# Test Automation .NET

## Diferentes Tipos de Tests

¿Entonces de qué tipos de pruebas automatizadas vamos a tratar exactamente?, Para establecer el marco sobre el cuál trabajaríamos vamos a ayudarnos de esta matriz.

Esta matriz cataloga diferentes tipos de testing, especialmente dentro de un contexto Agile, nos muestra los diferentes contextos que debemos tener en cuenta para probar adecuadamente nuestro producto.

Los tests dentro de esta matriz han sido categorizados respondiendo a 2 preguntas:

* ¿La prueba está más orientada al negocio o a la tecnología?
* ¿La prueba da soporte al trabajo del equipo o sirven para criticar las características del producto?
* *Orientadas al Negocio:* Son aquellas pruebas expresadas en un lenguaje que es perfectamente entendido por una persona de negocio. Son mejores definidas por BA, PO. Involucran la participación del cliente. Ejm: Sí se retira más dinero del que se tiene en una cuenta, se extenderá automáticamente un préstamo por la diferencia.
* *Orientadas a la Tecnología:* Expresadas en un lenguaje que es bien entendido por alguien relacionado al ámbito tecnológico. Se definen mejor por Developers Testers, DBA, etc. Ejm: La aplicación debe soportar IE7 y EI8.
* *Soporte al Equipo:* Son aquellas que definen qué es lo que el software debe hacer. Ejm: Si llamamos al método Suma con 2 y 2, debe retornar 4; Sí hacemos click en el link “Detalles de la cuenta” debemos ver los movimientos realizados.
* *Critican el Producto:* Son aquellos que tratan de identificar y evitar errores en el producto terminado. Ejm: Cuando me logeo con Brian, ve la información de Tom; Cuando hay 1000 usuarios concurrentes, el tiempo de respuesta se incrementa en 10 segundos.

**CUADRANTES**

* Q1: Ayudan al equipo en la búsqueda y prevención de defectos (Internal Quality) dentro del producto que se está trabajando. Se utilizan apara asegurar que el código escrito hace lo que se espera que haga.
* Q2: Guían el desarrollo realizado por el equipo a través de prueba orientadas al negocio. Estas pruebas permiten comprender y descubrir mejor el negocio las necesidades del cliente. (External Quality / Done Definition)
  + Pruebas de Aceptación: Escenarios o ejemplos expresados en tests ejecutables.
  + Prototipos / Diagramas / Mapas mentales.
* Q3: Evaluan el producto recreando la experiencia actual del usuario. Realizando pruebas realistas del sistema.
  + Demos:
    - Iteration Reviews (formal)
    - Exploratoring Testing con el cliente (informal)
  + Usability Testing.
  + User Acceptance Tests (UAT): Beta teststing.
* Q4: Evaluan el producto, principalmente en términos de atributos de calidad que son particularmente necesarios de cumplir para el software en cuestión.
  + Scalabilty / Compatibility / Security
  + Databa Migration / Memory Managment / Recovery

# Unit Testing

## Analogía del Automóvil

Para entender mejor a que nos referimos cuando hablamos de unit testing. En vez de hablar por el momento de software, vamos a imaginarnos la construcción de un carro.

Un carro tiene un motor, una transmisión, un sistema de frenos, refrigeración, cinturones de seguridad, faros, etc. Un automóvil es la suma de todas esas partes o subsistemas. Cada una de estas partes tienes sus propias especificaciones y requisitos que definen la funcionalidad esperada, tolerancias e interfaces con el resto del sistema.

Imaginen el punto en el cuál todos esos subsistemas son ensamblados y atornillados en una línea de montaje sin que se realice ninguna prueba previa. El conductor que va a realizar la prueba del auto se sienta y este no arranca. ¿ De que subsistema fue la culpa? Si no se realizaron pruebas independientes a cada uno de esos subsistemas y sus partes será muy difícil encontrar el problema.

Suponiendo que se pone en marcha y avanza ¿el piloto podrá probar cada una de las partes del auto? Que el sistema de refrigeración funcione a altas revoluciones o que el motor funcione si el aceite está más abajo que los niveles permisibles. Para realizar todas estas pruebas el piloto se tomará una gran cantidad de tiempo y también serían mejor realizadas por las personas especializadas que construyeron el subsistema y de una manera que el usuario final sería incapaz de hacerlo.

Lo mismo ocurre con el software, de manera similar a los autos, el software es una colección de componentes y partes. Realizar pruebas independientes a cada una de estas partes nos ayudará a depurar el API antes de que alguien intente usarla. Si a través de alguna de estas pruebas falla será muchísimo más fácil de encontrar y corregir el error en comparación a si estuviéramos probando todo el sistema, asimismo se permite ahorrar mucho más tiempo y dinero ya que es más eficiente para cualquier corregir errores antes de entrar a una fase de ensamblaje y detener a todo el equipo ante cada fallo.

Piensen en esto, ustedes preferirían conducir un automóvil cuyos frenos han sido alta y consecutivamente probados por los ingenieros especializados o confiarían en las pruebas de sistema que ha realizado el piloto de pruebas. La misma lógica debemos seguir al desarrollar un software, las pruebas no es algo que se deba realizar luego de que todas las partes han sido combinadas sino se debe realizar de manera independiente a cada parte antes de ensamblarlas en un sistema mucho más grande. Realizar estas pruebas desde el inicio y ejecutarlas constantemente cada vez que se realice algún cambio nos ayudará a a garantizar la calidad del software que las usar y a identificar problemas antes de llegar a QA o a los usuarios finales.

# Test Doubles

## Inversión de Dependencia

public interface IDataAccess  
{  
    decimal GetShippingCosts(Order order);  
  
    void SaveOrder(Order order);  
  
    Order GetOrder(int id);  
}

private IDataAccess dataAccess;  
public OrderServices()  
{  
    this.dataAccess = new DataAccess();  
}

## Inyección de Dependencias

[TestClass]  
public class OrderServicesTests  
{  
    [TestMethod]  
    public void CalculateTotal\_OrderFromUS\_FreeShipping()  
    {  
 Order order = new Order { Country = "US", ItemTotal = 100 };  
        OrderServices orderServices = new OrderServices(new DataAccess());  
  
        var total = orderServices.CalculateTotal(order);  
  
        Assert.AreEqual(100, total);  
    }  
  
    [TestMethod]  
    public void CalculateTotal\_OrderOutsideUS\_ShippingCostIsAdded()  
    {  
 Order order = new Order { Country = "PER", ItemTotal = 100 };  
        OrderServices orderServices = new OrderServices(new DataAccess());  
  
        var total = orderServices.CalculateTotal(order);  
  
        Assert.AreEqual(110, total);  
    }  
  
    [TestMethod]  
    public void Save\_ValidOrder\_TheOrderIsPersisted()  
    {  
 Order order = new Order { Id = 1, Country = "PER", ItemTotal = 100, Total = 110 };  
        OrderServices orderProcessor = new OrderServices(new DataAccess());  
  
        orderProcessor.Save(order);  
  
        Order orderFromDb = orderProcessor.GetOrder(order.Id);  
        Assert.IsNotNull(orderFromDb);  
    }  
}

## Test Doubles Manuales

    [TestClass]  
    public class OrderServicesTests\_ManualTestDoubles  
    {  
        [TestMethod]  
        public void CalculateTotal\_OrderFromUS\_FreeShipping()  
        {  
            OrderServices orderServices = new OrderServices(null);  
            Order order = new Order { Country = "US", ItemTotal = 100 };  
   
            var total = orderServices.CalculateTotal(order);  
   
            Assert.AreEqual(100, total);  
        }  
   
        [TestMethod]  
        public void CalculateTotal\_OrderOutsideUS\_ShippingCostIsAdded()  
        {  
            Order order = new Order { Country = "PER", ItemTotal = 100 };  
            var dataAccess = new SimpleDataAccess();  
            dataAccess.ShippingCosts = 10;  
            OrderServices orderServices = new OrderServices(dataAccess);  
   
            var total = orderServices.CalculateTotal(order);  
   
