# Entity Framework

<https://github.com/jerolan/dotnet-entity-framework>

[Entity Framework](https://docs.google.com/presentation/d/1kPhQ_dwHIPGc2KhIxv479TfiB7iEpHLGj8-rz_Y3lPQ/edit?usp=sharing)

## Introduccion

Entity Framework (EF) actúa como un puente entre nuestras entidades y un componente de persistencia. En el contexto de nuestra aplicación, una entidad es un modelo o un objeto capaz de encapsular las características necesarias para la funcionalidad de la aplicación.

Consideremos el flujo típico de una aplicación. Tomaremos como ejemplo una aplicación que gestiona clientes (Customer.cs) y productos disponibles para su compra (CatalogItem.cs).

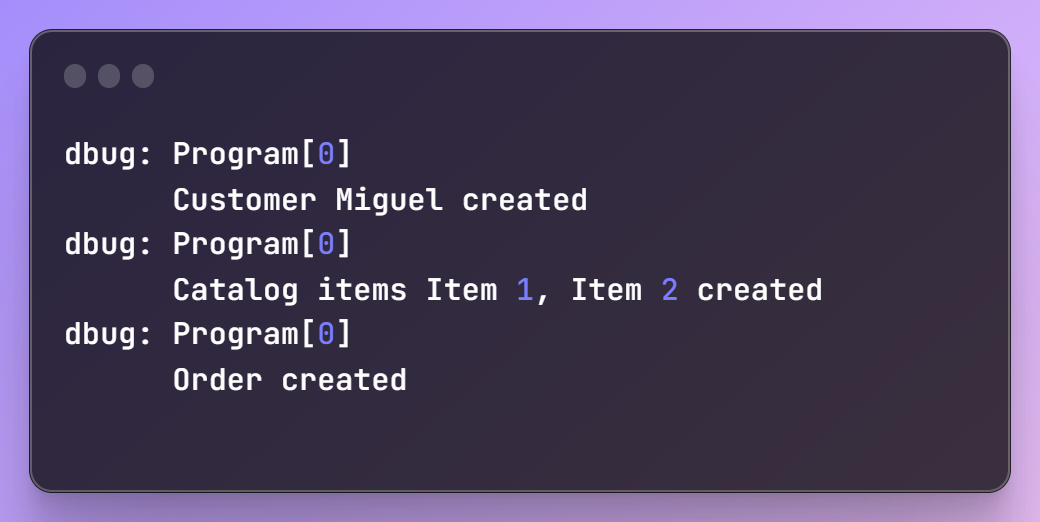


Estas entidades se construyen utilizando tipos primitivos del lenguaje de programación. Imaginemos ahora que queremos expandir nuestra aplicación para incluir un nuevo modelo, Order.cs, que registre las órdenes realizadas por nuestros clientes.

Comenzaremos trabajando con un archivo plantilla llamado Program.cs.template. Lo renombraremos a Program.cs para su uso en la aplicación. Este proceso de renombramiento se puede realizar de diferentes maneras, dependiendo del sistema operativo:

* En Windows: Utiliza el comando PowerShell Rename-Item -Path "Program.cs.template" -NewName "Program.cs".
* En Linux y macOS: Usa el comando mv Program.cs.template Program.cs.

Ejecutaremos la aplicación y observaremos el siguiente output:



Con la base de nuestra aplicación ya definida, es esencial incorporar una capa de persistencia. Esta capa nos permitirá manejar y conservar los datos relacionados con las órdenes, los clientes y el inventario de productos. Entity Framework facilitará este proceso, permitiéndonos interactuar con la base de datos de manera eficiente y efectiva.

## Enfoques

Entity Framework (EF) ofrece tres enfoques fundamentales para trabajar con datos y estructuras de bases de datos, adaptándose a diferentes necesidades y escenarios de desarrollo: Code First, Database First y Model First. Cada enfoque tiene sus características distintivas y casos de uso ideales.

1. Code First

* Descripción: Este enfoque permite a los desarrolladores definir sus modelos de datos utilizando clases C#. EF luego genera la base de datos a partir de estas clases.
* Características y Ventajas:
  + Flexibilidad y Control del Código: Ofrece un control total sobre el código y la estructura de la base de datos, permitiendo cambios ágiles y rápidos.
  + Ideal para Proyectos Nuevos: Particularmente adecuado para iniciar nuevos proyectos sin una base de datos existente.
  + Migraciones Automatizadas: Facilita la gestión de cambios en el modelo de datos con migraciones automáticas.
* Caso de Uso Recomendado: Recomendado cuando se inicia un proyecto desde cero, y se prefiere que la lógica de negocio y las estructuras de datos se definan en el código.

2. Database First

* Descripción: En este enfoque, se comienza con una base de datos ya existente, y EF genera las clases de modelo basadas en el esquema de esta base de datos.
* Características y Ventajas:
  + Integración con Bases de Datos Existentes: Ideal para proyectos que necesitan integrarse con una base de datos ya establecida.
  + Sincronización con el Modelo de EF: Permite mantener el modelo de EF alineado con cambios en la estructura de la base de datos existente.
* Caso de Uso Recomendado: Adecuado cuando ya existe una base de datos o cuando la base de datos es gestionada por un equipo separado. Es útil para proyectos que se integran con sistemas heredados o bases de datos preexistentes.

3. Model First

* Descripción: Este enfoque implica diseñar el modelo de datos visualmente, utilizando herramientas de diseño como Entity Framework Designer. A partir de este diseño, se generan tanto el esquema de la base de datos como las clases de modelo.
* Características y Ventajas:
  + Diseño Visual del Modelo: Facilita la visualización y comprensión de la estructura de datos.
  + Menos Enfoque en el Código: Ideal para aquellos que prefieren trabajar con modelos visuales en lugar de escribir código directamente.
* Caso de Uso Recomendado: Recomendado en las etapas iniciales del diseño de una base de datos o para usuarios que prefieren un enfoque más visual y menos orientado al código.

Cada uno de estos enfoques tiene sus propias fortalezas y se adapta a diferentes etapas del ciclo de vida del desarrollo de software, permitiendo a los equipos elegir el método que mejor se adapte a sus necesidades específicas.

## Quickstart (Model-First)

En este primer ejercicio, adoptaremos el enfoque Code-First en Entity Framework (EF). Este enfoque es ideal cuando estamos desarrollando nuestra aplicación y sus reglas de negocio desde cero, y deseamos integrar una capa de persistencia. Comenzaremos utilizando la carpeta "parte-1" como nuestro punto de partida.

Comenzaremos utilizando el archivo de plantilla DatabaseContext.cs.template. Este archivo necesita ser renombrado para su uso en el proyecto

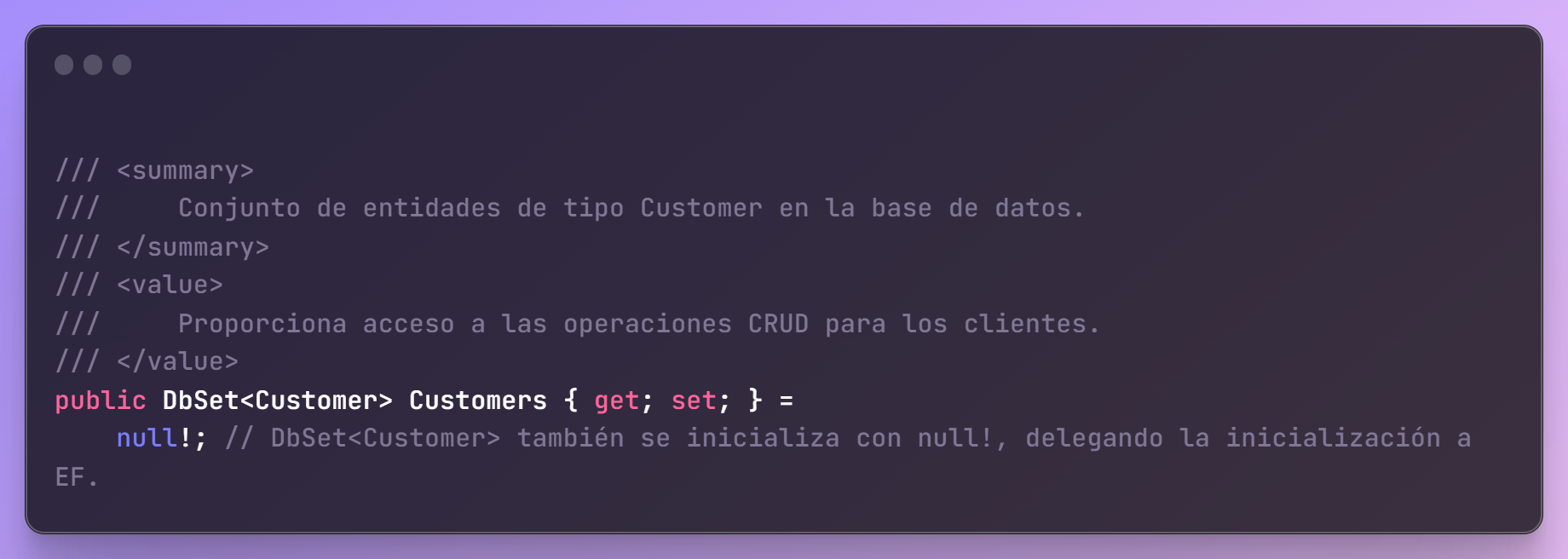
* En Windows: Utiliza el comando PowerShell Rename-Item -Path "DatabaseContext.cs.template" -NewName "DatabaseContext.cs".
* En Linux y macOS: Usa el comando mv DatabaseContext.cs.template DatabaseContext.cs.

Para interactuar con cualquier base de datos, EF requiere la definición de una clase DbContext. Esta clase será la piedra angular de nuestras interacciones con la base de datos.

