

Lab: Web Services Testing

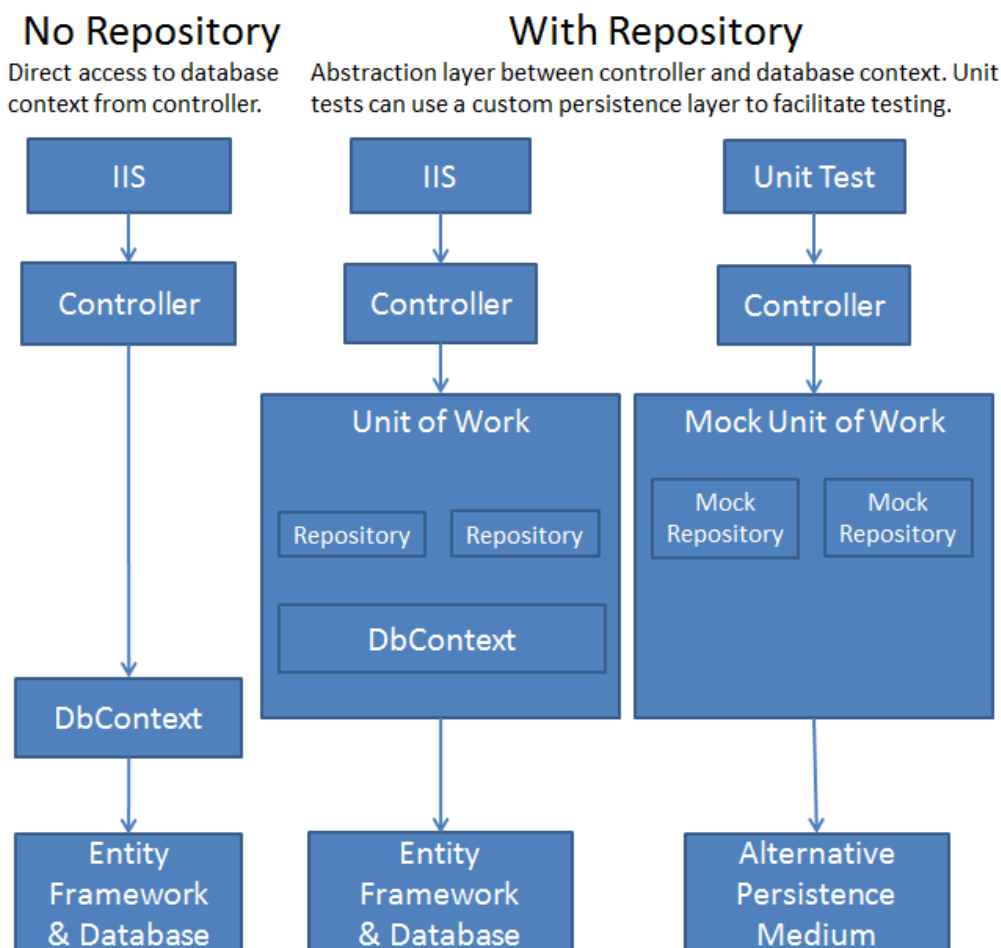
This document defines the lab assignment from the ["Web Services and Cloud" Course @ Software University](#).

This lab is a continuation of the **previous lab** assignment from the **Web API Architecture** topic. The goal is to practice writing **unit tests** (by mocking the data layer).

1. 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);
}

```

2. 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;
    }
}

```

3. 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:

```
OnlineShopData.cs

public interface IOnlineShopData
{
    IRepository<Ad> Ads { get; }

    IRepository<AdType> AdTypes { get; }

    // TODO: Define repositories for all DbSets

    int SaveChanges();
}
```

4. 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();
    }

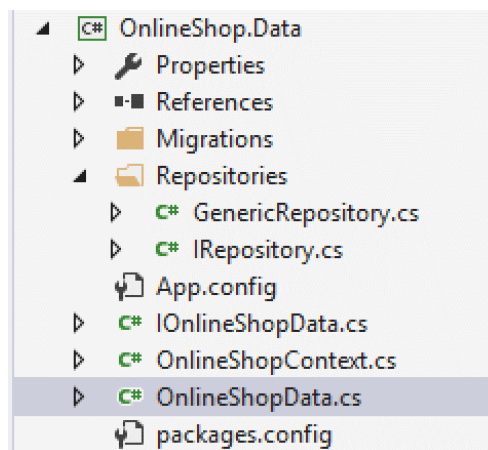
    private IRepository<T> GetRepository<T>() where T : class
    {
        var type = typeof(T);
        if (!this.repositories.ContainsKey(type))
        {
            var typeOfRepository = typeof(GenericRepository<T>);
            var repository = Activator.CreateInstance(
                typeOfRepository, this.context);

            this.repositories.Add(type, repository);
        }

        return (IRepository<T>)this.repositories[type];
    }
}

```

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



2. 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**.

