Refactorizando hacia una arquitectura más limpia.

En este módulo vamos a refactorizar nuestra app a una arquitectura limpia, usando los patrones correctos, una app con arquitectura limpia no tiene nada que ver con herramientas externas librerías o FrameWorks, nosotros vamos a usar patrones como el famoso patrón repositorio y el no tan usado patrón de unidad de trabajo “unit of work” para desacoplar nuestra app de Entity Framework, que si bien no recomiendo nunca el desuso de EF, es nuestro deber como desarrollador, crear un producto que sea los suficientemente desacoplado, para que el cambio, en caso de ser necesario, no se convierta en un dolor de cabeza, imagínate que mañana se descubre un bache enorme de seguridad en EF y tu aplicación es una aplicación bancaria que debe moverse de inmediato a Ado, bueno, para eso somos duros tirando código y eso es lo que vamos a hacer en este capítulo..

El primer paso es verificar la acción “Attending” en el EventsController, si nos fijamos esta acción es muy cargada, hay demasiadas cosas ocurriendo aquí, si dejó de trabajar un par de semanas en este proyecto, existe un alto grado de probabilidad, que me voy a topar con algo que me va a tomar un poco de tiempo entender lo que estoy haciendo o dicho de otra forma, de que al pasar un par de días no entienda ni mierda de lo que dice ahí, porque hay demasiadas líneas de código, de hecho, como ha pasado un buen tiempo entre el video anterior y este, yo dure un rato viendo la guía para poder acordarme de lo que hacia casa cosa. Lol, aquí voy al contexto, filtro datos, selecciono un evento, incluyo la información del coach y el tipo de evento,

Lo primero entonces es separar todos estos queries, seleccionamos el valor que le seteamos a la variable events, presionamos la bombilla de “quick refactoring” y presionamos en extraer el método. El nombre que le pondré al método debe ser algo que represente lo que estoy haciendo, para que en el futuro yo entienda de una lo que hace cada método sin tener que analizar mucho código, lo llamaré “GetEventsUserAttending”.

```
public ActionResult Attending()
        {
            var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);

            var events = _context.Attendances
            .Where(a => a.AttendeeId == userId)
            .Select(a => a.Event)
            .Include(p => p.Type)
            .Include(p => p.Coach)
            .ToList(); GetEventsUserAttending(userId);

            var attendances = _context.Attendances
…
}
```

Si bajamos un poco podemos ver que el método recibe un usuario como parámetro necesario para su ejecución, si ya estamos contentos con el nombre y vemos que todo está bien, le damos a aplicar.

```
  private List<Event> GetEventsUserAttending(string userId)
        {
            return _context.Attendances
            .Where(a => a.AttendeeId == userId)
            .Select(a => a.Event)
            .Include(p => p.Type)
            .Include(p => p.Coach)
            .ToList();
        }
```

Ahora voy a seleccionar el segundo query, pero solo hasta antes del lookup, porque en el futuro si yo quiero reutilizar esta lógica, no siempre voy a necesitar un lookup, después si quiero creo otro método que me retorne un lookup en lugar de una lista, pero por

ahora no es necesaria. Además, lookup depende de Linq, y como estos métodos que estamos extrayendo lo estamos haciendo con la finalidad de desacoplarnos y mandar estos para un proyecto de clase repositorio que no necesariamente va a tener Linq, mejor lo hacemos lo más standard posible, así que a este lo llamo "GetFutureAttendances"

```
  public ActionResult Attending()
{
…
        List<Event> events = GetEventsUserAttending(userId);

        var attendances = _context.Attendances
          .Where(a => a.AttendeeId == userId && a.Event.DateTime > DateTime.Now)
          .ToList()  GetFutureAttendances(string userId)
          .ToLookup(a => a.EventId);

        var vm = new EventsViewModel
…
}
    private List<Attendance> GetFutureAttendances(string userId)
    {
        return _context.Attendances
                    .Where(a => a.AttendeeId == userId && a.Event.DateTime > DateTime.Now)
                    .ToList();
    }
```

Ahora tenemos un método más hermoso que de forma directa nos cuenta una historia por sí sola, sin tener que durar mucho rato pensando que carajos queríamos hacer en este lugar. Podemos hacer esto aún más corto porque en este caso particular, nosotros ya no necesitamos las variables, pues no iteramos ni hacemos nada con la información que extraemos de la base de datos, sino que de forma íntegra la pasamos al modelo, por lo cual, podemos inicializar de forma directa los métodos en la construcción del EventsViewModel.

