Creating N-Tier Applications in C#, Part 2

by Steve Smith

Start Course

Bookmark Add to Channel Download Course Schedule Reminder

Table of contents Description Transcript Exercise files Discussion Learning Che

Testing a Domain-Centric N-Tier Application

Introduction

Hi, this is Steve Smith and in this module, we're going to continue the course series on Creating N-Tier Applications in C#. In Part 2, we're going to build on the application built in Part 1. In this first module of Part 2, we're going to look at Testing a Domain-CentricN-Tier Application. Let's get started. In Part 2 of this course, we're first going to look at how to test a Domain-CentricN-Tier Application. We'll consider a couple of different kinds of tests and some best practices for organizing our tests. Then, we'll look at a common source of confusion in this kind of application, persistence. Finally, we'll look at how we can take advantage of the separation of concerns this architectural approach uses to reuse business logic from our core domain objects in multiple front-end user interfaces. If you haven't already, you may want to watch Part 1 of this course found at the URL shown here. In this module, we'll start at the solution level with some thoughts on how to organize test projects. A lot of this is personal preference or maybe subject to your team's practices, but if you're starting from scratch, I have some advice you may find useful. Along the same lines, I have some tips for naming and organizing the tests themselves that I'll also share. Next, we look at how to deal with infrastructure since we will want some tests that exercise our application using its actual infrastructure such as the database. Finally, we'll briefly take a look at how we can automate the process of running our tests. Before we get started, let's review where we are so far.

Brief Review

Previously on Pluralsight. In part 1 of this course, we built a small web-based application using a couple of different approaches. The current Pluralsight Book app supports user registration and login, user profile creating and updating and adding and removing friends with some notifications whenever a new friend is added. You can download the latest version of this in the Exercise Files if you have the appropriate Pluralsight subscription. Let's take a quick tour of where we are. We have a fairly simple web-from solution. We've got a couple of class libraries configured for the core in the infrastructure and we've set this up in a domain-driven sort of setup so that website, references, both Infrastructure and Core and infrastructure references core, but core doesn't reference anything except a few Microsoft standard libraries. When we ran this application, we can login, view our profile and make changes to it. And we can add and remove friends. When we add a friend, it also will send an email which in this case we're faking by having it just do a debug. print which we can view with this debug view utility. That's it. All right now, our application doesn't have any tests which we're going to take care of in this module.

Test Organization

Now, let's talk about how we want to organize the tests for our application. The first level of organization to consider is based on what kinds of tests we might have for our application. These can include unit tests, integration tests, acceptance tests, performance tests, load tests and even others. The second level of organization that you'll likely want to use is to break your tests down into 2 categories, tests that are run all the time, which should be very fast and tests that are only run some of the time, which may be slower or may require additional resources. Let's look at each of these. The first kind of test, unit test, is frequently used to describe a lot of tests that aren't actually unit test. A true unit test will only execute the code that we are testing and typically, this is the code that we wrote. It doesn't access any infrastructure concerns. No database, file system, network, web services, system clock, et cetera. All these things should not be executed by your unit tests. As a result of following these conditions, unit tests are extremely fast. You should be able to run hundreds or thousands of them per second. Another kind of test is the integration tests. Many organizations incorrectly refer to integration tests which are connected to a database or other infrastructure as unit tests. Typically, integration test should include some infrastructure. They may only include some pieces of the final system or some of them may combine all of the pieces that the production system will use. Generally, integration tests should exercise higher level operations or scenarios of the application than individual unit tests which typically target a single method. Integration tests that use infrastructure tend to be much slower than unit tests. You may only be able to get 10 or a fewer of these tests per second depending on your application resources. Yet another kind of test is the acceptance test which is often an end-to-end test. These tests typically include the user interface and exercise actual user interface elements to drive the application. They are far slower and much more brittle than unit tests. Each one could take several seconds to run and frequently, they will require a custom tool that is tied to the UI technology choice. For instance, you would probably use a different set of tools to drive a web application, then a

WPF or Windows Forms application. Performance tests are used to verify an applications performance will be acceptable given its requirements. Individual tests may target specific methods on classes in the applications core or infrastructure projects or they may exercise the full stack through the user interface like an acceptance test does. Now, one or more performance test may be run by a simulated number of users concurrently in what is called a load test. These are the most resource-intensive kind of tests to run and frequently, an individual load test may require 5 or 10 minutes or more and sometimes multiple machines in order to execute.

Fast and Slow Tests

When we organize our tests, it's a good idea to divide them into tests which will run all the time which should be very fast and those will run some of the time or on a dedicated build or integration server. We should be able to run all of our unit tests and perhaps some or even all of our integration tests on a small project all of the time. By all the time, I mean every few minutes and perhaps even continuously if we're using a continuous testing tool. Our slower test, we should be able to run on-demand as well as on a scheduled or triggered basis. For instance, it's typically a good idea to run all integration tests and unit tests on a build server after every commit to source control. Frequently, because of the resource demands, acceptance, performance, and load tests may only be run nightly.

Demo: Test Organization

Now, let's take a look at how we can organize our test projects within our Visual Studios solution. Here you can see, I've updated our Pluralsight book solution to include some test projects. I recommend that you organize your test projects using solution folders which you can do by just saying Add, New Solution Folder within your solution. In this case, so far I've added a project for integration tests and a separate project for unit tests. Within each of these, I've created folders that describe which other project they're exercising. At the moment, I've only added tests from my core elements in unit tests and only for infrastructure in integration tests. If we drill down into the organization of these, you can see that I've adapted a naming convention where I have one test class per method and I use a particular type of naming that makes it so that the tests sound like a sentence when you read them. So in this case, if we drill down into this, you can see that we have a class here called FriendsService_CreateFriendShould CreateFriendAndSendNotification. This is describing exactly the scenario that this test is exercising. Likewise, in our integration test, if we drill down into our data, we see that the entity framework friend repository create should add a record. In this way, it will be very clear what each one of your tests should be doing if anyone is looking for a particular test. Now it may be that as the number of tests in your application grows, you'll want to split some of these folders into their own projects. At which point, you might have a PluralsightBook. Core. UnitTests and a PluralsightBook. Infrastructure. UnitTests. But for now, I think using folders and keeping the total number of projects down works fairly well. Now, what makes up the difference between unit test and integration test? Well, as we already saw, unit test should only be exercising our code and should have minimal dependencies and should also be very fast. So let's run these tests to see how fast they are. (Pause) In this case, we're using the built-in test runner for Visual Studio 2012 which is MS test and you can see the test results shown here. The first one represents our integration test which is actually going to be talking to the database in this case. It's actually performing an AddRecord and then it's cleaning up after itself when it's done. The other tests here are all running just unit tests and you can see that for the most part, they're much faster. This one is a little bit slower than I would like and we can spend some time analyzing why it's taking as long as it is. But you can see that all of our unit tests run in significantly less time than our AddRecord even with this one taking a little bit longer than it should. Let's also look at how our test projects are organized in terms of their relationship to other projects. If we look at the references for our unit test, you can see that it's only referencing the core project within our solution. We're also taking advantage of Microsoft's unit testing framework and Telerik JustMock tool to do some mocking which I'll talk about in just a moment. In our integration test, you can see that we're using entity framework with system data entity as well as our infrastructure and core projects as well as Microsoft's test framework. In our integration test at the moment, we don't have any need for a mock framework because we're going to be exercising our actual infrastructure code instead of mocking it out and replacing it with a fake implementation. Let's look at one of these in particular. In this class, you can see that we are exercising the FriendsService which does a create friend operation. When we create a friend, we want that to actually create a friend in the database and send a notification. So it's doing a little bit of business logic. If we look at the system under test in this case, we'll find that under Core, Services, FriendsService and it's a very simple operation. All we're doing is calling friendRepository Create and notificationService SendNotification. Now, the beautiful thing about this code is that it does not have any hard dependencies. This friendRepository is being passed in using dependency injection or the strategy design pattern here through the constructor. Likewise, the notificationService is being passed in similarly. The nice thing about this particular type of design is that when we go to test it, we don't have to test that this is actually adding a friend to a database or that this is actually sending an email. All we need to test is that these calls are occurring and that they're occurring in the order that we expect if we care about the sequence in which these take place. For instance, if we go later on and add some error handling to this, we would want to make sure that we didn't send the notification if we fail to create the friend. Looking at our tests, we are using mocks to fake out those dependencies. So when we instantiate our new friend service, we're going to pass in a mockFriendRepository and a mockNotificationService. There are a number of different mocking frameworks available including Rhino Mocks, MOQ and in this case, we're using Telerik. JustMock. Telerik. JustMock has a free component which is what we're using but can also be purchased at a commercial level which unlocks some additional functionality. The implementation of this particular mock framework is similar to others. We call some kind of a method to create a mock of a particular interface and so you can see in this case we're mocking the IFriendRepository interface and the INotificationService interface. We exercise our code here by calling the method under test and then our mock framework allows us to assert that certain things were called. In this case, we're going to assert that mock friend repository create was called. It had these particular arguments, in this case we're saying they could be anything as long as they are the correct type. But then, we're specifying that it occurs exactly once. Likewise for the notification methods, we're specifying these particular arguments by type and again saying that it occurs exactly once. If we wanted to make this fail to prove that our test is in fact working, there are a couple of things we could do. First, we could change how often it's expected to occur. For instance, we could say that we expect it to occur never. And now if we ran these tests again, since this will in fact get exercise, we should see that that test will fail. And in fact, it does. All these mock libraries are doing is saving us some work. We could have written our own implementation of IFriendRepository and INotificationService and that implementation could have had a boolean that said whether or not this particular methods were called and we could have then asserted whether or not that flag had been toggled on our fake class. By using the mock framework, we're able to avoid having to create all these classes ourselves and instead, we've reduced it down to 1 line of code per class to set it up and another line of code per class or perhaps per method to verify that the behavior we expected did in fact occur. It's important to note that it's a very different type of testing when you do testing with mocks. Generally, what we're testing here is the behavior of the system and not the state. You notice that in this case, we don't care about the state of any other values that we're passing in. We're not asserting that that state is represented anywhere. There's no assertion here that there is in fact a friend in the database now. All we're asserting is that this service, the FriendsService call and this particular method, AddFriend has the correct behavior in how it interacts with the other collaborating objects in our system. Now, let's take a look at a different type of test, in this case, an integration test. If we look at our entity framework repository test, you can see that there's going to be a little bit more setup and there's not going to be any sort of mock behavior. Everything we do here is going to be real. So first, we're going to create our context. This is our actual entity framework data context that we're using. We're going to extract a test user by simply pulling out the first member from our aspnet membership. We're going to take our friendRepository, this is what we actually want to test. We're going to create a test email which in this case we're going to fake with a guid so it's not an actual email but we need something unique and then we're going to create a new friend using the test user that we grabbed UserId and this new testEmail which is actually just a guid. Finally, we're going to check whether or not this friend now exists by using our context directly. Notice that we're not going back to the friendRepository. We're going to circumvent that and talk directly to our database through this entity framework class and we're going to verify that there is in fact a friend now that has the user ID that we specified and the email address that we passed in. And we simply assert that it's true that that friend does exist. Once we're done, we're going to do some cleanup. We're going to delete the things that we've added and we'll save our changes to commit everything. When we want to run our tests, we should get to know a few different features of our tool. In the case of Visual Studios built-in test runner, there are some keyboard shortcuts you can use so you can just do control R and then A to run your tests like this. (Pause) And there they've run. Note that it's not control R control A. If you do that, it may give you a different behavior like you're maybe debugging into your tests. Another thing that you can do in Visual Studio 2012 is you can actually set up your tests to run every time you build your project. (Pause) If you have multiple monitors, you can take your Test Explorer window, pull it off into another monitor, let's pretend this is on a separate monitor, and then as you're working, if you're making changes to your code, adding things, you go to build. You'll be able to see your test run in realtime every time you build operation. This can give you very rapid feedback if you're suddenly doing something that causes a test to break or if you're doing test first development and you are trying to implement something that will make a test past, you'll immediately see as soon as you build whether or not you've succeeded. Let's look at how we can automate our test process so that when we are doing check ins and check outs, we're able to run these tests very rapidly and we also might be able to use the same scripts that we use for this automation with some kind of a continuous or build server.