El DbContext es esta encargada de:

* Consulta de Datos: Permite consultar la base de datos utilizando LINQ (Language Integrated Query), que se convierte en consultas SQL.
* Rastreo de Cambios: Realiza un seguimiento de los cambios realizados en los objetos desde su carga o desde la última vez que se guardaron en la base de datos.
* Persistencia de Datos: Guarda los cambios realizados en los objetos en la base de datos.
* Gestión de Conexiones: Administra la conexión a la base de datos y puede ser configurado para controlar aspectos como el tiempo de espera de la conexión, el tamaño del pool de conexiones, etc.
* Configuración de Modelos: Permite configurar los modelos usando Fluent API o Data Annotations, definiendo aspectos como relaciones, índices, claves primarias, entre otros.
* Administración del Caché de Primer Nivel: Proporciona un caché de primer nivel para entidades, lo que significa que una entidad cargada se mantiene en memoria para consultas futuras, mejorando el rendimiento.

Para que EF reconozca y trabaje con nuestras entidades, es necesario definir cada una de ellas como un DbSet en nuestro DbContext. Esto nos permite aprovechar todas las funcionalidades de EF para cada tipo de entidad.



La asignación = null! Es una forma de decir al compilador: "Confía en mí, esta propiedad será inicializada a un valor no nulo antes de su uso". Es una promesa de que antes de acceder a la propiedad, se le asignará un valor adecuado. En este caso, EntityFramework se encargará de que la propiedad definida por nuestra entidad, tenga el valor esperado.

Una vez que hemos definido los DbSet para cada una de nuestras entidades, el siguiente paso es integrar y configurar nuestro DbContext en la aplicación. Esto nos permitirá manejar efectivamente la persistencia de la información.

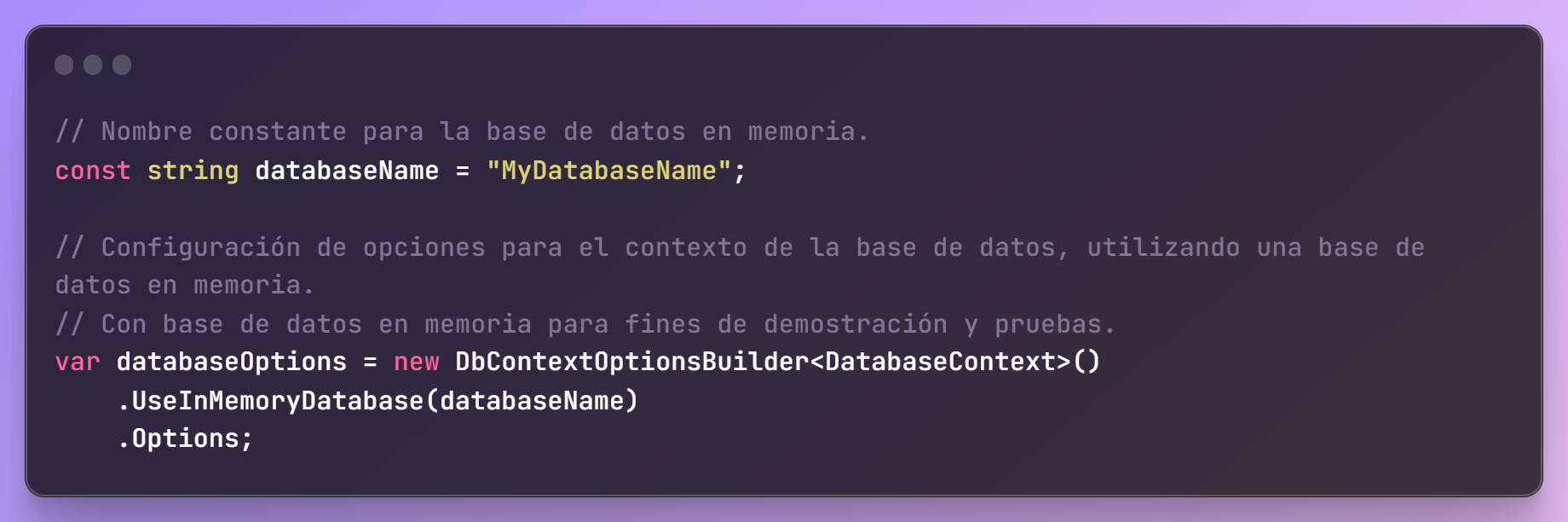
Integración del DbContext en la Aplicación:

* Regresa a tu archivo Program.cs para configurar el uso del DbContext.
* Aquí, inicializaremos el contexto de la base de datos y lo conectaremos con el sistema de inyección de dependencias de nuestra aplicación.

Nombrando la Base de Datos:

* Asigna un nombre a la base de datos en la cadena de conexión o en la configuración del contexto. Reemplaza el marcador de posición “escribe el nombre de tu base de datos” con un nombre significativo para tu proyecto.

Para este ejercicio, utilizaremos una base de datos en memoria proporcionada por el paquete NuGet Microsoft.EntityFrameworkCore.InMemory.



Checkpoint: Hasta este punto, Hasta este punto tenemos listas las precondiciones para poder usar Entity Framework. Hemos definido el contexto de base de datos con nuestras entidades y tenemos datos producto de la aplicación que necesita ser persistida. Aun no hemos seleccionado un motor de base de datos específico, como MySQL, SQL Server, etc. Esto se debe a que Entity Framework (EF) intenta ser agnóstico respecto al sistema de base de datos subyacente. Las funcionalidades proporcionadas por EF son aplicables a una amplia gama de bases de datos relacionales, dándote la libertad de elegir el motor que mejor se adapte a tus necesidades.

Es momento de instanciar la base de datos, con las características definidas



Para persistir el producto de nuestra primera entidad Customer



Una vez establecido nuestro contexto de base de datos en Entity Framework, podemos acceder a los DbSet que definimos, representando nuestras entidades de aplicación. Este acceso nos permite realizar operaciones de persistencia de datos. Por ejemplo, al agregar un nuevo cliente (Customer), creamos una instancia de esta entidad y la agregamos al contexto de EF utilizando el método Add. Sin embargo, es importante recordar que, aunque hemos agregado la entidad al contexto, todavía no se ha escrito en la base de datos. Para efectuar estos cambios, necesitamos invocar el método SaveChanges. Este método es crucial, ya que es el responsable de escribir en la base de datos todos los cambios ejecutados sobre las entidades de las cuales Entity Framework tiene conocimiento.

Además de agregar entidades individuales, Entity Framework Core, a partir de su versión 8, ofrece la capacidad de agregar múltiples entidades simultáneamente utilizando el método AddRange. Este método es especialmente útil cuando se trabaja con listas de entidades, ya que permite una inserción de múltiples registros en la base de datos.



En el contexto de nuestra aplicación, supongamos que deseamos crear una orden específica para un artículo llamado "Item 2". Realizar búsquedas en la base de datos es una funcionalidad común en muchas aplicaciones, y Entity Framework está diseñado teniendo esto en cuenta.

Cada DbSet en Entity Framework implementa la interfaz IQueryable<T>. Esto significa que podemos aprovechar los métodos de extensión de LINQ para construir consultas eficientes y expresivas.

NOTA:

Consultas SQL en el Servidor: Es crucial entender que todas las operaciones realizadas utilizando IQueryable<T> se traducen en consultas SQL y se ejecutan en el servidor de base de datos. Esta característica es vital para la eficiencia, especialmente con grandes conjuntos de datos.

Operaciones en Memoria con IEnumerable<T>: Por otro lado, cuando las consultas se ejecutan sobre IEnumerable<T>, estas ocurren en memoria. Esto puede conducir a problemas de rendimiento cuando se manejan grandes volúmenes de datos.

Filtrado con IQueryable: Para garantizar una mayor eficiencia y rendimiento, es recomendable realizar todos los filtros necesarios sobre la interfaz IQueryable antes de ejecutar la consulta en la base de datos. Esto permite que los filtros se traduzcan en consultas SQL, lo cual es más eficiente para el manejo de datos.

Limitaciones y Filtrado a Nivel de Aplicación: Ten en cuenta que IQueryable está limitado a aplicar filtros que se puedan traducir en consultas SQL. Si tus filtros requieren lógica más compleja, que involucra instancias de objetos que no se pueden convertir fácilmente en consultas SQL, entonces deberás realizar estos filtros a nivel de aplicación, utilizando la interfaz IEnumerable.



Checkpoint: Hemos logrado un progreso significativo en nuestra aplicación, pasando de no contar con una capa de persistencia a implementar un contexto de base de datos eficaz utilizando Entity Framework. Este avance nos sitúa en una posición óptima para seleccionar cualquier motor de base de datos que sea adecuado para nuestra aplicación, permitiéndonos reemplazar la base de datos en memoria (InMemoryDb) cuando sea necesario.

Ventajas del Uso de In Memory Db

La base de datos en memoria, In Memory Db, ha demostrado ser una herramienta excelente para realizar pruebas rápidas, como las que hemos llevado a cabo en este ejercicio. Además, es particularmente útil para ejecutar pruebas unitarias y de integración, especialmente en escenarios donde queremos evitar la interacción con una base de datos física. Su capacidad para simular operaciones de base de datos sin la necesidad de una configuración de base de datos externa la convierte en una opción ideal para pruebas y desarrollo ágil.

Transición hacia un Motor de Base de Datos Real

Mientras que In Memory Db es ideal para pruebas, para entornos de producción o desarrollo avanzado, podríamos considerar la transición a un motor de base de datos más robusto. Entity Framework ofrece la flexibilidad de cambiar fácilmente a cualquier motor de base de datos compatible, como SQL Server, MySQL, PostgreSQL, entre otros. Esta capacidad de adaptación asegura que nuestra aplicación puede evolucionar según las necesidades del proyecto y el entorno de despliegue.