Nos situamos sobre la variable y le damos al pincel de ayuda o Ctrl + ".", buscamos la opción "Inline Variable" vemos el cambio que va a sufrir nuestro código en el preview y le damos enter.

```
public ActionResult Attending()
      {
          var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);
          List<Event> events = GetEventsUserAttending(userId);

          var attendances = GetFutureAttendances(userId)
              .ToLookup(a => a.EventId);

          var vm = new EventsViewModel
          {
              UpcomingEvents = events  GetEventsUserAttending(userId),,
              ShowActions = User.Identity.IsAuthenticated,  ,Heading = "Events I'm Attending",
              Attendances = attendances  GetFutureAttendances(userId).ToLookup(a => a.EventId)
          };

          return View(vm);
}
```

Mira que vaina más chula y hermosa. Ahora bien, en el EventsController ya hemos extraído el método para obtener los asistentes a un evento, pero en el HomeController yo tengo exactamente la misma lógica, esto me dice que extraer el método de forma privada no es la forma correcta de hacer las cosas así que tenemos que mover nuestra lógica a una arquitectura con un poco más de sentido que me permita la reutilización de código en diversos lugares, para solucionar eso viene el repositorio.

Repositorio

Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects

Pattern of Enterprice Application Arquitectures – Martin Fowler

Media entre el dominio y las capas de mapeo de datos usando una interfaz similar a una colección para acceder a los objetos del dominio.

Esta definición es muy académica, así que vamos con la explicación aplatanada yendo directo al mambo, ¿Cuáles son los beneficios del repositorio?

- Minimizar la duplicidad de lógica de queries: so, en lugar de tener un método privado, que nos traiga los eventos, creamos una clase llamada repositorio donde tenemos ese método y este puede ser usado tanto por EventsController como por HomeController, esto me hace no tener lógica duplicada por ahí.
- Provee Mejor separación de Conceptos: El controlador ya no es responsable de lidiar con queries, que realmente nunca debió ser responsabilidad de este, estar manipulando data, el solo debe ser un manejador, un intermediario que pide lo que necesita, el no debe ir a la cocina a cocinar, el solo debe pedir al cheft el plato y cuando este listo ir a servirlo. El controlador es responsable del flujo de la aplicación, el controlador recibe una petición y genera una respuesta, todos los detalles en este proceso, deben ser delegados. El debe ser un manejador, delega a diferentes objetos el trabajo, el pregunta al repositorio por las asistencias futuras de un usuario que el va a pasar y el repositorio que se las ingenie como pueda para traer la información, como el repositorio va a traer esa data, que métodos va a usar, que patron, que framework, es asunto exclusivo del repositorio al controlador no le importa, el repositorio puede usar Linq, Store Procedures, strings, para traer data de la base de datos o de data en memoria o de donde el repositorio al que nos conectamos entienda
- Desacopla tu aplicación de un framework ORM especifico: si bien este beneficio no aplica en .net core, por la forma en que esta estructurado, una de las razones por la que nació era la capacidad de poder usar cualquier framework basado en modelos para trabajar nuestros proyectos sin necesidad de

hacer cambios drásticos al aplicativo. Cosa que argumentaban algunos desarrolladores que era así ya en MVC, pues Microsoft nos mentía en el pasado diciéndonos que los proyectos podías usar cualquier framework, pero no era nada cierto, con el nivel de modularización que existe ahora en core, esto es una realidad, siempre que sepamos como se hace y para eso estamos acá.

¿Tengo que usar Repositorio si o no?

Bueno, el patrón repositorio igual que cualquier otro patrón, está diseñado para resolver un problema en específico. Si no tenemos al menos uno de los 3 problemas que resuelve el patrón repositorio, no necesitamos implementarlo para nada, hacerlo no te hace un mejor desarrollador, a pesar de que hay ciertos latinos que te van a criticar por no usarlo, aunque ellos no hayan hecho nunca un hola mundo, solo para que crean que ellos saben, me pasó par de veces con mi app de redes sociales, la cual tenía la separación de conceptos bien hecha y cada cosa estaba centrada en su controlador, por lo cual no era necesario, tampoco tenía en mente cambiar de EF y la manipulación de los datos se hacía mediante métodos privados internos.

Usar el patrón repositorio tampoco incrementa la calidad de tu app, al contrario, solo incrementa la complejidad de nuestro diseño, hay que ser realistas, así que mantenlo simple y se pragmático

- Usa patrones para resolver problemas específicos.
- Si estamos trabajando en aplicaciones pequeñas no uses repositorio y menos si te están pagando chilates para que

entregues un proyecto rápido y te tienen a jugo con el látigo de la productividad.

En un aplicativo pequeño con queries simples que no involucran lógica compleja entre múltiples modelos, donde no hay duplicidad de estos, está bien escribir tus queries en el controlador, no es el fin del mundo, pero si tu app crece, puedes tener queries más grandes y complejos y como se pueden duplicar conviene usar repositorio.

Cuando la app va creciendo podemos ir teniendo querties complejos, controladores grandes y servicios grandes, es ahí que debemos usar la magia de los repositorios.

