Creating N-Tier Applications in C#, Part 1

by Steve Smith

Start Course

Bookmark Add to Channel Download Course Schedule Reminder

Table of contents Description Transcript Exercise files Discussion Learning Che

Application Evolution

Introduction

Hi, and welcome to this Pluralsight course on Creating N-Tier Applications in C# Part 1, Application Evolution. My name is Steve Smith and you can learn more about me on my blog at Ardalis. com or you can follow me on Twitter where I'm @ardalis. Creating N-Tier Applications in C# is a big topic, so this course is just part 1 in a series. In this course, you'll learn how N-tier applications evolve which is a subject of this module. In the next module, you'll learn about traditional data-centric N-tier design and architecture. And then in the third module, you'll learn about domain-centric N-tier design and architecture and how this differs from the data-centric style. In discussing the evolution of N-tier applications, we also need to talk through some of the basics of N-tier design and architecture. In this module, you'll learn what we mean when we refer to an application that's consisting of N-tiers or layers. You'll learn the difference between logical and physical separation. You'll learn about the typical tiers that make up an N-tier application design. You'll learn some of the pros and cons of applying an N-tier design to your application. And finally, you'll see a demo showing an application comprised of just one tier. Later on, we'll expand on this demo to show how we can evolve it into an N-tier application.

Definition

The N in N-tier refers to the number of tiers. A very common value for N is 3, comprised of the 3 tiers of the user interface or presentation, business logic, and data access logic. When an application's behavior is split among multiple tiers, these tiers are said to be logically separated. In some cases, the separate tiers are physically deployed to different devices or process boundaries on the same device, in which case they are said to be physically separated as well. It is not required for separate layers to be deployed separately, but they may be, likewise, it's not required that separate layers be developed independently of one another. But this approach means that it's possible for this to take place. In this way, you can allow for larger teams to work together concurrently or to have multiple teams working on separate parts of a larger application. Throughout this course, you'll hear me refer to tiers as well as layers. Generally, you can refer to both of these terms interchangeably from one another. However, sometimes, each one has a more specific meaning. In the case of layer, layer is often used to apply specifically to the logical structure of the application. Likewise, the tiers of an application frequently are used to refer to just the physical structure of the system architecture or infrastructure. In many cases, you can refer to an application as an N-tier application even if it is only deployed to a single machine. And it's also not uncommon to talk about the different layers of an application when in fact these layers are deployed to different physical tiers. When I use one of these terms in the more specific case shown here, I'll be sure to call it out so that there is no confusion.

Logical/Physical Separation

Now when we talk about logical versus physical separation of the different concerns within an application, we're talking about two different things. The logical separation of concerns is expressed in the source code of the application. The code's organization and design should reflect the layers that are in use. At a minimum, individual classes should not include responsibilities that correspond to separate layers. And typically, each layer will have its own name space, assembly, and project in the solution. The logical separation of an application across layer boundaries represents a software design pattern. The physical separation of the application generally refers to how it is deployed. It is possible to have a logically separated app that is physically deployed to a single location. For instance, a web application with separate assemblies for business logic and data access, all of which are deployed into the same slash bin folder on a single web server. However, it's possible to separate these assemblies and deploy them to different physical locations, whether on a single device or on separate machines. The physical separation of the application across process boundaries and/or devices represents a software architecture decision. Note that it is generally impractical to have the physical separation described here without the corresponding logical separation. Thus, you must have logical separation in your software if you are foreseeing the need to have physical separation in your deployment.

Evolution and Options

Let's consider this diagram showing different types of N-tier or various levels of tier application design. This diagram shows a number of different options for separating application responsibilities into layers and tiers. This diagram is actually from Microsoft's NSTM website from the year 2001, and it's one I've used in many of my presentations including my very first conference presentation on N-tier application development. I still like this diagram though today I prefer a more domain-centric organization of my application's responsibilities, which of course you'll learn more about in the third module of this course. Now you can see in this diagram that a typical application is comprised of a presentation layer, some business rules, a data access layer, and a data source. Though this is a rather a dated image, these layers are still typical for applications written today. As you move from left to right across the diagram, the different layers are further separated into their own separately deployable components. These could be separate assemblies in a. NET application or they could be separate-- yeah, applications themselves in a more distributed architecture. The process boundaries might be in between separate processes on a single machine or they could reside on different machines and communicate over a network. The three-tier design is one of the most common best practice application designs and is so common that most experienced software developers are familiar with the acronyms DAL and BLL used to refer to the data access layer and business logic layer respectively. In my experience, it's quite rare to see a workflow there separated into its own tier as shown in the far right. Generally, the workflow is encapsulated into services and different presentation layers who will use different services with interfaces appropriate to their workflow. For instance, the workflow of a large desktop's line of business application is likely to be quite different from the workflow expressed by a single screen of a mobile phone application even if both of them are trying to accomplish the same task. Now, let's look at how applications typically evolve to incorporate these tiers. Simple applications typically begin with few tiers. Added layers in these cases do not provide the return on investment or ROI for such small applications. In the example shown here, the green three-dimensional box represents a computing device. It could be a PC, a phone, or a server. The yellow data in this case could be stored in memory within the application or it could be stored separately in a device's file system or on a local database. At this stage, the application is as fast as it possibly can be for a single user because everything it needs resides on the device and possibly in the process of the application. Also, the skills required for the developer to work with the application are also extremely simple. For example, a developer knowing only C# could build such an application using a console or WinForms front-end, knowing nothing about SQL, HTML, XAML, XML, or other languages. Overtime, additional separation may be added via refactoring or rewriting the application. Sometime, separating the code logically is a first step before the application can be physically deployed in a different manner. Professional software developers building enterprise software will frequently start development of an application using a logically separate, but physically combined model like the one shown here. This offers simple deployment and fast performance while providing future flexibility with regard to the application's physical deployment. Although the data storage of an application isn't necessarily a logical layer in the application, it is often one of the first things that is physically deployed separately. Sometimes, the database is used for both behavior and business rules in addition to data storage, in which case it should be considered as a layer in your application logically as well as physically. It's also true that in the real world, database is almost always are deployed on their hardware and often are managed independently of the applications that utilize them. In this stage like the one shown here, the version of the database must be kept synchronized with the application or applications that access it which adds complexity to deployments. Further, the team responsible for supporting this app now must be able to deal with the SQL language, database server management, and network infrastructure considerations, just to name a few. Clearly, this is a more complex design than the original simple application design. Some advantages of this physical deployment strategy are a greater security for the application data and the ability to optimize the database server for database operations. Just for completeness, I want to also note that it's not uncommon to have an unstructured application with all of its logic bundled together with its user interface and data access code accessing a remote database, especially for newly created hobbyist or simple applications. However, this is not a design that I would recommend. At some point, there may be more than one application accessing your remote data source. This introduces an added layer of complexity to deployments which must now deal with the possibility of multiple versions of applications accessing the shared data store. Or, the deployment process must safeguard against this possibility by ensuring that application deployments affect all instances of the application simultaneously. You can think of examples of this as apps that are rolled to mobile devices which call back to remote storage services. It may be possible that different users will be writing different versions of these applications. Ultimately, you can think of each of the devices shown as web servers, perhaps in a web farm environment both talking back to a shared database server. When an update is performed to the web servers, it will be important that both servers get updated at the same time otherwise it's possible that they'll be out of sync with the database format and they won't be able to talk to it correctly. Frequently for deployment reasons or in order to consolidate business logic in one location and perhaps alleviate the aforementioned deployment considerations, business logic and application services will be deployed to a separate server. This server or servers should ideally shield the application from changes in the data store and can provide performance benefits by offloading work from the data server and/or the application devices. It can also provide caching or similar performance improving techniques. Eventually, the application may evolve into an architecture that has all the application services as well as data services deployed to some sort of a cloud or web farm environment. This provides both scalability as well as fault tolerance and makes it so that the units of deployment, the application devices are as simple as possible. As we see more and more infrastructure as the service applications entering the market, we can expect to see these designs to grow in popularity.

Benefits

By using an N-tier design, your application will gain a number of benefits. Your development team will be able to reuse code from lower modules between different parts of higher level modules. For instance, if the logic for accessing say a customer is encapsulated in a data access layer class, then multiple business logic layer classes can utilize this code without anyone having to rewrite it. Likewise, the entire DAL module may be reusable between different applications such as for a website as well as for WinForms or console applications. Separating the responsibilities of an application can also improve productivity by allowing multiple team members or teams to work concurrently on different parts of the application. Further, by encapsulating all the logic related to a particular responsibility in one area, N-tier design improves maintainability. If a change needs to be made to how the database is accessed, this change should only need to be applied to the data access layer. If a change is only made in how a business rule is implemented, then it should only be made to the business logic layer. And if a change is needed in how the user interface is designed or how data is formatted for the user, this change should only affect the user interface layer. Ideally, an N-tier design should provide a more loosely coupled application. That is, some parts of the application should be able to change without affecting the rest of the application. As an example, moving or restructuring the database should only affect the data access layer unless the changes are extreme, in which case BUI or business logic layers might also be affected. Finally, using an N-tier logical application design provides more options for physical deployment. The ideal number of physical tiers for your application is one because this is the simplest and most performed option. However, if your needs required you to add additional physical application tiers, the use of logical layers in your application should allow you to do this.

