3.Setup and Theory - Docker and Onion Architecture-ES

Preparación y teoría: Docker y la Onion Architecture

#aprendizaje

Este capítulo aborda dos componentes fundamentales de las arquitecturas modernas de microservices, que se utilizarán en la mayoría de los ejemplos del libro:

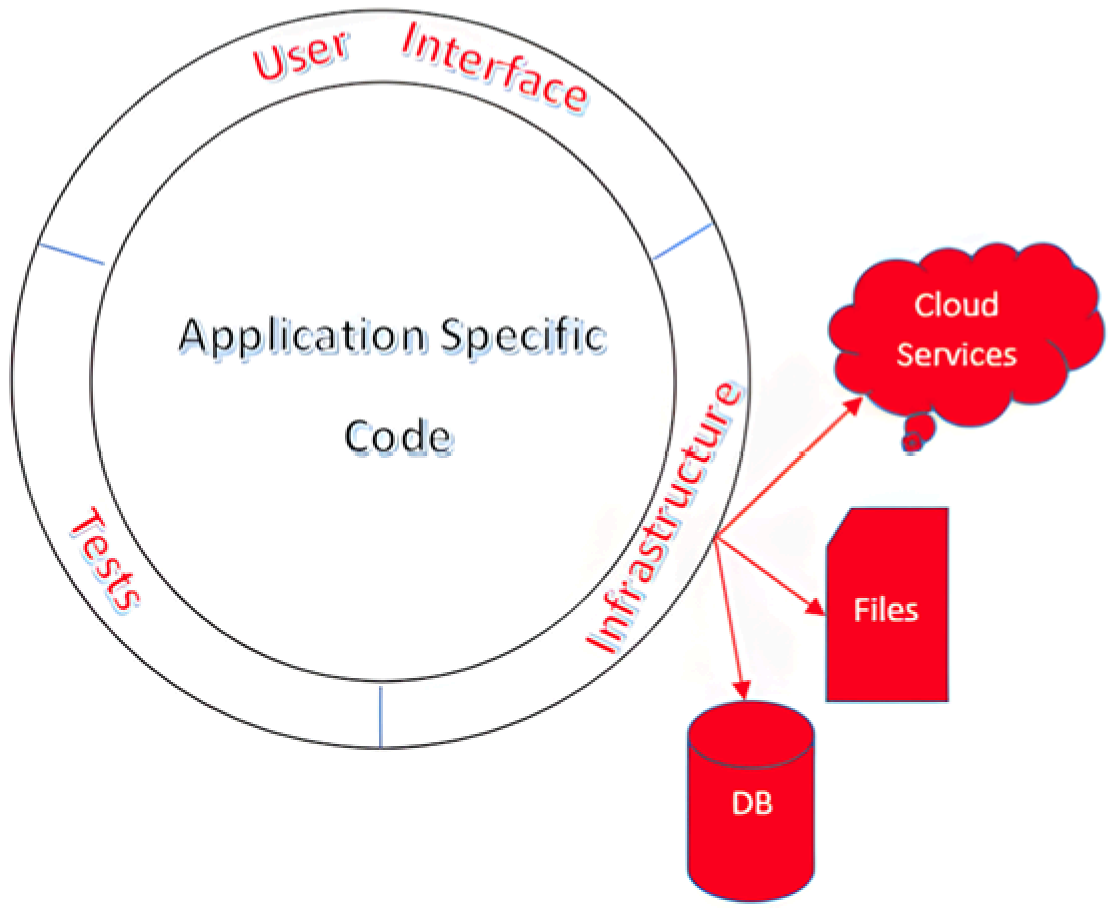
- **Docker Containers**: Los Docker containers son una herramienta de virtualización que permite ejecutar microservices en una amplia variedad de plataformas de hardware, evitando problemas de compatibilidad.
- **La Onion Architecture**: La Onion Architecture confina las dependencias tanto de la **interfaz de usuario** (**UI**) como de la plataforma de despliegue en drivers, de modo que los módulos de software que codifican todo el conocimiento del negocio sean completamente independientes de la UI, herramientas y entorno de ejecución elegidos. Además, para optimizar la interacción entre expertos del dominio y desarrolladores, todas las entidades del dominio se implementan como clases de la siguiente manera:
    1. Cada entidad interactúa con el resto del código únicamente a través de métodos que representan el comportamiento de todas las entidades del dominio real.
    2. Los nombres de las entidades y sus miembros provienen del vocabulario del dominio de la aplicación. El propósito es construir un lenguaje común entre desarrolladores y usuarios llamado **ubiquitous language**.

Mientras que los Docker containers están ligados principalmente a la optimización del rendimiento de microservices, la Onion Architecture no es específica de microservices. Sin embargo, la Onion Architecture descrita aquí fue diseñada específicamente para su uso con microservices, ya que hace un uso extensivo de algunos de los patrones específicos de microservices que describimos en el *Capítulo 2, Desmitificación de las aplicaciones de microservices*, como los eventos publisher-subscriber, para maximizar la independencia de los módulos de software y asegurar la separación entre módulos de actualización y de consulta.

La Onion Architecture

La Onion Architecture establece una distinción clara entre el código específico del dominio y el código técnico que maneja la UI, la interacción con el almacenamiento y los recursos de hardware. Esto mantiene el código específico del dominio completamente independiente de las herramientas técnicas, como el sistema operativo, la tecnología web, la base de datos y las herramientas de interacción con la base de datos.

Toda la aplicación se organiza en capas, donde la capa más externa tiene el único propósito de proveer toda la infraestructura necesaria (es decir, drivers), la UI y las suites de pruebas, como se muestra en la siguiente figura:



3.1-Basic Onion Architecture

A su vez, el código específico de la aplicación se organiza en varias capas anidadas adicionales. Todas las capas deben satisfacer la siguiente restricción:

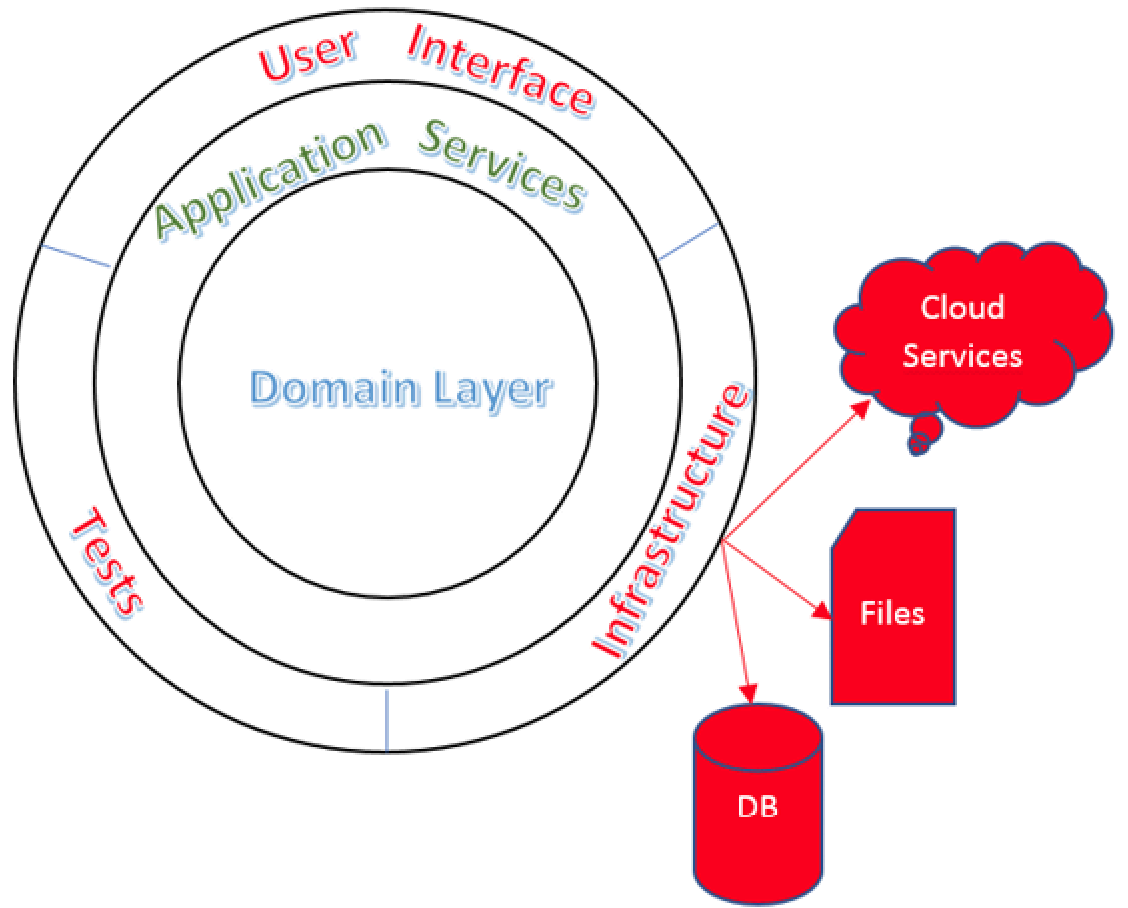
> **Important**
>
> Cada capa solo puede referenciar capas internas. La forma en que se implementa esta restricción depende del lenguaje y la tecnología subyacente. Por ejemplo, las capas pueden implementarse como paquetes, namespaces o bibliotecas. Nosotros implementaremos las capas con proyectos de biblioteca .NET que también pueden convertirse fácilmente en paquetes NuGet.

