Leveling Up Dependency Injection in .NET

5: DI Containers

Jeremy Clark
www.jeremybytes.com
@jeremybytes

Primary Benefits

- Late Binding
- Extensibility
- Parallel Development
- Maintainability
- Testability

- Adherence to S.O.L.I.D. Design Principles.

Dependency Injection Concepts

- DI Design Patterns
  - Constructor Injection
  - Property Injection
  - Method Injection
  - Ambient Context
  - Service Locator
- Dimensions of DI
  - Object Composition
  - Interception
  - Lifetime Management

Dependency Injection Containers

- C# Containers
  - Ninject
  - Autofac
  - Unity
  - Castle Windsor
  - Spring.NET
- Frameworks w/ Containers
  - ASP.NET Core
  - Angular
  - Prism

  and many others

Object Composition

- Composing objects should happen as close to the application entry point as possible.
- In a desktop application, this means application startup.
- In an ASP.NET MVC application, this means the start of the request (generally creation of the controller).
- For other web applications, the entry point may be framework specific.

Object Composition

- The composition root should be the ONLY place a DI container is used. If the container is used in other areas, this is a code smell that the code violates DI principles.
- This often happens when the Service Locator anti-pattern is used.

Lifetime Management

- Transient
- Singleton
- Scoped
- Thread (not as relevant as it used to be)

Transient Lifetime

- A new instance of a dependency is used whenever there is a request for that dependency.
- Each instance is independent and will get cleaned up / garbage collected as it goes out of scope.

Singleton Lifetime

- A single instance of a dependency is used whenever there is a request for that dependency.
- The lifetime is managed by the DI container. It may or may not be released when all references have been released.

Scoped Lifetime

- A new instance is used for each “scope” of an application.
- If a dependency is needed multiple times within the same scope, a single instance of that dependency is used.
- Scope example: In a web application, the scope generally refers to the current request.
- Container scopes can be explicitly defined.

Thread Lifetime

- A new instance is used for each thread of an application.
- This lifetime is less common due to an increase in asynchronous programming.
- Scoped lifetime is preferred over thread lifetime.

Interception

- Interception is used for cross-cutting concerns.
- By using a Decorator, an object can intercept calls to the underlying object and add its own behavior.
- Examples:
  - Auditing
  - Logging
  - Authorization
  - Caching

Interception – Authorization

```
public void UpdateOrder(Order order)
{
    if (!authorized)
      throw new NotAuthorized Exception;
    realUpdater.UpdateOrder(order);
}
```

Interception – Logging

```
public void UpdateOrder(Order order)
{
    LogMethodEntry();
    realUpdater.UpdateOrder(order);
    LogMethodExit();
}
```

Interception – Combining Decorators

- UpdateOrder on authorization decorator checks authorization and then calls…
- UpdateOrder on logging decorator logs the method entry and then calls…
- UpdateOrder on the real repository which returns to…
- UpdateOrder on logging decorator logs the method exit and then returns to…
- UpdateOrder On authorization decorator.

Configuring DI Containers

- Auto-Wiring (container can figure out concrete types)
- Auto-Registration
- Configuration as Code
- Configuration Files

Auto-Wiring

- The container can determine the object composition based on parameters and concrete types.
- This works as long as there are no abstract dependencies (such as interfaces or abstract classes).
- With abstract dependencies, additional configuration is required.

Auto-Registration

- Reflection is used to load an assembly and pull out the associated types. These types are registered with the container.
- Since this registration is at runtime, it allows for late binding / dynamic loading of types.
- Since there is no compile-time checking, this can lead to missing dependencies and other runtime errors.

Auto-Registration

- Auto-Registration works well when there is a naming convention, such as "…Repository" or "…Command".
- This also works well when each dependency implements only one interface.

Configuration as Code

- Abstractions are matched up to concrete types in code.
- This also allows for factory methods to be associated with particular types.
- Benefits include compile-time type checking.

Configuration Files

- Abstractions are matched to concrete types in configuration files (such as JSON or XML).
- This allows for runtime binding.
- The files can get complex quite quickly.
- There is no compile-time checking.
- There is no debugging.

Preferred Configuration

- Use Auto-Registration where possible
  - This leads to less configuration to maintain.
  - This works well when using name conventions.
- Prefer Configuration as Code
  - This allows for compile-time checking and easier debugging.
  - There is also flexibility when it comes to factory methods and other unusual bindings.
- Use Configuration Files sparingly.
  - These are the most brittle, but they are often the best solution for late binding.

Dependency Injection Containers

- ASP.NET MVC Core
- Ninject
- Autofac

ASP.NET MVC Core

Applies to ASP.NET MVC Core

- Built-in dependency injection
- Can be used as a stand-alone product, but a third-party container would be better.
- Supports Constructor Injection and Method Injection.
- Supports lifetime management.

ASP.NET MVC Core

Documentation:

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1

Ninject

Applies to .NET Applications (Framework & Core)

- Supports major DI features.
- Auto-wiring, configuration as code.
- Auto-registration requires manual reflection.
- Lifetime Management

Ninject

- Constructor Arguments
  - .WithConstructorArguments()
- Property Injection
  - .WithPropertyValue()
- Factory Method
  - .ToMethod<T>(c => myFactoryMethod())

Ninject

Documentation:

https://github.com/ninject/Ninject/wiki

Autofac

Applies to .NET Applications (Framework & Core)

- Supports major DI features.
- Auto-wiring, configuration as code.
- Auto-registration.
- Lifetime Management

Autofac

- Decorators
  - .RegisterDecorator<TDecorator>(…)
- Property Injection
  - .WithProperty<TPropertyType>(…)
- Late Binding (with configuration file)
  - .RegisterModule(new ConfigurationModule(config)

Autofac

Documentation:

https://autofaccn.readthedocs.io/en/latest/

Stable and Volatile Dependencies

- A stable dependency is one that is not likely to change over the life of the application. For example, classes in the .NET Base Class Library (BCL)
- A volatile dependency is one that is likely to change or needs to be swapped out for fake behavior in unit tests.

Criteria for Stable Dependencies

- The class or module already exists
- You expect that new versions won't contain breaking changes
- The types in question contain deterministic algorithms
- You never expect to have to replace, wrap, decorate, or intercept the class or module with another

Criteria for Volatile Dependencies

- The dependency introduces a requirement to set up or configure a runtime environment for the application
  - Web services, databases, network calls
- The dependency doesn't yet exist or is still in development

Criteria for Volatile Dependencies

- The dependency isn't installed on all machines in the development organization
  - Expensive 3rd party library
- The dependency contains non-deterministic behavior
  - Random number generator
  - DateTime.Now

Factory Methods

- Symptom: A class uses a factory method and has a private constructor.
- Problem: This breaks auto-wiring in DI containers.

Factory Methods

- Possible Solution:
  Most DI containers have a way to bind to a factory method.

- Ninject Example:

```
Container
    .Bind<ConcreteType>()
    .ToMethod(c => FactoryForConcreteType());
```

Configuration Strings

- Symptom: A class constructor needs a string as a parameter, such as a connection string.
- Problem: This breaks auto-wiring in DI containers.

Configuration Strings

- Possible Solution:
  Create a parameter object to hold the string. This gives a strongly-typed object that can be configured and resolved by the container.

- This is a preferred method since it gives additional type safety.

Configuration Strings

- Alternate Solution:
  Use the factory method syntax to inject the string manually.

- Ninject Example:

```
Container
    .Bind<ConcreteType>()
    .ToMethod(c => new ConcreteType(paramString))
```

More Information

https://github.com/jeremybytes/di-dotnet-workshop