Drawbacks and Risks

One of my favorite sayings is there's no such thing as a free lunch. And, choosing to implement an N-tier design is no different. One thing to consider when adding physical tiers to your application is the performance impact it can have. Frequently, we talk about adding additional servers to our application in order to increase its performance and scalability, but the reality is that the fastest an application can perform is when all of its processes are in-process on a single machine. As soon as it needs to start talking across process or even machine boundaries, performance for an individual request is going to suffer. Frequently, the granularity of operations that are performed in an application must be also be adjusted as the number of different physical machines involved in the process increases. That is, you will want to move from chatty to chunky interfaces as you move from in-process to add out of process and remote API boundaries. Of course, designing and developing N-tier applications is also a more complex endeavor than a simple monolithic application. As such, the skills and experience required for the team that's doing the development are somewhat greater. Finally, deploying N-tier applications can be much more complex. Frequently, multiple physical machines must be deployed in a batch or else the application must be designed to support multiple different versions of APIs concurrently. These complexities increase exponentially with the number of layers and versions of APIs deployed making it a good idea to limit the number of simultaneous versions your application can support, or else to design for multi-version support from the start.

Coupling

I mentioned that one advantage of N-tier application design is loose coupling. So, let's talk for a moment about what coupling means. When we have an application that has multiple modules that can each change independent of one another, we refer to that as being loosely coupled or is exhibiting loose coupling. The opposite of this is when we have modules that when changed, require other modules to change as well. This is referred to as "tight coupling" and is generally something we want to avoid as much as possible. You can design your applications to have looser coupling if you follow certain principles of object oriented design when you design the interactions between your tiers and layers. There's a Pluralsight course on the Principles of Object Oriented Design available at this URL.

End Runs

You should avoid making end runs around the layers in your application. An end run is when one application layer calls to another that is not directly below it. All of the benefits of separating your application into logical layers are lost if these layers are not used consistently. They should act as boundaries that insulate one layer from the next. Avoid making calls from one layer to another that is not immediately below it. Also, you should avoid allowing concerns from one layer to leak up to a higher layer. For example, when considering the return types and parameters to use between layers, avoid things like the SqlDataReader that automatically imply a connection to a SQL server database. For instance, if your data access layer returns a SqlDataReader, your business logic layer should take care to return something else to the user interface layer so that the knowledge of a SQL database being involved does not leak from the business logic layer up to the user interface layer. Ideally, this knowledge would not even enter the business logic layer which would be calling the data access layer through an interface such as an IReader interface that would also have no knowledge of the SqlDataReader. In this example, you can see that UI layer making a call directly to the database. For instance, in an ASP. NET application, a web page might make use of a SQL data source control to fetch some data. This end run around the BLL and DAL layers affectively couples the UI layer to the database and bypasses any rules or additional behavior that would have been applied to this operation if it had gone through these layers. End runs like this destroy the advantages of utilizing an N-tier application design resulting in an application with all of the disadvantages and complexity of an N-tier design and none of its advantages. Don't do this.

Demo: Monolithic App

Now, let's talk about the monolithic application. This is an N-tier application where N equals 1. That is, there is only a single unit of deployment and no logical or physical separation of concerns. All of the parts of the application are included in one assembly or one unit of deployment and some examples of this might be a Microsoft Access database that includes the data as well as the forms and the logic all inside of the one MDB file, or a console application that doesn't have any external dependencies, or a simple web application that includes or automatically creates whatever database it might require. In each of these cases, deployment is a simple matter of copying the files and running the application. Our sample monolithic application is a new social web site. We're going to call it PluralsightBOOK, and eventually this will be a Facebook killer app that will be worth billions. For now though, we're going to start and we're going to give it some initial features. We're going to be able to have users register and we're going to let them then login and we'll let create and edit there profile that they can share with other users. We'll build on this application in later modules. So, let's have a look at this code. So, let's look at the PluralsightBOOK solution in visual studio. First thing I want to call out is in the solution explorer, look at the different projects that are involved. In this case, we have just a single C# web forums ASP. NET application. Now, of course, you can build any sort of front-end application using the N-tier design. I'm only using this as a particular example. Now, what we've done is we've created this out-of-the-box application using one of the prebuilt templates and we've only added a tiny bit to it. So, the first thing that we've done is we've changed some of the title and style things in the default homepage and Site. Master page which you'll see in a moment when I run the application. Another important thing to note about this app is that it makes use of the built-in providers for membership and profile in ASP. NET. You won't see that there is some App Data folder and if you download this file from the before folder in the demos section of the course download, you should see that your app data folder is empty. Now, if I've to show all files, you'll see that mine has an ASTNETDB database in it which we're going to delete right now to reset this application. Now that we no database associated with it, we're going to go ahead and run this by hitting Ctrl-F5 and the application runs and looks like this. Now when-- if a user first accesses this application, we want them to sign up so that they can join the trillions of other PluralsightBOOK students that are currently using this app. So, I'm going to click the "Sign up now" which is going to register me and I'll create my user name of ssmith and my e-mail and my super-secret password and we'll create my user. Now, this takes a moment because it's creating the database. But now you can see that I've authenticated and I now logged in here as ssmith. Now, I can go and view my profile. You can see my profile simply says Welcome! and my favorite Pluralsight author's name is blank which we want to change. So, I'm going to click the edit profile button, I'm going to say my name Steve Smith, and favorite Pluralsight author is, we'll say Fritz Onion. Your welcome Fritz. And now if we view my profile, it says, "Welcome Steve Smith! Your favorite author's name is Fritz Onion. " So, let's look at how we implemented this code. Basically, we had to change the edit profile page and if we look at this, we can see it has a simple form with a couple of text boxes and a button for saving. If we look at the code behind for this, you'll see that I have now this MyProfile property on the page and it exposes the current user type that is-- we're going to call it CurrentUser. And then within the page, when the page first loads, I set those two text boxes to be whatever my current user name and current user's favorite Pluralsight author are, and I do the reverse when you click the save button and then I redirect back to the profile page. This is fairly typical ASP. NET web forums code. Now, to configure the profile, if you're creating a web site, you can actually do all of these in the web config. But it in the web application project which I prefer and recommend, you can do it with your own class. So, if we look here, scroll down and you'll find, there is a profile element and this profile element allows you to specify what provider it should use to persist the profile. And then within here if you want, again with the web site in particular, you can add certain properties. And so, you can add a new property, you know, call it whatever you want and give it certain attributes, or you can just define all these things in a class of your own. And if you do that, you specify the inherits, attribute on the profile, and give it the fully qualified name of that class. So, here, you can see I've got one called MyProfile and it's located here in the Code folder, MyProfile. cs. Now in here, I simply inherit from profile base which is System. Web. Profile. Profilebase and I specify a static property here that represents the current authenticated user and it's going to create one of these from the current user whenever it needs to. And then for each one of my properties, I'm simply wrapping ProfileBase and-- in converting the type as needed. So, I've got a name and I've got a favorite Pluralsight author. In this case, these are about strings. And whenever I set them, I'm just going to call Save on ProfileBase that will trigger the SQL data provider here to go ahead and make the call to save the change to the database. All in all, although this is still a monolithic application, it has some of the benefits of an N-tier design because of the way ASP. NET's provider model is set up. Since we're using the built-in providers for membership and profile, if we wanted to change how we talked to the data store, we could simply use a different provider for a membership or a different provider for our profile access. So, even out of the box using this design, we have something of an N-tier application design for these particular parts of the application. Now, if we were to add some additional logic such as for instance the ability for me to add friends and specify which of the other users in the system I was friends with, I would probably have to write that code by hand. And if I wrote it all within my ASPX web forms, you would see that it would be much less maintainable and much less easy for me to swap out than the code that is shown here. We'll look at an example of just that in the next module. With that, I think we're ready wrap up this module and move on to our summary. (Pause)

Summary

Let's review. An N-tier design can refer to both the logical and physical separations of responsibilities within an application. The terms, tier and layer, may be used interchangeably, though each has specific meaning as well. Logical separation into layers can improve code maintainability while physical separation into tiers can provide scalability, security, and fault tolerance, among other benefits. Applications can evolve overtime as their needs require the complexity of an N-tier design. And remember that you should always avoid making end runs around the layers in your application otherwise you'll eliminate all the benefits of using an N-tier design in the first place. For reference, you can learn more on Wikipedia about multitier architectures and also from the Principles of Object Oriented Design Course on Pluralsight which teaches how to design systems that have loose coupling and are easy to maintain. Thank you very much for watching this course on N-tier systems using C#. My name is Steve Smith and you can learn more about me from my blog on Ardalis. com or on Twitter where I'm @ardalis. Thanks and I hope to see you again real soon.

