# Lab: Web Services Testing

This document defines the lab assignment from the ["Web Services and Cloud" Course @ Software University](https://softuni.bg/courses/web-services-and-cloud/).

This lab is a continuation of the **previous** [**lab**](https://softuni.bg/trainings/1191/Web-Services-and-Cloud-Aug-2015) **assignment** from the **Web API Architecture** topic. The goal is to practice writing **unit tests** (by mocking the data layer).

# Implement the Repository and Unit of Work Patterns

**Note**: **Repository** and **Unit of Work patterns** on top of ORM is a controversial topic and somewhere it might be considered anti-pattern (bad practice). It adds additional complexity to our code, but it also allows us to **easily** **mock the data layer** when **unit testing the controllers**. Keep this in mind when applying it.

The image below explains the idea behind **Repository** + **Unit of Work**:



1. Define a generic repository **interface** **IRepository<T>**:

|  |
| --- |
| **IRepository.cs** |
| **public interface IRepository<T>**  **{**  **IQueryable<T> All();**  **T Find(object id);**  **void Add(T entity);**  **void Update(T entity);**  **void Delete(T entity);**  **}** |

1. Define a **generic class** for creating repositories of custom type **T**, which implements the above interface.

|  |
| --- |
| **GenericRepository.cs** |
| **public class GenericRepository<T> : IRepository<T>**  **where T : class**  **{**  **protected DbContext context;**  **protected DbSet<T> set;**  **public GenericRepository(DbContext context)**  **{**  **this.context = context;**  **this.set = context.Set<T>();**  **}**  **public IQueryable<T> All()**  **{**  **return this.set.AsQueryable();**  **}**  **public T Find(object id)**  **{**  **return this.set.Find(id);**  **}**  **public void Add(T entity)**  **{**  **this.ChangeState(entity, EntityState.Added);**  **}**  **public void Update(T entity)**  **{**  **this.ChangeState(entity, EntityState.Modified);**  **}**    **public void Delete(T entity)**  **{**  **this.ChangeState(entity, EntityState.Deleted);**  **}**  **private void ChangeState(T entity, EntityState state)**  **{**  **var entry = this.context.Entry(entity);**  **if (entry.State == EntityState.Detached)**  **{**  **this.set.Attach(entity);**  **}**  **entry.State = state;**  **}**  **}** |

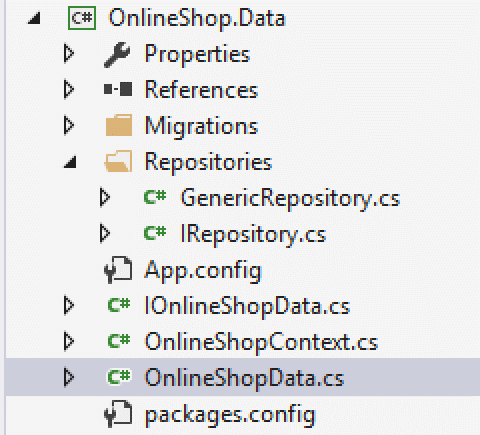
1. Now let's create an interface that represents our **unit of work** - it will hold **instances** of all **DbSets** wrapped in **repositories** and a **SaveChanges()** method for **committing** **changes** made through those repositories to the database:

|  |
| --- |
| **IOnlineShopData.cs** |
| **public interface IOnlineShopData**  **{**  **IRepository<Ad> Ads { get; }**  **IRepository<AdType> AdTypes { get; }**  **// TODO: Define repositories for all DbSets**  **int SaveChanges();**  **}** |

1. And finally, let's implement a concrete **unit of work**:

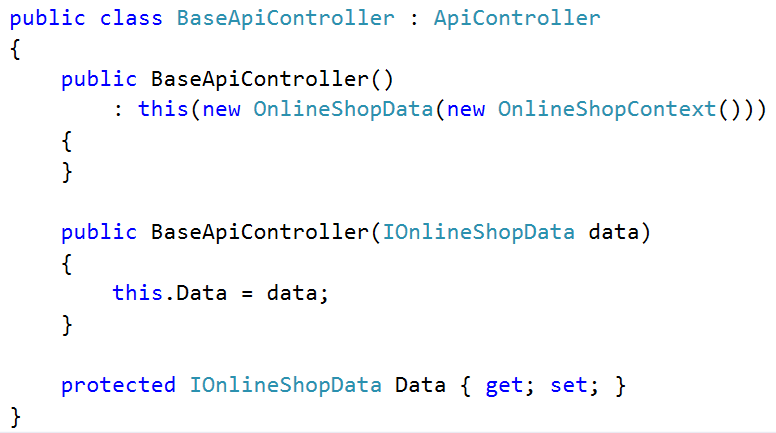
|  |
| --- |
| **OnlineShopData.cs** |
| **public class OnlineShopData : IOnlineShopData**  **{**  **private DbContext context;**  **private IDictionary<Type, object> repositories;**  **public OnlineShopData(DbContext context)**  **{**  **this.context = context;**  **this.repositories = new Dictionary<Type, object>();**  **}**  **public IRepository<Ad> Ads**  **{**  **get { return this.GetRepository<Ad>(); }**  **}**  **public IRepository<AdType> AdTypes**  **{**  **get { // TODO }**  **}**  **public IRepository<ApplicationUser> Users**  **{**  **get { // TODO }**  **}**  **public IRepository<Category> Categories**  **{**  **get { // TODO }**  **}**  **public int SaveChanges()**  **{**  **return this.context.SaveChanges();**  **}**  **public int SaveChangesAsync()**  **{**  **return this.context.SaveChanges();**  **}**  **private IRepository<T> GetRepository<T>() where T : class**  **{**  **var type = typeof(T);**  **if (!this.repositories.ContainsKey(type))**  **{**  **var typeOfRepository = typeof(GenericRepository<T>);**  **var repository = Activator.CreateInstance(**  **typeOfRepository, this.context);**  **this.repositories.Add(type, repository);**  **}**  **return (IRepository<T>)this.repositories[type];**  **}**  **}** |

