

Guía definitiva para aprender a desarrollar soluciones Web con C# y Net Core MVC.

(Manual con lo “reale” y “veldadero” truco para desarrolladores .Net)

Prof. Starling Germosen Reynoso.

La realización de este material viene como compendio personal y aplanamiento de diversos cursos y tutoriales adquiridos legalmente de diversas subscripciones en distintas plataformas, como Udemy, PluralSight, MVA, Coursera, Miriadax, EDX, entre otras, usando las metodologías de enseñanza de las cuales carecen estos cursos originales, por ser realizados sin técnicas pedagógicas para gente auto didacta, por lo que los temas a tocar quizá tengan un nombre relacionado a algunos de estos.

Caben destacar los cursos de Mosh Hamedani y Juan Carlos Zuluaga Cardona, quienes sirven de inspiración principal para el índice y la metodología de trabajo central de este material.

Si este material llegó a ti por una vía no legal y te sirve de algo ayudándote a encontrar los conocimientos necesarios para buscarte el real efectivo, espero tus aportes con lo que nazca de tu corazón, los cuales serán bien recibidos, si lo compraste, no mandes “na” que por algo tu pagaste, aunque nada te impide que dejes caer par de moneditas de todos modos :D

[Paypal.Me/sgermosen](https://www.paypal.com/payto/sgermosen)

Mi GitHub y Pagina si les sirve de algo o desean contratarme.

www.Sgermosen.com

www.GitHub.com/sgermosen

Introducción

Sobre el autor.

Mi nombre es Starling Germosen, bueno, Ingeniero Germosen, como suelen poner los posers en sus facebook cuando se gradúan, no hagan eso, eso le resta mucho a su perfil profesional, lo importante en estos tiempos no es tu título, sino lo que tú has realizado o hasta dónde has llegado y más en este mundo de la tecnología.

Soy Master en Comercio Electrónico con titulación de la universidad APEC, fundador de Juegos Mentales Online, una plataforma que genera dinero compartiendo retos y desafíos para los perfiles sociales de ciertas empresas, con clientes como Malta Morena, Perfumuda y más.

CEO de MersyRD, una solución de expedientes médicos para clínicas, hospitales y doctores independientes.

Creador de Torneo Predicciones, una app móvil desarrollada en Xamarin con el fin de desafiar a tus amigos intentando acertar predicciones y ganar puntos en el proceso.

Entre otras aplicaciones que pueden auditar en GitHub.

Analista programador en Dirección General de Impuestos Internos.

Además de tener 12 años de experiencia en el desarrollo a la medida de soluciones informáticas, páginas web y consultoría para diversas instituciones y pequeñas empresas en PraySoft dominicana.

¿Qué Vamos a Ver?

En este curso vamos a trabajar con ASP .Net Core, a diferencia de lo que se había planteado originalmente, claro, esto es si ustedes así lo desean, puesto que las diferencias entre .net core y MVC, son tan ínfimas, que no tiene sentido enseñarles algo que ya tiene 2 años de desfase, además de que los conocimientos adquiridos en .netCore, ustedes lo pueden llevar al mundo MVC sin problemas, con la ventaja de que si desarrollan directo en .netCore, pueden tener un desarrollo multi-plataforma.

La única diferencia radical que tendríamos entre .netCore y MVC, serían las inyecciones de dependencias y las configuraciones, cuando lleguemos a ese tópico en los módulos finales, me comprometo a enseñarles cómo se hacen las configuraciones en ambos y las inyecciones de dependencia, ya que cuando ustedes salgan a la calle se van a topar con muchas aplicaciones hechas en MVC.

La orientación de este curso es enfocada a convertirnos en Fullstack developer o desarrollador completo, pues hay que ser realistas, en el mundo real las empresas pequeñas con las cuales usted puede conseguir una picota, no andan contratando un desarrollador para el Front-End y otro para el Back-End, normalmente quieren uno que meta mano en todo, si bien es cierto que es bueno especializarse, la idea de este curso es que usted salga de acá a meter mano y a ganar dinero, por lo tanto, aunque usted no se debería volver un master en diseño, al menos debería conocer:

- Cuales reglas básicas de usabilidad debe seguir una aplicación.
- Cuáles Frameworks existen para facilitar tu trabajo y hacerlo ver como algo profesional.

Y eso no es solo una mentalidad tercermundista de RD, eso pasa en todos lados del mundo, solo las grandes empresas contratan gente especializada y normalmente esas solo sub contratan el trabajo.

Pero, aun cuando usted caiga en un lugar que contrate gente especializada, ¿imagine que usted no sepa nada de lo que es Css? Una cosa es que usted no tenga la vena de mariconcito, pero si usted no sabe lo básico de estilo, no se va a entender nunca con el diseñador, o con el encargado de infraestructura, por eso la orientación del curso será a meter mano con todo lo que involucra la creación de un proyecto completo desde cero.

Ojo, este no es un curso de programación ni de C Sharp, a pesar de que la forma en que yo explico es muy básica y de hecho es la que me ha funcionado con mis estudiantes de clases particulares, es bueno que conozca al menos de manera básica lo que es C# y conocimientos medios de programación orientada a objetos.

Algunos de los tópicos que vamos a tocar son:

- Construyendo interfaces de usuario.
- Contrayendo APIs.

- Programación Orientada a Objetos de verdad, verdad.
- Arquitectura Limpia.
- Manipulación de datos con Entity Framework Core.
- Manipulación de objetos con Linq.
- Servicios de Azure.

Todo esto y muchos conceptos más, los iremos viendo y aprendiendo en el contexto de una aplicación del mundo real, partiendo desde un documento de requerimientos, extrayendo los casos de uso y el análisis de cómo implementar estos.

Prerrequisitos

- Conocimientos básicos de C#,
- Conocimientos básicos de POO,
- Conocimientos de modelado y estructura de datos.
- Visual Studio Community Igual o Superior al 2017.
- Net Core Mayor o igual a 2.2

Nosotros no vamos a profundizar en aquellas herramientas que no tengan una implicación directa con la creación de nuestro programa, pero debemos conocer la mayoría de ellas al menos por arribita, corresponderá a la tutoría personalizada la parte de profundización sobre algún tema x que no tenga que ver con el desarrollo de nuestra aplicación o investigarlo por su cuenta, sin embargo siéntase libre de preguntar que yo no muerdo y soy muy abierto a los comentarios y debates, si hay algo de la clase que no les gusta lo podemos cambiar o algo que se necesite reforzar lo podemos debatir.

Nuestra aplicación, a pesar de ser un app de consumo, asumo que ustedes conocen la diferencia entre aplicaciones de consumo y empresariales, la vamos a enfocar a cómo funciona el mercado en la calle, el cual es bastante distinto a como nos enseñaron en la universidad, nuestro proyecto final, será una especie de mini red social para coaching donde los coach, pueden subir sus eventos y la gente puede darle seguimiento, una especie de EventBride pero orientado al nicho de mercado del coaching a ver si nos lo compra Herbalife o Amway, así que no vayan a creer que haremos una aplicación que le haga competencia a Facebook, aunque los conocimientos que vamos a adquirir aquí, si son suficientes para hacer una red social en .net completa, pero no es el objetivo del curso, háganla ustedes si así lo desean, de igual forma los conocimientos y el modo de trabajo que vamos a emplear, te va a servir para cualquier tipo de aplicación web que te puedas imaginar.

En este curso te llevaremos de la mano sin importar si eres un desarrollador junior o novato que apenas está incursionando en el mundo de la programación, iremos desde lo más básico hasta lo más avanzado para que mejores tus habilidades de desarrollo.

Iniciaremos conociendo la plantilla básica de netCore, como está estructurada y cómo funciona el esqueleto, para posteriormente ir transformándola en nuestra aplicación.

La aplicación como mencione previamente es una red social para conectar a coachs con personas que les interesan esos temas, los coach, pueden registrar sus eventos próximos, los usuarios pueden ver todos los eventos e indicar que van a asistir a estos,

así como seguir al coach para que se les notifique de la creación o cancelación de algún evento futuro, las personas también pueden buscar eventos por coach, tipo de charla o ubicación, los eventos que el usuario marca se añaden al calendario.

¿Qué aprenderemos?

Para desarrollar nuestra aplicación veremos algunos elementos del desarrollo web moderno, tales como:

- Bootstrap
- Css
- Usabilidad
- Javascript
- Ef code fisrt
- RESTful APIs
- Seguridad
- Plain Old CLR Object
- Object-oriented design.

Y lo que vallamos necesitando en el camino según lo requiera la trama.

Nosotros vamos a iniciar con tópicos básicos, quiero que esa parte quede bien clara, por tanto, el código inicial no estará optimizado y será la vergüenza de los dioses del software, pero a medida que vayamos avanzando iremos evolucionando el mismo a uno más limpio, mantenible y elegante, que hable por sí mismo con solo verlo sin necesidad de tener que añadir comentarios adicionales,

práctica que se conoce como código suficientemente descriptivo o auto descriptivo.

Te garantizo que, con esta guía, vas a poder llevar tus habilidades en el desarrollo .net al próximo nivel,

Estructura del Curso.

La estructura del curso será impartida o se divide en tres partes: Fundamentos, tópicos avanzados, y finalmente arquitectura.

Fundamentos.

En la primera parte abarcaremos los tópicos tales como, ¿por dónde carajos arranco?

Por eso en esta parte iniciaremos con el documento de requerimientos, te enseñare a extraer los principales casos de uso, que tienen un mayor impacto en el desarrollo de nuestra aplicación y luego la construiremos paso a paso, el enfoque de esta parte es enseñarte la mentalidad del ingeniero de software, pensar como lo haría un ingeniero de software, como abordar un problema romperlo en problemas más pequeños y resolverlos paso a paso. De igual forma tocaremos los cimientos del desarrollo de software Backend and Front End, usando y aprendiendo en el camino.

Entity Framework Code First, Bootstrap, validaciones de datos, seguridad, aspectos artísticos de interfaz de usuario para crear interfaces amigables, etc.

Tópicos Avanzados.

En la segunda parte, implementaremos un servicio de notificaciones.

En esta parte haremos bastante énfasis en el diseño orientado a objetos, construir APIs y valernos de Bootstrap para construir aplicaciones modernas.

Arquitectura.

Y finalmente en la última parte veremos un poco de arquitectura.

¡¡¡¡Vamos a iniciar!!!!

Te recomiendo que aunque creas que sabes demasiado de lo básico, no te saltes directo a la siguiente parte, porque créeme que hay algunos trucos que quizá no conozcas y que te serán de mucha ayuda en tu desarrollo profesional, en la primera parte es donde explicamos todos los conceptos, por lo que en la segunda y tercera parte ya nos vamos a enfocar directamente en trabajar sin explicar mucho a no ser que sea algo nuevo que no hayamos implementado antes.

Otra cosa de vital importancia para el aprendizaje, no importa que tan básico consideres que sean los ejercicios, te recomiendo encarecidamente que intentes resolverlos por tu cuenta, sin ver la solución primero, intenta por tu cuenta y después consulta la respuesta, también toma en cuenta que no existe una sola forma

de hacer las cosas, puede que algunas de las respuestas que yo de a algún ejercicio no este 100% optimizada por el nivel en el que nos encontremos por lo que quizá el tuyo este mucho mejor, en programación, a pesar de que todo es cero y uno, un resultado se puede conseguir por diversos medios, uno más óptimos que otros, pero mientras estamos aprendiendo lo importante es hacerlo.

Que es .Net Core

ASP.NET Core es un nuevo Framework de código abierto y multiplataforma para la creación de aplicaciones modernas conectadas a Internet, como aplicaciones web y APIs Web, aunque por ahí viene también las aplicaciones Winforms, pero eso está en veremos.

Se diseñó para proporcionar un Framework de desarrollo optimizado para las aplicaciones que se implementan tanto en la nube como en servidores dedicados en las instalaciones del cliente.

Se pueden desarrollar y ejecutar aplicaciones ASP.NET Core en Windows, Mac y Linux.

ASP.NET Core puede ejecutarse sobre el Framework .NET completo o sobre .NET Core.

.NET Core es una nueva versión modular del Framework .NET que permite el uso multiplataforma de .NET. Es un subconjunto del

Framework .NET por lo que no tiene toda la funcionalidad del Framework completo, y puede emplearse para creación de aplicaciones web, de escritorio y móviles.

El uso del Framework completo nos permitirá poder añadir cualquier dependencia que necesitemos del Framework, pero perderemos todas las ventajas que tienen las aplicaciones .NET Core, tales como la multiplataforma, la mejora del rendimiento, el menor tamaño de las aplicaciones, etc.

¿Por qué utilizar ASP.NET Core en lugar de ASP.NET?

ASP.NET Core es un rediseño completo de ASP.NET. No es una actualización de ASP.NET 4, por lo que su arquitectura ha sido diseñada para resultar más ligera y modular.

ASP.NET Core no está basado en System.Web.dll que aportaba un exceso de funcionalidad. Se basa en un conjunto de paquetes NuGet granulares y bien factorizados. Esto te permite optimizar tu aplicación para incluir solo los paquetes NuGet que necesitas.

Beneficios de ASP.NET Core contra ASP.NET

Seguridad más estricta: Menor intercambio de información y rendimiento mejorado, ya que está formado por paquetes NuGet, lo que permite un modularidad total, de esta forma solo añadiremos los paquetes con la funcionalidad que necesitemos.

Una plataforma unificada para la creación de interfaz web y las APIs web.

Integración de los Frameworks modernos de cliente y flujos de trabajo de desarrollo.

Un sistema de configuración basado en la nube. Preparado para su integración de forma sencilla en entornos en la nube.

Inyección de dependencias incorporada.

Las peticiones HTTP se procesan siguiendo un flujo que puede ser modificado de forma modular para adaptarse a nuestras necesidades y que nos permite poder controlar el procesamiento de las peticiones HTTP en nuestra aplicación.

Capacidad para alojar en IIS u otros servidores web como Apache. o self-host en su propio proceso.

Nuevas herramientas que simplifican el desarrollo web moderno.

Crea y ejecuta aplicaciones multiplataforma ASP.NET Core en Windows, Mac y Linux.

De código abierto y orientado a la comunidad.

Sistema de Control de Versiones

Sistemas de control de versiones, la idea de estos es que tu tengas tu proyecto en el servidor siempre resguardado de cualquier imprevisto, cuando vayas a trabajar algún cambio, la idea es que bajes los cambios más recientes a una carpeta de trabajo en tu computadora, cada vez que tienes algo que funciona o completa una responsabilidad que se te haya delegado en el equipo, tú haces un commit al repositorio, para que los cambios que están en tu máquina se combinen con los que están en el servidor, el repositorio no es más que una mini base de datos, que guarda el historial de todo tipo de cambios que se hayan hecho al código o los archivos que sean parte del proyecto. Los beneficios de trabajar siempre ordenados y con un control de versiones es que podemos devolvemos en cualquier momento a cualquier estado de la aplicación, como nuestra máquina del tiempo con más coherencia que las líneas temporales de avengers. Los controles de versiones nos sirven para comparar elementos o cambios que hayan realizado otros compañeros del equipo, descartar cambios entre otras funcionalidades chulas que iremos abarcando más adelante. Existen múltiples controles de versiones, están tfs, SourceSafe para la gente que desarrollaba para entornos Microsoft, scm, SubVersion que fue uno de los más famosos en el mundo anti microsoft, hasta que Linus Torvalds creó Git en 2005, el cual ha reemplazado a todos los demás, no confundir sistema de control de versiones con programa de control de versiones, el sistema es la forma en que funciona algo de manera conceptual, el programa es el que permite llevar esa conceptualización al mundo de la informática.

Dentro de los programas para control de versiones con la metodología Git, podemos encontrar GitLab, que tiene una versión en línea como descargable para montar en tu propio servidor, Team Foundation Server o Azure DevOps, y GitHub.

GitHub

Vamos a conocer al Octu-Cat, el gato pulpo, github es una plataforma colaborativa que permite compartir nuestro código fuente con la comunidad de desarrolladores y sirve a su vez para hacer BAM, así como que las empresas puedan ver tus proyectos, como trabajas, como codeas y de esa forma sumes puntos en la comunidad de desarrollo, también github te permite mantener el tracking del código de tu aplicación, ya sea de manera privada o pública, se vende como la red social de los desarrolladores, pero nunca fue así, aunque haga muchos esfuerzos en convertirla en tal. Es propiedad de Microsoft en la actualidad.

Github o cualquier otro control de versiones nos evita el tema de estar guardando en el disco duro nuestros códigos fuentes y que de repente pase algo que haga que se pierda, o estar con el can de guardar proyecto uno, proyecto uno final, proyecto uno final este si o final final.

Vamos a ir a GitHub.Com, Creamos un usuario, Creamos un Repositorio sea público o privado. Dentro del Repositorio creamos un proyecto llamado ThuRealProyecto. Añadimos columnas donde guardaremos el estado de las tarjetas o asignaciones, estas son para poder tener una visión del estado de nuestro proyecto.

Creamos Pendiente, escogemos la plantilla de ToDo escogemos como se van a trabajar las nuevas tarjetas, en mi caso voy a especificar que desde que yo creo una venga a Pending, porque como soy un único desarrollador, si yo mismo fui quien la creo, es porque las voy a trabajar, en una organización las cosas no funcionan así, pues el planning lo hace otro, pero en nuestro caso, lo vamos a trabajar de esta forma. De igual manera si por alguna razón se re abre algún pendiente, quiero que se mueva acá.

Cuando nosotros trabajamos en ambientes colaborativos, se da algo que se llama Pull Request, eso lo veremos más adelante, por lo pronto, ignoraremos estos dos apartados, pues yo no quiero que me llenen de posibles basuras mi proyecto.

Luego procedemos a crear la Columna Working donde vamos a mover las actividades cuando las estemos trabajando. Escogeremos la plantilla In Progress, acá no modificaremos nada más, podríamos escoger que se muevan acá las actividades que sean revisadas, pero eso no es práctico, pues lo correcto es que el desarrollador especifique cuando inicio a trabajar en algo, no que me digan wey pana, pero tú tienes 4 días atrasado en tal tarea y tu estés aéreo al respecto porque estabas trabajando en otro proyecto.

Y creamos la última columna que llamaremos Finalizado donde escogeremos la plantilla Done, seleccionamos que se muevan aquí las tareas cuando sean cerradas y no haremos nada con las que vengan de Pull Request.

Podemos añadir más columnas según sea necesario, pero con esas son suficientes para nuestro objetivo.

Posterior a eso vamos a proceder a descargar GitHub Desktop e instalarlo, también podemos instalar la extensión de github para visual studio por la opción de tools, extentions si no queremos usar programas adicionales, yo prefiero usar la versión de escritorio que me sirve de paso para otros controles de versiones como GitLab, el cual antes me permitía par de chulerías que GitHub no tenia, pero que ahora mismo aunque gitlab tiene muchas más herramientas que github yo no uso más que las ramas, los estados de trabajo y las tareas y eso ya se puede hacer por github, pero igual prefiero el programa.

File New Project por fin!!!!

Vamos ahora a crear nuestro nuevo proyecto, primero que nada, vamos a clonar nuestro repositorio en un folder cualquiera de nuestro disco duro.

Procedemos a crear el nuevo proyecto dentro del folder.

Una vez creado lo que vamos a hacer es sincronizar los cambios de lo que tenemos local con lo que está en el servidor, para evitar que se nos rompa el código y no podamos devolvemos en caso de hacer un disparate, los commits se deben hacer cada vez que tengamos un mínimo de funcionalidad resuelta.

Los archivos que podemos ver con el signo de + en verde son archivos que se están añadiendo desde el ultimo commit o más bien si lo comparamos con la rama en la que nos encontramos para lo que está en el server, es decir, ese archivo está en nuestra carpeta, pero no está en el servidor.

Los comentarios son etiquetas que nos permiten entender en el futuro que estábamos trabajando en el momento en que decidimos crear este commit.

Una de las herramientas que usaremos de manera opcional es reSharper, la cual pueden conseguir con una licencia de estudiante gratis por un año y esta te ayudara bastante en la optimización de código, pero sin embargo esta herramienta no es imprescindible para nada y más ahora que code lense es gratuito con visual studio community 2019

También vamos a usar una extensión gratuita, aunque quizá no la vamos a fondo, llamada web essentials, la cual nos permite hacer modificaciones en vivo en el HTML del proyecto los cuales se reflejan de forma automática en el código de nuestro proyecto, es decir, nos permite hacer un link entre lo que está en el navegador y el proyecto que estamos depurando y hacer ediciones directo en el navegador para que modifiquen el archivo.

Para eso vamos a extensiones y actualizaciones nos vamos a online y escribimos web essentials.

También de manera opcional Productivity Power Tool

Resumen

Vamos ahora a ver lo que aprendimos en esta lección introductoria.

- Conocimos al profesor.
- Vimos cual será la forma de trabajo y los conocimientos que vamos a adquirir al finalizar el curso, así como las herramientas que vamos a implementar y el proyecto que trabajaremos.
- Aprendimos lo básico acerca de .NetCore y las diferencias entre él y su predecesor, de igual forma vimos las diferentes plantillas que hay disponible de manera predeterminada y las diferencias entre estas.
- Aprendimos sobre lo que es un control de versiones, su importancia y comenzamos a usar un programa llamado GitHub que nos permite trabajar con esta metodología.

Extrayendo los principales casos de uso del requerimiento.

Una de las preguntas que más se hacen los desarrolladores cuando están iniciando es ¿Por dónde empiezo? Normalmente siempre arrancamos por el código, lo cual es un error garrafal.

Es por eso que es lo primero que vamos a cubrir en esta sección. Quiero que nos enfoquemos en un documento de requerimientos y luego vamos a ver los siguientes pasos que debemos de dar.

Vamos a extraer los principales casos de uso, que son los que tendrán mayor impacto en el diseño de tu aplicación, en lugar de implementar cada característica de principio a fin y totalmente pulido, nosotros vamos a implementar estos casos principales en primer lugar, porque implementar primero estos casos son los que nos darán una idea de los desafíos venideros y son los que nos permitirán tener un prototipo medianamente funcional, así como ver los principales retos involucrados con nuestro proyecto ahora echemos un vistazo al documento de requerimientos.

Típicamente el desarrollo de software inicia con un documento de requerimientos. A veces el documento puede ser una pequeña página del tamaño de un post it o puede ser un extenso documento completamente detallado de 500 páginas, (que nadie va a leer), independientemente de cómo te sea entregado estos, tu labor como ingeniero de software es extraer cuales son los casos de uso con los cuales puede trabajar un desarrollador o en este particular

tú, en este caso veremos el documento de requisitos para nuestro proyecto.

“Coaching events es una mini red social que permite a coachs comunicarse con sus seguidores para que estos participen en sus conferencias, permite crear y listar sus charlas, las cuales para ser creadas requieren una fecha, tipo, y la ubicación.

Un coach debe tener una página donde pueda editar o eliminar o añadir un evento a la lista.

Los seguidores y amantes de las conferencias pueden seguir a sus coachs favoritos y recibir notificaciones cuando ocurra un cambio en algún evento.

Los usuarios deben tener la posibilidad de ver los próximos eventos o buscarlos ya sea por coach, tipo o ubicación, así como ver el calendario en detalle y añadirlo a su propio calendario de actividades.

Cuando el usuario sigue a sus coach favoritos ellos deben poder ver en su pantalla de inicio sus próximos eventos.”

Este es básicamente un documento de requerimientos, puede ser más extenso o perfeccionado, pero con este podemos ir metiendo mano por ahora, no podemos desperdiciar mucho tiempo en esto, después de todo normalmente este documento nos lo dan a nosotros, no lo redactamos, lo verdaderamente importante para el desarrollador es saber extraer sus casos de uso.

Estos se expresan con pocas palabras, por ejemplo, en el primer párrafo podemos identificar dos casos de uso: Crear una cuenta y crear un evento.

Expresar en pocas palabras los casos de uso ayuda a un mayor entendimiento del proceso y simplifica la comunicación, así como reducir los problemas de interpretación que traen consigo esos documentos extensos con un montón de detalles, los detalles pueden venir después cuando nos acerquemos a la fase de implementación, vamos ahora a tomar unos minutos a extraer todos los casos de uso y en breve compararemos tu solución con la mía.

Autenticación

Módulo de registro, módulo de acceso, opción para cerrar sesión, módulo de cambio de contraseña, edición de perfil.

Acá podemos ver algo interesante, a pesar de que en el documento de requerimientos no se me dijo que debía crear una pantalla de acceso, es obvio que hace parte de nuestro proyecto, por lo que muchas veces como ingenieros, corresponde a nosotros agregar esos casos o pasos, sin los cuales no podemos realizar los procesos que se nos están pidiendo, y por eso es que debemos valernos de nuestra experiencia para que al momento de que un emprendedor o empresa nos venga con esos requerimientos ambiguos, nosotros podamos cobrar lo justo por nuestro trabajo, ya que al tener experiencia sabemos todas las aristas adicionales que debemos emplear para poder cumplir con los requerimientos

Por esto, los documentos de requerimientos, deben ser vistos como un entendimiento de alto nivel acerca de que es tu proyecto a nivel macro.

A medida que vayas avanzando, vas a usar tu propio criterio o mantenerte en constante comunicación con el cliente a medida que vas construyendo tu sistema para llenar esos vacíos existenciales y dudas que tengas en este caso use mi criterio para determinar que, si tengo un registro, debo tener un login, log out, etc.

La buena noticia con respecto a la primera parte de los casos de uso, es que la plantilla por defecto de visual Studio ASP viene con una forma simple de construir tu sistema de autenticación y registro, si tu deseas puedes construir el tuyo propio, pero, ¿en verdad vale el esfuerzo extra que le vas a poner? ¿En verdad crees que vas a construir algo mejor que Microsoft estando al nivel en el que estas de desarrollo? Si la respuesta es no, perfecto, usa el que viene por defecto y enfoquémonos en cosas más importantes, puesto que hasta en el mundo real la mayoría de las grandes empresas están delegando la seguridad en sistemas de autenticación externos, ya muchas apps permiten la autenticación con Facebook o Twitter, por poner un ejemplo, así que saca de tu mente la necesidad de querer hacer todo desde cero.

Eventos

En la parte de eventos, he podido sacar lo siguiente: añadir un evento, listar mis próximos eventos, editar y eliminar un evento,

ver todos los eventos próximos, Buscar eventos por distintos filtros, ver los detalles de un evento.

Calendario

Añadir eventos a un calendario, remover eventos de tu calendario, ver los eventos a los que marqué que voy a asistir.

A pesar de que el tema de añadir los eventos no eran parte de los requerimientos en si nuestra experiencia como desarrolladores siempre deben permitirnos agregar valor a las propuestas ambiguas que tengan nuestros clientes, eso es lo que marca la diferencia entre un profesional y un picapollero, sin denigrar a los que practican esta labor.

Tu experiencia debe ayudarte a llenar esos vacíos que el cliente no especificó, pero que tú sabes que él va a necesitar. Cuando estas en equipos de trabajo grandes, donde tú no eres otra cosa que una pieza más del equipo, no es muy bueno ponerse más creativo de la cuenta, en especial en latino américa, donde el jefe puede pensar que tú le quieres quitar el puesto o que ser más eficiente que el promedio puede hacer que el equipo no funcione de la manera correcta por la falta de sinergia, pero cuando estas en la calle bandeándotela como puedas, debes siempre dar lo mejor de ti.

Seguimiento

Seguir un coach, dejar de seguir un coach, ver a quien estoy siguiendo, ver los eventos en mi feed de a quienes estoy siguiendo.

De nuevo acá vemos cosas que no estaban en el documento de requerimientos original, pero como somos super analistas nosotros pudimos agregar estos.

Vamos ahora a Github y vamos a añadir todos estos ítems a nuestro proyecto. Yo solo voy a agregar dos, ahorita agrego los demás.

Tenemos dos formas de registrar, podemos irnos directamente a Issues o problemáticas, y registrar cada una de estas actividades como una problemática, pero esta opción es más viable cuando ya tenemos un sistema en producción, y recibimos colaboración de otros, en mi caso voy a ir al proyecto me situó en la pestaña o columna Pending, le damos a añadir y ponemos la nota que deseamos, desde acá al momento de programarlas las convertiré en problemática a resolver en el momento en que toque trabajarla, las problemáticas no son solo problemas de aplicativo, pueden ser documentación, solicitud de ayuda, preguntas, etc.

Dependencias

Una vez hayamos extraído los casos de uso, toca verificar las dependencias e introducirlas en nuestra agenda de trabajo.

Por poner un caso, volviendo al slide anterior, no podemos dejar de seguir un coach si primero no lo hemos seguido, ¿Cierto?

No sé si recuerdan de la universidad en investigación de operaciones la famosa ruta crítica, creo que fue en esa materia, buh, es un concepto similar el que se usa para determinar el orden de dependencia. Otro ejemplo para poder seguir a un coach primero debemos ver sus eventos porque al momento de ver los detalles de los eventos es donde vamos a mostrar al coach y es ahí donde el usuario podrá seguirlo, este orden no tiene que ser así, pero es la forma en que lo encuentro más lógico.

Las dependencias se representan con una flecha punteada señalando al padre de la funcionalidad de la que se depende.

Lo que quiere decir que, por ejemplo, tenemos que implementar primero la función de ver eventos, Luego seguir un coach y finalmente para cerrar el ciclo, dejar de seguir.

Ahora la tarea que te toca es ver cuáles son las dependencias de trabajo entre los demás casos que planteamos exceptuando el de la parte de autenticación, solo de las relacionadas con el core principal de la aplicación.

Mantenlo simple, imagina o intenta lograr una segregación en la que cada caso no tenga más de una dependencia o dos como máximo.

Una vez que tenemos todas las dependencias establecidas, debemos iniciar por aquel que no tiene ninguna dependencia, pero que dé el dependen muchos más. De esa forma podremos tener los órdenes de ejecución de nuestro proyecto.

Vamos a tomarnos unos minutitos para hacer esta tareita, en lo que yo las voy agregando todas al github.

Acá te muestro mi solución.

En primer lugar pondré añadir un evento, si el coach no tiene la habilidad de añadir un evento, no hay nada que se pueda hacer en el sistema, nadie puede buscar, nadie puede añadir a favorito etc, Una vez implementado este, podríamos implementar la opción de ver los próximos eventos Y a partir de ahí podríamos editar o remover eventos, De igual forma una vez que tenemos la implementación de añadir eventos, podemos ver todos los eventos próximos, no solo los míos Y cuando tenemos este podríamos trabajar en añadirlos al calendario.

Posterior a eso podemos crear una página donde ver aquellos eventos a los que estoy asistiendo y a su vez removerlos del calendario. Volviendo a la parte donde tenemos implementado todos los eventos, podemos ir a los detalles y de esa forma seguir un coach así que consecuentemente podríamos crear una página para los usuarios para ver quienes está siguiendo este y en esa

página tener la posibilidad de dejar de seguir un coach además cuando sigue un coach puede ver sus eventos en su propio feed y también cuando tenemos la forma de ver el listado de todos los eventos podemos implementar también la búsqueda de eventos o ver el detalle de un evento en concreto.

Si tu solución se parece a la mía, es genial, pero si no, no te preocupes, existen muchas formas de asimilar el todo de un proyecto, así que no te preocupes, quizá la solución tuya es mucho mejor que la mía de hecho.

Ahora es momento de identificar nuestra ruta crítica estos casos de uso principales son los que le dan forma al dominio de nuestra aplicación este es el que involucra la captura de data y cambia el estado de la aplicación así que, de esta lista, ¿cuál crees que es el caso principal?

Añadir el evento evidentemente cambia el estado por completo de nuestra aplicación.

Así que por supuesto es nuestro inicio obligatorio en la siguiente columna no tenemos ningún caso de uso que cambien el estado de nuestra aplicación pues no son más que casos de reportería, en la tercera columna si tenemos algunos elementos que cambian el estado de la aplicación, como editar el evento removerlo añadirlo al calendario y seguir un coach, pero editar no es un caso primordial porque es muy similar a añadir un evento por lo que añadir o editar tienen el mismo impacto en nuestro dominio, añadir eventos al calendario sin embargo si requiere extender el modelo

de domino, para cada usuario necesitamos llevar un histórico de los eventos a los que ha marcado que asistirá así que este es otro caso principal y lo mismo aplica a seguir un coach para cada usuario necesitamos llevar un histórico de cada coach a los que el sigue.

En la cuarta columna tampoco tenemos casos principales pues todos no son más que reportes remover eventos del calendario y dejar de seguir un coach es similar a seguirlo y añadirlo al calendario por lo tanto no son más que extensiones. Así que tenemos que los más importantes son los siguientes:

Ahora bien, ¿qué pasa acá?, que el momento de escoger estos como casos principales, si deseamos mostrar nuestro avance al cliente, no estaríamos mostrando nada significativo, pues no hemos escogido ningún modo de reportería y no creo que el cliente encuentre chulo que solo se le muestre la base de datos por debajo.

En mi caso escogeré todos los próximos eventos, como actividad critica adicional, porque a partir de esta podemos ir mostrando un avance significativo del aplicativo, Pero, de igual forma en la cuarta columna voy a escoger uno de los casos mostrados ahí a los fines de que el cliente se vaya enamorando de la aplicación, pues aunque nosotros tengamos un enorme trabajo por detrás, si el usuario no ve por delante algo que valga la pena, se va a decepcionar y quizá no quiera pagarnos el adelanto o peor aún decida cancelar el proyecto.

Aunque no son casos verdaderamente principales, son casos de soporte, por eso me veo obligado a escogerlos. Recuerda siempre planear la pieza mínima de funcionalidad.

Posteriormente vamos a pasar mis casos principales a working, aclarar que en el mundo real hay metodologías más efectivas de trabajar este tema de los requerimientos, metodologías ágiles, que no tienen nada que ver con velocidad, y normalmente se trabajan en entregables de dos semanas máximo, pero acá solo conceptualizamos esto como una forma organizada de trabajar.

Iniciando las iteraciones

Ok, ya tenemos nuestros casos principales o el flujo lógico de desarrollo de nuestra aplicación, Ahora nos toca ponerlos en nuestro planeador para mantener un histórico de nuestro trabajo e ir trabajando de manera ordenada, lo cual vende mucho cuando estamos tratando de mostrar al mundo que sabemos o que somos duros.

Puedes usar cualquier herramienta como Microsoft Project que te permita llevar un histórico de los trabajos que vayas realizando en que estas trabajando actualmente y que aún falta por hacer, pero en el curso usaremos esta metodología.

Ahora nos toca esquematizar de manera básica cómo será la experiencia de usuario para nuestra aplicación a veces estas trabajando en un equipo donde puedes tener una persona dedicada al diseño pero la mayor parte del tiempo te toca trabajarlo

a ti, lo que me interesa que se te quede de esta lección es que aprendas a tener el habito de tomar un pedazo de papel y dibujes como debe quedar tu flujo de trabajo dentro del aplicativo cuando este esté funcionando, que tu dibujo diga como el usuario va a navegar de una página a otra y como va a interactuar con los elementos que tiene en la página, como va a lucir la barra de navegación, etc. Es mucho más fácil dibujar esto en un pedazo de papel que ir directo al HTML y al CSS a tirar líneas que vamos a desbaratar una y otra vez, con eso en mente vamos a iniciar. No tires código sin un sketch.

Debemos determinar cómo los usuarios van a navegar por la aplicación y recuerda que esta experiencia de usuario no tiene que ser perfecta solo enfócate en la interacción así que no intentes desglosar como va a lucir toda la aplicación completa con todas sus características, sino que ve construyendo sobre la marcha así que mantenlo simple a medida que vayamos evolucionando la aplicación la iremos haciendo mejor, así que lo primero que necesitamos es darle al usuario la habilidad de poder crear un evento necesitamos por lo tanto un formulario donde pueda añadir los detalles del evento la pregunta es, ¿que debe pasar cuando el usuario haga clic en el botón? lo lógico sería redireccionarlo a la lista de los eventos para que pueda editar o eliminar si cometió un error pero no tenemos ese caso en esta iteración, nosotros tenemos ver todos los eventos, no solo los míos así que temporalmente lo vamos a redirigir al listado de todos los eventos que por ahora estará en la página principal.

En esta lista al frente de cada evento pondremos un botón que dirá “deseo ir” con un signo de interrogación cuando el usuario haga clic en este botón el color y el texto deben cambiar y este será

añadir a la lista de eventos a los cuales yo deseo ir es similar a como seguimos personas en Twitter ahora, una vez el usuario lo haya añadido a su calendario debe tener la opción de verlo así que podemos dedicar una página y poner un link en la barra de navegación, de forma similar al frente de cada coach que está en cada evento tendremos un link o botón llamado seguir cuando hagamos clic en ese botón, cambiara de color y de nombre y el artista será añadido a la lista de quienes yo sigo añadiremos otro link en la barra de navegación llamado “a quien sigo” que llevará al usuario a la lista del coach que él está siguiendo.

Nuestra barra de navegación va a lucir algo como esto, tendremos el nombre de nuestra app que nos llevará al inicio y un link de login que será reemplazado por el nombre una vez nos hayamos autenticado. Cuando haga clic en ese link debe aparecer un menú que solo estará disponible si el usuario este logueado.

Acá tendremos dos links el de ver los eventos a los que marque que planeo asistir y el de las personas a las que sigo, así como también un link adicional para salir del sistema o cerrar sesión.

MVC

Para trabajar nuestra solución nosotros vamos a utilizar el patrón MVC, que no es más que un acrónimo que significa Modelo-Vista-Controlador, este no fue creado por Microsoft, es un patrón que tiene un tiempo cómodo incluso antes de que existiera la web, sin embargo, es una metodología bastante practica a la hora de desarrollar y por eso ha tomado bastante auge, debido a la adopción por parte de diversos frameworks de este modelo como eje central de su modo de funcionamiento.

Es bien sabido que en programación existe una cosa que es lo que el usuario ve, lo que está en la pantalla, donde el usuario escribe o hace clics, esto no es más que la vista, el UI, el entorno gráfico, la vista es lo que se muestra al usuario mediante el navegador.

Luego tenemos el controlador, que podemos verlo como una especie de code behind, para los desarrolladores de la vieja escuela, con ciertas diferencias que veremos más adelante, el controlador es el que se encarga de manipular los datos ya sea suministrándolo o ejecutando las acciones que el usuario haya enviado desde la vista, el controlador es el enlace entre el modelo y la vista, es el que toma el control cada vez que se marca un verbo en el navegador sobre una vista. Lo de los verbos lo veremos más tarde.

El modelo es la representación abstracta de las entidades de datos, es lo que usamos de base para trabajar y manipular datos.

La idea del modelo, vista, controlador es separar y segregar la responsabilidad de cada cosa para evitar el código espagueti, que no es más que ese código que esta todo emburujado y que no hay forma de entrarle ni por delante con vaselina, así como hacer las apps lo más mantenible y testeable posible.

Un repaso breve de programación orientada a objetos

Una variable no es más que una cajita de información donde introducimos un valor que debe ser del tipo que le hayamos especificado a la variable, en un cover de iphone no podemos meter una licuadora.

Los tipos de datos son los que me permiten saber qué tipo de información puedo almacenar en las cajitas, si son números, “float”, “double” o “decimal”, así como “int”, si es letras (cadenas de caracteres) “string”, si son verdadero o falso “boolean”, si son fechas “DateTime”.

Asumo que ustedes saben que para la computadora todos caracteres incluyendo los números no son más que dibujitos que ella debe interpretar y para poder interpretarlos esta necesita que se le asigne el tipo de datos para poder diferenciar un 1 de un “1”.

Un objeto no es más que cualquier elemento sobre el cual puedo interactuar, normalmente son las representaciones abstractas de las cosas del mundo real, abstracto podemos verlo como una representación imaginaria de ese algo, mediante 0 y 1.

Un objeto en C# es una clase, la cual yo le voy a añadir atributos, y como dato curioso todo en c# es un objeto, incluyendo los tipos de datos primitivos.

Los atributos son esas características que debemos añadir a nuestro objeto para que sea ese objeto, es decir, es lo que hace que la cosa sea esa cosa que queremos representar, Ejemplo, si quiero representar un Potémon, el potemon, tiene ciertas características que lo hacen ser un potemon, como por ejemplo, quiero representar a pitachu, pitachu tiene cola, cachetes, ojos, boca, estas son sus características, estos son sus atributos; En el caso de un paciente, por ejemplo, tenemos que dentro de sus características están el record, el nombre, el apellido, el tipo de sangre, etc. Empleado de igual forma tenemos código de empleado, nombre, cedula, etc.