Data-Centric N-Tier Design

Introduction

Hi and welcome back. This is the Pluralsight course on Creating N-Tier Applications in C sharp, Part 1. This is the second module in the course where we're going to take a look at creating a Data-Centric N-Tier design by starting with a monolithic design and then refactoring it into separate tiers with their own responsibilities. My name is Steve Smith, I will be your instructor here and you can find more about me at my blog on Ardalis. com or on Twitter where I'm @ardalis. I hope you've had a chance to watch the first module in this course on how N-Tier applications evolved. In this module, we're going to focus on the traditional Data-Centric N-Tier architecture and how we can refactor an existing monolithic application to make use of this design. You'll get to experience firsthand some of the pinpoints that this design is meant to address as well as some of the effort that's required to perform this refactoring. In the next module, we'll perform a similar exercise but with the end goal being a Domain-Centric N-Tier Architecture rather than a Data Centric one.

Module Overview

For this module, we're going to spend most of our time actually writing code in Visual Studio. The purpose of this is going to be to extend the sample application that we looked at in the previous module which is our PluralsightBook application. Ultimately, we want this to be a Facebook killing app but for now, it's still just getting started. We're going to begin by building everything into the user interface and then eventually when we start to experience some duplication and some problems with this approach, we're going to refactor some of that common logic into separate business logic layer and data access layer projects. Finally, we'll review the pros and cons of this new design. So as you may recall, our sample application is a social website called PluralsightBook which you can see in the screenshot here. Currently, we've implemented features so that users are able to register and login as well as create and update or view their profile. For now, these features are based on the ASP. NET provider model and the default implementations of these features. For this exercise, we're going to add a couple of new features. We're going to be able to add a friend and view a list of our current friends and then also, the ability to remove a friend from that list.

Demo: Add Friend Support

In this first part of the demo, we're going to simply extend PluralsightBook with an add friend feature by writing all of the code in the web application. One thing you may notice is that we are now using Visual Studio 2012 for this demo, this is the same PluralsightBook website that we looked at the end of the previous module though. What we want to do is we want going to add the ability for users to add friends to their account. If we go ahead and run this, this is what you should see when you run this code out of your before folder. When you try to login, you'll be creating the ASPNETDB database in your app data folder when you do this, there are no users currently 'cause I don't have a database. So I'm going to go ahead and register, say, ssmith, and an email, this takes a moment because it's creating the database but now you're in and theoretically, the profile functionality should still work as well so you could put in-- let's start with Fritz and all this works. Now, the next thing we want to do is add support for adding friends. So let's come up really quick to do this. We need to add a menu link to our friends page, we need to create that page, we want to show our current friends on that page, we need to link to that add friend page from the current friends page so that we can add a friend. We could do this with some jQuery or AJAX or something like that but to keep things simple, we're going to just create a separate page. I'm going to create that page and then we'll also implement removal of friends from the current friends page. Now, somewhere along the lines here, we're also going to have to start storing those friends somewhere in persistence. And for that, we're going to use Entity Framework to add a new entity data model still talking to the same ASPNETDB database. So the first step here is very easy, we're going to add that menu link. So for that, you just go into Site. Master, find our menu which is right here with these menu items, and we'll just copy one, paste it back in, we're going to call it friends, with text of friends. And now, if we refresh the web page, you'll see we have a friends menu item. But if we click on it, it's not there yet. So going back to our to-do list, we've done that first one so we'll make that an X and we'll say the next one is to create a friends page. So for that, we're just going to right-click, I'm going to add a new item, and we want to add a web form with a Master page. And now, we're ready to go ahead and add our content. So our content that we're going to have is a grid view so we just go ahead and create each one for our friends and let me just save you a little bit of time here and we'll type out this grid view real quick. So there, I've created a grid view, a simply an empty data template with "You have no friends" and I've also copied that up here so that displays for now. Until we add a data source to this grid view, it's not going to actually show this data template. But if we run it now, we can see that it displays-- we have a friends page, it says, "You have no friends. " It's very sad. Now, we need to actually add some way to get that data. So at this point, we're going to add our Entity Framework EDMX file. We're just going to add that to the code, we're going to add a new item under data, we'll run an ADO. NET Entity Data Model. And in this case, we're going to call those-- since it's PluralsightBook, we call it PSB data, and we'll generate it from the database using our current database, save all the defaults, and we'll go ahead and just include everything. Now, we have our Entity Data Model, you can browse around and see it here, you can zoom in and out. The next step is for us to create our friends list. So we're just going to right click here and we're going to add a new entity, I'm going to call it friends, we'll give it a key ID, Int32, that's fine. We're going to add a new property to this guy, friends need to have an email address, and then they also need to have an association. So we're going to associate our friends entity with our ASPNET membership table-- ASP. NET membership has our user ID that we want to have so there might be one of these too many friends 'cause each user could have many friends. We don't need a navigation property on membership but we will take one on this side, we're going to add the foreign key to the friends entity, and say, okay. And we want to rename this so the simply, the user ID, and then we can generate the SQL for this so we could say generate database from model and this will generate a fairly large bit of SQL that includes everything. The only one of these that we actually care about though is the table we just added. The somewhat simplified version looks like this, so now, we have just the friends table, the primary key on friends, the foreign key on friends and an index. So now we want to connect this with back out here to that server, grab that connection string, and now come in here options, additional, paste it in. And at this point, we can actually run this, and we don't want to call it ASPNETDB, so we get rid of that, and otherwise we're good, it should check, it should run. And now if we go in and look at our data, we should see two things. We should see first of all that there is a friends table which there is. And we should also see, since I just registered not long ago with my username, when I show table data, this table's data should still be present, and it is. So I can snug my user ID here 'cause I need that in my friends table. And if I view the Show Table Data there, I can add some friends so I need to put in my user ID and some friend email address. So I'll say friend@domain. com and another one, okay? So now we got some data to work with. And now we're ready to go back to our actual page that we were working on. So let's close all these and return back to our friends page where at this point, we're ready to actually add a Data Source. Since we're using Entity Framework which I'm showing you here, we can create an ASP Entity Data Source and it looks like this. Once we have our Entity Data Source, we want to go ahead and add a couple of things up here so we're going to set a DataSourceID equals DataSource1, and we're going to add-- we're going to AutoGenerateColumns for now. So let's see what this looks like. All right, so you can see we've got our data being displayed. Let's clean that up a little bit now. We really only want to show the email addresses. So to do that we're going to add some columns and we're going to add a BoundField and then we'll also turn off AutoGenerateColumns. And let's see what that looks like and now we just have a list of friends, perfect. Now some of you may have noticed that I did not put in any sort of a WHERE clause on this. So if I come in here as me, I'll see that list. If I logout, I'll still see that list. And if I login as anybody else, I'll still see that list. So this is not filtering by my user ID. So let's go ahead and fix that. We need to add a WHERE statement. So for this, we're going to say WHERE equals IT which referring to the current entity dot UserId equals @UserId which will be our parameter. And we need to add some parameters now so we're going to say, here's my WhereParameters. And for this, we're going to need to get little tricky because the list of ASP colon parameters we have does not include anything that relates to the current user. So instead, what I'm going to do is create a custom type that does that for me and just grabs the user ID. So we're going to create a Pluralsight Book type called UserParameter and we're going to name that UserId. Now to get that Pluralsight Book type to work, we're going to have to register it and of course we have to add the code as well which I'll get to in a second. When we register that type, we need to give it the Namespace which is equal to our current Namespace, we're going to put it in the code folder and give it that TagPrefix which was just PSB, and then lastly, it needs to know the assembly even though it's the current assembly, it's still needs to know. So PluralsightBookWebsite should do the trick. Now let's go have that code so in my code file, we're going to add a new class. We're going to name it UserParameter and this user parameter is actually going to inherit from the parameter as we'll see in just a second. So-- and for that we need a Namespace. And now we just need to override the evaluate method. So let's say overwrite evaluate method. So let's say override the evaluate method and in here, we're going to snug the current UserId. So we're going to say, var currentUser equals Membership which we don't have yet, so we'll grab it, dot GetUser. This. DbType we have to set this for the Entity Framework stuff to work equals system. data. DBtype. Guid and finally we're going to return the current user. ProviderUserKey which will be that Guid of our UserId. Now this should build and at this point, if we view it again, go to friends, still having issues now it's because we're not logged in. So we can't really fix that just yet except the logging in, that's all right. So I'm going to login as me, and now if we click on friends, it shows that. If we logout and login as someone else, so register as somebody else and go to friends, we should see our empty data template. That tells us we can get rid of our old thing here as well. One last thing is get rid of that initial P tag and we're good. Now we don't want non-authenticated users to get to this 'cause obviously if they do currently, when they logout, if they come here, they get this nice ugly error page but we'll fix that up in just a moment in our web config so that non-authenticated users can't actually get to the page. So where does that put us on our to-do list? At this point, we've finished our create friends page so we can mark that with an X and we are showing our current friends on that page. We need to link to the AddFriend page and we'll do that by just adding an anchor up here. And then we can quickly mark that one off of our to-do list. So now I just need to create the add friend page. So we're just going to add another web form with a Master Page. And all of this, it doesn't do anything yet, we can test it out. So we see we have our page but doesn't do a whole lot yet. So in our main content, we need to add a few things, we'll give the-- basically we need an H1 to tell us that it's an add friend. We'll need a label, a text box, and a button so it's going to look pretty much like this. We'll wire off the button to a SaveButton click event which we can now do in Visual Studio 2012 in the text editor. And if we then navigate it to the code file, let's see our SaveButton click event here. Here we're going to simply add some code to persist this to Entity Framework, and then we're ready to redirect back to our friends page. We can build this and test it out. This is only going to work if we log in, go to friends, click on add a friend, and let's add a friend03 and you can see it works. So now we can go back to our to-do list and mark off the create AddFriend page. At this point, our last step is to implement the remove functionality. We're just going to add a column here to our grid view that lets us do that. So we need to go back to our friends page and adjust our columns a little bit. To add the remove link, we're simply going to go in and add here a command field. The command field is going to have a header with remove button type of link and we're going to tell it that we want to display the delete button. Now in order for this to work with the grid view, we need to specify the data key names, in this case that's just ID to read the ID on the friends table. And lastly, we need to tell the Data Source that we're going to support deleting. So we have to say EnableDelete is true. Now with that, if we run our page again, we should be able to delete a friend, and you see it works. We can go back, we can add a friend. So now we have add and delete functionality working. Let's go back to our to-do list and we're done with the current set of tasks.

