FullStack.Cafe - Kill Your Tech Interview

Q1: What is *Dependency Injection*? ☆☆

Topics: Design Patterns Dependency Injection

Answer:

Dependency injection makes it easy to create loosely coupled components, which typically means that components consume functionality defined by interfaces without having any first-hand knowledge of which implementation classes are being used.

Dependency injection makes it easier to change the behaviour of an application by changing the components that implement the interfaces that define application features. It also results in components that are easier to isolate for unit testing.

Q2: 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.

Q3: Why do I need an IoC container as opposed to straightforward DI code? ☆☆☆

Topics: Dependency Injection

Answer:

Compare this:

```
var svc = new ShippingService(new ProductLocator(),
   new PricingService(), new InventoryService(),
   new TrackingRepository(new ConfigProvider()),
   new Logger(new EmailLogger(new ConfigProvider())));
```

over this:

```
var svc = IoC.Resolve<IShippingService>();
```

Your dependencies chain can become nested, and it quickly becomes unwieldy to wire them up manually. Even with factories, the duplication of your code is just not worth it.

Q4: What is *Inversion of Control*? ☆☆☆

Topics: Dependency Injection

Answer:

The `Inversion-of-Control` **(IoC)** pattern, is about providing *any kind* of `callback` (which controls reaction), instead of acting ourself directly (in other words, inversion and/or redirecting control to the external handler/controller).

Inversion of Controls is also about separating concerns.

- **Without IoC**: You have a **laptop** computer and you accidentally break the screen. And darn, you find the same model laptop screen is nowhere in the market. So you're stuck.
- **With IoC**: You have a **desktop** computer and you accidentally break the screen. You find you can just grab almost any desktop monitor from the market, and it works well with your desktop.

Your desktop successfully implements IoC in this case. It accepts a variety type of monitors, while the laptop does not, it needs a specific screen to get fixed.

Q5: What exactly is IoC and DI? How are they related? ☆☆☆

Topics: Dependency Injection

Answer:

The `Inversion-of-Control` **(IoC)** pattern, is about providing *any kind* of `callback` (which "implements" and/or controls reaction), instead of acting ourselves directly (in other words, inversion and/or redirecting control to the external handler/controller).

> For example, rather than having the application call the implementations provided by a *library* (also know as *toolkit*), a *framework* calls the implementations provided by the application.

The `Dependency-Injection` **(DI)** pattern is a more specific version of IoC pattern, where implementations are passed into an object through constructors/setters/service lookups, which the object will 'depend' on in order to behave correctly.

> Every `DI` implementation can be considered `IoC`, but one should not call it `IoC`, because implementing Dependency-Injection is harder than callback (Don't lower your product's worth by using the general term "IoC" instead).

- **IoC without using DI**, for example, would be the Template pattern because the implementation can only be changed through sub-classing.
- **DI frameworks** are designed to make use of DI and can define interfaces (or Annotations in Java) to make it easy to pass in the implementations.
- **IoC containers** are DI frameworks that can work outside of the programming language. In some, you can configure which implementations to use in metadata files (e.g. XML) which are less invasive.