Cargar data, filtrar información, agrupamiento de datos, es responsabilidad de los datos, no del controlador o de los servicios, pero si el proyecto es simple, mantenlo simple.

Ahora, para iniciar, vamos a ver nuestros dos métodos en el EventsController, si nos fijamos, a pesar que el objeto que estamos manipulando en ambos casos es el dominio Attendance, nosotros estamos retornando en uno los Attendances y en otro los Events, por lo que ambos no son parte del repositorio Attendance, sino uno del repositorio Event y otro del repositorio Attendance, lo más correcto según el patrón, es un repositorio por cada dominio, donde siempre lo que se trabaje sean los elementos de ese dominio, por cierto, este es el motivo de porque no tiene sentido tener un repositorio genérico, mas que para ahorrar tiempo en una prueba de conceptos rápida o para algo bien chulo que les voy a mostrar más adelante.

Así que vamos a crear una carpeta llamada “Repositories” y dentro creamos una nueva clase que vamos a llamar “EventsRepository” y hacemos lo mismo para “AttendanceRepository”, nos vamos al EventsController, cortamos el método “GetFutureAttendances” y lo ponemos en el “AttendanceRepository”.

Lo primero que vamos a notar es que necesitamos el Contexto de datos, así que creamos un constructor que reciba el DbContext como parámetro y luego creamos el campo privado con la ayuda del Intellisense, luego hacemos el método publico y en lugar de retornar una lista, voy a retornar un IEnumerable, porque no tiene sentido para el cliente que llama esta clase pueda añadir un objeto de asistencia a la lista de asistentes devuelto por este método. Además de que la lista es ligeramente mas pesada que el IEnumerable, recuerden, usamos este solo cuando queremos que la data que servimos, no sea manipulable de forma directa y lo único que queremos es iterar sobre los objetos según el filtro que hagamos.

```
public class AttendanceRepository
 {
    private readonly DataContext _context;

    public AttendanceRepository(DataContext context)
    {
       _context = context;
    }

    private IEnumerable<Attendance> GetFutureAttendances(string userId)
    {
       return _context.Attendances
                  .Where(a => a.AttendeeId == userId && a.Event.DateTime > DateTime.Now)
                  .ToList();
    }

 }
```

Hacemos lo propio con el EventsRepository

```
public class EventsRepository
{
    private readonly DataContext _context;

    public EventsRepository(DataContext context)
    {
        _context = context;
    }
    public IEnumerable<Event> GetEventsUserAttending(string userId)
    {
        return _context.Attendances
        .Where(a => a.AttendeeId == userId)
        .Select(a => a.Event)          .Include(p => p.Type)          .Include(p => p.Coach)
        .ToList();
    }
}
```

Ahora nos vamos al controlador, creamos dos propiedades privadas del tipo de cada uno de nuestros repositorios e inicializamos los repositorios pasándole el context en el constructor, también podemos setear los repositorios directamente en el startup.cs que es lo más correcto para el tema de inyección de dependencias, pero lo veremos luego, después de esto llamamos a los métodos de sus respectivos repos.

```
public class EventsController : Controller
{
    private readonly DataContext _context;
    private readonly AttendanceRepository _attendanceRepository;
    private readonly EventsRepository  _eventsRepository;

    public EventsController(DataContext datacontext)
    {
        _context = datacontext;
        _attendanceRepository = new AttendanceRepository(_context);
        _eventsRepository = new EventsRepository(_context);
    }
public ActionResult Attending()
    {
        var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);

        var vm = new EventsViewModel
        {
            UpcomingEvents = _eventsRepository.GetEventsUserAttending(userId),
            ShowActions = User.Identity.IsAuthenticated,
            Heading = "Events I'm Attending",
```

```
            Attendances = _attendanceRepository.GetFutureAttendances(userId).ToLookup(a => a.EventId)
        };

        return View(vm);

    }
```

Y ahora que ya tenemos esto implementado podemos remover la lógica duplicada del home controller, creando una propiedad privada que llame al attendingRepository y eso.

```
 public class HomeController : Controller
 {
    private readonly DataContext _context;
    private readonly AttendanceRepository _attendanceRepository;

    public HomeController(DataContext context)
    {
        _context = context;
        _attendanceRepository = new AttendanceRepository(_context);
    }
 public IActionResult Index(string query = null)
    {
       ...
        var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);
        var attendances = _context.Attendances
            .Where(a => a.AttendeeId == userId && a.Event.DateTime > DateTime.Now)
            .ToList()  _attendanceRepository.GetFutureAttendances(userId)
            .ToLookup(a => a.EventId);

        var vm = new EventsViewModel
        {
            UpcomingEvents = upcomingEvents,
            ShowActions = User.Identity.IsAuthenticated,
            Heading = "Upcoming Events ", //My upcoming events on the other screen
            SearchTerm = query,
            Attendances = attendances
        };
…
```