

Optional 7a : Unit Testing Controllers

Objective

Best practice dictates that we should use dependency injection to inject entity framework context into controllers. One reason for this is because it enables controllers to be easily unit-tested.

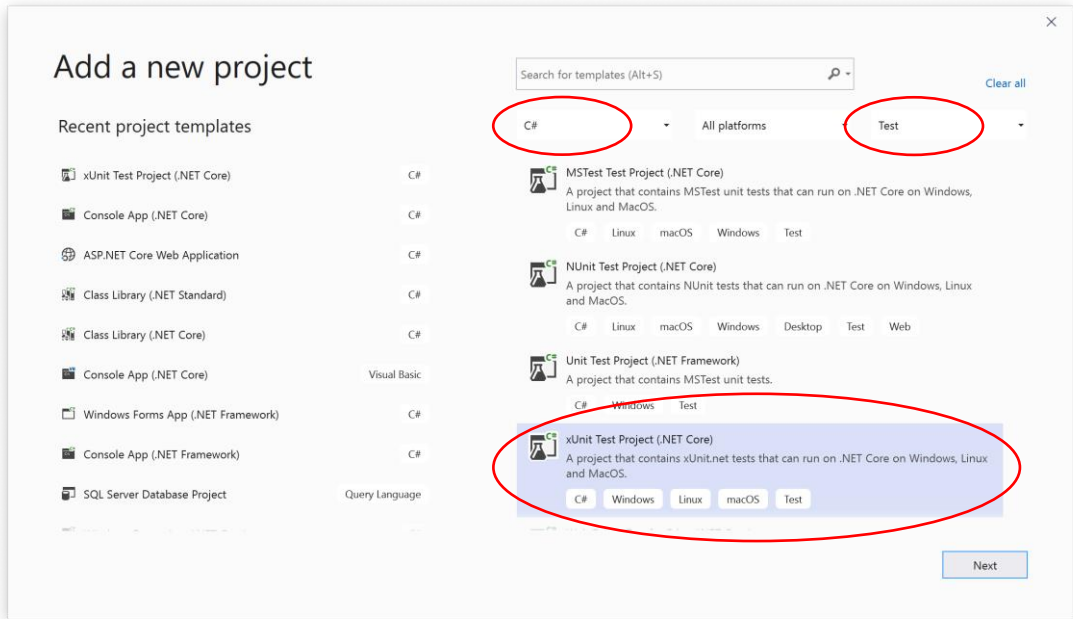
In unit testing, it's important that the test does not use a real database. If it were to use a real database, we may find that tests start failing if the data in the database changes. It's common to create a mock database, and entity framework core enables us to do this easily by using an InMemory database provider, where the test can set up exactly the data that's needed for the test.

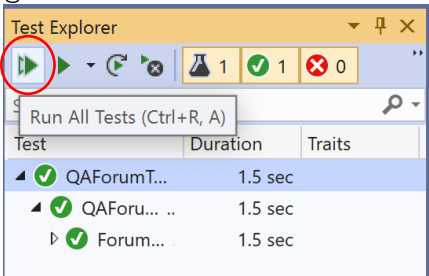
This exercise will take around 30 minutes.

Referenced material

This exercise is based on material from the chapter "Unit Testing Controllers".

Writing Unit Tests

1	Open the 'Begin' solution in Visual Studio and compile (Shift Ctrl+B). This project is identical to the End from the previous lab
2	<p>Right-click on the solution, and add a new project. For the project type, choose an xUnit Test Project (.NET Core) as shown below:</p>  <p>Call the project QAForumTests</p>
3	From the test project, right-click on Dependencies, and add a reference to the QAForum project
4	Use NuGet to add references to the following packages to the test project: Microsoft.EntityFrameworkCore.InMemory Microsoft.Extensions.DependencyInjection
5	Rename the file UnitTest1.cs – call it ForumControllerTests.cs. When Visual Studio prompts to ask if you also want to rename code elements, select Yes.
6	Inside that file, rename the test method Test1(). Call it Model_Contains_Correct_Thread_Count_Test()
7	<p>Create a method inside ForumControllerTests which will create a test forum:</p> <pre> private Forum GetMockForumWith2Threads() { var threads = new List<Thread> { new Thread { ThreadId = 1, Title = "Thread 1", UserName="Ted"}, new Thread { ThreadId = 2, Title = "Thread 2", UserName="Ted"}, }, }; return new Forum { ForumId = 1, Title = "Test Forum", Threads = threads }; } </pre>

