Worth 10% of ASP.NET 2

Due January 20, 4PM

Please only upload this Word document with all exercises completed in it. Please do not submit your code project.

**Testing**

Automated testing can help ensure that your project behaviour falls within expected boundaries. Including testing in extremely large projects can help eliminate vulnerabilities with automated checks for expected responses under specific conditions.

Testing is sometimes encouraged during development – this practice is called **Test Driven Development** (TDD). TDD is useful since relevant tests can be added while the developer is focused on the module. The danger of TDD is too much time can be spent writing tests on code that is eventually replaced or thrown out.

Regardless of whether or not automated testing is useful for your project, it is important to be aware of how to test so you can make a better decision as to when and how you wish to test your code.

**Attributes**

Attributes designate test classes and methods. A test class is denoted with the [TestClass] attribute. A test method is denoted with the [TestMethod] attribute.

**Assert**

Assert statements are used in testing to ensure that expected conditions are true. There are several types of conditions which can be checked. Here are some that will be used in these notes:

Assert.AreEqual(object expected, object actual);

Assert.AreNotEqual(object notExpected, object actual);

Assert.IsTrue(bool condition);

Assert.IsNull(object value);

**Unit Testing Introduction**

The defining factor of any unit test is the test must complete a single class as a stand-alone unit. At a very basic level, let’s see how this is done.

*Example 1: Unit Testing a Simple Class*

This example shows how to unit test a single class. To start, create a console application with this code inside the Program.cs file:

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace ConsoleApplication5 {  public class BankAccount {  public decimal Balance { get; private set; }  public BankAccount(decimal balance) {  Balance = balance;  }  public bool Withdraw(decimal withdrawal) {  if(Balance >= withdrawal) {  Balance -= withdrawal;  return true;  }  return false;  }  }  class Program {  static void Main() {  }  }  } |

Next, add in a Unit Test project by right clicking the *BankAccount* class and choosing *Create Unit Tests* from the drop down.



Leave all defaults when prompted.



Replace the test methods with this one:

|  |
| --- |
| [TestMethod()]  public void WithdrawTest() {  const decimal START\_BALANCE = 45;  BankAccount account = new BankAccount(START\_BALANCE);  // Ensure withdraw never occurs if > balance.  bool success = account.Withdraw(50);  Assert.AreEqual(false, success);  // Ensure withdraw occurs if <= balance.  const decimal WITHDRAWAL\_AMT = 20;  success = account.Withdraw(WITHDRAWAL\_AMT);  Assert.AreEqual(true, success);  // Check balance.  Assert.AreEqual(START\_BALANCE - WITHDRAWAL\_AMT,  account.Balance);  } |

When inside the test project, from the Test menu select Run | All Tests.



Wait a little but after starting the test and you should eventually see the Test Explorer appear. When the test is finished the test name will be listed with a green check mark if it passed.



*Exercise 1*

In the main application, create a *Deposit()* method which adds any value greater than zero to the balance. Then, in your test suite, create a separate test method which ensures:

* Any negative value passed to the *Deposit()* method does not affect the balance.
* Any positive passed to the *Deposit()* method affects the balance in an expected manner.

Show your Deposit() method here: (1 mark)

|  |
| --- |
| public bool Deposit(decimal deposit)  {  if (deposit > 0)  {  Balance += deposit;  return true;  }  return false;  } |

Show your Test method here: (2 marks)

|  |
| --- |
| [TestMethod()]  public void BankAccountTest2()  {  const decimal START\_BALANCE = 45;  BankAccount account = new BankAccount(START\_BALANCE);  // Ensure withdraw never occurs if > balance.  bool success2 = account.Deposit(0);  Assert.AreEqual(false, success2);  // Ensure withdraw occurs if <= balance.  const decimal DEPOIST\_AMT = 20;  success2 = account.Deposit(DEPOIST\_AMT);  Assert.AreEqual(true, success2);  // Check balance.  Assert.AreEqual(START\_BALANCE + DEPOIST\_AMT,  account.Balance);  } |

**Mocking**

As mentioned earlier, the goal of unit testing is to test classes and components in isolation. These components may have other dependencies though. To limit the test to the unit at hand, a mock or fake stub can be created to simulate the dependency.

