DE LAS CAPAS A LAS RODAJAS VERT CAL

Simplifique su código y centrarse en sus características

JON HILTON

Introducción

Un lunes por la mañana más

Es lunes por la mañana; te diriges a la oficina (sustituye el espacio virtual si lo lees en 2020...)

Esperas mantener la cabeza baja, introducirte suavemente en la semana laboral, quizás intentar recordar en qué estabas trabajando antes de que llegara el fin de semana.

Sin pensarlo, levantas la cabeza por encima del monitor y pum, tu jefe te llama la atención.

Ellos sonríen, tú les devuelves la sonrisa. Tras unos segundos incómodos de saludos y preguntas sobre los fines de semana de cada uno, te retiras despreocupadamente, bajando lentamente la cabeza por debajo del monitor.

Pero algo ha cambiado. Se han levantado y vienen hacia ti. Te apresuras a ponerte

los auriculares, pero es demasiado tarde...

"Er, antes de que te metas esta mañana, podrías..." Tu corazón

se hunde.

La petición *parece* bastante razonable:

- ¿Podría añadir un campo para el código postal del usuario en el formulario de contacto?
- Podrías modificar la lógica de envío de correos electrónicos para que no se envíen los
- domingos ¿Te importaría actualizar la página de perfil del usuario para que diga feliz cumpleaños (si es su cumpleaños)

Pero no es tan sencillo.

Nunca es tan sencillo.

Pasas el resto de la mañana agotando la tecla F12 de tu teclado, buscando desesperadamente la multitud de lugares donde reside esta función. Vas de un servicio a otro, pasando por gestores, ayudantes y fábricas, hasta que finalmente, y con no poco alivio, descubres lo que hay que ajustar.

Haces el cambio, ejecutas la aplicación y todo parece estar bien, pero entonces algo te llama la atención.

Otra parte de la aplicación (una pantalla en la que ni siquiera estabas trabajando) no se ve del todo bien. Tienes una sensación incómoda, esto no puede estar relacionado con tu cambio, ¿verdad?

¡Ni siquiera sabías que esas dos pantallas estaban conectadas!

Lo que debería haber sido un "pequeño cambio" te ha costado la mañana y sólo estás empezando.

Cuidado con la brecha

Si has tenido una experiencia así, sabes que no es un lugar

divertido. Entonces, ¿cómo lo solucionamos?

Pues bien, para ello tenemos que entender qué es lo que hace que los pequeños cambios se conviertan en tareas masivas y arriesgadas.

Según mi experiencia, todo depende de la brecha.

En concreto, el desfase entre el requisito (que parece bastante sencillo) y la complejidad de la aplicación.

Cuanto mayor sea la brecha, más difícil será realizar los cambios.

Esta brecha puede adoptar muchas formas: puede aparecer como un número innecesariamente grande de proyectos C# separados...

O una clase que toma muchas, muchas dependencias.

Puede ser que toda la lógica de negocio esté almacenada en una multitud de procedimientos almacenados, con nombres inconsistentes y engañosos.

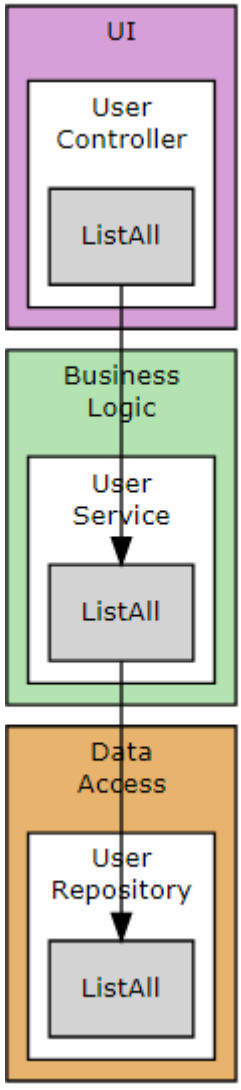
O bien, y aquí es donde nos centraremos durante el resto de este breve libro, puede encontrarse en la forma en que tu aplicación está diseñada, con características divididas en muchas piezas pequeñas y dispersas por toda la aplicación.

Cuando esto ocurre, puedes acabar pasando mucho tiempo buscando esas piezas, intentando desesperadamente sujetar todo lo que tienes en la cabeza para que todo siga funcionando cuando hagas los cambios.

Por qué tus proyectos se convierten en un desastre

Como la mayoría de las cosas en la vida, no podemos resolver un problema hasta que no lo entendemos, así que, en primer lugar, ¿qué hace que las bases de código se conviertan en una masa caótica y dispersa de métodos interconectados?

La cuestión es que tu proyecto no siempre fue así. Si pudieras retroceder en el tiempo hasta el momento en que el proyecto surgió por primera vez, parpadeando a la luz del sol, sin duda encontrarías una pequeña joya bellamente formada, con el código perfectamente organizado y separado.



Seguro que has visto alguna variación de esto.

La lógica de negocio va en la capa de Business Logic (manipulación de datos, combinación de datos de varias fuentes, etc.).

El acceso a los datos va en la capa de acceso a los datos (típicamente donde se conecta a su base de datos y ejecuta consultas y/o comandos).

Finalmente tienes tus controladores (o páginas si estás usando Razor Pages) que están ahí para manejar el lado ASP.NET de las cosas (peticiones, respuestas, HTTP y asuntos de autenticidad).

Aparte de eso, hacen llamadas a la capa lógica del negocio y manejan la respuesta.

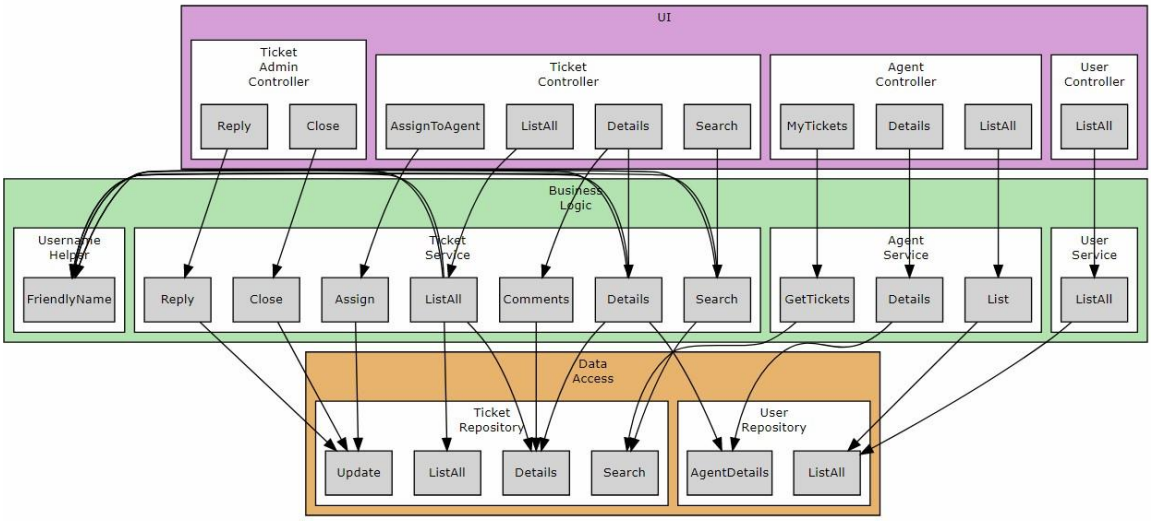
Todo esto está muy bien al principio, cuando el proyecto es pequeño, las funciones son pocas y se acaba de empezar.

Sin embargo, nada se queda pequeño por mucho tiempo, y a medida que llegan más solicitudes de funciones, se enfrentan a un sinfín de opciones...

- ¿Debe esta función utilizar el método de servicio existente o crear uno nuevo?
- ¿Debe reutilizar ese método de repositorio para esta función?
- ¿Hay que crear un nuevo servicio o utilizar el existente?
- ¿Deben estos dos métodos compartir el mismo modelo?
- ¿Debe el controlador llamar a un repositorio
- directamente? ¿Debe el mismo servicio llamar a varios métodos del repositorio?

Son muchas preguntas "debería".

Resulta que este enfoque arquitectónico, lejos de ser sencillo, claro y organizado, te obliga a tomar decisiones casi constantemente, y no pasa mucho tiempo antes de que esos bonitos y claros límites se desvirtúen...



Se trata de un proyecto relativamente pequeño, pero ya están surgiendo problemas.