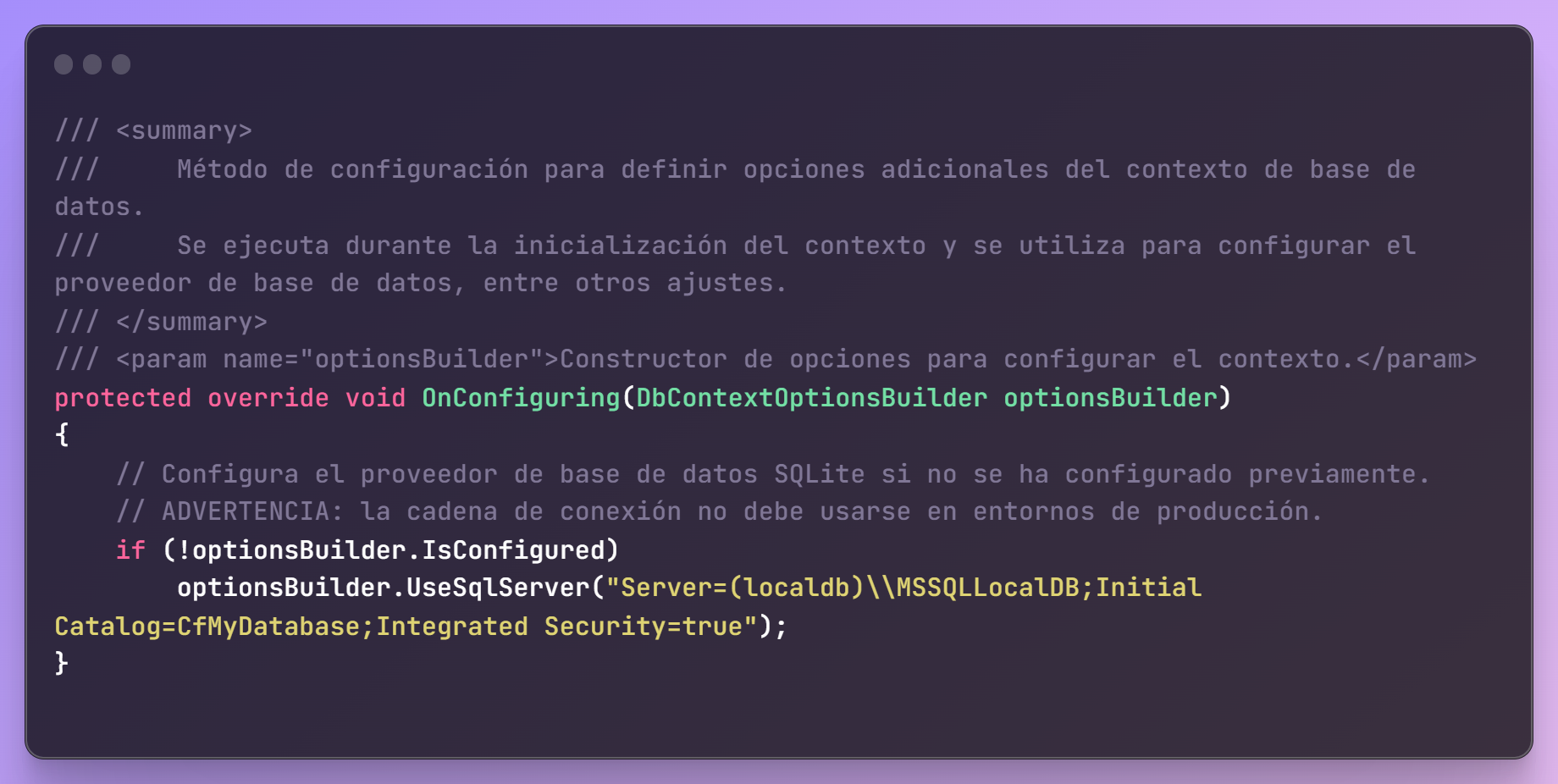
## Gestores de base de datos (Sql Server)

En esta segunda parte del curso, ampliaremos nuestra aplicación para incorporar SQL Server como gestor de base de datos. Continuaremos aprovechando el DbContext y las entidades que ya hemos definido, utilizando la carpeta “parte-2” como nuestro punto de partida.

Para integrar SQL Server, es necesario instalar el paquete NuGet Microsoft.EntityFrameworkCore.SqlServer. Al igual que con la base de datos en memoria, este paquete es esencial para trabajar con SQL Server en Entity Framework. Es interesante observar cómo Entity Framework proporciona las funcionalidades básicas en su paquete principal y, según necesitamos ampliar nuestras capacidades de gestión de bases de datos, se requiere agregar paquetes adicionales específicos.

Volveremos a nuestra clase DatabaseContext.cs y en esta ocasión, vamos a sobrescribir el método OnConfiguring. Entity Framework cuenta con varios métodos de ciclo de vida que se invocan durante la instancia del objeto. Si deseamos extender el comportamiento de uno de estos ciclos, debemos sobrescribir los métodos correspondientes.

El método OnConfiguring es particularmente importante, ya que se ejecuta en la etapa inicial de configuración de la instancia del contexto. Aquí es donde especificaremos que nuestra aplicación debe utilizar SQL Server. Para los propósitos de este ejercicio, emplearemos [LocalDB](https://learn.microsoft.com/en-us/sql/database-engine/configure-windows/sql-server-express-localdb?view=sql-server-ver16), una versión ligera y fácil de configurar de SQL Server, ideal para el desarrollo y pruebas



Checkpoint: Hasta este momento, hemos ajustado nuestra aplicación para utilizar SQL Server, reemplazando la configuración previa que utilizaba una base de datos en memoria. Para ello, hemos modificado el método OnConfiguring en nuestra clase DatabaseContext, indicándole que utilice SQL Server como nuestro sistema de gestión de bases de datos.

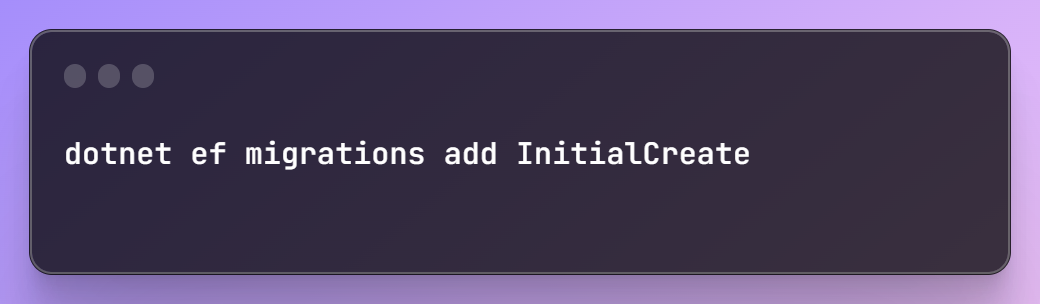
Aunque nuestra aplicación está configurada para usar SQL Server, todavía necesitamos establecer el esquema de la base de datos en el servidor. Las bases de datos relacionales como SQL Server utilizan esquemas de datos para modelar tablas y sus relaciones. Sin embargo, en nuestra aplicación actual, el esquema definido en el código C# aún no se ha implementado en el servidor SQL Server.

Entity Framework facilita la transferencia del esquema definido en nuestro código C# a la base de datos mediante un proceso conocido como "Migraciones". Las migraciones son scripts que se ejecutan en el servidor de base de datos y son cruciales para crear estructuras de tablas, migrar valores entre columnas, actualizar nombres y realizar otras modificaciones esenciales de manera controlada. Este proceso ayuda a mantener la consistencia y la integridad de los datos.

Para generar nuestra primera migración de base de datos, utilizaremos la herramienta de línea de comandos dotnet-ef.



Posteriormente podremos ejecutar el comando

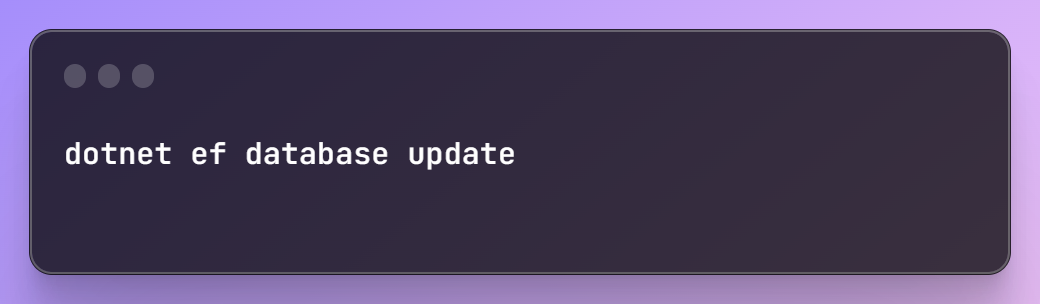


El cual producirá un archivo InitialCreate.cs en la carpeta Migrations, si inspeccionamos este archivo, veremos los comandos que utiliza Entity Framework, para que desde los modelos definidos por nuestra aplicación pueda crear las tablas adecuadas.

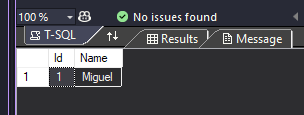


Particularmente para la tabla Orders, podemos ver cómo EF reconoce las relaciones entre los modelos y crea las llaves foráneas, necesarias para establecer relaciones entre tablas y poder hacer operaciones en conjuntos.

Nuestra base de datos aun no esta creada, por ahora solo tenemos las instrucciones que la van a materializar, por lo que ahora solo nos queda ejecutar el comando:



Finalmente ejecutaremos nuestra aplicación en Program.cs y haremos una consulta a nuestra base datos para confirmar que los registros se han realizado adecuadamente.



Checkpoint: Hasta este punto logramos la transición exitosa del modelo de base de datos en memoria a SQL Server. Este cambio nos ha permitido observar y gestionar nuestros datos directamente en la base de datos de SQL Server.