            Assert.AreEqual(110, total);  
        }  
   
        [TestMethod]  
        public void Save\_ValidOrder\_TheOrderIsPersisted()  
        {  
            Order order = new Order { Id = 1, Country = "PER", ItemTotal = 100, Total = 110 };  
            var dataAccess = new SimpleDataAccess();  
            OrderServices orderProcessor = new OrderServices(dataAccess);  
   
            orderProcessor.Save(order);  
   
            Assert.AreEqual(order,dataAccess.OrderSaved);  
        }  
   
        private class SimpleDataAccess : IDataAccess  
        {  
            public decimal ShippingCosts;  
            public decimal GetShippingCosts(Order order)  
            {  
                return this.ShippingCosts;  
            }  
   
            public Order GetOrder(int id)  
            {  
                throw new **NotImplementedException**();  
            }  
   
            public Order OrderSaved;  
            public void SaveOrder(Order order)  
            {  
                OrderSaved = order;  
            }  
        }  
    }

## Test Double Framework

[TestClass]  
public class OrderServicesTests  
{  
    [TestMethod]  
    public void CalculateTotal\_OrderFromUS\_FreeShipping()  
    {  
        Order order = new Order { Country = "US", ItemTotal = 100 };  
        OrderServices orderServices = new OrderServices(null);  
  
        var total = orderServices.CalculateTotal(order);  
  
        Assert.AreEqual(100, total);  
    }  
  
    [TestMethod]  
    public void CalculateTotal\_OrderOutsideUS\_ShippingCostIsAdded()  
    {  
        Order order = new Order { Country = "PER", ItemTotal = 100 };  
        var dataAccess = new Mock<IDataAccess>();  
        dataAccess.Setup(x => x.GetShippingCosts(order)).Returns(10);  
        OrderServices orderServices = new OrderServices(dataAccess.Object);  
  
        var total = orderServices.CalculateTotal(order);  
  
        Assert.AreEqual(110, total);  
    }  
  
    [TestMethod]  
    public void Save\_ValidOrder\_TheOrderIsPersisted()  
    {  
        Order order = new Order { Id = 1, Country = "PER", ItemTotal = 100, Total = 110 };  
        var dataAccess = new Mock<IDataAccess>();  
        OrderServices orderProcessor = new OrderServices(dataAccess.Object);  
  
        orderProcessor.Save(order);  
  
        dataAccess.Verify(x => x.SaveOrder(order));  
    }  
}

## Mocking Frameworks

Si se encuentra antes o igual en el índice de la lista, es válido

    [TestClass]  
    public class LogManagerTests  
    {  
        private Mock<IConfiguration> configuration;  
        private Mock<IEmailSender> emailSender;  
        private Mock<IAppender> appender;  
        private LogManager logManager;  
   
        [TestInitialize]  
        public void Setup()  
        {  
            this.configuration = new Mock<IConfiguration>();  
            this.emailSender = new Mock<IEmailSender>();  
            this.appender = new Mock<IAppender>();  
            this.logManager = new LogManager(configuration.Object, emailSender.Object, appender.Object);  
        }  
   
        [TestMethod]  
        public void IsEnabled\_MessageLevelBeforeLoggerLevel\_ReturnTrue()  
        {  
            configuration.Setup(x => x.LoggerLevel()).Returns(Level.Info);  
   
            bool isEnabled = this.logManager.IsEnabled(Level.Error);  
   
            Assert.IsTrue(isEnabled);  
        }  
   
        [TestMethod]  
        public void IsEnabled\_MessageLevelAfterLoggerLevel\_ReturnFalse()  
        {  
            configuration.Setup(x => x.LoggerLevel()).Returns(Level.Info);              
   
            bool isEnabled = logManager.IsEnabled(Level.Debug);  
   
            Assert.IsFalse(isEnabled);  
        }  
   
        [TestMethod]  
        public void Write\_LevelError\_SendMailtoAdmin()  
        {  
            logManager.Write("message", Level.Error);  
   
            emailSender.Verify(x => x.SendToAdmin("message"));  
        }  
   
        [TestMethod]  
        public void Write\_IsEnabled\_WriteToAppender()  
        {  
            configuration.Setup(x => x.LoggerLevel()).Returns(Level.Info);  
   
            logManager.Write("message", Level.Info);  
   
            appender.Verify(x => x.Write("message"));  
        }  
    }

# Database Testing

## Outside DB Testing

### ADO.NET DBUNIT: “External Data Source” and “Nuke and Pave”

Observamos que las pruebas de filtros realizan sus asertos esperando que se encuentren determinada información en la BD.

Podríamos asumir que la información correcta ya se va a encontrar en la bd al momento de realizar la prueba, pero esto trae 2 problemas:

* es muy muy probable que en ese momento la información no se encuentre en la bd o no sea la correcta como para ejecutar la prueba de manera repetitiva al menos que cada vez que estés ejecutando la prueba revises la BD e ingreses los datos manualmente, lo que haría que la prueba no se para nada repetible.
* si en caso esta información se encontrase en la BD, el test es poco legible ya que no se puede determinar a simple vista cuál es el contexto sobre el cuál se está creando la prueba.

Por estos motivos las propias pruebas deben ser capaces de inicializar sus datos. En esta oportunidad utilizaremos DBUnit como una herramienta q nos va a permitir cargar datos desde archivos XML.

* Agregamos la referencia a NDBUnit.SqlClient
* De acuerdo a los nombres de los archivos podemos observar que estamos creando un datasource por test fixture. Entonces como estos archivo serán utilizados por todo el tests fixture, vamos a configurarlos una sola vez en todo el test fixture.

private static SqlDbUnitTest database;  
  
[ClassInitialize]  
public static void FixtureSetup(TestContext context)  
{  
    database = new SqlDbUnitTest(ConfigurationManager.ConnectionStrings["DB"].ConnectionString);  
    database.ReadXmlSchema(@"EmployeeADODatabase.xsd");  
    database.ReadXml(@"EmployeeADOTestData.xml");  
}

* DbUnit necesita 2 archivos para su funcionamiento:
  + Un archivo XML con información de la tabla(s) de BD a las cuales vamos a insertar datos. No es necesario que estos archivos contengan toda la información de las tablas y columnas, solo lo necesario para poder insertar los datos.

Este archivo no es otra cosa que el famoso dataset que se utiliza para realizar operaciones en la BD en entorno conectado, solo que ahora se necesita únicamente para metadata de la BD. La forma más rápida de crear este archivo es crear un nuevo dataset.

* + El segundo archivo, es un xml con los datos q vamos a insertar en la BD.
* Ahora necesitamos indicar a dbunit que necesitamos insertar la información contenida en la los archivos, pero esta información la necesitamos insertar antes que se realiza cada prueba, ya que la prueba anterior ha podido alterar o eliminar la información que hemos insertado.

[TestInitialize]  
public void Setup()  
{  
    this.employeeADO = new EmployeeADO();  
    database.PerformDbOperation(DbOperationFlag.Insert);  
}

* Ahora que ya lo tenemos configurado, vamos a ejecutar únicamente la primera prueba, vemos que la prueba falla xq no se están encontrando los archivo, seguramente ya hemos observado que cada vez que se ejecuta una prueba con MSTest se crea una carpeta TestsResults que es donde se copian las entradas y las salidas de las pruebas. Por lo tanto necesitamos decirle a MStests que copie esos archivos a esa carpeta.
  + Copy Always en las propiedades del archivo.
  + Agregar los atributos DeploymentItem

    [DeploymentItem("EmployeeADODatabase.xsd")]  
    [DeploymentItem("EmployeeADOTestData.xml")]

* Abrimos el profiler para observar cuales son los queries generados.
* Primero ejecutamos únicamente la primera prueba y vemos que pasa sin problemas.
* Luego ejecutamos todos los tests y observamos que varios fallan, si vemos cuales son los datos de la BD, vamos a observar que tenemos más datos de los esperados. El problema es que necesitamos restablecer el estado de la BD para que la siguiente prueba se pueda ejecutar de manera consistente.