*Example 2: Mocking the Context When Testing a Repository Class*

This example uses the Moq framework to simulate a *DbContext* object. The Moq framework is popular with unit testers. To build this project, create an empty MVC application with an EDMX reference to the FoodStore database. Then, add in the following repository class:

|  |
| --- |
| public class ProductRepository {  public ProductRepository() { }  // The constructor allows us to pass in the mock context.  private FoodStoreEntities context = new FoodStoreEntities();  public ProductRepository(FoodStoreEntities context) {  this.context = context;  }  public List<Product> GetAllProducts() {  return context.Products.ToList();  }  public Product GetProduct(int productID) {  return context.Products.Where(p=>p.productID == productID)  .FirstOrDefault();  }  } |

Next, add a unit test project by right clicking the ProductRepository class name in the code editor and choose **Create Unit Tests**. In the Unit Test Project, add in the Moq framework with the NuGet package manager.



Also, add the entity framework using NuGet:



Next, add the following namespace references to your test project code page:

|  |
| --- |
| using Moq;  using System.Data.Entity; |

Then, in your unit test project, replace the existing class with the following:

|  |
| --- |
| [TestClass()]  public class ProductRepositoryTests {  IQueryable<Product> GetFakeProducts() {  List<Product> products = new List<Product>();  products.Add(new Product { productID=1, name="Cake Mix", mfg="Duncan Hines",  vendor = "Sysco", price=2.99m });  products.Add(new Product { productID=2, name="Cookie Dough", mfg="Duncan Hines",  vendor = "Sysco", price=1.25m });  products.Add(new Product { productID=3, name="Orange Juice", mfg="Florida Orange",  vendor = "GFS", price=4.25m });  // Convert List to IQueryable.  return products.AsQueryable();  }  [TestMethod()]  public void ProductRepositoryTest() {  var data = GetFakeProducts();  // Set up the mock data set for the Products entity using this pattern.  var mockSet = new Mock<DbSet<Product>>();  mockSet.As<IQueryable<Product>>().Setup(m => m.Provider)  .Returns(data.Provider);  mockSet.As<IQueryable<Product>>().Setup(m => m.Expression)  .Returns(data.Expression);  mockSet.As<IQueryable<Product>>().Setup(m => m.ElementType)  .Returns(data.ElementType);  mockSet.As<IQueryable<Product>>().Setup(m => m.GetEnumerator())  .Returns(data.GetEnumerator());  // Fake context so it returns the 'Products' DbSet.  var mockContext = new Mock<FoodStoreEntities>();  mockContext.Setup(c => c.Products).Returns(mockSet.Object);  var productRepository = new ProductRepository(mockContext.Object);  var products = productRepository.GetAllProducts();  Assert.AreEqual(3, products.Count);  Assert.AreEqual("Cake Mix", products[0].name);  Assert.AreEqual("Cookie Dough", products[1].name);  Assert.AreEqual("Orange Juice", products[2].name);  }  } |

Run the tests now. All tests should pass.

*Exercise 2*

Create a test to ensure the *GetProduct()* method returns the expected value. Show your revised test class. (1 mark)

|  |
| --- |
| [TestMethod()]  public void ProductIDRepositoryTest()  {  var data = GetFakeProducts();  // Set up the mock data set for the Products entity using this pattern.  var mockSet = new Mock<DbSet<Product>>();  mockSet.As<IQueryable<Product>>().Setup(m => m.Provider)  .Returns(data.Provider);  mockSet.As<IQueryable<Product>>().Setup(m => m.Expression)  .Returns(data.Expression);  mockSet.As<IQueryable<Product>>().Setup(m => m.ElementType)  .Returns(data.ElementType);  mockSet.As<IQueryable<Product>>().Setup(m => m.GetEnumerator())  .Returns(data.GetEnumerator());  // Fake context so it returns the 'Products' DbSet.  var mockContext = new Mock<JB\_FoodStoreEntities>();  mockContext.Setup(c => c.Products).Returns(mockSet.Object);  var productIDRepository = new ProductRepository(mockContext.Object);  var product = productIDRepository.GetProduct(1);  // Assert.AreEqual(3, productsID.C;  Assert.AreEqual("Cake Mix", product.name);  Assert.AreNotEqual("Cookie Dough", product.name);  Assert.AreNotEqual("Orange Juice", product.name);  } |

*Exercise 3*

Create a separate repository class. In this class, select all Manufacturers and a single Manufacturer by name. Show your repository class here: (1 mark)

|  |
| --- |
|  |

Show your test class which uses fake data and a mock context here. The test class needs to test methods for obtaining a single Manufacturer and all manufacturers. (3 marks)

|  |
| --- |
|  |

**Testing an MVC Controller with a Test Database**

You may want to test with live data from a test database. This test is not considered a pure unit test since the test is using a dependency – however it is also really convenient and easy to test this way.

