IServiceProvider vs IServiceScopeFactory

References: Net Core DI Truths

Internet’s Claim of Anti-Patterns

Contrary to what the internet says about service locator usage as anti-pattern, there will always be the need for logic to dynamically access services from DI.
This is because the NET Core runtime only provides automatic DI service injection for web requests. For all other use cases of logic that we may build (some listed below), we as developers, are responsible for our own dynamic service access and scoping.

This is especially true for singleton services.
For any singleton service that accesses datastores or use scoped/transient services, the NET Core runtime will throw an exception, when DI attempts to construct a singleton service that needs a scoped or transient service in its constructor.
So this use case alone, creates a reasonable need for logic to dynamically accessing services through a service provider.

Here’s a list of possible scenarios that need dynamic service access and disposal scoping:

- Event Handling of IO or socket data that is stored in a database
- Handling of received messages from a message broker that will be forwarded using a transient service
- A web request that spawns an async task to do post-processing after the web request returns.
- Singleton service that uses scoped or transient services

Basically, the above list is known examples of logic that executes outside of the lifetime of web requests.
And in any of the above scenarios, the logic may have a legitimate need to dynamically access services.

This is because the logic executing in each of the above scenarios, exists for its own lifetime.
So, passing service instances to the logic of these scenarios would hold those services captive, or be at risk of being disposed before usage.

Or, they may be explicitly disallowed by the runtime (for the case of singletons that use scoped/transient services).

Provider or Factory - Which to Pass?

The question becomes: Do we pass in an IServiceProvider or an IServiceScopeFactory?

The difference between these two is subtle, as they function very similar to each other.
And, showing their difference (the shortcoming of one, requiring the usage of the other) requires a specially crafted test that, like the use cases above, accesses DI-registered services in an asynchronous task/thread.

Short Answer

If you can reason that a service or an async task will remain active beyond the lifetime of whatever parent logic called it, then you need to pass in a service scope factory.

Or. If you can reason that a service will be ready for disposal or an async task will complete its logic before its parent logic, then the service scope of the parent logic may be safe to pass along.

Long Answer

If you have logic that satisfies all of the following, you should pass it an IServiceScopeFactory instance:

- The logic runs asynchronously to its calling parent (being a separate thread or an async task)
- The logic will live beyond the lifetime of its calling parent
- The logic will dynamically access DI services

If you have logic that satisfies the above requirements, then that logic or service needs to access DI through its own service scope.
And, the only way to safely provide a service scope is to pass a reference to the IServiceScopeFactory instance.
And since the IServiceScopeFactory instance is a runtime singleton and is registered with the app's DI, a reference to it can be retrieved from any IServiceProvider instance that your parent logic has access to.

Like this:

```
var scopeFactory = _serviceProvider.GetService<IServiceScopeFactory>();
```

Example.

Here’s a good example of an async task that will live beyond the service scope of its parent logic.

Say, we have a web request that spawns an async task, and, that task is meant to continue after the web request has returned, then the service scope the web request is under, can likely end up being disposed before the spawned async task finishes.

Here’s an example of one:

```
[HttpGet("/Echo/{word}")]
public IActionResult EchoAndLog([FromServices] IServiceProvider serviceProvider)
{
    var ipAddress = HttpContext.Connection.RemoteIpAddress;
    // No need to wait for logging, just fire and forget
    Task.Run(async () =>
    {
        await Task.Delay(1000);
        using (var scope = serviceProvider.CreateScope())
        {
            var context = scope.ServiceProvider.GetRequiredService<LogDbContext>();
            var log = new ActivityLog
            {
                IpAddress = ipAddress,
                Endpoint = "Echo",
                Parameter = word
            };

            context.Add(log);
            await context.SaveChangesAsync();
        }
    });
    return Ok(word);
}
```

The above is a rough example of an API endpoint that will trigger an async task to perform some action.

The web request may very well, return an Ok status to the caller before the async task has finished its work.
In which case, the `_serviceprovider` reference will likely be disposed during the async processing.
And, any used services inside the async task that are IDisposable, will as well, become disposed.

So, the async task will be the victim of a `System.ObjectDisposedException`.

Service Scoped Factory Usage

To prevent the above exception from happening, we need to recognize that an IServiceProvider instance passed into a controller, service, or method (from the DI container), where that controller, service, or method has been registered as transient or scoped, the IServiceProvider will have the same scope as the object it is passed into.
Meaning, when the web request (that needed controller, service, or method) returns, the service provider gets disposed.
And, we need to instead pass in a `IServiceScopeFactory`, instead of `IServiceProvider`.

Here's the above example, passing in the service scope factory, so the async task can create its own scope and service provider. Both of which, will be under disposal control by the using statements in the async task.

```
[HttpGet("/Echo/{word}")]
public IActionResult EchoAndLog([FromServices] IServiceScopeFactory serviceScopeFactory)
{
    var ipAddress = HttpContext.Connection.RemoteIpAddress;

    // No need to wait for logging, just fire and forget
    Task.Run(async () =>
    {
        await Task.Delay(1000);
        using (var scope = serviceScopeFactory.CreateScope())
        {
            var context = scope.ServiceProvider.GetRequiredService<LogDbContext>();
            var log = new ActivityLog
            {
                IpAddress = ipAddress,
                Endpoint = "Echo",
                Parameter = word
            };
```

```
            context.Add(log);
            await context.SaveChangesAsync();
        }
    });
    return Ok(word);
}
```

The above example is a slight revision from the previous one, where we've instead passed in the service scope factory singleton.
This allows the action endpoint to spawn an async task, and give it the service scope factory, so it can create scopes and services as it requires, and control their lifetime as necessary.

Revision #2
Created 8 October 2025 02:46:13 by glwhite
Updated 8 October 2025 02:53:32 by glwhite