More Requirements

Now that we have a basic system for managing our friends in place, the next thing that we want to add in terms of requirements is the capability to send out notifications whenever we add a new friend. Specifically, we want the application to send users who are already friends send email notifying them that we have, you know, verified our friendship. It's now a reciprocal friendship. We need to send users who have not yet signed up for Pluralsight Book, an invitation to register with the site and to add back the friend that just added them. And then application users should also be sent a link to add a friend if they're not already friends. So it's basically these three scenarios here. If they're not a user, then we have to let them have a chance to register so that they can become a user. This is how we're going to spread like a virus through the internet. If they are a user and they're already our friend, we're just going to notify them and say, "Hey guess what, they're your friend too". And if they're already a user but they're not our friend, we're going to send them a link and say, "So and so added you as a friend. Would you like to be their friend too? " Then we want to have in the email the ability for someone to just click a link to quickly add that friend so they don't have to go back into the form, type in their email address and go through that separate Workflow. So we're going to have a new Workflow that just does everything through the URL.

Demo: Adding Notifications

So let's get started. First, let's come up with our list of tasks. The first thing we should do is make it so that we can send an email whenever a friend is added via our Add a friend page. We have our three separate cases for non-members, non-friend members, and friend members. And we want to have a link inside these emails that goes to a particular page and has the email that we wish to add so that members can click on that link and immediately have that friend added. We need to implement that page which we're going to call QuickAddFriend. aspx, and then the other thing is if the user isn't already logged in, we don't want the page to just blow up or otherwise fail so we want to add a little bit more to the web application to make sure that the authentication is setup so that it will automatically just prompt the user to login when required. So the first thing we need to do is go into our AddFriend code where we're doing all of these addition and we're going to add in our notification here. And I'm going to go a little bit faster in terms of adding code snippets and then just walking through them so you don't have to watch me type quite as much. So here's the code that we're going to use to pretend to send these notification emails. I'll give you a quick look at the Namespaces up here because I just added a couple. So I've got System Diagnostics and also this link to code because I'm using this profile object that we had earlier. This is our code we started with that simply adding it to the database. Now we need to send this notification. So the first thing we need to test is whether or not they're a member of our site so we're going to do a check to see if we can look up that username from a membership provider. If they are a member then there's one of two choices. Either they're already friends with this user or they're not. If the current user is already a friend, we're just going to send them an email that says, good news, your friend just added you and pass it-- pass in their email. Otherwise, we're going to tell them that, so and so added them as a friend and that they can click here to do the same. Finally, if they are not a member of the site already, we're going to be in this else block and this is where we're going to tell them that someone has added them, they can click to register their own account. And then we're going to do a Debug. Print statement to pretend to send this email. So we're not actually going to send them emails but we'll be able to see in our development environment whether or not it's acting as if it were sending these email messages. Go ahead and test this out now? You can view your debug if you debug your system in Visual Studio. You can come down here and say, show output from debug. So now that we're running, let's bring up our browser and let's add a friend. And now you can see here in our output that we sent that email. This, because of the fact that it's telling him, "Click here to register, " that's telling us that this is actually a non-member which is correct. So it sent a correct email and this appears to be working. You can take my word that the rest of these things work. So let's go back now to my to-do list. We've implemented the sending of emails, we've verified that non-members get the registration mail. We haven't actually verified that these two items work but you can take my word for it, they're implemented. And the emails have this link now to QuickAddFriend. aspx but we haven't implemented it yet. So let's do that next. To do that, we just to add another web form. (Pause) Within this, we're going to give it a very simple user interface. Basically, we just want to be able to show a label saying that this was successful or an error message if it wasn't. Now in the page load, we basically want to do the exact same work that we did inside of our AddFriend button click event. So we're going to go and use that best practice of coding called copy-paste coding. We're going to copy all these code, well, except for the redirect and paste it into our page load and then from there, we're going to get some errors, some things that don't wok, we're going to add some Namespaces and fix up some stuff here so that we're using our query string. And we'll do that anywhere else that we need it. So again some more copy-paste. I am being sarcastic when I say this a best practice, this is something you want to avoid doing. It violates the don't-repeat-yourself principle which I covered in some detail in the solid principles of object oriented development in design course. We need one more Namespace and Namespace for debug. Now we're in good shape. So now if we run this again and we invite a friend, if we look at our debug output, we can see here that we've added this link. So we're going to save that link to our clipboard, we're going to come back to that. We need to register with the email address that we just used which is Friend04 so we'll logout, come back in and register. Now we're Friend04 and we're going to paste that link in. So when you create a link expression, you can't put in a function which is what I was trying to do here. We just need to pull out the email address into a string. So we're going to stop debugging for a second and come in here and say string emailAddress, let's call it currentUserEmail, equals Membership. GetUser. Email and then put that in here currentUserEmail, right? And run this again, paste back in our link, and we didn't actually display our label, it was the other thing that we didn't do so I'll have to go that, but let's see if it actually sent the thing so here we go. So Good News! Friend04 just added you as a friend. It's working and it actually work twice. The one thing that we would want to do also here is on our QuickAddFriend, we want to add our comment to our success label to note that it was successful. Now let's go back to our to-do list. We've implemented QuickAddFriend, it seems to work. You know, there's one last thing that I-- we didn't actually do there. Hmm, anybody else catch that? So we pulled this out here but we didn't ever actually test this in our AddFriend page and I don't think we pulled that out there. So let's look at this code here and go back to add friend and look at my code up, yeah, look, here is my link statement and it's going to fail the same way so I'd better fix this here as well. So now that's good but this duplication between these two classes, these two pages is already causing me bugs. So that's going to become a problem. All right, so now let's hope that that works. We'll talk about that in a minute. We'll come back to our last task which is the authentication rules. So to demonstrate what the issue is there, what we need to do is basically look at our friends page. Here's my friends page, the way we couple these duplicates. If I logout and go back to the friends page, as you may recall, it blows up and it looks like this because we don't have this CurrentUser. ProviderUserKey. Now we should be able to count on that being there. That's not actually in our page, that's in our custom user parameter code that we created and it should only ever get called if the users logged in, it doesn't make sense for a non-authenticated user to come to the friends page. So we're going to go into web config to fix because ASP. NET supports-- setting up authentication rules in the web config. So open up web config and here is what we're going to add. First, we need to add an authorization and we're going to specify that we're going to deny everybody. So we're going to say deny here users equal question mark. That means anyone who's not authenticated. And then we're going to go down to the bottom of the page, we're going to add some exceptions to that. So I had a location and here, we're going to say that for the default. aspx page we're going to allow any user. Now having saved that, if we run the application one more time, we try and go to the friends page, it will redirect us automatically now to the login page, same thing for profile. So anything we try and get to other than the home page, it's going to redirect us now to the login page because we're not authenticated. If we are authenticated, everything works as we expect. So that completes all of our tasks, let's go ahead and think about how our code is working right now and jump back in the slides and kind of do some analysis.

Problems with Design

So let's talk about some of the problems that we encountered as we extended the functionality of our application. The first one was that we don't really have an easy way to test our notification business logic. We've kind of gotten around that by just using a debug system and we could also use a tool like smtp4dev that would actually send the emails that let us capture them on our local machine. Both of these are, are good ways to test it but they only support manual testing and they don't provide us with the full confidence that we would have if we had an automated test for this that we could just put on a build script or a build server. The other problem with this is that our business of deciding who should get which notifications and whether or not another notification should be sent are tightly coupled with our data access logic but they're also duplicated with our two different ways of achieving this, the AddFriend interface and the QuickAddFriend interface, one of which uses a text box and the other uses a query string. In both of these cases, the notification system, any change to that, is tightly coupled to the user interface code so that would have to change. Furthermore, our decision to use Entity Framework and our local SQL database means that our data access code here also is tightly coupled to our user interface code. Because we're doing all of these operations completely within, either the page load event or the button click event. So let's look at how we can do some refactoring to do a little bit of separation of concerns and move some things into their own locations so that we get a much better design that is more maintainable, more flexible and less likely to suffer from bugs that are result of duplicate logic.