Test Automation

Now, let's talk a little bit about test automation. We saw that we can have our test automatically run with each save using the latest version of Visual Studio 2012 as well as certain third party tools. However, we can also automate a lot of the stuff using batch files and MSBuild. To do this, for each of your solutions, create a custom MSBuild project that supports a few common tasks. It should be able to build the solution, run some or all of the tests and also perform a release build of the solution so you have deployable release DLLs. It's a bad idea to deploy debug DLLs. Next, create a few batch files that perform common tasks. One, build. bat is just a utility that calls MSBuild for us. However, ClickToBuild. bat uses this utility and specifies a particular target that runs everything and produces release DLLs. We should run this before every check in to the shared source repository. If you're using a non-decentralized version control system like TFS or subversion, you should run this before every check in. If you're using get or mercurial or something similar, you should do this before you push your changes back to remote branch that everyone is sharing. You may also create another batch file like ClickToBuildFast that only builds things in debug mode and just runs the fast unit tests. When you're working in a team environment with shared source control, you want to perform what I call the check in dance before you commit your changes to the shared repository. This has a number of steps that you should always follow as sort of a quick checklist. First, make sure that the remote build is good. You should be using some kind of a build server or continuous integration server that easily lets you see the status of your remote build. Next, make sure you're working with the latest version of the shared source repository code and merge your changes if necessary. Now, build everything and run all tests using a standard script that everybody shares and ideally this should be the same script that the build server will utilize to run your build and exercise your tests. In the case of what we're going to see in a moment, this would be the ClickToBuild. bat script. Now, you can check in or commit to the shared repository and then you want to make sure that the build still is good after your check in if despite all these efforts we manage to break it. You can read more about this at the following URL where I blogged about it. Now remember, unit tests prevent small thermal exhaust ports in your shipping code.

Demo and Summary

Let's take a look at how these test automation scripts work and then we'll be done with this module. Here we see the root folder that I'm using for this project. This is where our solution file is located that we've been looking at. I've added a number of different scripts here including the build. bat, build. proj, ClickToBuild and also a ClickToBuildFast batch file. Let's run ClickToBuild to see what this does. If we run this, it's going to run a build of our solution first in debug mode. This will let us easily see any problems that are occurring with our build. Next, it runs the unit test which you can see just passed and then the integration test which also just passed. Finally, it runs everything in ReleaseBuild and you can see here it says Pluralsight Release Build Complete and the entire thing gives us a build succeeded with no warnings or errors, took about 16 seconds. Our ClickToBuildFast basically does the same thing but is only going to run the unit tests. So when we run this one, it's going to do our debug build, run our unit tests and it's done in just under 6 seconds. Now, let's look at build. proj which is our MSBuild file. You can see that it's only about 30 lines long and has a couple of different features. We have a complete course on MSBuild available on Pluralsight where you can learn more about creating these files. For now, I just want to walk through a couple of the high points. When you get the source code for this course if you have the correct subscription, you'll be able to see this yourself and modify it to suit your needs. I use a property for the ProjectName here called PluralsightBook and this property is then used elsewhere inside these other tasks so that by changing just this ProjectName here, I can reuse this file for other different projects that I might be working at. I'm also specifying the path to MSTest. This may vary depending on your environment. Next, I have a number of targets. Each target has a name. This one is the DebugBuild. It provides a message letting me know that it's doing the build and then it does a Clean and then a Build using the debug configuration. BuildAndTest is the next one to run and it will run the unit tests. You can see that it depends on the DebugBuild. So if I execute this target, MSBuild will automatically run the DebugBuild target first for me. If you've ever used to make files, this is very similar. Here's my command to actually execute this test and you can see again I'm making use of that ProjectName variable. Next, I have an integration test target which will simply run the integration test using the same type of syntax. And finally, my ReleaseBuild which you can see depends on IntegrationTests. This will do a Clean again and then it will do a release configuration build and that's it. That's our MSBuild script. Now, let's look at these other batch files. First, build. bat is simply 1 line that knows where my MSBuild. exe file is and calls into my build. proj file with whatever target I'm specifying from the calling script. That calling script might be ClickToBuild which is another 1 line and it's simply going to call build. bat, passing in the ReleaseBuild as the target and then specifying it we want a pause after we're done so that the screen doesn't just disappear after the built completes. And ClickToBuildFast of course is very similar. In this case, we're only specifying BuildAndTest instead of ReleaseBuild. So to summarize, in this module we've learned about how we can use tests in our N-tier application. We've learned that different tests provide different kinds of quality assurance and with different costs in terms of mostly the time and effort involved in running the tests as well as maintaining them. Using a variety of different kinds of test provides a sort of defense in depth against defects in our applications. We can organize our tests and one of the key areas that we may want to use to organize them is going to be into fast and slow test so that we can run fast tests all the time and slow tests only as often as we can and perhaps only in on our build server. Finally, we want to automate these processes as much as possible so that we're able to run these tests constantly and also run theme every time we do a check in so that we have a much less chance of breaking the build for our teammates. This course references a number of other Pluralsight Courses where you could learn more including my SOLID Principles of Object Oriented Design course and a Test-First Development course and Design Patterns Library, our Continuous Integration course and finally an Introduction to MSBuild. Thank you very much. I hope you're enjoying this course on Building N-Tier Applications with C#. My name is Steve Smith and I'll see you again in the next module.