Es importante destacar que el proceso de cambiar nuestro proveedor de base de datos a uno diferente no ha alterado el funcionamiento de nuestra aplicación. Esto demuestra la eficacia de las abstracciones proporcionadas por Entity Framework (EF), especialmente a través de DbContext. La capacidad de EF para abstraer los detalles específicos del proveedor de base de datos asegura que los cambios subyacentes en la gestión de datos no afecten la lógica de negocio de nuestra aplicación.

## Practica: Configura SQLite como gestor de base de datos

En este tercer ejercicio, adoptaremos un enfoque similar al anterior, pero esta vez, nuestro objetivo es configurar SQLite como nuestro gestor de base de datos.

Pasos Clave para la Configuración de SQLite:

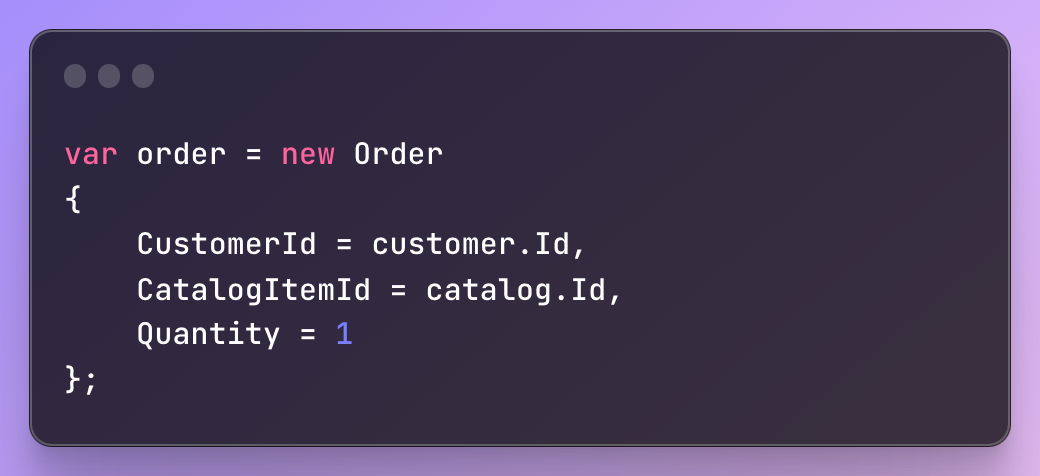
* Continúa desde donde terminaste en el ejercicio de la práctica 2. Esto te permitirá aprovechar la estructura y configuraciones ya establecidas en tu aplicación
* Para integrar SQLite en tu proyecto, es esencial añadir la dependencia Microsoft.EntityFrameworkCore.Sqlite. Esta librería permite que Entity Framework interactúe con SQLite.
* Antes de proceder, es importante eliminar la carpeta de migraciones creada previamente para SQL Server. Las estrategias de migración son específicas para cada gestor de base de datos, y las migraciones creadas para SQL Server no serán compatibles con SQLite. Eliminar estas migraciones previas evitará conflictos y garantizará que las nuevas migraciones estén optimizadas para SQLite.
* Tu enfoque principal en este ejercicio será trabajar en el método OnConfiguring de tu clase DbContext.

## Transacciones

De forma predeterminada, si el proveedor de la base de datos soporta transacciones, todos los cambios realizados en una sola llamada a SaveChanges se aplican dentro de una transacción. Si alguno de los cambios falla, la transacción se revierte y ninguno de los cambios se aplica a la base de datos. Esto significa que SaveChanges garantiza que los cambios se apliquen completamente con éxito o, en caso de error, no modifique la base de datos.

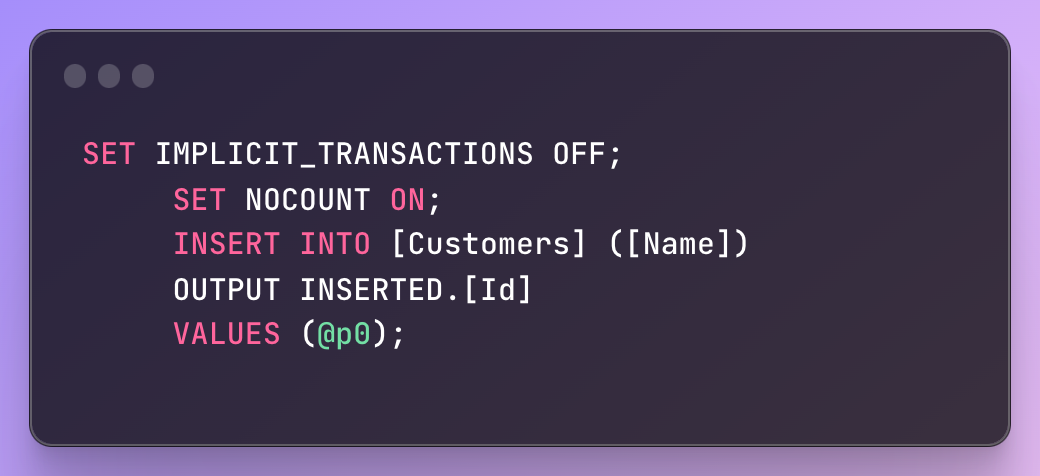
Sin embargo, EF te permite implementar transacciones en tu aplicación en caso de que necesites tener un control fino, esto es recomendable en casos de uso donde tu, tienes que trabajar o pensar en performance vs consistencia.

Una primera limitante es que EF controla los identificadores de base de datos automatizados y para lograr esto, hace una consulta directa a la base de datos para determinar cuál es su identificador. Por lo que Order en el contexto de nuestra aplicación, no pudiera mantener la relación entre las entidades, dado que no sabemos cual es el ID correcto en ese momento.



Las transacciones pueden usarse en conjunto con SaveChanges, EF crea automáticamente un punto de retorno antes de guardar los datos. Los puntos de retorno son puntos dentro de una transacción de base de datos a los que se puede revertir más tarde en caso de que ocurra un error o por cualquier otro motivo.

Cuando usamos transacciones junto con SaveChanges, a medida que se van creando los puntos de retorno, podemos ver como EF hace las llamadas a la base de datos para calcular los IDs auto incrementales



Es importante entender, que el uso de transacciones tiene la finalidad de mantener la consistencia de la información, cuando se trabaja con múltiples datos existen 4 niveles estándar: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ y SERIALIZABLE.

* READ UNCOMMITTED: Este nivel permite que una transacción lea datos que aún no se han confirmado en la base de datos. Ofrece el mayor rendimiento, pero también puede conducir a errores de consistencia, por leer datos que aún no están guardados de otra transacción y estos pueden desaparecer.
* READ COMMITTED: Permite que una transacción lea solo datos que ya están confirmados. Esto evita el problema de leer datos que "luego desaparecen", pero no protege contra leer datos que puedan ser modificados por otra transacción tras haberlos leído.
* REPEATABLE READ: Este nivel intenta evitar el problema de leer los mismos datos dos veces y obtener resultados diferentes. Básicamente, si pudo leer exitosamente información de una tabla, si la vuelve a consultar, resultará en la misma información consistente.
* SERIALIZABLE: Trata de evitar todos los fenómenos de lectura. Es el nivel más restrictivo y asegura que cada transacción se ejecute de manera aislada, pero puede reducir el rendimiento debido a un mayor bloqueo de datos. Por ejemplo, puedes causar que varias transacciones se apilen y fallen eventualmente por el tiempo en el que estas tardan en resolverse.

En un uso avanzado de estas estrategias, a la vez que debemos controlar el nivel de acceso, también debemos resolver el problema, cuando dos operaciones intentan leer datos en alguno de estos 4 niveles de acceso.

* Bloqueo Pesimista (Pessimistic Locking):
  + En el bloqueo pesimista, se evitan problemas asegurándose de que una transacción no introduzca cambios problemáticos, bloqueando partes específicas de la base de datos. Cuando una parte está bloqueada por una transacción, otras no pueden leer o escribir en ella, dependiendo del nivel de aislamiento de la transacción, para prevenir conflictos.
  + Los bloqueos en la base de datos pueden variar en nivel, desde el nivel de fila hasta el nivel de la base de datos completa, e incluyen diferentes tipos como bloqueos de lectura, de escritura, compartidos o exclusivos, de intención y otros.
* Bloqueo Optimista (Optimistic Locking):
  + El bloqueo optimista, funciona con un número de versión asociado a cada entidad en la tabla. Al modificar una fila, su versión aumenta, permitiendo que otras transacciones noten el cambio.
  + Al iniciar, una transacción registra el número de versión de las filas para conocer su estado. Solo lee filas modificadas antes de su inicio y, al intentar confirmar cambios, la base de datos verifica las versiones de las filas. Si han sido modificadas mientras tanto por otra transacción, la actualización se rechaza y la transacción debe comenzar de nuevo.

Checkpoint: Hasta este punto hemos aprendido cómo EF mantiene por defecto la consistencia en la información que escribimos en base de datos gracias a las transacciones. También como EF expone clases que nos permiten ajustar de forma fina, el comportamiento de la información en las transacciones, sin embargo, se debe entender como el manejador de base de datos expone los niveles de acceso a la información, como las estrategias de concurrencia a considerar para entender, qué combinación es la que nos ofrece mayor performance y consistencia.

## Personalización de migraciones

Como se explicó anteriormente, Entity Framework (EF) facilita la creación y configuración de la estructura de las tablas y columnas que deseamos implementar físicamente en el gestor de base de datos. EF automatiza gran parte de este proceso, simplificando significativamente el desarrollo de la base de datos.

Sin embargo, al igual que con las transacciones, tenemos la capacidad de ejercer un control detallado sobre la forma en que se crean las propiedades en la base de datos. Esta flexibilidad nos permite optimizar el uso de nuestro gestor de base de datos, adaptándolo a las necesidades específicas de nuestra aplicación.