La herencia no es más que yo heredar características de mi padre de forma implícita, lo cual hace que yo no necesite implementar esas características, ejemplo yo tengo un Potémon que se llama Pitachu y otro que se llama vamo a' calmano, ambos son Potémon, por lo cual, yo puedo perfectamente crear una clase llamada Potémon, con características base que todos los Potémons tengan, tales como: Nombre, Tipo1, Tipo2, Ataque, Defensa, Velocidad y Puntos de vida.

En el caso de un paciente, podríamos tener una clase padre que se llame Persona y ese padre tenga Nombre, Apellido y Cedula.

Luego tendríamos los hijos que si le especificamos que son nuestros hijos entonces no necesitamos repetir esas características adicionales.

El Potémon Pitachu hereda de la clase base Potémon y vamos a' calmano también, por lo que solo cabría añadir esas características únicas que hacen que Pitachu sea un Pitachu, como cola, cachetes rojos, etc, y en Encuero añadiríamos otras como caparazón, color azul, etc.

En el caso de pacientes y empleados seria solo añadir el Récord y el tipo de sangre si es paciente, y como empleado el salario.

Base de Datos 101

Tabla no es más que la representación de la entidad, es decir, aquello que queremos representar del mundo real o aquello que queremos conceptualizar mediante un objeto de base de datos.

Campos no son más que esos atributos o características de la cual deseamos almacenar información.

Los registros son la información que insertamos sobre la entidad que representa la tabla.

Llaves, son las que permiten establecer relaciones entre tablas, constan de dos vías por lo regular, una llave primaria que es el identificador único que no puede repetirse, (por algo es único), y

una llave foránea que es la que apunta hacia una llave primaria de otra tabla.

Las relaciones son las que me permiten establecer interconexiones entre las informaciones que deseo saber de una tabla pero que la información adicional de ese dato se encuentra en otra tabla. Existen relaciones de uno a uno, uno a mucho, mucho a mucho (no recomendado y técnicamente no soportado por ef).

Recuerden que no tiene sentido para fines prácticos tener una entidad persona con datos que no siempre van a ser llenados, por eso se realiza una normalización, recordando que normalización no es más que hacer que nuestra base de datos o más bien las tablas de nuestra base de datos, contenga la menor cantidad de incongruencia y datos repetidos o nulos, en una normalización muy estricta, campos como el teléfono, correo y dirección van en tablas separadas, pero ese no es el tema.

Entity Framework Code First

Nosotros vamos a trabajar con el modelo de trabajo code first, para construir nuestro modelo de dominio, el modelo de dominio no es más que la representación mediante clases, de las entidades de nuestra base de datos.

Usaremos migraciones para generar la base de datos y así poder llenarla con algo de data, lo primero que tendríamos que hacer si estuviéramos trabajando en asp.net framework, es habilitar las

migraciones, pero como eso en el mundo net core no pasa, nos saltamos este paso.

En framework, para habilitar las migraciones vamos a Tools, Nuget Manager, luego Package Manager Console, y escribimos el comando.

`Enable-migrations`

Este comando se ejecuta una sola vez en la vida de nuestro proyecto.

La conceptualización de las migraciones con la metodología code first, es que creemos una migración por cada cambio que afecte el modelo de nuestra entidad y luego ese cambio va a correr en la base de datos, entity framework va a actualizar el schema de nuestra base de datos de esa forma no tenemos que ir a SQL y actualizar de forma manual nuestras tablas, además de que te permite tener un histórico y control real de todos los cambios que ha sufrido la base de datos.

Algo a tomar en cuenta, yo soy amante de las pequeñas migraciones, no importa cuán insignificante sea el cambio que le realice al modelo, crea una migración. De hecho, esa es una de las razones por las cuales algunos desarrolladores odian Code First y prefieren Database First, es porque no son organizados con sus cambios, entonces cuando tienen que darle para atrás a algo, rompen una estructura completa de trabajo. Así que siempre ten en mente, pequeño cambio, pequeña migración.

Trabajando con nuestro primer modelo.

Vamos a comenzar entonces a trabajar sobre nuestro primer modelo.

En la carpeta modelos, vamos a crear una clase llamada Events.

¿Qué propiedades deberíamos añadir a nuestra entidad?, basándonos en nuestro documento de requerimientos, sabemos que necesitamos capturar **quien** está creando el evento, **cuando**, **dónde** y que **tipo** de evento es, pues iniciemos con el quien crea el evento, por lo que, el primer dato que necesitamos es el quien, podríamos usar un campo que se llame nombre del creador, pero esto no es nada practico, lo correcto es que para nosotros satisfacer esta cuestionante necesitemos asociar está a otra clase que represente a nuestro usuario en nuestra aplicación, nosotros podemos perfectamente crear una clase de usuarios, y comenzar a meterle toda nuestra lógica super mega vacana que llevamos años trabajando y a la cual le hemos puesto mucho cariño y amor, o mandamos todo eso al carajo y trabajamos con una que hasta ahora no le han encontrado baches de seguridad, en nuestro caso, vamos a agregar Identity a nuestro proyecto.

Eso se hace por clic derecho a la aplicación, luego añadir nuevo Scaffolded ítem, después Identity, escogemos lo que queremos que haga nuestra app, escogemos el DataContext, que es el contexto de datos, es como el enlace entre los objetos del lado de la aplicación y los objetos del lado de la base de datos, a mí me

gusta llamarlo DataContext, pero usted lo puede llamar como entienda, siempre que lleve la palabra Context, para que usted u otro desarrollador no se pierda cuando este leyendo su código, como no tenemos datacontext, debemos crearlo, nombramos la clase que él va a usar como esqueleto para albergar nuestro usuario, y finalmente le damos a aceptar, si queremos que la data se almacene en Sqlite, podemos hacerlo desde aquí.

Acá nos sale un Readme, algo que ustedes deben hacer siempre que les salte un Readme es leerlo, sabemos que en la práctica no lo hacemos, yo tampoco lo hago, pero en este caso particular, si debemos hacerlo para quitarnos algunos dolores de cabeza con los que yo me tope durante noches buscando en youtube para poner a funcionar esa vaina, que siempre estuvo en el readme.

Lo primero que él nos plantea es que todas las cosas que escogimos se crearon en un área llamado Identity.

Las áreas son pequeños entornos que me permiten separar múltiples aplicaciones que convergen dentro de un gran sistema, por ejemplo, tengo el sistema que se llama “Banco Solidario Sb”, donde tengo un área para préstamos, donde estén todas las clases, módulos, pantallas y Helpers para préstamos, pero tengo otra área para ahorros, y otra para certificados; Sin embargo esta práctica no es muy recomendable, lo correcto es que estos sistemas estén separados en micro servicios, aunque como te digo una cosa te digo la otra, los sistemas monolíticos, son lo que más ustedes van a ver en la calle, así que, aprender a usar las áreas te puede dar un plus, para que la gente te tilde de que sabes, o que trabajas organizado. De hecho, entre programadores .net, es hasta

cierto punto mal visto la separación de una aplicación en múltiples apps pequeñas, eso es un eterno debate de nunca acabar, un dato, cuando usted intenta hacer Scaffolding, siempre debe tener un código que este compilando porque de lo contrario le va a arrojar errores que no dicen nada.

Seguimos leyendo, las configuraciones están en ese archivo en esa ruta, vamos para allá a ver que lo que con que lo que. Miren que cosas interesantes podemos ver por aquí podemos ver que implementa una interfaz que vamos a ver después que son esto y vemos que configura el servicio diciéndole que va a usar este datacontext cada vez que sea llamado, que dentro de las opciones a usar será SQL Server, y que la conexión la va a hilar de una cadena que llama DataCotextConnection, vamos a ver de donde el saca esa cadena usando el buscador que se habilita presionando CTRL + F, aquí podemos ver que él me va a crear en local db una base de datos llamada de esta forma, que usa esta configuración, aquí nosotros podemos cambiar esto por supuesto.

Si seguimos viendo el archivo podemos ver que el usuario de identidad que va a estar logueado se va a basar en ese esqueleto.

Veamos ahora el esqueleto, podemos ver que no es más que una implementación de otra clase llamada IdentityUser, está heredando, es decir, recibiendo todos los atributos de su padre, si nos vamos a ver que tiene esta clase vemos que hereda de identityuser pero el que recibe como parámetro base un string y si le damos para allá podemos ver que tenemos todos estos campos por defecto que ya tenemos disponible para trabajar con nuestro usuario, estos hacen parte de las clases del FrameWork por lo que

no podemos modificarlas, podríamos sobrescribirlas, si así lo deseamos, para extenderlas o excluirlas, pero a eso no se le pone la mano, pao, pao, jum!!!

Seguimos con el readme, nos dice que si habíamos configurado antes el Identity en otro lugar, debemos removerlo y nos dice que sigamos algunos pasos para poder usar las interfaces de usuario o las vistas, porque nosotros podemos escoger el scaffolding para que nos cree el coco, pero nosotros deseamos implementar nuestras propias vistas y usar esas solo como base tipo api o que se yo, no le veo otra utilidad, para resolver este impase debemos irnos a nuestro archivo de configuraciones, que es nuestro Startup.cs y le especificamos esa línea que nos indica el texto, si nos damos cuenta, la línea ya está, porque las aplicaciones MVC por defecto la necesitan, recuerden que habíamos planteado que las aplicaciones netcore no eran más que una pinche consolé application, por lo que el scaffolding no sabe originalmente a que nosotros le estamos añadiendo identity, pero como al momento de escoger la plantilla escogimos web y web necesita el aditamento de poder trabajar con vistas, el las agrega por nosotros.

Seguimos, nos dice que le especifiquemos que vamos a usar autenticación, y que debe estar después del que ya pusimos, por lo regular el orden de estas configuraciones no importa, salvo ciertas excepciones como esta, pero a modo general usted puede ponerlas en el orden que desee, normalmente organizado por orden alfabético o de importancia. Finalmente, las otras dos ya están en el proyecto.

Ahora viene el momento de comenzar a trabajar con las migraciones.

Como nosotros queremos un histórico limpio de nuestra estructura de datos, por eso lo primero que vamos a hacer es eliminar la clase que creamos hace un momento y vamos a proceder a hacer nuestra primera migración.

Nos vamos a Tools, Nuget package manager, luego Package manager console, y acá vamos a teclear el primero de nuestros comandos, add-migration y le asignamos un nombre.

```
add-migration IdentityCoco
```

Ahora podemos ver el folder de migraciones, vamos a ver lo que el creo para nosotros, podemos ver la estructura y podemos ver que la migración no es más que una pinche clase la cual hereda de Db migration.

Esta migración expresa en c# cómo se va a crear la base de datos, tenemos, un método llamado up, en el que podemos ver diversas llamadas al método de crear tabla, donde se le asigna un nombre a las mismas, y sus tipos de datos, ahora vamos a la consola para que esa migración vaya a la base de datos, en la línea de comandos nuevamente vamos a teclear.

```
update-database
```

Si les sale un error de que no se pudo crear el archivo es probable que no tengas instalado LocalDb o que la ruta sea inaccesible.

Para eso vamos a darle permiso a la carpeta, problema que se puede solucionar abriendo la solución como administrador.

Podemos ver en la base de datos ahora como se nos crea la misma con las tablas descritas por ahí, un dato curioso, cuando estén trabajando con Azure, ustedes deben crear la base de datos primero, solo la base de datos y las migraciones te va a crear todas las tablas.

Recuerda siempre que haya un cambio sube al control de versiones, así que procedemos a hacerlo y ahora si continuamos con nuestro modelo.

Creamos de nuevo nuestra clase, y hacemos referencia al ApplicationUser.

Así que esta es la clase que estoy buscando, así que vamos a cerrar la ventana y aquí vamos a crear una propiedad del tipo, ApplicationUser, usted puede llamarlo como entienda, en mi caso particular, como quiero señalar quien está creando el evento, es decir, el dueño yo lo llamare Owner, o bueno, Coach mejor.

Luego, ¿cuándo va a ocurrir esto?, así que creamos una nueva propiedad usando el shortcut “prop” y presionamos la tecla tab dos veces para que el me auto complete la estructura base de una propiedad y solo reemplazamos DateTime y el nombre será DateTime, porque no solo es una fecha, y ojo con eso, las variables

deben ser nombradas conforme a lo que vayan a hacer, si voy a usar una variable para fecha aunque sea de tipo datetime, debería llamarla date, o startingDate, si voy a usar una variable para Cedula y Rnc, debería llamarla CedulaRnc.

Seguimos con ¿Dónde va a ocurrir?, así que llenamos el Location, o Venue que suena más lindo.

¿Qué tipo de evento es? Sería la siguiente pregunta, pero acá tenemos un tema de normalización, no hace sentido que yo tenga un campo string para almacenar el tipo de evento, cuando estos, deberían ser seleccionables o una vez creados estar disponibles para ser candidatos a selección, pues nosotros no queremos repetir todos estos tipos en la base de datos por cada registro, así que lo que hace sentido es crear una tabla adicional que a los fines de Ef, sería una nueva clase, y nuestra clase debe tener un Id y un Nombre.

Creamos la clase Type, le ponemos un Id le cambiamos el tipo de datos a byte, porque no vamos a almacenar tanta información y debemos ser responsables con nuestras aplicaciones para que estén lo más optimizadas desde el comienzo, y finalmente nombre.

Volvemos a nuestra otra clase, donde especificamos que este va a tener una propiedad publica que va a ser de tipo Type y que voy a llamar Type un truco que podemos usar también es construir la clase acá mismo para mayor facilidad y luego decirle a ReSharper o al Codelence que nos la mueva a un archivo separado vamos a necesitar también un id que en este caso vamos a usar el int.

Bien, hasta acá vamos a dejar nuestra iteración con la base de datos, no vamos a desarrollar las demás entidades, porque ellos hacen parte de casos futuros, recuerda, piensa en pequeño y mantenlo simple.

La metodología de trabajo siempre debe ser, agrego una porción de código que me permita realizar una iteración, la pruebo y luego me muevo a la siguiente, nadie se mete una tarta helada en la boca de un fuetazo, en programación debe ser igual, no podemos crear un modelo entidad súper robusto si no ir construyéndolo gradualmente.

Ya que creamos y por ende cuenta como una modificación a nuestro dominio, debemos añadir una migración y actualizar la base de datos.

Un dato curioso, podemos crear variables de entorno para facilitarnos la apertura de algunas herramientas, como la consola de paquetes, entonces y la podemos usar para cualquier otra cosa, vamos a hacerlo.

Vamos a “Tools” luego “Options” después vamos a “Environment” y la opción “Keyboard”, luego digitamos *packagemanagerconsole* tú puedes crear la combinación que desees, en mi caso voy a escoger la tecla “Alt + /” para que cuando presione esta combinación en el teclado me abra la consola. o siempre manténgala abierta, no sé.

Ahora añadimos la migración podemos ver algo chulo, si nos fijamos cometimos un error, no fue intencional, pero que bueno que ocurrió, es que se nos olvidó decirle al contexto de datos que cree nuestras tablas, y usted dirá, pero ¿porque él no las tomo y manipulo igual que lo hizo con las de identity?, bueno, la respuesta es que identity hace toda su magia negra por detrás, y en algún lado especifica la relación a estas tablas, por eso la migración funciona, pero no pasa lo mismo con las nuestras por eso debemos ir al contexto de datos y añadir estas tablas como propiedades de un tipo especial llamado Dbset que no es más que un tipo de datos List, y como es una lista debemos especificarle por referencia una lista de que él va a alojar, y él va a alojar una lista de evento, y como es una lista que puede tener varios evento, le llamamos eventos, ahora si vamos a crear la migración.

Si ejecutamos el comando nuevamente podemos ver un error, porque ya existe un archivo de migración con ese nombre, podemos hacer una de tres cosas, una es ir y eliminar manualmente ese archivo, la otra es eliminar esa migración mediante comando, o usar el comando -force para forzar la actualización de una migración, por alguna razón que desconozco y no he encontrado respuestas en stack overflow, en netCore, no funciona el comando forcé, por lo que vamos a optar por eliminar el archivo o usar.

Delete-migration

Fíjense que al intentar realizar la migración nos arroja un error, el nos dice que la tabla eventos necesita un Primary Key, esto es muy importante, para trabajar con Entity Framework, es necesario que todas las tablas o todas las clases que van a ser tablas, tengan

un identificador, que puede ser de cualquier tipo, siempre que sea único, e irrepetible, normalmente nosotros deberíamos especificar esto, de hecho en framework la especificación es obligatoria, pero en netCore, se infiere que si se llama lo que sea Id, es la llave primaria, si vamos a usar un campo distinto entonces deberíamos usar la anotación [Key]

Como no es el caso porque nosotros venimos de la escuela de programación que usa las mejores prácticas, sabemos que es estúpido usar nomenclaturas extrañas en las entidades, como EmpleadoId, Emp_Nombre, etc, y somos gente organizada que programa bien, por eso usamos ID, Nombre, por inferencia se determina que el Id o MiTablaId va a ser el id principal de la tabla que estamos referenciando, “ay pero es que ¿cómo en un select voy a saber de qué tabla es cada campo?” bueno, lo primero es por la tabla, daaah, segundo, por un alias si estamos trabajando en SQL.

Bueno, ahora vamos a inspeccionar un pequeño problema que tenemos, para ello vamos a ir a nuestra base de datos, donde podemos ver algo muy problemático para tema de rendimiento y una pésima practica de programación, si nos vamos a nuestra tabla de eventos podemos ver que Venue es de Varchar (max), lo cual no nos deja muy bien parados ante los diseñadores de base de datos, así que vamos a corregir este impase, también otra cosa es que acepta nulos y no tiene sentido que nosotros permitamos que se acepten nulos en el lugar donde se va a celebrar el evento, C# hizo esto porque usa un concepto llamado “Conventions Over Configuration”, es decir, convenciones sobre configuraciones, basado en las convenciones que tiene por detrás. Convenciones son el modo en que los desarrolladores de C# entienden que debe

funcionar algo cuando no hay una configuración, es como el valor por defecto de algo, el arma la estructura de nuestra tabla, estas funcionan la mayor parte del tiempo, pero no siempre, otra cosa el id del coach es nutable, lo cual no tiene sentido porque siempre necesitamos saber quien creó el evento.

El tipo de datos es string, porque el tipo de datos del id en la tabla ASP.NET Users es string, el tema de si es una buena o mala práctica está a opción de lo que usted quiera usar, ese no es tema de esta clase.

Sobrescribiendo las convenciones.

Entonces nos toca sobrescribir las convenciones del Entity, para esto tenemos dos formas de hacerlo, una es con Data Annotations, que son etiquetas que se le pone arriba de las propiedades y la otra es Fluent API, que es tener las configuraciones en un archivo separado, la gente que sabe usa FluentApi y de hecho es mi favorita, pero requiere un esfuerzo extra, así que vamos a iniciar con las Data annotations y después veremos las fluent apis, después de todo con la primera usted mete mano, la otra es más para hacer Bam que otra cosa.

Primero asegurémonos de que los campos no sean “nubles” eso se hace con la palabra reservada de anotación [Required] obviamente tenemos que agregar la referencia haciendo clic y añadir referencia o presionando (CTRL + .) y la tecla Enter para la primera opción o movernos entre opciones. Bien, hacemos el Venue requerido, y le vamos a especificar la máxima cantidad de

caracteres que va a soportar, con la etiqueta [StringLength(255)] y finalmente el tipo lo hacemos requerido, una regla de legibilidad nos dice que debemos poner un espacio entre cada propiedad y por consiguiente también después de sus notaciones, vamos a los tipos y hacemos requerido el nombre y especificamos la longitud, como tocamos el modelo, debemos añadir una migración y obviamente un commit.

Vamos a analizar ahora lo que hizo esta migración, vemos que elimino estos foreignkeys. Modificó las columnas para que tengan esta nueva configuración, crea unos índices, y nos añade las llaves foráneas, lo de crear índices es lo mas importante de este punto, el crea los índices que entiende pertinentes, cosa que nosotros normalmente no hacemos cuando hacemos nuestros modelos de datos.

Inspeccionamos ahora la base de datos y vemos nuestros cambios.

Decisiones de diseño.

Cuando yo voy a crear un evento, deseo que el usuario seleccione el tipo desde un select list, drop down list o cualquier otro mecanismo, tenemos dos opciones aquí, Podemos implementar una página para la administración, la cual nos permita registrar todos los tipos de eventos O simplemente podemos traer los datos desde la base de datos llenándolos nosotros ¿Cuál solución es mejor? Implementar una página de administración no es parte de las metas de nuestro proyecto, este es uno de los problemas de

los desarrolladores, empiezan a trabajar en el problema A, pero luego para esto comienzan con el problema B el cual no hace parte del proyecto, ojo, no estoy entrando en contradicción cuando en el capítulo anterior te hablé de que como desarrollador debes valerte de tu experiencia para agregar valor a las propuestas ambiguas del cliente, siempre que puedas debes agregar valor, pero, la pregunta que debes hacer siempre que vas a crear una nueva funcionalidad que no esté dentro de tus requisitos, es ver, ¿cuál es el costo que tiene esto en tu solución y cuál es el beneficio?, no siempre, la ventaja de crear un panel de administración será lo más factible, porque podremos añadir en cualquier momento un nuevo tipo, pero implementar esto es mucho más costoso que llenar una tabla, una pregunta obligatoria seria, ¿cuántas veces en la vida van a cambiar estos tipos? Por eso en mi opinión implementar una página de administración para este caso en particular sería un Overkill del proyecto y un costo innecesario de tiempo de desarrollo, lo cual puede impactar el presupuesto del proyecto, este tipo de preguntas no se las puedes delegar al cliente, porque si le preguntas ellos van a decir que es mejor tener un panel de administración, aunque nunca lo vayan a usar, ahora tu evalúas, si no te impacta crear una pantalla que tienes que probar y que te puede atrasar en el trabajo y tienes que darle estilos, asegurarla contra ataques, etc, adelante, nosotros tomaremos la ruta del cobarde.

Así que nos vamos a ir por la decisión fácil para nosotros, que es crear el Script de creación de datos.

Bien, cuando estamos trabajando con la metodología code Firts, todo inicia por el código, no es correcto ir a llenar los datos a la base de datos, funciona si lo hago así, pero nosotros no hacemos ninguna modificación en la base de datos, ni siquiera en temas de

datos, así que vamos a crear una migración en blanco, la cual vamos a llamar llenado de datos, la migración está vacía porque no tenemos ningún cambio en el modelo y aquí podemos usar la instrucción Sql para crear cualquier tipo de script, por ejemplo `Sql("Insert into Type (Id, Name) Values (1, 'Personal Superation')");` pero eso era en Framework, ahora tenemos que hacer una estructura un tanto más complicadita en netCore.

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    //Sql("Insert into Type(Id, Name) Values(1, 'Personal Superation')");
    migrationBuilder.InsertData(
        table: "Types",
        columns: new[] { "Id", "Name" },
        values: new object[] { 1, "Personal Superation" });
}

protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DeleteData(
        table: "Types",
        keyColumn: "Id",
        keyValue: 1);
}
```

La parte down, lo que nos permite es decirle a ef que hacer cuando hagas un downgrade de la base de datos, que es posible, yo nunca he hecho uno, pero es buena práctica hacer lo que dicen la gente que sabe respecto a un tema, lo que quiere decir que si aquí insertamos estos datos en un posible downgrade deberíamos borrarlos, pero ya eso es a opción y labor de ustedes, Ahora vamos a hacer commit.

Resumen.

Bien, vamos a ver que vimos en este módulo, cuando vas a resolver una pieza de funcionalidad tú puedes iniciar desde la interfaz de usuario o desde la base de datos o desde el dominio, en este curso haremos siempre énfasis en el dominio, cuando creas el modelo de dominio aprendes a crear un modelo simple y eficiente, que resuelva el problema que tienes en las manos.

No trates de modelar todo el universo de tu sistema desde el comienzo solo enfócate en el modelo que estás trabajando en este momento, posteriormente podrás extenderlo.

Recuerda siempre migraciones pequeñas.

Cada que hagas una pequeña pieza de código, envía al repositorio, así que, haz pequeños cambios, revisa y prueba el código, asegúrate que funciona, límpialo, y luego commit y finalmente cuando vas a realizar la solución a una problemática que se te presente, evalúa siempre los beneficios y costos de todas las evaluadas, después de todos es tu decisión realizar esos cambios que van a impactar en el producto final.

Construyendo un Formulario

Por fin tenemos ya construido nuestro primer modelo real y eso nos hace feliz :D, pero, pero, pero, ahora necesitamos crear un formulario en donde poner los datos de captura, en esta sección vamos a ver cómo usar Bootstrap para hacer un formulario que se vea llamativo y atractivo, también vamos a ver lo que son los ViewModels, en que se diferencian de los Models, porque los necesitamos y como se crean.

Mi Primer Controlador

Bien, primero vamos a enfocarnos en la vista, olvidémonos por ahora de validaciones de data, solo nos vamos a enfocar en nuestra interfaz, aunque tampoco nos vamos a enfocar en estética, solo me interesa tener un formulario sencillo, que nos permita de manera conceptual capturar los datos que requerimos para nuestro evento, necesitamos entonces en primer lugar, crear un controlador.

Para esto vamos al “Solution Explorer” y dentro de la carpeta “Controllers”, damos clic derecho, luego Añadir nuevo “Controller”.

Por ahora no vamos a trabajar la opción de generado de todas nuestras vistas usando la plantilla de Scaffolding, para que aprendamos a entender la esencia de las cosas, después nos vamos a valer del Copy paste y el scaffolding para algunas cosas. Así que vamos a escoger un controlador vacío.

Recuerden que un controlador es el que intercepta las peticiones que hace el cliente desde la vista y ejecuta las acciones que se le pidan, para arrojar un resultado, es en el controlador donde se

hace toda la lógica de negocio, ya sea consultar data, interactuar con el modelo, ya después que usted quiera partir esa capa en “n capas” según las mejores prácticas de desarrollo, eso es otra cosa, pero, después de todo el controlador es de donde debe partir todo lo demás.

Le ponemos un nombre, que debe terminar en la palabra Controller y como se supone que dentro el va a manejar varias acciones relacionadas con eventos, lo vamos a llamar Events en plural.

Él nos crea una acción por defecto llamada Index, por ahora la vamos a cambiar, porque dijimos que lo que vamos a trabajar es en la creación.

Aquí podemos ver que lo que hace es un método de tipo IActionResult, que renombramos a Create o usted puede usar el nombre que desee, Crear, si lo quiere en español, Register que sería una terminología más acertada, pero se confundiría bastante con lo de registro de usuario, o podemos llamarla insert o como usted desee como nazca de su corazón, Create es el standard así que nos quedamos con Create, posterior a eso, vemos que lo único que hace es retornar una Vista que como no especificamos nombre, significa que va a retornar una vista llamada igual que la acción en cuestión.

Mi Primera Vista

Ahora procedemos a crear la vista, podemos hacerlo manual, nos vamos a la carpeta de Vistas, creamos un folder con el nombre de nuestro controlador sin la palabra "Controller" y dentro de esa carpeta creamos un archivo nuevo de tipo ".cshtml" que vamos a llamar como la acción que va a representar, en este caso "Create".

Pero, también podemos darle clic derecho a cualquier lugar dentro del método de la acción y luego clic en Añadir View, le dejamos el nombre "Create" y procedemos a crearlo vacío, después veremos que significan esas opciones que salen ahí, por ahora ignorémoslas.

Un dato importante es que si usted desea cambiar la convención de cómo funciona el tema de controlador, para así poder llamar sus vistas como usted desee y sus controladores con el nombre arbitrario, lo puede hacer, es una de las ventajas de netcore, pero, ¿en verdad tiene alguna utilidad? Si la respuesta es sí, bueno, adelante, pero le advierto que cuando usted le envíe su proyecto a otra persona para que le dé continuidad va a tardar más tiempo intentando entender lo que usted hizo, que, trabajando, por eso, manténgase dentro de los standares, y solo salgase cuando sea estrictamente necesario.

Vamos a probar que las cosas estén andando bien y para eso vamos a añadir un link en nuestra barra de navegación que apunte a nuestra acción dentro de nuestro controlador, el cual va a llamar la vista con el contenido que tiene lo que acabamos de crear que por ahora no es nada más que una pantalla con una etiqueta "<h2>" que dice "Create". Probamos que todo funcione y seguimos.

Bien, ahora que vamos a iniciar a trabajar con nuestra vista lo primero es que debemos especificar el modelo con el que va a trabajar esta vista, este paso no es 100% necesario, pues si nosotros no le especificamos el modelo a nuestra vista, él va a aceptar cualquier modelo y si es compatible con los datos que nosotros estemos mostrando entonces hará el casteo, sin embargo, esto afecta negativamente el performance, pues no es lo mismo recibir algo convertirlo a “object” y trabajarlo como tal, a recibirlo desde el comienzo como la clase que es, así que siempre especifique el tipo de modelo que va a recibir la vista, mírelo como que este es el tipo de datos de la vista.

@model NombreDeProyecto.FolderModelos.Modelo

Bootstrap 100.5 (Porque no llegamos a 101, xD)

Bien, para construir nuestro formulario, vamos a usar Bootstrap, y para eso entonces nos vamos a internet y escribimos Bootstrap, escogemos la primera opción o vamos directo a la página <https://getbootstrap.com> buscamos como se construye un formulario bonito y nos copiamos ese formato.

Bootstrap es uno de los tantos FrameWorks que sirve para construir aplicaciones web que se adapten a todo tipo de pantalla, originalmente creado por Twitter (creo y me da pereza averiguarlo, así que se lo dejo de tarea), y liberado para la comunidad, nos permite maquetar de una forma elegante y responsiva las pantallas de nuestras vistas, de una manera menos tormentosa para

aquellos que sufrieron en carne viva el desastre de maquetar la pantalla entera a base de tablas.

Bootstrap fue referenciado en nuestro “_Layout” page, en el head para los Css y después del footer para los scripts.

Bootstrap funciona basado en un sistema de columnas en el cual tu tienes una especie de tabla imaginaria con 12 columnas, con ellas les dices a tus controles que usen la cantidad de espacio disponible según la cantidad máxima de columnas que haya. Si desea aprender más de cómo funciona el sistema de grilla columnar puede ir a la documentación, nosotros no vamos a profundizar en eso y solo haremos uso del maravilloso arte del copy paste, saber usarlo, pero como funciona apréndalo por su cuenta, el curso es de MVC, y haga lo que usted quiera, (Emoji puño).

Una vez estamos en la página oficial podemos leer toda la documentación, pero vamos directo a los que nos concierne, en el link documentación, ubicamos el link “layout”, después componentes, y finalmente formulario, la idea es que si nosotros queremos crear un formulario como el que vemos aquí, necesitamos usar los que podemos ver ahí.

En esencia, lo que podemos ver es que tenemos una etiqueta form, luego un div con la clase form-group, que dentro contiene un label y un input al cual le ponemos la clase form-control, esta clase es la que le da el estilo bonito y un efecto chulísimo de hover cuando nos colocamos dentro de un textbox.

Ahora vamos a meter mano con nuestro formulario volviendo a la vista.

Ponemos un renglón para el lugar de encuentro, los elementos que estamos usando no son elementos extraños, como lo hacíamos en Framework que teníamos que valernos 100% de los tag helpers y teníamos que hacer unas líneas rarísimas que nadie se aprendía de memoria, acá estamos usando elementos comunes de un html cualquiera, el label para mostrar información el input para capturar información, la única diferencia es que le estamos indicando con el asp-for, quien está gobernando a ese elemento, a quien está atado.

Un paréntesis, algunos expertos recomiendan que el label también este atado con el asp-for, porque de esa forma yo no tendría que escribirle el texto, sino que lo halaría desde el “data anotación”, lo cual facilitaría la traducción si fuera un sistema multi lenguaje, en esa misma sintonía, porque si mañana queremos que esa etiqueta no se llame así, solo cambiamos el data anotación, pero, aca viene un tema y por eso es que yo soy junior developer aun, porque no siempre me llevo de los expertos para llegar a ser uno de ellos, este es uno de los casos, en la vida real, en la realidad de nuestros países latinos, el modelo de trabajo en el que la gente vive en una integración continua y se saca una versión funcional del software semanal, está muy lindo, pero en la vida real, los procesos burocráticos impiden hacer muchas cosas lindas y chulas, para fines de que me aprueben una mejora es más rápido y fácil que el encargado de aprobar vea que solo se modificó una etiqueta a que vea que se modificó el modelo, así sea con una anotación. Así que,

siga el consejo de los expertos, pero yo ya tengo esta mala costumbre.

```
<form>
  <div class="form-group">
    <label asp-for="Venue">Venue</label>
    <input asp-for="Venue" class="form-control" placeholder="Enter a venue">
  </div>
</form>
```

View Models

Ahora nosotros vamos a necesitar un Textbox (input) para la fecha, pero se me presenta un pequeño inconveniente, en nuestro modelo, tenemos la fecha y hora en un solo campo, pero para temas de usabilidad, normalmente la hora siempre se escoge aparte de la fecha, a pesar de que con DateTimePicker hay formas de que en un mismo control se escojan ambas, esto no es muy práctico (user friendly) que digamos.

(Además de que tenemos que hacerlo separado para poder forzar el ejemplo que viene a continuación, lol)

Este es uno de los tantos casos con los que nos vamos a topar, en los cuales, el modelo no es igual que lo que le vamos a mostrar al usuario, en nuestro modelo, la clase eventos tiene un solo campo para fecha y hora, pero nosotros queremos dos columnas aparte para capturar esta data, es ahí donde usamos una extensión del patrón MVC que se llama “Presentation Model” o “View Model” que es como se le llama en la actualidad.

Acá debo aclarar otro tema, algunos ejemplos y algunos expertos sugieren una especie de MVC que viene siendo más un DVC (Dominio, Vista, Controlador), de hecho esta presunción es la más aceptada, solo que sin el nombre de manera oficial (que yo sepa), ¿a qué me refiero?, a que, así como nosotros en una carpeta data pusimos dentro una carpeta llamada dominio y ahí tenemos nuestro modelo, así lo hacen los expertos, solo que en una dll aparte, pero eso lo veremos luego, entonces, usted dirá, pero, ¿si mis dominios los expertos dicen que van aparte, entonces, porque se llama MVC y no DVC?, bueno, porque lo que pasa es que los expertos tampoco le tiran o arman querties sobre el modelo de dominio de forma directa, en verdad lo correcto es que nuestros modelos sean dtos (Data Tranfer Objects), que son un duplicado de nuestro dominio, solo que para fines de presentación, sin embargo, yo en lo personal, aun trabajando con proyectos medianamente grande en la institución donde laboro, no he encontrado problemas reales que no sean por negligencia o una mala práctica, a tener que usar un dto, el único caso práctico para usar dto y no el dominio directo es para trabajar con data que nunca debe ser expuesta o que no deba existir la posibilidad remota de que se cambie un dato de esa tabla como la tabla usuario, todo lo que tienen que ver con el módulo de seguridad, y quizá las tablas de transacciones de una entidad bancaria, pero fuera de ahí, y ojo, este es uno de esos tantos malos consejos que les voy a dar así que no me hagan caso en esta parte, mi modelo, es el mismo dominio, pero de igual forma si nos da el tiempo vamos a refactorizar esto un poco más adelante, imagínense que de entrada yo les dijera que necesitamos crear dos veces la misma clase para trabajar con MVC, ustedes inmediatamente abandonarían el curso y me quedaría solo, además de que sin ustedes siquiera saber que era un modelo y como funciona vienen

a meterle capas adicionales, no, esa metodología solo sirve para ahuyentar los estudiantes, bueh, seguimos.

Lo importante ahora es crear nuestra carpeta de ViewModels, o algunos los meten dentro de la misma carpeta de modelos, pero, pero, pero, yo prefiero hacerlo como lo hacen los expertos, creando la carpeta.

Dentro de la carpeta vamos a crear una nueva clase que llamaremos EventViewModel, le creamos las propiedades que vamos a necesitar o podemos hacer como lo hacen la gente vaga, le decimos que herede de nuestra clase de dominio, claro, esto es una pésima practica según los expertos, pues, un modelo no debe heredar de un dominio, porque al final, no estamos separando para nada nuestro dominio de nuestro modelo, sin embargo quiero aprovechar este punto para que entendamos bien cómo funciona la herencia, sepan que esto es una mala práctica de programación y cuando estemos trabajando con un dto, este debe estar limpio, solo puede recibir herencia de otro dto o una clase suelta. Yo lo que suelo hacer es que después que mi aplicativo ya está funcional y no necesito realizar muchas modificaciones al sistema, ahí entonces quito la herencia, pero como me da pereza que a la hora de crear un campo nuevo tenga que hacerlo en dos lugares (y que casi siempre se me olvidaba de igual forma) yo lo que opte fue por hacerlo de esa forma, heredo y ya cuando está listo separo.

Entonces nada, añadimos los otros dos campos que necesito, uno para Date y otro para Time, pero ambos como string, pues, aunque netcore puede leer la fecha en formato fecha, es preferible manejar las cosas en el formato en que de verdad existe dentro la vista del

usuario, luego nuestro controlador se encargara de convertir esto a una fecha o a quien nuestro controlador delegue para hacer esa función.

Cambiamos en nuestro formulario el tipo de datos (el modelo que va a recibir) por nuestro EventViewModel

```
public class EventViewModel:Event
{
    public string Date { get; set; }
    public string Time { get; set; }
}
```

ViewBag y ViewData

Nosotros ahora necesitamos una especie de Drop Down List para seleccionar el tipo de evento.

Y aquí necesitamos introducir un concepto que habíamos visto al principio, que es el ViewBag, o el ViewData, ambos sirven para pasar información del controlador a la vista, esta información pasada obviamente no es parte de nuestro modelo, porque si lo fuera, sencillamente se lo pasamos y punto, normalmente se usa para mostrar anuncios, guardar información que usaremos mediante scripts, entre otros, como transportar datos para realizar selecciones.

Entonces en el controlador le indicamos que deseamos crear una variable que reciba los datos de nuestra base de datos, para eso usamos nuestro contexto de datos, nosotros necesitamos crear una nueva instancia de nuestro DataContext y para eso

procedemos a crear un atributo privado, que llamaremos `_context`, por convención si tengo una propiedad privada que vamos a usar para la ejecución de nuestra clase, la nombramos con underscore.

Luego necesitamos inicializarla desde el constructor, recuerden que el constructor es lo que se ejecuta cuando se crea una nueva instancia de la clase.

```
private readonly DataContext _context;  
  
public EventsController()  
{  
    _context = new DataContext();  
}
```

Pero, acá viene un tema, imagina que yo me olvido de inicializar el controlador, o que por error cuando cambie la cadena de conexión, yo deje una inicialización antigua, para eso entonces debemos usar la inyección de dependencia, en nuestro caso, lo único que tenemos que hacer es sencillamente recibir como parámetro por inyección nuestro contexto, el cual viene inyectado desde que creamos nuestro Identity.

Mas Adelante profundizaremos en ese tema.

```
private readonly DataContext _context;  
  
public EventsController(DataContext datacontext)  
{  
    _context = datacontext;  
}
```

Si tú eres de los que sabe un poco y quieres brillar es probable que levantes la mano y dirás, que debemos usar el patrón repositorio,

porque no es correcto trabajar el controlador con el contexto desde el controlador, pero, a pesar de que yo y muchos desarrolladores .net entendemos completamente innecesario el patrón repositorio, en .netCore, ojo, solo en netcore, pues en FrameWork, donde en verdad no había una abstracción real de la base de datos, era vital hacerlo así, sin embargo, como en netcore, el entity si es agnóstico de tu base de datos, el patrón repositorio dejo de cobrar algo de sentido, sin embargo, lo vamos a ver para entenderlo, y que lo podamos usar en FrameWorks, así como tener ciertas ventajas que ofrece el patrón repositorio, cuando nos topamos con duplicidad de código, pero por lo pronto, no vamos a profundizar, recuerda, mantente simple, cumple las metas, después refactoriza, el cliente no te va a preguntar si usaste el patrón x o y a la hora de que no le entregues a tiempo lo que él te pidió o no, el cliente lo que quiere es ver su pantalla.

La razón de ponerlo readonly es porque nosotros solo lo vamos a inicializar en el constructor una vez, pero no lo vamos a volver a tocar sus valores dentro de la ejecución de nuestro código, por lo que no queremos que por error se me ocurra cambiar elementos de su inicialización.

Entonces ahora vamos a hacer una llamada al contexto de datos para que nos traiga la lista de los tipos de eventos, lo almacenamos en una variable y esa información la vamos a atrapar en un ViewBag, la única diferencia entre uno y otro es la forma de llenarlo y crearlo, por lo que es irrelevante cual use de los dos. De hecho, creo que los dos se convierten en un mismo objeto al final, no se averigüen esa parte y me dejan saber, igual no voy a actualizar el documento, pero me dejan saber.