Así, por ejemplo, en la figura anterior, la capa más externa puede referenciar todas las bibliotecas específicas de la aplicación, más todas las bibliotecas que implementan los drivers requeridos.

El código específico de la aplicación referencia las funcionalidades implementadas en los drivers de la capa más externa a través de interfaces, mientras que la capa más externa tiene la función principal de proveer un motor de inyección de dependencias que acopla cada una de estas interfaces con un driver que la implementa:

```
builder.Services.AddScoped<IMyFunctionalityInterface1,
MyFunctionalityImplementation1>();
builder.Services.AddScoped<IMyFunctionalityInterface2,
MyFunctionalityImplementation2>();
```

La capa específica de la aplicación, a su vez, se compone de al menos dos capas principales: una capa que contiene todas las definiciones de entidades del dominio, llamada **capa de Dominio** (Domain layer), y una capa que contiene la definición de todas las operaciones de la aplicación, llamada capa de **Application Services**, como se muestra en la siguiente figura:



3.2-Complete Onion Architecture

Si es necesario, la capa de Application Services puede dividirse en más subcapas, y se pueden colocar más capas entre las capas de Application Services y de Dominio, pero esto rara vez se hace.

La capa de Dominio se divide frecuentemente en dos subcapas: **la capa de Modelo** (Model layer), que contiene las definiciones reales de las entidades del dominio, y la capa de **Domain Services**, que contiene reglas de negocio adicionales.

La capa de Dominio

La capa de Dominio contiene la representación en clases de cada entidad del dominio con su comportamiento codificado en los métodos públicos de dichas clases.

Además, las entidades del dominio solo pueden modificarse mediante métodos que representan operaciones reales del dominio. Así, por ejemplo, no podemos acceder directamente y modificar todos los campos de una orden de compra; estamos limitados a manipularla únicamente a través de métodos que representan operaciones reales del dominio, como agregar o eliminar un ítem, aplicar un descuento o modificar la fecha de entrega.

Los nombres de todos los métodos y propiedades públicas deben construirse con el lenguaje real utilizado por los expertos del dominio, el ya mencionado **ubiquitous language**.

Todas las restricciones anteriores tienen el propósito de optimizar la comunicación entre desarrolladores y expertos. De esta manera, los expertos del dominio y los desarrolladores pueden discutir la interfaz pública de la entidad ya que utiliza el mismo vocabulario y las operaciones reales del dominio.

A continuación se muestra una parte de una entidad hipotética `PurchaseOrder`:

```
public class PurchaseOrder
    {
      …
      #region private members
      private IList<PurchaseOrderItem> items;
      private DateTime _deliveryTime;
```

```
    #endregion
    public PurchaseOrder(DateTime creationTime, DateTime deliveryTime)
      {
        CreationTime = creationTime;
        _deliveryTime = deliveryTime;
        items=new List<PurchaseOrderItem>();
      }
    public DateTime CreationTime {get; init;}
    public DateTime DeliveryTime => _deliveryTime;
    public IEnumerable<PurchaseOrderItem> Items => items;
    public bool DelayDelyveryTime(DateTime newDeliveryTime)
      {
        if(_deliveryTime< newDeliveryTime)
          {
            _deliveryTime = newDeliveryTime;
            return true;
          }
        else return false;
      }
    public void AddItem (PurchaseOrderItem x)
      { items.Add(x); }
    public void RemoveItem(PurchaseOrderItem x)
      { items.Remove(x); }
     …
}
```

Una vez asignado desde el constructor, `CreationTime` no puede modificarse más, por lo que se implementa como una propiedad `{get; init;}`. La lista de ítems puede modificarse a través de los métodos `AddItem` y `RemoveItem`, que son comprensibles para todos los expertos del dominio. Finalmente, podemos retrasar la fecha de entrega pero no adelantarla. Esto codifica automáticamente una regla de negocio del dominio al imponer el uso del método `DelayDeliveryTime`.

Podemos mejorar la entidad `PurchaseOrder` agregando una **propiedad get** `PurchaseTotal` que retorne el monto total de la compra, y agregando un método `ApplyDiscount`.

En resumen, podemos establecer la siguiente regla:

> **Important**
>
> Los estados de las entidades del dominio solo pueden cambiarse a través de métodos que codifican operaciones reales del dominio y que automáticamente hacen cumplir todas las reglas de negocio.

Estas entidades difieren mucho de las entidades habituales de **Entity Framework Core** a las que estamos acostumbrados, por las siguientes razones:

- Las entidades de Entity Framework Core son clases similares a records sin métodos. Es decir, son simplemente un conjunto de pares propiedad-valor.
- Cada entidad de Entity Framework Core corresponde a un único objeto relacionado de alguna manera con otras entidades, mientras que las entidades del dominio son frecuentemente árboles de objetos anidados. Por eso las entidades del dominio suelen llamarse **aggregates**.

Así, por ejemplo, el aggregate `PurchaseOrder` contiene una entidad principal y una colección de `PurchaseOrderItem`. Vale la pena señalar que `PurchaseOrderItem` no puede

considerarse una entidad del dominio separada ya que no existen operaciones del dominio que involucren un único `PurchaseOrderItem`, sino que `PurchaseOrderItem` solo puede manipularse como parte de `PurchaseOrder`.

Un fenómeno similar no ocurre con las entidades planas de Entity Framework, ya que carecen del concepto de operaciones del dominio. Podemos concluir lo siguiente:

> **Tip**
>
> Las operaciones del dominio sobre entidades del dominio pueden forzarlas a fusionarse con entidades dependientes, convirtiéndose así en un árbol complejo de objetos llamado **aggregates**.

Para el resto de este libro, nos referiremos a las entidades del dominio como aggregates.

Hasta ahora, hemos dado a las entidades una fuerte semántica del dominio de aplicación junto con el concepto de agregación. Estos aggregates difieren mucho de las tuplas de base de datos y también de su representación como objetos proporcionada por ORMs como Entity Framework Core, por lo que existe un desajuste entre los aggregates y las estructuras utilizadas para persistirlos. Este desajuste podría resolverse de varias maneras, pero todas las soluciones deben conformarse al principio de **persistence ignorance** (ignorancia de la persistencia):

> **Tip**
>
> Los aggregates no deben verse afectados por la forma en que podrían persistirse. Deben estar completamente desacoplados del código de persistencia, y la técnica de persistencia no debe imponer ninguna restricción sobre el diseño del aggregate.

Ahora observamos otro fenómeno: ¡entidades sin identidad!

Dos órdenes de compra con exactamente las mismas fechas e ítems siguen siendo dos entidades diferentes; de hecho, deben tener una entrega diferente para cada una.

Sin embargo, ¿qué ocurre con dos direcciones que contienen exactamente los mismos campos? Si consideramos la semántica de una dirección, ¿podemos decir que son dos entidades diferentes?

Cada dirección denota un lugar, y si dos direcciones tienen los mismos campos, denotan exactamente el mismo lugar. Así, las direcciones son como los números: aunque las repliquemos varias veces, cada copia siempre denota la misma entidad abstracta.