Lógica condicional == más difícil de razonar sobre el código

Cada vez que veas un método con múltiples llamadas (como el ejemplo) tiene múltiples razones para cambiar. `TicketRepository.Details` en este

El peligro con múltiples razones para cambiar es que terminas escribiendo código condicional para manejar esos diferentes casos. El código condicional hace que sea mucho, mucho más difícil de averiguar rápidamente lo que está pasando con cualquier método dado, lo que requiere que usted mantenga un seguimiento de la "rama" que está en el escenario que está tratando de entender.

Un modelo, múltiples usos

También se acaba con el problema del modelo.

Por ejemplo, `TicketRepository.Search` por ejemplo. Si hay detalles que `AgentService.GetTickets` necesita pero `TicketService.Search` no lo hace, o bien acabas devolviéndolos en todos los casos de todos modos o aceptando nulos en su modelo.

Si alguna vez has tenido que perseguir excepciones de referencia nula en tu código, conoces este dolor.

Fallo en cascada

A medida que los proyectos evolucionan, no es raro encontrar métodos con muchas referencias.

Cuando esto ocurre, tu "pequeño cambio" en una parte del código tiene de repente el potencial de propagarse a través de todas esas referencias y causar el caos en un área de la aplicación que ni siquiera sabías que estaba conectada.

Clases en constante expansión

¡Cuando tengas `clase` en tu código tienes esencialmente un cubo infinito en el que lanzar cualquier tu código!

No hay limitaciones en cuanto al tamaño de las clases y, con el tiempo, tienden a aumentar.

El problema con los controladores, y los servicios, y los repositorios es que los métodos en ellos no están realmente relacionados de ninguna manera significativa.

Una vez que su `TicketService` tiene 30 métodos, en la práctica la mayoría de ellos (si no todos) no están relacionados con
los demás. Cuando se trata de trabajar en un método determinado, hay que pasar por encima de todo ese código para encontrar la parte que te interesa.

Lo peor es que si varios métodos de la clase comparten la lógica (a través de métodos privados) acabas saltando de un lado a otro cuando intentas averiguar qué está pasando.

Ah, y recuerda el problema de que un método tenga múltiples referencias, ¡también se aplica a los métodos privados!

Una vez que un método privado de tu clase es llamado por muchos otros métodos de la clase, vuelves a tener el problema de que un pequeño cambio crea ondas en tu aplicación.

Y el problema con las ondas es que tienden a convertirse en olas...

La entropía se saldrá con la suya...

No hay ningún compilador o IDE en el mundo que haga cumplir su enfoque de arquitectura por capas.

Por supuesto, puede que sepas exactamente cómo quieres que funcione todo esto, con cada capa bien organizada, las llamadas entre clases/métodos se mantienen al mínimo, las clases y métodos con responsabilidades únicas, etc.

Y, si eres el único desarrollador del proyecto y realmente te centras al 100% en mantener todo organizado, con un mínimo de acoplamiento y lógica condicional puede que consigas mantener tu proyecto bajo control.

Pero entonces, cuando Sam, tu amable pero asertiva propietaria de Producto, asoma la cabeza por la puerta y te pregunta si puedes sacar tu arreglo en los próximos diez minutos porque el equipo de marketing lo necesita para su demostración, buena suerte para mantener intactas esas capas pulcramente organizadas. `si`

Tal vez, sólo por esta vez, tenga que reutilizar ese pequeño método, o declaración a lanzar una gestión de este requisito urgente de última hora.

Luego está el tema de los múltiples desarrolladores.

En cuanto haya varios desarrolladores trabajando en el código, es muy probable que algunos (o todos) no entiendan del todo cómo se supone que deben funcionar todas las capas, ni cómo mantener el acoplamiento al mínimo y los demás matices para mantener este enfoque arquitectónico particular bajo control.

Inevitablemente, se apilan más y más "soluciones rápidas" y, después de unos meses de desgaste, no es ninguna sorpresa que toda la aplicación se deshaga.

Teniendo en cuenta todos estos problemas, te preguntarás: ¿cuál es la alternativa?

¿Existe otra forma de preparar tu proyecto para que tenga más posibilidades de éxito, con menos posibilidades de que la entropía teja su enmarañada red?

Paso 1 - Utilizar las carpetas de funciones

El enfoque de las capas tiene que ver con la separación técnica.

Piensa en tu típica función. Empieza como una maqueta, una historia de usuario, un resumen vago que te grita un gerente cuando pasas por su habitación de camino a tomar un café...

Sea como sea, es probable que su enfoque para construir cualquier característica sea el mismo...

Coges la función, la divides en partes técnicas y luego piensas dónde colocar esas partes. Unas

horas más tarde (días o semanas), vuelves a trabajar en esa función.

¿Cómo se vuelve a meter la cabeza en esta función? ¿Cómo se puede saber a dónde ir para hacer ¿sus cambios?

Inevitablemente, su primer reto es encontrar su código.

Dependiendo de cómo se haya implementado tu función, esto puede llevar unos minutos (optimistas) u horas (y horas) de "Ir a la definición", navegando dentro y fuera de las clases y métodos compartidos, tratando de mantener todas las piezas en tu cabeza para que puedas averiguar dónde hacer este pequeño cambio.

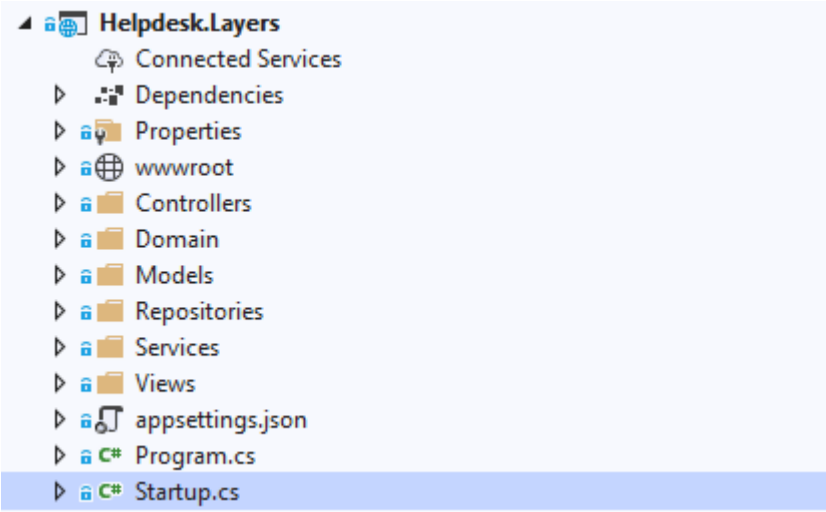
Entonces, ¿por qué puede acabar siendo tan doloroso encontrar su código?

Suele ocurrir porque las características, que por su propia naturaleza son unidades cohesivas de funcionalidad, se fragmentan en muchas piezas cuando se pasa de los requisitos a la implementación técnica.

Esta separación técnica se presenta desde el principio cuando se crea un nuevo proyecto con un marco de trabajo como ASP.NET MVC.

Las plantillas de proyectos le animan a seguir este camino desde el principio con características ya fragmentadas en múltiples carpetas; Modelos, Controladores y Vistas.

Al poco tiempo, a estos se les unen inevitablemente Repositorios, Servicios, Gestores, Ayudantes... la lista es interminable.

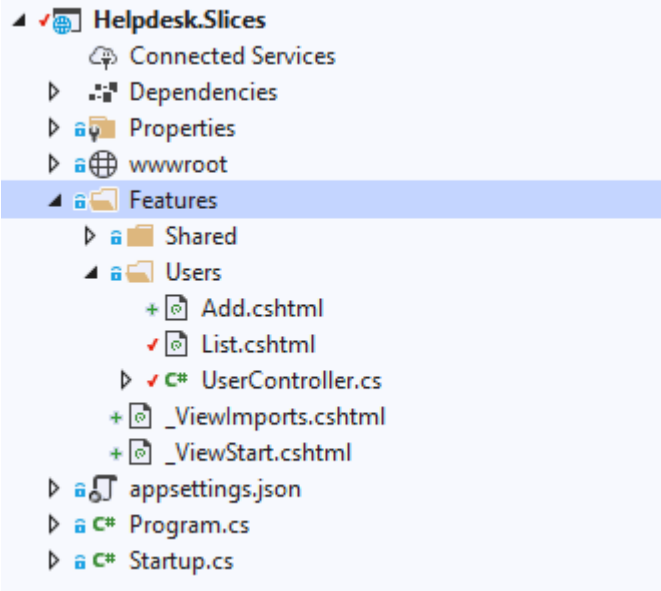


Pero, ¿y si no empezara con esta separación técnica? ¿Y si, en cambio, empezara a organizarse por características?