Persistence Best Practices

Introduction

Steve Smith: Hi. This is Steve Smith. And in this module, we're going to continue the course series on creating N-Tier applications in CSharp. In this module, we're going to take a look at persistence best practices in a domain-driven design. If you've been following along, you've just seen that we covered how to organize our tests in a domain-centric N-Tier application. In this module, we're going to look at persistence best practices, and then we'll wrap up this part of the course by looking at how to reuse some of our CoreLogic with multiple frontend applications. If you haven't already, you may want to watch part one of this course at the URL shown here. We'll be starting this module by considering object life cycles and where persistence fits into these life cycles. There are several patterns one can use here, each of which deals with responsibility of persistence differently. We'll discuss the responsibility of persistence and where it should live in our application and our solution, which classes should worry about persistence, and what does it mean for an object to be persistence ignorant? Next we'll take a look at object relational mappers and repositories in a bit more in depth than we have thus far. Finally, we'll consider some useful tips and tricks that relate to persistence in domain-driven design style applications. We'll see you again soon.

Object Life Cycles

All objects have a life cycle within an application. They are created, they reside in memory for some time, and ultimately, they are disposed of. Simple transient objects go through this process frequently within an application, and there is no need to worry further about them. However, most applications need to maintain and manipulate some kind of persistent state. The model objects we work with in our applications frequently exist prior to our application's start and must continue to exist after our application shuts down. They may be used by multiple users through our application or potentially through multiple applications entirely. Their life cycle frequently includes explicit phases in which the object is restored from or persisted to some kind of persistence. Let's look at this another way. All objects begin through construction, may be modified, and ultimately, are disposed of. We may need to pass around representations of these model objects between applications or layers in an application using some kind of service. We may need to store representation of an object graph to the file system. And commonly, we'll use some kind of database to manage the storage of many different kinds of objects for our application. Persistent objects are more challenging to work with in our applications than simple transient objects. Wherever possible, we should strive to keep our objects persistence ignorant, meaning that none of the details of how we might choose to persist the object leak into the object itself. Another challenge is ensuring our objects are in a consistent state within our application.

Consistency

Let's talk a little bit about consistency. In object-oriented programming, it's important that we keep our objects in a consistent state. Attempt to use objects that are in an inconsistent state may result in unnecessary exceptions or, worse, undetected logic bugs. Especially during construction, objects should not be returned in an inconsistent state. An object that is designed such that it must be constructed and then certain initialization methods must be called before it is valid for use is just asking for bugs. A common design pattern for avoiding this scenario is the use of a factory method that returns the fully-constructed and consistent object. You can learn more about the factory method design pattern in the Design Pattern Library available on Pluralsight. It's equally important that persistence patterns not result in inconsistent state for our objects. For instance, if we load an object from the database then change one of its dependent objects and finally save the original object to the database, should it save the changes to the dependent object? We need to consider these kinds of scenarios in our persistence strategy. And our decisions will affect the kind of design we end up with. Remember that as you design the persistence strategy your application will use, you're designing an API. Your API may be used only by yourself or your team, or it might be exposed to other developers or even the public through a public API. In any case, you want your API to hide the internal complexities of the implementation details that it uses, but to expose a simple, intuitive interface. Ideally, this interface should gently guide its users into the pit of success by making the wrong things hard and the right things easy. Don't design your interface to look like this.

Persistence Responsibility

Now let's consider persistence responsibility. In any application that requires persistence, there are generally three ways that we can house the logic that's responsible for persisting an object to or from the database. In the first case, the object in question can be responsible. This is an example of a pattern known as active record. In the second case, some other object is responsible for persisting the object in question. Ideally, the object that's being persisted is PI or persistent ignorant, and is also known in the. NET world as a plain old CLR object or POCO. Finally, there's the everything-else case. Frequently, as a matter of expedience, some persistence might happen wherever we happen to need it without any proper abstraction or consideration of separation of concerns. Let's look at each of these.

Demo: Active Record

In order to examine each of these different scenarios, I've created a new folder in my PluralsightBook solution inside of my integration test project. I've called this folder Spikes. And you can see here that I have included an active record, a code first, and an inline folder for each of the scenarios we're going to look at. Let's start with active record. I've created some tests that will use the active record pattern in order to have a friend domain class automatically persist itself. In this case, I'm calling that class an ActiveRecordFriend. And we'll look at its API in just a moment. In terms of the tests that I've chosen to create, I have shown that I can create a new friend, I can create an existing friend, which is, essentially, loading it from the database. I can also save changes to a friend, and I have a bit of logic that's in the friend class that is going to return to me the EmailDomain. And so that test looks like this where I can verify that when I create a new friend with this particular email address, I expect that the EmailDomain is going to yield back foo. com. Now let's look at whether these tests run. So you can see all of these tests pass. Now let's look at the API that we're exposing. Here's our friend class implemented using the ActiveRecord pattern. In the constructor, we require that we pass in an ID. Any time we're going to new up one of these, we expect that we're going to actually be pulling it from persistence. Thus, we'll pass in the ID of the friend that we want to load. And within the constructor, we'll use the data context to explicitly pull in that friend. We'll set that entity to an internal field. In this case, the FriendEntity field. And set some of the properties as well. We can set the properties any time we need to make a change between the entity and the properties themselves. And then use our auto properties like you can see here. Or in some implementations, we can simply pass through so that any time somebody asks for the ID property, we simply return back the FriendEntity. ID property, and similarly for each other property. Now, when it comes time to save our class, we're going to expose a save method that the end user can call any time they want. And whenever they do, we're going to use the context, once again, to attach the entity, and go ahead and make any updates that are necessary. In this case, we know the only thing that's changed is the email address. Again, if we were just wrapping the entity, this step wouldn't be necessary. And then we'll save the changes. Now, since we can only create these objects with an ID, that means that we can't construct new instances of the friend class directly. If we want to create a new friend, we need to call this static factory method here called CreateNew, pass in the required fields, in this case, the email address and the Guid of the user ID that this friend is associated with. And then, once more, we'll use the data context to do the necessary work of creating that friend for us, returning back a new instance. Finally, I have exposed here a little bit of business logic that is able to show how to get just the EmailDomain of the email address of the friend. Why do we care about this? That's not really important, but it gives us one more thing to test. Now let's look back at how our tests are organized for the ActiveRecord pattern. You can see that in each of these cases, we're having to talk directly to the database in order to perform these tests. Now, for the most part, these tests are all testing the actual integration of the system with the database and the persistence effects of our implementation of ActiveRecord, so these are probably fine. We want to be able to test that when we actually create one of these friends that it does, in fact, get stored in the database. We want to be able to test that we can create existing friends by passing in their ID, and this works as well. We want to be able to test that we can save the changes and then pull those changes back out and verify that they were persisted correctly, as this test does. However, when it comes time to test the business logic of our object, we are now complicating it, unnecessarily, with these persistence concerns. There's no way to actually simply create a friend object at this point without having it talk to the database. And so when it comes time to test that EmailDomain property, the only way that we're able to do that is by going through all the effort of creating a new friend and hitting the database. We can't simply test the logic that we are concerned with directly. So what would this look like if we were to change this and make it so that we could go ahead and test just the logic we want? Well, we could take what's called a persistence-ignorant friend here and inherit from our friend class that's in the core model that we saw earlier, add that property that we're concerned with straight to it, and then test that using just the logic necessary to test that particular method. You can see the difference here between the logic that's required to set up a test that uses the ActiveRecord pattern and the logic that's required to set up a test that does not. And you can see that it's much simpler without the unnecessary persistent concerns intermingled with the business logic of our domain object.

Demo: Repository

Now, we've seen how it's easier to test the business logic in a domain object that is not concerned with persistence. But let's talk about how we would actually persist an object like this. In the last module, we looked at how to create integration tests like this one that use the repository to add records to the database. You can see in this method that we are going to use a data context to get a test user ID that already exists in our database. Then we'll simply create a new friend through the repository here. And finally, we'll go back to the data context to verify that the friend that we just created exists. And then we'll delete the object and save the delete, cleaning up after ourselves. Now let's consider how we could do this initialization and cleanup in a little more friendly fashion. Here we have a different class that's using a transaction to do this. We're going to have a transaction scope that we create. And in our test initialize, we're going to set the scope and say that we allow read uncommitted so that were able to investigate the changes that we're making as we make them. In our test cleanup, we're going to go ahead and dispose of that scope, which will roll back any transaction that we've started. Now our AddRecord method still allows us to use a data context to pull out the test user ID, and does the same kind of work to actually create the friend. And then we're also still going to use a data context to verify that the create succeeded. But we no longer need to clean up after ourselves. The transaction will take care of that for us.

