Lab 03 - More Dependency Injection

Objectives

This lab shows how to use a dependency injection container. This specifically uses Ninject, but the concepts are applicable across containers. In addition to learning about containers, you will also see some techniques that let containers do more of the work for you while keeping code resilient when there are changes.

Application Overview

Please refer to the "Lab00-ApplicationOverview.md" file for an overview of the application, the projects, and how to run the application.

As a reminder, the "Starter" folder contains the code files for this lab.

> The state of the "Starter" files for Lab 03 are the same as the "Completed" files for Lab 02. So if you completed Lab 01 and Lab 02, you can continue coding in the same solution.

Objectives

1. Use the Ninject dependency injection container to create the `HouseController` at application startup.
2. Create a new location type to hold latitude/longitude setting for the sunset providers. A custom type makes configuring the dependency injection container easier.
3. Create a new schedule data filename type to hold the configuration string needed for the `Schedule` class. (Same as above.)

Hints

Ninject Container

- You will need to add the Ninject NuGet package to the `HouseControlAgent` project.
- Basic usage for Ninject can be found here: https://github.com/ninject/Ninject/wiki/Dependency-Injection-With-Ninject.

Injecting Location

- The new location type can be a class or record.
- After creating the type, configure the Ninject container just like the other objects.

Injecting a Filename

- The new schedule data filename type can be a record in the same file as the `Schedule` class.
- Configuration of the Ninject container will be similar to that of the location type.

If this is enough information for you, and you want a bit of a challenge, **stop reading here**. Otherwise, continue reading for step-by-step instructions.

Step-by-Step

Keep scrolling for the step-by-step walkthrough. Or use the links below to jump to a section:

Step-by-Step - Ninject Container

1. Open the `Program.cs` file in the `HouseControlAgent` project.

2. Review the `InitializeHouseController` method:

```
private static HouseController InitializeHouseController()
{
    var fileName = AppDomain.CurrentDomain.BaseDirectory + "ScheduleData";
    var sunsetProvider = new SolarTimesSunsetProvider(28.4672,-81.4687);
    var cachingSunsetProvider = new CachingSunsetProvider(sunsetProvider);
    var schedule = new Schedule(fileName, cachingSunsetProvider);
    var controller = new HouseController(schedule);
    controller.Commander = new FakeCommander();

    var sunset = cachingSunsetProvider.GetSunset(DateTime.Today.AddDays(1));
    Console.WriteLine($"Sunset Tomorrow: {sunset:G}");

    return controller;
}
```

Right now, all of the objects are created manually using "new". The goal is to have the container do all object creation.

3. Use the object graph to see what needs to be created. (Start at the bottom and work your way up).
    - `HouseController` => needs a `Schedule`
      A `FakeCommander` is also needed when hardware is not available
    - `Schedule` => needs data filename and `ISunsetProvider`
    - `CachingSunsetProvider` => needs `ISunsetProvider`
      Note, skip caching on the first pass and add it back in later
    - `SolarTimesSunsetProvider` => needs latitude and longitude

These are all of the object that we will need to configure.

4. Add the "Ninject" NuGet package to the `HouseControlAgent` project.

5. At the top of the `InitializeHosueController` method, create an `IKernel` (the Ninject container type) and initialize it to a new `StandardKernel` (the default Ninject container type).

```
private static HouseController InitializeHouseController()
{
    IKernel container = new StandardKernel();

    // other code still in place below this.
}
```

> Note: you may need to add a `using Ninject;` statement at the top of the class.

6. Replace the creation of the `HouseController` instance to `Get` it from Ninject.

```
//var controller = new HouseController(schedule);
var controller = container.Get<HouseController>();
controller.Commander = new FakeCommander();
```

Ninject does not have enough configuration information at this point, but it will tell us what we need.

7. Run the application and look at the details of the error message:

```
Ninject.ActivationException
  HResult=0x80131500
  Message=Error activating string
No matching bindings are available, and the type is not self-bindable.
Activation path:
  3) Injection of dependency string into parameter filename of constructor of type
Schedule
  2) Injection of dependency Schedule into parameter schedule of constructor of type
HouseController
  1) Request for HouseController

Suggestions:
  1) Ensure that you have defined a binding for string.
  2) If the binding was defined in a module, ensure that the module has been loaded
into the kernel.
  3) Ensure you have not accidentally created more than one kernel.
  4) If you are using constructor arguments, ensure that the parameter name matches
the constructors parameter name.
```

The error is nice because it details the path. (1) to create a `HouseController`, (2) a `Schedule` is needed. To create a `Schedule`, (3) a filename string is needed. The suggestions are also helpful to get us started.