Por lo tanto, podemos concluir que las direcciones con los mismos campos son indistinguibles. Las bases de datos relacionales usan claves principales para verificar cuándo dos tuplas referencian la misma entidad abstracta, así que podemos concluir que la clave principal de una dirección debería ser el conjunto de todos sus campos.

En la teoría de entidades del dominio, los objetos similares a las direcciones se llaman value objects, y su representación en memoria no debe contener claves principales explícitas. Un operador de igualdad aplicado a dos instancias de ellos debe retornar `true` si y solo si todos sus campos son iguales. Además, deben ser inmutables — es decir, una vez creados, sus propiedades no pueden cambiarse, por lo que la única forma de modificar un value object es crear un nuevo objeto con algún valor de propiedad cambiado.

En C#, los value objects se representan fácilmente con **records**:

```
public record Address
{
  public string Country {get; init;}
  public string Town {get; init;}
  public string Street {get; init;}
}
```

La palabra clave `init` es lo que hace inmutables las propiedades de tipo record, ya que significa que solo pueden inicializarse. Una copia modificada de un record puede crearse de la siguiente manera:

```
var modifiedAddress = myAddress with {Street = "new street"};
```

Si pasamos todas las propiedades en el constructor en lugar de usar inicializadores, la definición anterior puede simplificarse así:

```
public record Address(string Country, string Town, string Street) ;
```

Los value objects típicos incluyen costos (representados como un número y un símbolo de moneda), ubicaciones (representadas como longitud y latitud), direcciones e información de contacto.

En la práctica, los value objects pueden representarse en bases de datos con la tupla habitual con clave principal (por ejemplo, un entero autoincrementado). Entonces, se puede crear una nueva copia de cada tupla de forma diferente para cada ocurrencia de la misma dirección. También es posible imponer una copia única en la base de datos definiendo claves compuestas complejas.

> **Tip**
>
> Dado que los aggregates y los value objects difieren mucho de las entidades utilizadas por los principales ORMs como Entity Framework, cuando usamos ORMs para interactuar con bases de datos, debemos traducir las entidades del ORM a aggregates y value objects, y viceversa, cada vez que intercambiamos datos con un ORM.

Según las reglas generales de la Onion Architecture, la capa de Dominio interactúa con la implementación real proporcionada por un ORM a través de una interfaz. Esto se hace habitualmente con el llamado **repository pattern** (patrón repositorio).

> **Important**
>
> Según el repository pattern, un servicio de almacenamiento debe proporcionarse a través de una interfaz separada para cada aggregate.

Esto significa que la capa de Dominio debe contener una interfaz diferente para cada aggregate, que se encargue de recuperar, guardar y eliminar el aggregate completo. El repository pattern ayuda a mantener el código modular y fácil de buscar y actualizar, ya que sabemos que debemos tener una y solo una interfaz de repositorio para cada aggregate, por lo que podemos organizar todo el código relacionado con el aggregate en una sola carpeta.

La implementación real de cada repositorio se encuentra en la capa de Infraestructura de la Onion Architecture, en una especie de driver de base de datos (o persistencia), junto con otros

drivers que virtualizan la interacción con la infraestructura.

Cada interfaz de repositorio de aggregate contiene métodos que retornan aggregates, eliminan aggregates y realizan cualquier otro tipo de operación relacionada con la persistencia de aggregates.

En aplicaciones complejas, es una buena práctica dividir la capa de Dominio en una capa de Modelo, que contiene solo aggregates, y una capa exterior de Domain Services, que contiene las interfaces de repositorio y la definición de operaciones del dominio que no pueden implementarse como métodos del aggregate.

En particular, las interfaces de Domain Services manejan las tuplas utilizadas para codificar los resultados devueltos por los microservices de consulta. Estas tuplas no son aggregates sino una mezcla de datos tomados de diferentes tablas, por lo que se ajustan a un patrón de diseño completamente diferente. Se devuelven como objetos similares a records sin métodos y solo con propiedades que corresponden a los campos de las tuplas de la base de datos. Las interfaces adicionales de Domain Services también se implementan en el driver de persistencia de la capa de infraestructura.

Manejar consultas y modificaciones por separado y con diferentes patrones de diseño se conoce como el patrón Command Query Responsibility Segregation (CQRS).

> **Important**
>
> Dado que los microservices descritos en este libro son bastante simples, en nuestros ejemplos de código no dividiremos la capa de dominio en las capas de modelo y domain services. Por lo tanto, las interfaces de repositorio y otras interfaces de domain services estarán mezcladas con los aggregates en el mismo proyecto de Visual Studio. Sin embargo, al implementar aplicaciones más complejas, se debería usar la división de la capa de dominio en las capas de modelo y domain services.

Veamos algunos ejemplos de una interfaz de repositorio. El aggregate `PurchaseOrder` podría tener una interfaz de repositorio asociada con el siguiente aspecto:

```
public interface IPurchaseOrderRepository
   {
     PurchaseOrder New(DateTime creationTime, DateTime deliveryTime);
     Task<PurchaseOrder> GetAsync(long id);
     Task DeleteAsync(long id);
     Task DeleteAsync(PurchaseOrder order);
     Task<IEnumerable<OrderBasicInfoDTO>> GetMany(DateTime? startPeriod,
           DateTime? endPeriod, int? customerId
           );
        ...
   }
```

No hay un método de actualización ya que las actualizaciones se implementan llamando directamente a los métodos del aggregate. El último método del código mostrado retorna una colección de DTOs similares a records llamados `OrderBasicInfoDTO`.

> Vale la pena señalar que no existen interfaces de repositorio asociadas con value objects, ya que los value objects se manejan igual que los tipos primitivos, como enteros, decimales o strings.

Varios cambios a diferentes aggregates pueden manejarse de forma transaccional gracias al patrón **Unit Of Work**, que se describirá más adelante en la subsección *Command*.

Habiendo comprendido la representación en memoria de los objetos del dominio, podemos pasar a la forma en que una Onion Architecture orientada a microservices representa todas las transacciones/operaciones del negocio.

Application services

En la subsección *Organización de microservices* del *Capítulo 2, Desmitificación de las aplicaciones de microservices*, vimos que las arquitecturas de microservices frecuentemente utilizan el patrón **CQRS**, en el cual algunos microservices se especializan en consultas y otros en actualizaciones. Esta es la versión fuerte del patrón CQRS, pero también existe una versión más débil que simplemente requiere que las consultas y las actualizaciones se organicen en módulos diferentes, posiblemente pertenecientes al mismo microservice.

Aunque no siempre es conveniente aplicar CQRS en su forma fuerte, su forma débil es imprescindible al implementar microservices, ya que las actualizaciones involucran aggregates mientras que las consultas involucran solo DTOs similares a records, por lo que requieren tipos de procesamiento completamente diferentes.

En consecuencia, las operaciones definidas en la capa de application services de un microservice se dividen en dos tipos diferentes: **queries** (consultas) y **commands** (comandos). Como veremos, la ejecución de commands puede disparar eventos, por lo que junto con commands y queries, la capa de application services también debe manejar los llamados **domain events** (eventos del dominio). Discutiremos todas estas operaciones en las subsecciones dedicadas que siguen.

Queries

Un objeto query representa una o varias consultas similares, por lo que usualmente tiene uno o varios métodos que reciben algunos inputs y retornan los resultados de la consulta. La mayoría de los métodos de query simplemente llaman a un único método del repositorio que implementa la consulta necesaria, pero en algunos casos pueden ejecutar varios métodos del repositorio y luego combinar sus resultados de alguna manera.

Durante las pruebas del sistema, las implementaciones reales de queries deben reemplazarse por implementaciones falsas, por lo que usualmente cada query tiene una interfaz asociada que se acopla con la implementación real en el motor de inyección de dependencias. De esta manera, la UI puede simplemente requerir la interfaz en algún constructor, habilitando así las pruebas con una implementación falsa de la query.

La siguiente es una posible definición de una query que retorna todas las órdenes de compra emitidas después de una fecha dada, junto con su interfaz asociada:

```
public interface IPurchaseOrderByStartDateQuery: IQuery
{
  Task<IEnumerable<OrderBasicInfoDTO>> Execute(DateTime startDate);
}
public class PurchaseOrderByStartDateQuery(IPurchaseOrderRepository repo):
  IPurchaseOrderByStartDateQuery
```

```
{
  public async Task<IEnumerable<OrderBasicInfoDTO>> Execute(DateTime
startDate)
  {
    return await repo.GetMany(startDate, null, null);
  }
}
```

La interfaz hereda de una interfaz vacía cuyo único propósito es marcar tanto la interfaz como su implementación como queries. De esta manera, todas las queries y su implementación asociada pueden encontrarse automáticamente con la ayuda de reflection y agregarse al motor de inyección de dependencias. Proporcionaremos el código que descubre todas las queries en la sección *Una plantilla de solución basada en la Onion Architecture* junto con una plantilla de solución completa.