En esta oportunidad para restablecer el estado de la BD vamos a utilizar la técnica de Nuke and Pave, que es una de las técnicas más fáciles de implementar ya que elimina absolutamente todos los datos de las tablas involucradas sin importar si estos no son manejados por el tests.

Una delas características de DBUnit es que permite aplicar esta estratégia fácilmente. Para esto solo tenemos que utilizar el flag DbOperationFlag.CleanInsert, que hará que se realice un delete de toda la tabla antes de insertar los datos, observamos esto en el profiler.

* Aún el Delete no pasa, el problema es que está intentando eliminar un registro con un ID fijo, pero debido a que nuestras columnas son identity, este campo siempre es autoincrementando. Para evitar estos problemas podemos Utilizar el DbOperationFlag.CleanInsertIdentity que desabilitará esas columnas y permitirá insertar un valor fijo en la columna Id que podamos referenciar desde el test.

  <Employee>   
    <Id>1</Id>  
    <LastName>Pacheco</LastName>  
    <HireDate>2010-12-30</HireDate>  
  </Employee>

* Luego de finalizadas todas las pruebas y que estas pasan correctamente, podemos observar que aún permanecen algunos registros en la BD, ya que estamos restableciendo el estado antes de comenzar la siguiente prueba y no al terminar la anterior. Esto no necesariamente es malo ya que estamos asegurando que cada prueba se asegura de tener la BD en el estado adecuado.

En esta oportunidad vamos a eliminarlos para poder comenzar el siguiente ejercicio desde 0.

* Para eliminarlos podemos escribir una rutina que eliminé todos os registros y que únicamente se ejecute al finalizar todos los tests.

[ClassCleanup]  
public static void CleanUpDatabae()  
{  
    database.PerformDbOperation(DbOperationFlag.DeleteAll);  
}

### Entity Framework: “External DataSource” and “Transaction”

En esta oportunidad vamos a realizar nuestras pruebas a métodos de acceso implementado con Entity Framework.

*¿Quienes no conocen nada de EF Code First, o nisiquiera han hecho una aplicación de ejemplo? Explicar brevemente EF en caso sea necesario(modelo,mappings,tablas).*

Examinemos los métodos de acceso a datos.

* Ahora existen 3 clases: AppDbContext,Employee, EmployeeEF
* AppDbContext es la clase más importante para el funcionamiento y la configuración de EF. *(Explicar un poco de EF en caso sea necesario)*
* *Employee* que es nuestra entidad.
* *EmployeeEF* que es nuestra clase de acceso a datos. *Examinamos que hace esta clase.*
* *EmployeeEFTests* contiene las mismas pruebas que vimos anteriormente.

Vamos a seguir realizando Nuke and Pave pero ya no vamos a utilizar NBUnit sino la característica innata que tienen todos los ORMs como entity framework de poder generar automáticamente la BD. Si recordamos el Nuke and Pave elimina es borrar todo y volver a comenzar, en la anterior oportunidad eliminábamos todos los datos y los volvíamos a insertar, pero ahora vamos a ver como funciona el Nuke and Pave a otro nivel, eliminando también las tablas, prácticamente vamos a destruir toda la BD y vamos a volverla a crear.

¿Cómo funciona esto? Apenas abrimos una conexión con EF, este verifica si la BD de nuestra cadena de conexión existe, sino existe, crea toda la BD incluyendo sus tablas, pero además nos ofrece la posibilidad de poder insertar datos, estos datos se deben encontrar en otra clase y en forma de objetos.

Incluimos la clase EmployeeEFTestData dentro del proyecto.

public class EmployeeEFInitializer : DropCreateDatabaseAlways<AppDbContext>  
{  
    protected override void Seed(AppDbContext context)  
    {  
        Employee employee1 = new Employee("Luis", "Pacheco", new DateTime(2010, 12, 30));  
        Employee employee2 = new Employee("Luis", "Quispe", new DateTime(2010, 12, 29));  
        Employee employee3 = new Employee("Luis", "Tovar", new DateTime(2010, 12, 28));  
  
        context.Employees.Add(employee1);  
        context.Employees.Add(employee2);  
        context.Employees.Add(employee3);  
    }  
}

EF tiene el concepto de initializers que son clases que le indican a EF como es que se va a recrear toda la BD. Para el caso de las pruebas el initializers más adecuado es el DropCreateAlways, ya que nosotros queremos generar toda la BD siempre que ejecutemos el test.

En la anterior oportunidad creábamos y destruíamos todos los datos en cada test, ahora lo que vamos a hacer es crear toda la BD pero únicamente al inicio de los tests y vamos a utilizar la estrategia de *Transaction-Rollback* para mantener la consistencia entre test y test.

[ClassInitialize]  
public static void InitializeDatabase(TestContext context)  
{  
    Database.SetInitializer(new EmployeeEFInitializer());  
    using (var dbcontext = new AppDbContext())  
    {  
        dbcontext.Database.Initialize(force: true);  
    }  
}

Vamos a ejecutar únicamente la primera prueba, como esta prueba no altera la información de la BD, no nos vamos a preocupar por el momento por como vamos a mantener la consistencia entre los tests.

Ejecutamos todas las pruebas de consulta y abrimos el profiler. Observamos que se está abriendo una transacción y ningún commit.

Ahora vamos a descomentar las otras pruebas, lo primero que tenemos que hacer para poder ejecutar estas pruebas es ver como vamos a mantener la BD en su estado conocido, para esto la estrategia Transaction-Rollback que como dijimos consiste en encerrar el test en una transacción y hacer rollback de esta transacción al finalizar el test de tal manera que cualquier cambio que haya realizado el test va a ser revertido por el rollback.

private TransactionScope transactionScope;

[TestInitialize]  
public void Setup()  
{  
    this.context = new AppDbContext();  
    this.employeeEF = new EmployeeEF(context);  
    transactionScope = new TransactionScope();  
}  
  
[TestCleanup]  
public void TearDown()  
{  
    transactionScope.Dispose();  
}

Al no realizar ningún commmit, el dispose de la transacción realizará un rollback automático. Cabe destacar que el transactionScope no necesariamente involucra una transacción distribuida, sino este se mantiene lightweigh mientras no se abran conexiones diferentes o conexiones embebidas dentro del ámbito de la transacción.

Ahora necesitamos resolver un siguiente problema, cualquier alteración de información que realicemos a través de un ORM, primero se realiza en memoria dentro del ámbito del ORM y necesitamos indicarle explícitamente al ORM que todos esos cambios se transformen en sentencias SQL. Por lo tanto cuando nosotros realicemos una llamada al Delete o Create no significa que se han producido los cambios en la BD.

public void Create\_TheEmployeeIsPersisted()  
{  
    Employee employee = new Employee("Luis", "Carranza", new DateTime(2010, 12, 15));  
  
    employeeEF.Create(employee);  
    FlushChanges();  
  
    Employee employeePersisted = employeeEF.Get(employee.Id);  
    Assert.IsNotNull(employeePersisted);  
}  
  
[TestMethod]  
public void Delete\_TheEmployeeIsDeleted()  
{  
    employeeEF.Delete(1);  
    FlushChanges();  
  
    Employee employeeDeleted = employeeEF.Get(1);  
    Assert.IsNull(employeeDeleted);  
}  
  
private void FlushChanges()  
{  
    context.SaveChanges();  
}

Seguramente nos habremos dado cuenta de la gran cantidad de tiempo que toman los enfoques Nuke and Pave, especialmente este que reconstruyen absolutamente toda la BD. Una característica de EF que puede ayudar a aplicar este enfoque reduciendo un poco el tiempo de ejecución de la prueba, es que podemos crear nuestros propios initializers. El initializers que estamos utilizando por defecto elimina y crea toda la BD, pero podemos crear un initializers que lo único que haga alterar los datos de las tablas sin eliminar estas tablas.

