IMPLEMENTANDO EL PATRÓN REPOSITORIO Y UNIDAD DE TRABAJO (UNIT OF WORK & REPOSITORY) – ASP.NET MVC 5

Publicado el 20 noviembre, 201520 noviembre, 2015 por Isaac Ojeda

En esta entrada veremos un patrón de diseño que utilizo ya en todos los proyectos en los que participo, que consiste básicamente en la creación de una capa intermedia que se encuentra entre la Data Layer (acceso a datos) y la Bussiness Layer (reglas de negocio).

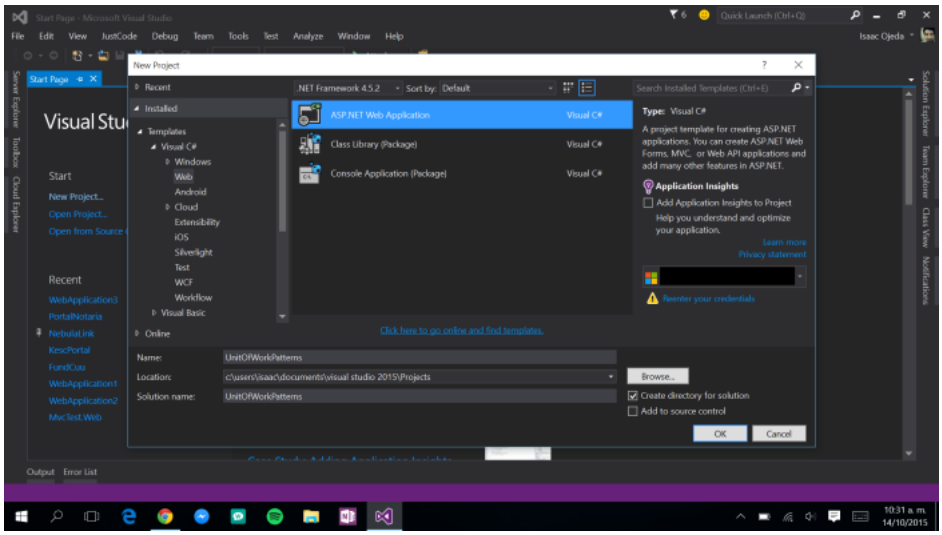
El uso de repositorios llega a ser muy común, ya que permite tener bien divida la aplicación, re utilización de código y ademas hace más sencillo el uso de pruebas unitarias, más si usamos interfaces e inyección de dependencias para poder crear mockups, fake data y ese stuff.

En una unidad de trabajo su función principal es juntar todos los repositorios que conforman nuestra capa de datos y ordenarlos de tal forma que permiten el trabajar en el mismo contexto de Entity Framework y poder hacer operaciones entre repositorios y todo en las mismas transacciones. La unidad de trabajo es utilizada por la capa de negocios, que por lo regular ahí se incluyen las reglas que nuestra aplicación tendrá. Y finalmente el controlador de MVC que es el que nos comunicará con la vista. Sin mencionar que en el front-end podemos tener arquitecturas MVW (AngularJS) o MVVM (KnockoutJS, KendoUI)… un sin fin de patrones que son tan emocionantes!