This is how the **Data** project should look:



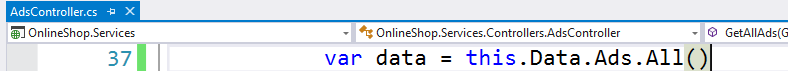
# Work through IOnlineShopData

Remember the **BaseApiController**? It defines a **Data** property which represents the **Unit of work**, responsible for CRUD operations with the database. So far it was **OnlineShopContext** - let's change it with the **IOnlineShopData** interface. That way we can **mock the data layer** by passing a **fake unit of work with fake repositories**.



Notice how there are two constructors - one accepting **IOnlineShopData** and one passing an instance of **OnlineShopData** to it. This is called **poor man's dependency injection** - that way we have a parameterless constructor (web api requires this) and one that accepts a custom IOnlineShopData instance.

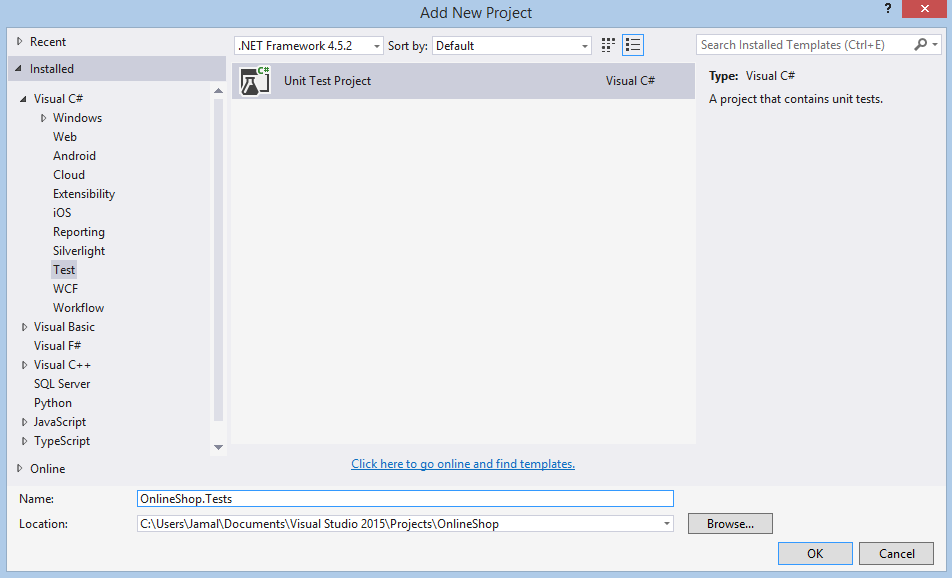
Add **.All()** to the old DbSet calls in the AdsController actions..