public class EmployeeEFInitializer : IDatabaseInitializer<AppDbContext>  
{  
    public void InitializeDatabase(AppDbContext context)  
    {  
        DeleteData<Employee>(context);  
        ResetIdentitySeed(context);  
        LoadData(context);  
    }  
  
    private void LoadData(AppDbContext context)  
    {  
        Employee employee1 = new Employee("Luis", "Pacheco", new DateTime(2010, 12, 30)) { Id = 1 };  
        Employee employee2 = new Employee("Luis", "Quispe", new DateTime(2010, 12, 29)) { Id = 2 };  
        Employee employee3 = new Employee("Luis", "Tovar", new DateTime(2010, 12, 28)) { Id = 3 };  
  
        context.Employees.Add(employee1);  
        context.Employees.Add(employee2);  
        context.Employees.Add(employee3);  
  
        context.SaveChanges();  
    }  
  
    private void DeleteData<T>(AppDbContext context) where T : class  
    {  
        var tableName = TableName<T>(context);  
        context.Database.ExecuteSqlCommand("Delete from " + tableName);  
    }  
  
    private static void ResetIdentitySeed(AppDbContext context)  
    {  
        var tableName = TableName<Employee>(context);  
        context.Database.ExecuteSqlCommand("DBCC CHECKIDENT ('" + tableName + "', RESEED, 0)");  
    }  
  
    private static string TableName<T>(AppDbContext context) where T : class  
    {  
        return typeof(T).Name;  
        // Sí se utiliza el pluralize convention  
        /\* var objectContext = ((IObjectContextAdapter)context).ObjectContext;  
           return objectContext.CreateObjectSet<T>().EntitySet.Name;  
        \*/  
    }  
}

Dense cuenta que podemos utilizar un enfoque similar utilizando DBUnit y Transactions.

### Entity Framework: “Self-Contained Tests” and “Transactions.”

Ahora vamos a ver el último enfoque “Pruebas autosuficientes” para inicializar el estado de la BD y vamos a utilizarlo junto con “Transacion-Rollback”.

Si recordamos el enfoque de pruebas autosuficientes hace que cada prueba se encargue de inicializar sus datos y únicamente aquellos datos q necesita.

Nos dirigimos a la primera prueba y dentro de esta creamos los datos que necesita, agregamos estos datos al contexto y luego hacemos flush a los datos.

public void Find\_WithLastNameFilter\_ReturnTheEmployeesWithTheExactLastName()  
{  
    Employee employee1 = new Employee("Luis", "Pacheco", new DateTime(2010, 12, 30));  
    LoadData(employee1);  
    Employee employee2 = new Employee("Luis", "Quispe", new DateTime(2010, 12, 29));  
    LoadData(employee2);  
    FlushChanges();  
  
    string lastName = "Pacheco";  
  
    IList<Employee> employees = employeeEF.Find(lastName, null, null);  
  
    Assert.AreEqual(1, employees.Count);  
}

private void LoadData(Employee entity)  
{  
    context.Employees.Add(entity);  
}

Ejecutamos únicamente esta prueba pero eliminamos los datos de la BD que quedaron del ejercicio anterior.

Ya no completaremos el resto de las pruebas de los filtros ya que son similares. La prueba del créate queda de la misma manera ya que no necesita datos previos para funcionar pero el delete si necesita por lo menos un dato en la BD.

[TestMethod]  
public void Delete\_TheEmployeeIsDeleted()  
{  
    Employee employee = new Employee("Luis", "Pacheco", new DateTime(2010, 12, 30));  
    LoadData(employee);  
    FlushChanges();  
  
    employeeEF.Delete(employee.Id);  
    FlushChanges();  
  
    Employee employeeDeleted = employeeEF.Get(1);  
    Assert.IsNull(employeeDeleted);  
}

Como se darán cuenta, este es el enfoque que menos tiempo toma y es el recomendado.

## Inside DB Testing

Podemos observar que en la BD tenemos 2 procedures con una lógica muy similar a la que vimos en los ejercicios anteriores. El objetivo de este ejercicio es realizar pruebas para ambos de estos procedures.

tSQLt se instala a nivel de BD por eso necesitamos ejecutar los scripts que vienen en el directorio que descargado desde internet.

* Debido a que usa cierta funcionalidad el CLR por eso necesitamos habilitarlo. Ejecutar el script SetClrEnabled.sql
* Ejecutar el script tSQLt.class.sql. este creará números objetos dentro del esquema tsqlt(procedures, tablas, ensamblados) dentro de la BD.

Ahora que tenemos los scripts instalados. Vamos a comenzar creando un Test Fixture. De manera similar a nuestras pruebas anteriores vamos a utilizar un TestFixture por Store Procedure.

Creamos el TestFixture.

EXEC tSQLt.NewTestClass 'EmployeeFind';

GO

Creamos el Primer Test.

Debemos seguir la siguiente convención:

* El nombre completo del test o procedure esta compuesto por [Esquema].[Nombre Procedure]: el Esquema debe tener el nombre del test fixture al cuál pertenece, el Nombre del Procedure debe comenzar con la palabra “Test”.

Entonces creamos el primer Test y le agregamos un aserto para que este falle. Gran parte de las utilidades que nos ofrece tsqlt está en forma de store procedures es por eso que constamente utilizamores el **EXEC** para llamar a estos procedures.

Podemos observar que dentro de los procedures creados dentro del esquema tsqlt hay otros procedures que nos permiten escribir asertos.

CREATE PROCEDURE EmployeeFind.[Test WithLastNameFilter ReturnTheEmployeesWithTheExactLastName] AS

BEGIN

EXEC tSQLt.Fail 'Test Incomplete'

END

GO

Para ejecutar todos los Test

EXEC tSQLt.RunAll

Podemos observar que dentro de los procedures creados dentro del esquema tsqlt hay otros procedures que nos permiten ejecutar los tests, a nivel de fixture, por nombre, etc.

Una característica muy importante de tsqlt, en general de las herramientas que trabajan de manera interna, es que deben se crean utilizando el patrón del “self-contained tests”.

CREATE PROCEDURE EmployeeFind.[Test WithLastNameFilter ReturnTheEmployeesWithTheExactLastName]

AS

BEGIN

-- ARRANGE

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Perez','2010-12-30')

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Garcia','2010-12-29')

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Tovar','2010-12-28')

END

GO

Ejecutamos el procedure y contamos el número de filas devueltas y escribimos el assert.

CREATE PROCEDURE EmployeeFind.[Test WithLastNameFilter ReturnTheEmployeesWithTheExactLastName]

AS

BEGIN

-- ARRANGE

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Perez','2010-12-30')

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Garcia','2010-12-29')

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Tovar','2010-12-28')

-- ACT

EXEC EmployeeFind 'Perez',null,null

DECLARE @RowCount int = @@ROWCOUNT;

-- ASSERT

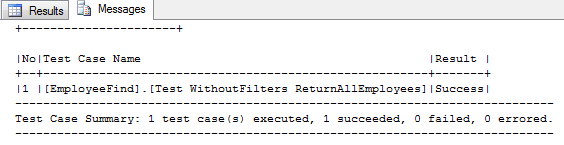
EXEC tSQLt.assertEquals 1, @RowCount;

END

GO

Lo que nos falta ahora, es como mantener en un estado conocido la BD luego del tests. Una característica muy importante de tsqlt es que ya implementa internamente el patrón “transaction-rollback”. Con lo cuál todos nuestros tests se ejecutan dentro de una transacción que es revertida al finalizar los mismos.

Si ejecutamos los tests veremos que estos pasan.



