use a transient scope unless you have a good, explicit reason to make it a singleton. That would be the reasons like maintaining state, utilizing limited resources efficiently

Scoped

- a better option when you want to maintain the state within a *request*,
- It is equivalent to a singleton in the current scope
- in MVC it creates one instance for each HTTP request, but it uses the same instance in the other calls within the same web request
- normally we will use this for SQL connection, which means it will create and dispose the SQL connection per request

Singleton

- memory leaks in these services will build up over time,
- also memory efficient as they are created once reused everywhere,
- use you when need to maintain an application-wide state,
- singletons are best when you're utilizing limited resources, like sockets and connections. If you end up having to create a new socket every time the service is injected (transient), then you'll quickly run out of sockets and then performance really will be impacted
- If something needs to persist past one particular operation, then transient won't work, because you'll have no state, because it will essentially start over each time it's injected

Application configuration or parameters, Logging Service caching of data is some of the examples where you can use singletons.

Q18: When using DI in Controller shall I call `IDisposable` on any injected service? ☆☆☆☆

Topics: .NET Core ASP.NET Dependency Injection

Problem:

Or will Autofac or other CoC Container Framework automatically know which objects in my call stack properly need to be disposed?

Solution:

So even if your `Controller` depends on dependencies that need to be disposed of, your controller should not dispose of them:

- Because the consumer did not create such dependency, it has no idea what the expected lifetime of its dependency is. It could be very well that the dependency should outlive the consumer. Letting the consumer dispose of that dependency will in that case cause a bug in your application, because the next controller will get an already disposed of dependency, and this will cause an `ObjectDisposedException` to be thrown.
- Even if the lifestyle of the dependency equals that of the consumer, disposing is still a bad idea, because this prevents you from easily replacing that component for one that might have a longer lifetime in the future. Once you replace that component with a longer-lived component, you will have to go through all it consumers, possibly causing sweeping changes throughout the application. In other words, you will be violating the Open/closed principle by doing that–it should be possible to add or replace functionality without making sweeping changes instead.
- If your consumer is able to dispose of its dependency, this means that you implement `IDisposable` on that abstraction. This means this abstraction is _leaking implementation details_–that's a violation of the Dependency Inversion Principle. You are leaking implementation details when implementing `IDisposable` on an abstraction because it is very unlikely that *every implementation* of that abstraction needs deterministic disposal and so you defined the abstraction with a certain implementation in mind. The consumer should not have to know anything about the implementation, whether it needs deterministic disposal or not.
- Letting that abstraction implement `IDisposable` also causes you to violate the Interface Segregation Principle, because the abstraction now contains an extra method (i.e. `Dispose`), that not all consumers need to call. They might not need to call it, because –as I already mentioned– the resource might outlive the consumer. Letting it implement `IDisposable` in that case is dangerous because anyone can call `Dispose` on it causing the application to break. If you are more strict about testing, this also means you will have to test a consumer for *not calling* the `Dispose` method. This will cause extra test code. This is code that needs to be written and maintained.

So instead, you should *only* let the *implementation* implement `IDisposable`. In case your DI container creates this resource, it should also dispose it. DI Containers like Autofac will actually do this for you.

Q19: Why shouldn't I use the *Repository Pattern* with Entity Framework? ☆☆☆☆

Topics: .NET Core Design Patterns Entity Framework

Answer:

The single best reason to not use the repository pattern with Entity Framework? Entity Framework *already* implements a repository pattern. `DbContext` is your UoW (Unit of Work) and each `DbSet` is the repository. Implementing another layer on top of this is not only redundant but makes maintenance harder.

In the case of the repository pattern, the purpose is to abstract away the low-level database querying logic. In the old days of actually writing SQL statements in your code, the repository pattern was a way to move that SQL out of individual methods scattered throughout your codebase and localize it in one place. Having an ORM like Entity Framework, NHibernate, etc. is a *replacement* for this code abstraction, and as such, negates the need for the pattern.

However, it's not a bad idea to create an abstraction on top of your ORM, just not anything as complex as the UoW/repository. I'd go with a service pattern, where you construct an API that your application can use without knowing or caring whether the data is coming from Entity Framework, NHibernate, or a Web API. This is much simpler, as you merely add methods to your service class to return the data your application needs. The key difference here is that a service is a thinner layer and is geared towards returning fully-baked data, rather than something that you continue to query into, like with a repository.

Q20: What's the difference between gRPC and WCF? ☆☆☆☆

Topics: .NET Core WCF

Answer:

The differences between gRPC and WCF:

- GRPC does not use SOAP to mediate between client and service over http. WCF supports SOAP.
- GRPC is only concerned with RPC style communication. WCF supports and promotes REST and POX style services in addition to RPC.
- GRPC provides support for multiple programming languages. WCF supports C# (and the other .net languages).
- GRPC uses *protobuf* for on-wire serialization, WCF uses either XML/JSON or windows binary.
- GRPC is open source
- gRPC does not support Windows/Kerberos authentication

In short:

- GRPC seems a much more focused services framework, it does one job really well and on multiple platforms.
- WCF is much more general-purpose, but limited to .net for the time being (WCF is being ported to .net core but at time of writing only client side functionality is on .net core)

FullStack.Cafe - Kill Your Tech Interview

Q1: What is the difference between `IHost` vs `IHostBuilder` vs `IHostedService`? ☆☆☆☆

Topics: .NET Core ASP.NET

Answer:

Starting with ASP.NET Core 3.0:

- `IHost`: The host is the component that hosts and runs your application and its services. This is a generalization of the previous `IWebHost` but fullfills basically the same task: It starts configured hosted services and makes sure that your app is running and working.
- `IHostBuilder`: The host builder constructs the host and configures various services. This is the generalization of the previous `IWebHostBuilder` but also basically does the same just for generic `IHost`. It configures the host before the application starts. There is the `Host.CreateDefaultBuilder` method that will set up a host with various defaults, e.g. configuration using `appsettings.json` and logging.
- `IHostedService`: A hosted service is a central component that the *host* hosts. The most common example would be the ASP.NET Core server, which is implemented as a hosted service on top of the generic host. You can also write your own hosted services, or add third-party services that allow you to nicely add things to your application.