```
public class BaseApiController : ApiController
{
    public BaseApiController()
        : this(new OnlineShopData(new OnlineShopContext()))
    {
    }

    public BaseApiController(IOnlineShopData data)
    {
        this.Data = data;
    }

    protected IOnlineShopData Data { get; set; }
}
```

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..

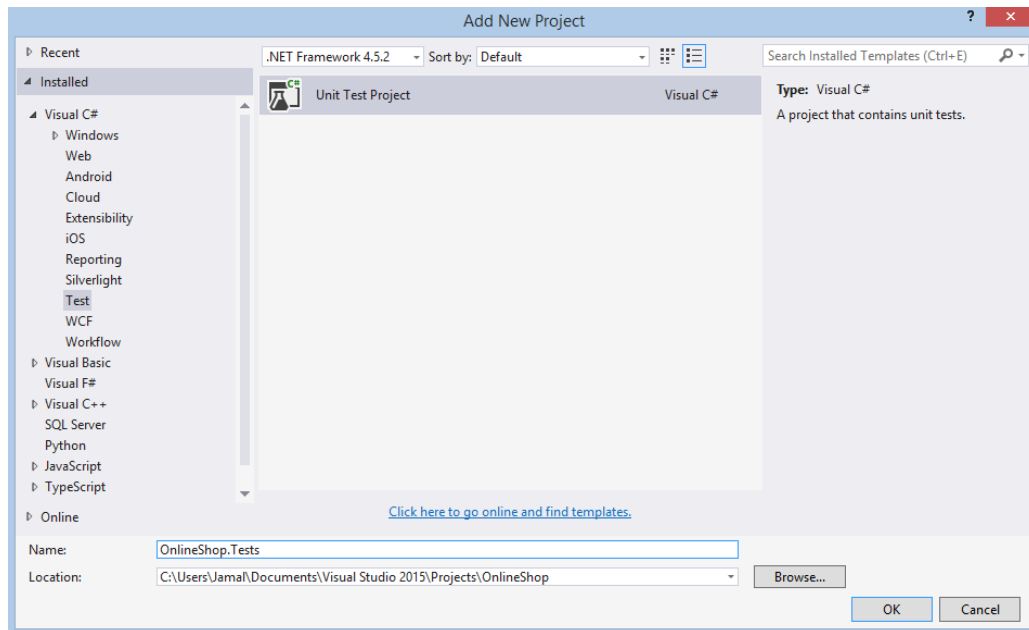