Demo: Persisting in UI

Now let's take a look at what happens when we don't have any particular organization to our data access logic. Imagine that this is a user interface and in the user interface we have a button. Whether this is on a Windows form or a

Web form, doesn't really matter, but we're in the user interface layer at this point. Here we're going to go and read some elements from user interface. Maybe we'll do some validation. And when it's time to save something to the database, we'll just new up our data context right here, add the thing that we care about, and save the changes, and then do some more user interface level stuff. The problem with this is that it's very difficult to test, and it comingles a lot of concerns. It tends not to follow the don't repeat yourself principal, and it certainly isn't following the single responsibility principle because this particular method, let alone the class, is doing way too many things. It's very important when we design the architecture of our system, if we want it to be flexible and modular and easily maintained, then we follow the principle of separation of concerns. Let's talk about that a little bit more as we get back into our discussion.

Separation of Concerns

Another way to think about separation of concerns is that we don't want to let our plumbing code pollute our software. Take a look at this refrigerator. What do you see that might be out of place? A key principle of software development and architecture is this notion of separation of concerns. The general idea is that one should avoid co-locating different concerns within the design or code. At an architectural level, separation of concerns is the key component of building layered applications. When we're looking at how to do persistence in our application, we want to make sure our persistence concerns are wrapped up in their own section of our code, in our own section of the architecture. In a visual studio solution, that means we should try to limit our persistence-related infrastructure into a single project or a set of projects that don't leak out into our user interface or our core domain model.

Entity Framework

Next let's look at the role of object relational mappers in our applications and how these relate to our core domain model. First, let's look at Microsoft's ORM tool and Entity Framework. Julie Lerman actually has a number of excellent courses on Pluralsight that cover Entity Framework than I can do justice to here. So I encourage you to take a look at some of her courses to learn more. What I'd like to do here is demonstrate a few different ways you can implement EF into your application and tests, and how you can swap out one flavor of any framework with another or replace it with another ORM entirely. There are two ways to use Entity Framework currently. The first is with an Entity data model, which is what we've been doing so far in our application for the PluralsightBook. And the more common approach when using a domain-driven design is using code first. You may use Entity Framework without code first using an Entity data model like this one. This is the Entity data model we're using in our PluralsightBook application. When you go this route, you can use these entities directly in your applications code, or preferably, you can map them to your domain objects. Assuming that you're using a repository for your data access, it's best if you design your repository so that it knows about your entities, but its clients do not. Thus, you should map between your entities and your domain objects inside of your repositories. Your repositories should return domain objects that your application works with. Now, you can do this mapping by hand, perhaps taking advantage of LINQ's select method for help, or you can use a tool like AutoMapper. Let's see how we did this in our application so far, and we'll investigate AutoMapper in a few minutes. In our application so far, we've created a IFriendRepository interface that has a few specific methods that we needed. And we implemented it using Entity Framework as shown here. Now, this code still could use some refactoring, of course, but I want to call your attention to the ListFriendsOfUser method, which is the only one we currently have that's returning anything from our repository. This friend here is actually a model class, so it's coming from Pluralsight. Core over here, this model friend. Note that when we're querying these friends here, these are actually friends in our data layer. So these are aspnetdbEntities friends, not our model friends. In order to map from one to the other, we're using LINQ's select statement here, and we're instantiating a new friend and setting its ID to the one that's in our entity and its email address also to the one within our entity. This kind of mapping code can get very tedious if you have a large data model and many different domain objects and you're having to implement this for every different repository that returns a model object. We'll see in a moment how we can make this a little bit easier using a special tool called AutoMapper.

Mapping with AutoMapper

AutoMapper is an object-to-object mapper. You can download it from AutoMapper. org or from its location on GitHub. You can also easily get it using NuGit with the install package AutoMapper option. Once you have it installed, to get started, you simply need to set up the initial mappings for each of your objects. This will look like this. Now, in our case, we're going to be performing this mapping inside of our repositories so that we're able to return back a model friend from an Entity friend. This code will look something like this. Let's see it in action in our test. Inside my integration test project, I've created another Spike folder or mapping, as you can see here. Here I've got a repository mapping test section. I've set up the test so that initially we create a map using AutoMapper. AutoMapper exposes a mapper object with a static method called CreateMap, and you can find it once you add the using statement for AutoMap. Now, within my code, I'm simply going to call this Friends. And I expect that once I call a create method under repository and then I list the friends of the expected user that this result should be of type friend. So you see it's a Pluralsight. Core. Model. Friend and not of the Entity type. And so I'm just asserting that my result is an instance of the expected type. Now, to do this, I'm using a custom mapping friend repository that I've created right here. And you can see that this is a separate implementation of the IFriendRepository. The create method is pretty much exactly what it was in the other one. The delete I'm not using, so we've left it unimplemented. And now here is the code that we're replacing. Instead of having to call. Select and specify all the mapping by hand, we are now able to just call Mapper. Map. And regardless of how many different properties there are or what different kinds of custom mapping logic there might be between the entity and the core domain object, AutoMapper takes care of all this for us. So this can potentially save us a significant amount of repetitive work, and we can move all of that logic for how we do mapping into our CreateMap method, which will exist somewhere at startup of our application, and it will all be in one place. So it typically will have some kind of an AutoMapper bootstrap class that will handle all of the mapping creation. And we'll be able to test that in isolation and keep that class with a single responsibility.

EF Code First

Entity Framework also now supports code first development. With code first, you don't need a point EF and an existing database schema and have its Entity model be based on your database tables. Thus, you don't need to map from EF entities to your domain objects. Rather, you can create any domain model you'd like using persistence-ignorant POCOs and use code to tell Entity Framework how to retrieve and store your data to and from the database. In my current application, I'm using Entity Framework 5, which supports mapping classes that can be used to help you organize your database initialization code. You may also want to look at how you can automatically generate your code first classes from a database using Entity Framework power tools. Yes, I realize that this is no longer code first, but the end result is that you'll have all your mapping files generated for you, so you can tweak them to suit your needs. You can download the tools here. After you run this tool, you can tweak the results and get something like this. If you run Entity Framework until it's reverse engineered the code first files for you and pointed at your data source, it'll generate some code like you see here. So I've run this against my PluralsightBook application, and then I tweaked it by deleting some of the tables and things that I need. So in here, it's created our models. So we have aspnet applications, we have membership, we have friend, and it's also created the mappings that it needs to convert these to and from the database. So if we look at our friend map, you'll see here that we've specified that it has this primary key, it has this property that's required, and it has some mappings to other things here, as well as the relationship from friend to membership. It also will create your context, which will include your DbSets for everything you need. And in your on-model creating, this is where you specify those map files. So you'll need to specify one for each of the classes that you want to map to and from the database.

nHibernate

Another popular relational mapper is NHibernate. NHibernate is based on Hibernate, which exists in the Java world. And you can use something called FluentNHibernate to achieve a very similar effect to what we get with Entity Framework code first. With FluentNHibernate, you create a session factory in code using something like this. Now, it's very important that if you're creating a session factory like this, particularly in a Web application, that you do not create session factories on every single request or every single database request. Session factories are relatively expensive to create, so you want me to make sure you only do this once when it's needed, and then use that same session factory to create sessions. You use individual sessions and transactions to do your actual database interactivity. You can learn more about NHibernate on Pluralsight from the NHibernate Fundamentals Course by

James Kovacs. Let's see how we can wire it into our application. To get NHibernate working in your application, first you need to set up a database configuration class with your create session factory where you're going to return the fluently configured application. Here we'll specify which connection string we want to use, as well as our mappings. And for the mappings, we're going to add all of the mappings that are found within the assembly that contains this class. So here are my mappings, which you'll find through reflection. I have one for friend mapping, which you can see looks rather similar to the Entity Framework map. And I have one for user mapping. Now, NHibernate expects that certain things in your model will be marked as virtual. So I've gone, and I've added virtual to those properties that didn't already have it inside of my model classes. I also add in a new AddFriend method to make it easier for me to add a friend to a user within my model. I implemented the NHibernate friend repository using the same interface, IFriendRepository, that we had previously used for Entity Framework. And I pass in my session factory through the constructor so that we won't have to recreate that any time that we use this repository. Here you can see the code to create, delete, and list the friends of a user using this NHibernate syntax. I also implemented the IQueryUsersByEmail interface that we created in the first part of this course so that it works within Hibernate as well. Now I wrote some tests in my integration tests. You'll find here a Spike for NHibernate called NHibernate Tests. And in this, I'm able to say that I can list all my friends from the database using NHibernate. I can list all the users. And I can create and delete a friend with my NHibernate repository. You should be able to run these tests to verify that NHibernate is working correctly for you. Now, in my Web application, I've wired up the NHibernate versions of these repositories in the appropriate pages. So where I add a friend, previously I used a friend service with a EFFriendRepository and a EFQueryUsers. I've replaced that now with an NHFriendRepository and an NH query users by email. Both of these are using my Global Session Factory. Likewise, when I list my friends, I replace the service calls here as well so that these are using the Entity Framework versions of my repository and my QueryUsersByEmail. I've had to do that in quite a few places. You'll see it's down here as well. We'll look at how we can avoid having to do all this manual construction later on in this module. Now let's look at the Global ASAX. Now in the Global ASAX, I've specified a single static read-only session factory, which I've set up to use the lazy of T type here. What this will do is it will ensure that when this is first requested, it will call this constructor to return back the configured session factory. However, it will only call this once globally for this application, so I won't end up creating a session factory every time someone accesses the session factory property in my Global ASAX. This is a good pattern to follow, something like this, for when you are using an NHibernate session factory in a Web application since if you do create a session factory on every request, it will dramatically slow down your Web application.