8. Create a binding configuration for the filename string.

```
var fileName = AppDomain.CurrentDomain.BaseDirectory + "ScheduleData";
container.Bind<string>().ToConstant(fileName);
```

> Note: It is generally not good to bind to a general type like `string` because there are often multiple strings needed by various types. In this case, it will get you headed in the right direction, and you can deal with fixing it in a later step.

9. Run the application.

Now we get a different exception:

```
Activation path:
  3) Injection of dependency ISunsetProvider into parameter sunsetProvider of
constructor of type Schedule
  2) Injection of dependency Schedule into parameter schedule of constructor of type
HouseController
  1) Request for HouseController
```

This tells us that the `ISunsetProvider` needs to be configured in the container.

10. Create a binding configuration for `ISunsetProvider`. Bind it to the actual sunset provider (skip caching for now).

```
container.Bind<ISunsetProvider>().To<SolarTimesSunsetProvider>();
```

11. Run the application.

```
  4) Injection of dependency double into parameter latitude of constructor of type
SolarTimesSunsetProvider
  3) Injection of dependency ISunsetProvider into parameter sunsetProvider of
constructor of type Schedule
  2) Injection of dependency Schedule into parameter schedule of constructor of type
HouseController
  1) Request for HouseController
```

The `SolarTimesSunsetProvider` needs latitude and longitude parameters.

12. Create a hard-coded binding to a new `SolarTimesSunsetProvider` with the parameters filled in.

```
//container.Bind<ISunsetProvider>().To<SolarTimesSunsetProvider>();
var sunsetProvider = new SolarTimesSunsetProvider(28.4672,-81.4687);
container.Bind<ISunsetProvider>().ToConstant(sunsetProvider);
```

> This code is not ideal. You want to eliminate manual object creation wherever possible. You will fix this in a later step.

13. Run the application.

```
Initializing Controller
Sunset Tomorrow: 11/1/2025 6:14:04 PM
Device 5: On
11/2/2025 10:44:54 AM - Device: 5, Command: On
Device 5: Off
11/2/2025 10:44:54 AM - Device: 5, Command: Off
Initialization Complete
```

The application is now running, and you can address the issues to clean up the container process.

> From this point on, the application should continue to run after each step unless otherwise stated. Keep running the application after making code changes to make sure it is still working.

14. Near the bottom of the method, update the `GetSunset` call so that it gets an object from the Ninject container.

```
//var sunset = cachingSunsetProvider.GetSunset(DateTime.Today.AddDays(1));
var sunset = container.Get<ISunsetProvider>()
    .GetSunset(DateTime.Today.AddDays(1));
Console.WriteLine($"Sunset Tomorrow: {sunset:G}");
```

By asking the Ninject container for an `ISunsetProvider`, you are guaranteed to be using the same sunset provider used in other places in the application.

15. Review the current method. (Note: the commented code and caching sunset provider have been removed).

```
private static HouseController InitializeHouseController()
{
    IKernel container = new StandardKernel();
    var fileName = AppDomain.CurrentDomain.BaseDirectory + "ScheduleData";
    container.Bind<string>().ToConstant(fileName);

    var sunsetProvider = new SolarTimesSunsetProvider(28.4672,-81.4687);
    container.Bind<ISunsetProvider>().ToConstant(sunsetProvider);
```

```
    var controller = container.Get<HouseController>();
    controller.Commander = new FakeCommander();

    var sunset = container.Get<ISunsetProvider>()
        .GetSunset(DateTime.Today.AddDays(1));
    Console.WriteLine($"Sunset Tomorrow: {sunset:G}");

    return controller;
}
```

16. Get the `FakeCommander` from the Ninject container.

```
var controller = container.Get<HouseController>();
//controller.Commander = new FakeCommander();
controller.Commander = container.Get<FakeCommander>();
```

17. As an alternative, you can configure Ninject to set the `Commander` property automatically. This is done by creating a configuration for the `HouseController` itself.

```
container.Bind<HouseController>().ToSelf()
    .WithPropertyValue("Commander", container.Get<FakeCommander>());
var controller = container.Get<HouseController>();
//controller.Commander = container.Get<FakeCommander>();
```

This creates a binding for the `HouseController` type. `ToSelf` says that this is not an abstraction, so the binding is to `HouseController`.