```
var types = _context.Types.ToList();
```

Los resultados los convertimos a lista, esa lista la convertimos a una nueva instancia de un tipo de lista seleccionable (SelectList) y ese valor se lo asignamos a un viewbag que llamaremos Typeld, el nombre es irrelevante, pero por convención para que haga alusión a lo que será seleccionado, se usa de esta estructura NombreTablald.

```
        ViewData["Typeld"] = new SelectList(types, "Id", "Name");  
//ó  
        ViewBag.Typeld = new SelectList(types, "Id", "Name");
```

El select list en una de sus sobrecargas, recibe una lista de objetos, que puede ser tipo list, array, colección, IQueryable, etc, luego recibe el elemento que le va a servir de identificador o cual va a ser el Value y después cual va a ser el valor a mostrar o el Text Value. Otra sobrecarga de este método acepta el valor actual seleccionado, pero eso lo veremos luego.

Volvemos a la vista y construimos nuestro elemento select al cual le indicamos a quien él va a llenar de nuestro modelo, que ojo, no tiene nada que ver con el Typeld del ViewBag y le decimos luego de dónde vienen los datos con la etiqueta asp-items.

```
<div class="form-group">  
    <label asp-for="Type">Type</label>  
    <select asp-for="Typeld" asp-items="ViewBag.Typeld"> </select>  
    <select asp-for="Typeld" asp-items="(SelectList)@ViewData["Typeld"]"> </select>  
</div>
```

En el caso del ViewData, debemos hacer el cast para indicarle a razor de que tipo es la información que va a recibir, en el caso del ViewBag no tenemos que especificarlo, pero si no es de un tipo compatible revienta cuando intente cargar la información.

Pero para subsanar el error que vemos de que Typeld no existe en nuestro modelo, debemos ir a crearlo, lo haremos en el ViewModel, porque es el único sitio donde lo necesitamos.

Otra forma también sería, creando un IEnumerable de SelectListItem, en nuestro ViewModel y llenarlo desde el controlador, pero, es cuestión de gustos, yo lo prefiero de esta manera, aunque una ventaja de usarlo mediante lista seleccionable atachada al modelo, es que cuando tengo que hacer validaciones en el controlador, y retornar a la vista un mensaje, la información del ViewBag se pierde y habría que llenarla nuevamente, mientras que si lo hacemos de esta forma los datos se mantienen en el modelo, por eso mucha gente prefiere hacerlo como lista seleccionable, en principio lo vamos a trabajar de esta forma, para no complicarnos mucho la vuelta, sin embargo si estas muy ansioso, acá te dejo como debería quedar..

Clase

```
public class EventViewModel:Event
{
    public string Date { get; set; }

    public string Time { get; set; }

    public byte Typeld { get; set; }

    public IEnumerable<SelectListItem> Types { get; set; }
}
```

Controlador

```
public IActionResult Create()
{
    var types = _context.Events.ToList();

    //ViewData["TypeId"] = new SelectList(types, "Id", "Name");
    //ViewBag.TypeId = new SelectList(types, "Id", "Name");
    var vm = new EventViewModel
    {
        Types = new SelectList(types, "Id", "Name")
    };
    return View(vm);
}
```

Vista

```
<select asp-for="TypeId" asp-items="@Model.Types" class="form-control"></select>
```

Probamos y si no nos aparece los estilos bonitos en nuestro control, es porque olvidamos agregarle el form control.

```
class="form-control"
```

Submit

Bueno, finalmente necesitamos un botón para enviar de la vista al servidor nuestro formulario.

El botón es un elemento que debe ser de tipo submit, es decir, que al hacer clic en el va a enviar un formulario, usted puede también hacerlo con un input al cual le asigne el type="submit" pero, si hay un elemento botón, ¿porque complicarse la existencia?. ¿Cuál es la necesidad de que si un orificio se hizo para entrar datos ahora usted lo quiera forzar a poner que es para enviar cosas?

Para darle el estilo de Bootstrap para botones, podemos ir a la página de ellos y en layout buscamos botones, y podemos ver que hay diferentes tipos de botones y que para tener cualquiera de ellos lo único que debemos hacer es añadir la clase `btn btn-ElQueMasMeGuste`.

La esencia de estos botones es que uses el azul marino para tus acciones primarias, si vas a indicar una alerta uses el anaranjado, si es algo peligroso el rojo y así.

Ya con esto tenemos nuestro formulario terminado y con eso damos por concluida la lección.

Resumen

Bien, revisemos que aprendimos en este capítulo, aprendimos a crear un formulario usando Bootstrap y que es este Framework que nos facilita la vida a los desarrolladores que no tenemos la vena de..., para construir interfaces bonitas y agradables.

También aprendimos sobre el ViewModel, que son usados en esas situaciones en las cuales queremos mostrar algo que es diferente de nuestro modelo.

Vimos de igual forma algunos controles como llenarlos y mostrar la información, así como llamar el contexto de datos para que se conecte a la base de datos y nos traiga información.

Guardando Información en la base de datos.

Ahora llego el momento de salvar esa información que hemos de introducir por el formulario que acabamos de crear en la base de datos.

Acá hablaremos, o más bien, vas a leer, sobre la separación de conceptos, este es una de los temas que diferencia a un junior developer de un senior, pues un senior sabe separar y delegar cada funcionamiento de su aplicación, me explico, si algo debe guardar un evento, es guardar un evento, no debe hacer cálculos, para eso usted debe crear un método aparte que lo haga, no es correcto que una función haga más de una cosa y lo veremos con ejemplos, donde haremos un toyo y luego lo refactorizaremos para entender mejor lo de la separación de conceptos.

Autorización de acceso.

Antes de implementar nuestro módulo de guardado, creo que es prudente que agreguemos una validación y es que solo un usuario con una cuenta y la sesión iniciada puede crear un evento, pues, el usuario que lo crea es un requisito de nuestro análisis inicial, eso se hace de manera simple, decoramos nuestra acción con el atributo de Authorize, de esa forma, solo usuarios autenticados pueden llamar esa acción.

[Authorize]

Y si corremos la aplicación podemos ver que si intentamos ir a esa ruta ahora nos redirige al login.

Nos registramos o iniciamos sesión, e intentamos ir nuevamente a la página anterior, viendo que ahora si nos deja ver la página sin problemas.

Insertando data.

Ahora, nos toca volver al create de eventos e ir a la etiqueta form, la cual debe ser reemplazada para indicarle a que acción él debe ir, pero eso era en Framework, porque ahora en netcore, basta con solo indicárselo al lado, que acción será ejecutada cuando yo haga un post de ese formulario.

```
<form asp-action="Create">
```

Volvemos al controlador y le indicamos que voy a crear una nueva acción.

Pero esta acción a diferencia de la anterior que no recibe nada y solo envía, esta va a recibir un parámetro, que es del tipo que sea el formulario y la nombramos como usted desee, en este caso a mí me gusta usar "vm".

Ahora bien, para que esta acción sea distinta de la anterior, debemos hacerle un decorado, debemos decirle que esta acción es el post de otra, en la anterior de forma implícita tenemos un Get,

que no se marca, es como el signo positivo en los números naturales, que está ahí pero no se marca, pero los demás (Post, Put, Delete) si necesitamos marcarlo.

El primero de ellos es la etiqueta de Authorize, pero, aunque es recomendado por algunos expertos, la práctica me dice que no hay forma de tu invocar este post sin previamente haber invocado la vista, pero, ya eso es opción de ustedes, el segundo decorado es HttpPost, para indicarle que es un post, porque deseamos que esta acción solo se llame con un submit de la vista que hayamos indicado que puede llamar este elemento.

```
[Authorize]
[HttpPost]
public IActionResult Create(EventViewModel vm)
```

Lo siguiente que debemos hacer es convertir nuestro ViewModel a un objeto del tipo de nuestro modelo para posteriormente salvarlo en la base de datos, porque de nada nos sirve enviar directo el viewmodel, si no va a haber ningún dominio que tenga esa estructura que nosotros enviemos.

Lo primero que debemos hacer es crear una variable en la que obtendremos el usuario conectado, por herencia tenemos una clase estática que nos da algunos datos básicos del usuario, así que buscamos en la base de datos el usuario donde la tabla usuario exista un id similar al id del usuario conectado. En Framework era un poco más simple la vuelta, por motivos de seguridad se complicó un poco en netcore, pero acá les dejo los dos métodos.

netCore

```
var user = _context.Users.Where(p => p.Id ==  
User.FindFirstValue(ClaimTypes.NameIdentifier)).FirstOrDefault();
```

Framwork

```
var user = _context.Users.Where(p => p.Id == User.Identity.GetUserId()).FirstOrDefault();//
```

Lo siguiente que debemos hacer es crear un objeto de tipo evento el cual vamos a agregarle el usuario que trajimos de la base de datos, le voy a llamar ev, porque la palabra event es reservada del lenguaje.

Ahora vamos a añadir el DateTime, pero tenemos un problema, y es el hecho de que atrapamos estos datos separados, por ahora no vamos a validar si es una fecha valida o no, vamos a partir de realizar cambios simples, y después vamos validando y mejorando nuestra aplicación.

Esa es la mentalidad que debemos adoptar en lugar de resolver todo de un fuetazo, vamos a resolver algo en concreto, probamos vemos que todo está bien y después nos movemos al siguiente problema.

Por ende, le indicamos que haremos un cast de estos dos datos que vamos que trae el viewmodel.

Ahora con el tipo, si nosotros hubiéramos usado el método de llenar la lista en el viewmodel, deberíamos ahora hacer una llamada a la base de datos sino hubiéramos agregado el Typed como dicen los gurus, pero como lo hicimos, puedo indicarle de

forma directa que el Typeld de mi evento es igual a mi Typeld del ViewModel.

Ahora tenemos que hacer lo propio con el tipo, traemos los datos basado en el que seleccionamos, y aquí viene uno de los otros problemas que se presentan en core, no hace mucho sentido que yo tenga que llamar el dominio cuando al final, aunque el modelo recibe los dos dominios, lo único que guarda es el Id, por eso, algunos desarrolladores, nos vamos a nuestro modelo y a este modelo le creamos una nueva propiedad del tipo de datos que sea el Id del que vamos a asociar, y lo llamamos el nombre del Dominiold, para fines de legibilidad lo ponemos arriba del objeto en cuestión y aunque no es requerido, podríamos ponerle el atributo ForeignKey para indicarle que tendrá una llave foránea por ese campo, pero esto solo tendría sentido si la propiedad no se va a llamar igual que como normalmente lo crea en la base de datos.

Deberíamos hacer lo mismo con el ViewModel, pero como lo estamos poniendo a heredar del modelo, el trae ese campo. En nuestro caso, tenemos que cortarlo del ViewModel y ponerlo en el dominio.

Así que de esa forma, me ahorro tener que hacer otra llamada adicional para traer esos datos.

```
public byte Typeld { get; set; }  
[ForeignKey("Typeld")]  
public Type Type { get; set; }
```

Y finalmente agregamos el Lugar de encuentro.

```
var ev = new Event
{
    Coach = user,
    DateTime = DateTime.Parse($"{vm.Date} {vm.Time}"),
    Typeld = vm.Typeld,
    Venue = vm.Venue
};
```

Entonces le decimos a nuestro contexto que va a agregar un nuevo objeto a la base de datos y que este va a ser el que acabamos de crear.

También podemos hacerlo especificando cual dominio es el que estamos agregando, pero él lo infiere, así que no está mal una u otra.

```
_context.Add(ev);
// o
_context.Events.Add(ev);
```

Y finalmente le decimos al contexto que guarde los cambios.

```
_context.SaveChanges();
```

Ahora, cuando esto termine, nosotros de manera temporal vamos a redireccionar al home, y posteriormente vamos a reemplazar nuestra HomePage por todos nuestros próximos eventos.

Así que, después de guardar, le decimos que retorne una redirección a la acción Index, que se encuentra en el controlador home.

```
return RedirectToAction("Index", "Home");
```

Corremos la aplicación y probamos.

Es posible que veamos un error, que era muy común en Ef Framework, consiste en que al momento de convertir esta expresión lambda, el aquí (en nuestro controlador) sabe que eso es una función, pero cuando se va al ORM (ef), el no entiende que es ese objeto, por lo que deberíamos atrapar en una variable el Id del usuario, para usar esa variable como filtro, como en Core eso no pasa, seguimos.

```
var userId = User.Identity.GetUserId();  
var user = _context.Users.Where(p => p.Id == userId).FirstOrDefault();
```

Probamos y ahora todo funciona nítido, verificamos la base de datos y vemos que todo está nice, guardando correctamente.

Bien, ya tenemos la implementación del guardado de nuestro evento, pero hay algunas cosas que no me gustan en lo particular, de cómo está estructurado lo que hicimos.

Nosotros tenemos una llamada a la base de datos y eso que nos ahorramos una de ellas, porque iban a ser dos.

Pero de todos modos estamos cargando dos llamadas a la base de datos, entonces, lo que vamos a hacer es lo mismo que hicimos para ahorrarnos los pasos innecesarios de ahorita lo vamos a hacer para modificar también nuestro dominio con el coach, por eso nos vamos al dominio y agregamos un CoachId, como la tabla

no se llama Coach, sino aspnetuser, aquí si debo añadir el foreignKey de manera obligatoria.

El requerido se lo asignamos al Id, y no a la propiedad de navegación, las propiedades de navegación, son aquellas que me permiten hacer enlaces entre un dominio y el otro, para indicar dependencias.

```
[Required]
public string CoachId { get; set; }
[ForeignKey("CoachId")]
public ApplicationUser Coach { get; set; }
```

Si añadimos una migración podremos ver que no hay un cambio significativo, solo se dropean las relaciones existentes para usar una relación idéntica con el mismo nombre.

Y ahora si podemos llamar directo sin necesidad de usar una variable extra, porque esto no es una expresión lamda, sino una construcción de una clase dentro de nuestro controlador. Que se queda en el controlador y no va al ORM como lo hacía en el caso anterior.

```
var ev = new Event
{
    DateTime = DateTime.Parse($"{vm.Date} {vm.Time}"),
    TypeId = vm.TypeId,
    Venue = vm.Venue,
    CoachId = User.FindFirstValue(ClaimTypes.NameIdentifier)
};
```

Corremos la aplicación y nos aseguramos de que todo sigue funcionando igual.

Separación de Conceptos.

Hay algo más que no me gusta mucho de esta implementación.

Es el hecho de hacer al controlador responsable de parsear una cadena de texto a fecha, lo cual no debe ser una responsabilidad de él, esto es separación de conceptos, el controlador debe actuar como un coordinador para la lógica de la aplicación, que es lo que sigue después, que pasos dar, es la única responsabilidad del controlador, la función de un administrador no es hacer el trabajo del programador, es su responsabilidad tomar a sus empleados asignarle tareas y obtener los resultados y usarlo posteriormente, es lo mismo para nuestro controlador, él debe actuar como un administrador.

Por eso debemos parsear esto en un objeto diferente, en poo hay un principio llamado information expert que significa que la clase o los objetos que tienen información para hacer algo debe ser el que se responsabilice para hacer esas cosas, podemos usar una metáfora, piensa en un Cheft, él sabe sobre la receta, la comida y cosas como esas, el es quien hace la comida, no el camarero, en este caso el ViewModel es la clase que sabe o está enterada sobre la fecha y la hora, por lo que la combinación de estos elementos debe ser la responsabilidad de la clase que lo gobierna, es este que debe retornar el valor que nosotros necesitamos.

Por lo que vamos a crear una nueva propiedad, de solo lectura que llamaremos DateTime o FechaCombinada o como usted desee, que solo tendrá un get y ese get será el retorno de la combinación

de la fecha y la hora. ¿Pero qué pasa? Nos topamos con un problema, ya tengo en mi dominio una propiedad llamada DateTime, así que tengo dos opciones, o le pongo otro nombre, o dejo la vagancia y comienzo a trabajar los ViewModels como dicen la gente que sabe. Así que vamos a en este caso, a arreglar ese disparate que hicimos al comienzo., nos veremos obligados a agregar Venue y Typeld.

```
public class EventViewModel
{
    public string Date { get; set; }

    public string Time { get; set; }

    public byte Typeld { get; set; }

    public string Venue { get; set; }

    public DateTime DateTime
    {
        get {
            return DateTime.Parse($"{Date} {Time}");
        }
    }

    // public IEnumerable<SelectListItem> Types { get; set; }
}
```

Después de esto probamos y validamos que esta refactorización no haya roto nada, y hacemos un commit., por supuesto, en cada uno de los temas anteriores ya habíamos hecho commit, por lo que asumo que en este commit solo tendrás dos archivos modificados.

Siempre es bueno mantener las refactorizaciones en commits separados a las funcionalidades que implementamos.

Resumen

Bien, veamos el resumen de los puntos principales de este módulo.

Aprendimos a usar el atributo de autorización, para limitar el acceso a usuarios conectados, después veremos cómo se hace para usuarios específicos.

Aprendimos a usar la clase estática de User para obtener datos del usuario conectado.

Aprendimos a usar las propiedades ForeignKey, su ventaja principal es que nos ahorra hacer llamadas innecesarias a la base de datos.

Algunos desarrolladores no les gustan estas propiedades porque entienden que esto hace mucho ruido o abulta del modelo de dominio, y establecen que eso no lo hace muy orientado a objeto, personalmente no tengo problemas en sacrificar un poco de reglas siempre que me ahorre mucho mas o que el beneficio de romper una regla sea mucho mayor.

Y finalmente aprendimos un poco de el principio de experto en información, el cual nos ayuda a identificar donde delegar la responsabilidad de algo.

Validaciones

Esta sección se trata completamente de validaciones, vamos a vernos del uso de data anotaciones, para restringir lo que se puede ingresar en nuestros campos de entrada y crearemos de igual forma un data-annotation personalizado para aprender como extender la funcionalidad de estos, el cual usaremos para validar nuestra fecha y hora.

Validaciones en el servidor.

Vamos a iniciar con las validaciones en el servidor, que no son más que esas validaciones que le ponemos a nuestro programa, del lado de lo que se procesa cuando se realiza una petición, ósea, en el código fuente de la aplicación.

Para eso tenemos que ir a nuestro ViewModel... una vez acá lo primero que debemos hacer es validar cuales son nuestros datos requeridos, recuerden que esto se hace con la etiqueta [Required] arriba de la propiedad que nosotros deseamos que siempre esté llena, le ponemos requerido a todas las propiedades excepto a la propiedad de solo salida y tampoco a la de enlace que se usa para formar la lista.

Una vez realizado este proceso, ya nuestro modelo sabe que si no hay un dato lleno en una de esas propiedades, el modelo deja de ser válido, pero ahora nos toca mostrarle esa información al usuario, debido a que si probamos hasta este punto, podemos ver que si intentamos guardar sencillamente no guarda y da un error,

así que para eso iremos a nuestra vista y le pondremos mensajes de validación a cada campo que sepamos que se le debe informar algo al usuario.

Esto se hace añadiendo una simple etiqueta HTML de tipo span, podríamos usar otro elemento si así lo deseamos, pero el span es el idóneo para esto, porque es uno de los elementos de enmaquetado que menos pesa, además es el que viene en la plantilla por defecto, use un span o un label, no use otro, bueh, el punto es que a esta etiqueta le agregamos el atributo de asp-validation-for, ya con eso le decimos que siempre que en necesite mostrar una validación para ese campo la muestre en ese span el resultado que diga el server.

```
<span asp-validation-for="Venue"></span>
```

Hacemos lo propio con las demás...

El siguiente paso es modificar el controlador, para eso nos vamos al post de la acción create, nosotros antes de enviar a la base de datos, debemos preguntar si el modelo es válido, y si no lo es, debe de retornar el mismo View en que nos encontramos, en caso contrario, debe guardar y redireccionar al usuario a la home siempre que haya podido guardar satisfactoriamente, usted puede preguntar “si el modelo es válido hazme el insert” o puede preguntar “si el modelo NO es válido retorna la vista”, eso ya es cuestión de gustos. Lo que si no debemos hacer es redundar poniendo un else, nosotros somos super programadores que no redundamos.

```
if ( ! ModelState.IsValid)
{
    return View();
}
//make save....

// or //

if (ModelState.IsValid)
{
    //make save....
}

return View();
```

Ahora si corremos la aplicación dejando los campos vacíos y le damos a guardar, podemos ver un error, nos dice que hubo un error intentando convertir las fechas, bueno, este error no deberíamos recibirlo, pues se supone que nosotros antes de llegar al momento en que consumimos el dato DateTime, hacemos la validación, los dejo unos segundos para que piensen en el porque estamos viendo este error, (claro, suponiendo que de verdad vas probando mientras vas leyendo esto, lo cual se supone que deberías ir haciendo) nada, déjame contar. Uno, dos, tres, cuatro, cinco, ¡ay!, ya me aburrí de escribir, nada, te respondo y listo.

Bueno, esto pasa porque ASP usa “Reflection” para construir el ViewModel, y como parte de esta construcción el va a tocar todas las propiedades de este, así que, por cada propiedad el va a inspeccionar el Request Http para ver si hay un valor con esa llave (nombre del campo) y como en este caso un nulo no se puede convertir a fecha, por eso revienta como un “patelito” (por cierto, si una palabra mal escrita está entre comillas, significa que fue escrita mal de forma intencional, “barsa de incurtos”).

Así que nada, para resolver este error debemos convertir esta propiedad en un método, de esta forma el MVC no va a tocar la propiedad DateTime usando Reflection, por lo tanto vamos al EventViewModel.

Renombramos la propiedad a GetDateTime, si le damos Ctrl + punto (.) podemos ver que el nos sugiere si deseamos renombrar esa propiedad y si le decimos que yes, el dónde quiera que teníamos una referencia a esa propiedad se va a cambiar por el nuevo nombre, evitando así tener que ir a todos los lugares donde estaba referenciada a cambiar el nombre.

Y ahora la convertimos a un método lo que quiere decir que ya no necesitamos el Get, sino que retornamos el valor de la concatenación solamente. Y como ahora esto es un método donde lo llamábamos antes, necesitamos agregar paréntesis, de esa si no nos libramos con el Code Lense de VS, creo que con el de Resharper si me hace el renombrado completo, pero no recuerdo y como me da pereza instalarlo para averiguarlo se los dejo de tarea.

Bien, si corremos e intentamos hacer clic de nuevo en el botón guardar tendremos otra excepción, nos dice que el Select no puede ser llenado debito a que perdió la información que tenía el ViewBag, (claro, si hiciste el envió de la información con el ViewBag o ViewData y no con una propiedad tipo lista en el ViewModel).

Nosotros si inspeccionamos el controlador, cuando el modelo no es valido solo le enviamos la información a la vista que él tiene almacenada en el modelo, así que antes de retornar la vista, debemos de llenar nuevamente un ViewBag, consultando la información de la base de datos, con la diferencia de que ahora debemos agregar un cuarto parámetro, que es el que esté seleccionado, en caso de que ellos hubieran escogido uno, para que no se pierda ese dato y haya que seleccionarlo de cero de nuevo, imaginen por ejemplo si ya seleccione Republica Dominicana en un combo de país, seria tonto volver a tener que seleccionarlo, por eso pasamos el que este seleccionado en el modelo como cuarto parámetro de nuestra consulta o llenamos desde un principio el modelo que enviamos en la parte Get del Create y nos ahorramos este paso, que apropiado es más práctico y profesional a la larga aunque con el ViewBag el código se ve más limpio y es mi preferencia personal.

Ahora si probamos podemos comprobar que nuestra validación funciona, deberíamos ponerla roja, pero, por ahora no estamos en estética, sino en funcionalidad, eso lo dejamos para después, basta con que funcione.

Validaciones personalizadas.

Ahora que comprobamos que nuestras validaciones funcionan, debemos asegurarnos de que el usuario ponga una fecha en el campo de fecha y que sea válida, de igual forma que siempre sea una fecha futura, en dotnet no hay una validación que haga eso por nosotros, hay para validar formatos de fecha pero no una que valide eso de forma automática y no, no quiero cargarle la

responsabilidad al controlador si podemos tener algo que sea reutilizable para todos nuestros futuros modelos.

Así que nos vamos a crear una carpeta llamada Helpers, donde vamos a crear una clase llamada FutureDate, donde dentro vamos a hacerla heredar de ValidationAttribute y sobre escribimos el método de validación, antes de continuar quiero que entendamos como va a funcionar esto.

Para eso volvemos a nuestro ViewModel y le ponemos nuestra nueva anotación.

En tiempo de ejecución, cuando el tenga que averiguar si una fecha es válida, él va a entrar en nuestra anotación personalizada, entonces debemos asegurarnos primero que es una fecha aceptada, usamos por lo tanto el TryParse, o podemos hacer un Try y capturar la excepción, pero nosotros somos super programadores que no redundamos en código innecesario así que no usamos esa basura, que sobrecarga mucho nuestra aplicación, debido a que nosotros sabemos usar métodos vacanos que resuelven, bueno, también podemos usar el TryParseExact, el cual intenta hacer la conversión usando un formato en específico, posteriormente usaremos un Placeholder del lado del cliente para indicarle al para indicarle al ñame, digo al usuario, el formato que debe introducir para la fecha.

So, usemos TryParseExact, que nos pide como primer parámetro una cadena, por lo que necesitamos convertir el valor que viene por referencia pasado acá a una cadena, esto lo hacemos con

`value.ToString()` o `Convert.ToString(value)`, el segundo argumento es el formato, en nuestro caso usaremos `dd/MM/yyyy`, y aquí una acotación, de Fechas 101, la `m` minúscula indica minutos, así que mucho ojo con eso, que minutos son 60 mientras meses 12, nada seguimos. El tercer argumento es un proveedor de formato o el `CultureInfo` que vamos a usar, por defecto la cultura es ingles de USA, siempre deberíamos aquí usar la cultura por defecto, aunque sea distinta de la que nosotros vamos a trabajar, la ventaja de usar el default es que podemos sobrescribir cual es el default una sola vez en el `Startup.cs` y de esa forma olvidarme de cambiar código a lo loco. El cuarto parámetro es una enumeración llamada `estilo` de la fecha y hora que honestamente no se para que es, así que usemos el que yo vi en internet, que es `DateTimeStyle.None`, y el ultimo argumento es un `DateTime` object, donde si la conversión es exitosa el va a almacenar la representación de ese string, por lo que necesitamos una variable de tipo fecha, y luego usaremos el modificador de salida (ultimo parámetro) para que lo use como cajón, una regla de legibilidad nos dice que nunca deberíamos tener la necesidad de hacer scroll a la derecha o izquierda, así que partamos ese pedazo grandote de código para que quede un argumento en cada línea.

Ahora vamos a almacenar el resultado de la conversión en una variable llamada `isValid`.

Lo próximo a verificar es que esta fecha sea una fecha futura, podemos comparar y luego obtener el resultado o podemos directamente decirle que nos retorne el valor de `isValid`, que si llega acá es porque es verdadero, siempre y cuando se cumpla la condición de que la fecha sea mayor a la del día.

```

public class FutureDate : ValidationAttribute
{
    public override bool IsValid(object value)
    {
        DateTime dateTime;
        var isValid = DateTime.TryParseExact(value.ToString(),
            "dd/MM/yyyy",
            CultureInfo.CurrentCulture,
            DateTimeStyles.None,
            out dateTime);

        return (isValid && dateTime > DateTime.Now);
    }
}

```

Si corremos la aplicación y ponemos una fecha invalida podremos ver el mensaje de error y si ponemos una valida ya no nos marca.

Ahora para validar la hora, lo único que tenemos que hacer es duplicar el contenido de esta clase que acabamos de crear, cambiarle el nombre y con CTRL + punto (.) indicarle que deseamos que nos ponga esa clase en un archivo separado.

Cambiamos la lógica de validación por una que aplique para hora, y al final solo retornamos si es válida.

```

public class ValidTime : ValidationAttribute
{
    public override bool IsValid(object value)
    {
        DateTime dateTime;
        var isValid = DateTime.TryParseExact(value.ToString(),
            "HH:mm",
            CultureInfo.CurrentCulture,
            DateTimeStyles.None,
            out dateTime);
    }
}

```



```
        return isValid;  
    }  
}
```

Nos vamos a nuestro ViewModel y le aplicamos el atributo personalizado, y probamos.

```
[Required]  
[FutureDate]  
public string Date { get; set; }  
  
[Required]  
[ValidTime]  
public string Time { get; set; }
```

Validaciones en el lado del cliente.

Bien, nosotros ya tenemos nuestros datos validados desde el lado del servidor, ahora debemos validar del lado del cliente, déjenme mostrarle algo que estoy seguro que ustedes saben, pero tengo que asumir que no, si le damos clic derecho inspeccionar elementos (este paso obviamente con nuestro programa cargado y situados en la vista Create, podemos ver además del código fuente generado de la página en puro HTML y CSS, podemos ver en el apartado Network, lo que pasa al momento de hacer clic en guardar (haz clic).

Podemos ver que se hace una petición al servidor, al principio tenemos un Request a la acción create y un montón de cargas que hace cuando se recarga una pagina (esto seria el infame equivalente alPostBack tan amado por desarrolladores DotNet de Web Forms y que da tanto asco a los desarrolladores de otras plataformas (y a los que somos .Net, pero tenemos un poco de conciencia y tolerancia al cambio).

Podemos ver que es un tipo de petición Post, y luego vemos otros ahí que hace MVC, ahora bien, no hace sentido que tengamos que mandar al server una petición para saber si un dato es valido o no, claro, tenemos que hacerlo de todos modos, pero, si nosotros podemos quitarle una carga adicional al server preguntando en el cliente si el dato es válido, sería “mucho más mejor” (¿recuerdan lo que planteé hace un ratito de las comillas verdad?), bien, nosotros dentro de los scripts que nos regala la plantilla por defecto de DotNetCore tenemos, además del desfasado JQuery y Bootstrap 4, unas librerías de validación que creó Microsoft, bien, si nos fijamos en los scripts que se cargan en el _Layout, podemos ver que no tenemos esta librería, nosotros podemos agregarla acá, para que este disponible para todo el proyecto que use este Layout, pero, como es una librería que solo nos sirve para los Create y los Edit, mas no para los Index, y Details, no es correcto que sobrecarguemos de mas el ancho de banda del cliente, aunque sean simples kilos, si no hay necesidad de cargar algo, no lo cargues, en la web todo cuenta y nosotros tenemos que pensar siempre en minimalismo, por eso es que todos deberían aprender a desarrollar web sin importar que sean desarrolladores solo móvil, para que aprendamos la importancia de la optimización de recursos.

Pero no podemos agregarlo como solemos agregar los scripts normalmente, sino que debemos decirle que lo agregue con un “Section Script”, para indicarle que esto va a correr después de que se ejecuten todos los scripts del Layout, a ver para que entiendan bien este punto, si nos vamos al _Layout, podemos ver que inicia por el head, carga los CSS, carga el menú, tiene una sección donde renderiza el cuerpo de la pantalla que esta llamándose, entonces se para aquí y va a nuestra pantalla (en el

caso del create, la vista Create, en el caso del home la vista Index), se va a nuestro formulario donde lo primero que hace es que carga el tipo de dato o la clase que lo gobierna, viene carga el formulario, en caso de que tuviera scripts los comienza a cargar, pero si uno de mis scripts depende de jquery por ejemplo, va a fallar, a no ser que lo ponga en un Section Script, donde el entonces va a continuar con el Layout cuando llegue a esa parte, donde pintará el Footer y cargara los scripts, para después ir a cargar los que tenga el formulario en la sección.

```
@section Scripts {  
<script src="~/lib/jquery-validation/dist/jquery.validate.js"></script>  
<script src="~/lib/jquery-validation-nobtrusive/jquery.validate.unobtrusive.js"></script>  
}
```

Bien, referenciamos jquery validation y unobtrusive, probamos y veremos que tenemos todas las validaciones que teníamos en el server sin necesidad de viajar al servidor.

Netcore tiene una vista parcial que también podemos usar para estos fines si así lo deseamos.

```
@section Scripts {  
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}  
}
```

Resumen

En este módulo aprendimos a trabajar con validaciones basadas en DataAnnotation, los beneficios de estos es que podemos tener validaciones simples tanto en el server como en el cliente, todo lo que tenemos que hacer es decorar nuestras propiedades, con los atributos de validaciones, solo usamos dos atributos y creamos uno, pero existen muchos más, queda buscar en Google cuales son y para que sirven cada uno de ellos y ver en que te pueden servir para validar alguna otra cosa, que necesites para tu proyecto, como en nuestro caso, con esos tenemos, ya lo dejaremos ahí, si necesitamos más adelante uno que otros lo incluiremos pero ya sin explicarlo a profundidad.

Previniedo ataques comunes en la web.

En esta sección hablaremos de algunos aspectos comunes en las aplicaciones web en materia de seguridad o vulnerabilidades, que cada desarrollador web debe conocer e intentar protegerse contra ellas.

SQL Injection

Permite a los atacantes ejecutar código malicioso de SQL en tu aplicación, por ejemplo, pueden obtener toda la lista de los usuarios, eliminar registros, como el clásico chiste de la mujer informática que nombro a su hija Juana d 'delete from students'.

Dependiendo de las intenciones del atacante, los riesgos y vulnerabilidades pueden variar, ¿cómo funciona? ¡Fácil!, Imagina que tenemos la siguiente aberración:

```
var strSql = "select * from person where rnc=" + rnc;
```

Esto es una instrucción valida que incluso podemos mandar a ejecutar y Entity Framework nos traería una respuesta, por si ustedes decían que no se iban a montar en Entity y MVC por ahora ya que tienen muchas instrucciones y lógica de negocio hecha en ADO, bueno, pues pueden ir pasándolas para acá sin problemas, no sé cómo se hace en verdad ni me interesa mucho, pero en FrameWork se usaba una función SQL que se le mandaba al ORM, en NetCore no funciona de esa forma, pero existe, les toca averiguar por su cuenta como se hace esa aberración de

programación, pero nada, seguimos, imagina que ese rnc es obtenido por un campo de texto, donde el usuario introduce un número, o imagina que ese dato viene por un query string (por referencia en la Url, que por cierto, en el futuro tenemos que aprender a bloquear los ataques por esta vía, pero para eso falta) o lo almacenamos en un campo oculto al momento de cargar el modelo, el problema con esta instrucción es que la sentencia SQL se genera en tiempo de ejecución, y esta estará basada en lo que ingrese nuestro usuario por lo que el atacante puede usar esta oportunidad para modificar el final de la sentencia, en este ejemplo, el puede poner algo simple como esto `rnc = "22310826229 or 1 = 1"`.

En este caso, no importa si hay un usuario con la cedula indicada, el va a traer todos los registros que cumplan la condición `1=1`, que es igual a todos los registros de la tabla persona, la forma de protegernos de este ataque, además de usar siempre consultas basadas en entidad, es usar consultas parametrizables, en lugar de concatenación de cadenas, en este caso, si cambiamos nuestra operación por algo como lo que está debajo, el atacante no podrá modificar el resultado final, porque el valor pasado se convertirá en una sola instrucción de valor y en este caso lo mas que puede hacer es explotar porque el "or" no es un número, pero a lo sumo lo mas que va a hacer es no retornar nada.

```
var strSql = "select * from person where rnc=@rnc";
```

Lo que sea que el pase en el input, será almacenado en este parámetro, ahora, en el contexto de nuestra aplicación, ¿tenemos esta vulnerabilidad o no? No, porque no estamos generando sentencias SQL en tiempo de ejecución, nosotros estamos

dejando que Entity Framework haga el trabajo por nosotros, mientras nosotros solo le tiramos y esperamos respuestas a las representaciones de nuestra entidad.

Nosotros simplemente añadimos un objeto al DbSet y EF se encarga del resto, pero si queremos usar sentencias SQL en lugar de un DbSet y usando concatenación de cadenas podremos tener vulnerabilidad de inyección de SQL, así que esto es algo que debemos tomar en cuenta, tanto en este como en todos nuestros desarrollos futuros.

Cross-Site Scripting (XSS)

La siguiente vulnerabilidad es Cross-Site Scripting o XSS, lo cual consiste en la posibilidad que tiene el atacante de ejecutar scripts malicioso en la computadora de una víctima que le permita robar las cookies del usuario e interceptar su sesión, dependiendo de que tan sofisticado sea el atacante, pueden obtener la ubicación del usuario, acceso a la webcam, al File System y así sucesivamente, por culpa de la facilidad que le dejemos en nuestra aplicación, ¿cómo ocurre esto?, el ataque empieza cuando el atacante inserta script malicioso en la pagina de un webzine seguro, por ejemplo en un foro, ellos pueden responder o crear post, e incluir algún JavaScript, luego cuando la victima visita la pagina oficial y confiable el script malicioso se activa junto con la carga de toda la página, y es ejecutado en el navegador del usuario, ¿cómo podemos prevenir esto?

Escaping Content



La primer forma de deshabilitar el ataque XSS es escapeando el contenido, en ASP de lo que se envía desde el server para que se genere, lo que hacemos es decirle al navegador que trate el contenido como una simple cadena y que no lo interprete de ninguna otra forma, si el atacante pone un script en tu página la víctima no será afectada porque el navegador no va a ejecutar ningún script si esta propiamente escapeado, a sinceridad esa es una de las palabras que como que no tienen una traducción asertiva al español.

En el contexto de nuestra aplicación, no tenemos esta vulnerabilidad, porque no permitimos que los usuarios puedan insertar este tipo de contenido, por defecto MVC rechaza el JavaScript en los campos de entrada de datos aun cuando de forma intencional deshabilitemos la funcionalidad de escaping.

Otra razón, es que Razor por defecto hace scaped de contenido, la única excepción es cuando tú de forma explícita usas le método `HTML.Raw`, nosotros no estamos haciendo uso por el momento, de este método, aunque no tenemos este inconveniente porque usamos un Framework robusto y que tiene muchas vacanerias chulas para nosotros, pero es algo que debemos tomar en cuenta

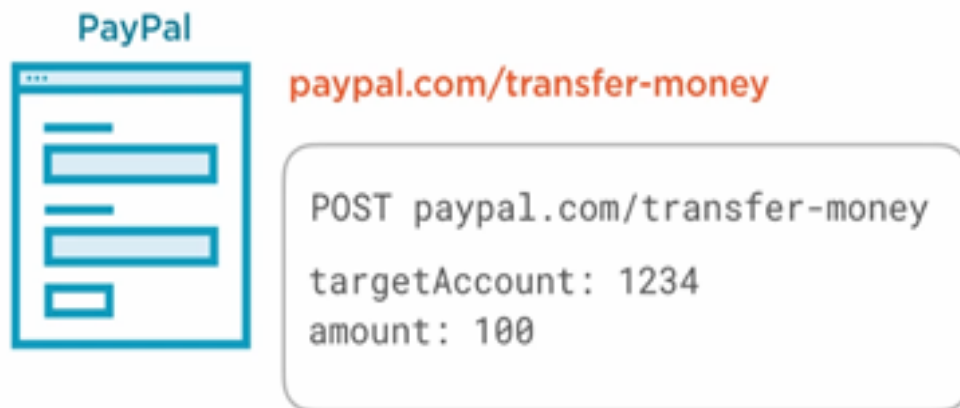
en el futuro si estamos trabajando con otras tecnologías, no hagan como el profesor que solo sabe .Net y pierde muchas picotas, aprendan varios lenguajes.

Cross-Site Request Forgery (CSRF)

La última vulnerabilidad de la que vamos a hablar es Cross-Site Request Forgery, que ni puta idea de cómo se traduciría, pero consiste en que permite al atacante efectuar acciones en nombre de un usuario sin su conocimiento, ¿cómo ocurre esto?

Imaginemos que Paypal hubiera sido desarrollado por hindúes, y por tanto fuera vulnerable a este tipo de ataques y tuvieran una dirección de envío de dinero como `paypal.com/transfer-money` que es un formulario donde ponemos un numero de cuenta y el monto y este se debitara de nuestra cuenta o `paypal.me/sgermosen/20` que sirve para enviar 20 dólares al profesor que tan bien se ha portado con ustedes :D

Cuando la víctima vaya a PayPal e inicie sesión y llene el formulario para enviar dinero a una persona, una petición post será enviada a esta Url, imagina que el cuerpo del request contiene dos campos, cuenta destino y monto.



Ahora, el atacante puede crear una página e intentar que la víctima entre en esta página, mandándole un correo por ejemplo de que te ganaste un dinero o que por error transferiste un dinero a alguien que si no reconoces ese cargo vayas a tal dirección, eso es lo que se conoce habitualmente como phishing, también puede ser ofreciendo alguna oferta, o algo gratis , en ese formulario siempre hay un botón que te va a dar el ultimo artículo que queda de una promoción x. El punto sería lograr que el usuario inicie sesión para que la transacción sea válida.

Cuando la víctima hace clic en alguna acción dentro de la página del atacante, hace un post del contenido oculto que está en el formulario que sería el mismo que vimos anteriormente, con la cuenta de destino modificada por supuesto, si vemos la imagen por ejemplo el pana está viendo memes e hizo clic en un botón siguiente o le dio clic al botón “Ganate un Iphone X”

Malicious Page



```
POST paypal.com/transfer-money
targetAccount: 666
amount: 100
```

Ahora, en este punto si la persona tenia su cuenta de PayPal con sesión iniciada, cuando haga clic en el botón se ejecutara el post con los datos indicados, lo que causara la transferencia, hacia la cuenta del atacante, lo interesante es que todo luce legitimo aquí , si PayPal intenta ver los registros, se va a dar cuenta que la transacción provino directamente del usuario que la efectuó, ósea, no hubo fraude real, tal cual si el hubiera llenado los datos por su cuenta, por eso es llamado de esta forma, porque el atacante arma la petición en otro sitio, así es como esto funciona, de todos modos este tipo de ataques no es solo para transferir dinero, un atacante puede efectuar acciones como si fuera el usuario real, estem, porque en verdad es el usuario real, en el contexto de nuestro proyecto, un atacante puede usar esta técnica para crear eventos falsos, en nombre de un coach, sin su consentimiento, puede enviar personas a un lugar de encuentro, donde no hay un evento y asaltar a todo el que vaya, como los narcos poniendo cepos en Pokémon Go, ¿cómo podemos prevenirlo?

En ASP es muy fácil, solo debemos agregar esta etiqueta dentro del formulario.

@Html.AntiForgeryToken()

Aunque en netcore no es necesario porque por defecto se implementa, ósea, siempre está ahí, aunque no queramos, a no ser que la desactivemos de forma explícita como sigue a continuación.