Como se mencionó, la implementación simplemente llama a un método del repositorio y le pasa los parámetros adecuados. Una implementación del repositorio se pasa en el constructor principal de la clase por el mismo motor de inyección de dependencias que inyectará la query misma en el constructor de un objeto de la capa de presentación (un controller, en el caso de un sitio web ASP.NET Core).

Commands

Los commands funcionan de una manera ligeramente diferente porque, para una mejor legibilidad del código, cada command representa una única operación de la aplicación. Por esta razón, cada instancia de command representa tanto la operación abstracta como su input. La implementación real de la operación está contenida en un objeto command handler. El siguiente es el código de un command hipotético que aplica un descuento a una orden de compra:

```
public record ApplyDiscountCommand(decimal discount, long orderId):
ICommand;
```

Los commands deben ser inmutables; por eso los implementamos como records. De hecho, la única operación permitida sobre ellos es su ejecución. De manera similar a las queries, los commands también implementan una interfaz vacía que los marca como commands (en este caso, `ICommand`).

Los command handlers son implementaciones de la siguiente interfaz:

```
    public interface ICommandHandler {}
    public interface ICommandHandler<T>: ICommandHandler
        where T: ICommand
    {
      Task HandleAsync(T command);
    }
```

Como se puede ver, todos los command handlers implementan el mismo método `HandleAsync` que acepta el command como su único input. Así, por ejemplo, el handler asociado con `ApplyDiscountCommand` es algo como la siguiente clase:

```
public class ApplyDiscountCommandHandler(
IPackageRepository repo):ICommandHandler<ApplyDiscountCommand>
  {
    public async Task HandleAsync(ApplyDiscountCommand command)
```

```
      {
        var purchaseOrder = await repo.GetAsync(command.OrderId);
        //llamar a los métodos adecuados del aggregate para aplicar la
actualización requerida
        //posiblemente modificar otros aggregates obteniéndolos con otros
        //repositorios inyectados
        ...
      }
    }
```

Todos los handlers deben agregarse al motor de inyección de dependencias, como se muestra en el siguiente ejemplo:

```
builder.Services.AddScoped<ICommandHandler<ApplyDiscountCommand>,
 ApplyDiscountCommandHandler>();
```

Esto puede hacerse automáticamente escaneando el assembly de application services con reflection. Proporcionaremos el código que descubre todos los command handlers en la sección *Una plantilla de solución basada en la Onion Architecture*.

Cada command handler obtiene o crea aggregates, los modifica llamando a sus métodos, y luego ejecuta una instrucción de guardado para persistir todas las modificaciones en el almacenamiento subyacente.

La operación de guardado debe implementarse en el driver de almacenamiento (por ejemplo, Entity Framework Core), por lo que, como es habitual para todas las operaciones de drivers de la Onion Architecture, está mediada por una interfaz. La interfaz que realiza las operaciones de guardado y otras operaciones relacionadas con transacciones se llama usualmente **IUnitOfWork**. Una posible definición de esta interfaz es la siguiente:

```
public interface IUnitOfWork
  {
    Task<bool> SaveEntitiesAsync();
    Task StartAsync();
    Task CommitAsync();
    Task RollbackAsync();
  }
```

Desglosemos esto:

- `SaveEntitiesAsync` guarda todas las actualizaciones realizadas hasta el momento en una única transacción. Retorna `true` si el motor de almacenamiento efectivamente cambió después de la operación de guardado, y `false` en caso contrario.
- `StartAsync` inicia una transacción.
- `CommitAsync` y `RollbackAsync` respectivamente confirman y revierten una transacción abierta.

Todos los métodos que controlan explícitamente el inicio y fin de una transacción son útiles para encerrar tanto una operación de obtención como el guardado final con `SaveEntitiesAsync` en la misma transacción, como en el siguiente fragmento simplificado de reserva de vuelo:

```
await unitOfWork.StartAsync();
var flight = await repo.GetFlightAsync(flightId);
flight.Seats--;
```

```
if(flight.Seats < 0)
{
  await unitOfWork.RollBackAsync();
  return;
}
...
await unitOfWork.SaveEntitiesAsync();
await unitOfWork.CommitAsync();
```

Si no hay más asientos disponibles, la transacción se aborta, pero si hay asientos disponibles, tenemos la certeza de que ningún otro pasajero puede tomar el asiento disponible porque tanto la consulta como la actualización se realizan en la misma transacción, evitando así la interferencia de otras operaciones de reserva.

Por supuesto, el código anterior funciona si la transacción tiene un nivel de aislamiento adecuado y si la base de datos soporta ese nivel de aislamiento. Podemos usar un nivel de aislamiento lo suficientemente alto para todas las operaciones en nuestro microservice; de lo contrario, nos vemos obligados a pasar el nivel de aislamiento como argumento de `StartAsync`.

Domain events

Podemos definir los **domain events** de la siguiente manera:

> **Important**
>
> Los **domain events** son eventos que se originan a partir de algo que ocurre en el dominio del microservice y se manejan dentro de los límites del propio microservice. Esto significa que involucran comunicaciones basadas en el patrón publisher-subscriber entre dos fragmentos de código del mismo microservice.

Por lo tanto, no deben confundirse con los eventos involucrados en las comunicaciones entre diferentes microservices, que se llaman **integration events** para distinguirlos de los domain events.

¿Por qué usar eventos dentro de los límites de un microservice? La razón es siempre la misma: asegurar un mejor desacoplamiento entre las partes. Aquí, las partes involucradas son los aggregates. El código de cada aggregate debe ser completamente independiente de otros aggregates para asegurar la modularidad y la modificabilidad, por lo que las relaciones entre aggregates están mediadas por command handlers o por algún patrón publisher-subscriber.

En consecuencia, si la interacción entre dos aggregates es de alguna manera decidida por el código de un command handler, el mismo command handler podría encargarse de procesar los datos de ambos y luego actualizarlos de alguna manera. Sin embargo, si la interacción está ligada al procesamiento dentro de un método del aggregate, nos vemos obligados a usar eventos porque no podemos hacer que un aggregate sea consciente de todos los otros aggregates que necesitan ser informados sobre algunos de sus cambios de datos. En resumen, podemos establecer el siguiente principio:

> **Important**
>
> Los domain events se disparan únicamente dentro de los métodos del aggregate porque otros tipos de interacciones se manejan mejor mediante el código de los command handlers.

Otro principio importante es el siguiente:

> **Important**
>
> Los eventos disparados dentro de un método del aggregate no deben interferir con el procesamiento en curso del método, ya que esto podría socavar el contrato entre el aggregate y los command handlers que lo manipulan.

En consecuencia, cada aggregate almacena todos los eventos dentro de sí mismo en una lista de eventos, y luego el command handler decide cuándo ejecutar estos handlers. Típicamente, todos los eventos de todos los aggregates procesados por un command handler se ejecutan justo antes de que el handler guarde todos los cambios llamando a `unitOfWork.SaveEntitiesAsync()`. Sin embargo, esta no es una regla general.

Los eventos se manejan de manera similar a los commands, con la única diferencia de que cada command tiene un solo handler asociado, mientras que cada evento puede tener varias suscripciones adjuntas. Afortunadamente, esta dificultad puede manejarse fácilmente con algunas características avanzadas del motor de inyección de dependencias de .NET.

Más específicamente, los eventos son clases marcadas con la interfaz vacía `IEventNotification`, mientras que los event handlers son una implementación de la siguiente interfaz:

```
public interface IEventHandler
{
}
public interface IEventHandler<T>: IEventHandler
    where T: IEventNotification
  {
    Task HandleAsync(T ev);
  }
```

Todas las estructuras de datos involucradas son completamente análogas a las necesarias para manejar commands. Sin embargo, ahora debemos agregar algunas mejoras para asociar cada evento con todos sus handlers. La siguiente clase genérica hace el trabajo:

```
public class EventTrigger<T>
        where T: IEventNotification
  {
    private readonly IEnumerable<IEventHandler<T>> _handlers;
    public EventTrigger(IEnumerable<IEventHandler<T>> handlers)
      {
        _handlers = handlers;
      }
      public async Task Trigger(T ev)
      {
        foreach (var handler in _handlers)
          await handler.HandleAsync(ev);
      }
  }
```

Aquí, `IEventNotification` es una interfaz vacía usada solo para marcar una clase como representación de un evento.