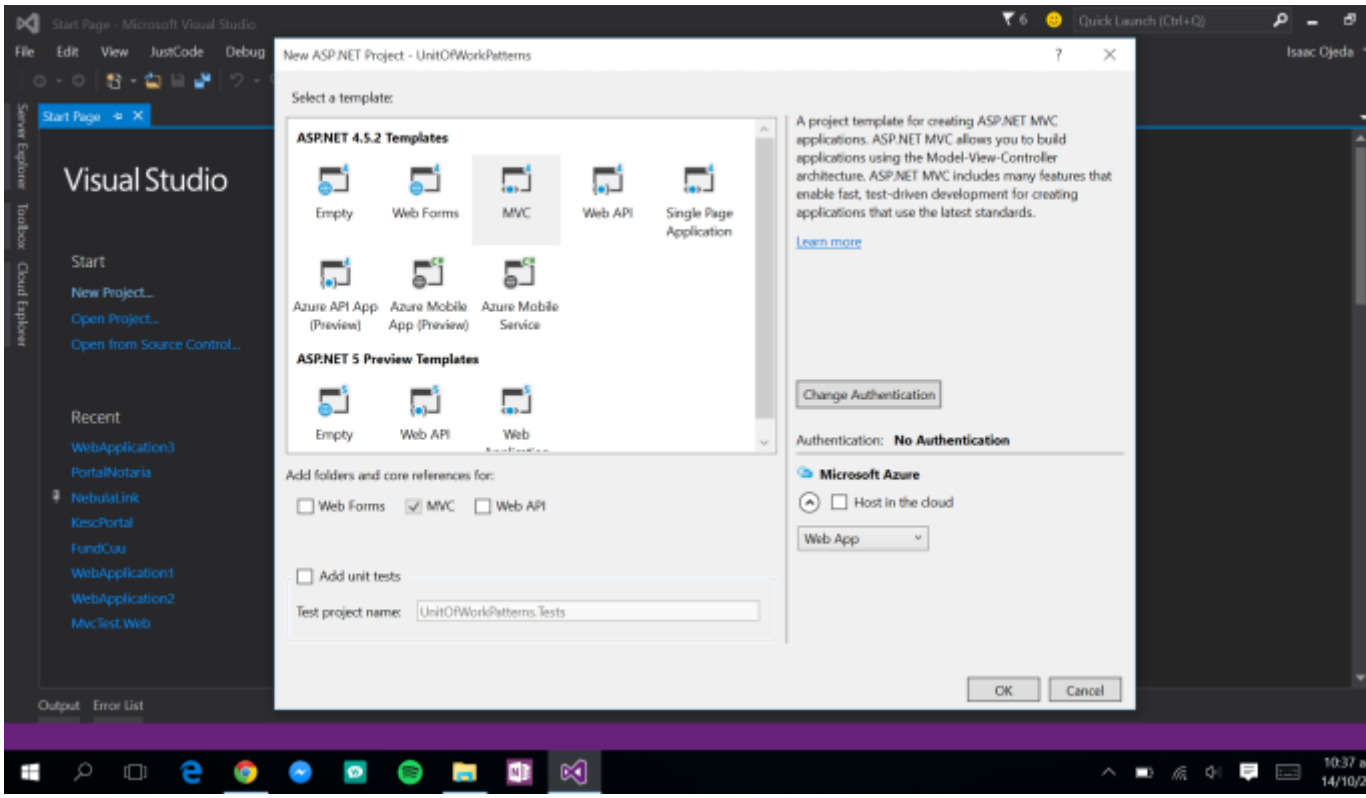
Si no sabes muy bien que es un repositorio, puedes leer este articulo (en ingles). Pero básicamente consiste en evitar la duplicación de código, la centralización del acceso a datos (a veces es necesario tener en caché ciertas cosas), toda la aplicación tendrá el acceso a datos en un solo lugar. Pero después de tener muchos repositorios, que cada uno tiene su contexto, hay que organizar lo que se supone tenemos organizado. Ahí entran las Unit of Work.

Es normal crear un repositorio por cada entidad de datos que tenemos (Ejem. Clientes, Contactos, Ventas, etc) y por lo regular cada uno de estos tiene básicamente las mismas operaciones CRUD (Create, Read, Update & Delete). Por lo tanto no es muy práctico el tener que escribir las mismas operaciones por cada entidad (tabla) que tenemos en la base de datos. Es por eso que en este post veremos como crear un repositorio genérico, que permitirá hacer las operaciones básicas en una entidad utilizando el mismo código para todas.

