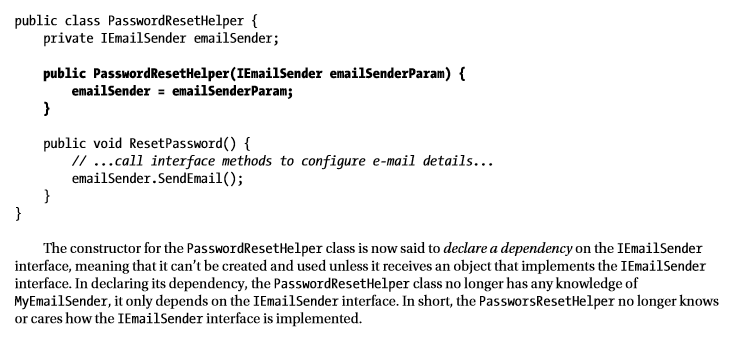
Chapter 6: Essential Tools for MVC Notes

Chapter Summary (from book):

In this chapter, I looked at the three tools I find essential for effective MVC development: Ninject, the built-in Visual Studio support for unit testing, and Moq. There are many alternatives, both open source and commercial, for all three tools and you will not lack alternatives if you do not get along with the tools I like and use. You may find that you do not like TDD or unit testing in general, or that you are happy performing DI and mocking manually. That, of course, is entirely your choice. However, I think there are some substantial benefits in using all three tools in the development cycle. If you are hesitant to adopt them because you have never tried them, I encourage you to suspend disbelief and give them a go, at least for the duration of this book.

1. Chapter Start
   1. Main topics/tools covered in the chapter:
      1. Dependency injection (DI) container
      2. Unit test framework
      3. Mocking tool
   2. Chapter Outline
      1. **Decouple Classes**: Introduce interfaces and declare dependencies on them in the class constructor
      2. **Automatically resolve dependencies using interfaces**: Use Ninject or another dependency injection container
      3. **Integrate Ninject into an MVC application**: Create an implementation of the IDepdencyResolver interface that calls the Ninject kernel and register it as a resolver by calling the System.Web.Mvc.DepdencyResolver.SetResolver method
      4. **Inject property and constructor values into newly created objects**: Use the WithPropertyValue and WithConstructorArgument methods
      5. **Dynamically select an implementation class for an interface**: Use an Ninject conditional binding
      6. **Control the lifecycle of the objects that Ninject creates**: Set an object scope
      7. **Create a unit test**: Add a unit test project to the solution and annotate a class file with TestClass and TestMethod attributes
      8. **Check for expected outcomes in a unit test**: Use the assert class
      9. **Focus a unit test on a single feature of component**: Isolate the test target using mock objects
2. Stages to using Ninject as a dependency injection container
   1. Prepare Ninject for use
      1. Create an instance of a Ninject kernel
         1. *Object responsible for resolving dependencies and creating objects…*
         2. *When object is needed… the kernel is used instead of the new keyword*
      2. Configure Ninject Kernel
         1. *Important so that the kernel understands which implementation of objects you want to use for each interface*
      3. Create an Object
         1. *Through the kernel Get method*
3. Dependency Resolver
   1. Used to create instances of classes that the MVC Framework needs to service requests…
      1. For example, ensures that the MVC Framework uses Ninject whenever it creates an object-including instances of controllers…
   2. What happened after running the constructor injection example app:
      1. MVC Framework received the request and figured out that the request is intended for the Home controller
      2. Framework asked the custom dependency resolver class to create a new instance o the HomeController class, specifying the class using the Type parameter of the GetService method
      3. Dependency resolver asked Nniject to create a new HomeControler class passing on the Type object to the TryGet method
      4. Ninject inspected the HomeController constructor and found that it is declared a dependency on the IValueCalculator interface, for which it has a binding.
      5. Ninject creates an instance of the LinqValueCalculator class and uses it to create a new instance of the HomeController class
      6. Ninject passes the HomeController instance to the custom dependency resolver, which returns it to the framework. The framework uses the controller instance to service the request.
4. General idea of chains of dependency…
   1. When you ask ninject to create a type… it examines the dependencies that the type has declared. It also looks at those dependencies to see if they rely on other types… or if they declare their own dependencies.
   2. If there exists additional dependencies, Ninject automatically resolves them and creates instances of all the classes that are required… working its way along the chain of dependencies so that it can ultimately create an instance of the type you asked for.
5. Ninject Conditional Binding Methods:
   1. When(predicate)
      1. Binding is used when the predicate – a lambda expression – evaluates to true
   2. WhenClassHas<T>()
      1. Binding is used when the class being injected is annotated with the attribute whose type is specified by T
   3. WhenInjectedInto<T>()
      1. Binding is used when the class is being injected into a class of type T
6. Ninject Scope
   1. Ninject allows you to control the lifecycle of the objects you create using a feature called scope… Examples include:
      1. You want to share a single instances throughout the entire application
      2. Create a new instance for each HTTP request that the ASP.NET platform receives
   2. The Ninject Scope Methods (that the author finds the most useful):
      1. InTransientScope() – Same as not specifying a scope and creates a new object for each dependency that is resolved
      2. InSingletnScope() – Creates a single instances which is shared throughout the application. Ninject will create the instance if you are using InSingleScope or you can provide it with the ToConstant(object) method
      3. InThreadScope() – Creates a single instance which is used to resolve dependencies for objects requested by a single thread
      4. InRequestScope() – Creates a single instance which is used to resolve dependencies for objects requested by a single HTTP request

Quick Recap… see below on information on how to declare a dependency



# Testing

Don’t forget to add references after creating test project… this ensures that the test project has access to the classes in the application project and can perform tests on them.

Best Unit Tests are simple and focused

Using Moq for testing

1. Create a mock object
   1. Tell Moq what kind of mock object you want to work with…
2. Specify the behavior
3. Defining the Result
4. Use the mock object in the unit test

Methods of the “It” Class:

1. Is<T>(predicate)
   1. Specifies values of type T for which the predicate will return true
2. IsAny<T>()
   1. Specifies any value of the type T
3. IsInRange<T>(min, max, kind)
   1. Matches if the parameter is between the defined values and of type T… final parameter is a value from the Range enumeration and can be either Inclusive or Exclusive
4. IsRegex(expr)
   1. Matches a string parameter if it matches the specified regular expression