The generic host introduced with ASP.NET Core 3.0 and .NET Core 3.0 basically replaces the previous IWebHost and IWebHostBuilder. It follows the very same architecture and idea but is simply reduced to non-web tasks so that it can work with a number of different purposes. ASP.NET Core then just builds on top of this generic host.

Q2: What is the difference between *Hosted Services* vs *Windows Services*? ☆☆☆☆

Topics: ASP.NET .NET Core

Answer:

Windows Services are typically hosted on other infrastructure that isn’t also hosting your website. They can be deployed independently, and don’t have really any tie into your website. But that also comes as a negative. If you are using PAAS on something like Azure or Google Cloud, you would then need a separate VM to host your Windows Service.

Q3: Explain when to use `Finalize` vs `Dispose`? ☆☆☆☆☆

Topics: .NET Core C#

Answer:

- The **finalizer method** is called when your object is garbage collected and you have no guarantee when this will happen (you can force it, but it will hurt performance).

- The **Dispose method**, on the other hand, is meant to be called by the code that created your class so that you can clean up and release any resources you have acquired (unmanaged data, database connections, file handles, etc) the moment the code is done with your object.

The standard practice is to implement `IDisposable` and `Dispose` so that you can use your object in a `using` statement such as `using(var foo = new MyObject()) { }`. And in your finalizer, you call `Dispose`, just in case the calling code forgot to dispose of you.

Q4: What is the difference between Node.js async model and `async/await` in .NET? ☆☆☆☆☆

Topics: .NET Core

Answer:

The async model that has Node.js is similar to the old async model in C# and .Net called Event-based Asynchronous Pattern (EAP). The C#'s async/await keywords make asynchronous code linear and let you avoid "Callback Hell" much better then in any of other programming languages.

Node.js is asynchronously *single-threaded*, while ASP.NET is asynchronously *multi-threaded*. This means the Node.js code can make some simplifying assumptions because all your code always runs on the same exact thread. So when your ASP.NET code *awaits*, it could possibly resume on a different thread, and it's up to you to avoid things like thread-local state.

The same difference is also a strength for ASP.NET, because it means async ASP.NET can scale out-of-the-box up to the full capabilities of your sever. If you consider, say, an 8-core machine, then ASP.NET can process (the synchronous portions of) 8 requests simultaneously. If you put Node.js on a souped-up server, then it's common to actually run 8 separate instances of Node.js and add something like nginx or a simple custom load balancer that handles routing requests for that server.

Q5: How many types of JIT Compilations do you know? ☆☆☆☆☆

Topics: .NET Core

Answer:

There are **three types** of JIT compilation in the .NET framework:

1. **Pre-JIT** complies complete source code into native code in a single compilation cycle. In .NET languages, this is implemented in Ngen.exe (Native Image Generator). All CIL instructions are compiled to native code before startup. This way the runtime can use native images from the cache instead of invoking the JIT Compiler.
2. **Econo-JIT** complies only those methods that are called at runtime. However, these complied methods are removed when they are not required.
3. **Normal-JIT** complies only those methods that are called at runtime. These methods are complied the first time they are called, and then they are stored in cache. When the same methods are called again, the complied code from cache is used for execution.

Q6: Could you name the difference between .Net Core, Portable, Standard, Compact, UWP, and PCL? ☆☆☆☆☆

Topics: .NET Core

Answer:

The difference:

- **.Net Standard** (`netstandard`) - this is the new cross-platform BCL API. It's a "standard" in the sense that it's just an API definition and not an implementation. The idea is that you can compile your library to (a version of) this API and it will run on any platform that supports that version.

- **.Net Core** - you can think of this as a reference implementation of `netstandard` (with a few extra bits). It is a cross-platform *implementation* of that API. It is possible that UIs and other frameworks may build on it, but for now its only sure foothold is acting as the platform of choice for ASP.NET Core. [Side note: for historical reasons, ".NET Core" is *completely different* than the `netcore` NuGet target; when you're in a NuGet context, [`netcore` means "Windows 8/8.1/10"]][3].
- **.Net Portable** and **Portable Class Libraries** - Portable Class Libraries (PCLs) are a least-common-denominator approach to providing a cross-platform API. They cover a [wide range of target platforms][4], but they are incomplete and not future-proof. They have essentially been replaced by `netstandard`.
- **.Net Compact** - This is a completely different .NET framework with its own unique API. It is completely incompatible with any other framework, PCL, or `netstandard` version; as such, it is much more difficult to support than any other platform. However, it is still used on devices with tight memory constraints.
- **Universal Windows Platform** - This was a Win10-era merging of the API between Windows Phone and desktop, allowing Windows Store apps/libraries to be written for both platforms. This has essentially been replaced by `netstandard`.

Q7: Explain some deployment considerations for *Hosted Services* ☆☆☆☆☆

Topics: .NET Core ASP.NET

Answer:

- If you deploy your *WebHost* on IIS or a regular Azure App Service, your host can be shut down because of app pool recycles.
- But if you are deploying your host as a container into an orchestrator like Kubernetes, you can control the assured number of live instances of your host. In addition, you could consider other approaches in the cloud especially made for these scenarios, like Azure Functions.
- Finally, if you need the service to be running all the time and are deploying on a Windows Server you could use a Windows Service.