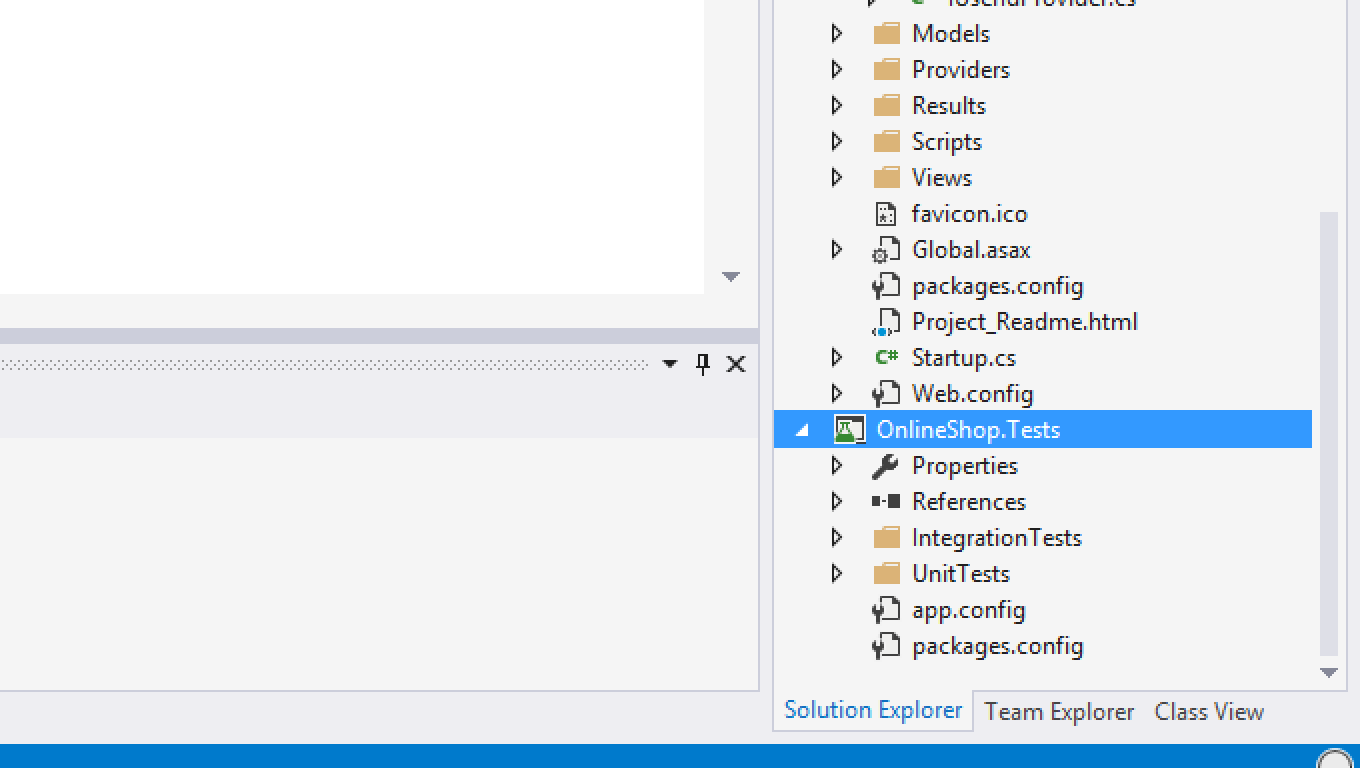
# Unit Testing the Ads Controller

Let's begin the testing.

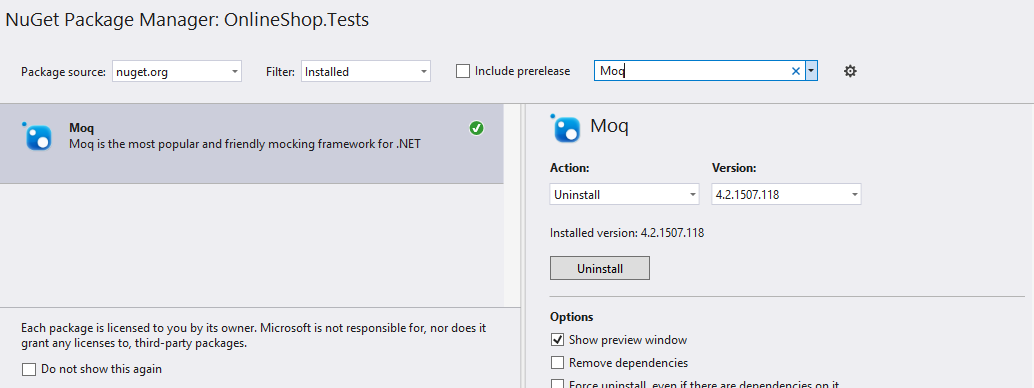
1. Create a new **Unit Test Project**. Name it **OnlineShop.Tests**.



1. Create 2 folders (namespaces) for holding our unit and integration test classes.



1. Install the **Moq** package from NuGet. Moq is a framework for easily mocking interfaces/classes.



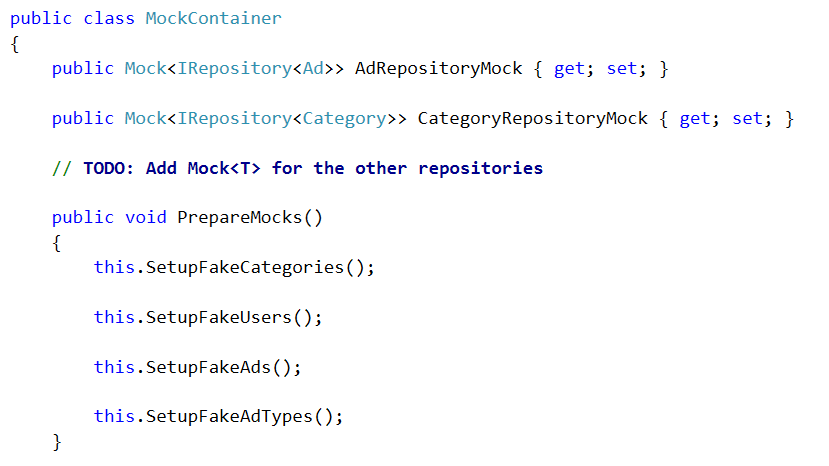
The below examples demonstrate how to create **fake repositories** with the Moq framework and manually by hand:

|  |  |
| --- | --- |
| **With Moq** | **Creating Fake Repository Class** |
|  |  |