`WithPropertyValue` lets you configure a property that will be set when the object is created. In this case, you specify that the `Commander` property should be set to a `FakeCommander`.

> This binding is only to configure the property setting. As you saw earlier, the Ninject container can still create the `HouseController` object without this configuration.

18. As a little bit of a safety net, surround this binding with `#if DEBUG` directives. This will help make sure this does not make it into a release build.

```
#if DEBUG
    container.Bind<HouseController>().ToSelf()
        .WithPropertyValue("Commander", container.Get<FakeCommander>());
#endif
```

> Note: If you are in an ASP.NET Core project (or other application that uses the application builder), you can set a qualifier based on the environment.

```
if (app.Environment.IsDevelopment())
{
    // development-only code here
}
```

19. You can test the setting by chaning to "Release" mode and running the application. You will get a runtime exception that COM5 is not available (the same message from Lab 01).

Be sure to set back to "Debug" mode before continuing.

20. Add caching back in. Change the binding for the `ISunsetProvider` to `CachingSunsetProvider`.

```
var sunsetProvider = new SolarTimesSunsetProvider(28.4672,-81.4687);
//container.Bind<ISunsetProvider>().ToConstant(sunsetProvider);
container.Bind<ISunsetProvider>().To<CachingSunsetProvider>();
```

21. Running at this point results in an exception.

```
Ninject.ActivationException
  HResult=0x80131500
  Message=Error activating ISunsetProvider using binding from ISunsetProvider to
CachingSunsetProvider
A cyclical dependency was detected between the constructors of two services.
```

The last line tells us a cyclical dependency is detected. This is because the `CachingSunsetProvider` needs and `ISunsetProvider` for a constructor argument. Ninject notices that trying to provide (another) `CachingSunsetProvider` for that would not work.

22. Use `WithConstructorArgument` to set the parameter for the `CachingSunsetProvider`.

```
var sunsetProvider = new SolarTimesSunsetProvider(28.4810,-81.5074);
container.Bind<ISunsetProvider>().To<CachingSunsetProvider>()
    .WithConstructorArgument<ISunsetProvider>(sunsetProvider);
```

This tells Ninject to use the `sunsetProvider` instance (`SolarTimesSunsetProvider`) as the constructor argument.

23. Rerun the application. The application runs and caching is in place. (You can follow the steps from Lab 01 to test caching again if you like).

In the next section, you will add configuration for the `SolarTimesSunsetProvider` so that you do not need to create it manually. But the parameters will need to be addressed first.

Step-by-Step - Injecting Location

1. Right now, the `SolarTimesSunsetProvider` is created manually.

```
var sunsetProvider = new SolarTimesSunsetProvider(28.4672,-81.4687);
container.Bind<ISunsetProvider>().To<CachingSunsetProvider>()
    .WithConstructorArgument<ISunsetProvider>(sunsetProvider);
```

Instead, the Ninject container should provide this instance. But there's a problem: the parameters (latitude and longitude) are both of type `double`, so there is no quick way to configure that in the container (like you did with the filename string).

A better approach in this case is to take a look at the types. Latitude and longitude are only useful in this application when they are combined. So, it would make sense that this should be a single type with two values.

2. In the `HouseControl.Sunset` project, create a new file/type for `LatLongLocation`.

```
namespace HouseControl.Sunset;

public record LatLongLocation(
    double Latitude, double Longitude) { }
```

This code uses a record type for the 2 values. This belongs in the `HouseControl.Sunset` project because this is needed to calculate the sunrise and sunset times.

3. Change the constructor for the `SolarTimesSunsetProvider` to use the new type.

```
//public SolarTimesSunsetProvider(double latitude, double longitude)
//{
//    this.latitude = latitude;
//    this.longitude = longitude;
//}

public SolarTimesSunsetProvider(LatLongLocation latLong)
{
    latitude = latLong.Latitude;
    longitude = latLong.Longitude;
}
```

4. For consistency, change the constructor for the `SolarServiceSunsetProvider` class as well.

```
public SolarServiceSunsetProvider(LatLongLocation latLong)
{
    latitude = latLong.Latitude;
    longitude = latLong.Longitude;
}
```

At this point, the application will not build.