Demo: Refactoring to N-Tier

We're going to refactor this into two different assemblies, a Business Logic Layer and a Data Acces Layer which we're going to add a separate class library projects to our solution. So the first thing we need to do is add two class libraries to our solution. So we're going to add a new project, we're going to add a Business Logic Layer or BLL project. (Pause) And a Data Access Layer or DAL project so that they're empty. And then the other thing we're going to do is we're going to reference these from our web project which is currently. NET 4 so we're going to just make these. NET4 as well. (Pause) You could probably make them all. NET4 or 5 but since I know some of you will be using Visual Studio 2010, I'm going to make them. NET4 so that these projects should work for you as well. Now we can start refactoring and we can start pushing things into the assembly where they belong. If it has to do with business rules, we want to push that into the business logic layer. If it has to do with data access, we want to push it into the data access layer. So to begin with, our biggest data access layer item is of course our Entity Framework stuff. So we want to pull the Entity Framework data stuff out of our code folder and push that into our data access layer. So pretty much, you can just cut and paste that code. We also need to update our references so we'll add those now as well so in our solution, we want to reference the business logic layer and the data access layer from the web, and the business logic layer wants to reference the data access layer. So let's try and build now, we probably going to have some problems. Actually we're good because we didn't update our Namespace, that's fine, we'll leave that alone. Now when I go back to our AddFriends page and think about what this should actually be doing. So when we have the save button click and we want to add a friend, we should call some kind of a service that knows how to do that. So perhaps we would have a friendsService that we would know of. (Pause) And we would say friendsService. AddFriend and pass it in the thing that we want to add as well as who we are. So maybe we would give it the current user ID and the email text box. Looking down here to see what types of things we're using, we need the memberships user's email, we maybe need our current user's name and that's about it. So we'll pass in-- let's make some variables out of this, right? Now we have the service, we want to pass it in these things, we'll say, we're going to pass in our currentUserEmail, that let us look up who the user is, currentUserName as well, and our friendEmail. Now assuming that that works, all we need to do at that point is do a redirect. So we can take all the rest of this code and put it inside that service. So we'll cut that out, and we're going to go create now this FriendService, capital F, and we'll just create it right here for now, so generate. That generated the class for us here. We also want to create a method stub for this AddFriend, generate method stub. And at this point, we can paste in our implementation. Now here too, we have some data access code here intermingled with our business logic. So this right here might belong inside of an AddFriend data access service or we might decide that working directly with Entity Framework is okay for our business logic layer at this point as long as Entity Framework itself is in a different assembly. I would probably lean toward having specific operations exposed as their own interface and types because that makes it a little easier to test and maybe a little easier to mock out later. But really the only way we're going to be able to test this the way I've had it designed right now is with integration tests anyway. So I'm not super worried about that. So we'll just leave this pretty much as it is except we're going to take this class and move it to our business logic layer. So our friend service, we'll just cut, we're going to allow and paste and that will cause us then to probably need to update to fix the

Namespace here and the Namespace here. Now, at this point, we need to fix a few errors. So back in here in our service, we've broken a few things, we need to add some references so we don't have system data entity, we're going to need that so we definitely need to add it to the BLL. So add a reference to the assembly membership. We can get around the need for this if we just pass in the user key. Alternately, we can look it up from Entity Framework using the email that we have. For now, we're going to go ahead and just get this off the website and pass it in and pass them so we'll make that the first one. In here, we just needed a Guid currentUserId so we can pass that in here and then anywhere we need the new FriendEmail address, we'll pass in this FriendEmail instead of the text box. Now at this point, we still need to call a membership on the FriendsEmail so we'll work on that in a second. And then we'll need the FriendsUserId so we need to get the friends user information somehow. And we should be pretty close. Four errors, they're all membership related except for this BLL which won't work until we get that to build. So we need to get the FriendsEmail and verify whether or not they're a member. So really that's just a separate query and we can say is friend a member, we can extract that out as a separate private, check to see if an email. So we'll say IsEmailRegistered. Now do you do that, we just need another context and then we can just do a simple link query. And basically this says, are there any rows inside of the ASPNET membership table that have this email? Now that we know is email registered, we can do IsFriendMember equals IsEmailRegistered friendEmail. And if we have to actually get the user ID, well that's going to probably just be a user service. So why don't we create another service for that and just do it right here. Now again, I want to stress that I'm sort of refactoring without a net here because I don't have tests. This is not what I would recommend that you do but it's what I'm able to do right now because I don't have these tests. And it's also important to note that I'm doing this on a very, very, very simple site. If you try and do this refactoring when you already have thousands of lines of code on a tightly coupled web forms project, it's going to be a lot more painful and a lot more likely to cause problems. So it sounds to me like this IsEmailRegistered doesn't really belong with friends so we're going to move that and throw it over here on the user service, make it public, and then this just becomes-- then we also need to see if we'll be able to get user. Now the type for this is going to actually be our entity type. So this is going to be code. ASPNET membership. That's the actual row that it is going to get us. So a small linked query again to just grab the first or default row that matches the email. (Pause) And since we already have a user service here, we can say that the FriendUserID is just-- and then our build succeeds. Now again, we don't have error handling code, this is not best practice code at all but this is kind of just showing us how we would go about refactoring this. We do still have some low label database access code happening right here, I'm going to say that we're going to leave that there for now in the interest of keeping this module from getting too much longer. But normally like I said I would probably move this stuff into its own data access layer repository style class that would have a method called add friend that would just add this particular data. And so your friend service in your business logic layer would very frequently call an add friend method on your data access layer and do the additional business logic here. Now let's move the user service to its own file. And we'll also clean up this Namespace, it shouldn't have website in the name. In our EDMX file, we're also going to want to get rid of the word website so that these things all work correctly. We really just change. code to. DAL and then with these references here will just be. DAL. (Pause)

And then we have do a quick replace on names. (Pause) Now we come to our other page QuickAddFriend. QuickAddFriend, we haven't yet updated but now we can take the much simpler AddFriend page and modify it so that it works for this. So we're going to come back here, look at how much code we've gotten rid of. We can take these exact same parameters, that same code, come back in here and paste it in, and adjust as needed. So in this case, we don't have a text box so that's going to be our query string, we need a Namespace, and everything else can go except for our status message at the end. So now, this has become much simpler as well. And what if we have an error in how our notification logic works? Well at this point, it's all in one place. Now this method is still way too big so I would certainly extract some methods from this and refactor it. In fact, I would probably create a notification service so that this guy here that says send notification email would really just be a method. And you can do that easily enough if you just grab all that code and right click and say refactor extract method, and this is send notification, and now we have a separate method. Now sending notifications doesn't really belong on the friend service so I would also move this to its own class. So let's just grab it and create a new notification service and anything on here that was private, now it needs to be public, and we just need to create one of these. So var notification service equals new notification service. And I don't like statics so take that off and then we just move this to its own file and then paste over what it gives us. All right, so we build. So let's review our code. At this point, if we close everything down, we come in and we find that we have an add friend page with a button when you click on the button, we're going to grab the parameters we need, we're going to new up the service that we need to use, we're going to call one method on that service and redirect. When we send those emails, they're going to have links to QuickAddFriend which is going to use a query string. When we look at that, it's going to be very similar, it's going to grab the fields it needs, create the collaborating objects or services that it requires, and set whatever label it has. It's only connecting to the business logic layer in both cases. In fact we don't even need to have a reference to the data access layer. When I added this, I didn't really have to, in fact we can delete it and everything should still work. Then we drill down here to the business logic layer and we see that we have a reasonable separation of concerns. The main thing that we've been working with is this friend service. The friend service has an AddFriend method. The AddFriend method does a little bit of data access. Again, I probably push this into a data access layer method and does some notification stuff. Does it need to know all the details of how notifications work? No, we push those into their own service here and even this can probably be broken down so that it has fewer IF statements and a little bit more simplification. We can get rid some of these comments now whether or not there are member, it's obvious because of the Boolean so we complete that comment. So this is longer than I would like, this one I would still clean up but we're running out of time here. And then the user service again is extremely simple. We can check for whether an email is registered, we can get a user. And then if we look at our data access layer, it's just our entity data model. Now, if we run this, let's verify that it works, there's one known problem that I want to point out. The first problem that you may run into is that when you're not logged in, you're unable to see any of your styles and your site looks something like this. However when you login, everything is fine again. There's a few ways you can fix this. The simplest one is to go and create a web. config file that looks like this one and make sure that you have it inside of any of the folders that have common file types that you want to be available to all users. So if we copy this webconfig to our styles folder, and then go back to our page and logout, you can see that now, even when we're not logged in, our styles persist. The other problem though is when we are logged in, because of where we moved our Entity Framework stuff, we need to fix one of our connection strings. So when I go to friends, I'm going to get this nasty error here. If you see this, there're a couple of things you can do to fix it. The one that we need to do in this case is go to our webconfig and note in here where this metadata is inside our connection string, we have three files and they all start with this code dot because that's where we originally had our file. So we're going to delete that because we changed the Namespace. We can do that three times and save and that should clear up this problem. And now we are able to get through and our Entity Framework stuff is working from our data access layer. Another thing that may work for you if need to try it if you're getting stuck with that error is come back into your entity data model and go ahead and do another generate database for model and that will do a lot of a setup for you to make sure that you've got an appconfig and you've got all these things in that folder correctly. Now there is one last area that we haven't looked at and that is our actual friends page. If we go back to that page, you'll see that we implemented all of our logic inside of the ASPX code. There's nothing inside the code behind and therefore there's nothing for our compiler to catch when we move around things like our Entity Framework. Now we saw that this is still working but the challenge here is that this is still tightly coupled to our Data Source. So we would really want to replace this with an object Data Source and have that talk to a service that would live inside our business logic layer. That would eliminate that coupling from this page to the data access layer. These are going to be one of the things you'll have to look for manually if you perform this kind of refactoring yourself. And actually fixing this is something I'll leave as an exercise for the reader if you're so inclined. So let's analyze what we've done.