Repository Design

Now let's talk a bit more about repositories. To learn more, please check out Scott Allen's description of the pattern in the Design Patterns Library shown here. Also, Julie Lerman discusses repositories and units of work in her Entity Framework in the Enterprise Course, which I recommend if you're interested in using EF and repositories. When it comes to designing repositories for your application, there is no one-size-fits-all solution. As you get started, you can simply create whatever interface has the exact methods you need for reading and writing to persistence, implement that interface, and you're done. This is what you've seen so far in our PluralsightBook application. However, as your application grows, you'll want to have some consistency in its API. And that extends to your repository implementation as well. Generally, you'll find that most of your objects that require persistence need some subset of basic create, read, update, and delete or CRUD methods, and so you'll usually be able to reduce the repetition in your code by abstracting these methods into a single generic implementation. However, you'll probably still want custom tailored methods for many of your domain objects, and you'll want to consider some different possibilities for how you organize your repository classes. Probably the most common approach, especially since it can benefit from the generic approach I just described, is to organize repositories by domain object type. You might create a separate object for every type, or you might determine that certain things should always be updated together. For example, an order in an order detail or in our case, a user and his friends, in which case you would organize repositories per aggregate route. There can also be performance and architectural benefits to separating your repositories based on whether they read or write to the database. That is whether they query or issue commands for writes. We can discuss this and the related command query responsibility separation topics later. It may also be useful to have separate data contexts and to organize your repositories by which contexts they are referencing. Julie Lerman shows an example of this approach in her EF in the Enterprise Course that I just mentioned. For now, let's assume we're simply going with the most common route and organizing our repositories by domain object type. We still have some questions to consider. Should we implement a generic repository? Should our repositories expose iQueryable results? Should we implement unit of work or transactions? Where? And how should we deal with sharing our ORM's data context? Let's look at some examples of these and see if we can answer these questions. As we consider these questions, let's look at some code. But remember, depending on what kind of application you're writing, the implementation that you choose to use for your repositories may vary. The needs of a Windows application like a WPF or WinForms app will often be very different from the needs of a Web-based application, so keep that in mind. First, let's consider generic repositories. Personally, I'm a fan of generic repositories because they allow me to centralize the code that accesses the database through the underlying ORM technology into a single class. I can define a standard interface called IRepository of T that defines all of my interactions with the underlying ORM by these types of signatures. This is an example of implementing CRUD functionality. I can get a single item, I can get all items, I can issue a query, and this will actually issue the query to the data source so I don't have to pull back all the results and query them later in memory using LINQ or something similar. And then I can do add, remove, or update. And finally, I can save. Now, since I'm exposing save at the repository level, that means that each repository is essentially implementing its own unit of work pattern. You can learn more about the unit of work pattern in the Design Pattern's Library. Now let's look at an implementation of this repository using Entity Framework. Here you can see I have Repository of T, which implements IRepository T. And it's doing very simple pass-through to an underlying context, which gets passed in via its constructor. When we do an update, we simply set the entry state to modified. This will inform the underlying context that when this item is saved, that it needs to make any changes back to the data source. And when I call save, I simply am calling back to underlying context. SaveChanges. Now let's see how we would use this generic repository by looking at a test. In this test, I'm adding a friend using the generic repository. I create a context, and then I initialize my data source. Then I simply call into the context to get the friends count. This is for test purposes only. It's not what I would actually use. When I'm using the repository, I wouldn't use the context directly. Likewise, I get a test ID from the context so that I have some good data to work with. Now, in this part, I'm actually exercising my system. And so here I'm going to instantiate my repository, add a friend, and save. And then finally, I'm going to assert that when I talk to the context directly, the total number of friends has increased by one. Notice that when I created this repository, I simply created a new repository of friend. I did not have to separately go and create an instance of friend repository. I didn't have to type it. I didn't have to define it. There is no friend repository class anywhere in my source code. So I'm able to use this generic repository for any of my domain objects in my system. If you want to avoid confusion and ensure that you don't end up creating repositories of classes that shouldn't be used, you can add a marker interface to your domain objects such as IEntity so that then these particular interfaces will only be these by your generic using a constraint. For instance, my constrain here is only that T is a class, but I could further have had an IEntity interface and constrained T to be a member of IEntity. Now let's look at whether we should return iQueryable. Let's go back to our tests. And I've created a test method here called WhyNotExposeIQueryable. Guess what I recommend. In this method, I'm doing some basic setup here, and then I'm going to call into my repository and look for all of the friends whose email address contains foo. I am expecting there to be exactly two of those, so I'm asserting that. And then I want to see how many of them are Microsoft employees. So I've created an extension method called IsMicrosoftEmployee. I've defined that here. And you can see that all it does is check to see if the email address contains Microsoft. com. We jump back up and look at our sample data, and you'll see that the third user includes that. So I am expecting when I execute this and then iterate through the results or get a count, that I'm going to have exactly one of those. Now, I can do that same check if I'm using an IEnumerable repository. So here is my nonQueryableRepo. It's using my generic repository of T. And I'm running the exact same query. And I'm also asserting that I have one such friend. However, when we run this test, we see that it fails because Entity Framework takes that extension method and tries to run it on the database. And so we get this exception that says LINQ to Entities does not recognize the method IsMicrosoftEmployee. And it cannot be translated into a store expression. So the problem here, in my opinion, is that when you expose iQueryable from your repository, you're exposing low level persistence concerns to your upper levels of your code. In this case, I, as a developer, should not have to worry about whether or not I can call IsMicrosoftEmployee here or here. In one case it works fine. In the other case it does not. Now, I do need to worry about whether or not I'm going to be fetching many, many rows of data and then filtering on them in memory versus doing that filtering on the database. But if I know that all of my repositories expose IEnumerable, then I can be certain that whatever I get back from them is disconnected from the data source. And if I need to have them do additional filtering, I can be certain that I passed that into them correctly. In my case, what I like to do if I need a particular custom filter is either have a special method that takes that, as we saw here in my repository, I support a query method that takes in an expression, and this will run this filter on the database. However, this still returns an IEnumerable at the end of the day. So once I pass around this result, if developers want to continue to filter it by using LINQ expression and. ware (assumed spelling), they won't run into the issue that we just saw. Now let's talk about one more question, which is what do we do with objects that have different contexts? It's not unusual if you have different objects or different methods working with your domain objects that you might retrieve a domain object from one repository instance or one data context and then want to try and save it with another data context. So for instance, here if I change this line so that now when I initialize my data and I get my test user ID here, the next thing that I'm going to do is create a friend and add it, and then save it. So this is saving a new instance of a friend. Now if I get another separate repository and try and remove that friend, this will fail because the friend that I'm removing, this instance, is the exact same friend that I've just saved here to a different repository. This repository is using the context that I have specified out here. This new repository, if I don't specify a context to pass into the constructor, it's going to go ahead and create a new context. Now, what will happen when you have two separate contexts is that you'll get an exception from Entity Framework. So again, if we run this test, we see that it, too, fails. And the problem here is that it now tells us that the object cannot be deleted because it was not found in the ObjectStateManager. Now, to solve this, you need to make sure that you're using the same context both to retrieve and to store the object in question. You'll get this exact same behavior whether you're trying to do an update or any other sort of operation with an object that you pulled out of a different context. Now, having made this change, we can run the test. And now you see it passes. And this is because we've ensured that this context is the one being used by both of the repositories that are doing the adding and the removing in this case. Now, before we wrap up, let's talk about how we can make sure that we're using the correct instance of a context in our repositories by using something called dependency injection and an IOC container like StructureMap. So we have one more sample here, and that's under IOC. And here what I've done is I've installed StructureMap by using NuGit and simply saying install packaged StructureMap. I'm initializing my class by configuring my object here. And what I've specified is that I want to -- any time I need a DbContext, I want to use a PluralsightBookContext, and I want it to be HybridHTTPOrThreadLocalScoped. What that means is that I'll only have one instance of this per HTTP request or per thread. And now I have this same method here where I'm going to delete the friend using a generic repository and IOC. You notice that I'm grabbing my context here using ObjectFactory TryGetInstance. This is a StructureMap method here. And now when I go and talk to my repository, I can tell it here, use that context. Now if I make another call to it, this could be from another method, this could be from another class, I can, once again, say ObjectFactory TryGetInstance of DbContext, and it will work. Now, this is not the code that I would actually write in my application. This is just for test purposes. In my application, what I would rather do is have an IRepository of T instance here in my configuration script for StructureMap, and it would automatically then know how to give me an IRepository. And so in whatever class needs to have this IRepository of T, it would simply have a constructor that would say I need an IRepository of T, and StructureMap would fill that class automatically. We'll look more at how we can use dependency injection and IOC containers in another module.