*Example 3*

To build this example, create an empty MVC project and add an EDMX that references the FoodStore database. Then, add an empty Home controller. Replace the existing ActionResult methods with the following code. Note also how the return View() instruction includes the name of the view. The default just assumes the view to display has the same name as the ActionResult method. To enable easier testing, including the name will allow the test.

|  |
| --- |
| FoodStoreEntities db = new FoodStoreEntities();  // GET: Home  public ActionResult Index()  {  throw new NotImplementedException();  }  public ActionResult Details(int Id) {  if (Id < 1)  return RedirectToAction("Index");    Product product = db.Products.Where(p=>p.productID == Id)  .FirstOrDefault();  return View("Details", product);  } |

Next, create a unit test project by right clicking your controller class name and choosing **Create Unit Tests**. In your new unit test project, add a reference to the entity framework through NuGet. Also in your new project, add a reference to the System.Web.mvc library. I don’t know a better way to do this in order to get the proper version so I recommend the following:

Right click the references node of your test project. Select the Browse button in the Reference Manager.



Navigate to the bin folder of your main project and choose the System.Web.Mvc dll to add the reference to your test project. You may be able to find a better way and if you find a way to get the dll with the proper version (5.2.x) please let me know. I did try some other ways of getting this library but experienced version differences which caused the test project to fail.



Also, you will need a reference to the System.Web dll. To do this, right click the references folder and on the Assemblies | Framework node select System.Web.



Also, create a test database on the same server as your FoodStore database. Call it TestFoodStore or whatever you want. Here is the script to build a product table in your new database:

|  |
| --- |
| CREATE TABLE Product(  productID INT PRIMARY KEY,  name VARCHAR(25),  mfg VARCHAR(25) FOREIGN KEY REFERENCES Manufacturer(mfg),  vendor VARCHAR(25) FOREIGN KEY REFERENCES Supplier(vendor),  price MONEY CHECK(price>0)  );  INSERT INTO Product VALUES(1, 'French Fries', 'McCain', 'Sysco', 1.22);  INSERT INTO Product VALUES(2, 'Apple Crumble', 'Duncan Hines', 'Sysco', 1.35);  INSERT INTO Product VALUES(3, 'Tropical Punch', 'Florida Orange', 'GFS', 6.25);  INSERT INTO Product VALUES(4, 'Apple Turnover', 'Pillsbury', 'GFS', 2.45);  INSERT INTO Product VALUES(5, 'Asparagus', 'Hot House', 'GFS', 1.22); |

Next, copy the connection string from your Web.config file in your main project and place it in the App.config file of your test project. Change the name of the database to your test database.

|  |
| --- |
| <connectionStrings>  <add name="FoodStoreEntities" connectionString="metadata=res://\*/Model1.csdl|res://\*/Model1.ssdl|res://\*/Model1.msl;provider=System.Data.SqlClient;provider connection string=&quot;data source=PMD6EA;initial catalog=TestFoodStore;integrated security=True;MultipleActiveResultSets=True;App=EntityFramework&quot;" providerName="System.Data.EntityClient" />  </connectionStrings> |

Next, replace the existing test methods with the test methods shown here:

|  |
| --- |
| [TestMethod()]  public void IndexTest() {  var controller = new HomeController();  // Must add reference to system.web.mvc  // had to browse to existing project to get it.  var result = controller.Details(1) as ViewResult;  Assert.AreEqual("Details", result.ViewName);  }  // Testing viewdata returned by a controller with result.ViewData  [TestMethod]  public void TestDetailsViewData() {  var controller = new HomeController();  var result = controller.Details(1) as ViewResult;  var product = (Product) result.ViewData.Model;  Assert.AreEqual("French Fries", product.name);  }  // Testing redirect  [TestMethod]  public void TestDetailsRedirect() {  var controller = new HomeController();  const int INVALID\_ID = -1;  var result  = (RedirectToRouteResult) controller.Details(INVALID\_ID);  Assert.AreEqual("French Fries", result.RouteValues["action"]);  } |

Remember to reference the System.Web.Mvc library:

using System.Web.Mvc;

*Exercise 4*

Try re-testing the repository class in Example 2. This time though, use the TestFoodStore database rather than a mock context. Show your revised test class here: (3 marks)

|  |
| --- |
|  |

**Testing a Web Api 2 Controller with Dependency Injection**

A popular way to isolate classes for testing is to inject mock objects to replace dependencies while testing.