Q6: Explain the IoC (DI) Container service lifetimes? ☆☆☆

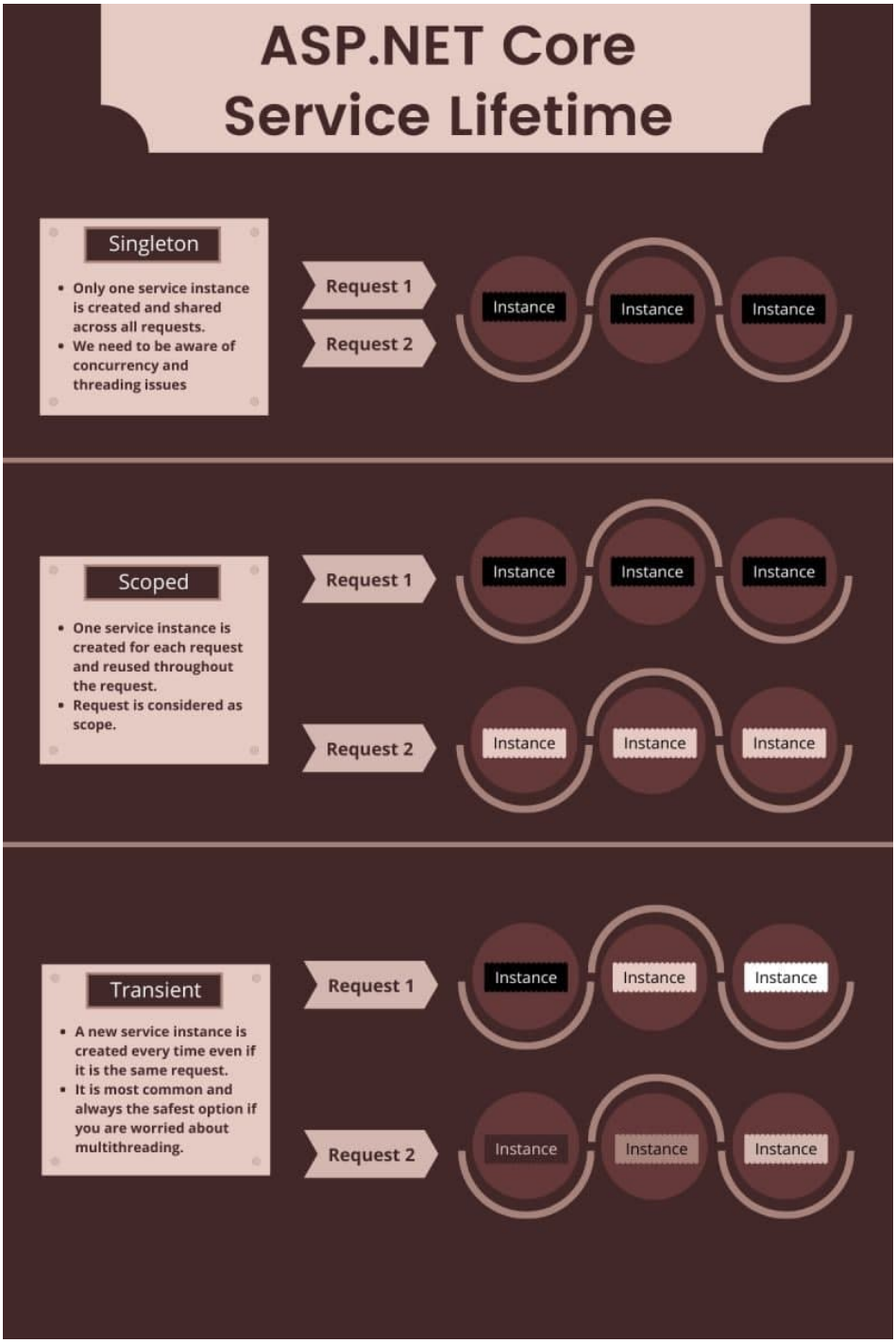
Topics: .NET Core Dependency Injection

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.



Q7: 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*.

Q8: What are some *advantages* of using Dependency Injection ☆☆☆

Topics: Design Patterns Dependency Injection

Answer:

Dependency injection is basically providing the objects that an object needs (its dependencies) instead of having it construct them itself. When using dependency injection, objects are given their dependencies *at run time rather than compile time (car manufacturing time)*.

- It allows your code to be more loosely coupled because classes do not have hard-coded dependencies
- Decoupling the creation of an object (in other words, separate usage from the creation of an object)
- Making isolation in unit testing possible/easy. It is harder to isolate components in unit testing without dependency injection.
- **Explicitly** defining dependencies of a class
- Facilitating good design like the single responsibility principle (SRP) for example
- Promotes *Code to an interface, not to implementation* principle
- Enabling switching/ability to replace dependencies/implementations quickly (`DbLogger` instead of `ConsoleLogger` for example)

Q9: Explain what is *Object Lifetime Management* when using DI? ☆☆☆☆

Topics: Dependency Injection

Answer:

Object lifetime management using DI container usually boils down to:

1. **Singleton** scoped (sometimes container scoped)
2. **Transient** (per request from the container)
3. **Scoped** per **scope** (context, web request) (example: presenter/controller in desktop applications, session in web applications/web servers)

In general, you're thread-safe with transients and singleton is good for performance reasons or state sharing. You also must make careful note of how your resources are being disposed.

Q10: What are some *best practices* for implementing services for DI? ☆☆☆☆

Topics: Dependency Injection

Answer:

When designing services for dependency injection:

- Avoid stateful, static classes and members. Avoid creating a global state by designing apps to use singleton services instead.
- Avoid direct instantiation of dependent classes within services. Direct instantiation couples the code to a particular implementation.
- Make services small, well-factored, and easily tested.

If a class *has many injected dependencies*, it might be a sign that the class has too many responsibilities and violates the Single Responsibility Principle (SRP). Attempt to refactor the class by moving some of its responsibilities into new classes.

Q11: 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.

Q12: 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.

Q13: What are some *disadvantages* of Dependency Injection? ☆☆☆☆

Topics: Design Patterns Dependency Injection

Answer:

Dependency injection is, like most patterns, a **solution to problems**. So start by asking if you even have the *problem* in the first place. If not, then using the pattern most likely will make the code *worse*.