Summary

We're about out of time, so let's wrap this up. To summarize, in this module, we talked about object life cycles and where persistence fits into it. We talked a little bit about different places where we might separate the responsibility of persistence, whether that's in the object itself or in a separate object like a repository. We looked at a couple of different object relational mappers, specifically Entity Framework and NHibernate. And we talked about how to design our repositories, which is a large topic, and there's some additional resources you can find on Pluralsight to learn more about that. We went through a couple of different trips and tricks in this module. To recap, remember that you want to avoid creating multiple session factories within your application if you're using NHibernate. You should be careful if you're passing around IQueryables from your repositories. And you want to make sure that you have a good strategy for sharing your Entity Framework data context between different parts of your application or different repositories. I also want to point out that you should check out the CachedRepository pattern if you're interested in speeding up the data access speed of your use of the repository. You'll actually find the CachedRepository is also covered in the Design Patterns Library on Pluralsight under the Proxy pattern. These are some other courses that you may find useful if you found this one interesting. And that's it. Thanks. I hope you're enjoying this course.

Core Logic Reuse

Introduction

Hi! This is Steve Smith, and in this module, we're going to continue the course series on Creating N-Tier Applications in C# Part 2. This module focuses on Core Logic Reuse in a domain driven design. If you've been following along so far in this course, we've talked about Testing a Domain Centric N-Tier Application. We just looked at some Persistence Best Practices using things like repositories and object-relational mappers like Entity Framework and Hibernate. And now, in this module, we're going to talk about reusing the Core Logic in our domain driven design in Multiple Front-End Applications. If you haven't already, you may want to watch Part 1 of this course which you can find at the URL shown below or by doing a search on Pluralsight for N-Tier.

Overview

One of the core benefits of N-Tier Architectures is what has been called the holy grail of software, code reuse. When we encapsulate our core business logic into a library that can be tested, shared, and reused, we start to see some huge benefits when it comes to code reusability, and that's one of the main topics of this module. Our goal is for our software's components to be like Legos, and that we can rearrange the pieces within the system in order to get new value out of existing components. In these examples, you can see a variety of different Legos have been used to produce the same bird shape. Similarly, in our application, we can spot out different parts of our infrastructure or even core logic within the application in order to change how that application needs to behave in a given circumstance. This is only possible if we've designed our system to follow some rules that allow for loose coupling. In strongly typed languages like C#, an important technique for achieving this loose coupling is to follow the dependency inversion principle. Applying this principle results in dependencies on interfaces, rather than concrete implementations. In systems designed in this fashion, it's often useful to have a component whose responsibility is to resolve requests for a particular interface. Such components are known as IOC containers or perhaps DI or Dependency Inversion Containers and we'll look at how to use one to flexibly configure our front-end applications. The Dependency Inversion principle is described in the SOLID Principles of Object Oriented Design course also available on Pluralsight. If you haven't checked it out, I strongly recommend it. In part 1 of this course, we talked about the Onion Architecture which is a core component of domain-driven design. Now, for this section, we're going to talk about some of the outside layers where we have Applications and Services. In this module, we're going to add a new front-end application which will consist of a simple console application. Then we'll move on to create a set of web API services. And this will be built into an ASP. NET MVC application. Ultimately, we'll refactor our new application so that it utilizes these new services.

Inversion of Control

Let's continue from where we left off in the last module where we looked at how to swap out our data access technology. We wrote separate repository implementations for Entity Framework and in Hibernate, and wired up our web forms code to use the correct one. However, because of how we've written our code up to that point, the only way to do this was to new up the correct dependent objects every time we instantiated a service. Let's ask an expert on software development, Boromir, whether this is a good idea. Obviously, this is not ideal and violates the "don't repeat yourself" principle since we'll quickly see the same logic scattered throughout our code. Further, every class that is making this decision is now violating the "single responsibility" principle since in addition to its main purpose, it must now also decide which dependent objects it wants to work with. Writing code in this fashion, we end up with code like we saw in the last module that looks like this. This is a pretty simple service, and you can see that already just to instantiate this friendsService, we have to instantiate four other dependent classes. Now, what we'd rather have is code that looks something like this or maybe this. In each of these latter cases, we have a factory responsible for constructing our service. We can write our own factories for everything but that tends to be a lot of low-level plumbing code that doesn't add a lot of value. Using an IOC container, either directly or through yet another abstraction, like the IOC helper class shown here, lets us avoid having to write our own factories. IOC containers are basically just factories on steroids. If you remember that, it should make it easier for you to wrap your head around them if you've not used them before.

StructureMap

StructureMap is one of the many IOC Containers for. NET. It happens to be one of my favorites. It's open source, it's free, and it was written by a friend of mine Jeremy Miller. You can quickly install it using NuGet by simply calling install-package structuremap within your application. There's documentation available online though the look of the site is rather dated at this point. Don't let that stop you from enjoying it. And, of course, there's a course on Pluralsight covering Inversion of Control in general including a variety of IOC containers including StructureMap by John Sonmez. If you're interested in learning more about StructureMap or Inversion of Control, I recommend this course.

Demo: Adding StructureMap

Now, let's look at how we can use StructureMap to clean up some of that code that was a little bit nasty from the previous module. I've already installed StructureMap which you can do either from the NuGet Package Manager Console or by calling Manage NuGet Packages from the Visual Studio Solution Explorer like so. Here, you can see StructureMap is installed. Currently, it's version 2. 6. 4. 1. Now, let's walk through what we need to do in order to configure StructureMap. First, you need to call it when your application starts, which in an ASP. NET application means that you should call it from your application start method found in Global. asax. It's best not to put a whole lot of actual code inside this method, so I've moved all of the extra logic for StructureMap into its own configure method. Now, if we jump in to this method to have a look, let's see how StructureMap is configured. Again, this is not a course on StructureMap or IOC containers, so I'm going to go over this fairly quickly. I encourage you to check out the Inversion of Control course if you want to learn more. When we configure StructureMap, we need to set up the object factory once and tell it what the different types are that it knows about so that when it gets a request for a particular type, it's able to fulfill that with an instance of that type. To start, we're going to scan the different assemblies to see where these types might live, so we're going to call or scan TheCallingAssembly which is our web project and also each of the other assemblies we're interested in which are the Core, Infrastructure, and Data project. Finally, we're going to specify WithDefaultConventions. This is a handy feature of StructureMap that basically makes it so that anytime you get a request for something that looks like say, IFoo, it's going to look and see if there is a class named Foo, and if there is, it will satisfy that request with that type if there's not another type already in its library. This can greatly reduce how much you have to actually configure StructureMap as long as you follow this convention. Now, down here, I am configuring Entity Framework so that when I get a request for a DbContext, I'm going to use my PluralsightBookContext which we created in the last module. I'm also configuring NHibernate so that requests for an ISessionFactory. I'm going to use this DatabaseConfiguration. CreateSessionFactory that we made last time as well. Note a couple of features here. First, I'm able to specify in StructureMap that this should be a ThreadLocalScoped or HTTP scoped object. What that means is that this DbContext will live only for the duration of each HTTP request in this particular case. In the case of NHibernate, I want a little bit different behavior. I don't want to create a different session factory for every request. I only want to have one session factory so I'm going to create it as a Singleton. You could learn more about the Singleton design pattern or anti-pattern in the Pluralsight Design Patterns Library. This is the preferred way to produce a Singleton as opposed to putting the logic for a Singleton into the class itself because it's very easy now to change whether or not this particular type is going to behave as a Singleton or not by simply adding or removing this call to. Singleton. Configuring everything else is a simple matter or saying, "For this type, I want to use this implementation. " And finally, for web forms, we're going to use something called property injection instead of constructor injection because web forms don't provide an easy way to construct web pages. And so we're going to automatically set any property that's of type IFriendsService whenever we call a particular helper method on StructureMap which we'll do from a base page class. So now, let's look at that base page. If you've been doing ASP. NET code for a while, you know that having a base page class can be a best practice for web forms development. It simply inherits from System Web UI Page and adds some common logic that your pages will use. In this case, the magic that's going to make StructureMap fill that property with the type from StructureMap's library is this call right here. So we're going to say ObjectFactory. BuildUp this. We're putting that directly into the BasePage constructor. Now, we have two classes that we wanted to use StructureMap to resolve dependencies in services. Those were the Friends and AddFriend classes. Let's look at the code behind. First off, notice that AddFriend is now inheriting from base page so it's going to get that new behavior. It has a new property which is this IFriendsService property and where we used to be calling code like this in order to instantiate this FriendsService, now, that FriendsService is going to get populated for us by StructureMap and we're just going to call it directly here. The nice thing about this is that, you'll notice I'm not instantiating anything in all of this code. It's all being handed to be StructureMap so my code is very loosely coupled. If I need to change, how am I going to do notification or how I am going to do sending an email? I can do that in one place, right here, change that for ISendEmail, so now I want to send them for real, and every page that deals with sending email will suddenly use the new implementation. Looking at the Friends code behind, it's much the same thing. We've got the FriendsService property now set we're inheriting from the base page. And down here, you can see, we just call the FriendsService method instead of having to instantiate it directly ourselves. Now, let's see how this works in action. I've added a little bit here to the bottom, there's a string list types I'm going to call on the page to see how we're doing this and this is going to display the ObjectFactory. GetInstance of the FriendRepository and the IQueryUsersByEmail. Tostring. So, if we look at our StructureMapBootStrap, you can see, we're using NHibernate for now. So, when we run our example, if we scroll to the bottom and look at our Friends, down here, you'll see we're using NHibernate FriendRepository and NHibernateQueryUsersByEmail, and our data is actually working. We can add a friend if we want. And now, let's go in and change how this works so that we are not using NHibernate, but instead we'll use our Entity Framework code. You could actually do this with just a button in your code if you wanted to. There is no need for you to have to rebuild. You can, at any time, change what you're using in StructureMap in terms of how it should fulfill these requests. In this case, I don't expect to have to change this except when I do a new deployment so I'm happy to just recompile. Now, let's look back at our page. Let's try and delete a friend, and if we look down at the footer, you'll see now were' using the EfCodeFirstFriendRepository and