```
AdsController.cs 37 var data = this.Data.Ads.All()
```

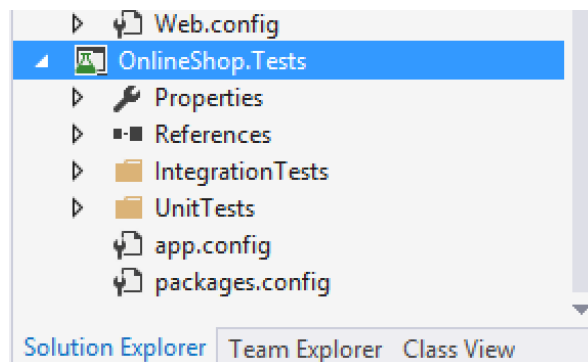
3. Unit Testing the Ads Controller

Let's begin the testing.

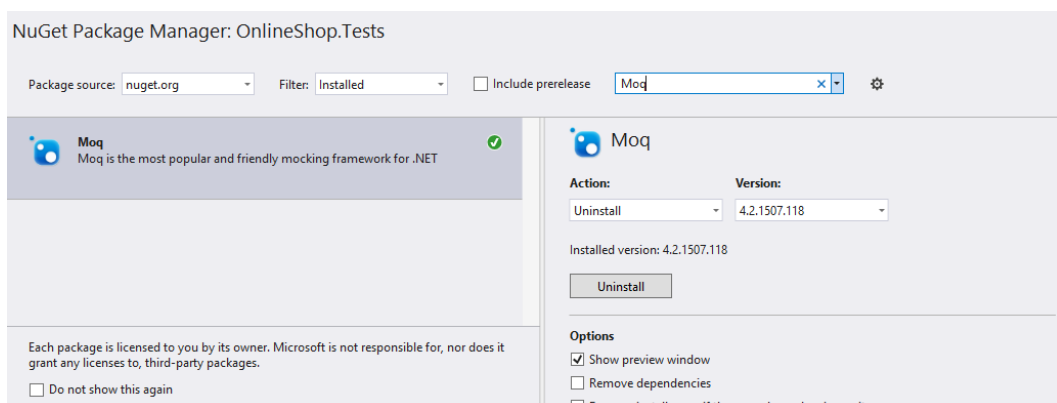
1. Create a new **Unit Test Project**. Name it **OnlineShop.Tests**.



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



3. 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
<pre> var fakeAds = new List<Ad>() { new Ad() { Id = 5, Name = "Audi A6" } }; var repositoryMock = new Mock<IRepository<Ad>>(); repositoryMock.Setup(r => r.All()) .Returns(fakeAds.AsQueryable()); repositoryMock.Setup(r => r.Add(It.IsAny<Ad>())) .Callback((Ad ad) => { fakeAds.Add(ad); }); var fakeRepository = repositoryMock.Object; </pre>	<pre> public class FakeAdsRepository : IRepository<Ad> { private List<Ad> fakeAds; public FakeAdsRepository() { this.fakeAds = new List<Ad>() { new Ad() { Id = 5, Name = "Audi A6" } }; } public IQueryable<Ad> All() { return this.fakeAds.AsQueryable(); } public void Add(Ad ad) { this.fakeAds.Add(ad); } } </pre>

You may choose either way for mocking.

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

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

```

public class MockContainer
{
    public Mock<IRepository<Ad>> AdRepositoryMock { get; set; }

    public Mock<IRepository<Category>> CategoryRepositoryMock { get; set; }

    // TODO: Add Mock<T> for the other repositories

    public void PrepareMocks()
    {
        this.SetupFakeCategories();

        this.SetupFakeUsers();

        this.SetupFakeAds();

        this.SetupFakeAdTypes();
    }
}

```

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

```

private void SetupFakeAds()
{
    var adTypes = new List<AdType>()
    {
        new AdType() { Name = "Normal", Index = 100 },
        new AdType() { Name = "Premium", Index = 200 }
    };

    var fakeAds = new List<Ad>()
    {
        new Ad()
        {
            Id = 5,
            Name = "Audi A6",
            Type = adTypes[0],
            PostedOn = DateTime.Now.AddDays(-6),
            Owner = new ApplicationUser() { UserName = "gosho", Id = "123"},
            Price = 400
        },
        new Ad()
        {

```

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

```

this.AdRepositoryMock = new Mock<IRepository<Ad>>();
this.AdRepositoryMock.Setup(r => r.All())
    .Returns(fakeAds.AsQueryable());

this.AdRepositoryMock.Setup(r => r.Find(It.IsAny<int>()))
    .Returns((int id) =>
    {
        // TODO: Return ad from fake ads with the corresponding id
        (return null if no such ad)
        return null;
    });

```

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

9. 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.

10. Create a new **AdsControllerTests** class in the **OnlineShop.Tests.UnitTests** namespace. Mark it with the **[TestClass]** attribute so VSTT (Visual Studio Team Test) knows the class holds unit tests.
11. Create a **InitTest()** method and mark it with the **[TestInitialize]** 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.

```
[TestClass]
public class AdsControllerTests
{
    private MockContainer mocks;

    [TestInitialize]
    public void InitTest()
    {
        this.mocks = new MockContainer();
        this.mocks.PrepareMocks();
    }
}
```

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**.