Por ejemplo, podemos especificar tipos de datos personalizados, definir índices para mejorar la eficiencia en las consultas, establecer restricciones y relaciones entre tablas, y configurar otras propiedades de las columnas como su longitud máxima, si son obligatorias o no, y cómo deben mapearse a las entidades de EF. Todo esto se puede hacer mediante el uso de Data Annotations en nuestras clases de modelo o mediante la API Fluent, que proporciona un enfoque más detallado y flexible para configurar el modelo de datos.

Vamos a continuar a partir del ejercicio exterior en “parte-2-extra”, donde vamos a personalizar algunas propiedades de nuestros modelos y continuando con el ejercicio de transacciones, vamos a aplicar una configuración avanzada para manejar la concurrencia optimista.

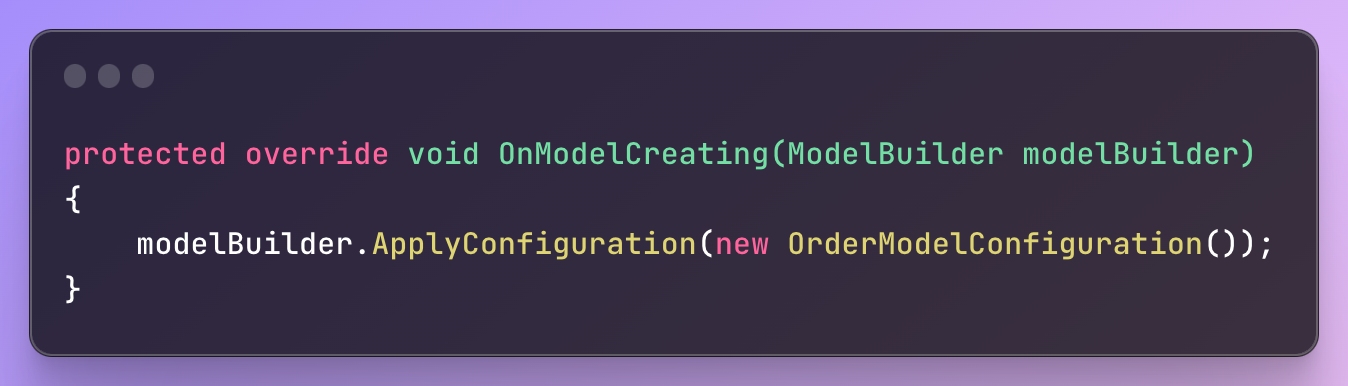
Crearemos una clase llamada OrderModelConfiguration la cual implementa la interfaz IEntityTypeConfiguration la cual está asociada a una entidad en cuestión. Para este ejercicio usaremos Order como referencia, al implementar la interfaz e implementar los métodos requeridos, vemos que tenemos acceso a un objeto “builder”, por el cual podemos configurar, propiedades, tipos de datos, relaciones, etc de una entidad.

Vamos a hacer uso de dos funcionalidades de EF, la capacidad de crear propiedades shadow y los concurrency tokens a través de row version.

* Propiedades Shadow en Entity Framework: Las propiedades shadow son aquellas que EF maneja en el modelo de datos, pero que no están definidas explícitamente en las clases de entidad de nuestra aplicación. Estas propiedades son útiles para mantener datos que son importantes para el modelo de la base de datos, pero no necesariamente para el dominio de la aplicación.
* Concurrency Tokens y Row Versioning: Los concurrency tokens son un mecanismo para manejar conflictos de concurrencia en una base de datos. Un enfoque común es utilizar una 'row version' (versión de fila). Cada vez que se actualiza un registro, esta versión de fila se incrementa, lo que permite a EF detectar cambios conflictivos.



Para aplicar cualquiera la configuración creada dabamos ir nuevamente a nuestro DbContext y sobreescribir el método OnModelCreating



Crearemos y aplicaremos la migración con nuestra propiedad shadow, para luego ejecutar el programa. Así podemos observar cómo, al crear objetos en Orden, se crea automáticamente el valor Versión, así también vemos como, cada que actualizamos una entidad, este valor se modifica.

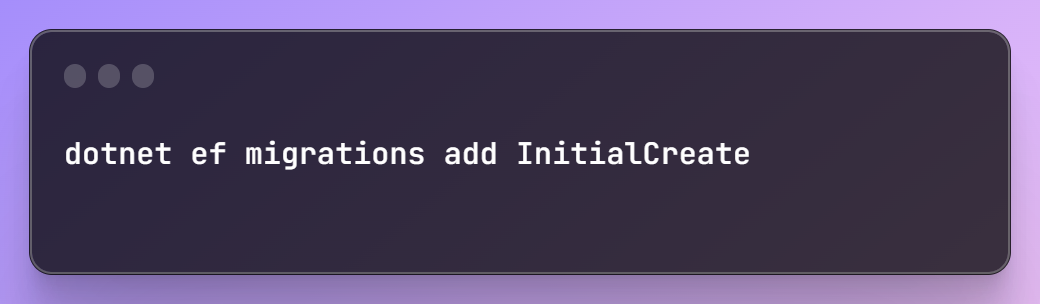
Las concurrency tokens se pueden usar en conjunto con un nivel de insolación más inseguro como ReadCommitted, aunque podamos leer datos de otra entidad, los cuales podrán haber cambiado por consecuencia de otra operación concurrente, EF nos alertara de esto cuando detecte que la versión ya no es válida y nos haga reintentar nuestra transacción.

Checkpoint: Hasta este punto hemos visto, que a pesar de que EF controla en gran parte las propiedades, transacciones y comportamientos de nuestra aplicación contra la base de datos, nos ofrece interfaces, clases y configuraciones que nos permiten aplicar técnicas avanzadas dependiendo del caso de uso de nuestra aplicación.

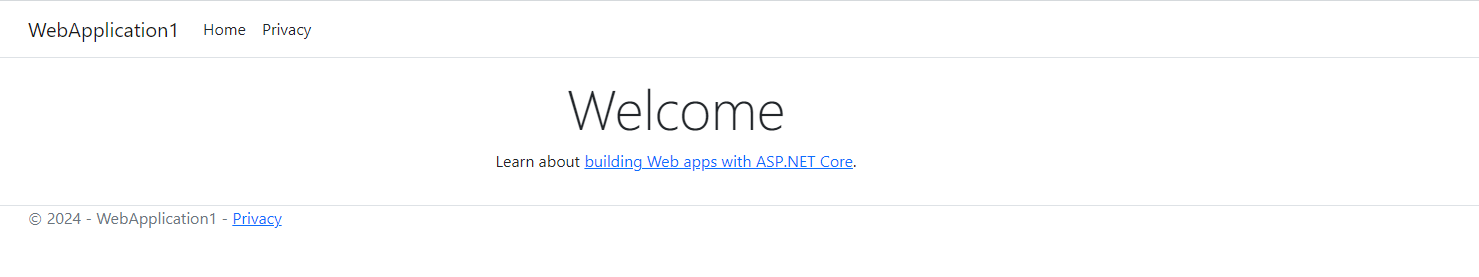
## Configura seeders

En este tercer ejercicio, profundizaremos en el entendimiento de los ciclos de vida de DbContext y el manejo de migraciones en Entity Framework. Las migraciones, como hemos mencionado anteriormente, son una herramienta crucial en la gestión de bases de datos, permitiendo modificar la estructura de datos de forma ordenada y segura, garantizando siempre la integridad y consistencia de la información.

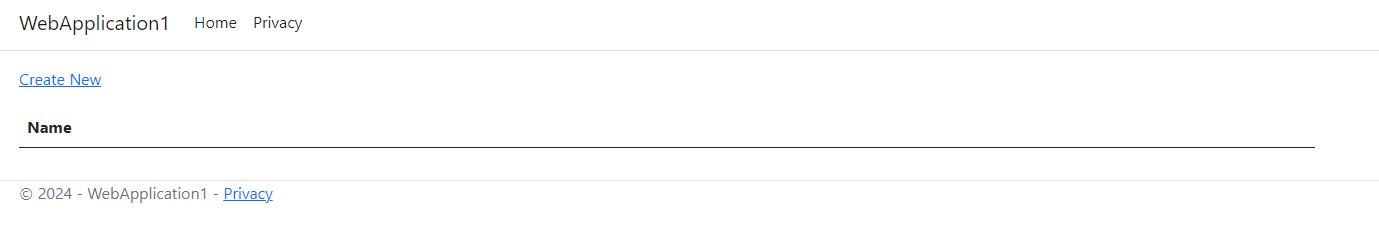
Tenemos que inicializar nuestra base de datos, como vimos en el capítulo anterior.



Cuando ejecutemos nuestro proyecto, nos encontraremos con una pantalla de bienvenida. Este proyecto aprovecha las herramientas proporcionadas por ASP.NET, particularmente su generador para modelos de Entity Framework. Aunque en esta sesión no profundizaremos en cómo trabajar detalladamente con ASP.NET, sí utilizaremos algunas de sus funcionalidades clave. Esto nos permitirá visualizar mejor el alcance de este capítulo y entender cómo Entity Framework se integra en una aplicación ASP.NET.