Creando un proyecto en Visual Studio



Yo tengo el Visual Studio 2015, pero utilizaremos MVC 5.



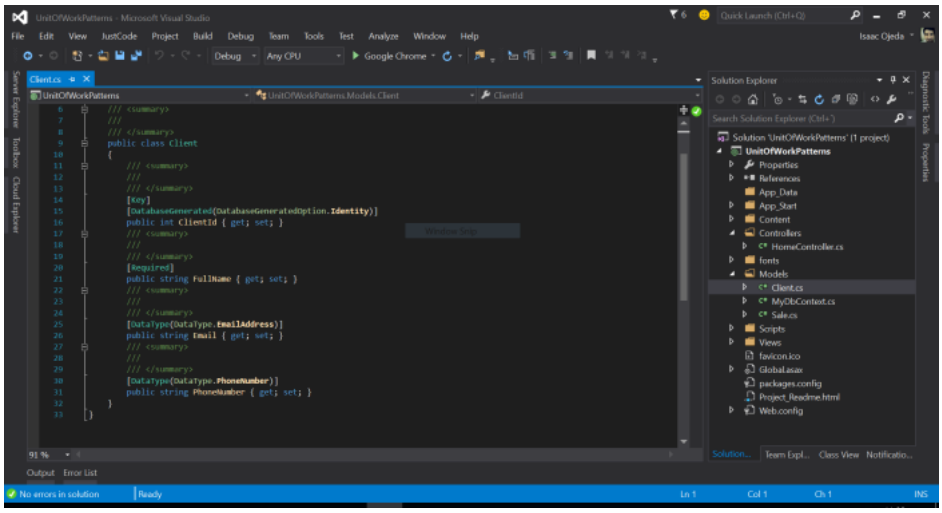
Estoy creando el proyecto MVC sin Authentication, pero la verdad esto viene siendo irrelevante.

Lo primero que voy a hacer, es instalar Entity Framework utilizando Nugget, después de eso crearemos nuestro modelo (muy sencillo) utilizando Code First en lugar de Database First (para hacerlo todo desde visual studio, pero la forma de hacerlo no importa al final).

Crearemos una base de datos de clientes y ventas utilizando Code First (al final solo utilizaremos la tabla Clientes para fines prácticos). Las siguientes clases

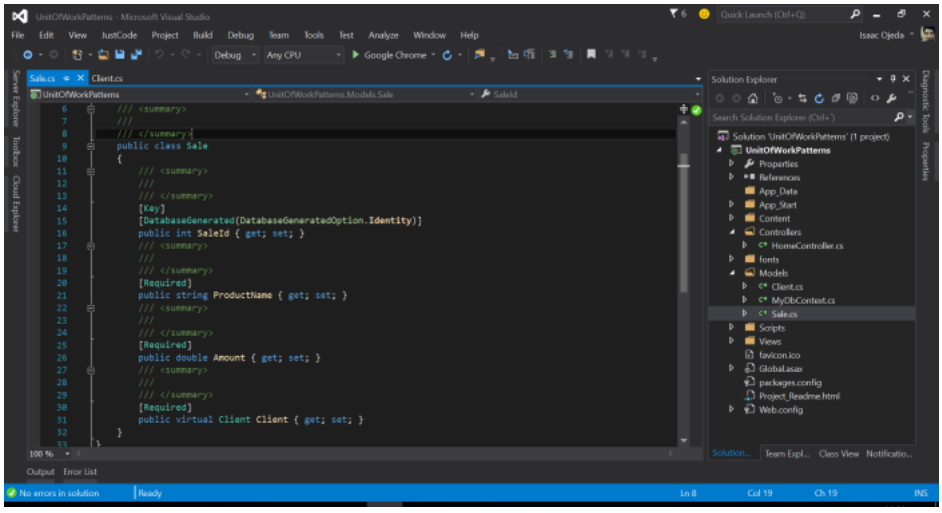
las agregaremos en la carpeta Models que ya existe, sino, donde tu gustes. Dejándolo de la siguiente forma:

User.cs

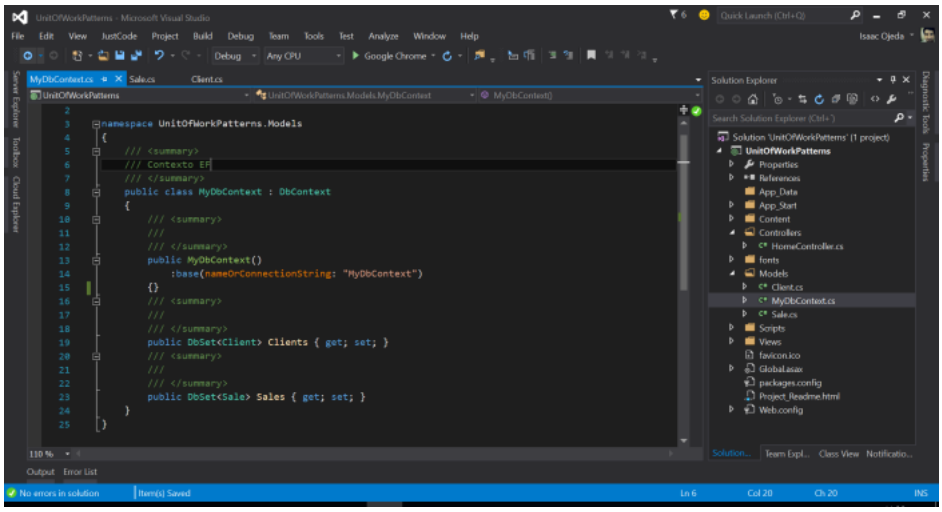


Aquí agregamos unos Data Annotations para agregar propiedades a la base de datos final que se generará (Auto ingremento, not null, etc)

Sale.cs



MyDbContext.cs



Aquí estamos creando nuestro contexto que incluye las dos entidades que hemos creado. En el constructor estamos indicando la cadena de conexión de la base de datos, que es la siguiente:

<add connectionString="Server=(local);Database=MyDbContext;Trusted_Connection=True;MultipleActiveResultSets=true" name="MyDbContext" providerName="System.Data.SqlClient"/>

Es importante decir que tenemos que tener instalada una instancia de Sql Server.

En este punto deberíamos de poder compilar sin problemas.

Agregando un repositorio genérico

Regularmente, esto que estamos haciendo lo hacemos en una librería aparte, pero por simplicidad lo hacemos todo en el mismo proyecto.

Lo que vamos a hacer ahora es crear una clase que será un repositorio genérico que pueda ser reutilizado por las dos entidades que acabamos de crear. Para esto crearemos una carpeta llamada **Repositories** a nivel de nuestro proyecto y agregaremos una clase llamada **GenericRepository** como se muestra a continuación:

GenericRepository.cs

```
1  public class GenericRepository<TEntity>
```

```
    where TEntity: class
{

    private readonly MyDbContext context;
    private readonly DbSet<TEntity> dbSet;

    public GenericRepository(MyDbContext context)
    {
        this.context = context;
        this.dbSet = context.Set<TEntity>();
    }
    public void Create(TEntity entity)
    {
        dbSet.Add(entity);
    }
    public void CreateRange(IEnumerable<TEntity> entities)
    {
        foreach (var entity in entities)
        {
            Create(entity);
        }
```

```
22      }
23      public async Task<TEntity> FindAsync(params object[]
keyValues)
24      {
25          return await dbSet.FindAsync(keyValues);
26      }
27      public virtual IQueryable<TEntity> SelectQuery(string
query, params object[] parameters)
28      {
29          return dbSet.SqlQuery(query,
parameters).AsQueryable();
30      }
31      public void Update(TEntity entity)
32      {
33          dbSet.Attach(entity);
34          context.Entry(entity).State = EntityState.Modified;
35      }
36      public void Delete(TEntity entity)
37      {
38          if (context.Entry(entity).State ==
EntityState.Detached)
39          {
40              dbSet.Attach(entity);
41
```

```
42          }
43          dbSet.Remove(entity);
44      }
45      public async Task Delete(params object[] id)
46      {
47          TEntity entity = await this.FindAsync(id);
48          if (entity != null)
49          {
50              this.Delete(entity);
51          }
52      }
53      public IQueryable<TEntity> Queryable()
54      {
55          return dbSet;
56      }
57  }
```