Pero también observamos que hay un comportamiento extraño, ya que la ventana de resultados muestra un result set y a la vez la salida del output. Esto se debe a que nuestro procedure devuelve un resultset y el ssms lo muestra automáticamente.

Esto no es tan malo, pero tenemos un problema mayor, que pasaría si nosotros queremos verificar que efectivamene el result set devuelto es la fila que contiene al empleado Perez. Entonces si queremos explorar el result set devuelto para luego examinarlo, tenemos que insertarlo en una tabla, para esto podemos utilizar una tabla temporal.

CREATE PROCEDURE EmployeeFind.[Test WithLastNameFilter ReturnTheEmployeesWithTheExactLastName]

AS

BEGIN

-- ARRANGE

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Perez','2010-12-30')

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Garcia','2010-12-29')

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Tovar','2010-12-28')

CREATE TABLE EmployeeFind.Actual (

Id int,

FirstName nvarchar(max),

LastName nvarchar(max),

HireDate DateTime

);

-- ACT

INSERT INTO EmployeeFind.Actual

EXEC EmployeeFind 'Perez',null,null

DECLARE @RowCount int = @@ROWCOUNT;

-- ASSERT

EXEC tSQLt.assertEquals 1, @RowCount;

END

GO

Ahora necesitamos verificar que el contenido de esta tabla tiene al empleado “Perez”. Tsqlt tiene para esto un aserto que nos permite comparar que el contenido de 2 tablas sea exactamente igual, el asserto falla si es que por lo menos el valor de alguna celda es diferente. Este aserto recibo como parámetro el nombre de las tablas.

CREATE PROCEDURE EmployeeFind.[Test WithLastNameFilter ReturnTheEmployeesWithTheExactLastName]

AS

BEGIN

-- ARRANGE

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Perez','2010-12-30')

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Garcia','2010-12-29')

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Tovar','2010-12-28')

CREATE TABLE EmployeeFind.Actual (

Id int,

FirstName nvarchar(max),

LastName nvarchar(max),

HireDate DateTime

);

-- ACT

INSERT INTO EmployeeFind.Actual

EXEC EmployeeFind 'Perez',null,null

DECLARE @RowCount int = @@ROWCOUNT;

-- ASSERT

EXEC tSQLt.assertEquals 1, @RowCount;

EXEC tSQLt.AssertEqualsTable 'EmployeeFind.Expected', 'EmployeeFind.Actual';

END

GO

Ahora necesitamos crear la tabla #expected e insertar los datos que esperamos que se encuentre en la tabla #actual para que estos sean comparados.

CREATE PROCEDURE EmployeeFind.[Test WithLastNameFilter ReturnTheEmployeesWithTheExactLastName]

AS

BEGIN

-- ARRANGE

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Perez','2010-12-30')

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Garcia','2010-12-29')

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Tovar','2010-12-28')

CREATE TABLE EmployeeFind.Expected (

Id int,

FirstName nvarchar(max),

LastName nvarchar(max),

HireDate DateTime

);

CREATE TABLE EmployeeFind.Actual (

Id int,

FirstName nvarchar(max),

LastName nvarchar(max),

HireDate DateTime

);

INSERT INTO EmployeeFind.Expected(LastName,HireDate) VALUES('Perez','2010-12-30')

-- ACT

INSERT INTO EmployeeFind.Actual

EXEC EmployeeFind 'Perez',null,null

DECLARE @RowCount int = @@ROWCOUNT;

-- ASSERT

EXEC tSQLt.assertEquals 1, @RowCount;

EXEC tSQLt.AssertEqualsTable 'EmployeeFind.Expected', 'EmployeeFind.Actual';

END

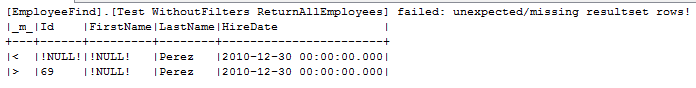
GO

Como ahora ambas tablas van a ser comparadas por completo no es necesario realizar otro aserto sobre el número de filas. Por lo tanto este código ya no es necesario.

DECLARE @RowCount int = @@ROWCOUNT;

EXEC tSQLt.assertEquals 1, @RowCount;

Si ejecutamos el tests observaremos que ha fallado, debido a que nuestra tablas #actual contiene los valores identity que se encuentran en la tabla Employee y son devueltos por el Procedures, asimismo no existe forma de predecir consistemente cuál serál el valor de la columna identity al momento de ejecutar la prueba.



Dentro de nuestros tests necesitamos concentrarnos únicamente en los datos que son relevantes para el test, pero las constraints y los campos autogenerados hacen esto muy difícil. Tsqlt nos ofrece una funcionalidad para crear una nueva tabla exactamente igual pero sin ningún tipo dee constraints. La tabla original será restaurada al final del tests debido al rollback.

CREATE PROCEDURE EmployeeFind.[Test WithLastNameFilter ReturnTheEmployeesWithTheExactLastName]

AS

BEGIN

-- ARRANGE

EXEC tSQLt.FakeTable 'Employee'

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Perez','2010-12-30')

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Garcia','2010-12-29')

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Tovar','2010-12-28')

CREATE TABLE EmployeeFind.Expected (

Id int,

FirstName nvarchar(max),

LastName nvarchar(max),

HireDate DateTime

);

CREATE TABLE EmployeeFind.Actual (

Id int,

FirstName nvarchar(max),

LastName nvarchar(max),

HireDate DateTime

);

INSERT INTO EmployeeFind.Expected(LastName,HireDate) VALUES('Perez','2010-12-30')

-- ACT

INSERT INTO EmployeeFind.Actual

EXEC EmployeeFind 'Perez',null,null

DECLARE @RowCount int = @@ROWCOUNT;

-- ASSERT

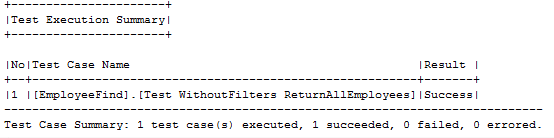
EXEC tSQLt.assertEquals 1, @RowCount;

EXEC tSQLt.AssertEqualsTable 'EmployeeFind.Expected', 'EmployeeFind.Actual';

END

GO

Luego de esto podremos observar que el tests ya está pasando.



Lo que acabamos de hacer es uno otro de los patrones más comunes al crear pruebas de BD “Compare Table Assert” . En el cuál construimos 2 conjuntos de datos(en este caso tablas): actual y esperado, para compararlos entre así dentro a través de un aserto.

Asimismo hemos podido observar la gran cantidad de código que se necesita para crear una prueba y lo poco mantenibles que són ya que cualquier cambio en el procedure o en alguna de las tablas originales, ocasionará que tengamos que modificar varios lugares en varias pruebas.

Para solucionar lo primero, tsqlt también nos ofrece procedimientos Setup que podemos agregar al Test Fixture para que sean ejecutados siempre antes de las pruebas. Vamos a refactorizar la prueba para incluir el Setup. La convención para crear un setup es que el nombre del procedure debe ser [Nombre Test Fixture].[Setup]

CREATE PROCEDURE EmployeeFind.SetUP

AS

BEGIN

EXEC tSQLt.FakeTable 'Employee'

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Perez','2010-12-30')

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Garcia','2010-12-29')

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Tovar','2010-12-28')

CREATE TABLE EmployeeFind.Expected (

Id int,

FirstName nvarchar(max),

LastName nvarchar(max),

HireDate DateTime

);

CREATE TABLE EmployeeFind.Actual (

Id int,

FirstName nvarchar(max),

LastName nvarchar(max),

HireDate DateTime

);

END

GO

CREATE PROCEDURE EmployeeFind.[Test WithLastNameFilter ReturnTheEmployeesWithTheExactLastName]

AS

BEGIN

-- ARRANGE

INSERT INTO EmployeeFind.Expected(LastName,HireDate) VALUES('Perez','2010-12-30')

-- ACT

INSERT INTO EmployeeFind.Actual

EXEC EmployeeFind 'Perez',null,null

DECLARE @RowCount int = @@ROWCOUNT;

-- ASSERT

EXEC tSQLt.assertEquals 1, @RowCount;

EXEC tSQLt.AssertEqualsTable 'EmployeeFind.Expected', 'EmployeeFind.Actual';

END

GO

Podemos crear otro test y verificar que este pase.