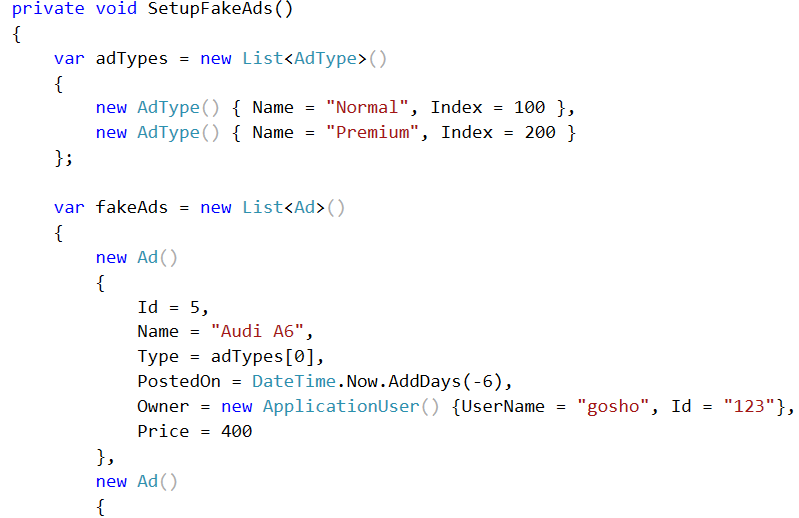
You may choose either way for mocking.

Let's start testing our **Ads controller**. We will be using the **Moq** framework.

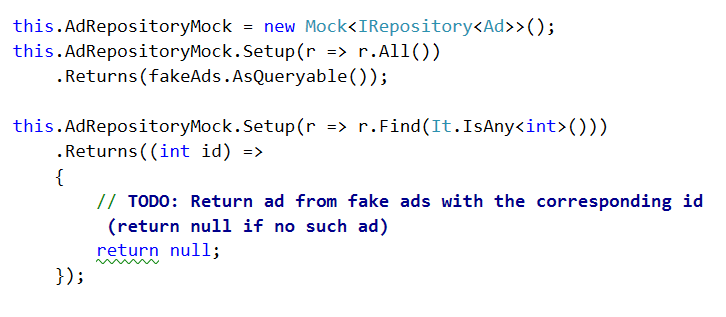
1. Create a new class **MockContainer**.
2. Define **Mock<T>** for each repository, where **T** is the **IRepository** interface.
3. Define a **public** **PrepareMocks()** method for setting up the fake repositories.  
   It will call **private** **SetupFake\*** methods.



1. Create a list with a couple of fake ad entities (along with their owner + type entities).



1. Then create the mock by initializing the **AdRepositoryMock** property.
   1. Setup the **All()** method to return the fake ads as queryable.
   2. Setup the **Find()** method to accept any **int id**. It should return the ad from the **fakeAds** collection with that id.



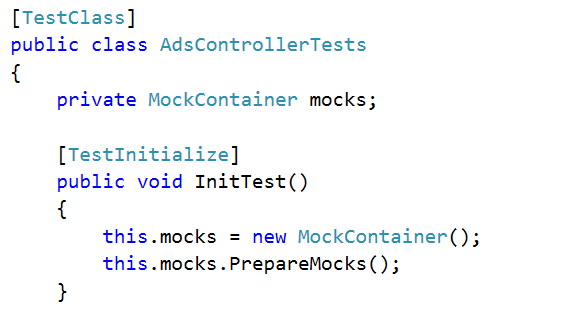
And we're done. Now do the same for every other mocked repository - **AdTypeRepositoryMock**, **UserRepositoryMock**, **CategoriesRepositoryMock**.

1. Setup the **All()** method to return a fake collection (no more than 3-4 entities) and the **Find()** method to return the fake entity with the corresponding **id**.

Continue after you've set all repository mocks.

1. Create a new **AdsControllerTests** class in the **OnlineShop.Tests.UnitTests** namspace. Mark it with the **[TestClass]** attribute so VSTT (Visual Studio Team Test) knows the class holds unit tests.
2. Create a **InitTest()** method and mark it with the **[TestInitiliaze]** attribute. This means the method will be called every time before a unit test is run.

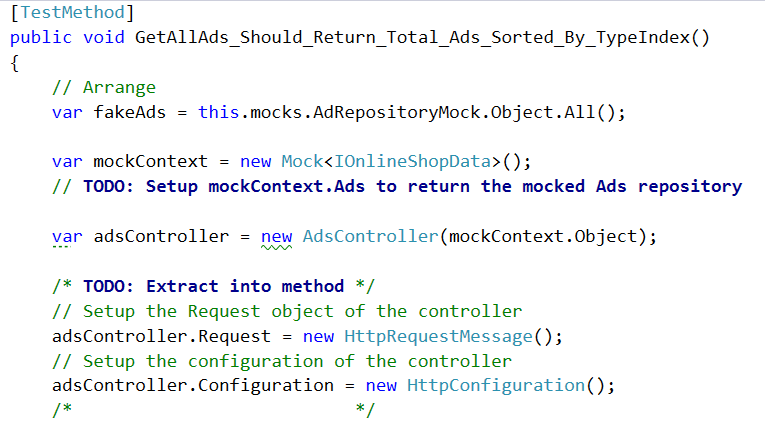
It should re-initialize the **MockContainer** and tell it to prepare the mocks. This way we make sure each unit test will run with the original mocks and it will not be affected by changes made in the previous tests.