```
<form method="post" asp-antiforgery="false">
```

El Framework toma control de toda la complejidad y nosotros solo tenemos que darle esta instrucción, y decorar luego en nuestro controlador nuestra acción de Post con el atributo que sigue:

```
[ValidateAntiForgeryToken]
```

El lo que hace es que fuerza a que nuestra acción solo sea llamada desde un formulario que tenga esa instrucción.

El lo que hace es que nos genera un campo oculto con un token aleatorio, cuando llamamos el `HTML.AntyforgeryToken`, ASP MVC va a poner un campo oculto aquí (hidden field), pero eso es la primera parte solamente, el también crea una cookie que incluye una versión encriptada de este mismo token, vamos viendo en el navegador en la vista para ir corroborando, recuerden clic derecho inspeccionar elementos o presionar F12, luego si nos vamos a recursos, expandimos cookies, localhost, y pueden ver que tenemos un RequestVerification Token y el valor que vemos es un valor encriptado del que vimos en el hidden, cuando hacemos Post al servidor (como decoramos nuestra acción con el Antiforgerytoken si estamos en Framework, ya que no es necesario en NetCore, ASP MVC va a comparar estos dos valores, va a obtener el hidden del formulario, encriptarlo y compararlo con

el valor en la cookie, si hacen match significa que hay un Request legítimo, de otro modo es un ataque CSRF, ¿porque? Porque el atacante es capaz de robar la cookie del usuario, pero no va a tener acceso al campo hidden, puesto que la única forma de tener acceso al hidden, es cuando el usuario visita la página de forma legítima, podemos probar borrando el valor que tiene el hidden, que sería el equivalente a tener un Form en otro lugar y ver qué pasa.

Resumen.

En este modulo aprendimos de los tres principales modos de ataque en la web, hay muchas otras formas de ataque, pero estas son las principales que debemos conocer todos los desarrolladores y debemos saber cómo evitar.

Recuerden, nunca usen quertys directos y si lo van a hacer, háganlo con parámetros y no concatenación de string.

También aprendimos de XSS que es el que usan para insertar script en nuestro contenido, y así dañar a la computadora de la víctima, y finalmente aprendimos de CSRF que permite a los atacantes ejecutar acciones en nombre de un usuario.

Pimpeando-Enchulando ~~la máquina~~, digo, el proyecto.

A veces, dependiendo de la empresa o de nuestras necesidades, podemos tener un diseñador dedicado especialmente a todo lo relacionado con bonitura, ya saben, esa gente que tiene la vena y el arte para combinar colores, y hacer cosas asombrosas, pero en la mayoría de los casos no es así, normalmente tenemos que nosotros mismos meter mano y aunque no vayamos a obtener un resultado esplendido, debemos al menos tener las nociones básicas de usabilidad y diseño para que las cosas se vean decentes y atractivas para el cliente, posteriormente podemos delegar el diseño a alguien que meta mano en esa dirección para darle mayor atractivo.

En esta sección vamos a tocar brevemente los aspectos artísticos del desarrollo web, hablaremos de tipografía, colores, y un par de técnicas simples para mejorar el estilo de tu aplicación.

Escogiendo un color

Nuestra aplicación de eventos se ve bastante aburrida, lo primero que debemos hacer para que se vea decente es escoger un color primario que transmita el mensaje central de nuestra aplicación.

Y eso es lo que debemos reflejar en nuestra barra de navegación, ejemplo, el color de Facebook ¿cuál es?..... Azul, ¿verdad?, pero, ¿ellos usan el azul en todas partes? Por supuesto que no, así que ojo con esto, tu color primario, no necesariamente es el que mas vas a tener en tu app, pero si es el que debe destacar del


resto de colores que vayas a usar, y el mejor lugar para destacar un color es la barra de navegación, así que será lo primero que vamos a cambiar.

Ojo, nunca escojas un color solo porque sea tu color favorito, el color que vas a escoger debe reflejar el significado de tu aplicación, no tus gustos personales.

Algo a tomar en cuenta es que los colores en diseño son muy subjetivos (que va a depender de la perspectiva de quien lo vea), lo que evoca una reacción en una persona, puede provocar una totalmente distinta en otra, lo cual puede ser por gustos personales o por temas culturales, ejemplo para nosotros el negro es malo, muerte, mientras que para algunos occidentales es el blanco, para nosotros el amarillo es alegría, para otras culturas el amarillo representa la enfermedad. Pero nada, basémonos en los estándares para nuestra región según la industria y los estudios científicos que se han realizado respecto al tema en este lado del mundo.

Basados en estos estudios, podemos sub categorizar los colores principales en tres categorías centrales.

Colores cálidos, fresco y neutrales. Hay más sub clasificaciones, pero por lo pronto para la web estas son las que más se usan, un dato, está prohibido usar la versión chillona de un color, lo que nunca me ha quedado claro y a sinceridad nunca he investigado, es si esta “prohibición” es por buenas practicas o hay una ley que realmente dicta que no puede ser así, se los dejo de tarea.

			Calidos
			Frescos
			Neutros

Podemos ver el significado de los colores detallados de la siguiente forma.

	Fuego, poder, energía, pasión, amor
	Energía, diversión, emoción
	Felicidad y alegría
	Calma, sanación, naturaleza y protección
	Verdad, lealtad, amistad, calma
	Romance, lo femenino, amor, belleza
	Poder, sofisticación, formalidad o Muerte, lo malo, lo misterioso
	Limpieza, perfección, pureza
	Conservador, formalidad

La idea siempre es escoger un color principal y dos secundarios o de soporte, nunca se debe escoger mas de 5 colores en tu aplicación, una de las razones que mató a Hi5 fue su principal atractivo, el hecho de que todo el mundo tenía un muro estrambótico.

Bueno, para EventsCoach vamos a escoger estos dos.



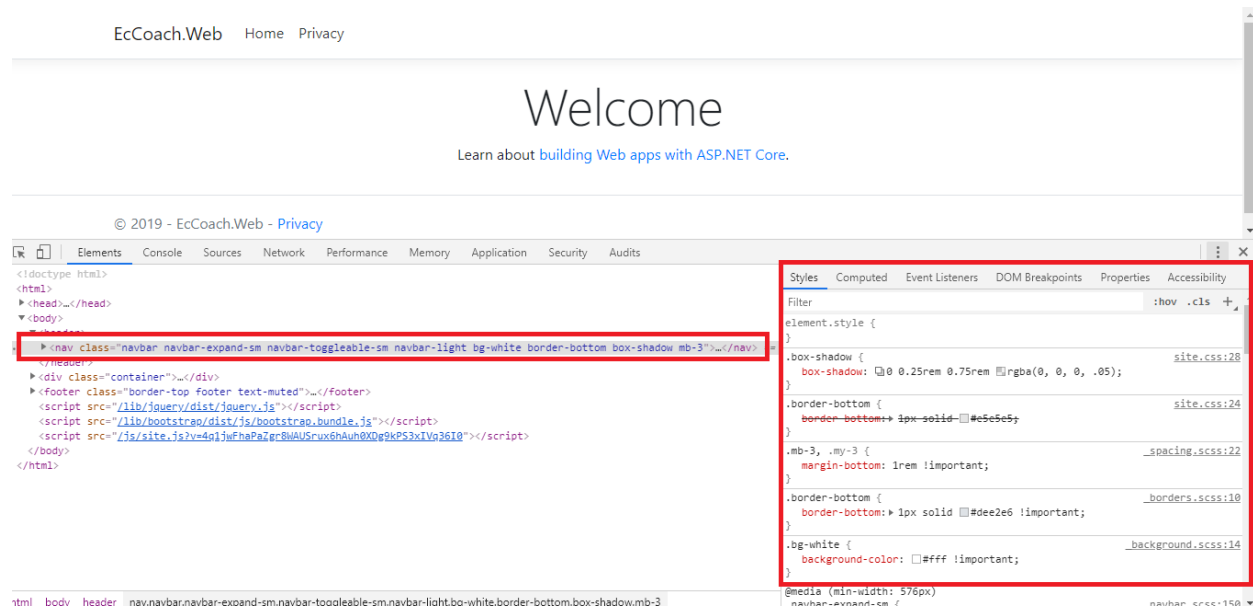
Porque Events es acerca de eventos, que, si bien pueden ser formales, deben ser excitantes, quiero evocar pasión, diversión, no monótonos eventos aburridos a los que vamos por obligación, así que usamos la combinación de estos como nuestro color primario.



[Sobre escribiendo los estilos de Bootstrap](#)

Ok, una vez escogido nuestro color principal, ahora toca cambiar el color de nuestra barra de navegación, corremos la aplicación, vamos a la barra de navegación clic derecho e inspeccionar elementos.

Podemos ver el HTML de los elementos que componen nuestra pantalla en el lado izquierdo, del lado derecho podemos ver las cosas que le han sido aplicadas a ese elemento, en este caso en particular, como del lado derecho tenemos seleccionado el nav bar, entonces del lado izquierdo podremos ver específicamente los estilos aplicados a ese elemento en cuestión.



Acá vamos a buscar un estilo que ha sido aplicado al fondo (background) del nav bar, asegurémonos de estar situados en el padre, ahí podemos ver el color seteado a blanco hueso, igual para el color del borde, bueno, como el color que yo escogí es FF4342, vamos a reemplazarlo.

☒ **background-color:** ■ #FF4342;

Si nosotros no tenemos idea de que color es el que queremos, podemos ir a Google y escribir color chooser, donde nos sale una herramienta que nos da el hexadecimal del color que seleccionemos, buscamos uno chulo y copiamos y reemplazamos.

Bien, los links no se ven porque el color negro no contrasta con el rojito raro que le acabamos de poner, pero antes de hacer alguna otra modificación, debemos agregar estos cambios a nuestro archivo de estilos, porque esto que acabamos de hacer es solo para ver como queda y si es el color que nos gusta, porque si le damos a F5, miren como se pierde el cambio, así que nos vamos a seleccionar la pieza de código que va desde el nav bar, el estilo css, vamos a Visual estudio.

Vamos al archivo Site.css que podemos encontrar en la carpeta wwwroot luego en “css” y ahí está nuestro archivo.

Podemos inspeccionar los elementos que él ya modifica donde podemos ver estilización a elementos como el footer entre otros, y acá vamos a pegar lo que copiamos, de esta forma, como el site.css se carga después de los estilos de Bootstrap, lo cual podemos corroborar en el _Layout, entonces lo que pongamos acá va a sobre escribir lo que venga de donde sea.

Sin embargo, la forma en que se estructura la plantilla por defecto de VS para Core, es que él le asigna una clase llamada bg White al nav bar, yo puedo sobre escribir la clase bg White, pero en verdad yo prefiero remover esta clase y asignarle el background con el color al nav bar, porque no tiene sentido que una clase que se usa para dar un color blanco, sea sobre escrita para que el color blanco sea rojo, así que vamos al layout, removemos esa clase y en el site.css ponemos la siguiente línea de código, probamos y verificamos que todo esté como deseamos.

```
.navbar {  
    background: #FF4342  
}
```

Nosotros podríamos modificar el archivo de Bootstrap, pero eso no es muy práctico, en especial si mañana queremos actualizar a la versión mas reciente y se nos van a perder todos los cambios.

Volvemos al navegador, tras guardar, refrescamos y podemos ver el cambio, ahora vamos a hacer que los links sean blancos, para eso clic derecho en un link de esos, inspeccionar elemento podemos ver los estilos aplicados, copiamos esta sección volvemos a vs, en nuestro site.css, pegamos y cambiamos el color a blanco, #fff, si el nombre de nuestra aplicación no cambia a blanco, es porque tiene una clase distinta, así que vamos a seleccionarlo inspeccionar elemento, lo cual podemos hacer también dándole clic a la flechita de inspección y luego un clic normal en el elemento, hacemos lo propio copiando el estilo, nos vamos al site.css, cambiamos el color y probamos.

El igual que con el navbar la plantilla de VS para Core, hace lo mismo con los links que le añade una clase especial, por lo que como a mi no me gusta así, yo le tiro directo la clase del elemento y ahí le sobre escribo y removemos en el layout los elementos que sobran.

```
.navbar-light .navbar-brand {  
  color: #fff;  
}  
.navbar-light .navbar-nav .nav-link {  
  color:#fff;  
}
```

Escogiendo una tipografía (Fonts)

Bien, ya cambiamos el color central, ahora debemos escoger una tipografía que se vea bien, porque la que trae por defecto aunque

es asombrosa, no refleja la intención de mi aplicación, en verdad si la refleja, pero tengo que mostrarles como se cambian estas cosas, porque si no, no sería una guía con los reales trucos, bueh, a la hora de realizar esto, hay varias opciones para escoger.

google.com/fonts (free)

fonts.com

fontsquirrel.com

typekit.com

A pesar de que el de Google es gratis, existen muchos fonts que solo existen para fines comerciales, aunque no lo crean, los tipos de letra son un negocio, usted no puede hacer un letrero con las letras usadas en transformer sin pagar derechos de autor o pagar uno de estos servicios que lo hacen por usted, pero acá nos vamos a ir con la opción al gratin, las pequeñas diferencias son las que hacen la diferencia y le dan personalidad a tu app, yo en verdad no tengo el mínimo arte para esta vaina, así que por lo regular escojo el que sea mas popular, el que se me sea más fácil leer en la distancia o el que me diga el cliente, en el caso de nuestra app probaremos Open Sans, Lato, y asul, a pesar de que nosotros tenemos Font awesome integrado, es preferible hacer un uso mas limpio solo de lo que necesitamos, por lo que vamos a Google.com/fonts, seleccionamos los Fons que vamos a probar, en el preview vemos como se ve, en la pestañita que esta abajo le damos clic para ver las que tenemos seleccionados, nos vamos a custom para editar los tamaños en que queremos descargarlo, no sugiero escoger uno inferior a 400 porque cuando tenga que estirar, se va a ver muy pixelado, seleccionamos bold 700 también. Luego podemos descargarlo o nos quedamos con embebed para que lo consuma siempre desde la web de Google.

Ahora vamos a simplemente añadir este link a nuestro Layout, y luego en nuestro CSS referenciar estos fonts por sus nombres, seleccionamos la línea de importación, vamos a VS, nos vamos al layout, la pegamos antes de que se rendericen los css que ya tenemos en nuestro proyecto por defecto.

Si analizamos el link podemos ver que hace referencias al api de Fonts, nosotros podríamos visitar ese link y guardar el archivo en nuestro proyecto, pero, la probabilidad de que ese api se caiga es nula, más fácil se cae nuestra app, así que mejor deje esas cosas por un tema de optimización, delegadas a otro.

Para poder ver la diferencia, deberemos correr la aplicación, abrir una nueva ventana donde abrimos también la aplicación, hacemos las modificaciones en una de las ventanas para poder notar las diferencias.

Probamos cambiando el Fons aplicado al formulario, probamos con cada uno de los estilos que decidimos probar.

En mi caso como ya yo sé cómo se ven y me gusta más Lato no voy a hacer este proceso tan largo, así que lo que voy a hacer es ir directamente a visual estudio, borraremos los fonts que no voy a usar.

Luego vamos a Bootstrap.css, buscamos con ctrl + f, la palabra helvética que es el Fons por defecto que usa Bootstrap, copiamos la parte que tiene relación con esto, vamos a nuestro site.css y

sobre escribimos delante el Fons que nosotros queremos, la razón para no sobre escribir los que ya están es que en caso de que el api de Google se caiga, podamos usar el que viene por defecto sin problema.

```
font-family: -apple-system, BlinkMacSystemFont, Lato, "Segoe UI", Roboto, "Helvetica Neue", Arial, "Noto Sans", sans-serif, "Apple Color Emoji", "Segoe UI Emoji", "Segoe UI Symbol", "Noto Color Emoji";
```

Mejorando la apariencia del formulario

Vamos ahora a nuestro formulario Create, podemos ver que el no se ve mal, pero puede ser mejorado, se ve todo como muy pegado, como muy monótono, muy formal, me gustaría que haya un poco más de espacio entre los elementos, si nos vamos a el campo “venue” por ejemplo y le damos clic derecho inspeccionar elemento, podemos ver si situamos el mouse arriba que hay dos áreas, una azul y una naranja, esta línea naranja es un padding, o espaciado entre el elemento, form group, lo primero que haremos es separar este margen para que se vea más separado.

Así que vamos a incrementar el padding a 20, copiamos la clase con la modificación, y pegamos esta sobre escritura en nuestro site.css

Ahora vamos con los labels, los cuales están muy pequeños para mi gusto que soy una persona ciega, bueno, vamos al label, nos vamos a **compute**, después nos vamos al **magnificador (la flechita que se pone cuando nos ponemos al lado del elemento)**, y podemos ver que el body tiene seteado un Font zize de 14, copiamos esta parte y vamos a nuestro site.css para actualizarlo con un tamaño de 17.

Hacemos lo propio con el texto de las cajas de texto, que, si nos vamos a inspect element, y al css podemos ver que en la clase form-control, esta seteando el Font size a 14, lo modificamos también seleccionamos y nos vamos al site.css a hacer estos cambios.

Otra cosa que vamos a modificar ya que estamos aquí, es el padding, el primer numero determina el padding de arriba y de abajo y el segundo de la izquierda y derecha, podemos tener padding específicos también, pero eso no es recomendable, siempre se debe tratar de buscar la uniformidad.

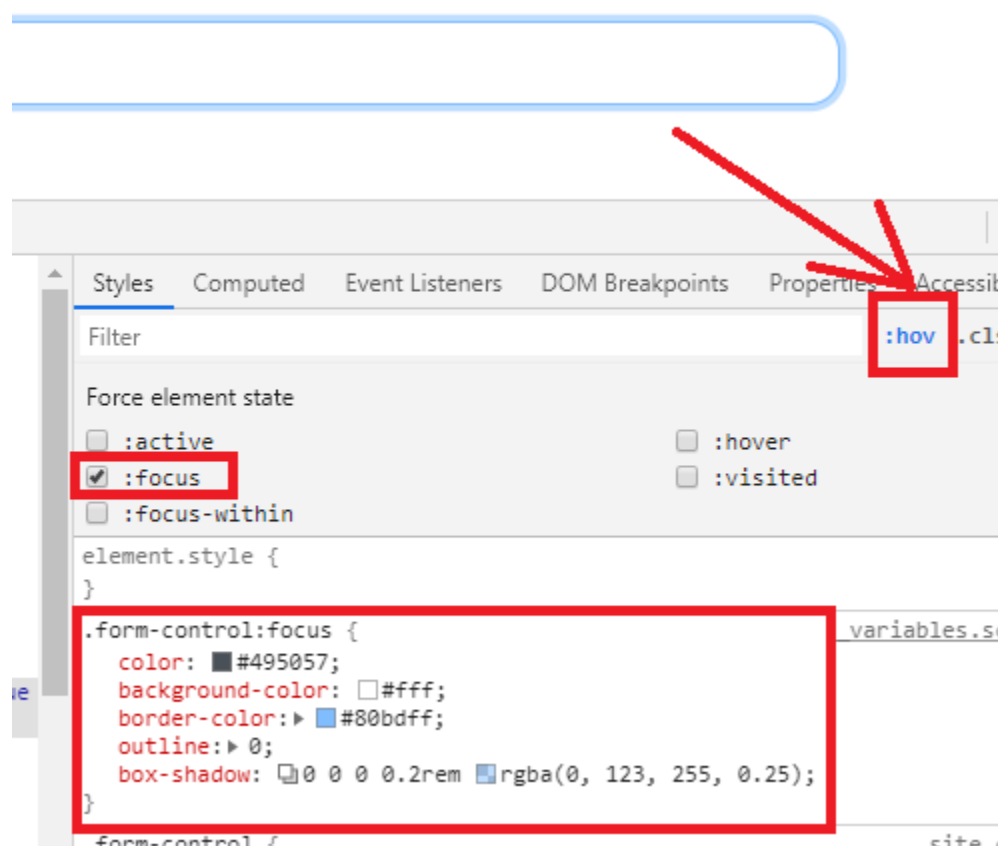
De igual forma algo que podemos cambiar es el border-radius, para que tenga un estilo mas curvado, lo incrementamos a 10px.

Ahora hacemos ahora la inspección al botón, cambiamos el Font-size y listo.

```
body {  
    ...  
    font-size:17px;  
    ...  
}  
  
.form-control {  
    padding: 20px ;  
    font-size: 17px;  
    border-radius:15px;  
    /*padding: 0.375rem 0.75rem;  
    padding-top: 0.375rem;  
    padding-right: 0.75rem;  
    padding-bottom: 0.375rem;  
    padding-left: 0.75rem;*/  
}  
  
.btn {  
    font-size:17px;  
}
```


Bien, ahora que ya tenemos estos aspectos artísticos hechos, nos podemos también cambiar el efecto hover que hace cuando entramos a un Textbox, puesto que el que trae por defecto es como muy sobrio, si se fijan hay un bordeado sombreado cuando nos paramos dentro, pero en la actualidad se aprecia más un diseño plano.

Así que vamos a inspeccionar elementos, por defecto nosotros no podemos ver propiedades que se aplican cuando ocurre un evento, para poder verlas debemos hacer clic lo que esta al lado de la caja de filtro, y aquí podemos ver cualquiera de los estados que tiene el elemento seleccionado, seleccionamos focus, podemos ver que el form-control, cuando activa el evento focus, se le aplica este css.



Podemos ver dos elementos de sombras que se le aplican y si los quitamos y visitamos el elemento podemos ver que se fue la sombra, podemos cambiar el color del borde si deseamos o removerlo, nosotros lo removeremos.

Volvemos a visual studio seteamos la propiedad box-shadow en none, y -webkit-box-shadow también en none (este último solo para la versión 3 de bootstrap)

```
.form-control:focus {  
    box-shadow: none  
}
```

Drop Down List en la barra de navegación.

No sé si recuerdan en nuestra presentación inicial, donde habíamos quedado como seria nuestra experiencia de usuario, se que no, pero tranquilos, habíamos dicho que tendríamos una barra donde veríamos el nombre del usuario logueado, y cuando hiciéramos clic tendríamos un menú desplegable con varias opciones, tales como los coach que estoy siguiendo, eventos a los que voy a asistir, y salir del app, para ver como se hace un ddl lo primero que vamos es ir a getBootstrap.com, nos vamos a componentes, y luego nos vamos a navbar, ahí podemos visualizar un ejemplo de varios que ellos tienen, de cómo construir una, buscamos lo que ellos usan para construir de manera especifica el elemento que queremos, y copiamos solo esa parte.

Nos vamos al layout, donde están nuestra nav bar actual, podemos ver un elemento de ul que dentro tiene varios li que son listas de objetos y acá es donde vamos a pegar lo que copiamos hace poco. Antes de hacer modificaciones nos aseguramos de que funcione.

```

<li class="nav-item dropdown">
    <a class="nav-link dropdown-toggle" href="#"
id="navbarDropdown" role="button" data-toggle="dropdown" aria-haspopup="true" aria-
expanded="false">
        Dropdown
    </a>
    <div class="dropdown-menu" aria-labelledby="navbarDropdown">
        <a class="dropdown-item" href="#">Action</a>
        <a class="dropdown-item" href="#">Another action</a>
        <div class="dropdown-divider"></div>
        <a class="dropdown-item" href="#">Something else here</a>
    </div>
</li>

```

Bien, ahora si vamos a modificar, lo primero que quiero es reemplazar el texto “Dropdown”, por el nombre email del usuario que esta logueado, eso se hace con el User que es una clase transversal a todo el proyecto, identity.Name, que es el username del usuario y es lo que debemos mostrar, como según nuestra lógica el username es el mismo que el email, podemos usar name sin problemas aquí.

```
@User.Identity.Name
```

También vamos a poner uno de estos links que haga referencia a el formulario de deslogueo/cerrar sesión.

```

<a class="dropdown-item" asp-area="Identity" asp-page="/Account/Logout">
    Close Session/Exit
</a>

```

Renombramos dos elementos de los que están ahí, por ahora no apuntarán a nada, pero tendrán el texto de a donde van.

```

<a class="dropdown-item" href="#">Events I'm Going</a>
<a class="dropdown-item" href="#">Coach I'm Following</a>

```

Bien, ahora si nos vamos al dropdown vemos que no cuadra con la experiencia de usuario que tienen los demás links, como que no va con nuestro tema (ojo, este fallo solo pasa en la versión 3 de

Bootstrap así que lo puedes saltar), por lo tanto cambiémoslo inspeccionamos elementos podemos ver el background color del elemento, seleccionamos el color picker para hacer clic en la barra de navegación y escoger el color correcto, pero, luego en el espectro de colores, yo voy a escoger uno un poco mas sobrio luego en el slider de abajo podemos moverlo un poco mas para reducir la opacidad, copiamos ese estilo y nos vamos a site.css lo ponemos cerca de los otros estilos que le agregamos a la barra de navegación, por un tema de ser ordenados.

En Bootstrap 4, nosotros tenemos que agregarle una clase porque ya en esta versión los li heredan el color del padre.

```
li.nav-item.dropdown:hover {  
    background: rgba(205, 40, 39, 0.55);  
}
```

La razón para usar la combinación de rgb, en lugar del valor hexadecimal es que nuestra versión de css no soporta el formato hexadecimal para los gradientes.

Otra cosa que vamos a cambiar es la sombra que esta alrededor del menú cuando se despliega, para seleccionarlo tenemos que situarnos en el padre para poder buscar a sus hijos, podemos ver que hay una clase dropdown que tiene un elemento menú al que se le aplica un box shadow, así que vamos al site.css y seteamos estos valores en none. En Bootstrap 4 no tendremos este problema porque viene sin el por defecto.

```
.dropdown-menu{  
    box-shadow:none;  
}
```

Un dato de usabilidad, nunca debes tener un link que vaya a home, si vas a tener el nombre de tu app, por convención el nombre de tu app debe llevarte al home, así que eliminemos eso.

El nombre de nuestra app, debe resaltar no puede tener el mismo formato que el de los links alternos, aunque no pasaría si usáramos una imagen o logo oficial.

Nos vamos a navbar donde podemos ver que hay un elemento Brand que es el que define la clase que se le esta asignando, fíjense que acá podemos agregar atributos que no nos aparezcan, en este caso es Font-weight:700 porque el Font que seleccionamos Lato, solo tenemos la normal y la Bold de 700.

```
.navbar-light .navbar-brand {  
  color: #fff;  
  font-weight:700;  
}
```

Resumen

Bien, ya con estos pequeños detalles, miren que tenemos una aplicación completamente distinta, no se ve lo mejor del planeta, pero tiene un gran trabajo detrás, que lamentablemente no te van a pagar, XD, el punto es que ahora tenemos una interfaz mas amigable, mas divertida, mas usable, posteriormente trabajaremos otros truquitos para los fines de darle más chulería.

Aprendimos a como se mejora la apariencia de una aplicación, la idea de este modulo no es hacer un tema que todos amen, sino aprender esos truquitos que te van a hacer entender como

funciona tal o cual plantilla que vas a comprar, porque seamos sinceros, no tenemos esa vena y no vamos a hacer diseños que valgan la pena, así que compre su plantilla, punche para que aprenda como funciona por detrás y saber que poder cambiar.

Además, que esos truquitos que te hacen destacar por encima de esos desarrolladores que solo saben tirar código.

Vimos el significado de los colores, aprendimos a usar fonts que también demarcan una personalidad, para nuestros sistemas, aprendimos que formularios pequeños son mas serios mientras que formularios grandes son más usables.

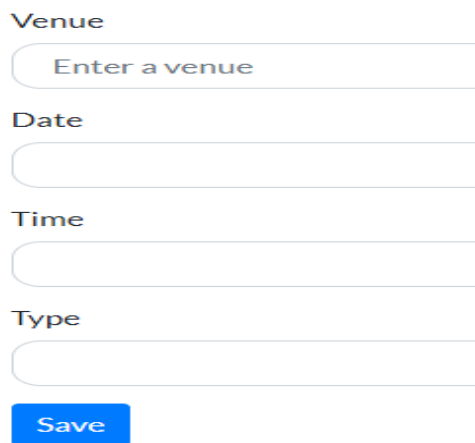
Aprendimos a sobre escribir elementos de Bootstrap, entre otras cosas más.

Mejores prácticas de usabilidad.

Construir un software no es solo escribir código, nuestras aplicaciones deben ser fáciles de usar, e intuitivas, cuando alguien dice que su hijo es un genio porque sabe usar una Tablet o un celular, un empleado de Apple y Google se suicidan al unísono, en esta sección vamos a estar hablando de los principales principios de usabilidad, que todos deberíamos saber, solo una advertencia, usabilidad requiere todo un curso y la lectura de muchos libros que hablan del tema para conocer las diferentes propuestas que dan los expertos, así que aquí solo hablaremos de lo básico.

Etiquetas (labels)

Si echamos un vistazo a como se ve nuestro formulario, podremos ver algo como esto, etiquetas cortas, claras y concisas.



Venue

Date

Time

Type

Save

Ahora imagine que nosotros cambiamos eso y mostramos un formulario como este.

Create

Where its going to happend the event

When, talking about the date, its going to happend this event

What its de Time to start this event

What its the kind of event than it is

Save

Inicialmente tu puedes pensar que como son mas descriptivos, es mejor y mas útil, pero actualmente es todo lo contrario, a no ser que estés trabajando con la ancianita que le teme al cambio, este formulario es demasiado ruidoso, los usuarios en la actualidad no tienen mucha paciencia, constantemente estamos siendo bombardeados con información, desde el momento en que nos despertamos hasta que nos vamos a acostar en la cama, por lo que menos queremos es estar leyendo mas de la cuenta, por eso en necesidad de reducir el ruido en nuestro formulario o en tus páginas en general, tú puedes incrementar la usabilidad de tus usuarios, y el enganche de estos con este principio, así que, este principio nos dice que los labels, deben ser una o dos palabras, no más, que sean concisas, no use palabras ambiguas o misteriosas, por querer privar en fino.

Keep them short

Be concise

Don't use ambiguous words

Ahora hablemos de alineación y balanceo, bueno, esa no, en términos de alineación, tenemos tres posibilidades.



Vamos a tocar cada uno de ellos, primero el menos recomendado.

Left Align

A form example for left alignment. It features three input fields stacked vertically, each preceded by a label: 'First Name', 'Age', and 'Post Code'. A thick vertical red line is positioned to the left of the 'Age' field, extending upwards past the 'First Name' field and downwards past the 'Post Code' field. Below the input fields is an orange 'Register' button.

La alineación a la izquierda requiere mas poder cerebral para ser interpretada y es más lento, porque constantemente estamos asociando cosas, que están una al lado de la otra como relacionadas, cuando pones un espacio entre una etiqueta y un campo de texto tu cerebro se ve obligado a trabajar más duro de la cuenta para relacionar esos dos elementos, ahora vamos a ver como se mueven los ojos de los usuarios, cuando intentan llenar este formulario.

A form example for left alignment. It features three input fields stacked vertically, each preceded by a label: 'First Name', 'Age', and 'Post Code'. A blue circle highlights the 'First Name' label. Below the input fields is an orange 'Register' button.

First Name

Age

Post Code

Register

First Name

Age

Post Code

Register

First Name

Age ??

Post Code

Register

First Name

Age

Post Code

Register

The image displays three sequential screenshots of a registration form, illustrating the concept of eye movement in user interface design. Each form contains three input fields: 'First Name', 'Age', and 'Post Code', followed by a 'Register' button.

- Top Screenshot:** The 'Post Code' label and its corresponding input field are highlighted with a blue circle, indicating the current point of focus.
- Middle Screenshot:** The 'Post Code' input field is highlighted with a blue circle, and a blue arrow points from the 'Post Code' label to the input field, showing the visual path of the user's eye.
- Bottom Screenshot:** The 'Register' button is highlighted with a blue circle, indicating the final point of focus after the user has completed the form.

Como puedes ver, son demasiados movimientos de los ojos y esa es una de las razones por la cual la alineación a la izquierda hace mas lento a los usuarios.

Right Align

First Name

Age

Post Code

A vertical red line is positioned to the left of the input fields, aligning the labels to the right.

Ahora veamos un ejemplo de alineación a la derecha, las etiquetas y los campos están muy cerca, como una pareja, no hay espaciado, pero si vemos el movimiento de ojos, podemos ver el mismo patrón que en el anterior, no es ideal, pero es mucho mejor.

Top Align

First Name

Age

Post Code

The labels and input fields are aligned to the top, with no horizontal spacing between them.

Finalmente vamos a ver la última alineación, aquí también las etiquetas y los campos están muy cerca, pero ¿cómo se hace el movimiento de ojos?

The image displays three identical examples of a top-down form layout, arranged vertically. Each example consists of a light gray rectangular container. Inside the container, the labels 'First Name', 'Age', and 'Post Code' are aligned to the left. To the right of each label is a corresponding input field: a wide rounded rectangle for 'First Name', a smaller rounded rectangle for 'Age', and another small rounded rectangle for 'Post Code'. Below these three input fields is a red rounded rectangle button with the white text 'Register'. In each of the three examples, a small blue circle highlights a different element: the 'First Name' label in the first example, the 'First Name' input field in the second example, and the 'Age' input field in the third example.

etc.

Podemos ver que el usuario solo tiene que mirar hacia abajo, pero, una desventaja es que se requiere mas espacio vertical, así que, si el espacio vertical es un problema para ti, mejor usa la alineación a la derecha, pero siempre que se pueda usar la alineación top-down.

Campos

La mejor y más importante practica del mundo mundial es: “reduce la cantidad de campos en un formulario”, seamos honestos, nadie quiere formularios grandes y complicados, menos es más, captura solo lo que es necesario, una vez que te hayas desecho de los campos innecesarios, revisa si hay otros campos opcionales que también puedas remover, es mejor capturar solo lo que es realmente requerido, después de eso si aun tienes campos opcionales, sepáralos visualmente de los que son obligatorios, para que el usuario no tenga que perder el tiempo, llenando todos aquellos campos que no son requeridos.

Luego debes agruparlos en un orden que realmente tenga sentido lógico, y si es necesario, rompe tu formulario en un wizard, muéstrale, obviamente, un indicador al usuario para notificarle del progreso, si hay algún campo que requiera un formato específico, debes recalárselo al usuario y finalmente siempre pon el foco en el primer campo de tu formulario, así le ahorras un clic al usuario y créeme que te lo va a agradecer.

Reduce the number of fields

Avoid optional fields

Separate mandatory and optional fields

Group related fields

Specify the format

Set focus on the first field

Bien, volvemos a nuestro proyecto, para implementar algunas de las practicas que ya indicamos acá, lo primero que vamos a hacer

es indicarle al usuario que campos son requeridos, una práctica comúnmente aceptada es ponerle un asterisco en rojo al lado del formulario para indicar esto o una etiqueta que diga requerido entre paréntesis, pero, en nuestro caso, como todos los campos son requeridos, es contraproducente y muy ruidoso que todos los campos tengan asterisco, además de que esto desvirtúa el sentido de diferenciación, entonces para esto, lo que nosotros vamos a hacer igual, por temas de usabilidad, es poner una etiqueta arriba del formulario que indique que todos los campos son requeridos, la palabra requerido debe estar en negrita, así que vamos a seleccionar requerido y vamos a usar un truco, solo valido si tienes resharper instalado, ctrl + j si tenemos resharper, te permite encerrar lo que esta seleccionado en algo. En este caso escogeremos etiqueta y escribimos strong, es mucho más sencillo que crear la etiqueta y luego cortar y pegar requerido dentro de él. Pero como no tenemos esa herramienta, nos toda escribirlo a mano.

Bien, si corremos el app y observamos podemos ver que se ve un tanto feíto porque hace contraste con el label del primer campo, así que vamos a aplicarle un estilo y para eso le añadimos la clase alert podemos poner la de info y así veremos que todo se ve mas bonito.

```
<p class="alert alert-info">  
    All fields are <strong>required</strong>  
</p>
```

Ahora pondremos el foco en el primer campo, lo único que tenemos que hacer es agregarle al campo la propiedad de auto focus, que es una funcionalidad de html5

```
<input asp-for="Venue" class="form-control" autofocus="autofocus" >
```

Y finalmente vamos a agregar una marca de agua, para indicar el formato en que queremos que nos inserten la fecha y la hora, eso se logra con el atributo placeholder que ya lo habíamos visto previamente.