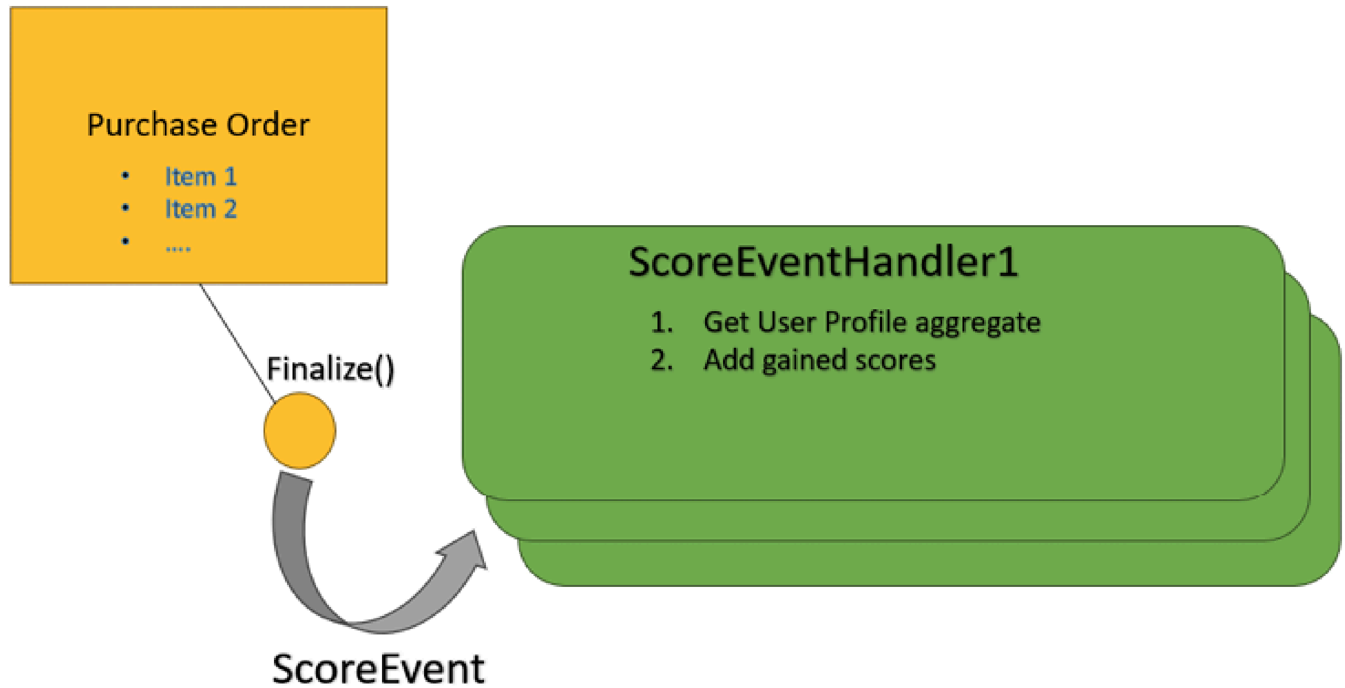
Si agregamos la clase genérica anterior al motor de inyección de dependencias con `service.AddScoped(typeof(EventTrigger<>))`, entonces cada vez que requiramos una instancia específica de esta clase (digamos, para el argumento genérico del evento `MyEvent`), el motor de inyección de dependencias recuperará automáticamente todas las implementaciones de `IEventHandler<MyEvent>` y las pasará en el constructor de la instancia `EventTrigger<MyEvent>` que se retorna. Después de eso, podemos lanzar todos los handlers suscritos con algo como lo siguiente:

```
public class MyCommandHandler(EventTrigger<MyEvent> myEventHandlers): …
{
  public async Task HandleAsync(MytCommand command)
  {
    …
    await myEventHandlers.Trigger(myEvent)
    …
   }
}
```

Vale la pena señalar que la interfaz `IEventNotification` debe definirse en la capa de dominio ya que debe usar aggregates, mientras que todas las demás interfaces y clases conectadas con eventos se definen en la DLL de application services.

Como ejemplo de un evento, consideremos un aggregate de orden de compra de una aplicación de e-commerce. Cuando la orden de compra se finaliza llamando a su método `Finalize`, si la compra es mayor que un umbral dado, entonces debe crearse un evento para agregar algunos puntos a los perfiles de usuario que el usuario puede gastar para obtener descuentos en compras futuras.

La siguiente figura ejemplifica lo que ocurre:



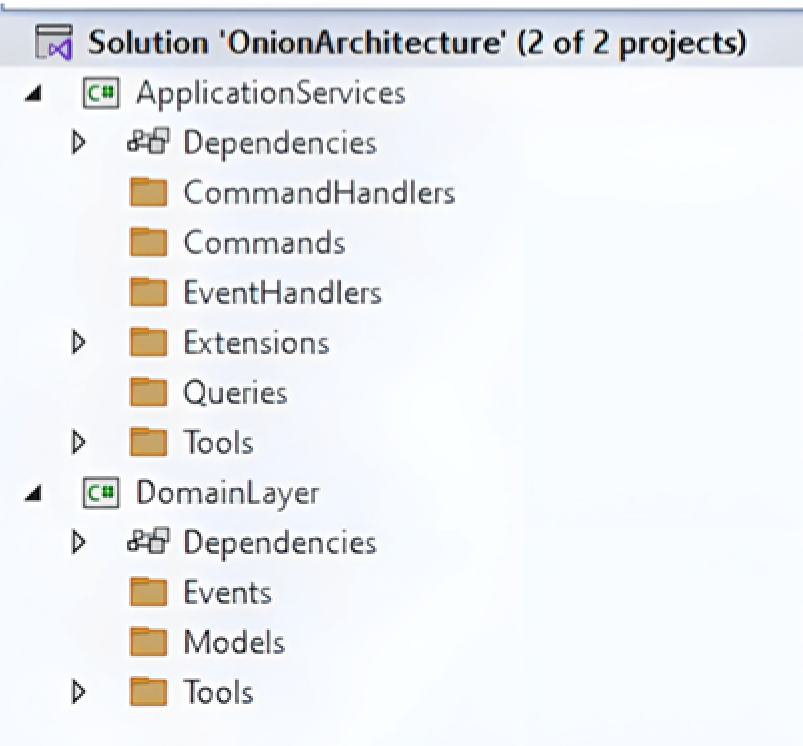
3.3-Domain event example

Al igual que en el caso de los command handlers, todos los event handlers definidos en la DLL de application services pueden descubrirse automáticamente y agregarse al motor de inyección de dependencias mediante reflection. Mostraremos cómo hacerlo en la siguiente sección, que propondrá una plantilla de solución .NET general para la Onion Architecture.

Una plantilla de solución basada en la Onion Architecture

En esta sección, describimos una plantilla de solución basada en la Onion Architecture que usaremos a lo largo del resto del libro, la cual se puede encontrar en la carpeta `ch03` del repositorio GitHub del libro (https://github.com/PacktPublishing/Practical-Serverless-and-Microservices-with-Csharp). Esta plantilla muestra cómo poner en práctica lo aprendido sobre la Onion Architecture.

La solución contiene dos proyectos de biblioteca .NET, llamados `ApplicationServices` y `DomainLayer`, que implementan, respectivamente, las capas de application services y de dominio de una Onion Architecture:



3.4-Solution template based on the Onion Architecture

Como prescribe la Onion Architecture, el proyecto `ApplicationServices` tiene una referencia al proyecto de arquitectura `DomainLayer`.