*Example 4*

This example presents a more elaborate way to mock and inject a context for testing a web api controller. This example is based on a tutorial found at:

<http://www.asp.net/web-api/overview/testing-and-debugging/mocking-entity-framework-when-unit-testing-aspnet-web-api-2>

First, create an empty MVC project that has an EDMX to the FoodStore database. Then, create a Web Api controller for the Products entity.



Once the wizard finishes you will notice that it adds in the IHttpActionResult methods. Then, add this code to the start of the **Application\_Start()** method in the Global.asax file:

|  |
| --- |
| GlobalConfiguration.Configure(WebApiConfig.Register);  GlobalConfiguration.Configuration.Formatters.JsonFormatter.SerializerSettings  .ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandling.Ignore; |

To enable this code you will also need to add a reference to the System.Web.Http namespace:

|  |
| --- |
| using System.Web.Http; |

**Enabling Dependency Injection**

Right now the controller is using the default DbContext. However, you can remove this dependency with an interface. Later you can use the interface to mock the context.

In the models folder, create an IFoodStoreEntities interface with the following code:

|  |
| --- |
| using System;  using System.Data.Entity;  namespace WebApplication4.Models {  public interface IFoodStoreEntities:IDisposable {  DbSet<Product> Products { get; }  int SaveChanges();  void MarkAsModified(Product item);  }  } |

Next, open the context class for your EDMX.



Then make the following changes:

|  |
| --- |
| public partial class FoodStoreEntities : DbContext, IFoodStoreEntities  {  public FoodStoreEntities()  : base("name=FoodStoreEntities")  {  }    protected override void OnModelCreating(DbModelBuilder modelBuilder)  {  throw new UnintentionalCodeFirstException();  }  public void MarkAsModified(Product item)  {  Entry(item).State = EntityState.Modified;  } |

You will probably need the proper namespace reference to use the IFoodStoreEntities interface. Next, open the ProductController.cs file and make the following changes. These changes will allow you to either use the default DbContext or set the context with a mock object during testing:

|  |
| --- |
| public class ProductsController : ApiController  {  private IFoodStoreEntities db = new FoodStoreEntities();  public ProductsController() { }  public ProductsController(IFoodStoreEntities context)  {  db = context;  } |

Also, make the following change for PUT product:

|  |
| --- |
| // db.Entry(product).State = EntityState.Modified;  db.MarkAsModified(product);  try  {  db.SaveChanges();  } |

Build the solution and test your project to ensure you can actually retrieve data.

Then add the unit test project for your Products controller. In your test project, add in the Entity Framework packge and the Microsoft ASP.NET Web API Core package.

Next, add the following TestDbSet to your project with the following two namespace references:

|  |
| --- |
| using System.Data.Entity;  using System.Linq; |

The code in the TestDbSet creates a mock DbSet for any entity since it uses a generic template:

|  |
| --- |
| class TestDbSet<T> : DbSet<T>, IQueryable, IEnumerable<T>  where T : class  {  ObservableCollection<T> \_data;  IQueryable \_query;  public TestDbSet()  {  \_data = new ObservableCollection<T>();  \_query = \_data.AsQueryable();  }  public override T Add(T item)  {  \_data.Add(item);  return item;  }  public override T Remove(T item)  {  \_data.Remove(item);  return item;  }  public override T Attach(T item)  {  \_data.Add(item);  return item;  }  public override T Create()  {  return Activator.CreateInstance<T>();  }  public override TDerivedEntity Create<TDerivedEntity>()  {  return Activator.CreateInstance<TDerivedEntity>();  }  public override ObservableCollection<T> Local  {  get { return new ObservableCollection<T>(\_data); }  }  Type IQueryable.ElementType  {  get { return \_query.ElementType; }  }  System.Linq.Expressions.Expression IQueryable.Expression  {  get { return \_query.Expression; }  }  IQueryProvider IQueryable.Provider  {  get { return \_query.Provider; }  }  System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()  {  return \_data.GetEnumerator();  }  IEnumerator<T> IEnumerable<T>.GetEnumerator()  {  return \_data.GetEnumerator();  }  } |