5. A build failure points to creating the `SolarTimesSunsetProvider` in the `InitializeHouseController` method that you were working with earlier.

```
var sunsetProvider = new SolarTimesSunsetProvider(28.4672,-81.4687);
container.Bind<ISunsetProvider>().To<CachingSunsetProvider>()
    .WithConstructorArgument<ISunsetProvider>(sunsetProvider);
```

The constructor parameters have changed.

6. Create a `LatLongLocation` object for the constructor.

```
var latLong = new LatLongLocation(28.4672,-81.4687);
var sunsetProvider = new SolarTimesSunsetProvider(latLong);
container.Bind<ISunsetProvider>().To<CachingSunsetProvider>()
    .WithConstructorArgument<ISunsetProvider>(sunsetProvider);
```

The application will now build and run.

7. Create a Ninject binding for the `LatLongLocation` type.

```
var latLong = new LatLongLocation(28.4672,-81.4687);
container.Bind<LatLongLocation>().ToConstant(latLong);
```

8. Get the `SolarTimesSunsetProvider` from the Ninject container.

```
//var sunsetProvider = new SolarTimesSunsetProvider(latLong);
var sunsetProvider = container.Get<SolarTimesSunsetProvider>();
container.Bind<ISunsetProvider>().To<CachingSunsetProvider>()
    .WithConstructorArgument<ISunsetProvider>(sunsetProvider);
```

9. Remove the intermediate `sunsetProvider` variable, and use the `Get` call inside the `WithConstructorArgument` parameters.

```
container.Bind<ISunsetProvider>().To<CachingSunsetProvider>()
    .WithConstructorArgument<ISunsetProvider>(
        container.Get<SolarTimesSunsetProvider>());
```

The configuration looks a bit complex at first, but once you get comfortable with using a dependency injection container, things look much more natural.

10. Here is the final configuration section for `LatLongLocation`, `CachingSunsetProvider`, and `SolarTimesSunsetProvider`.

```
var latLong = new LatLongLocation(28.4672,-81.4687);
container.Bind<LatLongLocation>().ToConstant(latLong);
container.Bind<ISunsetProvider>().To<CachingSunsetProvider>()
    .WithConstructorArgument<ISunsetProvider>(
        container.Get<SolarTimesSunsetProvider>());
```

One thing to notice is that you do not have any specific binding configuration for the `SolarTimesSunsetProvider`. When you ask the container for the `SolarTimesSunsetProvider` (as the constructor argument), the container looks at the constructor, sees that it needs a `LatLongLocation` that is already configured, and takes care of the object creation for you.

As a final step, you will add a bit of safety to the filename string configuration.

Step-by-Step - Injecting a Filename

1. Review the filename string configuration in the `IntializeHouseController` method.

```
var fileName = AppDomain.CurrentDomain.BaseDirectory + "ScheduleData";
container.Bind<string>().ToConstant(fileName);
```

As you have seen, this code works, but it is not very robust. What if another type needs a string for configuration? A more specific type could be helpful here.

2. Open the `Schedule` class in the `HouseControl.Library` project.

3. Review the constructor.

```
public Schedule(string filename, ISunsetProvider sunsetProvider)
{
    this.scheduleHelper = new ScheduleHelper(sunsetProvider);
    this.filename = filename;
    LoadSchedule();
}
```

The constructor has 2 parameter, both of which are injected by the Ninject container.

4. Add a new `ScheduleFileName` record type just above the `Schedule` class declaration.

```
public record ScheduleFileName(string FileName) { }

public class Schedule : List<ScheduleItem>
{
    // code omitted.
}
```

5. Update the constructor to use the new type.

```
public Schedule(ScheduleFileName filename,
    ISunsetProvider sunsetProvider)
{
    this.scheduleHelper = new ScheduleHelper(sunsetProvider);
    this.filename = filename.FileName;
    LoadSchedule();
}
```

6. If you try to build at this point, you will get build failures from the test project in Lab 02.

7. Open the `ScheduleTests.cs` file in the `HouseControl.Library.Tests` project.

8. Change the `fileName` field from a `string` to the new `ScheduleFileName` type.

```
//readonly string fileName = AppDomain.CurrentDomain.BaseDirectory +
"\\ScheduleData";
readonly ScheduleFileName fileName = new(AppDomain.CurrentDomain.BaseDirectory +
"\\ScheduleData");
```