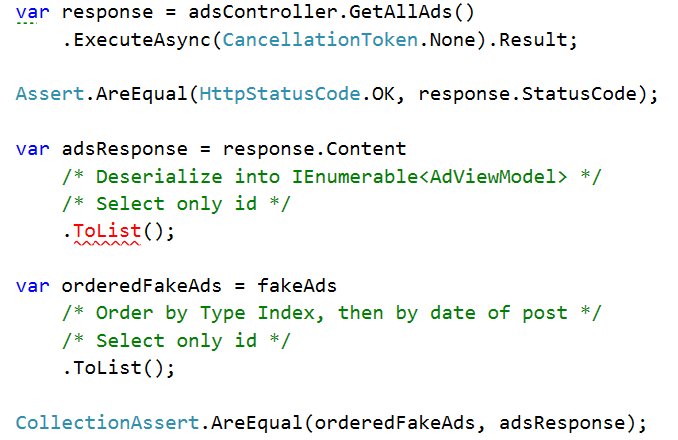
### 1. Get All Ads

Test the **GetAds()** action. Create a method with a **meaningful name** (describing what it tests) and mark it with the **[TestMethod]** attribute.

1. Arrange the **repositories** and the **context**.



1. **Act** - invoke the **GetAllAds()** method from the controller.
2. **Assert** that the response status code is **200 OK**
3. **Deserialize** the response data with **ReadAsAsync<T>()**. Select only the IDs.
4. Order the fake ads collection just like the controller does the ordering. Select only the IDs.
5. **Assert** that the two collections have the **same elements**

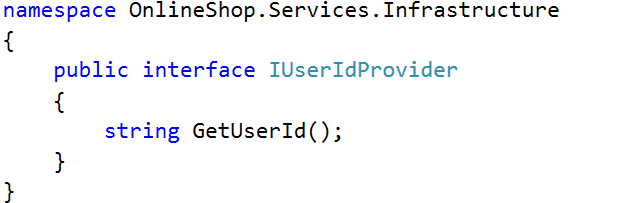


1. The test should successfully pass. If not, see the fail message and debug the unit test.

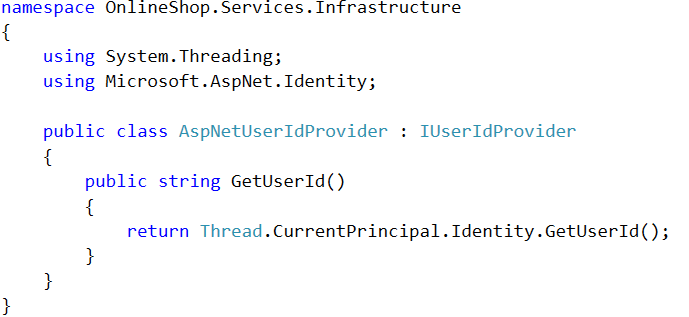
### 2. Create New Ad

Next we're going to test the **CreateAd()** action. However, there is a problem - it uses **User.Identity.GetUserId()** to get the author or the ad. When a request is sent, it has a **user** associated with it - but in our unit test there is no user, since there is no request and **GetUserId()** will always **return null** and our tests will fail. Therefore we need to mock **GetUserId()**.

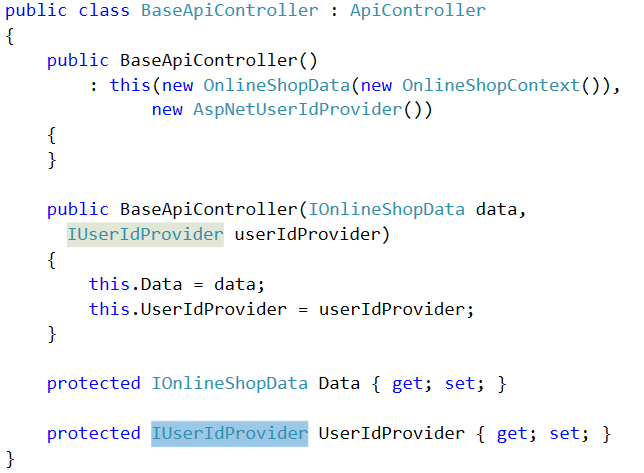
1. Define the dependency in an **interface** called **IUserIdProvider**:



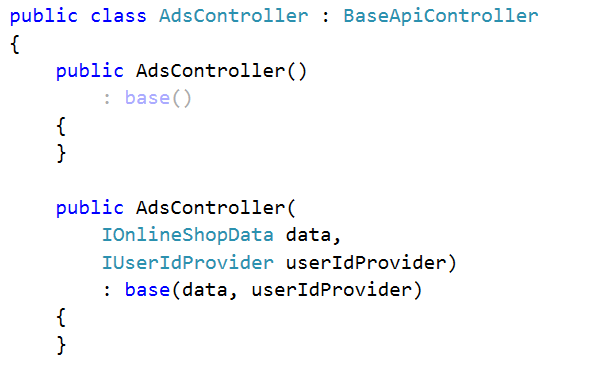
1. Create a concrete class called **AspNetUserIdProvider** which **implements the above interface**. The **GetUserId()** method will return the id of the current principal (user) of the executing thread.