El primer paso (y el más sencillo) es adoptar carpetas de características

para tus proyectos. Aquí es donde usted co-localiza todo lo necesario para

su característica en una carpeta.



Con este enfoque, las partes constituyentes de su función (controlador, modelo y vistas si se utiliza MVC) se encuentran en un solo lugar (¡espero que bien nombrado!).

Esto le proporciona inmediatamente varias ventajas;

Tienes un "in"

Cuando te piden que trabajes en una función de una aplicación basada en capas, ¿por dónde empiezas?

Supongo que tratas de encontrar el punto de entrada, una página Razor o un controlador MVC que parece que podría ser el lugar para empezar. O tal vez empieces ejecutando la aplicación en el navegador antes de intentar averiguar dónde está definida esa parte de la UI.

Desgraciadamente, los controladores suelen devolver múltiples vistas y, a menudo, se les asignan nombres engañosos o francamente incorrectos. Esto, junto con el hecho de que las vistas se encuentran en una carpeta completamente diferente, puede hacer que este proceso sea más arduo y que consuma más tiempo del necesario.

Las carpetas de características ayudan porque en su lugar puedes buscar la carpeta correspondiente.

Esto funciona especialmente bien si prestas cierta atención a los nombres que utilizas al nombrar dichas carpetas.

Una vez allí, no tienes que ir a la caza de la acción del controlador relevante (que puede o no estar en un controlador con nombre sensato), o vista, o modelo(s) porque todo lo que necesitas está ahí en un solo lugar.

Puede ver lo "grande" que es una característica

Una vez que tenga carpetas de características, si una de ellas consta de muchos archivos, lo más probable es que se haya convertido en una gran característica.

De hecho, podría ser demasiado grande. Muchos archivos (y código) podrían significar que esto está maduro para la refactorización, pero incluso si no, al menos puedes ver lo que estás

tratando.

Vea qué características existen

Este enfoque también le ayuda a hacerse una idea de la escala de la propia aplicación. Puedes ver rápidamente cuántas (y qué) características soporta esta aplicación.

Una vez más, la clave es pensar realmente en lo que es (y representa) la característica y utilizar esa terminología al nombrar las carpetas de características. Una forma de conseguirlo es prestar mucha más atención al lenguaje que utilizan los usuarios/especialistas del dominio.

Cambiar el nombre de una carpeta es fácil (y herramientas como Resharper o Rider también ayudan a actualizar los espacios de nombres cuando lo haces), así que no tengas miedo de cambiar los nombres de las carpetas de características a medida que cambia tu comprensión.

Disminuir la reutilización

La reutilización del código es peligrosa.

El hecho de que dos características diferentes muestren detalles sobre una persona no significa que sean lo mismo.

Es tentador mirar dos modelos de vista que devuelven detalles sobre una persona y pensar que son iguales, por lo que se debería usar un solo modelo, pero ¡aquí hay dragones!

Con el tiempo, estos rasgos, que al principio parecen muy similares, tienen tendencia a evolucionar, a menudo hasta el punto de tener que mostrar detalles diferentes.

En cuanto esto ocurre, corres el riesgo de que tus cambios para un caso de uso afecten negativamente al otro. Además, para que sigan comportándose de forma diferente tienes que escribir... sí, lo has adivinado... más lógica condicional.

Como siempre, la lógica condicional hace que su código sea más difícil de comprender.

El uso de carpetas de características no resuelve este problema, pero le da una pista. Si tienes un modelo en una carpeta de características, evita referenciarlo desde otra.

> Si te sirve de ayuda, imagina la carpeta como un muro de
>
> ladrillos sólido sin puertas. No puedes atravesar físicamente
>
> este muro desde el exterior.
>
> Por lo tanto, no se puede acceder al modelo de vista de una función para utilizarlo en una función completamente diferente
> característica.

Si es absolutamente necesario compartir un modelo entre funciones, póngalo en una carpeta compartida, pero de nuevo, hágalo raramente (si es que lo hace).

Si tiene una carpeta **compartida** rebosante de modelos compartidos, es casi seguro que está reutilizando demasiado y le convendría crear modelos separados para cada característica.

Más adelante, cuando exploremos los cortes verticales, hablaremos de ello.

Su enfoque se ha desplazado.

Este es un factor realmente importante para mí.

Centrarse en las características significa pensar en los requisitos, en la terminología que los rodea y en cómo encajan con el resto de la aplicación.

Esto disminuye significativamente los riesgos de que te vayas y te pierdas "en la maleza" del código, donde es muy fácil caer en la trampa de resolver "problemas" técnicos que en realidad no afectan a la característica que estás construyendo en ningún sentido material.

Las carpetas de características no resuelven este problema, pero sí cambian el enfoque y, con suerte, te empujan a pensar más en el **qué** de la característica que estás construyendo y menos en el **cómo** la estás implementando...

Razor Pages ya lo hace

Vale la pena decir que esto es más o menos el enfoque que ya está utilizando si ha optado por utilizar Razor Pages (en lugar de MVC) para sus aplicaciones.

Razor Pages prescinde de los controladores, modelos y vistas por separado y los consolida en dos archivos (estrechamente relacionados). Estos archivos contienen el marcado y la lógica del modelo/negocio.

Todo lo que ASP.NET aporta sigue funcionando con Razor Pages (enrutamiento, vinculación de modelos, validación, filtros, etc.) pero su código no está fragmentado en carpetas separadas dentro de la aplicación.

La buena noticia es que puede adoptar el mismo enfoque utilizando MVC, o si expone una API web desde su proyecto ASP.NET Core, y así puede obtener todas las ventajas mencionadas anteriormente.

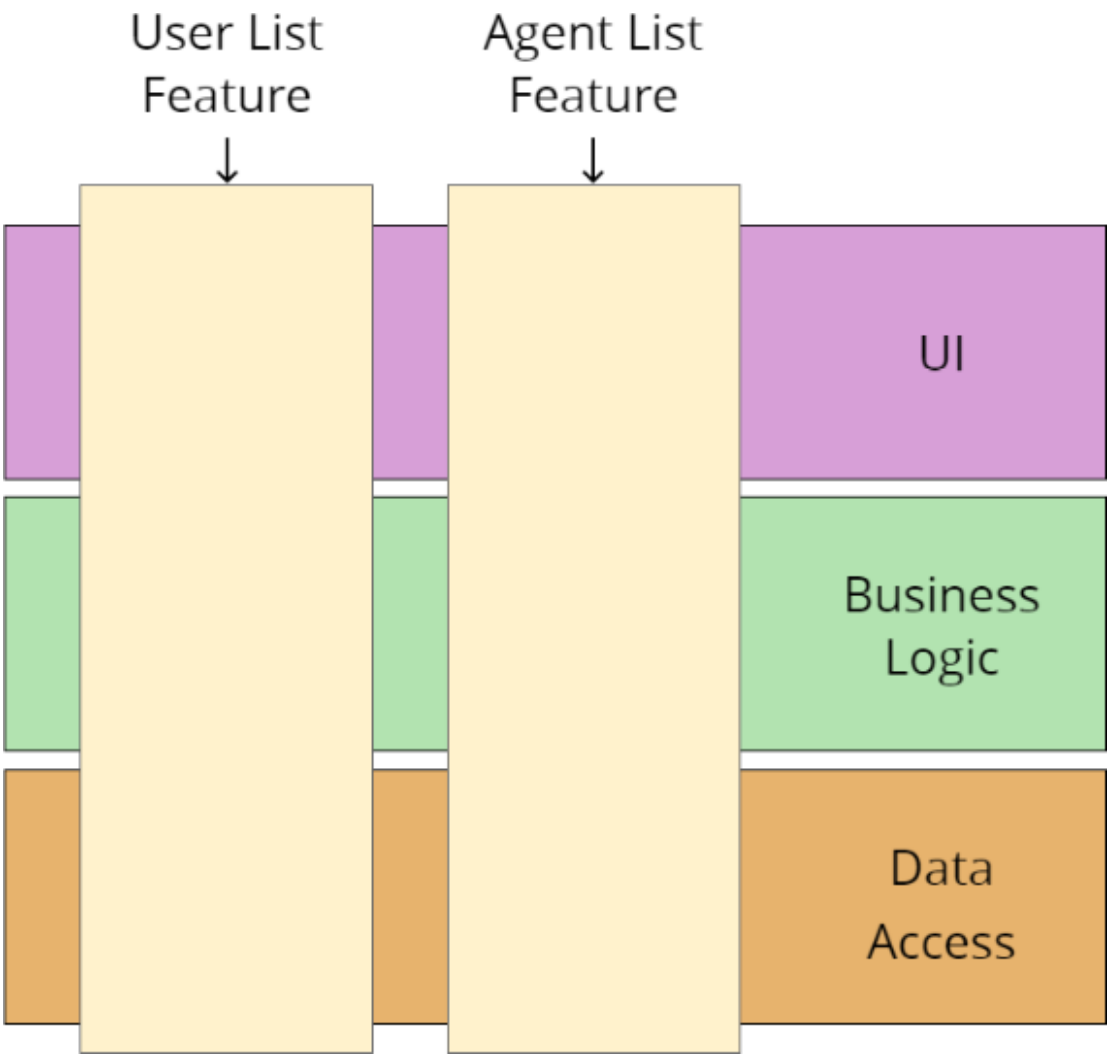
> Con la Web API puedes literalmente empezar a utilizar una carpeta de "Características" y organizar tu código por características.
>
> Para hacer que esto funcione para MVC necesitas hacer un pequeño ajuste en tu proyecto para que ASP.NET sepa dónde buscar tus vistas.
>
> Consulta el **capítulo Actúa - Utiliza las carpetas de funciones** para ver cómo hacerlo.