CREATE PROCEDURE EmployeeFind.[Test WithHireDateFilters ReturnTheEmployeesBetweenHireDates]

AS

BEGIN

-- ARRANGE

INSERT INTO EmployeeFind.Expected(LastName,HireDate) VALUES('Garcia','2010-12-29')

INSERT INTO EmployeeFind.Expected(LastName,HireDate) VALUES('Perez','2010-12-30')

-- ACT

INSERT INTO EmployeeFind.Actual

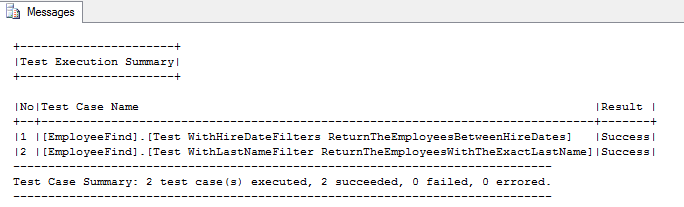
EXEC EmployeeFind null,'2010-12-29','2010-12-30'

-- ASSERT

EXEC tSQLt.AssertEqualsTable 'EmployeeFind.Expected', 'EmployeeFind.Actual';

END

GO



Ahora vamos a crear pruebas para el procedure EmployeeCreate. Si examinamos este procedure veremos que tiene una pequeña lógica condicional en la cuál si no se ingresa la fecha, esta será la fecha actual del sistema.

CREATE PROCEDURE [dbo].[EmployeeCreate]

@FirstName nvarchar(100),

@LastName nvarchar(100),

@HireDate DateTime

AS

BEGIN

IF @HireDate IS NULL

SET @HireDate=GetDate();

INSERT INTO [dbo].[Employee]

([FirstName]

,[LastName]

,[HireDate])

VALUES

(@FirstName

,@LastName

,@HireDate);

END

Creamos la primera prueba.

CREATE PROCEDURE EmployeeCreate.[Test WithExplicitHireDate TheEmployeeHasTheExplicitTheHireDate]

AS

BEGIN

-- ARRANGE

EXEC tSQLt.FakeTable 'Employee'

-- ACT

EXEC EmployeeCreate 'Juan','Marquez','2010-12-31'

DECLARE @Count int=(SELECT count(1) FROM Employee)

-- ASSERT

EXEC tSQLt.AssertEquals 1, @Count

END

Esta prueba no es suficiente ya que nuestro procedures tiene segmentos lógicos que necesitan se probados, en este caso necesitamos probar que al insertar el empleadado se debe usar la fecha ingresada si es que esta no es null. Necesitamos verificar que el valor del hiredate que se encontrará en la tabla es el misma fecha ingresada como parámetro del procedure. Para esto podemos volver a utilizar el patrón “Compare Table Assert”.

CREATE PROCEDURE EmployeeCreate.[Test WithExplicitHireDate TheEmployeeHasTheExplicitTheHireDate]

AS

BEGIN

-- ARRANGE

EXEC tSQLt.FakeTable 'Employee'

-- Store Expected Values

SELECT 'Juan' as FirstName

,'Marquez' as LastName

,'2010-12-31' as HireDate

INTO #Expected

-- ACT

EXEC EmployeeCreate 'Juan','Marquez','2010-12-31'

-- Collect Actual Data

SELECT FirstName,LastName,HireDate

INTO #Actual

FROM Employee

-- ASSERT

EXEC tSQLt.AssertEqualsTable '#Expected', '#Actual';

END

Podemos observar que en este caso podemos observar que estamos utilizando tablas temporales y por lo tanto no es necesario crear de manera anticipada las columnas de estas tablas sino que serán creadas apenas se estén ingresen valores la primera vez. Utilizar estas tablas pude ayudar a aumentar un poco la mantenibilidad de las pruebas aunque no siempre pueden ser utilizadas.

Ahora vamos a crear una nueva prueba para verificar la otra condición lógica de el procedure (que se utilize la fecha actual si el parámetro hiredate es nulo).

CREATE PROCEDURE EmployeeCreate.[Test WithoutHireDate TheEmployeeHasTheCurrentHireDate]

AS

BEGIN

EXEC tSQLt.FakeTable 'Employee'

EXEC EmployeeCreate 'Juan','Marquez',null

DECLARE @HireDate DateTime=(SELECT TOP 1 HireDate FROM Employee);

IF @HireDate IS NULL

EXEC tSQLt.Fail 'Expected: NOT NULL but was: NULL'

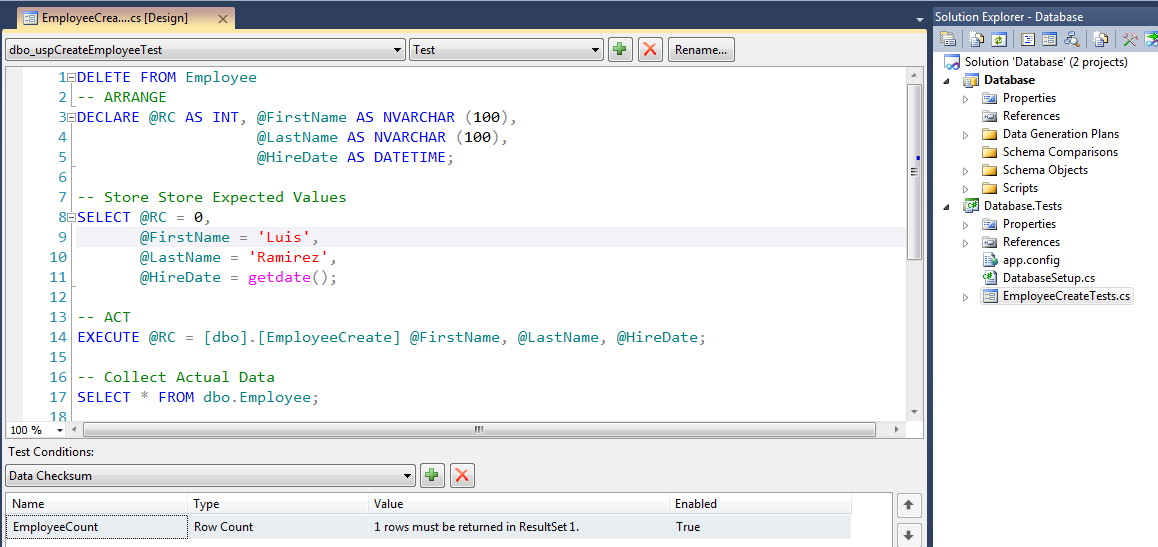
END

GO

En esta oportunidad ya no vamos a verificar el contenido de las otras columnas, únicamente vamos a verificar que el empleado no será insertado con un valor de nulo. Debido a que la implementación del Procedures asigna la fecha del sistema la HireDate es un muy dificil realizar una comparación con un valor fijo; como alternativa podríamos declarar 2 variables: una que guarde la fecha actual antes de ejecutar el procedure EmployeeCreate y otra que guarde la fecha al finalizar el procedure, como aserto podríamos verificar que el HireDate se encuentra entre esas 2 fechas.