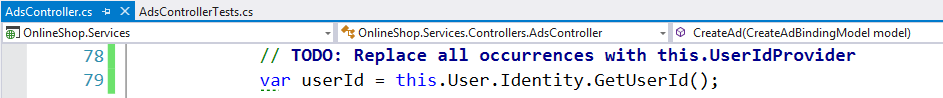
1. In **BaseApiController**, inject the **IUserIdProvider** dependency through the constructor and set it to a property.



1. The **AdsController** should have a **parameterless constructor** and a **constructor with 2 parameters** (making a base call).

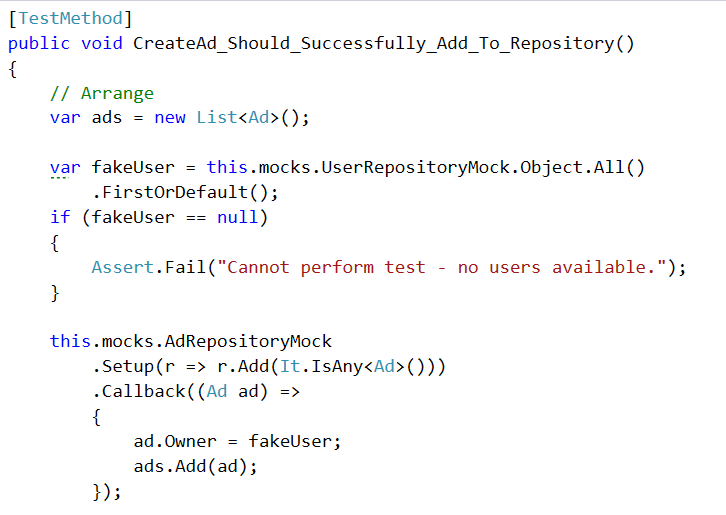


1. Change all uses of **this.User.Identity** in all controllers to **this.UserIdProvider**.

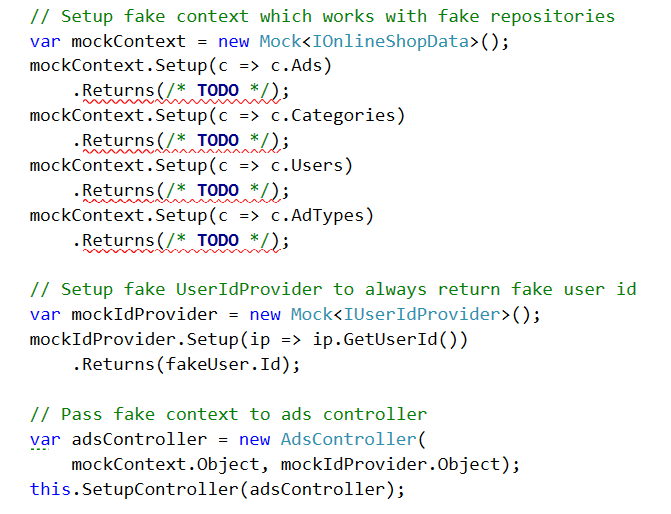


Now that we've extracted **getting the user id** into an **interface**, we can easily mock it.

1. Create a test method that tests if the **CreateAd()** action correctly adds an ad to the repository.
2. Create an **empty Ad list** (to which we will redirect **.Add()** to add new ads)
3. Get a **fake user** from the mocked user repository.
4. Setup the **.Add()** method of the **mocked** **AdRepositroy** to insert ads in the list.



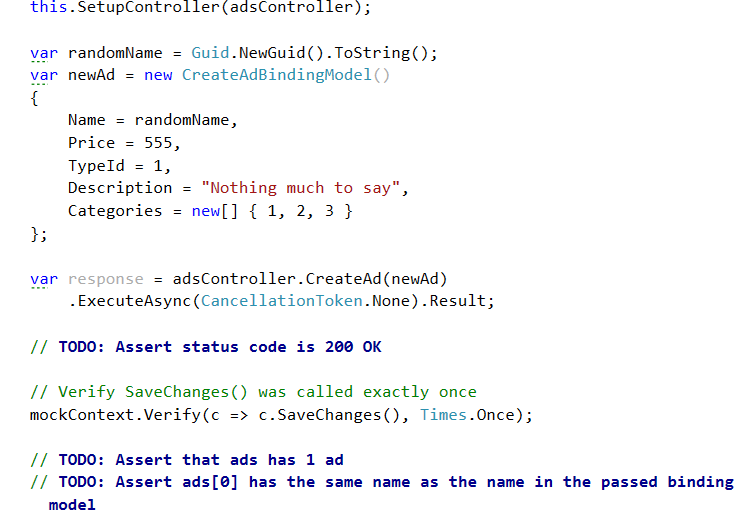
1. Setup a mock context. Setup its **Ads**, **Categories**, **Users** and **AdTypes** repositories to be the mocked repositories from our **MockContainer** object.
2. Setup a fake **UserIdProvider** with a **GetUserId()** method to always **return the id of fake user** we got earlier (see above).
3. Create a new **AdsController** and pass the **mocked context** and **mocked UserIdProvider** to its constructor.



**Note**: The **SetupController()** method is the one you were supposed to extract from the previous unit test.

Now the controller will work with a **fake context** (and respectively fake repositories which act as we have setup) and a **fake user id provider**.