QueryUsersByEmail. So, these interfaces have now been wired up differently using StructureMap and we're able to add friends once again now using Entity Framework.

New Requirements

All right, so now, let's talk about what we need to build for this module. We've got our Pluralsight book application and now users have come up with a new requirement. I know users do that sometimes. Our need is for a new front-end for Pluralsight book because some users have complained that they don't like having to use the web browser to see their friends. These are our power users and they prefer to just run a command periodically to see their current list of friends. They've even proposed a name for the new utility application, JustShowMeMyfriends. Since currently, anybody can be view the friends of users on the site, there's currently no need to include any authentication in the application so that will keep it fairly simple. Each user will simply need to configure the application once with their user information. So now, we have our new user requirement. It's pretty simple. Let's think about how we want to design this new front-end application. The first decision we have to make is pretty easy. This is going to be a console application. Now, the second one is what will this console application communicate with? Now, we could have it work directly with the database which should be the simplest thing or we could have it talk to some kind of a service or web service or web API. For now, we should probably do the simplest thing since nothing in our requirements prohibits this approach. This is an example following YAGNI, You Ain't Going to Need It. In any case, we want to ensure we continue to use our core business logic and domain types.

Demo: A Console App

So to create the new JustShowMeMyFriends console application, we've simply added a new console app and then updated it with some of the things that we need such as references to Entity Framework, FluentNHibernate, StructureMap, and the other projects that we'll work with. We start with our Program. cs file which you can see is pretty simple. In this case, we are going to configure StructureMap. Again, this is one of the first things you'll need to do if you're using an IOC container, is to configure that as soon as your application starts. We need to pull in some configuration data that has the user's email address. We're going to use this to look up which user we're dealing with. Now, we're going to pull out an instance of the Friends report type. This is where we've encapsulated all the logic of our application that deals with getting and displaying the friends of the current user. And we're going to call that in this console write line and then wrap up with a "press any key to end. " Note that we are using StructureMap to request an actual instance and not an interface in this case. It's not required that you only resolve interfaces with StructureMap or other IOC containers. You can actually use this to resolve a type. Now, since I already know this is the type I want to get back, how is this valuable? Well, in this case, I've done the standard dependency injection pattern here in the constructor of Friends report. So, I am saying that I want to get an IFriendsService and an IUserService injected when I get an instance of this class. And StructureMap will do that for me. So, it's going to, at runtime, determine which implementation of IFriendsService or IUserService I want to have and inject those into this class. The rest of this is fairly straightforward. It's simply going to call the userService to get to user, the friendsService to get the friends, and then a simple little loop here to display the friends out to the console. Our StructureMapBootStrap class looks extremely similar to what we just saw in the web forms application. And, in fact, we could clean this up and I'm sure I will before I wrap this module up so that some of the things that we don't need can be removed. So now let's run this and see how it works. So here, you can see just as we did on the web application, all the friends of steve@domain. com include these two listed right here, including the type that I made in the last demo in this module.

Demo: Adding Services

All right, so we're done, yay team. Remember, shipping is a feature. Your software must have it. It's time to celebrate. Oh, but wait, we just made an application for all of our users that talks directly to our database. And now that we've shipped it, we're getting some new information. It turns out that some users have complained that they can't connect to our database due to the firewall and our DBA is currently having a heart attack because we're letting random users hit our database directly. We need to introduce a Service layer. Currently, we only need a single method which is ListFriendsOfUser. Let's look at how we can replace the method that we're currently using with one that talks to an API layer. We're going to need to build the API layer along the way. Okay. So now, we're ready to create an API that will provide some services that our console application can talk to. To this, we've added a new web project. We've done a file, add new project. Choose an ASP. NET MVC for web application and then when it prompts you, choose the web API project template. In here, you're going to find some controllers including a default home controller. And there's going to also be when you first created a ValuesController in that same folder. I prefer to move the API controllers into a folder called API. Now, the URL for ValuesController which is your default hello world API is just going to return back value1 and value2 if you issue it a Get to this URL. So if we run the page, set it as your startup project, here it is. Note the port that you're running up. And then we can use a tool like Fiddler to test this. So, we can take our URL here, and we can open up Fiddler and put our URL in here with the substring that we expect to use for our API. When we run this, we can see the request here. And if we double-click on it, we can see here it notices that it's JSON and that we have value1 and value2 as we expect. You can learn more about Fiddler and you guessed it, a Pluralsight course. There's also, I'm sure, some great Pluralsight content out there on web API. Now, let's look a little bit more on how we wired this up. Back in our Global. asax, this time, we're in an MVC application, so we did still have to do some initial work here to set up our IOC container. In this case, because MVC and web API have some support for IOC containers built-in, we just have to set the dependency resolver to be a StructureMapDependencyResolver and pass in StructureMap's container. That stuff you'll find in the IOC folder, I grabbed this off of the internet and put in here, you can get the source files from this course or just look at the screen, you need to have a