La dependencia Microsoft.VisualStudio.Web.CodeGeneration.Design nos permite generar vistas CRUD de cada uno de los modelos en EF. Si navegamos a la ruta /Customers nos desplegará una lista vacía

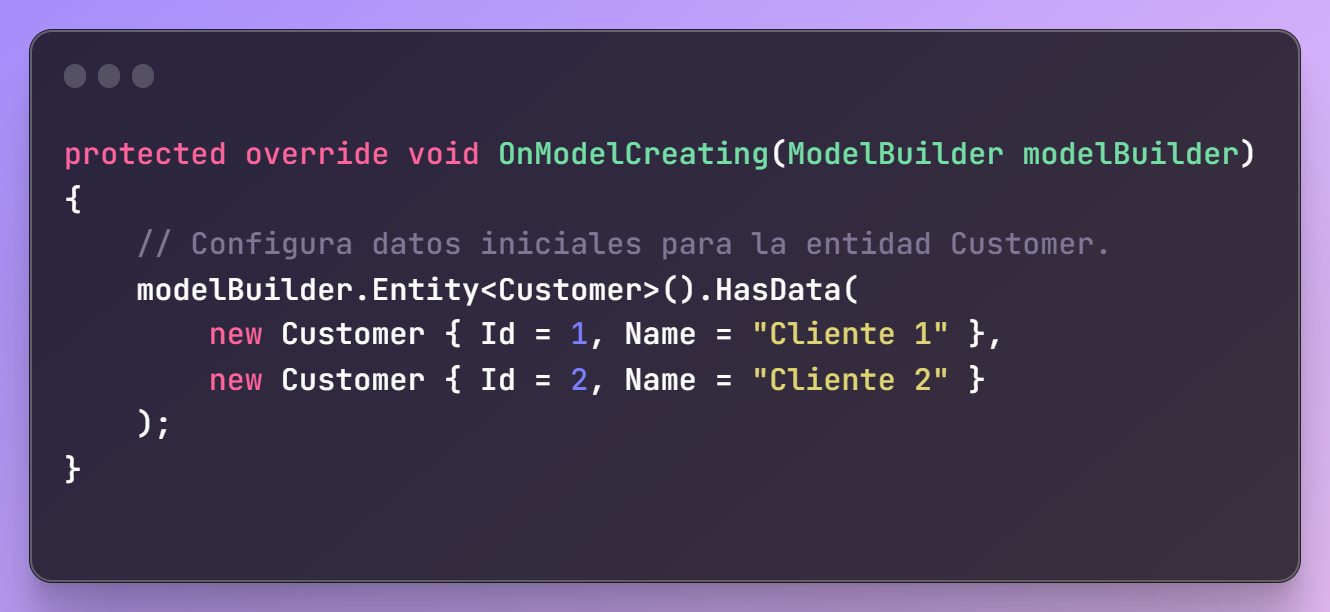


Entity Framework facilita la creación de datos de prueba o iniciales para nuestra aplicación mediante el uso de un "Seed" o información semilla. Esta funcionalidad es particularmente útil en diversos escenarios, como cuando se tienen tablas de catálogos con información que debe cargarse automáticamente debido a ciertas reglas de negocio, datos de configuración, o más comúnmente, para proporcionar datos de prueba en entornos de desarrollo.

Así como sobreescribimos el método OnConfiguring, parte del ciclo de vida de EntityFramework, esta vez trabajaremos con el método OnModelCreating

* OnConfiguring se llama cada vez que se crea una instancia de DbContext. Este método se utiliza para configurar el DbContext, especialmente la cadena de conexión y otros aspectos relacionados con la base de datos.
* OnModelCreating se llama una sola vez, cuando EF construye el modelo de datos por primera vez. Esto sucede generalmente la primera vez que se accede a DbContext en la ejecución de la aplicación. El modelo se cachea y se reutiliza en instancias posteriores de DbContext.

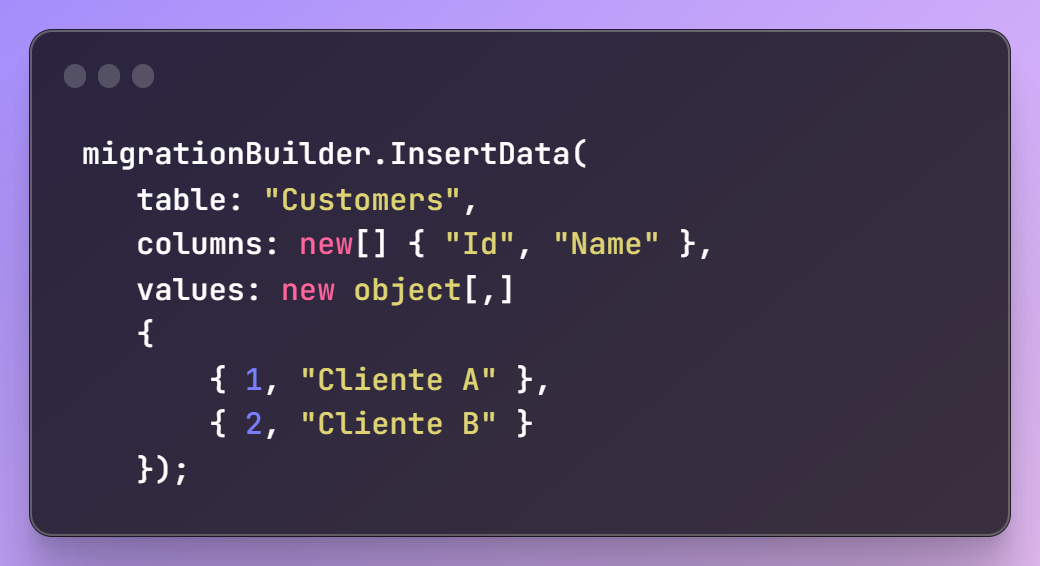
Nosotros usaremos OnModelCreating para definir la información de seed en nuestras tablas



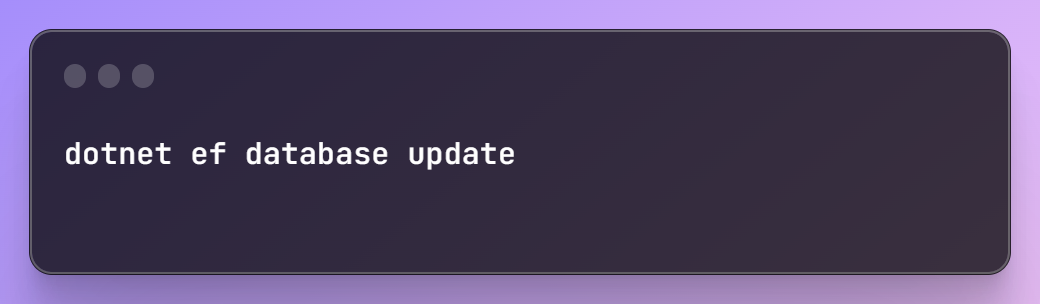
Seguimos utilizando la estrategia Code First, por lo que cada cambio que afecte el DbContext debe sincronizarse con la base de datos a través de una migración. Una vez más utilizaremos el comando `migration add` para generar una migración, a la cual nombraremos AddSeedData



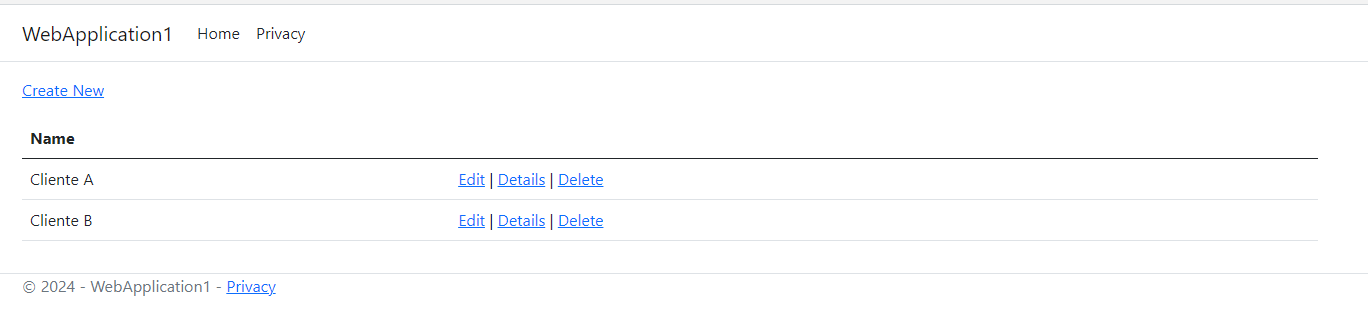
Esta migración, que se posiciona después de la que ya habíamos generado para definir las tablas, detecta que los únicos cambios entre el DbContext y la base de datos son, los datos definidos en OnModelCreating, por lo que solo incluye esta información en los comandos de migración



Aplicamos la migracion



Para finalmente refrescar la pantalla /Customers



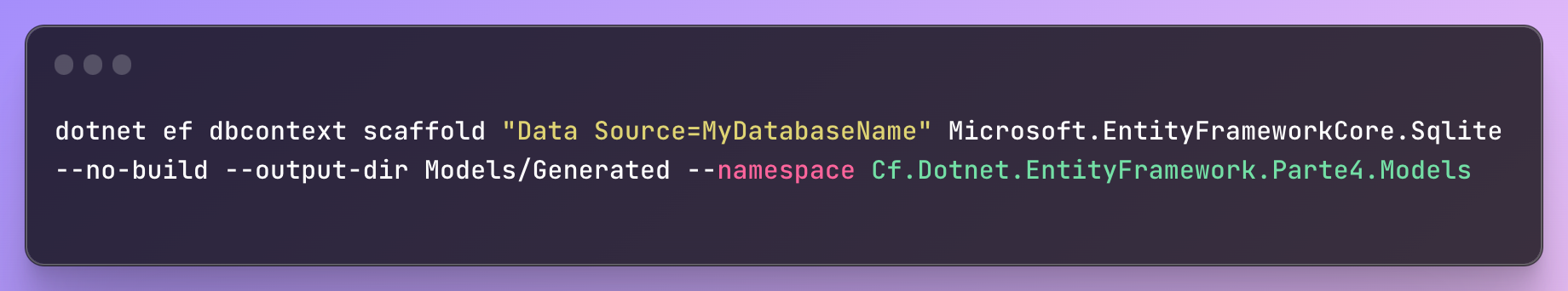
Checkpoint: Hasta este momento, hemos ampliado nuestro contexto de base de datos para configurar datos iniciales en nuestra base de datos. Mediante el uso de migraciones, hemos logrado aplicar estos cambios en la información existente, manteniendo su integridad.