After, add the following class to the test project to mock the Find() method for the Product entity. A namespace reference to your project will be needed to access the Product entity class.

|  |
| --- |
| class TestProductDbSet : TestDbSet<Product>  {  public override Product Find(params object[] keyValues)  {  return this.SingleOrDefault(product =>  product.productID == (int)keyValues.Single());  }  } |

Then, add a mock context class to your test project which references the mock Product DbSet. Additional references will be needed to access your interface and Product entity references as well as the entity framework specific references:

|  |
| --- |
| public class MockDbContext : IFoodStoreEntities  {  public MockDbContext()  {  this.Products = new TestProductDbSet();  }  public DbSet<Product> Products { get; set; }  public int SaveChanges()  {  return 0;  }  public void MarkAsModified(Product item) { }  public void Dispose() { }  } |

Next, replace the tests In your test file with the following:

|  |
| --- |
| [TestMethod()]  public void PostProduct\_ShouldReturnSameProduct()  {  var controller = new ProductsController(new MockDbContext());  var item = GetDemoProduct();  var result =  controller.PostProduct(item) as CreatedAtRouteNegotiatedContentResult<Product>;  Assert.IsNotNull(result);  Assert.AreEqual(result.RouteName, "DefaultApi");  Assert.AreEqual(result.RouteValues["id"], result.Content.productID);  Assert.AreEqual(result.Content.name, item.name);  }  [TestMethod]  public void PutProduct\_ShouldFail\_WhenDifferentID()  {  var controller = new ProductsController(new MockDbContext());  var badresult = controller.PutProduct(999, GetDemoProduct());  Assert.IsInstanceOfType(badresult, typeof(BadRequestResult));  }  [TestMethod]  public void GetProduct\_ShouldReturnProductWithSameID()  {  var context = new MockDbContext();  context.Products.Add(GetDemoProduct());  var controller = new ProductsController(context);  var result = controller.GetProduct(1) as OkNegotiatedContentResult<Product>;  Assert.IsNotNull(result);  Assert.AreEqual(1, result.Content.productID);  }  [TestMethod]  public void GetProducts\_ShouldReturnAllProducts()  {  var context = new MockDbContext();  context.Products.Add(new Product { productID=1, name="Cake Mix",  mfg="Duncan Hines", vendor="Sysco", price=2.99m });    context.Products.Add(new Product { productID=2, name="Cookie Dough",  mfg="Duncan Hines", vendor="Sysco", price=1.25m });    context.Products.Add(new Product { productID=3, name="Orange Juice",  mfg="Florida Orange", vendor="GFS", price=4.25m });  var controller = new ProductsController(context);  var result = controller.GetProducts() as TestProductDbSet;  Assert.IsNotNull(result);  Assert.AreEqual(3, result.Local.Count);  }    [TestMethod]  public void DeleteProduct\_ShouldReturnOK()  {  var context = new MockDbContext();  var item = GetDemoProduct();  context.Products.Add(item);  var controller = new ProductsController(context);  var result = controller.DeleteProduct(1) as OkNegotiatedContentResult<Product>;  Assert.IsNotNull(result);  Assert.AreEqual(item.productID, result.Content.productID);  }  Product GetDemoProduct() {  return new Product() { productID=1, name="Cake Mix",  mfg ="Duncan Hines", vendor="Sysco",  price =2.99m };  } |

You will need appropriate namespace references. Then, build and run your tests.

*Exercise 5*

In the same project used to build Example 4, add all of the code needed to the methods for getting a single manufacturer by id and all of the manufacturers.

* Show your revised IFoodStoreEntities interface here: (1 mark)

|  |
| --- |
|  |

* Show your revised Manufacturer controller here: (1 mark)

|  |
| --- |
|  |

* Show your TestManufacturerDbSet class here: (1 mark)

|  |
| --- |
|  |

* Show your revised MockDbContext class here: (1 mark)

|  |
| --- |
|  |

* Show your entire test class here which tests the get all Manufacturers and get Manufacturer by ID methods in your web api controller: (4 marks)

|  |
| --- |
|  |