```
[TestMethod]
public void GetAllAds_Should_Return_Total_Ads_Sorted_By_TypeIndex()
{
    // Arrange
    var fakeAds = this.mocks.AdRepositoryMock.Object.All();

    var mockContext = new Mock<IOOnlineShopData>();
    // TODO: Setup mockContext.Ads to return the mocked Ads repository

    var adsController = new AdsController(mockContext.Object);

    /* TODO: Extract into method */
    // Setup the Request object of the controller
    adsController.Request = new HttpRequestMessage();
    // Setup the configuration of the controller
    adsController.Configuration = new HttpConfiguration();
    /* */
}
```

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

```

var response = adsController.GetAllAds()
    .ExecuteAsync(CancellationToken.None).Result;

Assert.AreEqual(HttpStatusCode.OK, response.StatusCode);

var adsResponse = response.Content
    /* Deserialize into IEnumerable<AdViewModel> */
    /* Select only id */
    .ToList();

var orderedFakeAds = fakeAds
    /* Order by Type Index, then by date of post */
    /* Select only id */
    .ToList();

CollectionAssert.AreEqual(orderedFakeAds, adsResponse);

```

7. 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**:

```

namespace OnlineShop.Services.Infrastructure
{
    public interface IUserIdProvider
    {
        string GetUserId();
    }
}

```

2. 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.

```

namespace OnlineShop.Services.Infrastructure
{
    using System.Threading;
    using Microsoft.AspNet.Identity;

    public class AspNetUserIdProvider : IUserIdProvider
    {
        public string GetUserId()
        {
            return Thread.CurrentPrincipal.Identity.GetUserId();
        }
    }
}

```

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

```

public class BaseApiController : ApiController
{
    public BaseApiController()
        : this(new OnlineShopData(new OnlineShopContext()),
            new AspNetUserIdProvider())
    {
    }

    public BaseApiController(IONlineShopData data,
        IUserIdProvider userIdProvider)
    {
        this.Data = data;
        this.UserIdProvider = userIdProvider;
    }

    protected IONlineShopData Data { get; set; }

    protected IUserIdProvider UserIdProvider { get; set; }
}

```

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

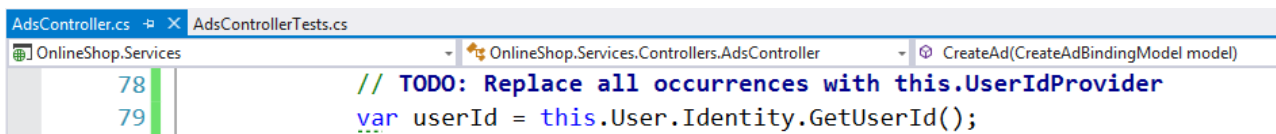
```

public class AdsController : BaseApiController
{
    public AdsController()
        : base()
    {
    }

    public AdsController(
        IONlineShopData data,
        IUserIdProvider userIdProvider)
        : base(data, userIdProvider)
    {
    }
}

```

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



The screenshot shows a Visual Studio window with two tabs: 'AdsController.cs' and 'AdsControllerTests.cs'. The 'OnlineShop.Services' namespace is selected. The 'OnlineShop.Services.Controllers.AdsController' class is open, and the 'CreateAd(CreateAdBindingModel model)' method is selected. The code editor shows line 78 with a TODO comment: '// TODO: Replace all occurrences with this.UserIdProvider' and line 79 with the code: 'var userId = this.User.Identity.GetUserId();'.

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 AdRepository** to insert ads in the list.

```
[TestMethod]
public void CreateAd_Should_Successfully_Add_To_Repository()
{
    // Arrange
    var ads = new List<Ad>();

    var fakeUser = this.mocks.UserRepositoryMock.Object.All()
        .FirstOrDefault();
    if (fakeUser == null)
    {
        Assert.Fail("Cannot perform test - no users available.");
    }

    this.mocks.AdRepositoryMock
        .Setup(r => r.Add(It.IsAny<Ad>()))
        .Callback((Ad ad) =>
        {
            ad.Owner = fakeUser;
            ads.Add(ad);
        });
}
```

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

```
// Setup fake context which works with fake repositories
var mockContext = new Mock<IOOnlineShopData>();
mockContext.Setup(c => c.Ads)
    .Returns(/* TODO */);
mockContext.Setup(c => c.Categories)
    .Returns(/* TODO */);
mockContext.Setup(c => c.Users)
    .Returns(/* TODO */);
mockContext.Setup(c => c.AdTypes)
    .Returns(/* TODO */);

// Setup fake UserIdProvider to always return fake user id
var mockIdProvider = new Mock<IUserIdProvider>();
mockIdProvider.Setup(ip => ip.GetUserId())
    .Returns(fakeUser.Id);