Analysis

Now that we've refactored things into separate assemblies for data access and business logic, we have all of our data access in a single location in one project. All our business logic in a separate location in another project and no more business logic or data access logic in the user interface with the exception of that Entity Data Source. Now our user interface may require more code 'cause we can't use tooling like that built-in Entity Data Source directly. But we do have support for things like Object Data Source that are almost as easy to use. It's also now possible to test our data access code completely independent of the user interface or the business layer but it would require a separate testing database. And as it stands now, I wouldn't bother with testing it because it's simply going to be testing Entity Framework at that point. Assuming that you trust the Entity Framework works, there's no need to write continuous test for it. You might write a test for it once just to make sure everything is wired up correctly. But I wouldn't run them as part of your test suite that you do before every check in. However, you're also able to test your business logic and that is something that it's important that you test on a continuous basis if you're following best practices. However, the way it's written right now, our business layer is tightly coupled to our data access layer which means that the only way it's going to work is if that data access layer has a database to talk to. So you may need a test database in order for your test suite to run correctly. In the next module, we'll look at how we can setup an interior application so that it doesn't have this kind of tight coupling and you're able to test the different components independent of one another.

Summary

We started this module with the goal of adding a couple of simple features to our existing application. Initially, we were able to very quickly add these to the user interface. However, as business logic and additional requirements were added to the project, we very quickly started to have pain due to duplication and tight coupling between the different concerns such as presentation logic, data access, and our business logic. We were able to eliminate this duplication by refactoring and when we did this refactoring, we also chose to separate the concerns of user interface, business logic, and data access into separate areas of our code, in this case, different projects and assemblies. It is best when you perform this that you make sure that the communication between these different layers uses a well-defined cohesive interface. That is you want to have specific methods for the things that the calling code requires rather than exposing the entire interface to it. In our case right now for instance, the business logic layer is able to access anything on that Entity Framework object. If we wanted to follow this last bit of advice, we would create data access layer services or repositories and you can learn more about repositories in our design patterns library, and these services or repositories would expose only those function calls that the business logic layer actually was using rather than exposing the world the Entity Framework interface. For more on object oriented design in Pluralsight, I recommend my course on the solid principles of object oriented design. And thank you very much. This has been another Pluralsight module from me, Steve Smith. You'll find me online @ardalis. com or on Twitter as Ardalis.

Domain-Centric N-Tier Design

Introduction

Hi, this is Steve Smith. And in this module we're going to continue Part 1 of my Creating N-Tier Application in C# course. In this module we'll be considering a Domain-Centric approach to N-Tier system design. Let's get started. If you've been following along, you've seen how N-Tier application evolved, and how traditional data centric N-Tier systems are designed. One of the problems with this approach is that it tends to lead to a significant amount of coupling between the applications at every tier in the database. In this module, we'll se how you can invert this dependence on external system like databases, so that the application can remain more loosely coupled, maintainable, and testable.

Module Overview

In this module, we'll examine what is meant by Domain-Centric design, and examine some principles and patterns related to this designed of approach. Then we'll refactor our existing N-Tier PluralsightBook application in order to invert its dependencies and produce a more Domain-Centric design. We'll be able to add some unit test that don't depend on external infrastructure, as a result of this refactoring. And then we'll analyze what does new design has done for us. Finally we'll wrap up with a quick look at what a Green Field or brand new ASP. NET MVC solution might look like when following this design approach.

Domain-Centric Design

Domain-Centric Design differs from Data-Centric Design and that its emphasis is on the domain. The domain simply describes the problem-space of your application. Domain-Centric design in as object-oriented design approach, and it such, it puts domain objects that model the problem domain at the center of the design. Everything else in the system we'll depend on this module object. Domain objects should describe the problem domain. It includes any in all business logic and behavior required to model the application. The object should include any state required to properly model the system's behavior, but they should not depend on external infrastructure concerns such as databases or files system. As such, they should not be responsible for their own persistence. We'll cover Domain-Centric or Domain Driven Design further in a later part of this series. We're going to refactor our existing N-Tier application, in order to invert its dependencies. This is going to make it, so that we have a more flexible application. As we consider how we can refactor our current design, we'll want to keep in mind principle of object-oriented design. The most important of this will be the Dependency Inversion Principle. The Dependency Inversion Principle states that high level module should not depend on low level modules, both should depend on abstractions. And furthermore abstraction should not depend on details, but detail should depend upon abstractions. You can learn more about the Dependency Inversion Principle as well as other principles in the SOLID Principles of Object Oriented Design course in the Pluralsight catalog.

Onion Architecture

The goal of refactoring is going to be a system that places the domain model at the center of the application architecture. This particular architectural designer pattern goes by several names including Onion Architecture, which is named because of the circular layer surrounding the core. It's also known as the Hexagonal Architecture, a pretty terrible name that came from a picture like the one shown here, as well as the Ports and Adapters

Architecture, which is probably the most descriptive and academically correct of these names. The initial design was documented by Alistair Cockburn over a decade ago. While the most recent name Onion Architecture was coined by Jeffrey Palermo in 2008. If we examine this diagram shown here, you'll see that we have a domain layer represented as a green hexagon in the center that has different ports that are use to interacts with external layers such the infrastructure layer and the presentation layer, hence, the name Ports and Adapters, where each one of the layers around the center domain layer writes an adapter that is capable of communicating through the port that the domain layer exposes. A more typical layout of the Onion Architecture is shown here. In this depiction, you can see how the domain model and services live at the center of the Onion while the UI, test, and infrastructure all live at the edge, and depend inwardly on the core. You can think of this as having every can Centric Circle depending only on the circle inside of it. So the infrastructure has no other things depending on it, in the application. Of course, you don't have to change the traditional layer cake diagram used to describe application layers, and replace it with hexagon, or onion circles. If you show the Onion Architecture, using Traditional layers it looks like this. The orange arrows represent dependency, as you can see there is no transitive dependency between the UI layer or test in the database, which means that we can test this application without the need to setup brittle, slow, and difficult to test infrastructure. The result will be a match more loosely coupled system with separate modules that can truly be swapped in and out and tested in isolation from another. In order to properly use the Onion Architecture System, it's important to follow certain principles. The first guiding principle when following the Onion Architecture Pattern is that you should ensure that all code is written to only reference code at the same layer, or layers closer to the center. The Domain Model objects themselves should all live at the center and thus should not have any dependency on infrastructure concerns. Inner layers defined the interfaces that implemented by outer layers. These interfaces are the abstractions that are described in the definition of the Dependency Inversion Principle, and their implementations are the details described in that definition. Behavior and services should be layered around the Domain Model. Some services will live in the application core, while other Implementations will need to live in infrastructure because of their dependencies. Strive to push as much dependency free code into the core as possible. Push infrastructure in UI concerns to the edge of the design and minimize the amount of code in these layers. These layers should always depend on the application core, and not vice-versa. By minimizing the code that you have in these layers, you'll maximize the amount of code that you can easily unit test without any type of dependencies.