8	<p>Now create a test:</p> <pre data-bbox="322 232 1476 824">[Fact] public void Model_Contains_Correct_Thread_Count_Test() { // Arrange var options = new DbContextOptionsBuilder<ForumDbContext>() .UseInMemoryDatabase("MockForumData").Options; var context = new ForumDbContext(options); context.Forums.Add(GetMockForumWith2Threads()); context.SaveChanges(); var controller = new ForumController(context); // Act var result = controller.Index() as ViewResult; var model = result.Model as IEnumerable<ForumViewModel>; // Assert Assert.Equal(2, model.Single().NumberOfThreads); }</pre> <p>Take some time to understand what's going on here. The ForumDbContext class does not contain any options (it doesn't even include the option to use SqlServer). In the web application, those options are injected into it using dependency injection.</p> <p>So, all we need to do to use different options is to create a ForumDbContext and pass in our own options – in our case, this means we pass in the option to use an in-memory database. We can then add whatever data we want to our in-memory database, as required by the test, without it having any effect on the real database.</p>
9	<p>From the Visual Studio menu bar, select Test/Test Explorer. In the test explorer window, click on the Run All Tests button, and check that your test bar turns green:</p> 
10	<p>This pattern of creating our own in-memory database is very straightforward. If our controller requires multiple constructor parameters, though, and they in turn require their own constructor parameters, it can be useful to use dependency injection directly within our test project. Let's do that next.</p>

11	<p>Create a method that gets the service provider for us, with all the required services pre-registered:</p> <pre data-bbox="322 264 1481 568"> private IServiceProvider GetServiceProvider() { ServiceCollection services = new ServiceCollection(); // Add whatever services we need to our dependency injection here services.AddDbContext<ForumDbContext>(options => options.UseInMemoryDatabase("MockForumData")); services.AddScoped<ForumController>(); return services.BuildServiceProvider(); } </pre> <p>Note that we add the forum controller to the service provider using the “scoped” lifetime. When we add the DbContext, we don’t choose the lifetime because we’re using the AddDbContext() helper method to add it for us – but this method adds the DbContext also with a scoped lifetime. If we ask for the DbContext more than once, so long as we do that in the same scope, we will get the same DbContext, with the same in-memory data set.</p>
12	<p>Modify the test to use the service provider:</p> <pre data-bbox="322 824 1481 1480"> [Fact] public void Model_Contains_Correct_Thread_Count_Test() { // Arrange var services = GetServiceProvider(); using (var scope = services.CreateScope()) { var context = scope.ServiceProvider.GetService<ForumDbContext>(); context.Forums.Add(GetMockForumWith2Threads()); context.SaveChanges(); var controller = scope.ServiceProvider.GetService<ForumController>(); // Act var result = controller.Index() as ViewResult; var model = result.Model as IEnumerable<ForumViewModel>; // Assert Assert.Equal(2, model.Single().NumberOfThreads); } } </pre> <p>(In this example, we explicitly create a scope. This is not strictly necessary – if you create a service provider outside of a web app, and don’t explicitly create a scope, the scope is considered to be the lifetime of the service provider.) Because we are now using the service provider to create the controller, it will automatically use dependency injection to inject not only the DbContext, but also any other dependencies that the controller has, so long as they were registered in the GetServiceProvider() method.</p>
13	<p>Re-run the test and make sure it still works.</p>
14	<p>Now, we’re going to write another test. This test is going to update the database, and we are going to assert that the database was updated correctly – all without touching a real database at all!</p>

15	<p>Add the following test to the ForumControllerTests class:</p> <pre> [Fact] public void Welcome_Post_Added_Test() { // Arrange var services = GetServiceProvider(); using (var scope = services.CreateScope()) { var context = scope.ServiceProvider.GetService<ForumDbContext>(); var controller = scope.ServiceProvider.GetService<ThreadController>(); var viewModel = new ThreadCreateViewModel { ForumId = 1, Title = "Test thread", UserName = "Test user", AddWelcomePost = true }; // Act controller.Create(viewModel); var post = context.Posts.Single(); // Assert Assert.Equal("Welcome to this thread. We hope it provides you with useful information", post.PostBody); } } </pre>
16	<p>Register the Thread Controller with the dependency injection service provider:</p> <pre> private IServiceProvider GetServiceProvider() { services.AddScoped<ForumController>(); services.AddScoped<ThreadController>(); return services.BuildServiceProvider(); } </pre>
17	<p>Run the tests and check that the new test passes. Spend a moment understanding how it works. Without InMemory databases, we would normally need to use a mocking framework to check how a service has been used by our method. But with an InMemory database we can simply check that it contains the data we expect.</p>

In this lab, we have created only two relatively simple test. In real life, you should unit test everything you write.

If you are already familiar with unit testing, then we hope that this lab shows how you can use your existing knowledge of unit testing and apply it to ASP.Net Core applications. And if you are not familiar with unit testing, we hope that this lab encourages you to learn more about the topic and introduce it as part of your regular workflow.