9. Build the solution. At this point you will get a successful build.

10. Go back to the `InitializeHouseController` method in the `HouseControlAgent` project.

11. Change the type of the `fileName` from `string` to the new type, and update the binding configuration.

```
//var fileName = AppDomain.CurrentDomain.BaseDirectory + "ScheduleData";
//container.Bind<string>().ToConstant(fileName);
ScheduleFileName fileName = new(AppDomain.CurrentDomain.BaseDirectory +
"ScheduleData");
container.Bind<ScheduleFileName>().ToConstant(fileName);
```

12. Run the application.

The application runs just as it did before, but the configuration is less likely to break as the application evolves. If another object needs string configuration, then the existing configuration with the custom type continues to work.

ALTERNATIVE
There are a few alternative approaches to handling the `string` configuration.

Ninject (and other containers) support "named" bindings. This lets you supply a name for a binding and then ask for that specific item later.

```
container.Bind<string>().ToConstant("Hello").Named("greeting");
container.Bind<string>().ToConstant("Goodbye").Named("farewell");
```

Option 1:
In the class that needs this dependency, an attribute can be added to the constructor parameter.

```
public IntroductionMessage([Named("greeting")]string message) {...}
```

Ninject would use the "greeting" string for this constructor.

> One downside to this approach is that the "Named" attribute is part of the Ninject project and namespaces. In the lab application, this would mean that Ninject would need to be added to the `HouseControl.Library` project. My preference is to not include dependency injection-specific items if I do not need them. It feels out of place to have Ninject (a specific container) as a dependency in my library project.

Option 2:
Another option is to pull a named item out of the container during configuration of the container.

```
container.Bind<IntroductionMessage>().ToSelf()
    .WithConstructorArgument<string>(
```

```
        container.Get<string>("greeting"));
```

This has the advantage of keeping the Ninject pieces in the composition root (where all of the objects are created). So, this would not require Ninject to be added to any other projects.

This approach does not have much of an advantage to the approach in the samples above because it is more or less the same amount of code but with more indirection. This is why I chose the particular path for the step-by-step.

Wrap Up

In this lab, you saw how to add a dependency injection container to an application. You went step-by-step to better understand what the container does and how binding (configuration) works.

Once you are familiar and comfortable with using a dependency injection container, you are more likely to start by setting up bindings for all of the abstractions (interfaces in these examples), and then asking the container to "get" the objects that you need to use.

You also saw how to use custom types for easier configuration. Containers are good at mapping abstractions to concrete types and mapping types to values. If you have a generic type (such as a string), things can get ambiguous, particularly if more than one object has a dependency on a string. More robust types can help with that.

Here is the final code for `InitializeHouseController`:

```
private static HouseController InitializeHouseController()
{
    IKernel container = new StandardKernel();
    ScheduleFileName fileName = new(AppDomain.CurrentDomain.BaseDirectory +
"ScheduleData");
    container.Bind<ScheduleFileName>().ToConstant(fileName);

    var latLong = new LatLongLocation(28.4672,-81.4687);
    container.Bind<LatLongLocation>().ToConstant(latLong);
    container.Bind<ISunsetProvider>().To<CachingSunsetProvider>()
        .WithConstructorArgument<ISunsetProvider>(
            container.Get<SolarTimesSunsetProvider>());

#if DEBUG
    container.Bind<HouseController>().ToSelf()
        .WithPropertyValue("Commander", container.Get<FakeCommander>());
#endif

    var controller = container.Get<HouseController>();


    var sunset = container.Get<ISunsetProvider>()
```

```
            .GetSunset(DateTime.Today.AddDays(1));
        Console.WriteLine($"Sunset Tomorrow: {sunset:G}");

        return controller;
    }
```

This has about the same amount of code as the original, but now it relies on the Ninject container to do the work.

One advantage to having the container is that we can let the container do other things for us as well -- such as lifetime management.

Lifetime Management
We can give configure a binding with `.InSingletonScope()` to denote that only a single instance of a type should be created. Or, we could use `.InTransientScope()` to get a fresh instance each time we need one.

This particular application does not have many features, so you would not get much benefit from this. But in a larger application with screens, modules, and functions, you can let the container deal with which items to keep around, which to discard, and which to maintain at a scope level.

Lifetime management is a good subject to look into when you are ready to dive deeper into dependency injection.

END - Lab 03 - More Dependency Injection