Demo

Now it's time to return to our simple application, the PluralsightBook social website. Currently in this application has user registration in log in, user profile, and profile updating, as well as the ability to add and remove friend, implemented using the N-Tier design pattern. In this demo, we will attempt to invert the dependencies that currently exist between the UI and business logic layers, and the DAL or Data Access Layer. We'll rename our new projects in accordance with standard Domain-Centric design naming methods, specifically core and infrastructure. Once these projects have been created and their dependencies are established to be a project reference, we'll move to behavior from our existence layers into the new projects. Inverting dependencies using interfaces and a few design patterns like the strategy pattern as we go. To start with we're going to create a couple new projects that we're going to start moving behavior into. So in our solution which you can see, it has a business logic layer and a data access layer, as well our web-based user interface layer. We're going to add two new projects. First project that we're going to add is going to be a core project, and it just needs to be a class library, and we'll follow our existing naming convention. That's going to be a. NET 4. We're going to call it PluralsightBook. core. And then we'll add a second one for our infrastructure. Next, we need to make sure that our references are setup correctly. We don't want the core project to reference any other project. But we do want infrastructure to reference core. So add a reference to core. Our website project needs to know about both the core and infrastructure. And now we're done. We've setup the proper dependency structure. So that the core has no dependencies and infrastructure depends on core, not vice-versa. The website depends on core, but we'll need to be able to implement things from the infrastructure project. Now let's look at how we can invert some of the dependencies that exist currently in our business logic layer. If we look first at our notification service, we'll see that this needs to be able to use, the UserService, so it has dependency there. And eventually we need to send email, where right now just doing a debug print. So actual sending of emails would be an infrastructure concern. Since this depends on UserService let's look at that one first. UserService depends on our data access entity framework repository here of aspnetdbentities, and that would need to exist in our infrastructure. Now a common pattern to use when talking about infrastructure concerns related to persistence is the repository pattern. So in this case, I could create a repository to encapsulate some of these methods. In this case though, none of the usage of to do data access layer represent the standard crud type operations that typically repository implements. So for now, I'm going to just implement my own interface that represents these exact operations. Since it seems like both of the operations here are using the email as the basis for their queries. I'll go ahead and create a query interface that notes that it needs to use email. Now we've created an interface that represents the operations we need. So next, it's time to use the strategy pattern to implement this interface. It's important to note though that we're using a type that is only defined inside of the Data Access Layer for our return type on this interface. We're going to have to fix that before we can process with this. Since we don't actually want to change the UserService that's located in the business logic layer, let's go ahead and move this interface into our core. And start writing our own implementation of the service that doesn't have any of these dependencies in the core. It's typical to create a folder for your interfaces in core. Now you see the issue with aspnet_Membership return type. We're going to have this return instead one of core model types. We don't currently have any of those so let's create the next. Again, it's typical to create a folder for your model objects. (Pause) For now we'll just create a class called user. Now we don't need to define anything in it, we just need to show that it can be returned. (Pause) At this point, our interface works. Now we need to implement the interface somewhere. Since this only has dependency on some implementation of this interface, we can actually implement this entire service class in our core. Again, it's typical in this case if you have a service that does not maintain any state to put those inside of their own folder. (Pause) Our user service class should have the same signature as the one that's in the BLL. So we'll start by just copying that implementation. We're going to change this to use our user type, and we'll also change this to use an implementation of our query interface that we're going to implement in the infrastructure. Now when we say it will depend on this that means it's going to take one in through its constructor. (Pause) And we'll set this to a private variable. Now we can simply use this dependency to implement the methods of this class. (Pause) At this point our build succeeds. Now we need to implement this, and the implementation of the IQueryUsersByEmail should live in our infrastructure. We'll create a new folder for data concerns, and then we'll add a class. For now, we'll call it QueryUserByEmail. And since this implementation is going to use entity framework, we'll prefix it with EF. It will implement the interface. We can actually take the implementation directly from our-- in original service. (Pause) Now in this case the aspnetdbentities does not yet exist. We need to store this as well in our infrastructure. So going back to our data access layer, we're going to grab that EDMX file, and we have to config, and simply copy these into our infrastructure data folder, after config should go into the root of infrastructure, so we'll just move it there. We'll update the name space used by the EDMX file so that it does not use DAL any longer, and instead uses PluralsightBook infrastructure data. (Pause) At this point we have a problem because we are trying to return a core model user, but our aspnet_Membership type is returning an aspnet_Membership object. What we need to do in order to resolve this is simply transform object that we're getting into the one that we want to return. To do those we'll use the link select operator. In this example we're projecting from the entity type defined in the entity framework EDMX file in the infrastructure to a domain type inside of our actual class that is defined inside of the infrastructure project. This is because of the fact that we used the EDMX style of entity framework. If we had used the code first style of entity framework, we might have placed the actual entities inside of the core and then only place in the implementation of entity framework inside of infrastructure that would will allow us to avoid doing this projection here inside of our infrastructure class or project and rather do it inside of the core or to avoid having do it at all. Now let's go and see where we might use this particular new service, the user service that we just created inside of our application. Let's go and find the old UserService and locate all references. It appears that the only reference to this is in the notification service that we look at first. So now we can use the same approach that we did before to inject in this user service. We need to define our own notification service inside of the core and when we do we'll have a dependency on our UserService. Let's go ahead and take the signature for this class and we'll use it inside of our core implementation. (Pause) In our implementation, we certainly won't be passing in the context to entity framework as part of our method signature so we'll delete that. However let's take the rest of the implementation and use that as our starting as we refactor. (Pause) So right away we see that we need a UserService. And our UserService takes in a dependancy. So we can't just new it up as we are here. I think it's important that classes should identify the things they depend on in their constructor. So we're going to specify here that we have this dependancy in IUserService. In our case it appears that UserService isn't really doing a whole lot except passing through to our-- to our thing that we're injecting so it's not really adding any value. At that point I think we could actually just get rid of the UserService. This is something it's okay to do as you refactoring because you're learning about how the application works and coming up with a better design. So in this case we've determine that UserService isn't really adding any value, we can just remove it. What does that mean for us here? Well, that means that instead of using an IUserService, we really want to use that IQueryUserByEmail's interface. So we'll do the same thing we did before and implement that interface as the dependancy. Then instead of calling it UserService down here, we'll simply use our new private field QueryUsersByEmail. We give a slightly different signature. (Pause) And it turns out that we do have a method that is missing here which is our context call where we'd looking for any friends that have this particular query associated with them. The other thing is that we don't currently have a UserId defined on our user type. This is easy to rectify. (Pause) That clears up that compiler problem. So next we have this check to see whether the current user is a friend. This looks something that could certainly live somewhere else. It's a fairly complex query to do just in the middle of this already long send notification method. So I think we should refactor this out into a separate class. Now in this case, the thing we're doing is we're checking the friends of our existing user to see whether or not they already list the current user as one of their friends. This seems like something that we could put inside of a service. And since we're doing a check on the user email I think it would be one that we could add to our existing QueryUsersByEmail interface. The two parameters that it needs are the email and the current UserId. (Pause) We'll create this new method and then we'll need to implement it inside of the infrastructure class. We can simply take the existing logic and copy it right into that method. So moving to infrastructure now where we have this service already. We can update our implementation of the interface and simply return result to this query. Now we need to adjust a couple of the parameters here. Now that we've implemented this method we can call this from our notification service. (Pause) And if this why we should be able to build again. Now it turns out that we can't actually send the emails using debug right now. So we can either add system diagnostics to our core. Or if we want to keep that pure, we can actually move that implementation to a separate interface. So since this is an exercise and showing how to invert dependencies let's go ahead and do that. So when you had interface that defines how we send these emails we are doing something that just takes in a string at the moment. So we can do that by just creating a simple notification interface. Since we're pretending to send an email, we'll go ahead and call it something simple like ISendEmail. So we're going to add a new item interface and just give it a single method. Now we can implement this in our infrastructure and since this doesn't data we'll call it a service or add a new folder. (Pause) We'll call as a DebugEmailSender since it's not really sending emails. (Pause) Implement the interface and copy the implementation. Now we still haven't implemented debug here, so let's pull that in and change some parameters so we can build. Now here again we're going to replace this detail column here that's showing the implementation of what we want to do with an abstraction that we're going to effectively inject into this class. So we'll have-- we'll have an email sender but you'll have a SendEmail method. And in this case we'll just pass at the emailBody. Now we need that email sender to be defined and we need to get its implementation from somewhere, so we're going to add it through the strategy pattern in our constructor again. (Pause) And once again our build succeeds. So now we've replace the NotificationService and the UserService, let's look at the FriendsService. The FriendsService does two things when it pass the AddFriend method called it does the database insert and it actually implements NotificationService. This, because it has two responsibilities does sound like a candidate for an actual service we can place inside of our domain layer with dependencies on these other two services. Now we don't actually have an interface for a NotificationService, so we could create one or we could just pass in the specific implementation. Since they're at the same level of our application, they're both in the core, it would be okay for us to just use a new NotificationService that would be passed in directly by class reference. Depending on how we're composing our object graph that may or may not be simple to do. So I'm going to go ahead and create an interface for NotificationService because I know that it will make it easy for certain IOC containers to do their job of creating that object graph. (Pause) It's a simple refactoring to create the INotificationService which will then move into our interface's folder and update its names space. Now we're going to create a FriendService. (Pause) We'll start copying the implementation-- (Pause) -- of the add method. And in this case we can see that we're going to have two dependencies, one on the NotificationService and the other on data access layer. Since this is doing an actual CRUD operation of creating a new entity, this will be a good opportunity for us to use the repository path. (Pause) Now we've created our injection of our dependencies but we haven't yet defined what IFriendRepository looks like. This IFriendRepository simply needs to have a create or add method that takes in the different parameters that we need in this section right here. So looking at this, we can see that we need our UserId, an email address, and that appears to be it. So in our interfaces now, we're going to create this IFriendRepository that takes in a UserId and an email address. (Pause) With that create method defined, we can now go back to our FriendService and replace the call that we were doing with one that uses this repository. Likewise, we can replace the notification call with our call to add service. Note that we no longer need to pass at the context. Since we've already implemented the NotificationService, we can delete this code. For the FriendRepository, we need this implementation code when we go and implement that repository. Going to back to the infrastructure project in data, I'm going to add a new class, called the FriendRepository. Since we're using entity framework, we'll go ahead and called the EfFriendRepository and then we'll paste in our implementation. (Pause) Now we can back and we can delete that implementation code since we no longer need it and you can see that our AddFriend service now has gotten much simpler. If we were to write a test for this, the only thing it would need to verify is that it does in fact call each of these two dependencies in a particular order, assuming that we care about the order. Now let's take is all the way have to the user interface level and see what it would take to use the core instead of our business logic layer. Here's where we implement our FriendService in the business logic clear. This is what is allowing us to add a friend. Our new FriendService that we want to use just located in the core, it requires these implementations of these other interfaces when we create it. At the simplest, we can do that all in line here just as we did before. (Pause) Let's change out our dependency on BLL, use Core. Services. With that, our FriendService now is expecting these implementations. We can just new this up for it in place. We want to use an EfFriendRepository and a notification service as well. This is one also requires an implementation. Now, all of these things that are coming out of the infrastructure project require their name spaces. So we'll import those. (Pause) And we call this EfQueryUsersByEmail. So let's go ahead and rename here. (Pause) There we go and one more parenthesis and we're done. Now we've refactored out the business logical layer from the AddFriend page. Let's go ahead and test our code using the user interface. Actually there are few things we need to change about the website before it'll work. I've gone ahead and I have removed the BLL and data access layer from the solution, eliminated their references from the web project and I've also deleted the entity framework data source that was under friends. I also made a small change to web config. Because we put our EDMX inside of a subfolder data, we need to update the metadata tags here so that they have a data. prefix. So you'll put in Data. PSBData in these three spots. Having done that, we can run the application now, sign in. When we go to friends our grid won't show anything, but we can add a friend and add it without error. And at this point, you'll just have to take my word that that's working. So the next step would be for us to implement what we need for this grid. So that we're able to both list the friends of the current user as well as delete them when this delete command is called. I'll go ahead and implement this and in so doing I'll be adding a couple more interface methods and implementation methods to the core and to the EfFriendRepository and I'll just show you the results of those real quick. Now we've gone ahead and implemented the Friends page using the new domain centric design. You can now add friends (Pause) As well as delete them. To do this we modified our Friends page. So that we specified here a template for the removal indicating here in event to do that deleting and if we look at the code behind, we're binding the grid view in code now. So we specify the current user. In this case, you can see we still had dependency on membership. We haven't actually refactored that out. We create a new FriendsService and bind our grid to the ListFriendsOf the currentUser method. Then for deleting, we simply pull back the CommandArgument with the ID of the friend to delete, new up our FriendsService again, and delete the friend. Now, you notice there are some repetition here with this FriendsService, we're instantiating it here as well as here, this is something we can eliminate through the use of a container or encapsulate into a base page class that would have this FriendsService available as a property. Cleaning up some of the code like this is something that we may look at in a future course in this series.