```
<input asp-for="Date" class="form-control" placeholder="eg 15/12/2019" >
```

Acciones

Ahora vamos a hablar de acciones, cada formulario debe tener una sola acción primaria, que siempre debe estar visualmente distinguida de las demás, las acciones primarias son links o botones que realizan una funcionalidad final, como guardar.

Acciones secundarias como cancelar o volver atrás en general es mejor evitarlas, si de verdad necesitas usarlas, visualmente sepáralas, de las acciones primarias.

Aunque nosotros no tenemos cancelar, porque nosotros no programamos para gente bruta y nuestros usuarios deben aprender a la mala que si usted le dio a nuevo es porque quiere registrar un nuevo evento, vamos a imaginarnos de forma hipotética que tenemos un botón cancelar como este:

Create

All fields are **required**

Venue

Date

Time

TypeId

Los dos botones están usando el mismo color y el mismo tamaño, nosotros deberíamos usar un elemento distintivo, como usar un background diferente, algo como esto:

O podemos convertirlo en un link para restarle importancia.

[Cancel](#)

En términos de alineación, alinea siempre las acciones primarias conforme a tus campos, lo cual permitirá reducir el movimiento de ojos, miren este ejemplo que obviamente no nos aplica, porque aunque no sabíamos de diseño antes de esta clase, al menos sabíamos de sentido común.

Venue

Enter a venue

Date

eg 15/12/2017

Time

TypeId

????

Save

Hay un tema que está más que estudiado científicamente, que nos dice que los elementos arriba a la izquierda de una pantalla captan más nuestra atención, mientras que lo que está abajo en la derecha capta menos nuestra atención, por eso ten en cuenta esto a la hora de desear captar la atención del usuario en algo.



- Each form should have a primary action
- Avoid secondary actions if possible
- Otherwise, visually separate them
- Align primary actions with input fields

Validaciones

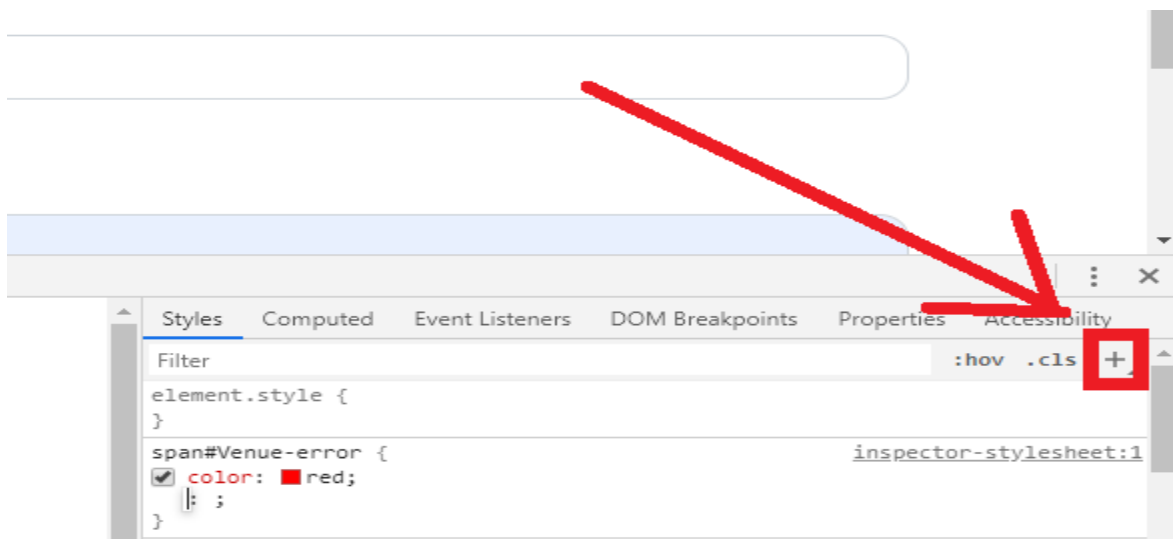
Finalmente hablemos sobre validaciones, cuando el usuario introduce data invalida que no cumple nuestro formato, debemos proveer mensajes claros y concisos, los mensajes de validaciones, deben estar ahí para ayudar al usuario, muchas veces nos incomodamos cuando llenamos un formulario y ponemos una información incorrecta en un campo y tenemos que ponernos como locos a llenar información de forma aleatoria a ver si la pegamos, así que debemos ser claros y concisos siempre, esto no es opcional, la segunda practica es usar rojo, pero acá hay un tema, los expertos recomiendan el rojo, pero, mi experiencia, lidiando con usuarios en diferentes empresas que parecen orangutanes, es que eviten los rojos, yo les voy a dar la recomendación porque así lo dicen las mejores practicas de los expertos, pero en la vida real en nuestra jungla, cuando salta un rojo, de una vez el usuario lo asocia con error del sistema, pero tranquilos, eso solo pasa en tercer mundo, aquí los usuarios te llaman sin leer y te dicen que el sistema le esta arrojando un error, yo uso el azul o el negro, pero nada, seguimos, la regla de usabilidad nos dice también que debemos usar el verde para mensajes de información exitosa.

Tu meta debe ser siempre reducir el estrés del usuario, no pongas un mensaje de “se ha creado tu cita satisfactoriamente” en un pop up rojo, otra cosa para reducir los errores, es proveer valores por defecto, si tu tienes que llenar un campo de precio de compra y uno de impuestos, entonces de forma automática calcula el precio total de compra, si puedes cargar la ubicación del usuario en un campo donde tiene que poner el país, hazlo,

- Provide clear validation messages
- Use red to indicate errors
- Use green to indicate success
- Provide smart defaults

Nada, ahora deberíamos agregarles el color rojo a nuestros mensajes de error en el llenado del evento.

Si queremos corregir esto, verificamos cual es la clase aplicada al elemento de error, y en el site.css sobre escribimos el color para este. Pero primero y para mostrarle una funcionalidad extra de Chrome, vamos a situarnos arriba de la clase que deseamos y le damos clic al botón adicionar que esta en la parte derecha.



De esta forma le indicamos a chrome que deseamos agregar modificaciones a esta clase, dentro ponemos color:red y probamos, y también haremos el Font bold, después copiamos la clase y modificamos tras añadirla al site.css.

```
span.field-validation-error {  
  color: red;  
  font-weight: bold;  
}
```

Esta clase no es de Bootstrap, sino de MVC, así que la ponemos al principio del archivo, después de los que ya estaban ahí antes de comenzar a trabajar con este archivo. No es obligatorio que sea así, pero es bueno que los archivos, estilos, scripts, y todo lo que se relacione, este junto.

Resumen

En esta sección vimos los principales principios de usabilidad, y crear formularios amigables para el usuario, en el mundo real hay personas que se dedican a esto, pero no en todo equipo se pueden dar el lujo de contratar a uno para que se involucre en cada proyecto, así que como ingeniero de software, es tu responsabilidad conocer las cosas comunes que hacen que tu aplicación sea una solución, aprendimos algo del sentido común a la hora de diseñar interfaces y experiencias asombrosas para tus aplicaciones, adicional a eso entender como aplicar estos principios te ayuda a diferenciarte de los demás desarrolladores, que lo único que saben es escribir código.

Labels

- Keep them short and concise
- Use top-aligned or right-aligned labels

Input Fields

- Avoid optional fields
- If not, separate them

Actions

- Separate primary actions

Validation Messages

- Keep them clear

Extendiendo Asp Identity User

Cada aplicación asp.net que creamos, usando la plantilla por defecto ya implementa por nosotros la autenticación y la autorización (en Framework, porque en Core tenemos que hacer un proceso adicional que vimos capítulos atrás), usando ASP net Identity, tenemos Login y registro, pero este formulario es muy básico y en la mayoría de aplicaciones del mundo real, nosotros necesitamos capturar una que otra información adicional, así que en esta sección vamos a extender Identity y añadir unos campos adicionales.

Construyendo una vista básica

Bien, lo primero que quiero hacer es reemplazar este home page, por el listado de todos los eventos, puesto que ya nuestra aplicación guarda eventos, así que vamos al home controller, en la acción index, necesitamos traer todos los eventos de la base de datos, traemos nuestro dbcontext que viene inyectado, lo iniciamos desde el constructor con el auto completado que nos crea también la propiedad.

Después nos vamos a cargar los eventos, pero acá viene un tema, si yo cargo solo una lista de eventos y yo quiero obtener el nombre del coach, no puedo, a pesar de que sea su hijo, para eso debo especificar que quiero que me cargue la data de su hijo y eso lo hacemos con el include, el método include recibe un string, el cual es el nombre de la propiedad de navegación a la cual voy a apuntar, pero los string son peligrosos, si mañana yo decido

cambiar el nombre de la propiedad (dbSet) donde quiera que se referencie me dará error y tendré que cambiarla, pero, cuando se trata de cadenas, visual studio no va a arrojar error, sin embargo en tiempo de ejecución cuando intente llamar ese campo por un nombre distinto va a reventar. Incluso puedes mandar esa aplicación a producción y no enterarte de nada, así que por eso siempre que podamos, debemos usar expresiones lambda, usamos una letra cualquiera, a mi me gusta siempre usar p de parámetro, pero las buenas practicas dicen que debes asignar una relacionada con la tabla, en nuestro caso seria c, entonces esto se lee c va a o es igual a, “=>” eso es una clave reservada que se inventaron los creadores que era así por lo que no me pregunten porque y aquí ponemos el nombre de la propiedad de navegación, si yo renombro esta propiedad o me marca un error o hace el renombrado en tiempo de compilación y no se me va con ese error, lo próximo que vamos a hacer con nuestros eventos es filtrarlo con un Where, para obtener solo los eventos futuros, así que p va a p.DateTime > DateTime.Now, el código me obliga a hacer scroll a la derecha por eso debo romperlo.

```
var upcomingEvents = _context.Events
    .Include(c => c.Coach)
    .Where(p => p.DateTime > DateTime.Now);

return View(upcomingEvents);
```

Ahora debemos poner esto que trajimos en la vista y finalmente debemos ir a la vista y recibir esto que estamos enviando, borramos todo el contenido, solo dejamos el viewbag del título, debo especificar el modelo, que en este caso va a ser un IEnumerable de Event, IEnumerable es una simple interfaz que es implementada por todas las listas y me permite iterar sobre un objeto, ahora debemos iterar sobre el modelo, para mostrar todos los eventos, podemos hacerlo con una tabla, a base de divs, o

usando ul, eso ya es opción de ustedes, yo casi siempre uso tablas, pero como quiero tener un diseño más versátil que me copy pasteé de por ahí antes de preparar la clase, voy a usar ul.

Escribimos @foreach tab tab para que auto complete el formato, le renombramos el nombre de la variable si deseamos, pero el objeto a iterar si es requerido que cambie, porque lo que vamos a iterar es el modelo.

Ahora nos enfocaremos en mostrar lo que queremos, después nos ponemos a dar chulerías, mostramos el DateTime completo, luego yo quiero ver el nombre del coach, pero no tenemos esa propiedad, así que vamos a mostrar el Username y después trabajamos esto, corremos la aplicación, verificamos que podemos ver la fecha y el coach.

```
@model IEnumerable<Event>

@{
    ViewData["Title"] = "Home Page";
}

<ul>
    @foreach (var item in Model)
    {
        <li>
            @item.DateTime - @item.Coach.UserName
        </li>
    }
</ul>
```

Extendiendo la clase ApplicationUser

Buscamos ApplicationUser, que si recuerdan cuando creamos Identity explicamos detalladamente que era cada cosa de allá, si no recuerda dele para atrás, aquí lo único que vamos a hacer es

añadir una propiedad, en el caso de nuestro aplicativo, nosotros tenemos la intención de saber como se llama el Coach, pero, no tenemos un campo en la tabla `AspNetUser` que me permita almacenar esto. Así que vamos a agregar la propiedad `Name` o `FullName` o lo puedes dividir en dos propiedades una para nombre y otra para apellido, como extendimos el modelo necesitamos crear una migración y actualizar la base de datos, porque siempre debemos ser ordenados.

Pero antes de eso, hay algo mas que necesitamos hacer, los tipos de datos `string` en `c#` por defecto son nulables, pero nosotros queremos que no lo sean porque no queremos que nos inserten registros sin nombres así que pongámosle que es requerido y el tipo de datos será `varcharmax` si no le ponemos una restricción con las `data annotations` para sobre escribir las convenciones por defecto de este campo, lo ponemos requerido y usamos el `stringlength`, ahora si hacemos la migración.

```
public class ApplicationUser : IdentityUser
{
    [Required]
    [StringLength(100)]
    public string Name { get; set; }
}
```

¿Si ya tenemos un usuario registrado (que lo tenemos) y creamos un campo nuevo que no permite nulos, que ustedes creen que va a pasar? La lógica nos dice que debe reventar y no permitirnos hacer la migración, pero EF es tan inteligente que como el valor nulo y el `string` vacío son similares, el nos llenará este campo con vacíos, cosa que no pasaba antes donde había que crearlos nulos, actualizar la base de datos, llenarlo con vacíos o data real y después volverlo no nulos, al momento de preparar esta tarea no

se si aplica con los demás tipos de datos, así que les dejo de tarea experimentar.

Siempre que hagan una migración revisen que es lo que se está implementando antes de actualizar la base de datos.

Nos vamos a la base de datos y vamos a la tablaAspNetUsers y podremos ver el nuevo campo, le ponemos data, ahora vamos a la vista de home y mostremos el nombre en lugar del Username, probamos y vemos los cambios.

```
<li>  
    @item.DateTime - @item.Coach.Name  
</li>
```

Extendiendo el formulario de registro

Bien, tenemos un pequeño problema, cuando un nuevo usuario se registra no hay forma de capturar el campo nombre que acabamos de crear así que necesitamos modificar el formulario de registro, busquemos el formulario de registro que está en el área identity, si vemos register deriva de un ViewModel llamado “RegisterModel”, si, Microsoft violando sus propias recomendaciones usando la palabra Model en lugar de ViewModel, pero bueh, lo buscamos, lo editamos, después volvemos a la vista para agregar el nombre copiamos cualquiera de los que están ahí y le ponemos para el nombre, ahora nosotros tenemos que hacer que cuando el usuario haga clic en guardar debemos enviar el nombre, por lo que debemos modificar el momento en que guarda. Luego probamos la aplicación y verificar que se guarde el dato en la base de datos.

Clase

```
public class InputModel
{
    [Required]
    [StringLength(100)]
    public string Name { get; set; }

    .....
}
```

Vista

```
<div class="form-group">
    <label asp-for="Input.Name"></label>
    <input asp-for="Input.Name" class="form-control" />
    <span asp-validation-for="Input.Name" class="text-danger"></span>
</div>
```

Controlador/CodeBehind

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    .....
    var user = new ApplicationUser
    {
        UserName = Input.Email,
        Email = Input.Email,
        Name = Input.Name
    };
    .....
}
```

Resumen

En esta clase aprendimos que la plantilla por defecto de Identity, no siempre va a tener todos los campos que necesitamos y para eso necesitamos extenderla, añadiendo propiedades adicionales, aunque la forma mas correcta de añadir algunas de estas como fecha de nacimiento, debe ser añadiendo Claims, que no se si lo lleguemos a tocar, pero sepan que existe, las extensiones que le hagamos al modelo base de User, deben ser cosas que realmente sean necesarias.

Creando un diseño más chulo y bonito con CSS

Esta sección es acerca de CSS, ojo, igual que en las secciones anteriores solo lo básico para que adquieran los conocimientos de como resolver una cosita breve de forma rápida, y para que estén familiarizados con estas, no me vayan a acribillar por no ser un experto en esta área de la cual seguro ustedes saben más que yo.

Maquetado

Supongan que yo quiero implementar algo como esto.



Para implementar este diseño, lo primero que necesito es un contenedor padre.



Este será el elemento que va a representar un evento completo, ósea, cada row (registro) que este en la base de datos, dentro de

este contenedor vamos a tener dos contenedores, uno para la fecha y uno para el detalle del evento como tal.



Ahora si miramos el contenedor de la fecha, podemos ver que este contenedor tiene dos hijos, uno para el mes, y otro para el día.



Así que lo primero que debemos hacer es crear o en maquetar esta estructura, usando HTML luego usamos CSS para ajustar los colores, el espaciado y el aspecto final que le queremos dar.

Zencoding

Bien, volvemos a VS, nos vamos al index, aquí podemos ver que estamos usando un li, para renderizar los elementos que nos interesan de cada uno de nuestros eventos.

Este va a ser nuestro contenedor padre, dentro de este debemos tener dos contenedores adicionales, uno para la fecha y otro para

el detalle, así que borremos el contenido de este li para trabajar en el en maquetado en primer lugar, por cierto, no maquetes con tablas, es muy mal visto en la industria que usted lo haga, aunque lo que usted vaya a hacer lo pueda resolver perfectamente con una tabla, no la use, mejor maquete a base de divs, es un tema ya de percepción, muchos desarrolladores incluso esos que no son radicales, cuando ven maquetación con tablas te miran con asco. Así que dígame ¡No! a las tablas.

Para esto yo quiero crear un div con una clase llamada date que es la que vamos a estilizar mediante código CSS, es decir, nos vamos a inventar una clase que puede llamarse chinchurria, y mediante código CSS, vamos a asignarle colores, estilos y diseño en general a cualquier elemento que tenga esa clase con ese nombre, para esto usamos una técnica que viene con web essential que fue una de las herramientas que dijimos que instaláramos en la primer clase, para eso escribimos: "div.date" presionamos tab tab y vemos como el nos crea un div que abre y cierra con una clase date, en este caso.

Ahora, podemos hacer las cosas un poco mas sofisticadas, ya nosotros sabemos la estructura que queremos entonces podemos decir que quiero un div con una clase date, y al lado de eso, yo quiero crear otro div con la clase details, junto con eso quiero especificar sus hijos de una vez, así que los pongo entre paréntesis, cada uno representa un grupo, nos vamos al grupo de date, y dentro de el quiero indicar que va a haber un div con la clase month, y al lado quiero tener un div con la clase day, de manera similar dentro del div detalle yo quiero tener un span con la clase coach, y al lado quiero tener un span con la clase tipo, me pongo al final de la línea y tab

`(div.date>div.month+div.day)+(div.details>span.coach+span.type)`

De esta forma es mucho más sencillo y rápido armar nuestro enmaquetado, siempre que sepamos de antemano la estructura que le vamos a poner a nuestra maqueta claro está.

Ahora dentro de estos divs ponemos los valores que vamos a mostrar, para el mes usamos `item.DateTime.ToString("MMM")` para especificar que deseamos mostrar solo el mes con formato de tres letras, si ponemos cuatro sería el mes completo, y para el día hacemos lo propio.

Pero hay un tema, si yo pongo `d` solamente el mes asume la fecha completa pero en formato corto y si yo pongo `dd` entonces el mes le va a agregar un cero al lado, yo prefiero que me le agregue el cero, pero algunas personas no, así que para eso el truco está en usar la tecla `escape` para que el intellisense no me auto complete el cerrado es como decirle al VS, no te quiero no te necesito, y después ponemos un espacio `item.DateTime.ToString("d ")`.

Esto es mucha mojiganga, pero es para que sepan que no es lo mismo `d` que `d` espacio y cuando vean un código de alguien más que lo tenga así, sepan el porqué. Hacemos lo propio con `coach` y con el tipo.

En caso de que te preguntes porque usar un `span` en lugar de un `div` o viceversa, es que, por temas de preferencias de alguna gente dura, los `div` son usados para cosas donde voy a dar estilos chulos,

mientras que los span cuando solo quiero mostrar texto sin diseño asombroso, pero perfectamente podrías usar otro div.

Si corremos ahora veremos un error, pues nosotros estamos incluyendo en nuestro query a los coach, pero no le indicamos en ningún momento que vamos a incluir el tipo, así que cuando intento encontrar el nombre del tipo de evento, revienta. Por lo que nos vamos al controlador e incluimos el tipo con Include.

Corremos y aunque no se ve como queremos, ya sabemos que el enmarcado está ahí.

Posicionamiento absoluto y relativo

Ok, nuestra estructura esta lista, pero no luce en nada a la imagen que planteamos al principio, para implementar algo como eso, primero necesitamos entender lo que es posición relativa y absoluta en CSS, aquí hay un truco.

Un elemento con **posición relativa** nos permite posicionar de manera **absoluta** sus hijos.

```
position: relative;
```



Por ejemplo, en el contenedor azul como este, si seteamos la posición de este contenedor a relativo, podemos precisar la posición para cualquiera de sus hijos, así que podríamos poner un elemento dentro del contenedor, en una posición exacta, para que ocurra de esta forma, nosotros tendríamos que especificar como absoluta la posición de sus hijos y luego usar el top y el left, atributos, para ponerlo exactamente donde queremos que esté, en este caso esta 10 pixeles de distancia hacia arriba y 100 de distancia hacia los lados de su padre.



Ojo con los márgenes XD

En nuestro caso, témenos que hacer algo similar a lo que vemos en pantalla.



Entonces para esto nos vamos a Visual Studio, lo primero que haremos es poner al padre que es el li, como relativo y luego el hijo que, en detalles, lo quiero poner como absoluto, para que de esa forma quede al lado de la fecha.

Pero antes de iniciar con el CSS, tenemos que hacer algo, es que el ul, nosotros tenemos que darle una clase, para poder identificar a cualquiera de sus hijos, en nuestro CSS, en este particular le pondremos events.

Abrimos nuestro archivo de site.css, nos vamos al final y abrimos una sección para nuestros estilos específicos, la primera parte de nuestro site.css es para todo aquello que es genérico a nuestro sistema y luego para lo que es específico.

Entonces ponemos. `events > li {}` esta instrucción le indica que, este CSS va a modificar el elemento que esta inmediatamente al lado de la clase mencionada (events) y que es del tipo indicado, en este caso li, entonces acá le ponemos la posición a relativa,

también quiero agregar algo de margen, para que cada evento no este tan pegado del otro.

```
.events > li {  
    position: relative;  
    margin-bottom: 30px;  
}  
  
.events > li .details {  
    position: absolute;  
    top: 0;  
    left: 70px;  
}
```

Nos vamos al navegador probamos y vemos que todo se ve mejor, ahora nos toca eliminar las bolitas esas feas que salen ahí, eso se hace seteando el formato de lista en none.

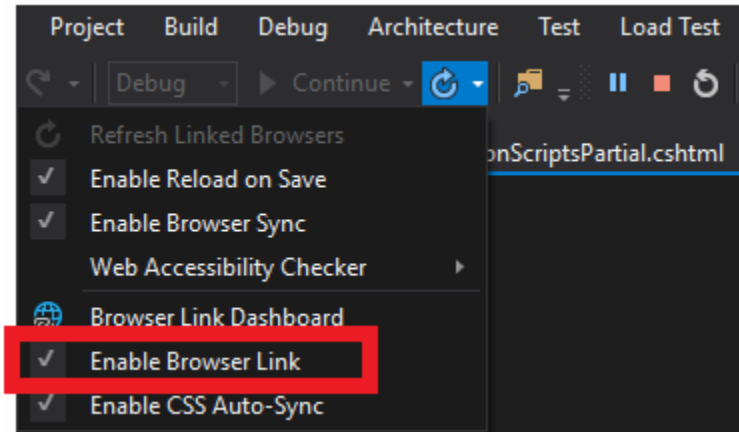
```
.events {  
    list-style: none;  
}
```

Trabajando con atributos CSS.

Ahora nos toca trabajar con la construcción del icono de calendario, para eso nosotros necesitamos trabajar de manera paralela entre Visual Studio y el navegador para ver los cambios de una vez, para esto vamos VS y tenemos luego que presionar la tecla Windows y la tecla derecha, esto pone la pantalla de VS en modo pantalla dividida. Nos vamos al navegador y le damos Windows + Izquierda y se pone una al lado de la otra, de esa forma podemos ir viendo ambos para trabajarlos, una al lado de la otra sin necesidad de tener que ponerte a hacerlo de forma manual.

Hay una opción llamada browser link lo que me permite esta herramienta es que mientras voy trabajando cambios en el html del proyecto estos se vayan refrescando automáticamente sin tener

que dar f5, pero primero debemos activarla, se supone que esta funcionalidad viene nativa en core y lo único que tenemos que hacer es habilitarla.

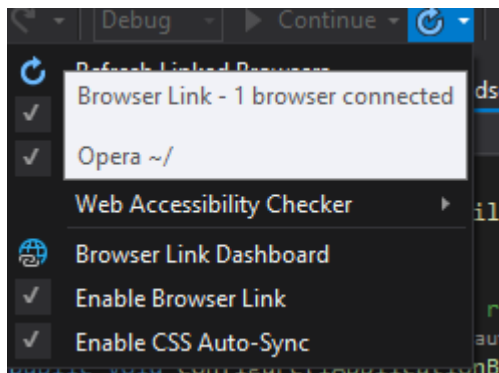


Pero, por alguna extraña razón a mi y a un montón de gente en internet no le funciona, así que según lo que he podido encontrar lo que tenemos que hacer es instalar un paquete de nugget, los paquetes de nugget son repositorios donde la gente va subiendo librerías que le dan funcionalidades extra a nuestra aplicación, extendiendo el framework con esas pequeñas piezas de funcionalidad que no siempre vienen por defecto en el Core, la que vamos a instalar en cuestión es: `Microsoft.VisualStudio.Web.BrowserLink` y después de instalarlo vamos al `Startup.cs` y en el modo development habilitamos el browserlink.

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
    app.UseBrowserLink();
}
```

Si miramos el tooltip podemos ver que dice que tenemos un navegador conectado, si no ven esto, es porque no están corriendo en modo debug, asegúrense de que todas las opciones estén seleccionadas, con esto, siempre que haga un cambio y guarde mi

archivo y situé el foco en el navegador, el se va a refrescar el cambio de manera automática, sin tener que presionar f5.



Bien, ahora vamos a trabajar en la clase date, que esta en nuestro elemento li de nuestro padre events, hacemos su color de fondo rojo o azul, y el color del texto blanco, y la alineación del texto al centro, guardamos corremos y podemos ver un error, porque el estilo se va a aplicar a la clase, pero esta no tiene delimitación, y al contenedor le ponemos 60 pixeles, de longitud.

Hacemos las modificaciones que deseemos para que las cosas nos vayan quedando como queremos y como no hay mucho que explicar aquí, solo mostrare el código, final, porque no vamos a introducir o aprender nada que no hayamos visto, lo único destacable es que necesitamos un espaciado entre los dos span que se ven uno al lado del otro, esto se debe a que por defecto los span se alinean en una línea (in line alignment), por eso tenemos que indicarle el formato de display de los elementos hijos dentro del detalle a block, para que muestre en forma de bloques (pilas, ósea, uno arriba de otro).

Vista

```
<ul class="events">
  @foreach (var item in Model)
  {
    <li>
      <div class="date">
        <div class="month">
          @item.DateTime.ToString("MMM")
        </div>
        <div class="day">
          @item.DateTime.ToString("d ")
        </div>
      </div>
      <div class="details">
        <span class="coach">
          @item.Coach.Name
        </span>
        <span class="type">
          @item.Type.Name
        </span>
      </div>
    </li>
  }
</ul>
```

CSS

```
.events {
  list-style: none;
}

.events > li {
  position: relative;
  margin-bottom: 30px;
}

.events > li .date {
  background: blue;
  color: white;
  text-align: center;
  width: 60px;
  border-radius: 10px;
}

.events > li .date .month {
  text-transform: uppercase;
  font-size: 14px;
  font-weight: bold;
  padding: 2px 6px;
}

.events > li .date .day {
  background: #f7f7f7;
  color: #333;
  font-size: 20px;
  padding: 6px 12px;
}
```

```
    }  
  
    .events > li .details {  
        position: absolute;  
        top: 0;  
        left: 70px;  
    }  
  
    .events > li .details .coach {  
        font-weight: bold;  
        display: block;  
    }  
  
    .events > li .details .type {  
        font-size: 15px;  
        display: block;  
    }  
}
```

Resumen

En este módulo, vimos cómo usar CSS, para crear un diseño preciso y bonito, si quieres ser un verdadero desarrollador full stack, debes “dominar” estos conceptos.

Aprendimos de posicionamiento relativo y absoluto, que son usados para construir apariencias precisas, también aprendimos algo básico de zencoding, que te permite de manera rápida hacer en maquetado HTML.

Implementando los casos de uso restantes

En esta sección vamos a terminar de implementar los casos de uso de principio a fin, porque ya estamos “jartos” de aprender cosas y aun no tenemos un sistema funcional, pues ya llevamos más de 150 páginas y apenas tenemos una sola pantalla. Veremos dos formas de extender nuestro modelo de dominio y luego explicaré porque prefiero una sobre la otra (spoiler, la que dicen los expertos que además de programar también saben educar), también vamos a crear un API que vamos a llamar desde AJAX.

Diseño pobre :’(

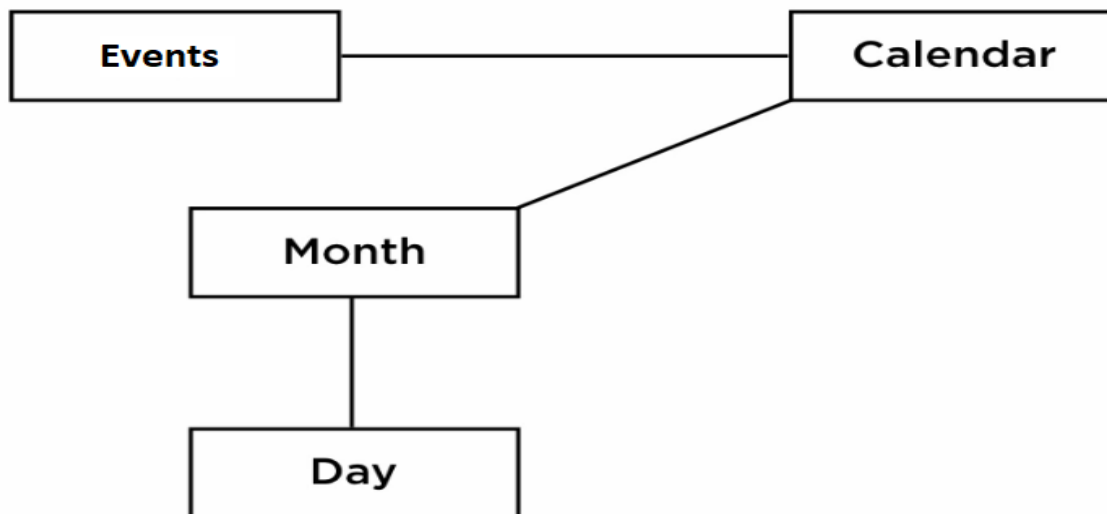
Bien, nosotros hasta el momento ya implementamos de nuestros casos de uso principales, el mas importante que era el de Crear un evento, de igual forma ya solucionamos uno de los casos de uso secundarios o mas bien de soporte, que es el de mostrar la lista de los eventos futuros. Ahora nos toca implementar el caso de uso ***Añadir Evento al Calendario***, para eso vamos a hablar un poco de modelado de clases.

Uno de los errores mas comunes en desarrolladores no experimentados cometen es que se confían demasiado de las palabras que están escritas en el documento de requerimientos, yo en lo personal a pesar de que no puedo decir que soy un experto en diversas tecnologías, me cuesta mucho aprender algo nuevo o caerle atrás a estos muchachos que se aprenden una tecnología nueva cada dos semanas, puedo jactarme de decir que he trabajado en una cantidad considerable de proyectos diversos

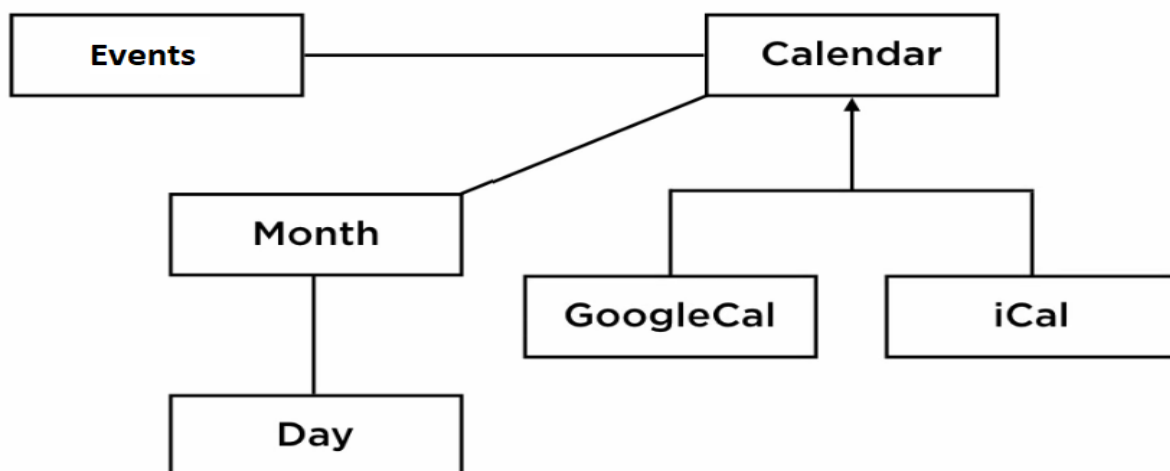
con diferentes tipos de personas, quizá no de montos elevados que es uno de los principales problemas que tengo (el no saber vender mi trabajo), pero ese no es el tema, tu vez gente que sabe mucho y que domina muchas tecnologías, que son unos monstros en lo que se refiere a tirar código, que saben mucho de devops etc, etc, pero a la hora que tu le preguntas “ok, ¿en cuantos proyectos haz trabajado?” se quedan pensando, siempre tienen un cuento y nunca han hecho nada productivo con el código, saben mucho, es verdad, son fuente de inspiración algunas veces, pero nunca han hecho nada, solo aprenden y aprenden, coincidentalmente esa misma gente siempre te dicen: **“lo que el cliente pide eso es lo que hay que darle”** o **“si el cliente lo quiere así, así hay que dárselo aunque sea estúpido”** o **“eso no estaba en el documento de requerimientos”** o **“ustedes tienen que aprender a levantar bien los requerimientos para evitar que uno trabaje en valde”** etc, etc, ok, si el cliente es el que quiere su disparate y hay que dárselo, pero no siempre el cliente esta claro de lo que desea, por eso te corresponde a ti orientarlo y ofrecer siempre que se pueda el plus, sin entrar en las confrontaciones, porque si el lo quiere rojo lumínico es tu deber explicarle porque no debe ser rojo lumínico, pero si el insiste dáselo rojo, malo es que el cliente se entere por otra vía que el rojo lumínico hace daño y nadie le va a visitar la página y no por su desarrollador estrella a quien le esta dando la paca para que haga un buen trabajo, sea su consultor en temas por los que no te va a pagar y de paso seas su confidente si un día pelea con la esposa.

Volviendo al tema, tenemos que trabajar en añadir un evento al calendario y un desarrollador novato va a pensar, oh, tenemos dos clases aquí. Event y Calendar, entonces viene la pregunta, ¿cuáles van a ser los atributos y comportamientos de la clase

calendario? ¿Vamos a tener clases como Mes y Dia para luego asociarla a nuestro calendario?



En verdad podemos, porque después de todo, ese es el modelo del mundo real, en un calendario tiene meses y días, por lo cual ellos “deberían” ser parte de nuestro modelo de dominio, además alguien puede argumentar que en el futuro queremos integrar nuestra app con Google Calendar u otros calendarios, por lo tanto, “deberíamos” tener nuestra clase calendario aquí, quizá podríamos heredar de otras clases, y que estas sean las que alimenten.



Pero, ¿esto realmente esta solucionando el problema que tenemos en la mano o por el contrario lo está agravando más de la cuenta?

No realmente, no resuelve nuestro problema, recordemos, el problema que estamos intentando resolver es: **“Necesitamos mantener un registro de los eventos a los que el usuario desea asistir.”**

El problema no pide integración con Google calendar, no dice que necesitamos registrar los días y los meses con todos los días festivos tampoco, es únicamente llevar el histórico de los eventos a los que el usuario desea asistir, y ese es el punto principal al que quiero que le prestes mucha atención.



Do not model the universe!

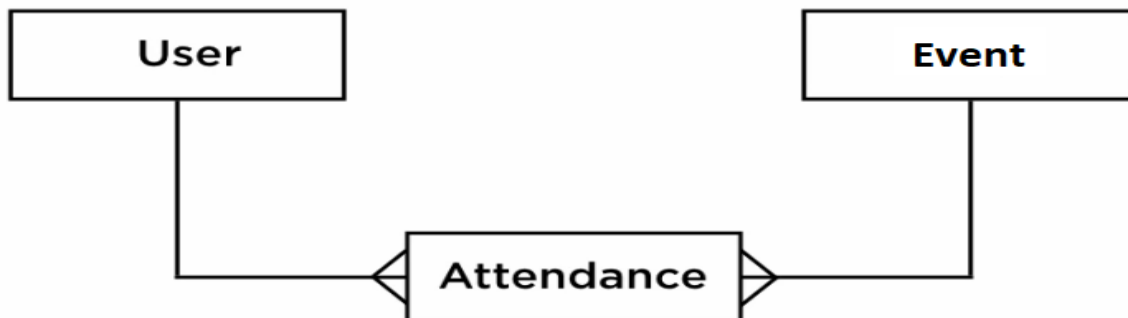
Cuando construimos software, no intentes modelar el universo, solo crea esos pedazos de modelo que te ayuden a solucionar el problema en cuestión que tienes en las manos.

En una ocasión me tocó conversar con el encargado de un proyecto en un hospital local, el tipo me habló del montón de tablas que necesitaba para su hospital, mas de 200 tablas según él, adivinen ¿de qué era el sistema?. Una simple pantalla de programación de citas, lo cual MersyRD resuelve con 6 u 8 tablas,

el pana tenía la visión de que debía controlarse cada aspecto de la cita, modelando todo el universo involucrado en la cita, imagínense que se le hubiera hecho caso, recuerdo por ejemplo un caso que Mosh cita, respecto a una empresa que tenía un caso similar, la cual gastó 500 mil dólares y no se realizó ni siquiera la primer beta del proyecto porque el gerente quería un control absoluto de todo, para que no crean que solo en Latino América piensan de esa forma tan loca, bien, quiero que se nos quede grabado, no trates de modelar el universo, no trates de resolver hoy los problemas del futuro que quizá nunca existan, resuelve hoy los problemas del ahora, ojo, con criterio sensato, porque tampoco te pases de verga usando int16 como identificador de una tabla de transacciones o sbyte para el Id de la tabla usuarios.

Así que vamos a separarnos de las palabras específicas que están en el documento de requerimientos y como nosotros somos super analistas vamos a pensar de manera lógica y sensata, vamos a enfocarnos en el concepto detrás del requerimiento, un **usuario puede agregar un evento a su calendario** y en una pagina separada **ver los eventos a los que el desea asistir**, esencialmente lo que necesitamos es una forma de **mantener el registro de los eventos que el usuario desea participar**, así que una simple asociación entre el usuario y el evento es mas que suficiente para resolver nuestro problema, un usuario puede asistir a varios eventos y un evento puede tener varios usuarios registrados como posibles asistentes, como puedes imaginar tenemos una relación mucho a mucho en nuestros objetos de modelos, y no necesitamos ninguna clase llamada calendario o mes o día, etc, a los usuarios lo vamos a llamar asistentes, participantes o público.

Ahora cuando nosotros tomemos este modelo orientado a objetos y lo implementemos en nuestra base de datos relacional necesitaremos una estructura como esta:



Necesitamos una tabla intermedia para nuestra relación mucho a mucho, EF puede crear esta tabla por nosotros, como parte de una migración, y manejarlo por detrás, pero en este caso, yo deseo agregar esta tabla a nuestro modelo, así que tendré una clase llamada *Attendance*, en nuestro modelo de dominio, ¿Por qué? Puesto que en algunas partes yo quiero tener un objeto de manera directa un objeto que represente esa tabla a la cual tirarle de forma inmediata sin tener que hacer queries complicados, por ejemplo cuando quiero verificar si un usuario ya marcó que desea asistir a un evento, ósea, verificar si ya existe ese registro en nuestro calendario, así que cuando añadimos esta clase a nuestro modelo, podemos de manera simple usar LINQ, veremos esto mas adelante cuando estemos construyendo el API, ahora por lo pronto vamos a seguir con nuestro modelo.

Extendiendo el modelo de Dominio

Bien, dicho todo lo anterior, vamos a VS, en nuestra carpeta de modelos, clic derecho, añadir nueva clase así que creamos una

nueva clase llamada Attendance, lo primero que necesitamos son las propiedades de navegación que hacen referencia a nuestro usuario y nuestro evento, de manera opcional por un tema más de optimización que otra cosa, también vamos a añadir las propiedades de las llaves foráneas, expliqué en previos temas, la razón para hacer esto a pesar de que algunos dicen que es mala practica y redundancia, recuerden que el simplemente añadir estas propiedades me ahorra el tener que cargar el objeto evento completo al momento que deseamos crear un nuevo evento en la base de datos, así que un poco de romper las reglas a cambio de un algo de optimización, creo que es valido de cuando en vez.

Bien, para respetar las normas de Identity, necesitamos un Id, la cual en este caso puede ser la representación de la combinación de dos elementos y no un Id único, por lo que le agregamos la anotación Key a AttendeeId y a EventId, cuando usamos dataannotationa para llaves compuestas, necesitamos especificar el orden de las columnas, como el evento es para nosotros mas relevante y menos repetible dentro de esta tabla que el usuario, le añadimos Column(order=1) a la columna evento y order=2 a la otra.

```
public class Attendance
{
    [Key, Column(Order = 1)]
    public int EventId { get; set; }
    public Event Event { get; set; }

    [Key, Column(Order = 2)]
    public int AttendeeId { get; set; }
    public ApplicationUser Attendee { get; set; }
}
```

Ahora añadimos el DbSet a nuestro contexto porque yo deseo de forma directa hacer queries usando LINQ de forma directa sobre el objeto.

```
public DbSet<Attendance> Attendances { get; set; }
```

Y como cambiamos el dominio debemos agregar una migración, la revisamos y posterior a eso corremos la migración para que actualice nuestra base de datos.

Bueno, al intentar correr vamos a tener un error, este error nos indica que estamos intentando hacer una llave primaria compuesta con data annotations, pero que eso ya no es permitido, que si deseo hacer esto debo usar Fluent API, si estuviéramos trabajando en Framework, todo estaría genial, pero hay cosas que en NetCore se complica un poco y esta es una de ellas, así que nos vemos obligados a tocar un tema que originalmente estaba dos capítulos más adelante.

Sobre escribiendo convenciones usando Fluent-API

Ya nosotros sabemos que por defecto (convención) ASP net configura valores predeterminados para ciertas propiedades, y también sabemos que para cambiar esas convenciones hay dos formas, una es a base de data annotations y la otra es a base de Fluent API, vamos a ir a nuestro datacontext, para usar Fluent API necesitamos sobre escribir el método **OnModelCreating**, que es el evento que se construye cuando se crean los modelos, para sorpresa nuestra en netCore este método es tan usado que para los desarrolladores novatos ya le pusieron una especie de shortcut para que sepamos exactamente donde debemos indicar las sobre

escrituras del modelo antes de su construcción esto en Framework no estaba.

En este punto, podemos usar el builder que se nos pasa aquí, para proveer configuraciones adicionales, que sobre escriban las convenciones por defecto, por lo que nos corresponde hacer las configuraciones necesarias, tales como indicar que campos son requeridos, cual es la llave primaria en caso de que vayamos a usar una compuesta definirla y debemos identificarle el tipo de relación que tiene con el campo que estamos indicando que es requerido, podemos decirlo en otra configuración, o podemos hacerlo ahí mismo, ponerle el tipo de relación que hay entre evento y la tabla en cuestión, pero eso era en la vieja escuela, porque la moderna privando en vaina complica un poco las cosas, lo primero es que si hay que separarlo, adjunto el código, ah, un dato que en ninguna parte nadie te va a decir, es que al momento de crear llaves compuestas de manera explícita las propiedades de navegación son requeridas, por ende no tienes necesidad de indicarlo, pero si lo indicas EF te va a dar un error que no tiene respuesta en internet.