Paso 2 - Utilizar cortes verticales

Bien, esto es lo más importante. Es un paso más allá del uso de las carpetas de características y no está totalmente libre de riesgos, pero las recompensas son muchas.

¿Recuerdas las trampas que identificamos inherentes a la organización de tu código mediante capas?

Los cortes verticales eliminan esas capas y le animan a adoptar un enfoque pragmático para construir sus características.



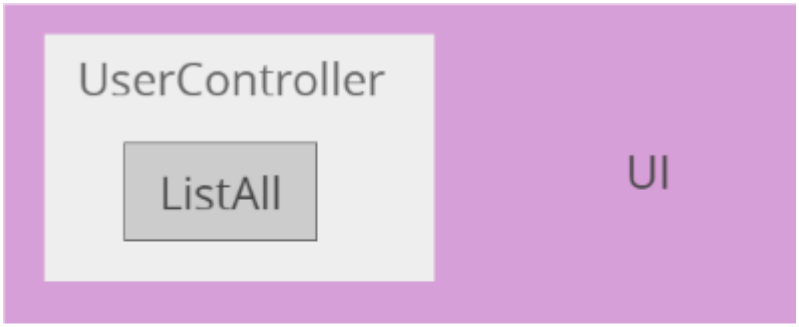
Es un poco críptico descifrar lo que esto significa con sólo mirar un diagrama como éste.

¿La idea? En lugar de crear servicios, repositorios (y una vez que has empezado por este camino tiendes a seguir, así que ayudantes, fábricas, fábricas de servicios, etc.) queremos crear un "**espacio**" para empezar a construir nuestra función.

Este "**espacio**" no tiene ninguna estructura predeterminada (porque la estructura tiende a estorbar, sobre todo cuando se empieza con una nueva función).

Sólo necesitamos un lugar para poner la lógica de nuestra función.

Si asumimos por ahora que seguimos con MVC, seguiremos necesitando un controlador y una acción, pero ¿qué más?



En este punto podríamos simplemente crear una clase, con un método y empezar a codificar todo allí. Pero, ¿cómo llamarías a esa clase (y al método)?

El peligro es que volvamos a nuestras viejas costumbres y lo llamemos algo así como...

```
public class ServicioUsuario {

    public List< User> ListAll(){
        // lógica
    }

}
```

Lo que nos devuelve al punto de partida. Ahora tenemos una clase, que estaremos tentados de reutilizar para muchas otras cosas, que probablemente crecerá con el tiempo y que nos obligará a tomar muchas decisiones para cada nueva función que construyamos.

Aquí es donde una pequeña abstracción puede ayudarnos a...

Caer en el pozo del éxito

Ahora probablemente quieras maravillarte con la brillantez de las habilidades de dibujo que se exhiben aquí, así que te daré un momento...

Bromas aparte, este es probablemente el mayor "hack" para hacer que tu aplicación escale y evolucione en una mejor dirección.

Como seres humanos estamos predispuestos a tomar el camino más fácil de A a B.

Si es fácil... probablemente lo haremos y si hay algún tipo de fricción, o percepción de fricción, automáticamente buscaremos la alternativa más fácil.

Así que cuando se llega a trabajar en una función, en una aplicación existente, ¿cómo se decide qué enfoque tomar (dónde poner el código, cómo nombrar los métodos, las clases, etc.?)

Supongo que te basas en lo que ya existe, al igual que los demás desarrolladores que trabajan en la aplicación.

El problema es que cada persona ve las cosas de forma ligeramente diferente. Si alguna vez has jugado al juego "Gossip" (también conocido como Telephone (EE.UU.), o Chinese Whispers (Reino Unido)) ya sabes cómo va esto.

En el juego, una persona susurra un mensaje a otra, que a su vez lo susurra a otra, y así sucesivamente hasta que la última persona revela el mensaje que ha escuchado, y se produce la hilaridad.

Por desgracia, cuando los susurros son código, y cada persona saca sus propias conclusiones basándose en el único trozo de código que decidió copiar, la hilaridad puede o no ser el resultado final.

En última instancia, es difícil controlar lo que hace la gente, pero podemos poner algunos raíles para guiarla suavemente hacia un resultado mejor.

Si alguna vez has ido a jugar a los bolos y has colocado las barandillas de los parachoques, ya sabes cómo funciona esto. Esas barandillas no te garantizan una puntuación perfecta, pero te protegen del fracaso más absoluto.

Para nosotros, esto significa diseñar nuestras aplicaciones de manera que cuando un desarrollador trabaje en ellas sea más probable que haga lo "correcto", porque es lo más obvio/fácil de hacer.

Creación de una estructura mínima (carriles guía) con MediatR

Aquí es donde me gusta utilizar el patrón Mediator y específicamente MediatR para mis aplicaciones .NET.

> MediatR es una biblioteca de código abierto creada y mantenida por Jimmy Bogard. Debo

decir en este punto que Jimmy ha estado escribiendo durante años sobre muchos temas, incluyendo cómo modelar más eficazmente los dominios de negocio utilizando, entre otras cosas, rodajas verticales.

He aprendido muchos de los conceptos que estamos explorando aquí de las publicaciones de Jimmy a lo largo de los años y recomiendo encarecidamente que visites su blog.

MediatR proporciona una abstracción ligera que le empuja a pensar en términos de características cohesivas, con entradas y salidas claramente definidas.

Puede añadir MediatR a su aplicación mediante Nuget.

```
dotnet add package MediatR
```

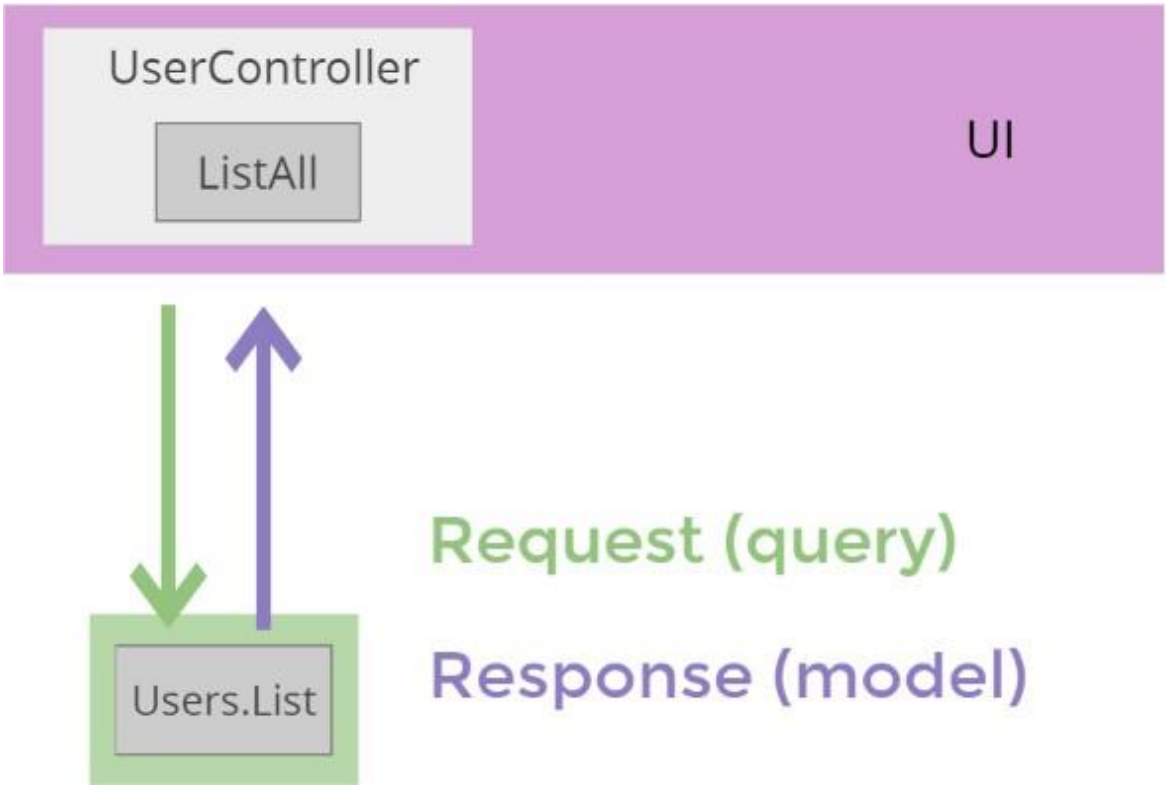
> Hay un par de pasos más antes de poder utilizar MediatR en sus aplicaciones ASP.NET Core.
>
> Consulta los capítulos "Actúa" para ver las instrucciones paso a paso sobre el uso de MediatR en tus proyectos.