En `ApplicationServices`, agregamos las siguientes carpetas:

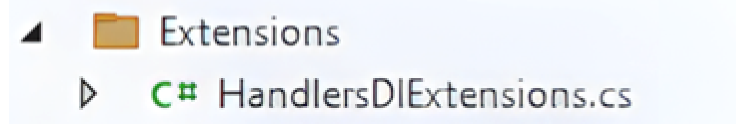
- `Queries` para colocar todas las queries e interfaces de queries
- `Commands` para colocar todas las clases de commands
- `CommandHandlers` para colocar todos los command handlers
- `EventHandlers` para colocar todos los event handlers
- `Tools`, que contiene todas las interfaces relacionadas con la Onion Architecture utilizadas por los application services que describimos en la sección anterior
- `Extensions`, que contiene el método de extensión `HandlersDIExtensions.AddApplicationServices()` que agrega todas las queries, event handlers y command handlers definidos en el proyecto al motor de inyección de dependencias

Todas las carpetas anteriores pueden organizarse en subcarpetas para aumentar la legibilidad del código.

En el proyecto `DomainLayer`, agregamos las siguientes carpetas:

- `Models` para colocar todos los aggregates y value objects
- `Events` para colocar todos los eventos que pueden ser disparados por los aggregates
- `Tools`, que contiene todas las interfaces relacionadas con la Onion Architecture utilizadas por el dominio que describimos en la sección anterior, y algunas clases utilitarias adicionales

La carpeta `Extensions` del proyecto `ApplicationServices` contiene un solo archivo:



3.5-ApplicationServices extensions

La clase estática `HandlersDIExtensions` contiene dos sobrecargas de un método de extensión que agrega todas las queries, command handlers, event handlers y la clase `EventMediator` al motor de inyección de dependencias:

```
public static IServiceCollection AddApplicationServices
  (this IServiceCollection services, Assembly assembly)
{
  AddAllQueries(services, assembly);
  AddAllCommandHandlers(services, assembly);
  AddAllEventHandlers(services, assembly);
  services.AddScoped<EventMediator>();
  return services;
}
public static IServiceCollection AddApplicationServices
  (this IServiceCollection services)
{
  return AddApplicationServices(services,
    typeof(HandlersDIExtensions).Assembly);
}
```

Utiliza tres métodos privados diferentes que escanean el assembly con reflection, buscando respectivamente queries, command handlers y event handlers. El código completo está disponible en la carpeta `ch03` del repositorio GitHub asociado al libro. Aquí, analizamos solo `AddAllCommandHandlers` para mostrar las ideas básicas explotadas por los tres métodos:

```
private static IServiceCollection AddAllCommandHandlers
  (this IServiceCollection services, Assembly assembly)
{
  var handlers = assembly.GetTypes()
    .Where(x => !x.IsAbstract && x.IsClass
      && typeof(ICommandHandler).IsAssignableFrom(x));
  …
```

Primero, recolectamos todas las clases no abstractas que implementan la interfaz vacía `ICommandHandler`. Esta interfaz fue añadida específicamente a todos los command handlers para recuperarlos a todos con reflection. Luego, para cada uno de ellos, recuperamos el `ICommandHandler<T>` que implementa:

```
foreach (var handler in handlers)
{
  var handlerInterface = handler.GetInterfaces()
    .Where(i => i.IsGenericType &&typeof(
      ICommandHandler).IsAssignableFrom(i))
    .SingleOrDefault();
```

Finalmente, si encontramos dicha interfaz, agregamos el par al motor de inyección de dependencias:

```
foreach (var handler in handlers)
{
  …
  if (handlerInterface != null)
  {
    services.AddScoped(handlerInterface, handler);
  }
}
```

La carpeta `Tools` del proyecto `ApplicationServices` contiene los archivos mostrados aquí:

3.6-ApplicationServices tools

Ya analizamos todas las interfaces y clases contenidas en la carpeta `Tools` anterior, excepto `EventMediator`, en la sección previa. Recordémoslas:

- `IQuery` e `ICommand` son interfaces vacías que marcan, respectivamente, queries y commands
- `ICommandHandler<T>` e `IEventHandler<T>` son las interfaces que deben ser implementadas, respectivamente, por command handlers y event handlers
- `EventTrigger<T>` es la clase que hace la magia de recolectar todos los event handlers asociados con el mismo evento, `T`

`EventMediator` es una clase utilitaria que resuelve un problema práctico. Un command handler que necesita disparar todos los event handlers asociados con un evento, `T`, debe inyectar `EventTrigger<T>` en su constructor. Sin embargo, el punto es que un command descubre que necesita disparar el evento `T` solo cuando encuentra el evento `T` en las listas de eventos de un aggregate, por lo que debería inyectar todos los posibles `EventTrigger<T>` en su constructor.

Para superar este problema, la clase `EventMediator` usa `IServiceProvider` para requerir los event handlers asociados con una lista de eventos que se le pasa en su método `TriggerEvents(IEnumerable<IEventNotification> events)`.

En consecuencia, es suficiente con inyectar `EventMediator` en el constructor de cada command handler para que, cada vez que encuentre una lista de eventos no vacía, `L`, en un aggregate, simplemente pueda llamar lo siguiente:

```
await eventMediator.TriggerEvents(L);
```

Una vez que `EventMediator` recibe la llamada anterior, escanea la lista de eventos para descubrir todos los eventos contenidos en ella, luego para cada uno de ellos requiere el `EventTrigger<T>` correspondiente para obtener todos los event handlers asociados, y finalmente ejecuta todos los handlers recuperados, pasándoles los eventos correspondientes.

Para realizar su trabajo, la clase `EventMediator` requiere `IServiceProvider` en su constructor:

```
public class EventMediator
{
  readonly IServiceProvider services;
  public EventMediator(IServiceProvider services)
  {
    this.services = services;
  }
  ...
```

Luego, usa este service provider para requerir cada `EventTrigger<T>` necesario:

```
public async Task TriggerEvents(IEnumerable<IEventNotification> events)
  {
    if (events == null) return;
    foreach(var ev in events)
    {
      var triggerType = typeof(EventTrigger<>).MakeGenericType(
        ev.GetType());
      var trigger = services.GetService(triggerType);
```

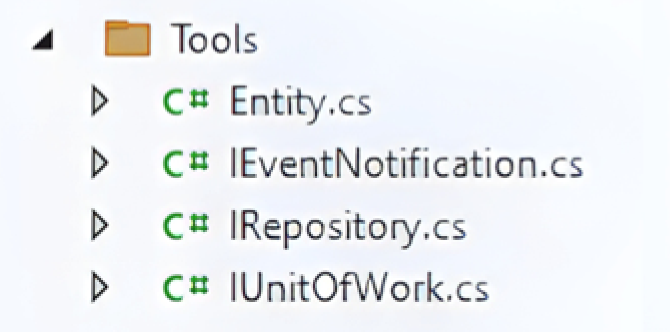
Finalmente, invoca los métodos `EventTrigger<T>.Trigger` con reflection:

```
var task = (Task)triggerType.GetMethod(nameof(
  EventTrigger<IEventNotification>.Trigger))
  .Invoke(trigger, new object[] { ev });
await task.ConfigureAwait(false);
```

El siguiente es el código completo de la clase `EventMediator`:

```
public class EventMediator
{
  readonly IServiceProvider services;
  public EventMediator(IServiceProvider services)
  {
    this.services = services;
  }
  public async Task TriggerEvents(IEnumerable<IEventNotification> events)
  {
    if (events == null) return;
    foreach(var ev in events)
    {
      var triggerType = typeof(EventTrigger<>).MakeGenericType(
        ev.GetType());
      var trigger = services.GetService(triggerType);
      var task = (Task)triggerType.GetMethod(nameof(
        EventTrigger<IEventNotification>.Trigger))
        .Invoke(trigger, new object[] { ev });
      await task;
    }
  }
}
```

La carpeta `Tools` del proyecto `DomainLayer` contiene los siguientes archivos:



3.7-DomainLayer tools

`IEventNotification` e `IRepository` son interfaces vacías que marcan, respectivamente, eventos e interfaces de repositorio. Ya las discutimos en la sección anterior. También discutimos `IUnitOfWork`, que es la interfaz necesaria para que los command handlers persistan cambios y manejen transacciones.

`Entity<T>` es una clase de la que todos los aggregates deben heredar:

```
public abstract class Entity<K>
    where K: IEquatable<K>
{
  public virtual K Id {get; protected set; } = default!;
  public bool IsTransient()
  {
    return Object.Equals(Id, default(K));
  }
  >Domain events handling region
  >Override Equal region
}
```

La clase anterior contiene dos regiones de código minimizadas. El parámetro genérico `K` es el tipo de la clave principal `Id` del aggregate.