```
// old and simple and beautifull way of do this on ef6
//builder.Entity<Attendance>.HasRequired(a => a.Event).WithMany(g => g.Attendances);

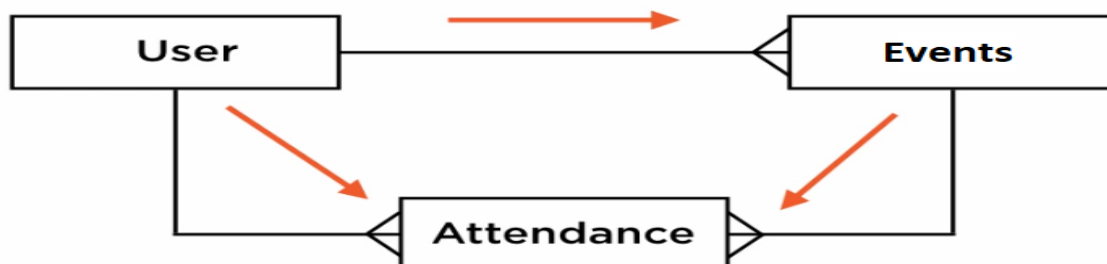
//Defining the composed primary Key
builder.Entity<Attendance>().HasKey(p => new { p.AttendeeId, p.EventId });
//Relation
builder.Entity<Attendance>()
    .HasOne(a => a.Event)
    .WithMany(e => e.Attendances)
    .HasForeignKey(a => a.EventId);
```

Nosotros podemos agregar esa propiedad de navegación, la cual no tenemos (`e.Attendances`), en caso de que en el futuro yo desee tirar queries a la inversa pero de manera temporal no lo necesito, por lo que solo le establezco `WithMany` vacío, porque no quiero llenar

el dominio con propiedades innecesarias que no vamos a usar por ahora, so, ya configuramos la relación. Hacemos lo propio con el ApplicationUser.

Debemos asegurarnos de llamar el modelo base al final de nuestras configuraciones, nosotros siempre que sobre escribamos un método, deberíamos dejar que el método funcione como va a funcionar después que le hayamos configurado ciertas cosas en específico, o viceversa. Ahora necesitamos eliminar la migración que ejecutamos puesto que no nos sirve, proveemos una nueva migración, actualizamos la base de datos y todo debe salir bien, o quizá no...

Es muy probable que veamos un error, muy común en Framework, introducir un foreign key en nuestra tabla Attendance, puede causar ciclos o borrado en cascada, por ejemplo, si yo borro un usuario al eliminarlo (si la base de datos esta configurada para eso) va a eliminar todos los eventos creados por el así como los eventos a los que el desea asistir, eso es borrado en cascada, pero supón que al mismo tiempo cuando un evento es eliminado el va a intentar eliminar todos los registros en los que un usuario haya marcado que va a asistir pero puede que ese evento ya haya sido eliminado por el primer borrado, por lo que a SQL no le gusta eso, esto es llamado borrado en cascada cíclico, el cual no es posible manejarlo por SQL hasta la fecha.



Lo que nos corresponde es deshabilitar el borrado en cascada el cual es de hecho una brecha de seguridad enorme, lo podemos deshabilitar en una de las dos relaciones o en ambas, también podemos hacerlo en el sistema entero, antes en Framework era fácil eliminar el borrado en cascada, en la actualidad es un poco mas complejo, porque la idea es forzar al desarrollador a que sepa que es lo que esta haciendo, sin embargo en mi tiempo de vago y basado en unos tutoriales para hacer cosas de forma dinámica sobre el modelo, yo les voy a compartir un pedacito de código que les va a facilitar la vida, pero para eso, falta. Mientras vamos a usar la siguiente línea de código.

y finalmente ahí mismo, en caso de que deseemos eliminar el borrado en cascada, después hablamos de eso, usamos:

```
.onDelete(DeleteBehavior.Restrict); //old way withmany().WillCascadeOnDelete(false);
```

El código final debería quedar de esta forma:

```
protected override void OnModelCreating(ModelBuilder builder)
{
    builder.Entity<Attendance>().HasKey(p => new { p.AttendeeId, p.EventId });

    builder.Entity<Attendance>()
        .HasOne(a => a.Event)
        .WithMany()
        .HasForeignKey(a => a.EventId); //.onDelete(DeleteBehavior.Restrict);

    builder.Entity<Attendance>()
        .HasOne(a => a.Attendee)
        .WithMany()
        .HasForeignKey(a => a.AttendeeId); //.onDelete(DeleteBehavior.Restrict);

    //Disabling Cascade delete on all entities without care if I use fluent API or not
    var cascadeFKs = builder.Model.GetEntityTypes()
        .SelectMany(t => t.GetForeignKeys())
        .Where(fk => !fk.IsOwnership && fk.DeleteBehavior ==
            DeleteBehavior.Cascade);

    foreach (var fk in cascadeFKs)
        fk.DeleteBehavior = DeleteBehavior.Restrict;

    base.OnModelCreating(builder);
}
```

Diseñando y creando un API

Bien, ahora nos toca trabajar en el UI, nosotros queremos ponerle una especie de botón al lado de cada evento que nos pregunte si deseamos ir al mismo y en caso de que ya hayamos marcado que vamos a ir, entonces debe cambiar el texto, sin embargo, nosotros tenemos un problema, cada botón va a ejecutar una acción, esa acción debe ir al controlador y hacer algo, entonces, ¿Qué usted cree que va a pasar con la página? Bueno, constantemente se va a ver la llamada a al server y la pagina entera debe recargar, por eso nosotros necesitamos hacer llamadas AJAX, que son llamadas al server que no necesitan recargar la página, así que en el servidor vamos a crear un API RESTful, que podamos llamar usando JQuery Ajax, a pesar de que jquery esta desfasado y dique nadie lo usa, debido a que la mentalidad en la web cambió respecto a como se manipulan las cosas, en la calle vas a encontrar muchísimos proyectos que lo usan, y yo soy de los que pienso, si algo no esta muerto y resuelve y sus creadores aun no lo han matado y el producto es bueno, siga usándolo.

Bueno, nosotros vamos a extender nuestra app y le añadiremos un API, hay dos formas principales de trabajar con apis, Post y Put, nosotros vamos a utilizar post porque es la forma más común además de ser la que ya conocemos, entonces nosotros vamos a exponer un api publica donde vamos a incluir data en el cuerpo de la petición, ¿Qué debería incluir? Definitivamente debo incluir el EventId, pero, ¿Qué creen del UserId? Esto no lo debo añadir, porque seria una estupidez y una falla grave de seguridad, de hecho, esta es una de las razones principales para usar Dtos

donde no se expongan los datos sensibles, o ViewModels por cada representación que deseamos de nuestro modelo.

Para crear un api clic derecho en Controller y vamos a crear una carpeta que llamaremos "API", posteriormente cuando ya nuestra aplicación este funcional nosotros vamos a separar los modelos en un proyecto, los ViewModels en otro, las apis en un proyecto distinto etc, pero por lo pronto vamos a tener todo por aquí.

Luego clic derecho, agregar clase, la voy a llamar Attendances, la voy a poner a heredar de Controller, importamos el using que nos recomienda y dentro vamos a crear una acción llamada Attend que recibe un parámetro de tipo int llamada eventId.

Por razones de seguridad no vamos a enviar el usuario, sino que vamos a tomar el usuario logueado en un api que no este dentro de este proyecto debemos enviar con cada petición dentro del cuerpo de la misma, las credenciales y el usuario que se va a loguear, pero esto mediante un token, sin embargo, para eso falta mucho, seguimos.

Ahora para agregar un asistente a un evento necesitamos el DbContext, nosotros podemos usar un truco que es usar la primera letra de cada palabra que compone una clase por ejemplo adbc me va a auto completar ApplicationDbContext, (si nosotros la hubiéramos llamado de esa forma), inicializamos el constructor, y procedemos a guardar, hacemos este método como accesible solo para usuarios logueados a los fines de que siempre este método nos devuelva el usuario logueado y no reviente porque no puede instanciar ese objeto, bien, vamos a implementarlo de manera

básica por ahora, después le agregamos más lógica de que si no esta duplicado etc, solo quiero hacer una pequeña pieza de funcionalidad, verificar que esta nítido y después continuo metiendo mano.

Esta acción solo puede ser llamada mediante un post, por lo que debemos decorar con ese atributo a la misma.

```
private readonly DataContext _context;

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

[Authorize]
[HttpPost]
public IActionResult Attend(int eventId)
{
    var attendance = new Attendance
    {
        EventId = eventId,
        AttendeeId = User.FindFirstValue(ClaimTypes.NameIdentifier)
    };
    _context.Attendances.Add(attendance);
    _context.SaveChanges();
    return Ok();
}
```

Esta es la estructura que tenemos, la cual si probamos no va a funcionar, por una sencilla razón, al momento de poner un parámetro de la forma en que lo tenemos, le estamos diciendo que para entrar a esta acción deben hacerme una petición que incluya el parámetro, ejemplo attend/5, sin embargo nosotros deseamos hacer la petición y no enviar en la url información porque una regla no escrita de cuando hacemos llamadas ajax, es la de no mandar valores en la Url, sino solo en el cuerpo de la petición, por lo cual debemos indicarle frombody, para que lo tome del cuerpo de la petición los valores que necesite.

```
public IActionResult Attend([FromBody] int eventId)
```

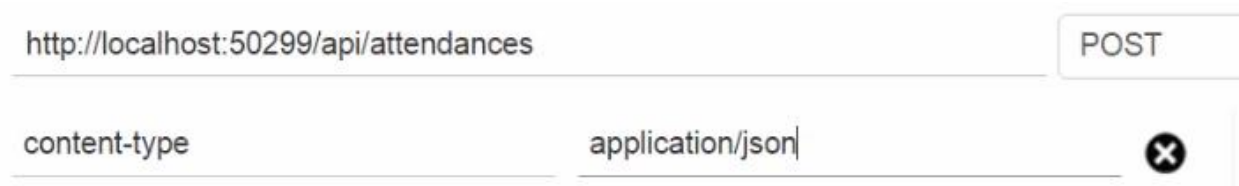
Ahora para probar nuestro api, corremos la aplicación y nos vamos al super héroe de los programadores.

Metiendo mano con PostMan

Nosotros para probar un api podemos usar la app oficial de postman o un plugin de chrome, para encontrar el plugin en Google ponemos “Tabbed Postman - REST Client”, en mi caso usaré este para evitar unos temas de permisos que no quiero explicar en estos momentos.

En la url ponemos la dirección de nuestra api, cambiamos el get a post en donde dice tipo de petición que deseamos hacer, luego donde dice el modo de petición (dentro de la opción body) hacemos clic en raw, para indicar el cuerpo de nuestra petición, aquí ponemos el EventId, que como tenemos en la base de datos un par de ellos, podemos usar cualquiera siempre que exista y finalmente hacemos clic en enviar, si tenemos un HTML en el resultado es que algo anda mal con el api a la cual le estemos tirando, eso no debe pasar en aplicaciones profesionales, pero por ahora simplemente vamos a buscar, en algún lugar por ahí podremos ver que nos dice que nos logueemos, lo que pasa es que en algún lado de nuestra app en una de esas configuraciones raras que hace entity, cuando no tienes permiso a un lugar el lo que hace es que te redirecciona al Login, y esa es la respuesta que esta enviando acá, eso no es correcto, nosotros sencillamente deberíamos darle un BadRequest, pero habría que ponerse a modificar Identity y no tenemos tiempo para eso.

Nada, ahora vamos a iniciar sesión, después veremos par de cositas, pero por lo pronto resolvamos iniciando sesión y vamos a ver que lo que con que lo que, volvemos a postman y hacemos clic en enviar cuando nos da el siguiente error, significa que no pusimos un header, cada petición debe contener ciertos datos, dentro de los que se encuentra como requerido la cabecera en donde vamos a especificar el tipo de petición que vamos a hacer.



The image shows a Postman interface for a POST request. The URL bar contains 'http://localhost:50299/api/attendances'. To the right of the URL bar is a button labeled 'POST'. Below the URL bar, there is a header section with 'content-type' as the key and 'application/json' as the value. A close button (an 'x' in a circle) is located to the right of the header section.

Y si ahora nos devuelve estatus 200 es que todo salió bien.

Si hago clic en enviar de nuevo tenemos una excepción, porque nosotros tenemos configurada nuestra base de datos para que tenga una llave primaria compuesta, donde no pueden repetirse el usuario y el evento.

Ahora vamos a prevenir que nos inserten información duplicada.

```
var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);

var exists = _context.Attendances.Any(a => a.AttendeeId == userId &&
                                           a.EventId == eventId);

if (exists)
    return BadRequest("The attendance already exists");
```

Probemos, como ya tenemos nuestro api funcionando, vamos a crear un botón en la lista de eventos para indicar que deseo asistir.

Implementando el botón

En el contenedor de detalle, yo voy a poner un botón después del tipo de evento, le damos las clases **btn btn-default** para darle un background blanco, y la etiqueta va a decir ¿Going?, corremos el app y verificamos que todo esté jevi, el botón es un poco grande así que usamos una clase de Bootstrap para que el botón sea más pequeño, que es **btn-sm**, y si probamos nos damos cuenta de que no funciona, ¿saben por qué? Pues porque en nuestro site.css nosotros estamos sobre escribiendo las clases de Bootstrap y para todos los casos le indicamos que el Font-size y el padding son un poco grandes, así que tenemos que tener cuidado cuando sobre escribimos elementos que no nos pertenecen, de manera temporal vamos a deshabilitar el padding y el fontsize en nuestro site.css después arreglamos las cosas.

Tenemos otro problema, los botones no están alienados, no se ve mucho, pero hay un pequeño desalinee por ahí, vamos a alinear a la derecha, le ponemos la clase pull-right como empuja a la derecha, probamos y vemos que se alinea perfecto. Bien, ahora vamos a agregarle funcionalidad, para ello vamos a escribir algo de JavaScript.

```
<button class="btn btn-default btn-sm pull-right " >  
    Going?  
</button>
```


Iniciamos con la sección de scripts para que esto se ejecute después de la sección de los scripts del `_layout`, luego creamos la etiqueta de scripts.

JavaScript 99

Los eventos en la web, a diferencia de los lenguajes de programación tradicionales donde yo de antemano debo indicarle que eventos posee y desde un principio el compilador conoce todos los eventos posibles que tiene un objeto y que si nosotros derivamos de un objeto base, todos sus hijos heredan sus eventos, como por ejemplo, si nosotros declaramos un botón, todo botón tiene un evento click por defecto, entre otros eventos, un textbox, siempre va a tener un evento textchange, focus, etc, en la web las cosas no son así (inserte meme de tachala) en la web tú debes subscribirle (añadirle, adjuntarle) a los elementos (objetos) los eventos y después indicar que se va a ejecutar o disparar cuando ocurra ese evento, los diversos Frameworks por defecto nos atan diversos eventos por defecto, por eso se nos hace un poco complicado entender esta parte al inicio, pero si usted viniera de la vieja escuela (como afortunadamente no vengo yo, gracias al señor, hubiera entendido de una todo el tema de subscribir eventos porque recuerden que una vez algo llega a donde el usuario no es mas que un documento de texto con vitaminas.

Entonces, yo tengo que subscribir un evento cuando ocurra el clic del botón por lo que lo primero que necesitamos es un identificador para nuestro botón, hay dos formas de referenciar a un objeto mediante JavaScript, una es por clase y la otra por id, nosotros ya hemos referenciado por clase antes cuando estábamos trabajando

con css donde buscábamos reemplazarle o ponerle atributos a elementos que tengan una clase creada por nosotros, sin embargo en nuestro caso, como es un elemento que se carga de forma dinámica, no podemos usar el id, así que nos vamos a referenciar a este botón por clase, voy a usar la nomenclatura js- para indicar que esta clase no es de presentación sino de JavaScript y la voy a llamar toggle-attendance porque si ya estoy asistiendo debo marcar que no deseo asistir pero si no marque que deseo asistir entonces debo marcarme como asistente, así que nada, copiamos esa clase y la usamos como un selector, nos subscribimos al evento click y dentro deseamos hacer una petición a nuestra api, para eso usamos el post de jquery, el primer elemento es la url, como estamos en el mismo proyecto que deseamos llamar podemos ponerlo así, y el segundo parámetro es el cuerpo, en nuestro cuerpo lo único que mandamos es el EventId, pero no tenemos forma de mandarlo, a no ser que a nuestro botón nosotros le añadamos un atributo dinámico o usamos uno que ya existe para nosotros que es el data-attribute, en este caso vamos a crear uno con el nombre **data-event-id** y renderizamos aquí el EventId que venga del ítem. Valiéndonos de la “e” dentro de la función, significa que este va a representar el evento (no el de nuestro modelo, sino el que se ejecuta cuando se hace clic en un botón) y de este evento nosotros podemos obtener su data attribute, o la fuente del evento que está en nuestro botón.

Bien, eso va a nuestra api y retorna una promesa (que es un concepto muy usado en crypto monedas), con los escenarios ya sea de error o de evento exitoso, validamos que si la función terminó correctamente cambiamos el color del botón y el texto, pero para eso primero removemos una que tenga, y luego le añadimos la que deseamos, porque si solo le agregamos una

clase, el puede tomar una de las dos, por lo regular será la segunda, pero no es correcto dejarle clases que no están haciendo nada, esta parte es ya por buena practica mas que otra cosa.

```
<button data-event-id="@item.Id" class="btn btn-default btn-sm pull-right ">
    Going?
</button>
```

@section scripts

```
{
    <script>
        $(document).ready(function () {
            $(".js-toggle-attendance").click(function (e) {
                $.post("api/attendances", $(e.target).attr("data-event-id"))
                .done(function () {
                    $(e.target).removeClass("btn-default").addClass("btn-info").text('Attending')
                });
            });
        });
    </script>
}
```

Si nos fijamos a este punto estamos utilizando el e.target dos veces lo que quiere decir que jquery va a recorrer todo el documento en busca de este elemento en dos ocasiones, lo cual es incorrecto. Es mas eficiente hacerlo una sola vez y almacenarlo en una variable. Ahora tenemos algo más chulo, pero necesitamos llenar que pasa si falla la petición, por ahora solo vamos a mostrar una alerta.

```
$(".js-toggle-attendance").click(function (e) {
    var button = $(e.target);
    $.post("api/attendances", button.attr("data-event-id"))
    .done(function () {
        button.removeClass("btn-default").addClass("btn-info").text('Attending')
    })
    .fail(function () {
        alert('Something fail');
    });
});
```

Sin embargo esto nos va a dar un error, esto funcionaria nítido si estuviéramos trabajando en el api haciendo una llamada normal,

pero en nuestro caso le especificamos que el eventId lo va a tomar desde el cuerpo, por lo cual nosotros tenemos que armarlo para que él pueda bindear la propiedad, una forma de hacerlo en el caso que tenemos un parámetro único es enviando un string vacío como parámetro y el valor, o el nombre de la variable como parámetro, ya eso está a opción de ustedes,

```
$.post("api/attendances", {"": button.attr("data-event-id") })
```

Sin embargo, déjame mostrarte una forma mas limpia de hacer eso, que es especificando el parámetro con el cual el se va a bindear, ósea con el que se va a enganchar.

```
$.post("api/attendances", {eventId: button.attr("data-event-id") })
```

Esta es una mejor forma de hacer las cosas, más elegante.

DTOs

Vamos a echar un primer vistazo a los dto, yo quiero crear una nueva clase para almacenar este parámetro que recibo, voy al inicio del archivo (estamos en nuestro api) creo una nueva clase que llamare AttendanceDto, el dto es data transfer object, es un patrón usado para transferir información a través de procesos, aquí nosotros tenemos pedazos de código corriendo en el cliente, y otras piezas corriendo en el servidor, cuando queremos comunicarnos entre estos dos procesos, nosotros podemos usar un dto, no es correcto que tu expongas tus modelos completos al publico y menos con toda la data; la tabla de la junta para ciudadanos, tiene mas de 60 campos, pero cuando le tiras al api de la junta solo te devuelve el nombre y el colegio electoral, bueno, la junta es un mal ejemplo porque esos ni MVC usan, pero si

usaran mvc tendrían el mismo concepto de ocultación del modelo, se que parece tonto repetir lo mismo que ya tenemos y que parece duplicidad de código, pero créame que es por temas de seguridad.

Después lo ponemos en un archivo separado, creamos una nueva carpeta para los dtos y reemplazamos nuestro método para que reciba en lugar de una petición que tenga que extraer un elemento, lo que reciba sea una petición de lo que sea de donde el va a intentar machear todas las propiedades que le envíen con el modelo en cuestión que hayamos escogido, el punto es que veamos que hay varias formas de recibir información.

```
public async Task<IActionResult> Attend(AttendanceDto dto)
```

Probamos y vamos a tener un error, inspeccionamos elementos, y podemos ver en la consola que se nos devolvió una respuesta que dice que algo fallo, la respuesta es porque en el tópico anterior ya registramos que asistiríamos al primero evento, y ahora estamos validando que si ya existe una asistencia para un evento x me retorne un notfound con el mensaje que tenemos en pantalla, por lo tanto para fines de prueba y que no digan que yo hablo mentira, vamos a probar con otros de los eventos, ya en el futuro nos concentramos en que cuando cargue por defecto nos cambie de color cuando ya hayamos marcado que deseamos asistir.

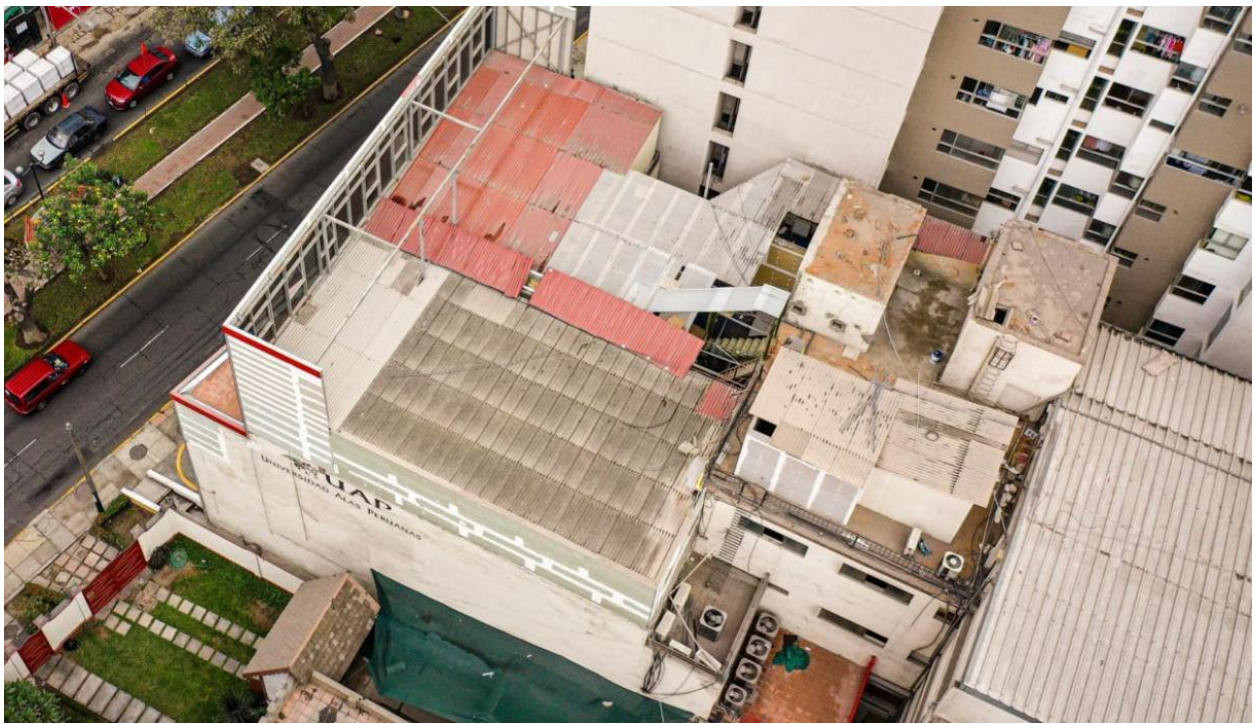
Resumen

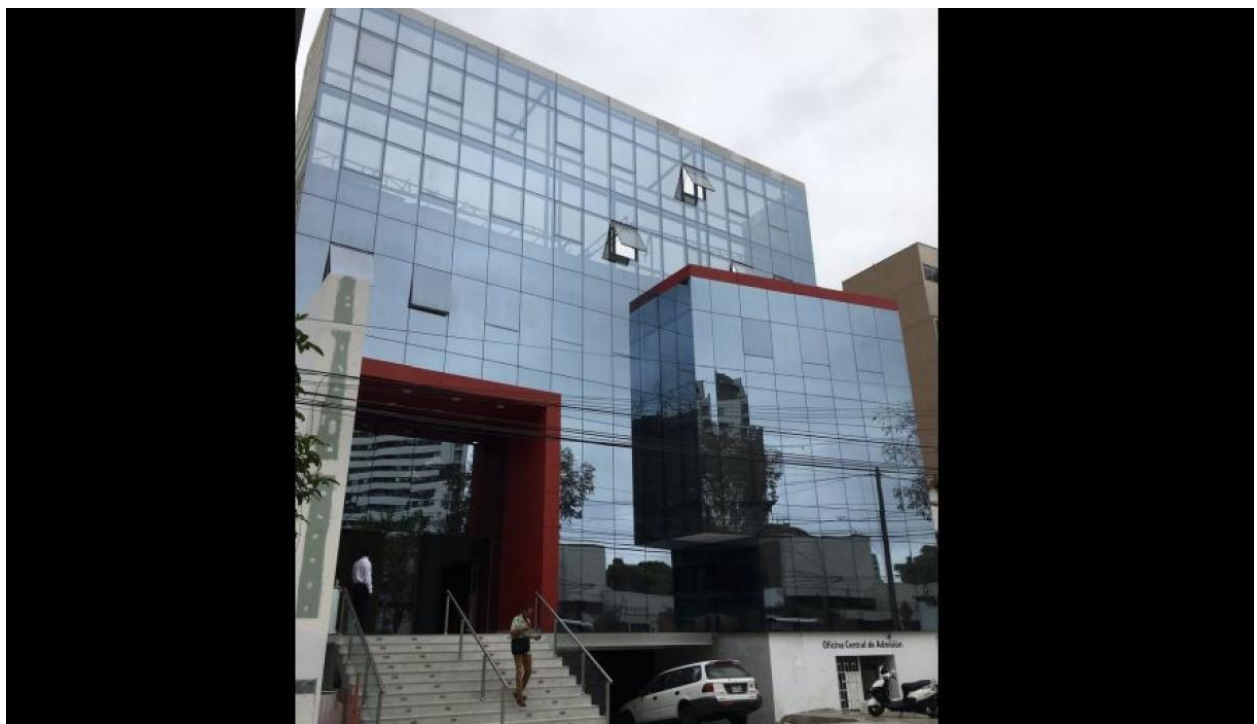
En este modulo implementamos otro de los casos de uso de arriba abajo, empezamos creando un modelo simple, un modelo simple para resolver el problema que tenemos en la mano, nada más, de

nuevo enfatizo una vez más, no intentes modelar el universo en tu app, solo modela los aspectos que importan para el problema, que intentas resolver.

Construimos un API y la probamos con postman, y finalmente implementamos AJAX, para evitar recargar las paginas con llamadas al servidor.

Cada aspecto de este modulo fue enfocado en una única cosa a la vez, no intentes hacer muchos cambios de una, enfócate en una pequeña pieza de funcionalidad a la vez, no eres mas productivo cuando haces muchas cosas al mismo tiempo, eres mas productivo cuando resuelves de a poco muchos pedacitos de funcionalidades, si lo traspolamos a la vida real, imagina que un edificio le construyan un pedazo de la zapata, pero de una vez comienzan a poner la puerta y las ventanas de la primera planta, pero aun no hemos tirado la base de la primera planta cuando ya queremos montar el techo sin haber tirado ninguna de las columnas, aunque en el mundo real si suceden este tipo de cosas, tal es el caso de la universidad de Telesup, la cual no es mas que una empresa con la finalidad de estafar.





Siempre que te enfocas en una cosa, eres mas productivo y produces mejor código. Bien, como ejercicio te corresponde a ti completar el caso de uso de “Seguir a un Coach”

Implementando los casos de uso secundarios

Ya estamos en las últimas secciones, acá vamos a implementar los restantes casos de uso para finalizar nuestra primera iteración, como parte de esta sección voy a mostrarte algunas técnicas chulas para aparentar que sabes mucho programar, veremos cómo usar los Partial views, y algunos trucos de CSS,

Ok, vamos a echar un vistazo a lo que hemos realizado hasta el momento para implementar y seguir a un coach, que era la tarea que tenias asignada, pero se que no realizaste, pero nada, así es que son de mal agradecidos.

Al frente de cada evento al lado del nombre del coach, tenemos un link que dice follow, cuando hacemos clic el cambia a following, con el tiempo implementaremos el detalle del evento donde podremos mostrar información adicional y también mostrar un botón de seguir, pero para eso, falta,

Para esto añadí una nueva clase de dominio llamada following,

```
public class Following
{
    [Key]
    [Column(Order = 1)]
    public string FollowerId { get; set; }
    public ApplicationUser Follower { get; set; }

    [Key]
    [Column(Order = 2)]
    public string FolloweeId { get; set; }
    public ApplicationUser Followee { get; set; }
}
```

Esta posee un primary compuesto, tenemos dos propiedades de navegación, el funcionamiento es muy similar a la clase attendance, después de esto modificamos las propiedades de navegación en el application user para que albergue una colección de following con dos nombres distintos para poder hacer queries sobre estos. Para poder hacer un map doble. Ambas son colecciones que inicializo en el constructor,

```
public class ApplicationUser : IdentityUser
{
    [Required]
    [StringLength(100)]
    public string Name { get; set; }

    public ICollection<Following> Followers { get; set; }
    public ICollection<Following> Followees { get; set; }

    public ApplicationUser()
    {
        Followers = new Collection<Following>();
        Followees = new Collection<Following>();
    }
}
```

Siempre que añadas una propiedad y esa propiedad sea una colección, procura siempre inicializarla en el constructor, porque es responsabilidad de esa clase inicializar sus colecciones,

Luego añadimos un nuevo DbSet, en el DataContext

```
public DbSet<Following> Followings { get; set; }
```

Luego en el on model creating, configuramos la relación entre el application user y el following

```
builder.Entity<Following>().HasKey(p => new { p.FolloweeId, p.FollowerId });

builder.Entity<Following>()
    .HasOne(a => a.Follower)
    .WithMany(c => c.Followers)
    .HasForeignKey(a => a.FollowerId).onDelete(DeleteBehavior.Restrict);
```

```

builder.Entity<Following>()
    .HasOne(a => a.Followee)
    .WithMany(c => c.Followees)
    .HasForeignKey(a => a.FollowerId).OnDelete(DeleteBehavior.Restrict);

```

Corremos una migración, y actualizamos la base de datos

Después de eso creamos un API

Muy similar a la anterior

```

[HttpPost]
public ActionResult Follow(FollowingDto dto)
{
    var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);

    var exists = _context.Followings.Any(a => a.FollowerId == userId &&
a.FollowerId == dto.FolloweeId);
    if (exists)
        return BadRequest("The attendance already exists");

    var follow = new Following
    {
        FolloweeId = dto.FolloweeId,
        FollowerId = User.FindFirstValue(ClaimTypes.NameIdentifier)
    };
    _context.Followings.Add(follow);
    _context.SaveChanges();
    return Ok();
}

```

Después vamos a la vista y agregamos un link y en el jquery hacemos lo siguiente

```

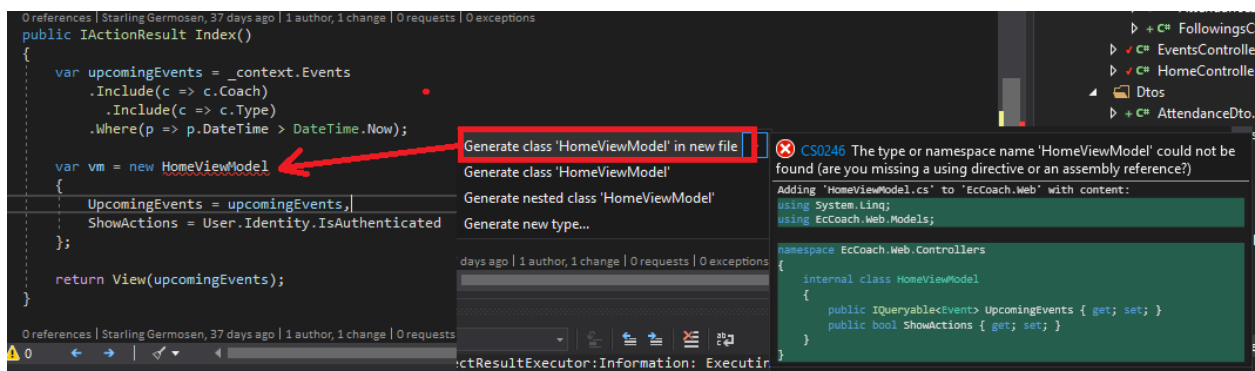
$(".js-toggle-follow").click(function (e) {
    var button = $(e.target);
    $post("/api/followings", { followeeId: button.attr("data-user-id") })
        .done(function () {
            button.text("Following");
        })
        .fail(function () {
            alert("Someting fail");
        })
});

```

Ocultando acciones de usuarios anónimos

Estaba probando la aplicación y me percate de algunos problemas, a pesar de estar deslogueado, puedo ver las acciones following y going me interesa que estas se muestren solo para usuarios autenticados.

Volvemos a vs, yo necesito que el controlador sea el que me diga si un usuario está logueado o no, por lo que nos vamos al controlador y nos vemos con la necesidad de crear un campo adicional yo podría usar un viewbag pero ya saben que la data se perdería, este campo no va a ir a la base de datos, eso quiere decir que en verdad lo que necesitamos es un viewmodel, algo como esto



Después nos ponemos arriba de la clase con la estructura que deseamos y le damos control punto y le decimos que cree la clase después la movemos a donde va

```
public class HomeViewModel
{
    public IEnumerable<Event> UpcomingEvents { get; set; }
    public bool ShowActions { get; set; }
}
```

Yo prefiero cambiar este tipo de IQueryable que trae por defecto a IEnumerable, porque cuando paso el modelo a la vista yo quiero poder iterarla, yo no quiero que en la vista sea posible extender el query, esa no es responsabilidad de una vista, ahora configuramos que vamos a mandar el vm en lugar del objeto y por ende tenemos

que cambiarlo en la vista, cambiamos el enumerable de events a un único objeto o modelo del tipo `homeviewmodel` que dentro contiene un elemento y una lista que vamos a iterar.

```
@model EcCoach.Web.ViewModels.HomeViewModel
```

```
@{  
    ViewData["Title"] = "Home Page";  
}
```

Obviamente comenzaremos a tener errores, en lugar de iterar sobre el modelo, vamos a iterar sobre un objeto dentro del modelo.

```
@foreach (var item in Model.UpcomingEvents)  
{ ... }
```

Y configuramos que solo vamos a renderizar esos botones solo si tenemos la opción de mostrar acciones seteada en true, probamos y vemos que no está, si nos logueamos comprobamos que ahora si aparezcan.

Baches en los requerimientos

Bien, si volvemos a nuestra lista de casos de uso, podemos ver que hemos implementado los siguientes: Añadir Evento, Ver los Eventos, Marcar eventos a los que deseo asistir, Seguir un Coach

Sin embargo hay un punto, a pesar de que implementamos el añadir un evento al calendario, no tenemos forma de verificarlo, una vez mas nos topamos con un tema, el requerimiento dice que el usuario puede añadir un evento a su calendario, en ningún lugar dice que debe poder verlo, pero, seamos sinceros, ¿volverías a recontratar a un profesional que te diga que te va a cobrar adicional por poder mostrar el calendario cuando aunque yo no te lo dije, es lógico que eso estaba implícito?, sé que podríamos decir que nos estamos contradiciendo con relación al tema de los

requerimientos, pero es que, hay cosas que tu vas a tener que hacer gratis aunque no te la hayan pedido, si quieres que te recontracten pero habrá otras donde tengas que intentar convencer a tu cliente de la inviabilidad de lo que propone, así como también habrán ocasiones que cuando no te pidan algo aunque tu sepas que lo necesita, no deberías incluirlo porque te implica una pérdida de tiempo, como por ejemplo el crud de las tablas que se llenan de cada año un día.

Implementado los casos de soporte

Ahora nos toca implementar nuestro modulo para ver los eventos a los que yo deseo asistir, el cual creamos como un caso de uso secundario o de soporte.

Volvemos a Vs, nos vamos a EventsController

Dentro creamos una nueva acción.

Necesitamos la lista de eventos a los que yo voy a atender, una vez tenemos nuestro filtro principal tenemos que valernos de LINQ, porque yo no quiero solo el numero del evento al que deseo ir, sino los datos del evento como tal, y finalmente le doy ToList(), para que mi query se ejecute de forma inmediata y lo convierta en una lista.

```
[Authorize]
public ActionResult Attending()
{
    var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);
    var events = _context.Attendances
        .Where(a => a.AttendeeId == userId)
        .Select(a => a.Event)
        .ToList();
}
```

Yo deseo que esa vista se parezca a lo que tengo en el index, así que vamos a pasar de un pronto por allá, ahora bien, si yo quiero algo idéntico a lo que tengo en un lugar y estoy usando los mismos objetos, resulta ilógico tener que copiar el mismo código en otro lugar, por eso vamos a recurrir a algo llamado partial view, del cual habíamos visto ya uno, pero no teníamos idea de lo que era, así que vamos a crear un Partial View.

Partial View

Son unas vistas que sirven para cargar información parcial de algún contenido, muy usadas cuando queremos tener diversos modelos dentro de una misma pantalla.

Para crear un partial view, debemos hacerlo en la carpeta del controlador que vaya a consumir el partial, pero como en este caso dos vistas de dos controladores tenemos que crearla en la carpeta Shared, le damos clic derecho, añadir nueva vista, seleccionamos que es una vista parcial, escogemos el nombre que debe terminar en la palabra Partial, y por costumbre se usa el underscore, no es que sea un requisito, pero es ya por tradición, después de que se genera cortamos el contenido de la vista Index que nos interesa mover (toda la parte de la iteración)

Y después en el index donde solía estar el foreach, ponemos la renderización del partial y le mandamos la data que en nuestro caso está en el modelo.

```
@await Html.PartialAsync("_EventsPartial", Model)
```

Una vez realizado esto, podemos ver que el nombre homeviewmodel, como que no cuadra con el nombre de la pantalla

en la que nos encontramos, aca tengo dos opciones, copiar la clase homeviewmodel con otro nombre, lo cual, por mas estúpido que le parezca lo hacen muchos desarrolladores expertos a pesar de que ninguna buena practica plantea eso, y la otra es renombrar homeviewmodel por otra cosa, ahora bien, te preguntaras, porque no dejarle ese nombre? Porque las clases tienen que ser lo suficientemente descriptivas, tu no debes usar una clase llamada home en una pantalla que no sea la home, en nuestro caso la llamaremos eventsviewmodel.

Bien, en el layout nosotros habíamos puesto dos links que estaban de sobra, vamos a ponerlo a trabajar ahora,

Si corremos, podemos probar ir a los eventos a los que estoy atendiendo, y verlo vacío, vamos al home le doy click a uno de los eventos que deseo ir y todo pasa jevi, vuelvo a la lista y revienta, porque en ningún momento he cargado el listado de coach o types que, por eso debo incluirlos en el query,

```
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();

    var vm = new HomeViewModel
    {
        UpcomingEvents = events,
        ShowActions = User.Identity.IsAuthenticated
    };

    return View(vm);
}
```

Podemos refrescar y ver que todo está nítido, lo único que nos faltaría es que por defecto el cargue el label con otro color

indicándonos que ya marcamos como que quiero ir, pero eso será responsabilidad de otro curso, además de que ustedes pueden meter mano con eso.

Refactorizando usando el principio DRY

Tenemos dos vistas que lucen exactamente igual, este principio que reza “dont repeat yourself” nos dice que debemos anular las cosas que son idénticas.

A pesar de que nosotros tenemos un partial, seguimos teniendo dos vistas que hacen exactamente lo mismo, y ambas se ven igual, no hay diferencia entre una y otra, por eso debemos hacer algo mas que usar un partial en casos como este con el que nos topamos acá, no tiene sentido que dos vistas tengan el mismo código, así que vamos a eliminar una de ellas, ahora bien, cuando tenemos una vista que debe ser común entre dos controladores, esta debe estar obligatoriamente en el folder share, así que la que esta en el home la movemos;

Sin embargo, no tiene sentido que algo que almacene un listado de todos los eventos, en especial si no sabemos que en nuestro proyecto vaya a tener otras acciones con el mismo nombre, se llame Index, por lo cual, vamos a renombrarlo a algo que haga mas sentido, como Events por ejemplo, ahora, tenemos que ir al controlador a indicarle que se ubique en otro lugar, corremos el app y verificamos que ambas pantallas estén bien y no hayamos dañado nada.