Es más fácil de entender con un ejemplo, así que aquí está cómo podríamos mostrar una lista de usuarios en nuestra aplicación ficticia de Helpdesk.

En lugar de crear un "servicio" general, queremos representar sólo nuestra función de "Lista de usuarios". Hay tres partes que necesitamos para que MediatR haga lo suyo.

Los dos primeros van juntos:

- La petición (esencialmente nuestra consulta)
- La respuesta (los datos que deseamos devolver, es decir, el viewmodel)



Si somos capaces de crear una representación de nuestra consulta y de los datos que esperamos que nos devuelva, ya tenemos la mitad del camino recorrido.

Features\Users\List.cs

```
public class Lista
{
    public class Consulta : IRequest< Modelo>
    {
    }
```

```
    public class Modelo
    {
        public List< UserDetails> Users { get; set; }

        public class DetallesDeUsuario
        {
            public string FirstName { get; set; }
            public string LastName { get; set; }
            public DateTime FirstRegistered { get; set; }
        }
    }
}
```

Este par de clases Request/Response representa explícitamente (y exclusivamente) la entrada y la salida de esta única función.

> No es necesario anidar las clases como yo lo he hecho, pero tiendo a favorecer este enfoque porque ayuda a dejar claro que estas no son clases de propósito general para ser utilizadas en cualquier otra parte de la aplicación.

Este enfoque aporta algunas ventajas.

Una terminología más precisa

Cuando sabes que tus modelos de C# representan una característica, puedes ser súper específico con la terminología que utilizas. En lugar de comprometer el nombre de una clase o propiedad porque se utiliza en varios lugares, puede darle el nombre preciso que tiene sentido en este contexto.

Así, por ejemplo, si se modela una función de búsqueda de entradas de esta manera, se podría nombrar la lista resultante
de usuarios en lugar de un nombre más genérico como `Usuarios`. `Resultados de la búsqueda`

Representación específica de sus datos

También puede crear representaciones muy específicas de sus datos para cada característica.

Si esta lista de usuarios necesitara mostrar el nombre del usuario en su totalidad, y no tuviera necesidad de mostrar el nombre y el apellido
nombres por separado se podría ajustar fácilmente `Modelo.UserDetails` para exponer sólo `Nombre completo`
confiando en que este cambio no se extenderá por su aplicación y romperá otras características.

Cuando se define una característica de esta manera, la preocupación inicial es modelar sus entradas y salidas...

> En este caso no hay parámetros en nuestro `Consulta` (porque queremos a todos los usuarios y estamos
> sin filtrar en base a ningún criterio) y tenemos un número `Modelo` que representa una lista de usuarios bastante sencillo.

De nuevo, el punto crucial a tener en cuenta es que esto representa una característica muy específica.

Pero, ¿dónde va la lógica real?

Ahora bien, esta solicitud y respuesta está muy bien, pero necesitas un lugar donde colgar tu

lógica. Con MediatR puedes lograr esto con un **manejador** para tus peticiones.

Features\Users\List.cs

```
public class Lista {
```

```
    // otro código (solicitud/respuesta)

    public class RequestHandler : IRequestHandler< Query, Model>
    {
        public async Task< Model> Handle(
            Query request,
            CancellationToken cancellationToken)
        {
            lanza una nueva NotImplementedException();
        }
    }
}
```

El manejador es una `clase` que implementa `IRequestHandler<TRequest, TResponse> .`

> He optado por anidar esto como parte de la `List a` clase pero de nuevo esto es personal preferencia general.
>
> Personalmente me gusta que la clase "externa" proporcione el contexto para esta función y que luego la solicitud, la respuesta y el manejador se encuentren en un solo lugar.

Una vez creado el manejador, el último paso es hacer lo más sencillo para satisfacer el requisito y devolver los datos relevantes a través del modelo.

En muchos casos, y para muchas funciones, esto puede ser tan sencillo como consultar la base de datos, hacer un mapeo y devolver los resultados.

```
public class RequestHandler : IRequestHandler< Query, Model>
{
    private readonly HelpdeskDbContext _dbContext;

    public RequestHandler(HelpdeskDbContext dbContext)
    {
        _dbContext = dbContext;
    }

    public async Task< Model> Handle(
        Query request,
        CancellationToken cancellationToken)
    {
        var users = _dbContext. Users. Select(x => new Model. UserDetails
        {
            Nombre = x. Nombre, Apellido =
            x. Apellido, PrimerRegistro = x.
            FechaCreación
        });

        devolver el nuevo Modelo
        {
            Users = await users. ToListAsync(cancellationToken)
        };
    }
```

En este punto eres libre de utilizar cualquier ORM que desees (o cualquier otro medio que tengas para acceder a la base de datos).

Con este gestor, hemos delimitado perfectamente nuestra primera función.

Obsérvese que no hemos optado por un enfoque multicapa. Así que a menudo cuando usamos capas terminamos con "servicios intermedios" sin sentido que simplemente delegan en los métodos del repositorio (que realmente hacen todo el trabajo).

El verdadero peligro es que acabemos estructurando por estructurar, nombrando a todo en función de su función técnica y del encasillamiento que hayamos decidido hacer.

En cambio, el enfoque de las rebanadas verticales consiste en centrarse en la función que se está construyendo, en hacer lo más sencillo posible para satisfacer los requisitos iniciales, y luego refactorizar en función de los problemas que surjan con este código a medida que los requisitos vayan apareciendo y la complejidad aumente de forma natural.

En muchos, muchos casos el código seguirá siendo así de simple y realmente no hay necesidad de complicar las cosas creando un montón de abstracciones innecesarias.

Utilice MediatR para "enviar" sus solicitudes

El último paso es ejecutar realmente la consulta y manejar los resultados.

Para ello debe inyectar `IMediator en` su controlador ASP.NET (o Razor Page, o Blazor Server), luego llame a `Mediador.Enviar` para enviar su consulta real.

```
public class UserController : Controlador
{
    private readonly IMediator _mediator;

    public UserController(IMediator mediator)
    {
        _mediador = mediador;
    }

    public async Task< IActionResult> Index()
    {
        var model = _mediator. Send(new List. Query());
        return View("List", await model);
    }
}
```

El resultado de `_mediator.Send()` será su modelo, poblado por el manejador que eso creó.

Como puedes ver, la única dependencia que necesitamos inyectar en nuestros controladores ahora es `IMediator` .