El método `IsTransient()` retorna `true` si al aggregate aún no se le ha asignado una clave principal.

La región `Override Equal` contiene el código que sobrescribe el método `Equal` y define los operadores de igualdad y desigualdad. El método `Equal` redefinido considera iguales dos instancias si y solo si tienen la misma clave principal.

La región `Domain events handling` maneja la lista de eventos disparados durante todas las llamadas a los métodos del aggregate. El código expandido se muestra aquí:

```
#region domain events handling
public List<IEventNotification> DomainEvents { get; private set; } = null!;
public void AddDomainEvent(IEventNotification evt)
{
  DomainEvents ??= new List<IEventNotification>();
  DomainEvents.Add(evt);
}
public void RemoveDomainEvent(IEventNotification evt)
{
  DomainEvents?.Remove(evt);
}
#endregion
```

No necesitamos una clase abstracta para los value objects porque, como se discutió en la sección anterior, el tipo `record` de .NET representa perfectamente todas las características de

los tipos de valor.

Antes de discutir con más detalle cómo conectar los dos proyectos de biblioteca de la plantilla con los drivers de almacenamiento reales y con una UI real, necesitamos entender cómo manejar el desajuste entre aggregates y clases ORM similares a records. Lo haremos en la subsección dedicada que sigue.

Correspondencia entre aggregates y entidades ORM

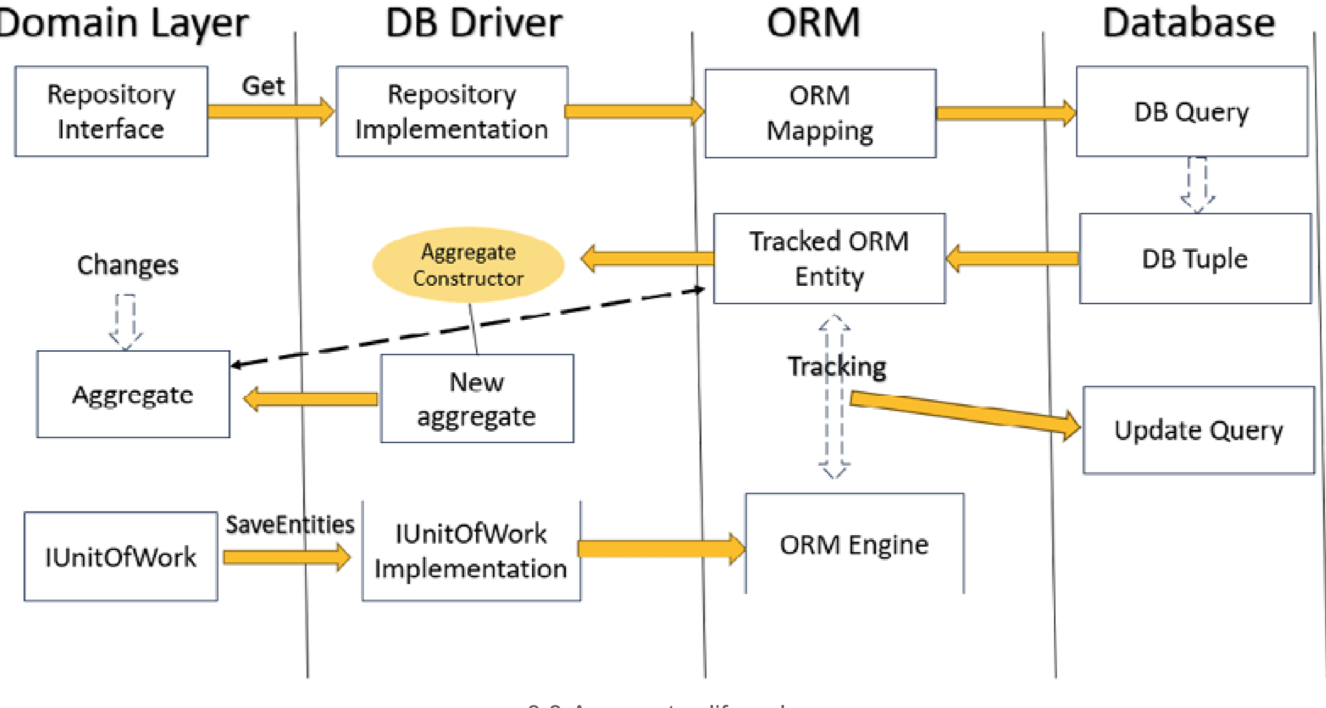
Existen varias técnicas para hacer corresponder entidades ORM y aggregates. La más simple consiste en implementar los aggregates con las propias entidades ORM. La principal dificultad con este enfoque es que los aggregates no exponen las propiedades que deben corresponder a los campos de la base de datos como propiedades públicas. Sin embargo, dado que usualmente las exponen como campos privados, podemos intentar usar estos campos privados para el mapeo de campos de base de datos si el ORM elegido soporta el mapeo con propiedades privadas.

Entity Framework Core soporta el mapeo con campos privados, pero si buscamos independencia completa del driver de base de datos, no podemos depender de esta particularidad de Entity Framework Core. Además, este enfoque nos obliga a definir las entidades ORM en la capa de dominio ya que también son aggregates. Esto significa que no podemos decorar los miembros de la clase con atributos específicos del ORM y que necesitamos preocuparnos por cómo la clase será usada por el ORM mientras definimos cada aggregate, socavando así la independencia de un driver de almacenamiento específico.

Un mejor enfoque es el enfoque de **state object** (objeto de estado):

1. Asociamos cada aggregate con una interfaz que almacena el estado del aggregate en sus propiedades. De esta manera, en lugar de usar campos de respaldo privados, el aggregate usa las propiedades de esta interfaz.
2. La interfaz de estado se pasa en el constructor del aggregate y luego se almacena en una propiedad privada `readonly`.
3. La entidad ORM asociada con el aggregate implementa esta interfaz. De esta manera, el driver de base de datos se adapta a los aggregates y no al revés, logrando así la independencia requerida de la capa de Dominio respecto al driver de base de datos.
4. Cuando la capa de dominio requiere un nuevo aggregate o un aggregate ya almacenado en la base de datos a través de un método de interfaz de repositorio, la implementación del método del repositorio en la base de datos crea o recupera la entidad ORM correspondiente y luego crea un nuevo aggregate, pasando esta entidad ORM en su constructor como un state object.
5. Cuando los aggregates se modifican, todas sus modificaciones se reflejan en sus state objects, que al ser entidades ORM, son rastreados por el ORM. Por lo tanto, cuando instruimos al ORM para guardar todos los cambios, todos los cambios de los aggregates se pasan automáticamente a la base de datos subyacente porque estos cambios están almacenados en objetos rastreados.

La siguiente figura muestra el flujo anterior:



3.8-Aggregates lifecycle

Intentemos modificar nuestro aggregate `PurchaseOrder` anterior usando la siguiente interfaz de estado:

```
public interface IPurchaseOrderState
{
  public DateTime CreationTime { get; set; }
  public DateTime DeliveryTime { get; set; }
  public ICollection<PurchaseOrderItem> Items { get; set; }
  …
}
```

Las modificaciones son directas y no aumentan la complejidad del código:

```
public class PurchaseOrder
{
  private readonly IPurchaseOrderState _state;
  public PurchaseOrder(IPurchaseOrderState state)
  {
    _state = state;
  }
  public DateTime CreationTime => _state.CreationTime;
  public DateTime DeliveryTime => _state.DeliveryTime;
  public IEnumerable<PurchaseOrderItem> Items => _state.Items;
  public bool DelayDelyveryTime(DateTime newDeliveryTime)
  {
    if(_state.DeliveryTime < newDeliveryTime)
    {
      _state.DeliveryTime = newDeliveryTime;
      return true;
    }
    else return false;
    }
    public void AddItem (PurchaseOrderItem x)
    { _state.Items.Add(x); }
    public void RemoveItem(PurchaseOrderItem x)
    { _state.Items.Remove(x); }
}
```

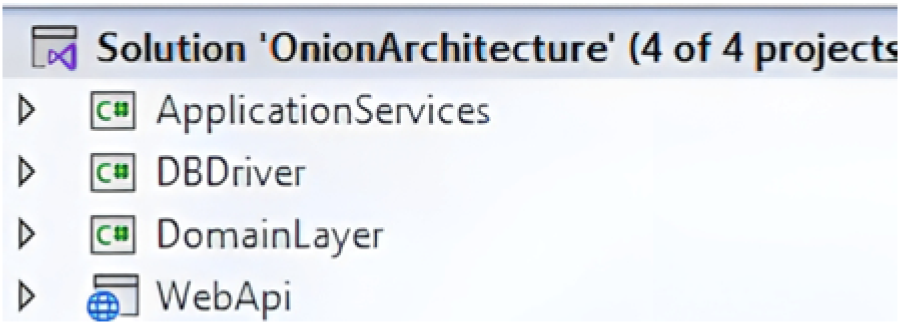
Ahora, estamos listos para entender cómo conectar los dos proyectos de nuestra plantilla con un driver de base de datos real y una UI real.