StructureMapDependencyResolver as well as a StructureMapScopeClass, in order for this work with WebApi. Now let's look at the actual implementation of the code that we're going to use. We needed to have a simple service that we could call to get all the friends of a particular user. So now let's look at the FriendsController, the FriendsController has only one method that we're interested in and that is get friends of user. I've noted here what the API looks like to call this so we're going to call API slash friends and then pass in the user ID in the query string. So let me go and grab my user ID from SQL Server. So here I'm going to make a quick call, figure out that my user ID looks like this from ASP Net membership in my PluralsightBook application. We're going to use that in fiddler to do a quick test, we're going to call api/friends? userId=this, so composer api/friends? userId=and now if we look at the response, we'll see that we get friend2@domain. com and yetanother@domaini. com just like we were getting before. So now, we have verified that our service using Web API actually works. Let's look at how we can change our console application so that instead of talking directly to the database, it goes through this new Web API. So we return to JustShowMeMyFriends and let's look at the current implementation of FriendsReport, FriendsReport currently only knows about services, doesn't know anything about data or databases or repositories. Let's look at our StructureMapBootstrap, you can see it's still configured to talk to any framework and then hibernate. Now, let's go back and look at how we can wire up JustShowMeMyFriends to talk to this new set of Web API services. If we look back at our program, we'll see once again that nothing has changed. We're still just using everything out of FriendsReport so this code was not touched. If we look at FriendsReport, again we can see none of this code has changed, everything here is exactly the way it was when we were talking directly to the database have added a new folder called WebApiServices. And in this folder, I've added implementations of the two interfaces that FriendsReport uses, that is IFriendsService and IUserService. In WebApiFriendService, I'm simply creating a new HTTP client in the ListFriendsOf implementation calling out to the address that we expect, specifying that the headers that we want to use and the String format that we want to use, getting back the response and then parsing that response into these friend types. You will get a response, I return back null. Notice that these Friends are of type Pluralsight. Core. Model. Friend still so we're continuing to reuse our core domain logic. We'll talk more about that in a moment though. Our Web Api user, service is similar, it has the same kind of code here that we'll refactor out in a moment, to eliminate duplication and then it returns back a user when it gets back a successful result from the Web API. Now, if we look finally at our StructureMapBootstrap, we can clean this up substantially. As you can see, I've commented out all of the database specific ORM stuff so we could completely eliminate everything related to Entity Framework, everything related to Nhibernate and EF code first and also everything related to web forms. So at that point, we would have a pretty simple implementation when you will need ISendEmail, so we have something that would be very, very small for this very, very simple application. Now let's look a little bit at how we could refactor these API calls, so that they don't have so much repetition. I've already gone having done this so let me just update this code while you watch. Now in this case, what I've done is I've created a new object whose responsibility is configuring the API, I've called that ApiConfig. ApiConfig has common logic that you saw before in each of the methods. These services simply have the ApiConfig passed in through dependency injection and then they call into it in order to get their request. So they used client here in order to make that call instead of using a local variable. Likewise, Web API user service does the same thing, and then wire this up so that we call ApiConfig whenever we instantiate one of these services and we get this passed in. We just need to set something up in StructureMap. And so here, you can see I'm configuring WebAPI calls so that whenever I have a request for HttpClient, I'm going to have a Singleton so I only use of these for the entire application and it will be constructed by calling ApiConfig. GetClient. So this is an example of using a lambda to designate how you want to construct the object that's going to be returned from StructureMap. Now, let's see if this actually runs so the moment of truth, and we can see that we still have a working application. Now, we are talking through services to our WebApi implementation.

DTOs and App Logic

Okay, with all that said, what about DTOs and App-Specific logic? If you're not familiar with the term, DTOs are simply Data Transfer Objects. They're typically classes that lack any behavior. They're just sets of properties and data. DTOs frequently act as messages between components or systems. They're what we send to or get back from services especially over the wire. Now, DTOs can provide a breed between two systems, sets the domain objects on either into the bridge can vary independently of one another, while the DTO layer acts as an adaptor between two or more systems. If an application requires a very simple new front end, especially one that only runs on your company's internal network, then frequently, all or most of the core in infrastructure layers can be reused. However, as applications grow in complexity or are deployed separately or externally from your main network, frequently each one will need to have its own core and infrastructure with domain logic and mobile objects specific to that application's behavior. The infrastructure for this new application will include implementations specific to the external dependencies that this system works with. At that point, the original system and the new system should communicate with one another through services which implement core interfaces. This is something you can learn more about as service-oriented architecture. This process mirrors the biological reproduction process of budding as shown here. Where eventually, the small dependent application becomes its own self-sufficient application. At that point, its implementation of services and its dependency on infrastructure are all completely isolated within that application. The idea of having code reused has given way to having in-code independence and the way that you would reuse code from the original application is by calling into it through a service layer.

Solution Organization

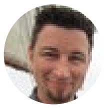
Now before we go back into the slides, let's have a look at what our solution looks like. Currently we have eight projects in here, two of which are test projects that we added in the first module in this part of the course. And then we have our core, our data, our infrastructure, our original website, just using ASP. Net web forms and our new WebApi ASP. Net MVC application as well. Now, a couple of these we can certainly merge for instance, PluralsightBook. Data only exists as a separate project because of how we constructed it using the entity framework reverse engineering tool. We can certainly pull that code and put it into infrastructure perhaps replacing this EDMX file that we were previously using. However, we still have a few different front end applications all sharing the same structure, it's good that JustShowMeMyFriends is able to reference and make use of our core so we're getting that reused, likewise infrastructure and data, our referencing core, web references core, website references core, everything thing in this solution references core, so we're getting a lot of reuse out of that. On the flipside however, every time we want to run this application, we have to set which one of these we're actually wanting to run. Although, we can't come into our solution properties and set startup projects. So if we were doing this a lot, we wanted to say, let me have multiple startup projects, we could actually set up all of these things at the same time, so that when we have our services and our console app that need to both start together, we can do it like this. But that's not my point. My point is, that we want to have some tests that refer to our core with the unit test and some integration with our infrastructure with our integration test and cover our main application which at this point is still our web forum's website. Our API services and our JustShowMeMyFriends app are really different applications. And ultimately, I expect that they're going to get some complexity of their own, such that they're going to need their own core domain objects that won't really necessarily have anything to do with our main website. So let's talk a little bit about how these applications might evolve, and how we can configure our solutions. When you're thinking about how you structure you solutions in Visual Studio, remember a few tips. First, you don't have to live with just one solution that your company or your team has decided as the one right solution for you to use. It's really simple for you to create your own solution that has just the projects you need. And it's a good idea if you just strive to find the Goldilocks solution. That is, the solution that has not too many projects and not too few. You'll find that if you have too many projects, your build process will be slow, and too many things will have to happen every time you build in order for you to be productive. If you have too few projects, you may find that when you do some sort of a refactoring using an automated tool, that automatically renames a class or changes a signature, that it does not change things that call that class in projects that aren't in your solution. Thus, you won't find out that these are your problem and that you've broken the code until your continuous integration server or, even worse, one of your fellow developers runs into the issue. I personally prefer to have one solution per unit of deployment or execution for my primary work. So that is, each front end, each deployable execution, whether that's a web app or WPF app or a console app, gets its own solution. Now, that doesn't mean that that's the only solution that I'm going to work with. I will frequently also have "an everything" solution. Now, the everything solution includes all of the projects, and this is the one that you want to run when you're doing a continuous integration process. It can run all the unit test, it can run all the integration test, and it can make sure that everything builds together. If you know that you're going to be making some sweeping API changes to one of the highly dependent parts of your application, it's also probably a good idea to open up the everything solution before you start doing those refactorings. Finally, if you do adopt a multiple solution strategy, it's a good idea for you to make sure that the way that you get updates to the libraries that you depend on is automated. Now, you can do this in a number of ways, you can use a batch file or some other kind of scripting technique, or

NuGet, perhaps, with your own local NuGet repository. The important thing here is that when updates get made to a core central library, that all the apps that depend on this library are able to get the newest bits in a very simple straightforward and automated fashion.

Summary

So in this module, we talked about code reuse, the holy grail of software development. We talked about how we can primarily reuse the core logic and some of the infrastructure in our application across multiple front end applications. The reason that we're able to do this is because we originally designed our application using the dependency inversion principle, and this is made easier using tools like IOC containers to automatically inject the correct instance of a type whenever we need one in our application. Finally, we showed how to reuse that application logic in a console application, as well as in a set of web API services within an MVCF. We started out with our console application using literally the same libraries to talk to the same database as our original web application, but quickly determined that this was not the ideal solution. And because of our design, we were able to quickly adjust how the console application talked to the underlying database and the services that it used to go through our web API services, instead of directly through our repositories. It's important to note that when we did this, we didn't have to change any of the actual implementation code in the console application. All we had to do was add a couple of new types, and then update StructureMap to use the new types instead of the old ones. A couple of good courses for you to check out if you're interested in learning more are my SOLID Principles of Object Oriented Design course and John Sonmez's Inversion of Control course, which talks about StructureMap and other IOC containers. This wraps up part two of creating N-tier applications in C#. I hope this has been helpful for you. Part three is going to be coming in 2013. It's going to start from a cleaned-up version of part two. We're going to still figure out what the topics are if part three hasn't been released by the time you're watching this, there might be an opportunity for you to vote on what the topics should be. Some possibilities include some more DomainDrivenDesign, including a little bit more theory than we've talked about so far, possibly implementing CQRS, Command Query Responsibility Separation. There's actually a course on Pluralsight available for this now. I've had some request to implement XAML and Prism, in terms of how those would interact at an N-tier architecture. So we may look at that. And I've also had some request to look at Cross-Layer Concerns such as Security, Logging, Validation, etc. In my examples that I've showed in this last module, you'll notice that I did not talk about Security, or any of these other concerns, that make it a little bit more complicated. So we can look at those as other possible topics to include in part three of this course series. Thank you very much for watching, my name is Steve Smith, you'll find me online @ardalis. com. I hope you've enjoyed this course now, see you again soon.

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	★★★★☆ (469)
My rating	★★★★★
Duration	1h 40m
Released	31 Dec 2012

Share course