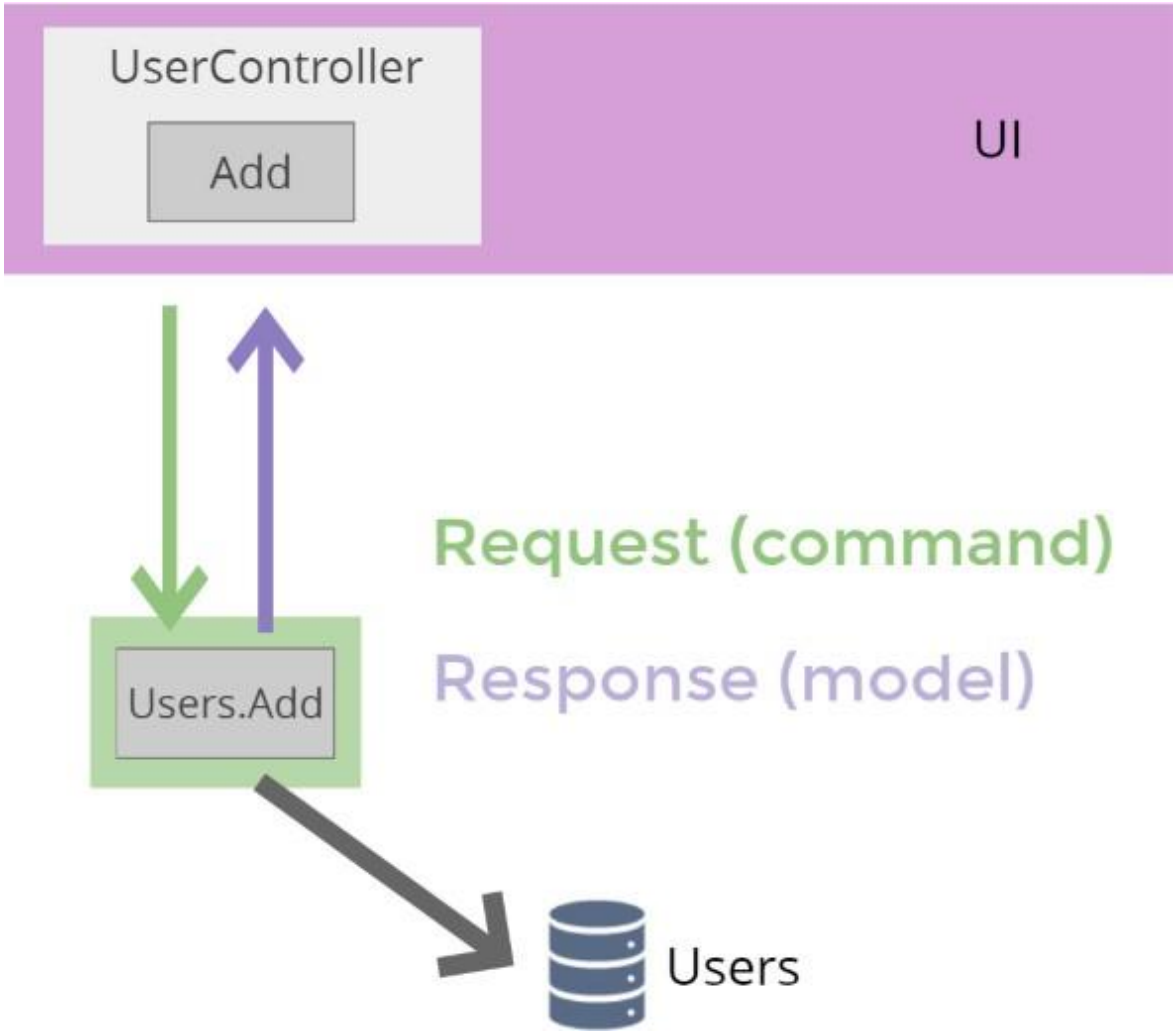
Bajo el capó, MediatR localizará su manejador, creará una instancia del mismo (utilizando la inyección de dependencia para proporcionar cosas como su Contexto DB y otros servicios) y luego lo invocará y devolverá los resultados.

No sólo consultas

Naturalmente, es probable que su aplicación requiera una lógica más compleja que una simple consulta.

A veces tendrá que cambiar (o añadir, o eliminar) datos o realizar cálculos más complejos.

Supongamos que necesita añadir un nuevo usuario al sistema. Este es en gran medida el mismo proceso que utilizamos cuando implementamos la lista de usuarios.



Todavía tenemos que definir una entrada (a la que podríamos referirnos como un comando en este caso porque no estamos ejecutando una consulta, sino haciendo un cambio en los datos).

Tenga en cuenta que MediatR implementa un único patrón de Solicitud/Respuesta y no tiene opinión sobre este tema, por lo que depende de usted el nombre de sus clases de Solicitud y Respuesta.

Me gusta utilizar **Query** and **Command**, ya que me ayuda a clarificar mi pensamiento sobre lo que estoy implementando.

Es mucho más fácil razonar sobre una función cuando recupera datos o los modifica, pero no hace ambas cosas al mismo tiempo.

Se trata de un enfoque pragmático de lo que se denomina separación Comando-Consulta.

> "Todo método debe ser un comando que realiza una acción, o una consulta que devuelve datos a quien lo llama, pero no ambas cosas.
>
> En otras palabras, hacer una pregunta no debe cambiar la respuesta".
>
> -- Betrand Meyer

En este caso se trata de elegir si la emisión de este comando debe devolver algún dato.

Puede ser útil para recuperar algo, como la opción `id` del registro insertado, pero puede igualmente de no devolver nada.

```
public class Añadir
{
    public class Comando : IRequest
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
    }
}
```

En este caso, mediante la implementación no se devolverá una respuesta.

`Solicitud de información` (y no `IRequest<TResponse>`) estamos afirmando que este

Con eso podemos crear un manejador.

```
public class CommandHandler : IRequestHandler< Command>
{
    public async Task< Unit> Handle(Command request, CancellationToken
cancellationToken)
    {
        devolver la Unidad. Valor;
    }
}
```

Puedes ver cómo esto es muy similar al manejador anterior e implementa `IRequestHandler<Command>` .

> Bajo el capó MediatR tiene sólo un "algo" `IRequestHandler` y requiere que se devuelve de un controlador.
>
> Hay buenas razones técnicas para ello (para asegurar que MediatR se comporta de forma coherente tanto si devuelve una respuesta como si no).
>
> Desde nuestro punto de vista, la clave es que si no definimos nuestra propia respuesta, deberíamos devolver `Unidad.Valor` en su lugar. `Unidad` representa "nada" y por lo tanto es esencialmente un representación de `void` . Un método que devuelve no devuelve `nada` (usando `void`). `Unidad` equivale a un método que
>
> F# tiene un tipo de unidad incorporado.
>
> C# no tiene `Unidad` atype así que Jimmy Bogard implementó el suyo propio para MediatR.

Ahora sólo tenemos que rellenar los espacios en blanco.

Siempre me parece que implementar características desde este punto es un poco como pintar por números, tenemos el espacio para llenar (nuestro manejador vacío) y ahora sólo tenemos que llenarlo.

```
public async Task< Unit> Handle(
    Solicitud de comando,
    CancellationToken cancellationToken) {

    await _dbContext. Users. AddAsync(new User
    {
        FirstName = request. FirstName,
        LastName = request. LastName
    }, cancellationToken);

    await _dbContext. SaveChangesAsync(cancellationToken);
```

```
    devolver la Unidad. Valor;
}
```

De nuevo, esto es lo más sencillo que podemos hacer con este código (si usamos EF Core, como en este caso).

Necesitamos un pequeño mapeo entre la entidad entrante `Solicitante` y la entidad de dominio que realmente queremos
para guardar, pero por lo demás podemos simplemente añadir nuestro nuevo usuario a la BD, guardar nuestros cambios y ya está.

Cablear el mando

La buena noticia es que el cableado de una solicitud de MediatR que no devuelve nada es idéntico al cableado de una solicitud que devuelve una respuesta.

```
[HttpPost]
public async Task<IActionResult> Add(Add.Command command)
{
    await _mediator.Send(command);
    return RedirectToAction(nameof(Index));
}
```

Por cierto, puedes utilizar el model binding de ASP.NET Core para rellenar tus peticiones de MediatR, como he hecho aquí.

> Model Binding en ASP.NET Core hará coincidir automáticamente los datos entrantes en la solicitud HTTP con las propiedades del modelo, utilizando los nombres de las propiedades.
>
> Así que si hay una propiedad llamada "Edad" y la petición HTTP entrante incluye `Edad` en el campo cuerpo de la `Edad` se rellenará en el `comando` modelo.

Ahora sólo hay que enviar esa vía `MediatR` como hemos visto antes y, de repente, un nuevo usuario para que se añada a la base de datos.

Construir un mejor software con cortes verticales

Así que ahora que hemos visto una forma de implementar técnicamente las rebanadas verticales, vale la pena dar un paso atrás y considerar las implicaciones de construir su software de esta manera.