```
return View("Events", vm); // return View(vm);
```

Ahora como perdimos el h1 que teníamos con el título lo que vamos a hacer es que vamos a crear una nueva propiedad en el viewmodel que llamamos Heading y la llenaremos con un título distinto dependiendo de donde lo estemos llamando.

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

Resumen

Bien, ya técnicamente aprendimos todo lo necesario para aprender a meter mano con net core, virtualmente no existe un tipo de aplicación que no puedas realizar basado únicamente en lo aprendido acá, obviamente sin mucha organización o un código más limpio, pero al menos con lo aprendido aquí puedes decir a toda honra que metes mano en dot net usando netcore con entity Framework, te toca a ti implementar el caso de uso faltante.

Fin

Primero, si llegaste hasta aquí es una prueba de tu pasión y tu determinación para aprender nuevas cosas, y el deseo de mejorar tus habilidades primero que nada deseo felicitarte por eso, segundo, deseo decirte... “gracias”, gracias por permitirme ser tu instructor y estar en este curso de principio a fin, espero grandemente que hayas aprendido cosas nuevas pero nuestro viaje aún no termina, aún hay dos partes adicionales, aun nos falta implementar las operaciones CRUD, y refactorizar nuestro código al modelo orientado a objetos, nos falta aprender a usar componentes, y de igual forma aprender a usar librerías, también aprender las bases de less, el cual nos permite tener un css más mantenible en el tiempo, de igual forma nos falta la creación de animaciones y otras técnicas que puedes usar en diversos proyectos, por lo que espero verlos en la siguiente parte, cuando me decida lanzarla, hasta la próxima :D

RUD... la C la hicimos hace rato

Read y el Dilema de la reutilización (reutilizar o no reutilizar).

Bien lo primero que vamos a implementar es la lectura, aunque ya esto lo hemos hecho, pero la idea es explicarlo y poner en perspectivas algunos temas, y vamos a hacerlo con el caso de uso “**mis próximos eventos**”, para eso en nuestro controlador debemos crear una nueva acción, pero, pero, pero, si ya yo lo que voy a mostrar es eventos y ya tengo una pantalla a la cual cargamos eventos por diferentes tipos de filtros, uno para los eventos a los que deseo asistir y el otro para todos los eventos futuros, en teoría yo puedo resolver de la misma forma en que ya trabajé antes, simplemente poniendo que mi acción apunte a mi pantalla de eventos con el filtro correspondiente, bueh, resulta que aquí viene una paradoja que rara vez la tienen los desarrolladores junior, porque ellos de antemano ni siquiera conocen el principio DRY, y ellos la tienen fácil desde el comienzo, sin embargo nosotros que ya somos super desarrolladores nos topamos con esta problemática, “No siempre reutilizar es una buena idea”.

Si inspeccionamos la vista genérica para “Events”, podemos ver que tenemos botones que no se van a mostrar en la vista que yo quiero crear, es decir, si yo quiero ver los eventos que yo acabo de crear, no hace sentido que yo tenga un botón que diga going, pues es obvio que yo voy, yo creé el evento, tampoco tiene sentido que yo proponga seguirme a mí mismo, además, habrán acciones que debo agregar que no se deben ser mostrados en la pantalla del home o de los eventos a los que deseo asistir, por lo que, en este caso en particular, no es una buena idea reutilizar, porque yo

no quiero tener un Frankenstein lleno de cosas ocultas y misteriosas.

Otra cosa que me motiva a duplicar en lugar de reutilizar es que aquí hay un javascript que solo aplica para la pantalla upcoming events y los eventos a los que deseo asistir, pero no aplicaría en una nueva, claro yo sé que no debería siquiera estar en nuestra pantalla la sección de scripts sino en nuestra carpeta de scripts, pero eso es algo que arreglaremos más tarde.

Copiamos el contenido tanto de la vista Events como la de su partial y lo pegamos en una nueva vista que llamaremos “Mine”, voy a remover los botones de going and following y también voy a remover el nombre del coach porque no tiene sentido que yo vea mi propio nombre en el evento, en lugar de eso prefiero mostrar el venue, en este caso en particular podemos tirarle directamente al modelo y no necesitamos un ViewModel intermedio, como no es nada que no hayamos visto antes, solo te mostraré el código final.

EventsController

```
[Authorize]
public ActionResult Mine()
{
    var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);

    var events = _context.Events
        .Where(a => a.CoachId == userId && a.DateTime > DateTime.Now)
        .Include(p => p.Type)
        .ToList();

    return View(events);
}
```

Vista

```
@model ICollection<EcCoach.Web.Models.Event>
@{
    ViewData["Title"] = "My Events";
}
```

```

}
<ul class="events">
  @foreach (var item in Model)
  {
    <li>
      <div class="date">
        <div class="month">
          @item.DateTime.ToString("MMM")
        </div>
        <div class="day">
          @item.DateTime.ToString("d ")
        </div>
      </div>
      <div class="details">
        <span class="coach">
          @item.Coach.Name
        </span>
        <span class="type">
          @item.Type.Name
        </span>
      </div>
    </li>
  }
</ul>

@section scripts
{
  <script>
    $(document).ready(function () {
    });
  </script>
}

```

Ahora tenemos que ir al layout donde vamos a agregar un nuevo link el cual nos va a permitir llegar a la acción que acabamos de crear, nos paramos arriba de cualquiera de las líneas y presionamos control + d para duplicar la línea, cambiamos y corremos, creamos un evento, y en la ruta para los eventos podemos ver este nuevo que hayamos creado con el usuario que estemos logueado.

```

<li class="nav-item">
  <a class="nav-link"
    asp-controller="Events"
    asp-action="Mine"> My Events
  </a>
</li>

```

Vamos ahora a cambiar algo, si el usuario acaba de crear un evento, actualmente él va al home, pero lo que hace más sentido es que se vaya a el listado de los eventos que le pertenecen. Por lo cual nos vamos al post de create y hacemos la redirección hacia la lista de Mine.

```
return RedirectToAction("Mine");
```

En este tópico estábamos implementando la parte Read de un Crud, aprendimos que no siempre es bueno reutilizar, en especial cuando la reutilización va a convertir nuestro código en una maraña de cosas horribles.

Reusar

- Con: Puede incrementar la complejidad.
- Pros: Previene el tener que realizar cambios en múltiples lugares.

Duplicar

- Pros: Piezas separadas que crecen y se pueden cambiar de manera independiente.

Update

En este topico vamos a refactorizar nuestro formulario de crear eventos y reutilizarlo con la finalidad de este nos sirva para actualizar registros también.

Vamos a poner en el formulario de my Events, un link para editar el evento, pero vamos a hacerlo de una manera chula, así que vamos a vs, abrimos myEvents, vamos a agregar otro contenedor llamado acciones justo debajo del de *details*. (*div.actions + tab*),

después agregamos dos links vacíos por ahora uno para editar y el otro para eliminar.

Nos vamos al site.css donde para esa clase, le asignamos un fontsize más pequeño, también le ponemos display none para que no los muestre por defecto, entonces creamos otro para que cuando se le haga hover se muestre y de esa forma tengamos un diseño chulambrico sin necesidad de usar JavaScript o JQuery.

```
.events > li .actions {  
    font-size: 14px;  
    display: none  
}  
  
.events > li:hover .actions {  
    display: block  
}
```

Ahora necesitamos una acción en el controller para editar. Para ello copiamos el create completo y lo modificamos conforme a nuestras necesidades, lo primero que debemos hacer es recibir un id, ahora bien, vamos a decir que yo quiero buscar el evento según el Id que me hayan pasado, ¿es correcto hacer esto?

```
public IActionResult Edit(int id)  
{  
    var ev = _context.Events.Single(g => g.Id == id);  
}
```

No, eso es una brecha de seguridad enorme, con esta implementación cualquiera puede editar el evento de cualquier otro, solo pasando en la barra de navegación el evento por número, si cada evento me pertenece solo a mí, solo yo debo poder editar el mismo.

Y yo tampoco debería poder editar el de los demás, así que necesitamos una segunda condición aquí, lo cual nos llevara en próximos tópicos a implementar el patrón currentfactory user para no estar llamando tanto la clase abstracta de User, pero para eso, falta.

Bien, después tenemos que pasarle los datos que vienen al viewmodel y finalmente una práctica que a mí en lo personal no me gusta pero que veo que la mayoría de los gurus y la gente que sabe, usa, es usar un mismo formulario para editar y crear, si, es más fácil hacer ciertas cosas pero en lo personal prefiero mis dos formularios aparte, pero ese no es el tema, yo voy a seguir los ejemplos como lo dicen los expertos, después al finalizar les mostrare como lo prefiero yo, y usted elija la que más le guste.

Así que, en este caso, como lo hacen los expertos, es que le vamos a tirar a la acción create, pero, este nombre no es correcto para ser utilizado en otra cosa que no sea crear, así que debemos renombrarlo, pero eso será luego, por ahora vamos a enfocarnos en el problema en sí que tenemos actualmente.

```
public IActionResult Edit(int id)
{
    var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);
    var ev = _context.Events.Single(g => g.Id == id && g.CoachId == userId);
    var types = _context.Types;

    var vm = new EventViewModel
    {
        TypeId = ev.TypeId,
        Date = ev.DateTime.ToString("dd/MM/yyyy"),
        Time = ev.DateTime.ToString("HH:mm"),
        Venue = ev.Venue,
        Types = new SelectList(types, "Id", "Name")
    };

    return View("Create", vm);
}
```

Y después nos vamos al link y le ponemos que apunte a la acción edit, ahora bien, nosotros necesitamos pasar parámetros de un lado para otro, esto lo podemos hacer pasando un objeto anónimo como se hacía en Clasic, o podemos usar una de las facilidades que nos da Core que son los atributos de ruteo, donde con asp-route-NombreParametro yo puedo setear cualquier parámetro desde la vista, en este caso vamos a rutear el id por lo que sería asp-route-id

```
<a asp-controller="Events" asp-action="Edit" asp-route-id="@item.Id">  
    Edit  
</a>
```

Bien ahora corremos y al hacer clic en editar podemos ver que nos envía a la acción create, pero con los datos llenos, obviamente tenemos que cambiar la cabecera y también podemos ver que el tipo de evento está vacío ósea el dropdownlist aparece vacío.

Vamos a corregir el tema del dropdownlist, la solución es simple, tenemos que cambiar en nuestro css para la clase form control, en lugar de usar el padding debemos usar la propiedad height que vamos a poner en 44.

```
.form-control {  
    height: 44px;  
    /*font-size: 17px;*/  
    border-radius: 15px;  
}
```

Vamos ahora a renombrar el formulario de la acción create a EventForm, recuerda que tenemos que hacer cambios en el controlador y en el mismo formulario.

Tenemos que hacer lo mismo en la parte get del create porque como ya no se llama igual no puedo decirle que sencillamente le tire a la vista o que retorne la vista normal porque ahora es distinta.

```
return View("EventForm",vm);
```

Corramos y asegurémonos que todo funcione bien, creamos uno y le damos a editar, para verificar que al menos me envía, posterior a eso creamos la propiedad Heading en nuestro ViewModel.

Bien, ahora nosotros necesitamos que la acción llame, en caso de que venga de create, llame una cosa y ejecute un proceso, mientras que, si viene de edit, llame otra, por lo que vamos a crear en nuestro viewmodel una nueva propiedad llamada acción.

Sin embargo, no voy a ir a mi controlador a llenar, vamos a hacer las cosas diferente, una forma de hacer las cosas es crear una propiedad "Id" que si viene vacía el mismo view model me retorne que se está intentando crear algo, y si viene llena esta me retorne que se quiere actualizar, no hay una forma más correcta que la otra, es solo para que vean que hay diferentes formas de lograr tu objetivo.

```
public class EventViewModel
{
    ...

    public string Heading { get; set; }

    public int Id { get; set; }

    public string Action
    {
        get { return (Id != 0) ? "Update" : "Create"; }
    }
}
```

Pero, hay un problema por estar usando magic string, es que en el futuro si renombro estas acciones el código se va a romper pero no en tiempo de compilación, sin embargo eso lo resolvemos después, por este momento esto funciona, ahora bien, nosotros cuando mandamos a la vista unos datos, si por alguna razón la vista no le envía para tras estos datos, el view model los arroja a la basura, para evitar eso entonces cuando nosotros mandamos el Id en el controlador, debemos almacenarlo en un campo oculto dentro de nuestra vista, porque no tiene sentido que se le muestre al usuario además de que no queremos que este lo pueda modificar.

```
<input type="hidden" asp-for="Id" />
```

Ya que estamos aquí también tenemos que modificar la cabecera del form o más bien a donde el form va a hacer Post, por nuestra variable dinámica que creamos llamada action.

```
<h1> @Model.Heading </h1>  
<form asp-action="@Model.Action">
```

Bien, ahora modifiquemos nuestro post para update el cual viene de copiar el post del create, si el modelo es válido, entonces nosotros tenemos que en lugar de crear un nuevo objeto de tipo evento, debemos de actualizar el existente.

```
[HttpPost]  
public IActionResult Update(EventViewModel vm)  
{  
    if (!ModelState.IsValid)  
    {  
        var types = _context.Types;  
        vm.Types = new SelectList(types, "Id", "Name", vm.TypeId);  
        return View("EventForm", vm);  
    }  
  
    var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);
```

```

var ev = _context.Events.Single(g => g.Id == vm.Id && g.CoachId == userId);

ev.DateTime = vm.GetDateTime();
ev.TypeId = vm.TypeId;
ev.Venue = vm.Venue;

_context.SaveChanges();
return RedirectToAction("Mine", "Events");
}

```

Evitando los Magic Strings

Ahora vamos a entender un tema profundo de .net y que es la única razón por la cual ponemos el viewmodel a que nos diga que acción va a ejecutar, vamos al viewmodel, en la parte en que retorno “create” o “update” yo deseo retornar a base de una expresión lambda algo como esto

```

get {
    var update = (c => c.Update());

    return (Id != 0) ? "Update" : "Create";
}

```

Ahora bien, ¿Qué es esto?

```

get {
    var update = (c => c.Update());

    return (Id != 0) ? "Update" : "Create";
}

```

Esto es un **método anónimo**, así que nosotros podemos usar un delegado llamado func para representarlo, al cual le pasamos con el delegado func<¿Cuál es el argumento de ese método? donde “e” representa a nuestro eventscontroller,

```

get {
    Func<EventsController, > update = (c => c.Update());

    return (Id != 0) ? "Update" : "Create";
}

```

¿Que retorna este método? cuando llamamos la acción update, nosotros retornamos un ActionResult

Así que nosotros necesitamos poner un ActionResult, como posibilidad de retorno.

```
get {  
    Func<EventsController, ActionResult> update = (c => c.Update());  
    return (Id != 0) ? "Update" : "Create";  
}
```

Ahora tenemos un problema porque la acción update espera un viewmodel, así que yo le voy a pasar this, podemos pasar null si queremos, en este caso no importa, nosotros no la vamos a llamar, solo la vamos a almacenar como una expresión.

Repasemos, este lambda representa a un método que toma a “c” como argumento, y retorna un action result, en c# podemos usar func que es un delegado que representa eso, el primer argumento es un input como método anónimo, y el segundo argumento es el tipo de retorno, pero, como yo no quiero llamar el método sino solo encapsularlo como una expresión, puedo usar de forma directa una expresión de func de todo lo demás, esto para evitar que cuando pase por acá se llame, aunque si mi interés es hacerlo puedo sin problemas solo que tendría que pasar this en lugar de null, hago lo propio para el create.

Ahora bien, una vez tengo esto, yo no puedo venir a pasar simplemente estas dos expresiones acá, las cosas no son tan fáciles.

Debo crear una variable llamada action donde yo voy a almacenar una de esas dos expresiones, entonces luego determinada cuál de las dos expresiones es la que voy a usar, le extraigo el nombre, para extraer el nombre se usa la siguiente expresión.

```
get {  
  
    Func<EventsController, ActionResult> update = (c => c.Update(this));  
    Func<EventsController, ActionResult> create = (c => c.Create(this));  
  
    var action = (Id != 0) ? update : create;  
    return (action.Body as MethodCallExpression).Method.Name;  
  
}
```

En este tópico aprendimos a como se realizan las actualizaciones, las cuales son similares a las operaciones Create, reusamos el formulario para que ahora haga dos cosas diferentes dependiendo de una variable, como nosotros tenemos varias clases y tópicos solo hablando del create, imaginen que desde ese momento hubiéramos planteado también las cosas necesarias para actualizar, sin embargo cuando ya terminamos nuestros casos de uso principales, podemos de forma más efectiva simplemente reemplazar un par de cositas y listo.

Aprendimos a evitar los magic string y su debilidad, por eso usamos complicadísimas expresiones lambda, que, si les soy sincero, yo trato de ser consistente en los nombres para evitar el renombrado de una etiqueta y uso de forma directa magic strings.

Delete

Normalmente para fines de consistencia nosotros no deseamos eliminar realmente la información, sino que por asuntos de negocio

deseamos que estos queden a modo de histórico por si las moscas, por lo cual, para esto se implementa el borrado lógico en lugar del borrado físico, que no es mas que inactivar un registro, imagina que se pudieran borrar las transacciones de una venta, las inconsistencias y los fraudes fiscales que sucederían constantemente. Además, con el borrado lógico evitamos por defecto los problemas de borrado en cascada.

En el contexto de nuestra aplicación para eventos, seria mas bien para proporcionarle al usuario la habilidad de arrepentirse de cancelar un evento y darle la facilidad de poder reactivarlo.

Para esto vamos a Vs, vamos a nuestra clase Event, y vamos a añadir una nueva propiedad llamada IsCanceled de tipo booleana, podríamos usar Deleted u otra palabra pero como son eventos que cuando no lo vamos a dar se cancelan prefiero usar esta palabra en este contexto, creamos una migración y actualizamos la base de datos, lo siguiente que haremos es añadir un link al lado del Edit de los eventos en la vista **Mine**, este va a llamarse Delete o Cancel y después vamos a crear un api similar a como ya hicimos otras previamente para indicar que asistiremos a un evento o no, podemos copiar la de Attendance y renombrarla a EventsController que no debe confundirse con el EventsController para la vista, este nuevo va a ser para el api el anterior es para la vista.

Clase:

```
public bool IsCanceled { get; set; }
```

Controlador:

```
[HttpDelete]
public IActionResult Cancel(int id)
{
    var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);
```



```

    var ev = _context.Events.Single(a => a.CoachId == userId && a.Id == id);
    ev.IsCanceled = true;
    _context.SaveChanges();
    return Ok();
}

```

Vista:

```

<a class="js-cancel-event" data-event-id="@item.Id">
    Cancel
</a>

```

Script:

```

@section scripts
{
    <script>
        $(document).ready(function () {
            $(".js-cancel-event").click(function (e) {
                var link = $(e.target);

                if (confirm("Are you sure you want to delete this event?")) {
                    $.ajax({
                        url: "/api/events/" + link.attr("data-event-id"),
                        method: "DELETE"
                    })
                    .done(function () {
                        link.parents("li").fadeOut(function () {
                            $(this).remove();
                        });
                    })
                    .fail(function () {
                        alert("Something failed!");
                    });
                }
            });
        });
    </script>
}

```

Bien, sin embargo, aunque removemos del dom con el remove el elemento, si refrescamos la página podemos ver que carga de nuevo, esto se debe a que independientemente de todo estamos diciéndole a nuestro controlador que cargue esos datos, por lo que debemos ir y excluirlos en la acción mine del controlador de Events.

```
&& !a.IsCanceled
```

De acuerdo, ya que tenemos la funcionalidad de cancelar, cuando un coach o usuario de nuestra plataforma había marcado que asistiría a un evento y este sea cancelado por su creador, ¿qué creen que deba pasar? ¿Este debería desaparecer mágicamente de la pantalla de attending del usuario? No, por temas de usabilidad deberíamos seguirle mostrando el evento, pero con una señal de que el mismo fue cancelado y de paso si ya está cancelado un evento retornemos un notfound en el api de cancelar.

Api

```
public IActionResult Cancel(int id)
{
    ...
    if (ev.IsCanceled)
        return NotFound();
}
```

Partial View Events

```
@if (item.IsCanceled)
{
    <span class="label label-warning">Canceled</span>
}
```

Evitando Trampas en el modelo de dominio

Uno de los grandes problemas del desarrollo de software es los cambios constantes en el documento de requerimientos, en nuestro documento de requerimientos original, teníamos peticiones como eliminar, crear, consultar y editar un evento, pero no teníamos nada relacionado con que al momento de cancelar un evento se les notifique a los usuarios, eso no estaba contemplado en el documento de requerimientos original, sin embargo ahora se nos presentan unos nuevos escenarios a los cuales no le podemos decir que no al cliente, pero este tipo de cambios si hay que cobrarlos porque escapa del flujo de funcionamiento base del

sistema basado en los requerimientos originales, es decir, en este caso no es algo que se debe asumir tal como el “Ver listado de mis eventos”

Event is Canceled

Event is Updated

Event is Created

Y el cliente desea que para cada escenario se envíe una notificación como las siguientes.

Juan Perez ha cancelado el evento en la puya 52 del 21 de noviembre a las 6pm
Carlos Gonzales ha cambiado la fecha del evento en el hoyo de chulin del 20 de julio/7pm al 22 de julio/7pm
Angel Garcia estará en el aula 17 de Apec el 23 de Diciembre/7pm

¿Qué clases vamos a necesitar para realizar esto?



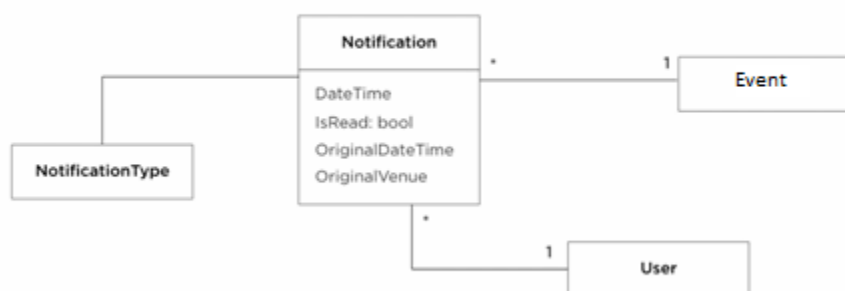
¿Acaso esta puede ser una solución?



Esta estructura cumple perfectamente en nuestra primera interacción, pero no con la segunda. Para solucionarlo podríamos agregar herencia de clases con clases abstractas, pero eso seria agregar mucha complejidad innecesaria. Entonces vamos a tener el siguiente escenario.

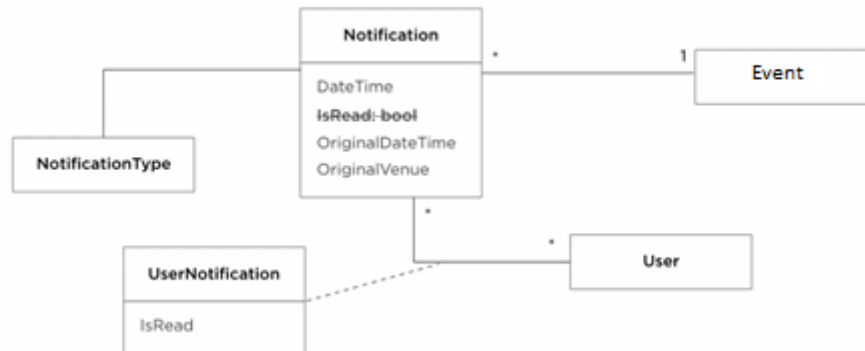


Y dirán algunos, bueno, pero no podemos tener una enumeración y menos una tan limitada porque en el futuro podemos tener la necesidad de tener otro tipo de notificación, pero, ¿qué tal si no? Se nos enseña que tenemos que tener sistemas muy flexibles a cambios, pero, ¿recuerdan el tópico en que aprendimos a no intentar modelar el universo? No nos acostumbremos a agregar complejidad innecesaria que haga in mantenible nuestros sistemas a lo largo del tiempo.



Este es el modelo propuesto, sin embargo, hay un problema, yo voy a necesitar una notificación por cada usuario lo cual es un poco tedioso para mi base de datos, en lugar de eso yo prefiero tener una tabla adicional que solo almacene un solo dato mil veces en

lugar de mil registros que me almacenen 4 datos, por eso vamos a sacar de ahí el Isread y lo movemos a una tabla aparte de esta forma tengo algo mas optimizado para la base de datos.



Vamos ahora a construir nuestro modelo y trabajar en el.

```
public enum NotificationType
{
    EventCanceled = 1,
    EventUpdated = 2,
    EventCreated = 3
}

public class Notification
{
    public int Id { get; set; }
    public DateTime DateTime { get; set; }
    public NotificationType NotificationType { get; set; }
    public DateTime? OriginalDateTime { get; set; }
    public string OriginalVenue { get; set; }

    [Required]
    public Event Event { get; set; }
}

public class UserNotification
{
    [Key]
    [Column(Order = 1)]
    public string UserId { get; set; }

    [Key]
    [Column(Order = 2)]
    public int NotificationId { get; set; }

    public ApplicationUser User { get; set; }

    public Notification Notification { get; set; }

    public bool IsRead { get; set; }
}
```

Agregamos los DbSet y configuramos la relación en el DataContext.

```
...
    public DbSet<Notification> Notifications { get; set; }
    public DbSet<UserNotification> UserNotifications { get; set; }
...

builder.Entity<UserNotification>().HasKey(p => new { p.UserId, p.NotificationId });
```

Agregamos la migración, actualizamos la base de datos y de no haber usado nuestro método para deshabilitar el borrado en cascada genérico aquí tendríamos un error conocido, pero ya lo tenemos cubierto, agregamos la relacion.

```
builder.Entity<UserNotification>()
    .HasOne(a => a.User)
    .WithMany()
    .HasForeignKey(a => a.UserId).OnDelete(DeleteBehavior.Restrict);
```

Nos vamos al api de Events en el cancel y agregamos los pasos necesarios para implementar las notificaciones.

```
public IActionResult Cancel(int id)
{
    ...

    ev.IsCanceled = true;
    var notification = new Notification
    {
        DateTime = DateTime.Now,
        Event = ev,
        NotificationType = NotificationType.EventCanceled
    };

    var attendees = _context.Attendances
        .Where(a => a.EventId == ev.Id)
        .Select(a => a.Attendee)
        .ToList();

    foreach (var attendee in attendees)
    {
        var userNotification = new UserNotification
        {
            User = attendee,
            Notification = notification
        };
        _context.UserNotifications.Add(userNotification);
    }
    _context.SaveChanges();
    ...
}
```

Resumen

Aprendimos a usar los diagramas UML, pero solo para esquematizar ideas y como una herramienta de comunicación, podemos usarlos de forma habitual, pero, estos agregan complejidad y nos hacen caer en el famoso error de que: “Es una pérdida de tiempo o no es correcto empezar por el papel”, al final la documentación debe quedar en tu código el cual se supone que es completamente auto descriptivo, usa los UML solo como pequeños bocetos conceptuales de ideas, no te siñas a ellos a raja tabla porque tus sistemas no deben ser intolerantes, o autoritaristas sin posibilidad de cambios y el uso del uml en la practica conlleva a sistemas robustos que no son sensibles a cambios y también por arte de magia te convierte en un programador Old School.

Sin embargo, en nuestro método de cancelar cometimos un error, y es que estamos delegando a la acción cancelar muchas cosas, y uno de los principios que habíamos hablado en capítulos anteriores era que, es una mala práctica tener algo que se dedique a muchas cosas, pero eso es algo que vamos a arreglar en el siguiente modulo.

Refactorizando para conseguir un diseño orientado a
objetos

Uno de los errores comunes en muchos desarrollos .net, es que al tener una curva de aprendizaje menor a la de otros lenguajes, se cometen muchos errores básicos por inexpertos, uno de ellos es lo que se llama **Anemic Domain Model**, un dominio anémico es un **dominio sin comportamiento** (behaviour), sin esencia, sin métodos, sin lógica, a medida que crece nuestro dominio, comenzamos a delegar la lógica de nuestro dominio a elementos que no tienen que ver con eso, como por ejemplo el controlador o servicios, esto los vuelve in mantenibles, y difíciles de testear ya que el controlador debe ser solamente un coordinador, no quien haga el trabajo, en este modulo vamos a aprender a extraer nuestra lógica de dominio y ponerla en nuestro modelo de dominio, lo cual va a resultar en un modelo de **dominio rico en comportamiento**, esto ayuda a lo que es la separación de conceptos, hacer nuestro sistema más mantenible y fácil de testear.

Quitando responsabilidades al controlador

Si vamos y echamos un vistazo a nuestra acción cancel de nuestro controlador, podemos ver que esta acción hace demasiadas cosas, que no le competen, ¿qué tal si en lugar de ponerlo a el a crear una instancia para cada notificación, nosotros delegamos en la clase correspondiente a que sea ella quien cree esta instancia?

Para eso vamos a crear un método dentro de la clase que vamos a llamar Notify el cual va a recibir la notificación que creamos más arriba y con Ctrl + Punto (.) creamos el método en la clase. Nota: no le pasamos el attendee (user) porque es el mismo y podemos referenciarlo con this dentro de la clase de ser necesario. Después

cortamos el contenido del foreach y nos vamos a este método donde implementamos.

```
foreach (var attendee in attendees)
{
    attendee.Notify(notification);
    ...
}
```

Cambiamos el attendee a this porque estamos dentro del contexto de una instancia a attendee, por ende, cada que llamamos este método tenemos como referencia el elemento completo.

Context es algo relacionado con infraestructura y persistencia de datos, por lo que no debe estar dentro de nuestra clase de dominio, sin embargo, nosotros necesitamos persistir la información antes de que sea enviado al Entity para ser procesado, para esto, si nosotros hacemos uso de una propiedad de navegación tengo el mismo resultado al momento de que yo vaya a guardar el objeto completo por lo que creamos la propiedad de navegación y le incluimos la inicialización en el constructor.

```
public ICollection<UserNotification> UserNotifications { get; set; }

public ApplicationUser()
{
    ...
    Followees = new Collection<Following>();
    UserNotifications = new Collection<UserNotification>();
}
```

Ya con esto podemos sencillamente decirle al método Notify que agregue la notificación a la colección del asistente del evento.

```
internal void Notify(Notification notification)
{
    var userNotification = new UserNotification
    {
        User = this,
        Notification = notification
    };
}
```

```
        UserNotifications.Add(userNotification);  
    }
```

Evitando los Estados Invalidos del objeto

Un pequeño problema que tenemos dentro de la arquitectura que hemos elaborado para nuestra aplicación es que yo, aunque dentro de nuestro contexto no lo estoy haciendo, tengo la posibilidad de crear un objeto que este en un estado no valido, es decir, al momento de crear una notificación yo necesito si o si un usuario y la notificación, sin embargo dado el contexto actual yo puedo hacer lo siguiente:

```
var x = new UserNotification();
```

Esto me crearía un objeto vacío que yo tengo la posibilidad de enviar a base de datos, obviamente EF gritaría y explotaría, pero eso sería ya en producción. Aunque es mi desarrollo y yo se que debo pasar el usuario siempre, otro desarrollador lo podría tomar después e ignorar esa restricción, por lo que vamos a trabajar esa parte.

Nos vamos a la clase de UserNotification creamos el constructor de la misma con ctor + tab Y le pedimos como argumento para inicializar esta clase dos parámetros uno el usuario y el otro la notificación, e inmediatamente pasamos estos como parámetros inicializadores de nuestros objetos internos, podemos hacerlo con Ctrl + punto (.) seleccionando la segunda opción.

```
public UserNotification(ApplicationUser user, Notification notification)  
{  
    User = user;  
    Notification = notification;  
}
```

```
}
```

De esta forma nos aseguramos que estos argumentos no van a ser nulos nunca porque no podrán haber instancias de esta clase sin que se le pase estos dos parámetros. Aunque para asegurarnos de que no nos pasen null deberíamos validar, que el valor pasado no sea nulo porque puede haber uno que se pase de verga.

```
if (user == null)
    throw new ArgumentNullException(nameof(user));

if (notification == null)
    throw new ArgumentNullException(nameof(notification));
```

O podemos hacerlo de una manera más elegante.

```
public UserNotification(ApplicationUser user, Notification notification)
{
    User = user ?? throw new ArgumentNullException(nameof(user));
    Notification = notification ?? throw new ArgumentNullException(nameof(notification));
}
```

De esta forma al momento de programar nos aseguraremos de exigir en todo momento pasar estos dos valores, siempre que hagamos una instancia a esta clase, sin embargo, EF, necesita instanciar las clases y EF no sabe como crear estos objetos, por esto, cada vez que creamos un constructor personalizado, necesitamos crear un constructor por defecto porque EF no tiene forma de llamar el constructor para crear una instancia de UserNotification (por eso yo no hago mucho uso de esta funcionalidad, pero no me hagan caso a mí, sino a los expertos).

Creo uno por defecto, pero lo pongo como protected para asegurarme de que no lo puedan usar los desarrolladores y que me creen eventos inconsistentes.

Otra cosa que tiene esta clase y que también está mal, es que el User y la Notificación, pueden ser cambiadas una vez han sido inicializadas lo cual no es correcto, por ende le quito el set como indicativo de que este solo puede ser obtenido pero no se le puede setear valores fuera del constructor, esto lo hago cambiándolo a private, hacemos lo propio con los Ids.

```
public string UserId { get; private set; }

public int NotificationId { get; private set; }

public ApplicationUser User { get; private set; }

public Notification Notification { get; private set; }

protected UserNotification()
{
}
}
```

Con estos simples cambios nos aseguramos de que nuestra aplicación siempre este en un estado valido. Ahora nuestra instancia anterior en el método Notify va a dar error por lo que dentro de la misma instancia debemos pasarle como parámetros lo que antes enviábamos por construcción de clase.

```
var userNotification = new UserNotification (this, notification)
{
    User = this,
    Notification = notification
};
```

O también podemos saltarnos la inicialización de una variable que no vamos a usar para nada. Por lo que pasamos directamente la instancia a la colección de UserNotification.

```
UserNotifications.Add(new UserNotification(this, notification));
```

Tenemos el mismo problema en la clase de notificación, en este podemos enviar un evento vacío, pero se supone que no

deberíamos, por lo que debemos hacer lo mismo que acabamos de hacer, solo muestro el código porque la explicación es la misma.

```
public DateTime DateTime { get; private set; }
public NotificationType NotificationType { get; private set; }
public DateTime? OriginalDateTime { get; set; }
public string OriginalVenue { get; set; }

[Required]
public Event Event { get; private set; }

protected Notification()
{
}

public Notification(NotificationType notificationType, Event ev)
{
    Event = ev ?? throw new ArgumentNullException(nameof(ev));
    NotificationType = notificationType;
    DateTime = DateTime.Now;
}
```

La fecha la inicializo en el constructor porque nadie debe tener la posibilidad de modificar la fecha.

```
public IActionResult Cancel(int id)
{
    ...
    var notification = new Notification (NotificationType.EventCanceled, ev)
    {
        DateTime = DateTime.Now,
        Event = ev,
        NotificationType = NotificationType.EventCanceled
    };
}
```

Refactorizando las llamadas a la base de datos

Volvamos a nuestra acción cancel, podemos ver que tenemos dos llamadas a la base de datos, una para listar todos los asistentes al evento y otra para obtener el evento que va a ser cancelado.

Una mejor forma de evitar sobrecarga es obtener el evento y todos sus asistentes, de un fuetazo. Lo primero que vamos a necesitar es crear la propiedad de navegación en la clase Event que nos permita almacenarla

```
public ICollection<Attendance> Attendances { get; private set; }

public Event()
{
    Attendances = new Collection<Attendance>();
}
```

Le pongo el set privado para que nadie pueda cambiar la colección por accidente, ahora puedo simplificar mis dos llamadas y combinarlas en un solo script

```
var ev = _context.Events
    .Include(p=> p.Attendances.Select(a=> a.Attendee))
    .Single(a => a.CoachId == userId && a.Id == id);
```

Principio de Cohesión

Este principio nos dice que todo lo que esta relacionado, debe estar cerca o junto, no debes y no puedes separarlos, si leemos la historia que nos cuenta la acción cancelar, podemos ver que esta busca el evento dado un id, si no lo encuentra retorna un notfound, posterior a eso cancela el evento y envía una notificación a cada participante y finalmente guarda los cambios.

Eso quiere decir que el cambiar el estado a cancelado y la creación de la notificación, están enteramente correlacionadas, no pueden convivir la una sin la otra, cuando cancelamos un evento siempre vamos a vernos obligados a enviar una notificación, a todos los participantes, además, todas estas líneas cambian el estado de

nuestro dominio, el controlador no debe verse involucrado en detalle en cambiar o el estado del mismo, el solo debe delegarlo, asi que creamos el método Cancel dentro de la clase Event (porque es a el a quien le ejecutamos la acción de cancelar, es decir, el es el que cancelamos cuando marcamos Cancel), cortamos desde el ev.Cancel hasta el cuerpo del foreach y nos lo llevamos para este método. Posterior a eso lo acondicionamos aca cabe destacar que el Context no es necesario porque la lista a iterar ya la tengo incluida en el this solo tengo que iterarla sobre ella.

```
public void Cancel()
{
    IsCanceled = true;
    var notification = new Notification(NotificationType.EventCanceled, this);

    foreach (var attendee in this.Attendances.Select(a => a.Attendee))
    {
        attendee.Notify(notification);
    }
}
```

Si ya creamos un método u operación dentro de nuestro dominio para cancelar entonces no debemos tener la posibilidad de crear una instancia de la clase Event y modificar desde fuera la propiedad IsCanceled, porque esto pondría nuestro modelo en un estado invalido, por lo que ponemos privado el setter de la propiedad isCanceled.

Aún tenemos unas líneas que hacen query y queriar no es responsabilidad del controlador, pero eso es algo que veremos mas adelante cuando trabajemos con el patrón repositorio.

```
public bool IsCanceled { get; private set; }
```

Relación Inversa

Si corremos la aplicación podemos ver un error, porque EF piensa que nuestro modelo ha cambiado, de igual forma si corremos una migración podremos ver supuestos cambios, sin embargo no es así, esto se debe a que al agregar algunas propiedades identificadoras de navegación (como el EventId) EF entiende algunas veces que son propiedades perse de la clase en cuestión.

Normalmente el resolvería esto por nosotros, pero lo que pasa es que si nos vamos a la configuración de nuestro DataContext podemos ver un fallo donde nosotros le decimos que ignore su configuración por defecto (configuration over convention) nosotros decimos que hay una relación de uno a mucho pero no especificamos quien es el mucho, porque al momento de crear esta relación no necesitábamos esa propiedad de navegación que con el tiempo si necesitamos, recomendación, siempre cree las propiedades de navegación de ambos lados aunque no las vaya a usar y las configuraciones hágalas completas aun cuando creas que solo necesitas una parte de la misma. De paso cuando trabajas de esta forma, le indicas a tus evaluadores que comprendes muy bien el concepto de entidad relación en el modelo relacional, es decir, siempre desde que inicies la construcción de tu clase, en todo momento debes armar la relación, tanto del lado mucho a uno (Type en la clase Event) como del lado uno a mucho (Collection<Event> en la clase Type), podemos eliminar la migración, correrla de nuevo y ver que ahora saldrá limpia.