Summary of Refactoring

Now let's summarize the steps we had to take to refactor from a traditional MT or application design to a domain-centric design. First we created new projects to house our core domain logic as well as our infrastructure. Next we move our business logic layer methods to core or infrastructures services or domain objects as appropriate. We created interfaces to represent dependencies and provided these dependencies via our classes constructor. In some cases, we identified new dependencies such as the debug implementation of SendEmail. Now we created the interface and place it in core while we created a new class to implement the behavior and place it in the infrastructure. Finally, we've removed the references to the old BLL and DAL projects from our user interface layer, updated its code to use our new services in core and provided all the needed dependencies when we instantiated the core objects in the UI code. In a real application, making use of this approach, it will be worth while to consider implementing an inversion of control container to eliminate the duplication involved in specifying the objects to instantiate as dependencies. Our design still has a few a problems. We have repetitions on how we create our services and their dependencies. Web form is also architecturally limited and that we cannot intercepts page creation so we can't specify a given pages dependencies in its page constructor. There are ways around this using other patterns which we'll discuss in a future course. The other problem is that we still do not have any test. We do have a testable architecture, but we haven't actually proven that. So as an exercise, see if you can write test for our two nontrivial methods in our core, AddFriend and SendNotification. Remember that unit test do not involve any infrastructure and should only exercise the method under test, not your whole application stack. Once you have unit test written for these two methods in the core services folder try and do the same thing for the original implementations in the BLL project and see how difficult it is to run these tests without infrastructure like a database in place.

Starting from Scratch

If you would like to adapt a domain-centric approach to creating applications with a new solution follow these steps. First, create a Blank Solution then add your Core Class Library Project and Infrastructure Class Library object and your UI project which might be anyone of the Console, WinForms, WPF, ASP. NET MVC or any other user interface front end project for your solution. Finally, add references to core from you Infrastructure and your User Interface. Once you add a test project, you want it to reference core as well. Then typically you want to add a reference to Infrastructure form UI. Note though that you may not always need to do this depending on how you implement your IOC container assuming you have one. If you want your application to be able to use the implementations located in the infrastructure project, you will need that assembly to be in the UI projects bin folder, but this can sometimes be accomplish by just copying the file rather than adding a reference in Visual Studio. If you don't have that reference, you'll be guaranteed that there won't be any direct references to classes in the infrastructure project from your user interface project and this will guarantee that all of your use of code in your user interface will be using the domain objects in your core or the interfaces that are defined there and not any of the implementations that are define infrastructure directly.

Demo

Let's look at how we would set up a domain-centric solution using an ASP. NET MVC project. Start with a Blank Solution, add a class library for core, and another for infrastructure, reference core from infrastructure, add your MVC project-- (Pause) -- references core in infrastructure from the MVC project and now you have the basic ready. To demonstrate how this would actually work, let's show a simple example. (Pause) We'll create a home controller for the home page in a simple view. We'll have the home controller use an interface that will define in core that will display a message. So add our folder for interfaces and a message provider. The message provider will simply return a message. We'll implement this in our infrastructure. (Pause) And we'll just return back the actual item. ToString. Now back in our home controller, we'll use this message provider to return our message. We haven't actually defined this yet. So we'll create a constructor. We'll specify that we need an IMessage provider. Now we have everything wired up. If we want to take this message and actually pass it to the view, we can just send it as a string and change our view to take a string. And we'll just have the message be here in bold. Now of course we need a constructor that doesn't take one of these because we need a default constructor. So for this, we'll just create one of these here. We have a default constructor where we set up the actual message provider we want to use from infrastructure. Now this will force the infrastructure one to be called every time we-- we run this. So now we can run this application. We can't use a string here. Let's just add this to the view bag-- view data and then we'll change our view back to just display that. You can't actually use the string as your strongly typed view model. So I'll run this again and you see we are using the Infrastructure. MessageProvider. If you want to change this so that it doesn't actually need to have the implementation specified here in the home controller, you can do this very easily, simply add a new IOC container such as an inject, structure map, or unity, all of which have NuGet packages that you can use. So you could come in here under NuGet Packages and you could do a search you could find for instance StructureMap searching online. And here you might find, for instance, StructureMap. MVC3 or StructureMap. MVC4 depending on whatever version of MVC you're using and it will go ahead and set everything up for you and then you can just edit the dependency tree that it needs in the IOC. CS class that it creates. But that's it. You now have a very simple implementation of a domain-centric solution using ASP. NET MVC for your user interface layer.

Summary

To summarize, we found that the domain centric design forces you to focus your application on the applications domain logic rather than on database. Typical traditional data centric N-tier design tended to force all of your dependencies in your system to point toward the database. By using the Domain-Centric design, we are able to use something called dependency injection which relies heavily on the dependency inversion principle to make it so that our application logic was at the center of the dependencies in the application. A common design pattern that we use when applying the DI principle is the strategy pattern which you can learn more about in the design pattern's library on Pluralsight. One immediate benefit we get from this domain-centric design is that we have a unit testable set of objects, both services and domain objects in our core project. It's much easier, you'll find, to begin with a domain-centric design as the basis of your solution than it is to refactor an existing system to use this approach. You may find the following references helpful as they relate to concepts discussed during this module. The first is my course on SOLID Principles of Object Oriented Design also available on Pluralsight. This course discusses ceratin principles including the dependency inversion principles that were used in this module. The second is a book "Domain Driven Design" by Eric Evans shown here. This book cover many of these concept and more as they relate to Domain-Centric or Domain Driven Design. And his book is really the best references on this topic available. And thirdly we have a post series by Jeffrey Palermo on Onion Architecture. The First of which is available from the link shown here. In creating N-Tier applications in C# Part 2,

What's Next

we'll cover how to test a domain-centric application as we continue building out our Pluralsight app including unit integration and UI level tests, then we'll look at some best practices for integrating entity framework code first persistence into the application. Finally, we'll add a second front end to our Pluralsight book application and show how we can reuse our well-tested and dependency free domain logic between multiple front ends. Obviously, building N-Tier applications in C# is a big topic. Please suggest additional topics, you'd like to see included in this course series by adding comments to the course, using the bit. ly URL shown at the top here. My intent is to add additional part to the series for as long as there continuos to be demand for new topics related to N-Tier application development. You can also suggest new courses for Pluralsight at the link shown at the bottom. Thanks for listening. This is the end of the first part of creating N-Tier applications in C#. I hope you've enjoyed this course and that you're looking forward to part 2 which will continue to explore concepts related to N-Tier application design. My name is Steve Smith and you'll find me online at Ardalis. com and on twitter @ardalis.

Course author

Steve Smith

Steve Smith (@ardalis) is an entrepreneur and software developer with a passion for building quality software as effectively as possible. He provides mentoring and training workshops for teams with...

Course info

Level	Intermediate
Rating	★★★★⯪ (918)
My rating	★★★★★
Duration	2h 1m

Released	17 Jul 2012

Share course