Encapsular por característica

El gran cambio está en la forma de encapsular el código.

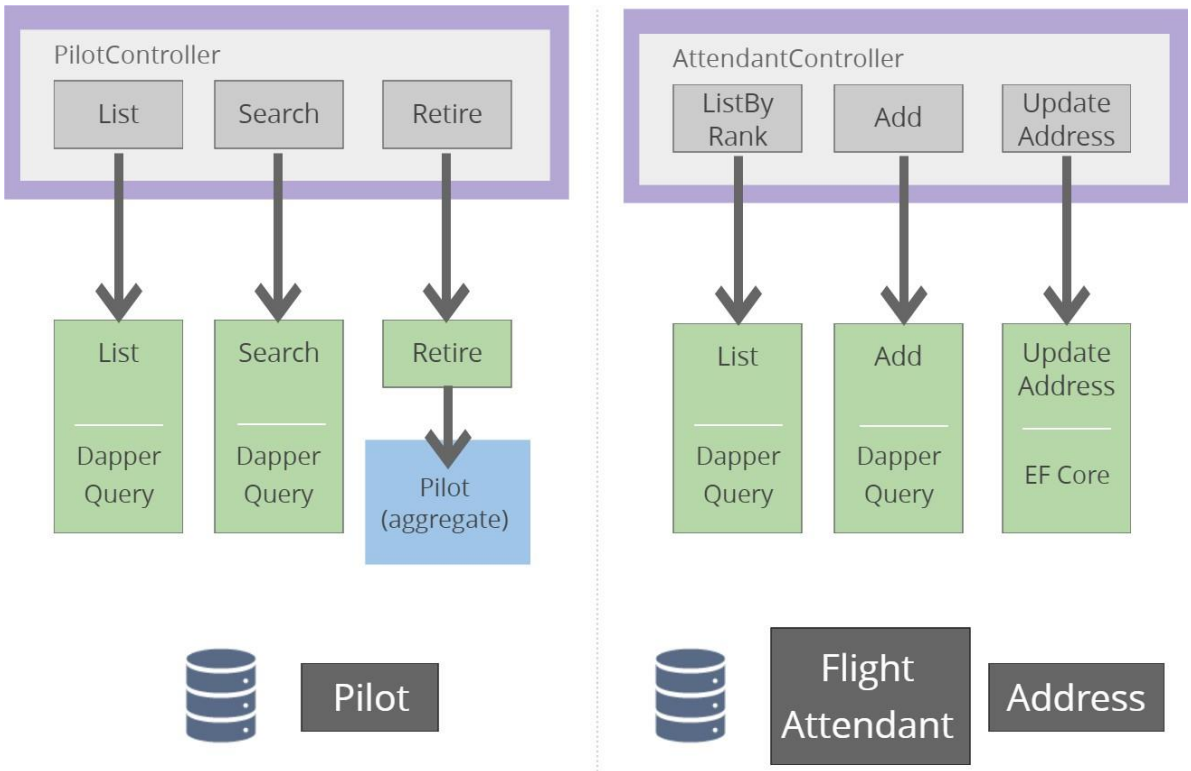
En lugar de pensar en "términos técnicos" como repositorios, servicios y gestores, puedes centrarte en la función en sí.

A medida que su sistema crece de esta manera, una cosa queda muy clara: la arquitectura de corte vertical reduce drásticamente las posibilidades de que se unan varias funciones de una manera que haga más difícil el mantenimiento de su aplicación.

Si se da un paso atrás y se observa el panorama general, se puede

ver por qué. Tomemos como ejemplo este ficticio software de

gestión de aerolíneas...



Debido a que cada función se implementa como un manejador de solicitudes de MediatR, es (intencionalmente) difícil hacer conexiones entre las funciones.

De forma natural, no inyectarías una instancia de uno de esos manejadores de peticiones en otro, por lo que esta pequeña abstracción (el patrón mediador en sí mismo) nos ayuda a

empujarnos a ese pozo de éxito que mencionamos antes.

> **Imagina que tu aplicación fuera una serie de tubos de plástico**
>
> Decides disparar una pelota de ping-pong en uno de esos tubos...
>
> Con un enfoque tradicional de capas, esa bola se lanzará en muchas direcciones diferentes mientras sigue las conexiones entre las partes de su aplicación.

> Incluso puede ir en círculos o pasar mucho tiempo viajando por las diferentes ramas de cada sección de su aplicación.
>
> Sería difícil predecir por dónde iba a salir, difícil entender por qué salió por donde lo hizo y prácticamente imposible averiguar qué partes de la aplicación visitó por el camino.
>
> Por el contrario, si la máquina de ping-pong se construyera como una serie de tubos individuales que van en su mayoría en línea recta, podrías enviar la pelota de ping-pong en un extremo y verla bajar directamente por el tubo y salir por el otro.
>
> Sería mucho más sencillo predecir por dónde podría salir, y observar por dónde viajó mientras estaba dentro.

Esto es lo que los cortes verticales pueden aportar a su aplicación, ¡más líneas rectas!

Modelos de uso único

Los modelos de vista, cuando se comparten entre varias funciones, tienen

numerosos problemas. Tienden a ser más grandes para acomodar más y más

casos de uso

Cuando se reutiliza un modelo para múltiples características es probable que se acabe con propiedades que no son necesarias para todos los usos del modelo.

Por ejemplo, digamos que tenemos un `UserInfo` viewmodel.

No sería tan sorprendente en una aplicación tradicional encontrar que se utiliza en múltiples lugares, para múltiples vistas.

Tal vez se utilice al devolver una lista de usuarios, pero también al ver los detalles de un usuario específico.

En este caso, es probable que la vista de lista sólo necesite mostrar un puñado de campos, mientras que la vista de detalles necesita mostrar muchos más.

Al compartir este `UserInfo` modelo vamos a acabar bien:

- Devolver a la lista muchos más datos de los que necesita
- El envío de `UserInfo` con los datos que faltan (nulos) cuando sabemos que ciertos campos instancias no es necesario

En cualquier caso, la fiabilidad de ese modelo y su capacidad para representar claramente las características individuales se ven comprometidas.

El enfoque de los cortes verticales elimina esta ambigüedad y deja claro que cada modelo representa esa característica específica.

Concéntrese en sus características

Cuando empiezas a utilizar cortes verticales algo queda muy claro.

Al estar centrado únicamente en la implementación de características específicas, puedes razonar sobre ellas con mucha más claridad.

Vale la pena ser inquisitivo cuando se empieza a construir cualquier cosa en software.

Haga preguntas, cuestione el requisito, aclare su propósito, pregunte por qué alguien lo necesita/quiere y profundice en los matices de cómo funcionará realmente.

Después de todo, no se puede construir algo que no se entiende.

A medida que su comprensión mejora, tiene el lugar perfecto para capturar esa comprensión en la **Solicitud**, la **Respuesta** y el **Manejador de** su función.

Refactorización por características

Pero un momento, si usamos manipuladores, ¿no es el peligro de que sean cada vez más grandes y difíciles de mantener?

Sí, y la pregunta no es tanto "¿debes refactorizar?" sino "¿cómo debes refactorizar?".

A medida que aumenta la complejidad de su aplicación, siempre vale la pena estar atento a los problemas y refactorizar según sea necesario.

La gran diferencia con los cortes verticales es que puedes hacer esto caso por caso, refactorizando según lo requieran las características particulares.

¿Por qué tan lento?

El aspecto de esta refactorización variará, pero hay muchos casos en los que trabajar en una característica concreta de forma aislada puede ser ventajoso.

Tal vez descubra que una consulta vital en su sistema está funcionando lentamente

Si ha representado esa consulta en un manejador MediatR, resulta bastante fácil analizar esa consulta por sí sola.

Puede ser que esta sea una parte súper importante de su aplicación y por lo tanto emplear Dapper tiene sentido para acelerar esta consulta en particular (pero EF Core está absolutamente bien para todo lo demás porque funciona lo suficientemente bien y los desarrolladores son productivos usándolo).

Como la función está encapsulada en un solo lugar, es mucho más fácil considerar su cambio de forma aislada.