// Pass fake context to ads controller
var adsController = new AdsController(
    mockContext.Object, mockIdProvider.Object);
this.SetupController(adsController);
```

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**.

8. Now let's create a binding model (the data we pass to the **CreateAd()** action).
 - a. Generate a random string name by calling **Guid.NewGuid().ToString()**.
 - b. 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**).
9. Execute the action.
10. Assert that the action executed successfully and the status code is 200 OK.
11. **Verify** that **SaveChanges()** was called **once** (thus the changes were successfully pushed to the database).
12. 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.
13. Assert that the newly added ad has the **same name** like the one in the binding model.

```
this.SetupController(adsController);

var randomName = Guid.NewGuid().ToString();
var newAd = new CreateAdBindingModel()
{
    Name = randomName,
    Price = 555,
    TypeId = 1,
    Description = "Nothing much to say",
    Categories = new[] { 1, 2, 3 }
};

var response = adsController.CreateAd(newAd)
    .ExecuteAsync(CancellationToken.None).Result;

// TODO: Assert status code is 200 OK

// Verify SaveChanges() was called exactly once
mockContext.Verify(c => c.SaveChanges(), Times.Once);

// TODO: Assert that ads has 1 ad
// TODO: Assert ads[0] has the same name as the name in the passed binding
model
```

14. 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**.

```

[TestMethod]
public void Closing_Ad_As_Owner_Should_Return_200OK()
{
    var fakeAds = this.mocks.AdRepositoryMock.Object.All();
    var openAd = fakeAds.FirstOrDefault(ad => ad.Status == AdStatus.Open);
    if (openAd == null)
    {
        Assert.Fail("Cannot perform test - no open ads available.");
    }

    var adId = openAd.Id;

    var mockContext = new Mock<IOnlineShopData>();
    mockContext.Setup(c => c.Ads)
        .Returns(this.mocks.AdRepositoryMock.Object);

    var mockIdProvider = new Mock<IUserIdProvider>();
    mockIdProvider.Setup(ip => ip.GetUserId())
        .Returns(/* Return open ad owner id */);
}

```

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

```

var adsController = new AdsController(mockContext.Object,
    mockIdProvider.Object);
this.SetupController(adsController);

var response = adsController.Close(adId)
    .ExecuteAsync(CancellationToken.None).Result;

// TODO: Assert status code is 200 OK

mockContext.Verify(c => c.SaveChanges(), Times.Once);
// TODO: Assert ad close date is not null
// TODO: Asset ad status was changes to closed
}

```

10. 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).

```
public void Closing_Ad_As_NonOwner_Should_Return_400BadRequest()
{
    Ad openAd = null; // TODO: Get open ad
    if (openAd == null)
    {
        Assert.Fail("Cannot perform test - no open ads available.");
    }

    var adId = openAd.Id;

    ApplicationUser foreignUser = null; // TODO: Get user who is not ad owner
    if (foreignUser == null)
    {
        Assert.Fail("Cannot perform test - need user who is non-owner of ad.");
    }

    var mockIdProvider = new Mock<IUserIdProvider>();
    mockIdProvider.Setup(ip => ip.GetUserId())
        .Returns(/* TODO */);

    var mockContext = new Mock<IOnlineShopData>();
```

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

```
var mockContext = new Mock<IOnlineShopData>();
mockContext.Setup(c => c.Ads)
    .Returns(this.mocks.AdRepositoryMock.Object);

var adsController = new AdsController(mockContext.Object,
    mockIdProvider.Object);
this.SetupController(adsController);

var response = adsController.Close(id)
    .ExecuteAsync(CancellationToken.None).Result;

// TODO: Assert status code is 400 Bad Request

// TODO: Verify SaveChanges() was never called
// TODO: Assert ad status is still Open
}
```

12. 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**, **IOneShopData** -> **OnlineShopData**, etc.).
3. Create these bindings (mappings) by calling **Bind<T>().To<K>()** for each dependency.



4. **app.UseNinjectMiddleWare()** 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
<pre>public AdsController(IOnlineShopData data, IUserIdProvider userIdProvider) : base(data, userIdProvider) { }</pre>	<pre>var adsController = new AdsController(new OnlineShopData(new OnlineShopContext()), new AspNetUserIdProvider());</pre>

5. 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.