## DB Projects

C:\Program Files (x86)\Microsoft Visual Studio 10.0\VSTSDB\Extensions\SqlServer\2008\DBSchemas\master.dbschema



# Web Testing

## Selenium IDE

Custom Formaters: <http://koenwillemse.wordpress.com/2011/02/16/custom-c-formatter-for-selenium/>

Instalar previamente:

* Firebug: <https://addons.mozilla.org/es/firefox/addon/firebug/>
* FirePath: <https://addons.mozilla.org/en-US/firefox/addon/firepath/>
* Chrome Path: <https://chrome.google.com/webstore/detail/hgimnogjllphhhkhlmebbmlgjoejdpjl>
* Batch Export: https://addons.mozilla.org/en-US/firefox/addon/test-suite-batch-converter-sel/versions/

Antes de empezar a crear las pruebas vamos a explorar primero la página Stackoverflow. Como sabemos es una página donde se escriben preguntas sobre programación y otras personas las responden. Una de las funcionalidades de esta página es su buscador que nos permite buscar posts tanto por palabras clave como por “tags”.

TestSuite: SearchPostsByTag

TestCases:

* SearchResultHasTheCorrectTag
* Relatated Tags

Descargamos Selenium IDE: Abrir Firefox e ingresar a la siguiente dirección: <http://seleniumhq.org/download/>

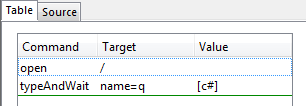
Abrimos Selenium: Ver- PanelLateral o Herramientas-SeleniumIDE.

Por defecto el plugin ya se encuentra en grabación. (Botón Rojo). Paramos la grabación por el momento.

Lo primero que tenemos que hacer para comenzar agravar algo es escribir la URL base de la página desde la cuál se van a comenzar a grabar las pruebas.

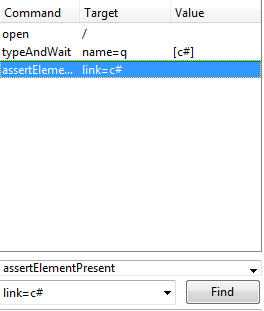
En la parte izquierda observamos la lista de tests cases que tenemos actualmente, podemos observar que no tiene aún un nombre, lo primero que vamos es dar un nombre que represente la prueba que queremos hacer. “SearchResultHasTheCorrectTag”.

* Escribimos [c#] en la barra del búsqueda.

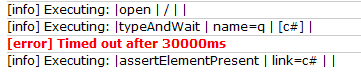


En la tabla podemos observar todas las acciones que están siendo grabadas, esta tabla tiene 3 columnas:

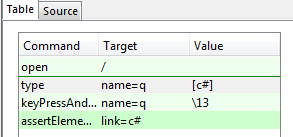
* + Command: El nombre del comando
  + Target: Representa el elemento sobre el elemento con el cuál estamos interactuando, este puede ser representando de muchas maneras como veremos luego.
  + Value: Son parámetros adicionales que son ingresados dependiendo del comando.
* Si hacemos click en alguno de estos comandos de la tabla podremos ver el panel inferior con los datos de este comando. A través de este panel podemos editar y crear nuevos elementos manualmente.
* En la parte inferior podemos observar documentación sobre el comando actual.
* Nos necesitamos asegurar que el resultado encontrado tiene el tag correcto. Es lo verificando que el link con el nombre “[c#]” se encuentra presente. Para esto utilizamos assertos. Para agregar un aserto solo tenemos que hacer click derecho en el elemento sobre el cuál queremos crear un aserto. Como sabemos que los tags están representados por links, el aserto más correcto sería constatar que existe un link con el texto “c#” en el primer resultado encontrado.



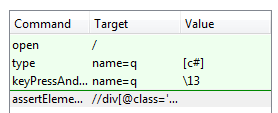
* Ahora q tenemos la prueba con el aserto, podemos ejecutarla para ver su funcionamiento. Vamos a observar que la prueba se detiene en el segundo comando y falla. De esto sacamos la primera lección al utilizar estas herramientas “No confiar en la grabación”, no son herramientas perfectas y muchas veces requieren que los comandos se modifiquen manualmente.
* En esta oportunidad la grabación no ha podido detectar que la búsqueda se ejecuta con el Enter y no con el click de un botón. Podemos observar el detalle del error en el log de la parte inferior.



* Para simular el enter en el teclado tenemos que agregar manualmente un comando y editar el comando type.



* Si observamos en los detalles del tag agregado podremos ver que su target es “link=c#”, este target podría no ser el más adecuado ya que es muy general (cambiar por otro texto y probar con el botón “find”) y tenemos muchos otros links con el mismo texto. Las forma más recomendada es utilizar el ID, pero en este caso nuestro elemento no tiene un id. Para esto usamos algo que se llama “XPATH”.
* Copiamos uno de los XPATH sugeridos y utilizamos el FirePath del Firebug para probar lo que significa.  
  //div[@id='question-summary-12138603']/div/div/a[contains(text(),'c#')]
* Con esto no quiero decir que siempre deban usar XPATH bastante precisos ya que hace que el test se vuelva más frágil a cambio. Cada vez que estén ingresando un locator deberán balancear la precisión de este con la fragilidad del test. Recordar que siempre es preferible usar el id en la medida de lo posible.
* Ejecutamos el test y grabamos.
* Creamos un nuevo test **ReleatedTags**.
* Grabamos ambos tests como parte del mismo Test Suite “SearchPostsByTag”.



//div[@class='module']/a[contains(@href, '/questions/tagged/c%23+.net')]

//h4[@id='h-related-tags']/following-sibling::a[contains(@href, '/questions/tagged/c%23+.net')]

* Ejecutamos y grabamos el test.
* Guardamos el testsuite “SearchPostByTag.testsuite.html”.
* Exportamos ambos testcases a C#. Podemos observar que el código generado pertenece a NUNIT y que tampoco ha podido exportar un elemento. Nueva lección: Selenium IDE no tiene una correspondencia directa con WebDriver y no es la forma correcta de crear scripts con WebDriver.

# Scripting

* Instalamos Selenium.Webdriver y Selenium Support
  + IE Problems: <http://www.jimmycollins.org/blog/?p=583>
  + Drivers: <http://code.google.com/p/selenium/downloads/list>
* Crear BD Bakery2 y cambiar el app.config y web.config con esa cadena.

## StackOverflow

[TestMethod]  
public void GettingStarted()  
{  
    //ARRANGE  
    var driver = new FirefoxDriver();  
    driver.Url = "http://www.google.com";  
  
    //ACT  
    driver.Navigate();  
  
    //ASSERT  
    Assert.AreEqual("Google", driver.Title);  
  
    //TEARDOWN  
    driver.Quit();  
}

[TestClass]  
public class StackoverflowTests  
{  
    private IWebDriver driver;  
    [TestInitialize]  
    public void Setup()  
    {  
        //driver = new InternetExplorerDriver(@"C:\Users\Snahider\Documents\Visual Studio 2010\Projects\ATes  
        //driver = new ChromeDriver(@"C:\Users\Snahider\Documents\Visual Studio 2010\Projects\WebTesting\Dri  
        driver = new FirefoxDriver();  
        driver.Manage().Timeouts().ImplicitlyWait(TimeSpan.FromSeconds(10));  
    }  
  
    [TestMethod]  
    public void ReleatedTags()  
    {  
        driver.Navigate().GoToUrl("http://stackoverflow.com/");  
  
        var textsearch = driver.FindElement(By.Name("q"));  
        textsearch.SendKeys("[c#]");  
        textsearch.Submit();  
  
        var relatedTags = driver.FindElements(By.XPath("//h4[@id='h-related-tags']/following-sibling::a"));  
        var exists = relatedTags.Any(x => x.Text == ".net");  
        Assert.IsTrue(exists);  
    }  
  
    [TestMethod]  
    public void ReleatedTags\_PageObject()  
    {  
        var stackoverflowPage = new StackoverflowPage(driver);  
  
        stackoverflowPage.Open();  
        stackoverflowPage.SearchFor("[c#]");  
  
        var relatedTags = stackoverflowPage.RelatedTags;  
        var exists = relatedTags.Any(x => x == ".net");  
        Assert.IsTrue(exists);  
    }  
  