Aquí estamos permitiendo hacer las operaciones básicas que se podrán hacer con una entidad. La combinación de estas te permitirán hacer cualquier cosa.

Los repositorios se pueden implementar de muchas formas, pero siento yo que entre menos escribas y más reutilices es mejor, por eso me gusta irme por hacer un repositorio genérico.

Si se dieron cuenta, existen métodos CRUD y algunos más, pero en ningún lado podemos ver el método “SaveChanges” o guardar nuestro contexto. Esto es porque el que se encargará de hacer las operaciones en la base de datos

será la unidad de trabajo, para tener centralizado el contexto y cuando se guarden los cambios se haga todo en una sola transacción sin importar cuantos repositorios modificamos.

UnitOfWork.cs

```
1  public class UnitOfWork
2  {
3      public UnitOfWork()
4      {
5          this.context = new MyDbContext();
6      }
7      private readonly MyDbContext context;
8
9      private GenericRepository<Client> clientsRepository;
10     public GenericRepository<Client> ClientsRepository
11     {
12         get
13         {
14             if (this.clientsRepository == null)
15             {
16                 this.clientsRepository = new
GenericRepository<Client>(this.context);
17             }
```

```
18              return this.clientsRepository;
19          }
20      }
21
22      private GenericRepository<Sale> salesRepository;
23      public GenericRepository<Sale> SalesRepository
24      {
25          get
26          {
27              if (this.salesRepository == null)
28              {
29                  this.salesRepository = new
GenericRepository<Sale>(this.context);
30              }
31              return this.salesRepository;
32          }
33      }
34
35      public async Task  SaveChangesAsync()
36      {
37          await this.context.SaveChangesAsync();
```