1. Now let's create a binding model (the data we pass to the **CreateAd()** action).
   1. Generate a random string name by calling **Guid.NewGuid().ToString()**.
   2. Make sure you pass a **valid** **typeId** (one that exists in the mocked AdTypeRepository) and **valid** **category IDs** (ones that exist in the mocked CategoryRepository).
2. Execute the action.
3. Assert that the action executed successfully and the status code is 200 OK.
4. **Verify** that **SaveChanges()** was called **once** (thus the changes were successfully pushed to the database).
5. Remember the empty ad list we created in the beginning of this test method? Assert that it has exactly 1 ad inserted in it during this action call.
6. Asser that the newly added ad has the **same name** like the one in the binding model.

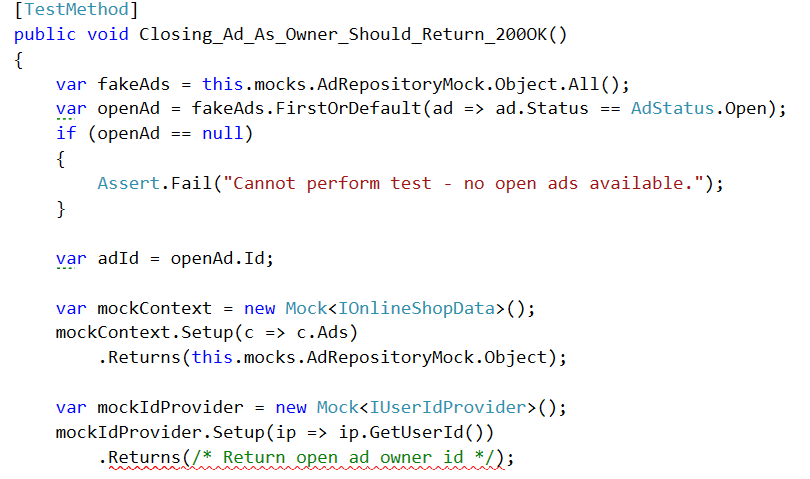


1. The test should successfully pass. If not, see the fail message and debug the unit test.

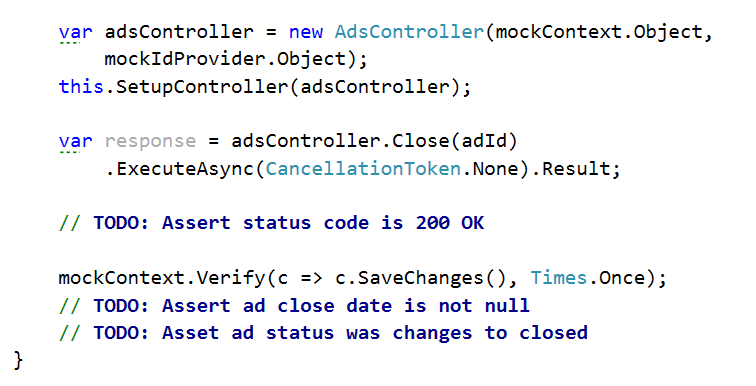
### 3. Close Ad as Owner

Write a test that checks if an ad owner can successfully close his ad.

1. Create a test method and give it a meaningful name.
2. Get a **single open ad** from the mocked **AdRepository**.
3. Create a **mock context**. Setup its **Ads** **repository** to return the **mocked AdRepository**.
4. Create a mock **IUserIdProvider**. Setup its GetUserId() method to return the **open ad's ownerId**.



1. Create a new **AdsController** and pass the **mocked context** and **mocked IUserIdProvider** to its constructor.
2. Execute the action.
3. **Assert** that the operation returned status code **200 OK**.
4. **Verify** that **SaveChanges()** was called exactly **once** (meaning the ad entity was actually updated).
5. **Assert** that the ad **CloseDate** isn't null(meaning it was changes by the action) and that the ad **status** is changed to **Closed**.



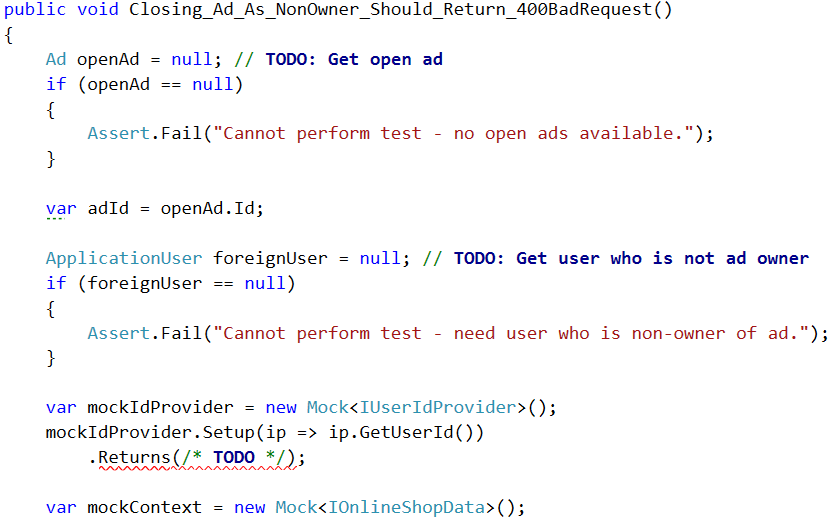
1. The test should successfully pass. If not, see the fail message and debug the unit test.