## Quickstart (Existing Database)

Hasta ahora, hemos aplicado la estrategia Code First para trabajar con nuestra base de datos, realizando cambios en el nivel del código C# y aplicando estos cambios en la base de datos a través de los comandos de Entity Framework (EF).

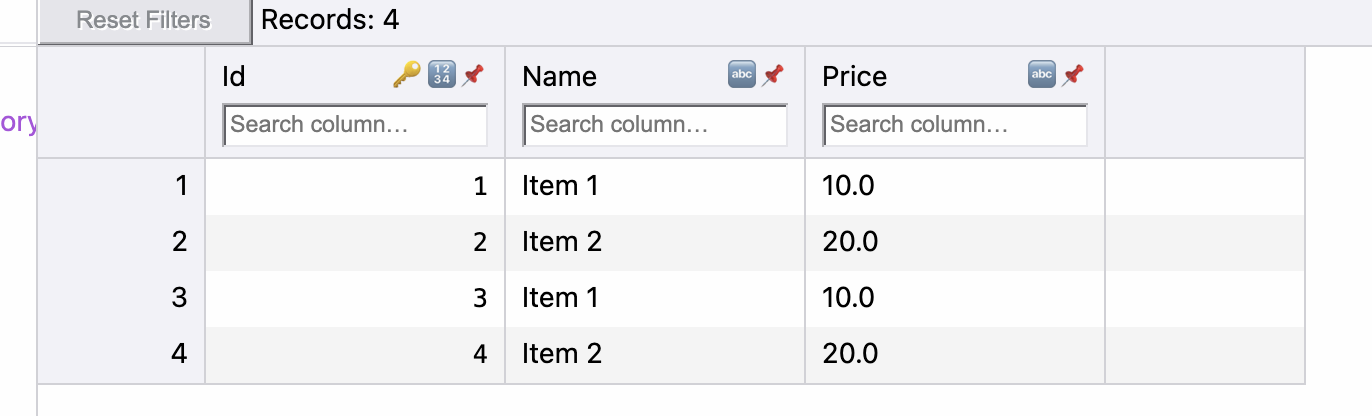
Sin embargo, EF también ofrece la capacidad de trabajar con bases de datos ya existentes. En esta fase, paso-4, exploraremos cómo iniciar con una base de datos existente. Dentro de la carpeta “parte-4”, encontraremos una base de datos SQLite con la cual debemos trabajar, utilizando la línea de comandos de EF y el nombre de la base de datos, **MyDatabaseName**.

Para comenzar, observaremos los datos iniciales presentes en dicha base de datos utilizando herramientas como <https://sqliteviewer.app/>. Posteriormente, generaremos el contexto de base de datos a partir de esta base de datos existente, lo que nos permitirá interactuar con ella utilizando EF en el contexto de nuestra aplicación.



* dbcontext scaffold: Esta subcomando de dotnet ef indica que se va a generar (o "scaffold") un contexto de base de datos y clases de entidad basadas en la estructura de una base de datos existente.
* "Data Source=MyDatabaseName": Esta es la cadena de conexión a la base de datos. En este caso, se conecta a una base de datos SQLite. MyDatabaseName debería ser el nombre del archivo de la base de datos SQLite.
* Microsoft.EntityFrameworkCore.Sqlite: Este es el proveedor de base de datos que se utiliza. En este caso, se especifica que la base de datos es SQLite.
* --no-build: Esta opción evita que el comando intente compilar el proyecto antes de ejecutar el scaffolding. Para este caso, como la aplicación que tenemos preparada para el ejercicio depende de que la clase DbContext esté creada.
* --output-dir Models/Generated: Esta opción especifica el directorio donde se colocarán los archivos generados. En este caso, los archivos se generarán en un directorio llamado Generated dentro de un directorio Models.
* --namespace Cf.Dotnet.EntityFramework.Parte4.Models: Esta opción establece el espacio de nombres para las clases generadas. Las clases tendrán el espacio de nombres Cf.Dotnet.EntityFramework.Parte4.Models.

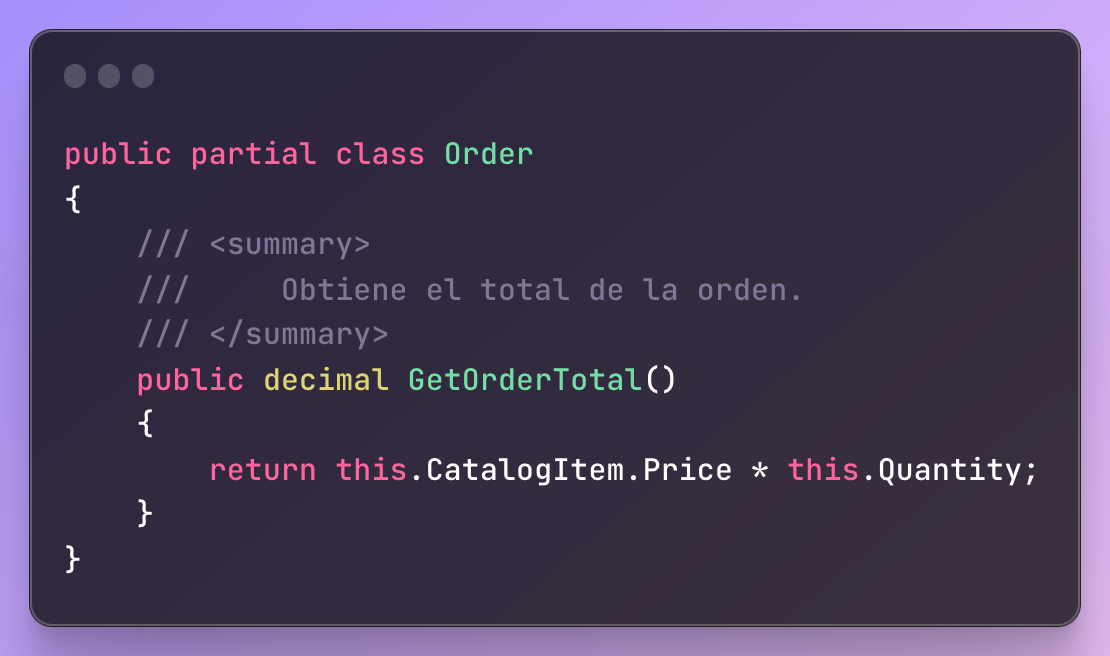
Ejecutamos la aplicación una vez generadas las clases de contexto de base de datos y modelos y observemos a través del explorador web de sqliteviewer para confirmar que nuestros datos se crearon con éxito.



Una estrategia destacable del enfoque Code First es que nos permite mantener los modelos de la base de datos separados de nuestras entidades de aplicación. Esta práctica se alinea estrechamente con los principios de separación de responsabilidades y ofrece la flexibilidad de cambiar fácilmente el proveedor de base de datos.

Para evitar que nuestra aplicación dependa exclusivamente de los modelos generados por el proceso de scaffolding, es recomendable usar clases parciales. Alternativamente, en un enfoque diferente, podemos emplear métodos de extensión. Esto es crucial para garantizar que nuestros métodos, que contienen reglas de negocio esenciales, no se pierdan cada vez que actualicemos los modelos locales. Cabe recordar que cada vez que la base de datos de origen experimente cambios, es necesario actualizar nuestros modelos locales para reflejar esos cambios.

Para que las clases parciales funcionen, es importante que ambas clases de entidad que compartan el mismo nombre se encuentren dentro del mismo espacio de nombres (namespace).



Checkpoint: Hasta ahora, hemos explorado cómo comenzar con una base de datos existente, automatizando la generación de modelos de base de datos. Además, hemos aprendido a preservar nuestros métodos que contienen reglas de negocio, utilizando clases parciales para evitar su pérdida durante este proceso de generación automática.

## SPs

Hasta el momento, hemos discutido cómo Entity Framework (EF) se integra con modelos que se corresponden de manera directa con tablas en la base de datos. Sin embargo, EF también ofrece la capacidad de trabajar con modelos no convencionales o que no se mapean directamente a tablas específicas.

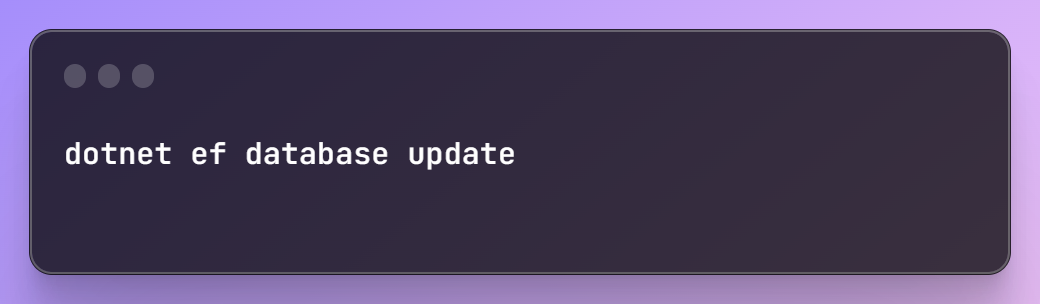
Esta funcionalidad resulta particularmente valiosa en varios casos de uso, tales como:

* Modelos Solo de Consulta: En muchas aplicaciones, es común necesitar modelos que se construyan a partir de datos provenientes de múltiples tablas. Estos modelos suelen seleccionar propiedades específicas y realizar uniones (joins), principalmente por cuestiones de rendimiento.
* Llamadas a Procedimientos Almacenados: EF permite la integración con procedimientos almacenados en la base de datos. Estos procedimientos se utilizan para encapsular lógica de la base de datos, ya sea por motivos de rendimiento o para facilitar la reutilización de esta lógica entre diferentes sistemas.