    [TestMethod]  
    public void SearchResultHasTheCorrectTag()  
    {  
        driver.Navigate().GoToUrl("http://stackoverflow.com/");  
  
        var textsearch = driver.FindElement(By.Name("q"));  
        textsearch.SendKeys("[c#]");  
        textsearch.Submit();  
        //driver.FindElement(By.LinkText("save username but shows name in c#")).Click();  
        driver.FindElement(By.XPath("//div[@id='questions']/div[1]/div/h3/a")).Click();  
  
        var relatedTags = driver.FindElements(By.XPath("//div[@class='post-taglist']/a"));  
        var exists = relatedTags.Any(x => x.Text == "c#");  
        Assert.IsTrue(exists);  
    }  
  
    [TestCleanup]  
    public void Teardown()  
    {  
        driver.Quit();  
    }  
}

public class StackoverflowPage  
{  
    private readonly IWebDriver driver;  
  
    [FindsBy(How = How.Name, Using = "q")]  
    private IWebElement searchInput;  
  
    public StackoverflowPage(IWebDriver driver)  
    {  
        this.driver = driver;  
        PageFactory.InitElements(driver, this);  
    }  
  
    public void Open()  
    {  
        driver.Url = "http://stackoverflow.com/";  
    }  
  
    public void SearchFor(string query)  
    {  
        searchInput.SendKeys(query);  
        searchInput.Submit();  
    }  
  
  
    public IEnumerable<string> RelatedTags  
    {  
        get  
        {  
            return driver  
                .FindElements(By.XPath("//h4[@id='h-related-tags']/following-sibling::a"))  
                .Select(x => x.Text);  
        }  
    }  
}

## Bakery

### Organización

Separa los proyectos UnitTests de Integration y UI.

Representar todo en términos del usuario final.

Organizar los test por pantalla o por funcionalidad.

[TestClass]  
public class PurchasingCakes  
{  
    [TestMethod]  
    public void SeeProductDetailsWhenPlacingOrder()  
    {  
          
    }  
  
    [TestMethod]  
    public void PurchaseOneCake()  
    {  
  
    }  
}

**Primera Prueba (**

[TestMethod]  
public void SeeProductDetailsWhenPlacingOrder()  
{  
    //ARRANGE  
    //Load Data (Insert product with PRODUCTNAME y PRODUCTPRICE)  
  
    //ACT  
    var driver = new ChromeDriver();  
    //Navegar a la pagina  
    driver.Navigate().GoToUrl(ROOT\_URL + RELATIVE\_URL);  
    //Realizar click en un producto (nos muestra una nueva pantalla)  
    driver.FindElement("Find link by PRODUCTNAME").Click();  
  
    //ASSERT  
    //Verificar que se muestra el nombre del producto seleccionado  
    var productName = driver.FindElement("Find productname element").Text;  
    Assert.AreEqual("PRODUCTNAME", productName);  
    //Verificar que se muestra el precio del producto seleccionado  
    var price = driver.FindElement("Find price element").Text;  
    Assert.AreEqual("PRODUCTPRICE", price);  
  
    //TEARDOWN  
    //Cleanup Data

driver.quit();  
}

Completando los locators.

public void SeeProductDetailsWhenPlacingOrder()  
{  
    //ARRANGE  
    //Load Data (Insert product with PRODUCTNAME y PRODUCTPRICE)  
    //Configurar Driver  
    var driver = new ChromeDriver(@"C:\Users\Snahider\Documents\Visual Studio 2010\Projects\A  
  
    //ACT  
    //Navegar a la pagina  
    driver.Navigate().GoToUrl("http://localhost:6709/");  
    //Realizar click en un producto (nos muestra una nueva pantalla)  
    driver.FindElement(By.XPath("//h3[contains(text(),'Carrot Cake')]/../.././/a")).Click();  
  
    //ASSERT  
    //Verificar que se muestra el nombre del producto seleccionado  
    var productName = driver.FindElement(By.TagName("h1")).Text;  
    Assert.IsTrue(productName.Contains("Carrot Cake"));  
    //Verificar que se muestra el precio del producto seleccionado  
    var price = driver.FindElement(By.Id("orderPrice")).Text;  
    Assert.IsTrue(price.Contains("8.99"));  
  
    //TEARDOWN  
    //Cleanup Data  
    //Cerrar Driver  
    driver.Quit();  
}

Completando la inicialización de los datos y removiendo problemas.

[TestMethod]  
public void SeeProductDetailsWhenPlacingOrder()  
{  
    context.Products.Add(new Product  
        {  
        Name = "Apple Cake",  
        Description = "Default Descripcion",  
        Price = 8.99m,  
        ImageName = "carrot\_cake.jpg"  
    });  
    context.SaveChanges();  
  
    driver.Navigate().GoToUrl("http://localhost:6709/");  
    driver.FindElement(By.XPath("//h3[contains(text(),'Carrot Cake')]/../.././/a")).Click();  
  
    var productName = driver.FindElement(By.TagName("h1")).Text;  
    Assert.IsTrue(productName.Contains("Apple Cake"));  
    var price = driver.FindElement(By.Id("orderPrice")).Text;  
    Assert.IsTrue(price.Contains("8.99"));  
}

* Remover la inicialización de los datos en los test setup y teardown

        private static IWebDriver driver;  
   
        private AppDbContext context;  
   
        [AssemblyInitialize]  
        public static void SetupOnlyOnce(TestContext testContext)  
        {  
            driver = new ChromeDriver();  
        }  
   
        [TestInitialize]  
        public void Setup()  
        {  
            context=new AppDbContext();  
        }  
   
        [TestCleanup]  
        public void Teardown()  
        {  
            context.Database.ExecuteSqlCommand("Delete from Products");  
        }  
   
        [AssemblyCleanup]  
        public static void TearDownOnlyOnce()  
        {  
            driver.Quit();  
        }

* Page Objects

public class ChooseProductPage  
{  
    private readonly IWebDriver driver;  
  
    public ChooseProductPage(IWebDriver driver )  
    {  
        this.driver = driver;  
    }  
  
    public void Open()  
    {  
        driver.Navigate().GoToUrl("http://localhost:6709/");  
    }  
  
    public PlaceOrderPage GoToPlaceOrderForProduct(string name)  
    {  
        driver.FindElement(By.XPath("//h3[contains(text(),'"+name+"')]/../.././/a"))  
              .Click();  
        return new PlaceOrderPage(driver);  
    }  
}

public class PlaceOrderPage  
{  
    private readonly IWebDriver driver;  
  
    [FindsBy(How = How.TagName, Using = "h1")]  
    private IWebElement productName;  
  
    [FindsBy(How = How.Id, Using = "orderPrice")]  
    private IWebElement productPrice;  
  
    public PlaceOrderPage(IWebDriver driver)  
    {  
        this.driver = driver;  
        PageFactory.InitElements(driver, this);  
    }  
  
    public string ProductName  
    {  
        get  
        {  
            return productName.Text;  
        }  
    }  
  
    public string ProductPrice  
    {  
        get  
        {  
            return productPrice.Text;  
        }  
    }  
}

[TestMethod]  
public void SeeProductDetailsWhenPlacingOrder()  
{  
    DataFactory.Load(new Product  
        {  
            Name = "Apple Cake",  
            Description = "Default Descripcion",  
            Price = 8.99m,  
            ImageName = "carrot\_cake.jpg"  
        });  
  
    var chooseProductPage = new ChooseProductPage(driver);  
    chooseProductPage.Open();  
    var placeOrderPage = chooseProductPage.GoToPlaceOrderForProduct("Apple Cake");  
  
    Assert.IsTrue(placeOrderPage.ProductName.Contains("Apple Cake"));  
    Assert.IsTrue(placeOrderPage.ProductPrice.Contains("8.99"));  
}

**Segundo Test**