### 4. Close Ad as Non-Owner

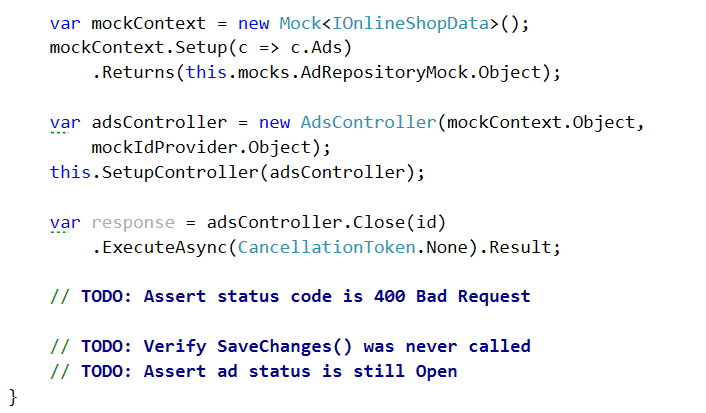
Let's perform the previous test again, but this time we will "send the request" as a non-owner of the ad (ads can only be closed by their owners).

Try writing this test without looking at the previous example!

1. Create a test method and give it a meaningful name.
2. Get a **single open ad** from the mocked **AdRepository**.
3. Get **a single user** from the mocked **UserRepository** who is **NOT the ad owner**.
4. Create a mock **IUserIdProvider**. Setup its **GetUserId()** method to return the **foreign user** (non-owner of the ad).



1. Create a **mock context**. Setup its **Ads** **repository** to return the **mocked AdRepository**.
2. Create a new **AdsController** and pass the **mocked context** and **mocked IUserIdProvider** to its constructor.
3. Call the **SetupController()** method.
4. Execute the action.
5. **Assert** that the returned status code is **400 Bad Request** (or **401 unauthorized**, depending on what your action returns).
6. **Verify** that **SaveChanges()** was **never called** (no changes were pushed to the database).
7. **Assert** that the ad status is still **Open**.

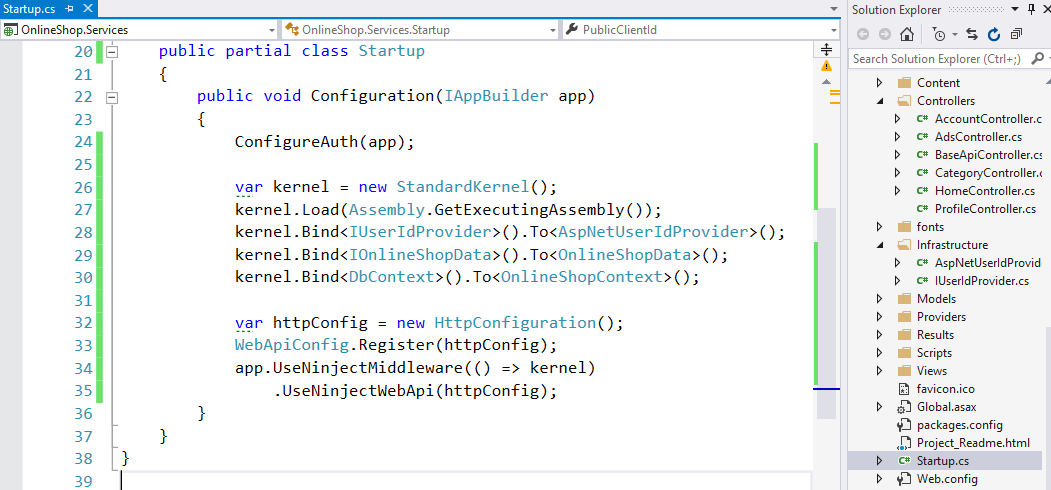


1. The test should successfully pass. If not, see the fail message and debug the unit test.

# 5. Using Ninject

Don't you think those parameterless constructors in the controllers that call **: this(…)** are ugly? Yes, they are! Let's do something about it.

1. Install **Ninject** and **Ninject.Web.WebApi.OwinHost** in the Services project.
2. In the Startup class create a new StandardKernel object. It will hold all dependencies and the classes we want to inject in their place (e.g. **IUserIdProvider -> AspNetUserIdProvider**, **IOnlineShopData -> OnlineShopData**, etc.).
3. Create these bindings (mappings) by calling **Bind<T>().To<K>()** for each dependency.



1. **app.UserNinjectMiddleWare()** makes sure for us that whenever Web API creates an instance of a controller it will call the 2-parameter constructor and pass the corresponding dependencies like so:

|  |  |
| --- | --- |
| **Constructor** | **Controller creation by Web API** |
| **public AdsController(**  **IOnlineShopData data,**  **IUserIdProvider userIdProvider)**  **: base(data, userIdProvider)**  **{**  **}** | **var adsController = new AdsController(**  **new OnlineShopData(**  **new OnlineShopContext()),**  **new AspNetUserIdProvider());** |

1. You may go to the controllers and the delete the parameterless constructors.

If you're getting an exception on startup of the sort "**Error activating HttpConfiguration**", go to the installed packages and **update** **Ninject.Web.WebApi** to the latest version.