Hasta la versión 8 de Entity Framework (EF), era una práctica común combinar EF con Dapper, una librería de base de datos que ofrecía ventajas significativas en términos de rendimiento para ciertos casos de uso. Además, EF requería configuraciones adicionales, como la necesidad de registrar manualmente cada modelo no indexado en el DbContext.

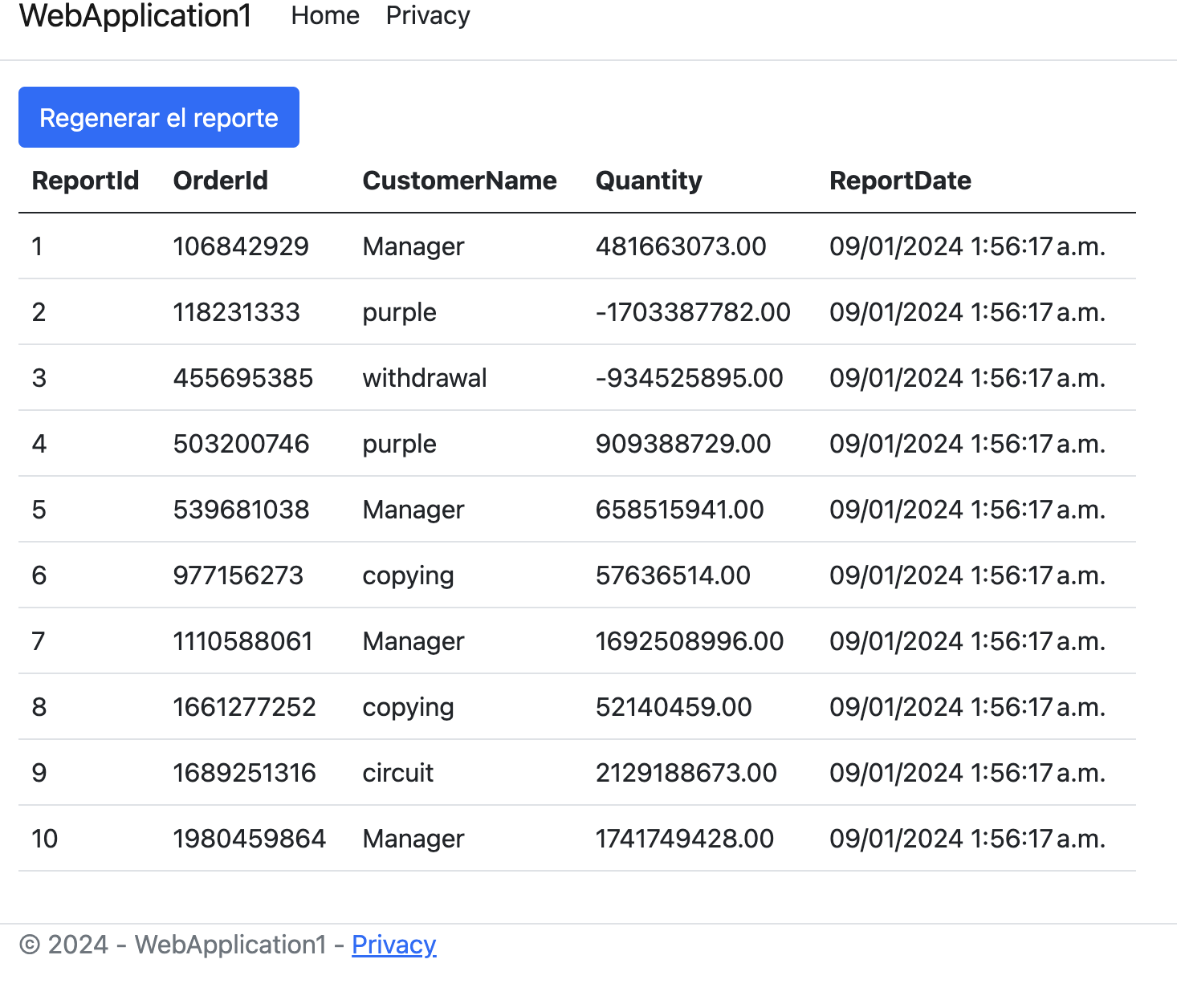
Sin embargo, esta situación ha cambiado en las versiones más recientes de EF. Ahora es posible crear Data Transfer Objects (DTOs) y realizar proyecciones de vistas, consultas o procedimientos almacenados (SPs) a modelos de C# sin la necesidad de registrar estos modelos en el DbContext. Esta mejora en EF amplía su flexibilidad y facilita su uso, permitiendo una integración más eficiente y directa con una variedad de fuentes de datos y estructuras de modelos.

Para este ejercicio tomaremos como base la parte-5. Crearemos nuestra base de datos con los comandos que hemos estado implementando hasta ahora.



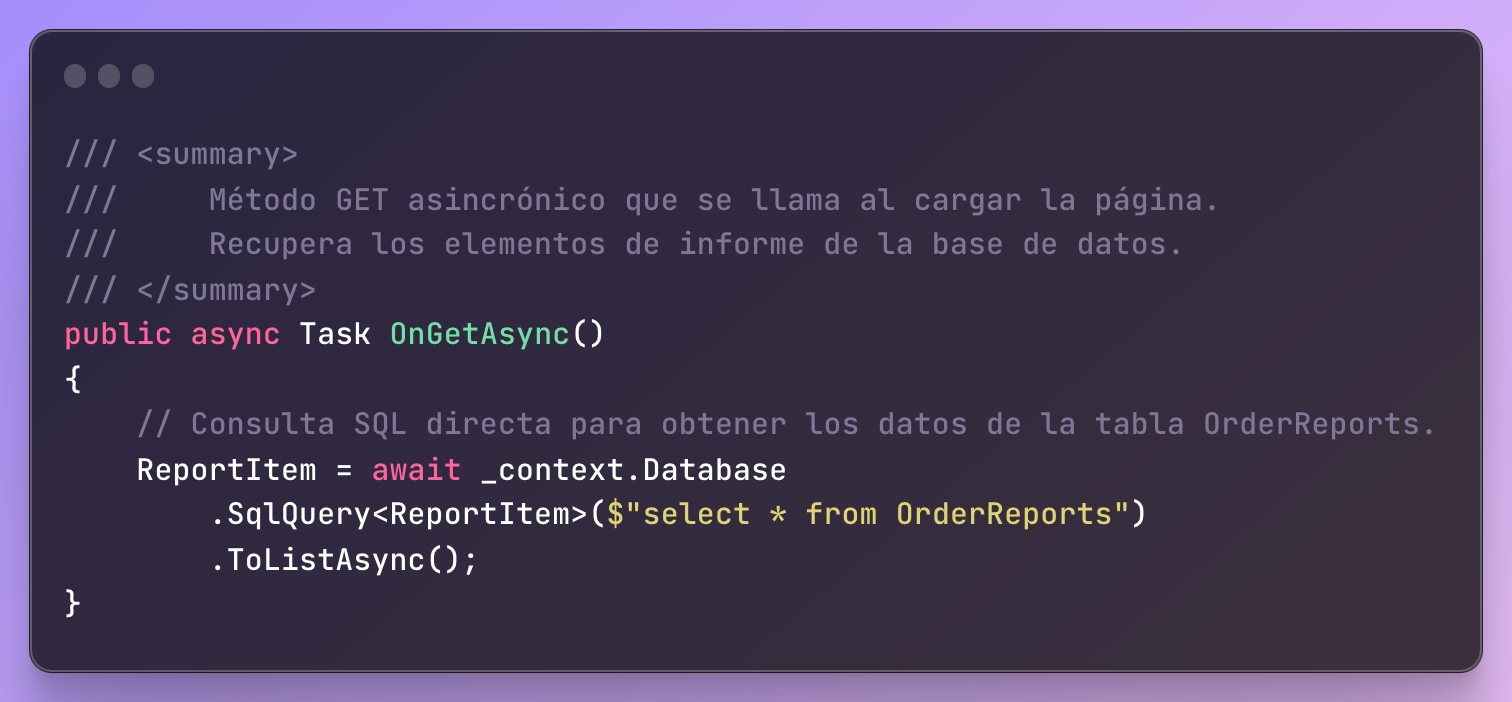
Ejecutaremos el contenido del archivo script.sql en la base de datos SQL Server de nuestra aplicación. Este paso es esencial para crear una nueva tabla y un procedimiento almacenado (SP). En este ejercicio, simularemos un escenario en el cual, como parte de un requerimiento específico, debemos generar reportes de las órdenes de la aplicación y almacenarlos en una tabla. Esta tabla, aunque no forma parte integral de nuestra aplicación, es un recurso auxiliar crucial para el almacenamiento de dichos reportes.

Lanzaremos nuestra aplicación /Report donde ejecutaremos la única acción que está definida en el botón “Regenerar el reporte”. Al hacerlo, la aplicación actualizará su estado y presentará los nuevos datos generados.



Los datos que alimentan esta aplicación provienen de la consulta realizada a la tabla OrderReports. Gracias al contexto de la base de datos, podemos ejecutar consultas SQL nativas utilizando la función SqlQuery, la cual admite una cadena de texto con el comando SQL a ejecutar.

Una característica destacada de este método es su capacidad para aceptar un FormattableString en lugar de una cadena de texto primitiva. FormattableString permite la interpolación de valores en la consulta, convirtiéndolos automáticamente en parámetros. Esto refuerza la seguridad de los comandos SQL que se ejecutan, protegiéndo contra ataques de inyección SQL.



Por otro lado, la función que activa el botón para generar el reporte opera de manera similar a la consulta, pero utiliza un método que no devuelve valores de la consulta. En cambio, este método se limita a ejecutar un comando SQL. En este caso concreto, se utiliza para llamar al procedimiento almacenado (SP) que hemos definido previamente.