So the problems you shall experience to consider the use of DI:

1. You know you'll be doing automated testing using mock implementations
2. You know you'll be doing unit testing using mock implementations to avoid hardcoded dependencies
3. You think you might have more than one implementation in the future or the implementation might change
4. You using global services with side effects like a logger and might change it later

Also, consider:

1. Using DI does not simplify **automated testing** as much as advertised. In short, the mock implementation does NOT let you validate that the class will behave as expected with a real implementation of the interface.
2. DI makes it much harder to **navigate through code**. The problem is often finding the actual implementation being used. This can turn a simple bug fix into a day-long effort (especially for a new person on the project).
3. The use of DI and frameworks comes at the expense of needing detailed knowledge of the particular DI framework. Understanding how dependencies are loaded into the framework and adding a new dependency into the framework to inject can require reading a fair amount of background material and time spent.
4. The more you inject, the longer your startup times are going to become. Most DI frameworks create all injectable singleton instances at startup time, regardless of where they are used.
5. Dependency injection, outside of cases where it is truly needed, is also a warning sign that other superfluous code may be present in great abundance. Executives are often surprised to learn that developers add complexity for the sake of complexity.

Q14: What is the *Dependency Inversion Principle (DIP)* and why is it important? ☆☆☆☆

Topics: Software Architecture Dependency Injection

Answer:

DIP reduces coupling between different pieces of code and says:

- High-level modules should not depend upon low-level modules. Both should depend upon abstractions.
- Abstractions should never depend upon details. Details should depend upon abstractions.

But why do we *invert* dependency?

- Basically, we ensure the most *stable* things are not dependent on less stable things, that might change more frequently.

Q15: Do I need *Dependency Injection* in Node.js and how to deal with it? ☆☆☆☆

Topics: Node.js Dependency Injection

Answer:

Due to the untyped, dynamic nature of JavaScript, you can actually do quite a lot without resorting to the dependency injection (DI) pattern or using a DI framework. The great thing about JS is that you can modify just about anything to achieve what you want. However, as an application grows larger and more complex, DI can definitely help the maintainability of your code.

Now the real question is, what can you achieve with a Node.js DI container, compared to a simple `require()` or `import`?

Pros:

- better testability: modules accepts their dependencies as input
- Inversion of Control: decide how to wire your modules without touching the main code of your application.
- a customizable algorithm for resolving modules: dependencies have "virtual" identifiers, usually they are not bound to a path on the filesystem.
- Better extensibility: enabled by IoC and "virtual" identifiers.
- Other fancy stuff possible:
  - Async initialization
  - Module lifecycle management
  - Extensibility of the DI container itself
  - Can easily implement higher level abstractions (e.g. AOP)

Cons:

- Different from the Node.js "experience": using DI definitely feels like you are deviating from the Node way of thinking.
- The relationship between a dependency and its implementation is not always explicit. A dependency may be resolved at runtime and influenced by various parameters. The code becomes more difficult to understand and debug
- Slower startup time
- Most DI containers will not play well with module bundlers like Browserify and Webpack.

As with anything related to software development, choosing between DI or `require()` / `import` depends on your requirements, your system complexity, and your programming style.

Q16: What's the difference between the *Dependency Injection* and *Service Locator* patterns? ☆☆☆☆☆

Topics: Design Patterns Dependency Injection

Answer:

- With the **ServiceLocator**, the class is still responsible for creating its dependencies. It just uses the service locator to do it.
- **Service locators** hide dependencies - you can't tell by looking at an object whether it hits a database or not (for example) when it obtains connections from a locator.
- With **DI**, the class is given it's dependencies. It neither knows, nor cares where they come from.

One important result of this is that the DI example is much easier to unit test -- because you can pass it mock implementations of its dependent objects. You could combine the two -- and inject the service locator (or a factory), if you wanted.

Q17: How to *Dispose* resources with Dependency Injection?

☆☆☆☆☆

Topics: Dependency Injection

Answer:

The general rule of resources is that:

> He who owns the resource is responsible of disposing of it.

This means that if a class owns a resource, it should either dispose of it in the same method that it created it in (in which case the disposable is called an ephemeral disposable), or if that's not possible, this generally means that the owning class must implement `IDisposable`, so it can dispose of the resource in its `Dispose` method.

But when a resource is injected, this means that this resource already existed before the consumer did. The consumer didn't create the resource and should in that case not dispose of it. Although you can pass on the ownership of the resource to the consumer (and communicate this in the class's documentation that ownership is passed on), in general you should not pass on the ownership, because this complicates your code and makes the application very fragile.

you should *only* let the *implementation* implement `IDisposable`. This frees any consumer of the abstraction from the doubts whether it should or shouldn't call `Dispose` (because there is no `Dispose` method to call on the abstraction).

Because only the implementation implements `IDisposable` and only your Composition Root creates this implementation, it is the Composition Root that is responsible for its disposal. In case your DI container creates this resource, it should also dispose it. DI Containers like Autofac will actually do this for you.