Una solución completa basada en la Onion Architecture

La carpeta *ch03* del repositorio GitHub del libro (https://github.com/PacktPublishing/Practical-Serverless-and-Microservices-with-Csharp) contiene una solución completa que, junto con las bibliotecas de application services y capa de dominio, también incluye un driver de base de datos basado en Entity Framework Core y una capa de presentación basada en un proyecto ASP.NET Core Web API.

El propósito de este proyecto es mostrar cómo usar la plantilla general de Onion Architecture descrita en esta sección en una solución real.

La siguiente figura muestra la solución completa:



3.9-A complete solution based on the Onion Architecture

El proyecto `DBDriver` es un proyecto de biblioteca .NET donde agregamos una dependencia a los siguientes paquetes NuGet:

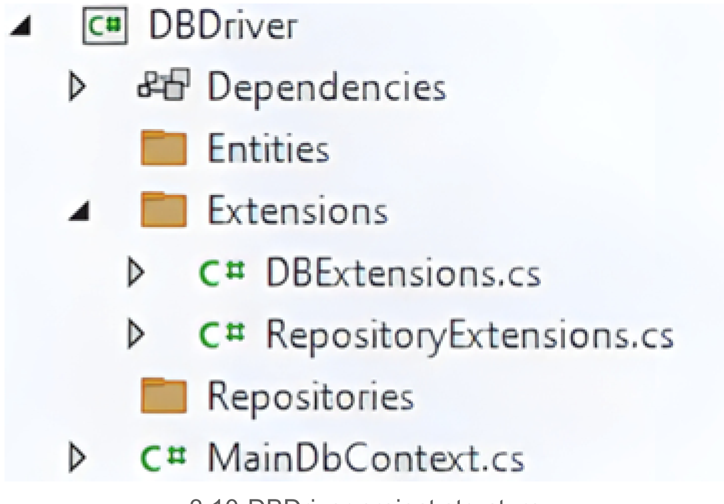
- `Microsoft.EntityFrameworkCore.SqlServer` : Este paquete carga tanto Entity Framework Core como su proveedor de SQL Server
- `Microsoft.EntityFrameworkCore.Tools` : Este paquete proporciona todas las herramientas para scaffolding y manejo de migraciones de base de datos

> **Important**
>
> Dado que el proyecto `DBDriver` debe proporcionar un driver de almacenamiento, también tiene una dependencia del proyecto de biblioteca de dominio.

El proyecto `WebApi` es un proyecto ASP.NET Core Web API. Funciona como la capa más externa de la Onion Architecture.

> **Important**
>
> La capa más externa de la Onion Architecture (en nuestro ejemplo, `WebApi`) debe tener una dependencia del directorio de application services y de todos los proyectos de drivers (en nuestro ejemplo, solo `DBDriver`).

Agregamos algunas carpetas y clases al proyecto `DBDriver` que deberían usarse en todos los drivers basados en Entity Framework Core. La siguiente figura muestra la estructura del proyecto:



3.10-DBDriver project structure

Esta es la descripción de todas las carpetas:

- `Entities`: Coloca aquí todas tus entidades de Entity Framework Core, posiblemente organizadas en subcarpetas.
- `Repositories`: Coloca aquí todas las implementaciones de repositorios, posiblemente organizadas en subcarpetas.
- `MainDbContext`: Este es el esqueleto del DB context de Entity Framework del proyecto, que también contiene la implementación de la interfaz `IUnitOfWork`.
- `Extensions`: Esta carpeta contiene dos clases de extensión. `RepositoryExtensions` simplemente proporciona el método de extensión `AddAllRepositories`, que descubre todas las implementaciones de repositorios y las agrega al motor de inyección de dependencias. Su código es similar al de los métodos de extensión `AddAllCommandHandlers` que describimos en la subsección anterior, por lo que no lo describiremos aquí. `DBExtension` contiene solo el método de extensión `AddDbDriver`, que agrega todas las implementaciones proporcionadas por `DBDriver` al motor de inyección de dependencias.

La implementación del método de extensión `AddDbDriver` es directa:

```
public static IServiceCollection AddDbDriver(
  this IServiceCollection services,
  string connectionString)
{
  services.AddDbContext<IUnitOfWork, MainDbContext>(options =>
    options.UseSqlServer(connectionString,
      b => b.MigrationsAssembly("DBDriver")));
  services.AddAllRepositories(typeof(DBExtensions).Assembly);
  return services;
}
```

Acepta la cadena de conexión de la base de datos como su único input y agrega el contexto de Entity Framework `MainDbContext` como implementación de la interfaz `IUnitOfWork` con el método de extensión habitual `AddDbContext` de Entity Framework Core. Luego, llama al método `AddAllRepositories` para agregar todas las implementaciones de repositorios proporcionadas por `DBDriver`.

Esta es la clase `MainDbContext`:

```
internal class MainDbContext : DbContext, IUnitOfWork
{
  public MainDbContext(DbContextOptions options)
    : base(options)
    {
```

```
    }
  protected override void OnModelCreating(ModelBuilder builder)
   {
  }
  region IUnitOfWork Implementation
}
```

La clase se define como internal ya que no debe ser visible fuera del driver de base de datos. Todas las configuraciones de entidades deben colocarse dentro del método `OnModelCreating` como de costumbre.

La implementación de `IUnitOfWork` está minimizada. El código expandido se muestra aquí:

```
#region IUnitOfWork Implementation
public async Task<bool> SaveEntitiesAsync()
{
  return await SaveChangesAsync() > 0; ;
}
public async Task StartAsync()
{
  await Database.BeginTransactionAsync();
}
public Task CommitAsync()
{
  return Database.CommitTransactionAsync();
}
public Task RollbackAsync()
{
  return Database.RollbackTransactionAsync();
}
#endregion
```

La implementación de `IUnitOfWork` es directa ya que consiste en un acoplamiento uno a uno con los métodos de `DBContext`.

> **Tip**
>
> Dado que solo exponemos `IUnitOfWork` en el motor de inyección de dependencias, todos los repositorios que necesiten `MainDbContext` para su trabajo deben requerir `IUnitOfWork` en sus constructores, y luego deben convertirlo (cast) a `MainDbContext`.

Habiendo discutido lo que necesitamos saber sobre `DBDriver`, pasemos al proyecto Web API.

> **Important**
>
> Conectar el proyecto más externo de una Onion Architecture es sencillo. Solo necesitamos llamar al método de extensión expuesto por los application services, que inyecta todas las implementaciones de application services en el motor de inyección de dependencias, y necesitamos llamar a los métodos de extensión de todos los drivers.

En nuestro caso, necesitamos agregar solo dos llamadas a `Program.cs`:

```
..
builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
```

```
builder.Services.AddSwaggerGen();
builder.Services.AddApplicationServices();
builder.Services.AddDbDriver(
  builder.Configuration?.GetConnectionString(
    "DefaultConnection") ?? string.Empty);
..
```

En este punto, en el caso del proyecto ASP.NET Core, todo lo que queda es adquirir los command handlers para los commands que necesitamos en los constructores de nuestros controllers. Después de eso, cada método de acción debe simplemente usar el input que recibió para construir commands adecuados, y luego debe invocar el handler asociado con cada command.

La breve descripción de cómo manejar la capa más externa de una Onion Architecture completa nuestra breve introducción a esta arquitectura, pero encontraremos ejemplos a lo largo del resto del libro ya que la usaremos para la mayoría de nuestros ejemplos de código.

Relacionado

- 00-Practical serverless and microservices with csharp