```
builder.Entity<Attendance>()  
    .HasOne(a => a.Event)  
    .WithMany(e => e.Attendances)  
    .HasForeignKey(a => a.EventId).OnDelete(DeleteBehavior.Restrict);
```



```
builder.Entity<UserNotification>()
    .HasOne(a => a.User)
    .WithMany(u => u.UserNotifications)
    .HasForeignKey(a => a.UserId).OnDelete(DeleteBehavior.Restrict);
```

Ahora podemos correr la aplicación y ver que todo está nice sin problemas. Ahora si vamos a probar el cancelar:

1. Creamos dos eventos.
2. Con otro usuario marcamos que deseamos asistir a esos eventos que creó el fulano anterior.
3. Entramos como el dueño del evento y cancelamos estos.
4. Revisamos la base de datos y debemos ver que hay dos notificaciones.

Ahora vamos a crear una interfaz para estos fines.

Sin embargo, antes vamos a crear el módulo de notificaciones cuando un evento es actualizado, esto deberías intentarlo por tu propia cuenta antes de ver la solución. (Igual te la dejo porque se que no lo vas a hacer)

```
public ActionResult Update(EventViewModel vm)
{
    ...
    var ev = _context.Events
        .Include(g => g.Attendances.Select( a=> a.Attendee))
        .Single(g => g.Id == vm.Id && g.CoachId == userId);

    ev.Modify(vm.GetDateTime(), vm.Venue, vm.TypeId);

    ev.DateTime = vm.GetDateTime();
    ev.TypeId = vm.TypeId;
    ev.Venue = vm.Venue;
}

public void Modify(DateTime dateTime, string venue, byte typeId)
{
    var notification = new Notification(NotificationType.EventUpdated, this);
    notification.OriginalDateTime = DateTime;
    notification.OriginalVenue = Venue;

    DateTime = dateTime;
    TypeId = typeId;
```

```

Venue = venue;

foreach (var attendee in Attendances.Select(a => a.Attendee))
    attendee.Notify(notification);
}

```

Factory Methods para crear un objeto.

Previamente habíamos visto que debemos evitar que nuestros objetos caigan en un estado invalido, sin embargo nosotros tenemos una violación de ese principio en el Modify de Event

```

notification.OriginalDateTime = DateTime;
notification.OriginalVenue = Venue;

```

Nosotros podemos por accidente olvidar que debemos inicializar esas dos propiedades, para esto debemos recurrir a un factory method, este es el responsable de crear objetos en un estado valido, lo primero que voy a hacer es poner privado este constructor, porque no quiero que haya posibilidad de crear una instancia invalida de este.

```

private Notification(NotificationType notificationType, Event ev)

```

Luego voy a crear un método estático factory. Este método será responsable de crear una instancia de notificación cuando un evento sea creado, repetimos el proceso para cuando sea actualizado o cancelado. Después de eso ponemos privadas las propiedades que no deben ser modificadas desde fuera.

```

public int Id { get; private set; }
...
public DateTime? OriginalDateTime { get; private set; }
public string OriginalVenue { get; private set; }

public static Notification EventCreted(Event ev)
{

```

```

        return new Notification(NotificationType.EventCreated, ev);
    }

    public static Notification EventUpdated(Event newEvent, DateTime
originalDateTime, string originalVenue)
    {
        var notification = new Notification(NotificationType.EventUpdated, newEvent);
        notification.OriginalDateTime = originalDateTime;
        notification.OriginalVenue = originalVenue;

        return notification;
    }

    public static Notification EventCanceled(Event ev)
    {
        return new Notification(NotificationType.EventCanceled, ev);
    }

```

Si vemos ahora el método Notify y el Cancel de la clase Event da error, porque ya no podemos hacer instancia de la misma de forma directa una está protegida (la default) para que EF no se quille y la otra es privada que solo la puede acceder la misma clase, por lo que debemos usar los métodos disponibles para accederla.

```

public void Cancel()
{
    ...
    var notification = Notification.EventCanceled(this);
    var notification = new Notification(NotificationType.EventCanceled, this);
    ...}

public void Modify(DateTime dateTime, string venue, byte typeId)
{
    ...
    var notification = Notification.EventUpdated(this, DateTime, Venue);
    var notification = new Notification(NotificationType.EventUpdated, this);
    notification.OriginalDateTime = DateTime;
    notification.OriginalVenue = Venue;
    ...}

```

Resumen

En este módulo mejoramos la arquitectura de nuestro sistema haciendo una refactorización paso a paso, iniciamos con un modelo de dominio anémico y poco a poco lo fuimos transformando en algo realmente de utilidad, sin embargo hay mucho debate entre que el dominio debe ser integro y no se debe tocar, yo soy de los que me inclino por esa corriente, pero si la lógica sobre tu dominio crece vas a tener que delegarla a alguien más y en estos casos es más conveniente delegar esas funciones a su responsable (yo lo delego en repositorios pero para esa clase, falta :D).

Vimos unos de los principios fundamentales de la orientación a objetos y que es uno de los principios más violados por los desarrolladores **nuestro objeto “¡siempre!” debe estar en un estado valido** para esto nos aseguramos de que el dominio proteja sus propiedades y sea el encargado de forzar a que siempre al llamarla usen instancias válidas.

Mapeando los Objetos de Dominio a Objetos de Transferencia de Datos (DTO's)

En este módulo vamos a crear un api que nos va a traer las notificaciones del usuario, al momento de usar Apis para traer datos nos vamos a topar con la necesidad de usar un dto y por fin vamos a explorar estos a profundidad.

Construyendo el Api de notificaciones

Acá no hay nada que no hayamos visto, solo necesitamos crear un api que sea llamado con jquery y traiga un listado de las notificaciones de un usuario nada del otro mundo.

API:

```
public class NotificationsController:Controller
{
    private readonly DataContext _context;

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

    public IEnumerable<Notification> GetNewNotifications()
    {
        var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);
        var notifications = _context.UserNotifications
            .Where(un => un.UserId == userId && !un.IsRead)
            .Select(un => un.Notification)
            .Include(n => n.Event.Coach)
            .ToList();

        return notifications;
    }
}
```

Si nosotros probamos esto en Postman podemos ver algo sorprendente y es que estamos exponiendo demasiada información.

```
POST https://localhost:44344/Notifications/GetNewNotifications Send Save

Pretty Raw Preview JSON

{
  "id": 1,
  "coachId": "d1c08414-a719-492f-829b-6883a8408449",
  "coach": {
    "name": "Juan Almonte",
    "followers": [],
    "followees": [],
    "userNotifications": [],
    "attendances": [],
    "id": "d1c08414-a719-492f-829b-6883a8408449",
    "userName": "sgermosen@outlook.com",
    "normalizedUserName": "SGERMOSEN@OUTLOOK.COM",
    "email": "sgermosen@outlook.com",
    "normalizedEmail": "SGERMOSEN@OUTLOOK.COM",
    "emailConfirmed": false,
    "passwordHash": "AQAAAAEAAcQAAAAED6/xBI+ISnvYHrSCAb/wb+wxv0Q/D4RjV4Y78VnAB86++QfKpaXNV0CqbzJXjEmbA==",
    "securityStamp": "YXB6W7JLANHPFXCJLGVH4TEXP7YWE6NE",
    "concurrencyStamp": "c103d386-9c20-4edf-a525-422a79883bf2",
    "phoneNumber": null,
    "phoneNumberConfirmed": false,
    "twoFactorEnabled": false,
    "lockoutEnd": null,
    "lockoutEnabled": true,
    "accessFailedCount": 0
  },
  "dateTime": "2020-12-12T12:12:00",
  "venue": "xxxxxx",
  "typeId": 2,
  "type": null,
  "isCanceled": false,
  "attendances": []
}
```

Podemos ver no solo la información del usuario logueado, sino que también podemos ver la información sensible del usuario que crea el evento y esto no es correcto.

Por temas de mejores prácticas no debemos exponer nunca nuestros dominios, sino dtos, para evitar este tipo de inconvenientes, por lo que vamos a crear en la carpeta DTOs un Dto para las notificaciones que llamaremos NotificationDto, donde copiaremos todas las propiedades que tiene el modelo y vamos eliminando las que no necesitamos. Los DTOs no llevan validaciones son solo para transferir datos, de igual forma nos

vemos obligados a crear un EventDto y a su vez un TypeDto y un UserDto.

```
public class NotificationDto
{
    public DateTime DateTime { get; set; }
    public NotificationType NotificationType { get; set; }
    public DateTime OriginalDateTime { get; set; }
    public string OriginalVenue { get; set; }
    public EventDto Event { get; set; }
}

public class EventDto
{
    public int Id { get; set; }
    public bool IsCanceled { get; set; }
    public UserDto Coach { get; set; }
    public DateTime DateTime { get; set; }
    public string Venue { get; set; }
    public TypeDto Type { get; set; }
}

public class UserDto
{
    public string Id { get; set; }
    public string Name { get; set; }
}

public class TypeDto
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

Ahora nos toca hacer la transformación de un objeto al otro.

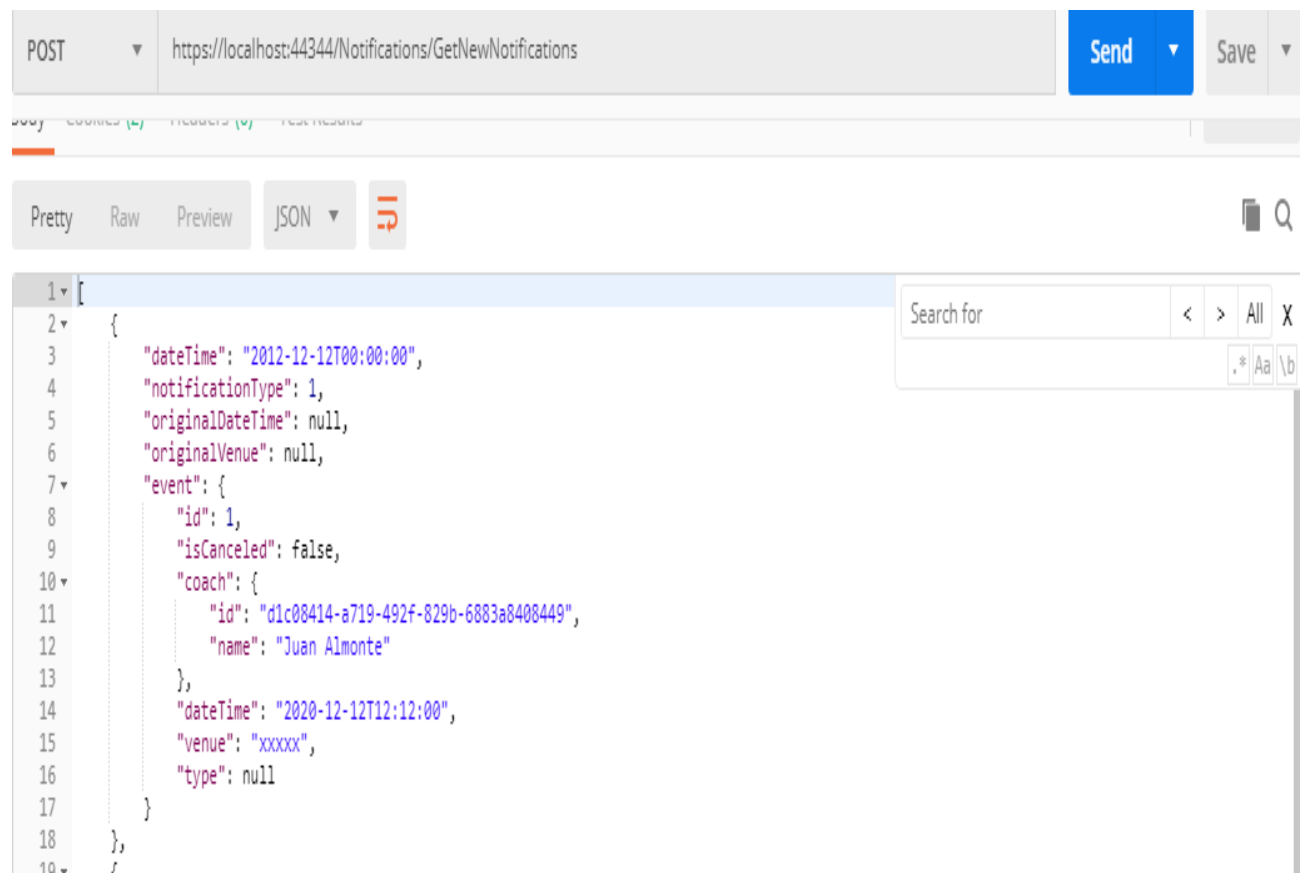
```
public IEnumerable<NotificationDto> GetNewNotifications()
{
    ...
    return notifications.Select(n => new NotificationDto()
    {
        DateTime = n.DateTime,
        Event = new EventDto
        {
            Coach = new UserDto
            {
                Id = n.Event.Coach.Id,
                Name = n.Event.Coach.Name
            },
            DateTime = n.Event.DateTime,
            Id = n.Event.Id,
            IsCanceled = n.Event.IsCanceled,
```

```

        Venue = n.Event.Venue,
    },
    OriginalDateTime = n.OriginalDateTime,
    OriginalVenue = n.OriginalVenue,
    NotificationType = n.NotificationType
});
}

```

Corremos la aplicación y vemos que si probamos ahora con PostMan no estamos exponiendo tanta información además de que la carga es más ligera.



Auto Mapper

Este es el mapeo del modelo que tenemos hasta el momento, este código es demasiado ruidoso y redundante, Automapper es una herramienta que usa reflexión para machear las propiedades que se llamen similares lo instalamos desde el Nugget:


```
install-package AutoMapper
```

Antes de indicarle a auto mapper que mapee dos clases tenemos que crear un mapeo entre estas. Este método es genérico y recibe dos parámetros, la fuente y el destino.

Clasic

```
Mapper.CreateMap<ApplicationUser, UserDto>();
Mapper.CreateMap<Event, EventDto>();
Mapper.CreateMap<Notification, NotificationDto>();

return notifications.Select(Mapper.Map<Notification, NotificationDto>());
```

Net Core

```
var config = new MapperConfiguration(cfg => {
    cfg.CreateMap<ApplicationUser, UserDto>();
    cfg.CreateMap<Event, EventDto>();
    cfg.CreateMap<Notification, NotificationDto>();
});
IMapper IMapper = config.CreateMapper();

return notifications.Select(IMapper.Map<Notification, NotificationDto>());
```

Podemos probar y ver que sigue todo funcionando igual. (debería), para que las cosas se vean más profesional, podemos sacar eso a un archivo de configuración, creamos la carpeta Config y creamos una clase MappingProfile que hereda de Profile y en su constructor pegamos parte de la configuración

Clasic

```
public class MappingProfile : Profile
{
    public MappingProfile()
    {
        var config = new MapperConfiguration(cfg =>
        {
            cfg.CreateMap<ApplicationUser, UserDto>();
            cfg.CreateMap<Event, EventDto>();
            cfg.CreateMap<Notification, NotificationDto>();
        });

        config.CreateMapper();
    }
}
```

ApplicationStar

```
Mapper.Initialize(c => c.AddProfile<MappingProfile>());
```

NetCore

```
public class MappingProfile : Profile
{
    public MappingProfile()
    {
        CreateMap<ApplicationUser, UserDto>().ReverseMap();
        CreateMap<Event, EventDto>().ReverseMap();
        CreateMap<Notification, NotificationDto>().ReverseMap();
    }
}
```

Startup.cs

```
var configMapper = new AutoMapper.MapperConfiguration(cfg =>
{
    cfg.AddProfile(new MappingProfile());
});
var mapper = configMapper.CreateMapper();
services.AddSingleton(mapper);
```

Posterior a eso hacemos los siguientes cambios en nuestro Api

```
private readonly IMapper _mapper;

public NotificationsController(DataContext context, IMapper mapper)
{
    _context = context;
    _mapper = mapper;
}

public IEnumerable<NotificationDto> GetNewNotifications()
{
    ...
    return notifications.Select(_mapper.Map<Notification, NotificationDto>);
}
```

Camel Notation

Si echamos un vistazo a la respuesta del API, podemos ver que esta esta tiene la primera letra de la propiedad en mayúscula, el cual es el estándar en c#, sin embargo, en la web esta notación no

es muy bien vista, por lo que tenemos que configurar nuestro api para que retorne los nombres de las propiedades en camel case.

Pero en netCore ya las retorna como las queremos, por lo que debería eliminar este tópico, nada, después actualizo la de classic o veré que hago.

Resumen

Lo primero que vimos son los contratos de un api, asegúrate de que tu api recibe y envía DTOs nunca tu objeto de dominio, puedes cambiar tu dominio, modificarlo y estos cambios nunca deben impactar tu contrato con el cliente web, también te evita exponer data sensible. Aprendimos a mapear después les muestro como hacerlo de una forma más chula pero como es algo que se está trabajando lo guardaré para después.

Trabajando en la interfaz de usuario de nuevo

Lo primero que deseamos es agregar un icono de notificación al lado de nuestro username, esto lo hacemos con glyphicon el cual bajamos e integramos.

```
<a>  
  <i class="glyphicon glyphicon-globe"></i>  
</a>
```

Luego deseo saber el número de notificaciones que tengo sin leer, para eso creo esto

```
<a>  
  <i class="glyphicon glyphicon-globe"></i>  
  <span class="badge"> 2</span>
```

```
</a>
```

Por ahora voy a hardcodear el número pero posteriormente lo voy a reemplazar por la cantidad real que venga desde la base de datos, al elemento padre le voy a añadir un identificador para poder modificarlo por css

```
<li class="nav-item notifications">
  <a>
    <i class="glyphicon glyphicon-globe"></i>
    <span class="badge"> 2</span>
  </a>
</li>
```

Agregamos un par de configuraciones de css

```
.nav-item.notifications {
  position: relative;
}

.nav-item .notification .badge {
  position: absolute;
  top: 8px;
  right: -1px;
  background: #a80a0a;
}
```

Después agregamos un identificador al badge para mostrar el numero de notificaciones y le agrego la clase hide para que por defecto no se muestren luego agregamos el respectivo js.

```
<li class="nav-item notifications">
  <a>
    <i class="glyphicon glyphicon-globe"></i>
    <span class="badge js-notifications-count hide"> </span>
  </a>
</li>
```

```
@section scripts
{
  <script>
    $(document).ready(function () {
      $.getJSON("/api/notifications", function (notifications) {
        $(".js-notifications-count")
          .text(notifications.length)
          .removeClass('hide');
      });
    });
  </script>
}
```

Lo de arriba funciona, pero queremos que se vea mucho mas elegante y nos muestre las notificaciones en un popover para eso vamos a interceptar.

```
$.getJSON("/api/notifications", function (notifications) {  
    $(".js-notifications-count")  
        .text(notifications.length)  
        .removeClass('hide')  
        .addClass("animated bounceInDown");  
});  
$(".notifications").popover({  
    html: true,  
    title: "Notifications",  
    content: function () {  
        return "Hellow this is a pop over";  
    },  
    placement: "bottom"  
});
```

Underscore

Para mostrar las notificaciones (las cuales tenemos que mostrarlas en donde actualmente se muestra el hellow..., tenemos dos opciones, una es podemos hacerla de esta forma.

```
$(".notifications").popover({  
    html: true,  
    title: "Notifications",  
    content: function () {  
        return "<ul>"  
            + "<li>" + .. + "</li>";  
    },  
    placement: "bottom"  
});
```

Sin embargo haciéndolo de esta forma no estamos haciendo separación de conceptos y escribir html en javascripts es horrible, la mejor forma de hacer esto es usando plantilla similares a lo que hace razor, que tenemos el contenido en un lugar y luego le decimos al engine que renderice el contenido basado en un objeto json, igual a como renderizamos views en asp mvc, hay diferentes

engine de plantillas, pero me voy a concentrar en uno que es para mí el más simple y sencillo (mentira, no he usado otro para estos motivos) lo bajamos y añadimos la referencia.

Creemos la plantilla y en la plantilla se va a reemplazar el valor del placeholder con el ítem que le indiquemos

```
$(".notifications").popover({
  html: true,
  title: "Notifications",
  content: function () {
    var compiled = _.template("Hello <%= name %>");
    var html = compiled({ name: "sGermosen" });
    return html;
  },
  placement: "bottom"
});
```

Pero aun esta forma de trabajo no obedece a las mejores practicas por lo que nos vemos obligados a crear una plantilla que es la que vamos a renderizar cuando le pasemos el json.

```
<script type="text/x-template" id="notifications-template">
  <ul>
    <%
      _.each(notifications, function(notification){
        if (notification.type == 1 { %>
        <li> Coach has canceled the Event at venue at datetime.</li>
        <% }
      })
    %>
  </ul>
</script>
```

Bien, dentro de las mejoras algo que quiero hacer es remover los bullets que tiene el popover por lo que voy a agregarle una clase al ul donde enmascaro todo y despues reemplazo los textos por los parámetros y luego voy al site.css para removerlos con css, de igual forma quitaremos las líneas separadoras por algo más estilizado y la última línea divisora cuando mostramos el último elemento de la lista la vamos a remover.

```

<ul class="notifications">
  <%
    _.each(notifications, function(notification){
      if (notification.type == 1 { %>
        <li> <%= notification.event.coach.name %> has canceled the Event at <%=
notification.event.venue %> at <%= notification.event.dateTime %>.</li>
      <% }
    })
  %>
</ul>

```

```

ul.notifications {
  list-style: none;
  padding-left: 0;
}

ul.notifications > li {
  border-bottom: 1px solid #ddd;
  padding: 10px 0;
}

ul.notifications > li:last-child {
  border-bottom: none;
}

```

Además de completar la plantilla en el escenario de que actualicen un evento o de que lo creen. Para el caso del update, voy a crear un array para saber cuales han sido los cambios y poder indicarle al usuario que fue lo que cambio, si el venue, la fecha o ambas.

```

<ul class="notifications">
  <%
    _.each(notifications, function(notification){
      if (notification.notificationType == 1) { %>
        <li><%= notification.event.coach.name %> has canceled the event at <%=
notification.event.venue %> at <%= notification.event.dateTime %>.</li>
      <% }

      else if (notification.notificationType == 2) {
        var changes = [],
            originalValues = [],
            newValues = [];

        if (notification.originalVenue != notification.event.venue) {
          changes.push('venue');
          originalValues.push(notification.originalVenue);
          newValues.push(notification.event.venue);
        }

        if (notification.originalDateTime != notification.event.dateTime) {
          changes.push('date/time');
          originalValues.push(notification.originalDateTime);

```

```

        newValues.push(notification.event.dateTime);
    }

    %>
    <li><%= notification.event.coach.name %> has changed the <%= changes.join('
and ') %> of the event from <%= originalValues.join('/') %> to <%= newValues.join('/')
%></li>

    <%
    }
    })
    %>
</ul>

```

Otra cosa es que debemos cambiar a negrita el nombre de quien hace el cambio el evento, una forma seria añadiendo el tag ``.

```

<li> <strong> <%= notification.event.coach.name %> </strong> has canceled the event at
<%= notification.event.venue %> at <%= notification.event.dateTime %>.</li>

```

Pero esta es una forma pobre de hacerlo. Lo mas correcto es añadir un span o div o lo que sea, con una clase que vamos a modificar por css, de paso aprovechamos y agregamos unos identificadores a los css que agregamos recientemente, debido a que yo no quiero que me le agregue esos estilos a todos los ul o li que cumplan esa condición, sino solo a los de pop over.

```

<li> <span class="highlight"> <%= notification.event.coach.name %> </span> has canceled
the event at <%= notification.event.venue %> at <%= notification.event.dateTime %>.</li>

```

```

.popover-notifications ul.notifications {
    list-style: none;
    padding-left: 0;
}

.popover-notifications ul.notifications > li {
    border-bottom: 1px solid #ddd;
    padding: 10px 0;
}

.popover-notifications ul.notifications > li:last-child {
    border-bottom: none;
}

.popover-notifications ul.notifications .highlight {
    font-weight: bold;
}

```


Moment.js

Siempre que queremos mostrar la fecha no podemos hacerlo con el formato por defecto porque eso se ve horrible, moment es un archivo que le busca la vuelta a lo que sea que reciba que le digan que es una fecha y la muestra en el formato que se le indique de una forma bien eficiente, así que lo bajamos integramos y reemplazamos donde mostramos fecha con el formato que deseemos.

```
<li><%= notification.event.coach.name %> has canceled the event at <%=  
notification.event.venue %> at <%= moment(notification.event.dateTime).format("D MMM  
HH:mm") %>.</li>
```

Otra cosa a hacer es remover la sombra del pop over.

```
.popover {  
  box-shadow: none;  
}
```

Less

Bien, hasta el momento hemos trabajado mucho con css de forma directa, sin embargo, en el mundo real esto se vuelve algo inmantenible, por lo que se creó less



Less es un lenguaje de estilos dinámicos que se compila en un archivo de css. Con less se escribe **less code**, chiste malo que solo tiene sentido en inglés :’D (hello darkness your my friend).

```

.popover-notifications ul.notifications {
  list-style: none;
  padding-left: 0;
}

.popover-notifications ul.notifications > li {
  border-bottom: 1px solid #ddd;
  padding: 10px;
}

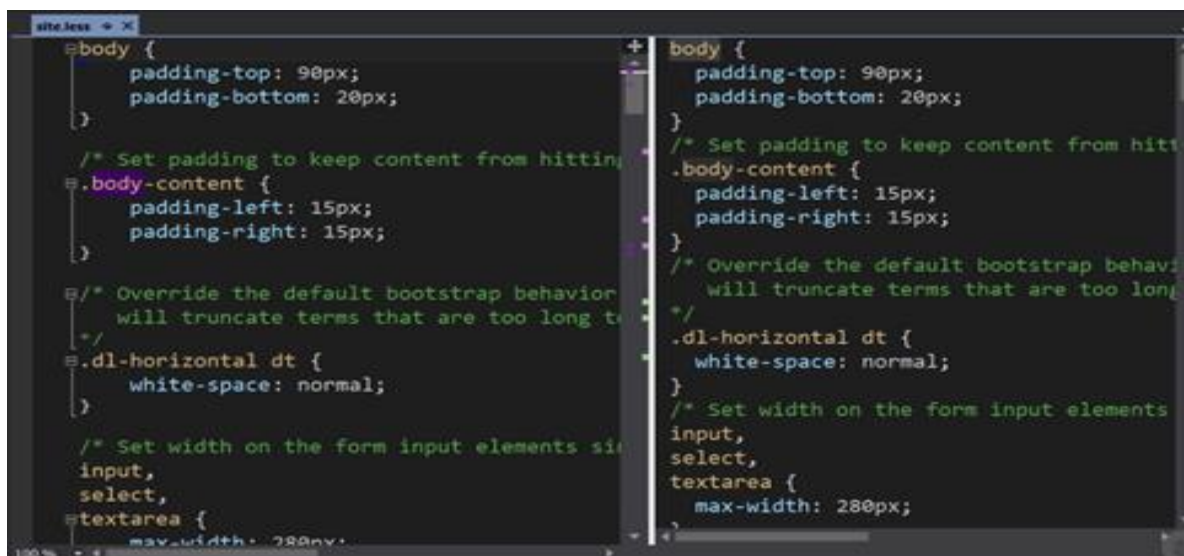
.popover-notifications ul.notifications > li:last-child {
  border-bottom: none;
}

.popover-notifications ul.notifications .highlight {
  font-weight: bold;
}

```

Por ejemplo, nosotros hemos repetido 4 veces esta misma línea por lo que vamos a hacer uso de la lessificación.

Cerramos el archivo site.css, Modificamos el nombre a site.less y lo abrimos y podemos ver una pantalla como esta.



```

site.less
body {
  padding-top: 90px;
  padding-bottom: 20px;
}

/* Set padding to keep content from hitting edges */
body-content {
  padding-left: 15px;
  padding-right: 15px;
}

/* Override the default bootstrap behavior which truncates terms that are too long to fit
dl-horizontal dt {
  white-space: normal;
}

/* Set width on the form input elements since the bootstrap default is 100%
input,
select,
textarea {
  max-width: 280px;
}

```

```

body {
  padding-top: 90px;
  padding-bottom: 20px;
}

body-content {
  padding-left: 15px;
  padding-right: 15px;
}

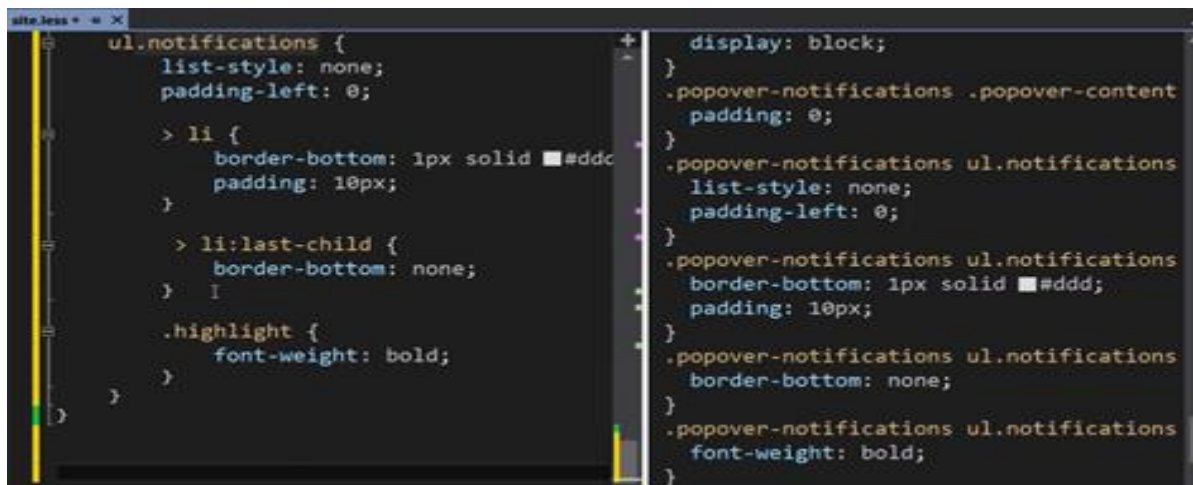
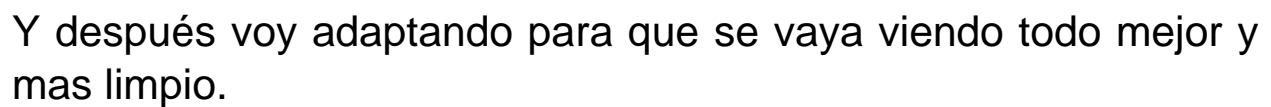
dl-horizontal dt {
  white-space: normal;
}

input,
select,
textarea {
  max-width: 280px;
}

```

En la izquierda tenemos less y en la derecha tenemos el compilado en css. Actualmente lucen idénticos, con less nosotros podemos escribir nested styles, lo que no puede ser con css

Por lo que puedo indicar que la clase popover-notifications va a contener dentro otra clase con las características de la otra.



Pero, en netcore eso no pasa, porque por alguna extraña razón que desconozco el webessential no activa esa funcionalidad :(así que ignoremos que eso existe en core.

Cuando trabajamos con proyectos legacy es muy difícil transformar todo nuestro css a less de forma manual, así que podemos usar un convertidor online que haga eso por nosotros

<https://jsonformatter.org/css-to-less>

Interceptando eventos

Ahora necesitamos que cuando el usuario haga clic en el icono de notificación el contador de notificaciones desaparezca, para esto necesitamos interceptar el evento del pop over que es un elemento de bootstrap, me interesa de manera particular el evento show del popover en la documentación nos dice que esta es la manera de hacerlo.

```
$(".notifications").popover({
  html: true,
  title: "Notifications",
  content: function () {
    var compiled = _.template($("#notifications-template").html());
    return compiled({ notifications: notifications });
  },
  placement: "bottom",
  template: '<div class="popover popover-notifications" role="tooltip"><div
class="arrow"></div><h3 class="popover-title"></h3><div class="popover-
content"></div></div>'
}).on("shown.bs.popover", function () {
  console.log("this is a indicator than the popover was shown");
});
```

Creamos el api

```
[HttpPost]
public IActionResult MarkAsRead()
{
    var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);
    var notifications = _context.UserNotifications
        .Where(un => un.UserId == userId && !un.IsRead)
        .ToList();
    notifications.ForEach(n => n.Read());
    _context.SaveChanges();
    return Ok();
}
```

A pesar de que estoy actualizando información, para los fines semánticos nosotros estamos realizando una nueva petición, para marcar las **nuevas notificaciones** del usuario como leídas, por ende es una nueva petición es un nuevo recurso, por eso uso post en lugar de Put que sería lo “correcto” dentro de la óptica de muchos, debido a que estoy actualizando.

En el UserNotification creamos el método para marcar como leídas las notificaciones y hacemos privado el setter de la propiedad.

```
public bool IsRead { get; private set; }

public void Read()
{
    IsRead = true;
}
```

Y finalmente la llamamos desde jquery

```
$(".notifications").popover({
    html: true,
    title: "Notifications",
    content: function () {
        var compiled = _.template($("#notifications-template").html());
        return compiled({ notifications: notifications });
    },
    placement: "bottom",
    template: '<div class="popover popover-notifications" role="tooltip"><div class="arrow"></div><h3 class="popover-title"></h3><div class="popover-content"></div></div>'
}).on("shown.bs.popover", function () {
    console.log("this is a indicator than the popover was shown");
});
```

```

        $.post("/api/notifications/markAsRead")
        .done(function () {
            $(".js-notifications-count")
            .text("")
            .addClass("hide");
        });
    });

```

Hay un bug o un feature, **los bugs nunca son errores, son funcionalidades no programadas del sistema**, que les voy a dejar como tarea y que no vamos a corregir y solo quien lo corrija va a poder recibir el diploma, es que nosotros cuando marcamos como leídas las notificaciones ya no se muestran más, así que eso se les queda como tarea resolverlo.

Implementando la búsqueda

Aquí no hay mucho que agregar mas que iniciar a meter mano, creamos en nuestro EventsViewModel una propiedad llamada SearchTerm que luego vamos a agregar en nuestro formulario de Events, dentro de un form-group que va a estar nestado en un elemento form, posterior a eso crearemos un botón que es el que se va a encargar de hacer submit

```

public string SearchTerm { get; set; }

<form asp-action="Search">
    <div class="form-group">
        <input asp-for="SearchTerm"
            class="form-control"
            placeholder="Enter a search term by Venue or Coach"/>
    </div>
    <input type="submit" value="Search" />
</form>

```

Después de esto le ponemos bonitura como un iconito y un estilito chulo.

```
<form asp-action="Search" asp-controller="Events">
  <div class="form-group">
    <div id="searchEvents" class="input-group">
      <input asp-for="SearchTerm"
        class="form-control"
        placeholder="Enter a search term by Venue or Coach" />
      <span class="input-group-addon">
        <i class="glyphicon glyphicon-search"></i>
      </span>
    </div>
  </div>

  <input type="submit" value="Search" />
</form>
```

Si se nos descuadra el icono de la lupa es porque tenemos que ponerle un tamaño a elemento de input-group en el site.css

```
.searchEvents{
  width:300px
}
```

Query strings

Si yo quiero implementar este campo de búsqueda yo puedo filtrar mi búsqueda dentro del controlador y ya, pero el problema es que si voy a ver el detalle y luego vuelvo al index el filtro va a desaparecer.

Para evitar eso debemos hacer uso de los query strings, que son aquellos que se marcan en la url como criterio de la misma, para esto primero creamos la acción Search y la ponemos a redirigir hacia el home con el parámetro que le haya sido enviado, posterior a eso creamos la variable query de tipo string y que acepte nulos en nuestra acción index del home donde validamos y filtramos si es necesario.


```

public IActionResult Search(EventsViewModel vm)
{
    return RedirectToAction("Index", "Home", new { query = vm.SearchTerm });
}

public IActionResult Index(string query = null)
{
    var upcomingEvents = _context.Events
        .Include(c => c.Coach)
        .Include(c => c.Type)
        .Where(p => p.DateTime > DateTime.Now && !p.IsCanceled);

    if (!String.IsNullOrEmpty(query))
    {
        upcomingEvents = upcomingEvents.Where(g =>
            g.Coach.Name.Contains(query) ||
            g.Type.Name.Contains(query) ||
            g.Venue.Contains(query));
    }
    var vm = new EventsViewModel
    {
        UpcomingEvents = upcomingEvents,
        ShowActions = User.Identity.IsAuthenticated,
        Heading = "Upcoming Events ", //My upcoming events on the other screen
        SearchTerm = query
    };
    ...}

```

Resumen

En este módulo trabajamos en la interfaz de usuario para mostrarle las notificaciones al mismo, exploramos una manera distinta de renderizar contenido.

Normalmente renderizamos el contenido en el server y se lo servimos en bandeja de plata todo procesado al cliente, ahora aprendimos a renderizar contenido en el cliente, que es una corriente que ha ganado muchos adeptos y fama con frameworks como React y Angular, donde la premisa es poner al cliente a consumir sus recursos.

Índice

Introducción.....	3
Sobre el autor.	3
¿Qué Vamos a Ver?	4
Prerrequisitos.....	6
¿Qué aprenderemos?	8
Estructura del Curso.....	9
Fundamentos.....	9
Tópicos Avanzados.....	10
Arquitectura.....	10
¡¡¡¡Vamos a iniciar!!!!	10
Que es .Net Core	11
Sistema de Control de Versiones	14
GitHub.....	15
File New Project por fin!!!!	17
Resumen	19
Extrayendo los principales casos de uso del requerimiento.	20
Autenticación.....	22
Eventos.....	23
Calendario	24
Seguimiento	25
Dependencias.....	26
Iniciando las iteraciones.....	30
MVC.....	33
Un repaso breve de programación orientada a objetos	34
Base de Datos 101.....	36
Entity Framework Code First.....	37
Trabajando con nuestro primer modelo.....	39
Sobrescribiendo las convenciones.....	49
Decisiones de diseño.	50
Resumen.	52

Construyendo un Formulario	53
Mi Primer Controlador	54
Mi Primera Vista	55
Bootstrap 100.5 (Porque no llegamos a 101, xD)	57
View Models	60
ViewBag y ViewData	63
Submit	68
Resumen	69
Guardando Información en la base de datos.....	70
Autorización de acceso.	70
Insertando data.....	71
Separación de Conceptos.	78
Resumen	80
Validaciones.....	81
Validaciones en el servidor.....	81
Validaciones personalizadas.....	85
Validaciones en el lado del cliente.	89
Resumen	92
Previniendo ataques comunes en la web.....	93
SQL Injection.....	93
Cross-Site Scripting (XSS)	95
Cross-Site Request Forgery (CSRF).....	97
Resumen.	101
Pimpeando-Enchulando la máquina, digo, el proyecto.....	102
Escogiendo un color	102
Sobre escribiendo los estilos de Bootstrap.....	105
Escogiendo una tipografía (Fonts)	108
Mejorando la apariencia del formulario	111
Drop Down List en la barra de navegación.	114
Resumen	117
Mejores prácticas de usabilidad.	119
Etiquetas (labels).....	119
Campos	126

Acciones	128
Validaciones	131
Resumen	133
Extendiendo Asp Identity User	134
Construyendo una vista básica	134
Extendiendo la clase ApplicationUser.....	136
Extendiendo el formulario de registro.....	138
Resumen	139
Creando un diseño más chulo y bonito con CSS	140
Maquetado.....	140
Zencoding.....	141
Posicionamiento absoluto y relativo.....	144
Trabajando con atributos CSS.	147
Resumen	151
Implementando los casos de uso restantes.....	152
Diseño pobre :(.....	152
Extendiendo el modelo de Dominio	157
Sobre escribiendo convenciones usando Fluent-API.....	159
Diseñando y creando un API	163
Metiendo mano con PostMan	166
Implementando el botón	168
JavaScript 99	169
DTOs.....	172
Resumen	173
Implementando los casos de uso secundarios	177
Ocultando acciones de usuarios anónimos	179
Batches en los requerimientos.....	181
Implementado los casos de soporte	182
Partial View	183
Refactorizando usando el principio DRY	185
Resumen	186
Fin.....	187
RUD... la C la hicimos hace rato	188

Read y el Dilema de la reutilización (reutilizar o no reutilizar)	188
Update	191
Evitando los Magic Strings	197
Delete	199
Evitando Trampas en el modelo de dominio	202
Resumen	207
Refactorizando para conseguir un diseño orientado a objetos.....	207
Quitando responsabilidades al controlador	208
Evitando los Estados Invalidos del objeto.....	210
Refactorizando las llamadas a la base de datos.....	213
Principio de Cohesión	214
Relación Inversa	216
Factory Methods para crear un objeto.....	218
Resumen	220
Mapeando los Objetos de Dominio a Objetos de Transferencia de Datos (DTO's).....	220
Construyendo el Api de notificaciones	221
Auto Mapper (spoiler, hay uno mas duro hecho por un dominicano)	224
Camel Notation	226
Resumen	227
Trabajando en la interfaz de usuario de nuevo	227
Underscore.....	229
Moment.js	233
Less.....	233
Interceptando eventos.....	236
Implementando la búsqueda	238
Query strings.....	239
Resumen	240
Índice.....	242