# Introducción

## About Me.

Mi nombre es Starling Germosen, bueno, Ingeniero Germosen, como suelen poner los possers en sus facebooks 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 y 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.

## Que 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 deberías 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 dioxes 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 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 Botstrap 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 truquitos 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 procesado 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 maquina 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 devolvernos 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 famositos 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 Fundation 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 traking 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 devolvernos 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 comunity 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 caos 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 vea 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, bueh, 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 minututos 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, ¿cual 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 agiles, 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 detales 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 Micrososft, 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”, “doublé” 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 mas 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 vamo 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 fisrt, 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 fisrt, 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 framwork 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 cuan 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 codefirst 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**, **donde** 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 scafolded 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 practica 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 el 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 scafolding, 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 SQLserver, y que la conexión la va a halar 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 el 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,

Y 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 scafolding 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 Starup.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 aplication, por lo que el scafolding 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 Pacakge 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 AplicationUser

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 shorcut “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, options vamos a environment y keyboard, luego digitamos packagemanagerconsole tu 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 mas que un tipo de datos list, y como es una lista debemos especificarle por referencia una lista de que el va a alojar, y el 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 que 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 nulable, 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 AspNet 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 Anotations, 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, asi que vamos a iniciar con las Data anotations 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 “nulables” 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 este dentro de tus requisitos, es ver, ¿cuál es el costo que tiene esto en tu solución y cual 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 mas costoso que llenar una tabla, una pregunta obligatoria seria, ¿cuantas 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 esta 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 mas 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 tu 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 estas 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 costros 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 como 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 Scafolding, para que aprendamos a entender la esencia de las cosas, después nos vamos a valer del Copy paste y el scafolding 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 practicas 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.

El 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 terminologia 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 como 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 si, bueno, adelante, pero le advierto que cuando usted le envíe su proyecto a otra persona para que le de continuidad va a tardar mas 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 mas 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, el 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 mas de como 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", despues 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 esta 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 anotation”, 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 anotation, 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, esta 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 mas un DVC (Dominio, Vista, Controlador), de hecho esta presunción es la mas 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 modulo 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 esta 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 tu 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();
```

y 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 TypeId, el nombre es irrelevante, pero por convención para que haga alusión a lo que será seleccionado, se usa de esta estructura NombreTablaId.

```
            ViewData["TypeId"] = new SelectList(types, "Id", "Name");
o
            ViewBag.TypeId = 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, Iquerable, 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 TypeId 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="TypeId" asp-items="ViewBag.TypeId"> </select>
        <select asp-for="TypeId" asp-items="(SelectList)@ViewData["TypeId"]"> </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 TypeId 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 seria, 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 TypeId { 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 pagina 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 llenaros 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 mas 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 modulo 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 créate de eventos e ir a la etiqueta form, la cual debe ser reemplazada para indicarle a que acción el 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 mi 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 mas 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 TypeId como dicen los gurus, pero

como lo hicimos, puedo indicarle de forma directa que el TypeId de mi evento es igual a mi TypeId 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 del tipo de datos que sea el Id del que vamos a asociar, y lo llamamos el nombre del DominioId, 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 TypeId { get; set; }
[ForeignKey("TypeId")]
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}"),
    TypeId = vm.TypeId,
    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 el lo infiere, así que no esta 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 lamda, 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 esta 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 como esta 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 hacia 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 funcionado igual.

## Separación de Conceptos.

Hay algo mas 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, el 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 esta 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 que 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 TypeId.

```
public class EventViewModel
{
```

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

        public string Time { get; set; }

        public byte TypeId { 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 modulo

Aprendimos a usar el atributo de autorización, para limitar el acceso a usuarios conectados, después veremos como 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.