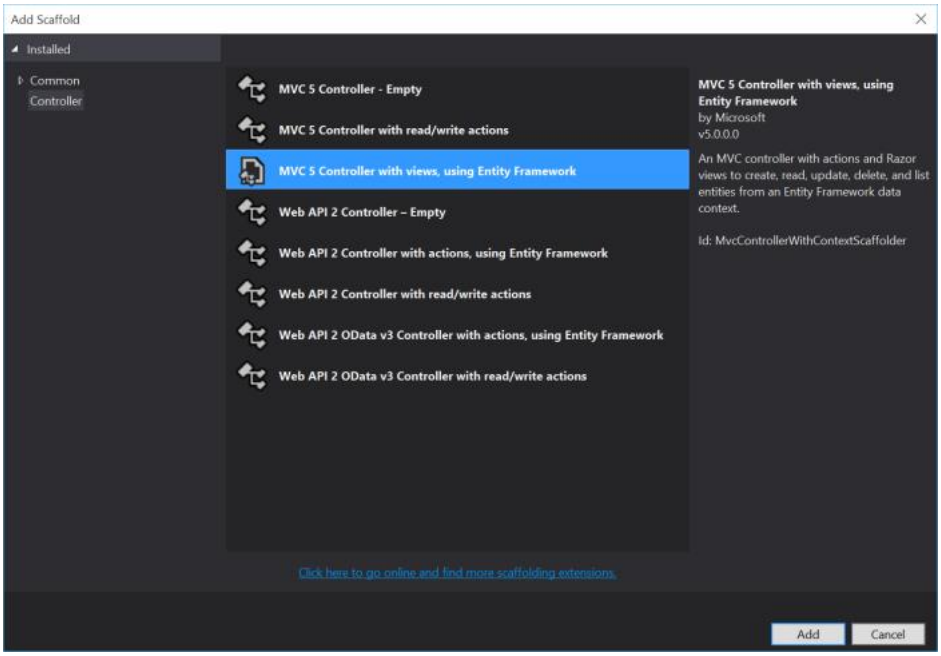
```
38          }
39  }
```

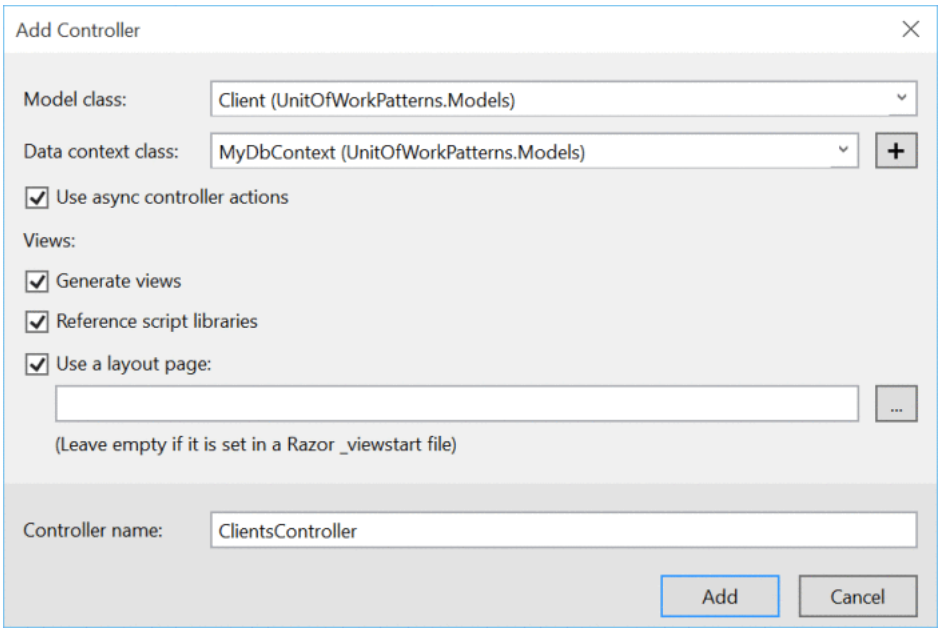
De esta forma estamos centralizando nuestros repositorios y todos trabajando en un mismo contexto.

De aquí en adelante ya podemos usar nuestra unidad de trabajo para hacer cualquier operación. Se puede usar directamente en el controlador MVC o desde una clase Servicio (business layer) que haga todo. Nosotros lo haremos desde el controlador, luego podremos ver la implementación de servicios e inyección de dependencias para facilitar las pruebas unitarias.

Controlador ClientsController.cs

Crearemos un controlador llamado Clients, simplemente para llenar de información sencilla y poder mostrarla. Utilizaremos el scaffolding de Visual Studio para no batallar. Agregamos un nuevo controlador y seleccionamos la opción mostrada:





Este scaffolding nos generará el acceso a datos de forma directa con el contexto, por lo tanto tendremos que modificar el controlador a la siguiente forma (realmente lo que nos importa de este scaffolding son las vistas) y haremos uso de la unidad de trabajo que acabamos de crear:

```
1  public class ClientsController : Controller
2  {
3      private readonly UnitOfWork unit = new UnitOfWork();
4
5      // GET: Clients
6      public ActionResult Index()
7      {
8          return View(unit.ClientsRepository.Queryable().ToList());
9      }
10
```

```
11      // GET: Clients/Details/5
12      public async Task<ActionResult> Details(int? id)
13      {
14          if (id == null)
15          {
16              return new
HttpStatusCodeResult(HttpStatusCode.BadRequest);
17          }
18          Client client = await
unit.ClientsRepository.FindAsync(id);
19          if (client == null)
20          {
21              return HttpNotFound();
22          }
23          return View(client);
24      }
25
26
27      // GET: Clients/Create
28      public ActionResult Create()
29      {
30          return View();
        }
```

```
31
32      // POST: Clients/Create
33      [HttpPost]
34      [ValidateAntiForgeryToken]
35      public async Task<ActionResult> Create([Bind(Include =
"ClientId,FullName,Email,PhoneNumber")] Client client)
36      {
37          if (ModelState.IsValid)
38          {
39              unit.ClientsRepository.Create(client);
40              await unit.SaveChangesAsync();
41              return RedirectToAction("Index");
42          }
43
44          return View(client);
45      }
46
47      // GET: Clients/Edit/5
48      public async Task<ActionResult> Edit(int? id)
49      {
50          if (id == null)
```

```
51          {
52              return new
HttpStatusCodeResult(HttpStatusCode.BadRequest);
53          }
54          Client client = await
55  unit.ClientsRepository.FindAsync(id);
56          if (client == null)
57          {
58              return HttpNotFound();
59          }
60          return View(client);
61      }
62
63      // POST: Clients/Edit/5
64      [HttpPost]
65      [ValidateAntiForgeryToken]
66      public async Task<ActionResult> Edit([Bind(Include =
"ClientId,FullName,Email,PhoneNumber")] Client client)
67      {
68          if (ModelState.IsValid)
69          {
70              unit.ClientsRepository.Update(client);
```

```
71              await unit.SaveChangesAsync();
72
73              return RedirectToAction("Index");
74          }
75          return View(client);
76      }
77
78      // GET: Clients/Delete/5
79      public async Task<ActionResult> Delete(int? id)
80      {
81          if (id == null)
82          {
83              return new
HttpStatusCodeResult(HttpStatusCode.BadRequest);
84          }
85          Client client = await
unit.ClientsRepository.FindAsync(id);
86          if (client == null)
87          {
88              return HttpNotFound();
89          }
90          return View(client);
```

```
91      }
92
93      // POST: Clients/Delete/5
94      [HttpPost, ActionName("Delete")]
95      [ValidateAntiForgeryToken]
96      public async Task<ActionResult> DeleteConfirmed(int id)
97      {
98          Client client = await
unit.ClientsRepository.FindAsync(id);
99          unit.ClientsRepository.Delete(client);
100         await unit.SaveChangesAsync();
101         return RedirectToAction("Index");
102     }
    }
```

Si tuviéramos que hacer operaciones con ventas o cualquier otra cosa que se encontrara en nuestra unidad de trabajo, al guardar los cambios a nivel **UnitOfWork** se guardarían los cambios de un solo contexto, por lo tanto todo los cambios hechos en todos los repositorios se harían en una sola transacción. Y si nos ponemos a pensar, esto mismo lo podría hacer **MyDbContext**, porque básicamente es una unidad de trabajo también, pero hacerlo de esta forma hace que esté totalmente ligado a Entity Framework y no se podrá crear una abstracción para delegar responsabilidades (Capas).

Manejando un solo contexto permite que todo sea persistente si algo falla, reutilizar código y dar la posibilidad de manejar TDD (Test Driven Development) implementando técnicas de mockups o fake data. Claro, para lograr TDD tendríamos que hacer uso de interfaces y algunas modificaciones para que las implementaciones no estén totalmente ligadas.

Si corremos la aplicación y nos vamos al url http://localhost:xxxx/Clients podemos ver ya un CRUD de clientes utilizando la unidad de trabajo.

Espero que les sea de utilidad, porque ya tengo utilizando este patrón bastante tiempo y siempre que se viene un proyecto nuevo, esto ya es un estandard en la oficina.