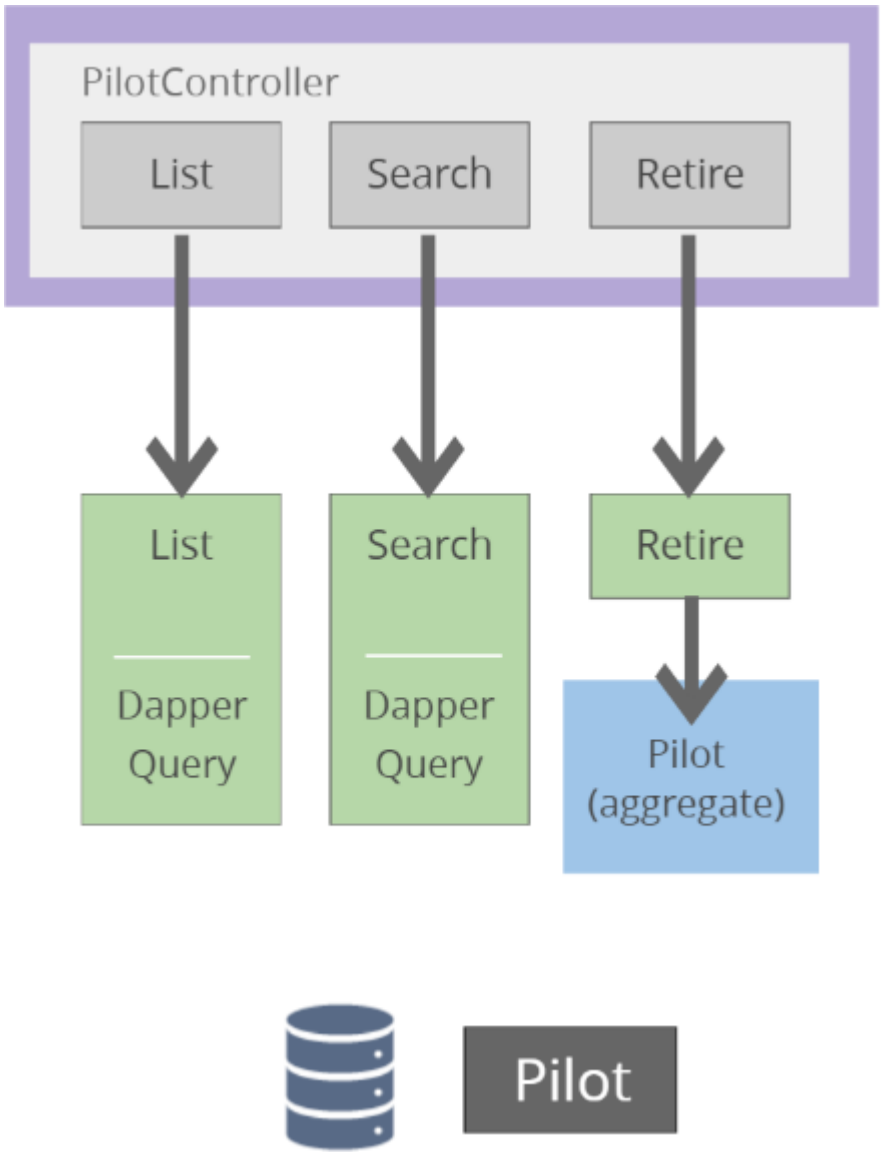
Con un enfoque de capas, el peligro, cuando te encuentres metido hasta las rodillas en tu repositorio, mirando montones de consultas de aspecto similar, es que te sientas tentado a tratar de mantener todo en ese repositorio funcionando de la misma manera, incluso si no hay un caso real de negocio para hacerlo.

Esto puede llevar fácilmente a un montón de "trabajo ocupado" cambiando las implementaciones de repositorios enteros (o incluso toda la aplicación) cuando, en realidad, cambiar a Dapper para una consulta de alto impacto es mucho menos trabajo y tiene un impacto general mucho más positivo para la aplicación (y su mantenibilidad en el futuro).

Reutilización, pero no en las capas exteriores

Nadie está prohibiendo la reutilización. Una de las preocupaciones que a menudo se expresan sobre el patrón Mediator es que se termina con código duplicado en diferentes manejadores.

Pero fíjate de nuevo en que los manejadores de nuestra aplicación de aerolíneas representan la parte superior, o la capa exterior de nuestra aplicación; sus características.



Si terminas con una lógica ligeramente compleja en un manejador, puede ser que el manejador necesite una pequeña refactorización.

Fíjese en el `Pilot` conjunto.

La existencia de este agregado sugiere que hay suficiente lógica de negocio en juego con los pilotos como para que tenga sentido modelar esa lógica en un agregado.

> **Los agregados son un concepto clave del Domain Driven Design**
>
> El Diseño Dirigido por el Dominio (DDD) es un enfoque para la creación de software que se centra firmemente en el dominio del negocio y representa ese dominio de forma muy explícita en el código (para que sea más fácil de escalar y mantener).
>
> Está un poco más allá del alcance de este libro, pero la idea es que usted puede modelar su aplicación como una serie de agregados donde cada agregado representa una parte distinta del dominio del negocio.
>
> Cuando quieres hacer algo con un agregado lo "hidratas" a partir de datos (normalmente de una base de datos) realizas acciones a través de sus métodos públicos y luego guardas los cambios (en el caso de EF Core, llamando literalmente a `SaveChangesAsync` para persistir su estado modificado).

En este caso el `Piloto` agregado representaría un Piloto en nuestro sistema.

Para procesar un piloto que se retira del servicio, el `RetireHandler` cargaría una instancia del método que `Piloto` agregado (de la base de datos) e invocar su `Retirar` se encargaría de haciendo todo lo necesario para retirar a un piloto del

sistema. Algo parecido a esto...

Piloto (agregado)

```
public class Piloto {

    private DateTime _retired;

    public void Retire(){
        _retired = DateTime. Now;
    }

}
```

El gestor guardaría entonces los cambios (persistiría el estado actualizado).

Así que sí, puede tener sentido empujar la lógica a otro lugar cuando se reutiliza entre múltiples (en este caso, en el agregado de dominio).

Pero hágalo caso por caso cuando una característica determinada le lleve en esa dirección.

Intenta centrarte en el modelado de tu dominio, no sacando código arbitrariamente en un servicio, repositorio, ayudante o gestor separado sólo porque no quieres escribir código "de aspecto similar" dos veces.

La sequedad está sobrevalorada

Unas palabras sobre la duplicación.

A muchos nos han inculcado que debemos evitar almacenar datos "duplicados".

DRY (o "don't repeat yourself") nos haría tomar cualquier cosa que muestre el más mínimo indicio de ser "datos duplicados" y consolidarlos en un solo lugar.

Pero ya hemos visto a dónde nos lleva esto, directamente a los modelos de vista compartidos y a todos los problemas que eso conlleva.

Me hago eco de los pensamientos de Jimmy Bogard sobre este tema y asumo la postura por defecto de que es más importante modelar con precisión tus características que fijarte en si los datos con los que estás trabajando se parecen o no a los datos que estás manejando en otro lugar.

> "Soy súper cuidadoso a la hora de encontrar duplicidades: si las pantallas son literalmente la misma vista renderizada, entonces hay un solo viewmodel. Si no, las mantengo separadas".
>
> -- Jimmy Bogard

Esto es cierto para los viewmodels (la **respuesta** de nuestros manejadores) pero también vale la pena tenerlo en cuenta mientras trabajas con el resto de tu aplicación.

Una vez conocí a alguien que era aficionado al fútbol, padre y juez del condado.

Ahora bien, técnicamente sí es una persona. Pero me sorprendería mucho que fuera al juzgado y se comportara igual que en el fútbol.

Dudo que señale con el dedo a los acusados en el juzgado un lunes por la mañana y coree "¡no sabes lo que haces!" como hace un sábado por la tarde cuando una decisión polémica va en contra de su querido equipo de fútbol.

Asimismo, lo menos interesante de los pilotos y auxiliares de vuelo es que son personas, que tienen nombre.

Por muy tentador que sea crear una tabla genérica de "personas", o crear un objeto de dominio "persona" y utilizarlo cada vez que tratemos con "personas", la realidad es que los pilotos, los auxiliares de vuelo, los aficionados al fútbol y los jueces funcionan de formas completamente diferentes en contextos distintos.

El almacenamiento es barato, y el contexto es clave

El enfoque de las rebanadas verticales nos obliga a considerar cómo modelamos los conceptos y los comportamientos en nuestra aplicación.

En DDD los contextos delimitados representan áreas del dominio de negocio que se intenta modelar.

En efecto, eso es lo que vemos en juego aquí, con Pilotos y Asistentes representados en sus propias "islas" en código, con sus propios comportamientos, datos y matices.

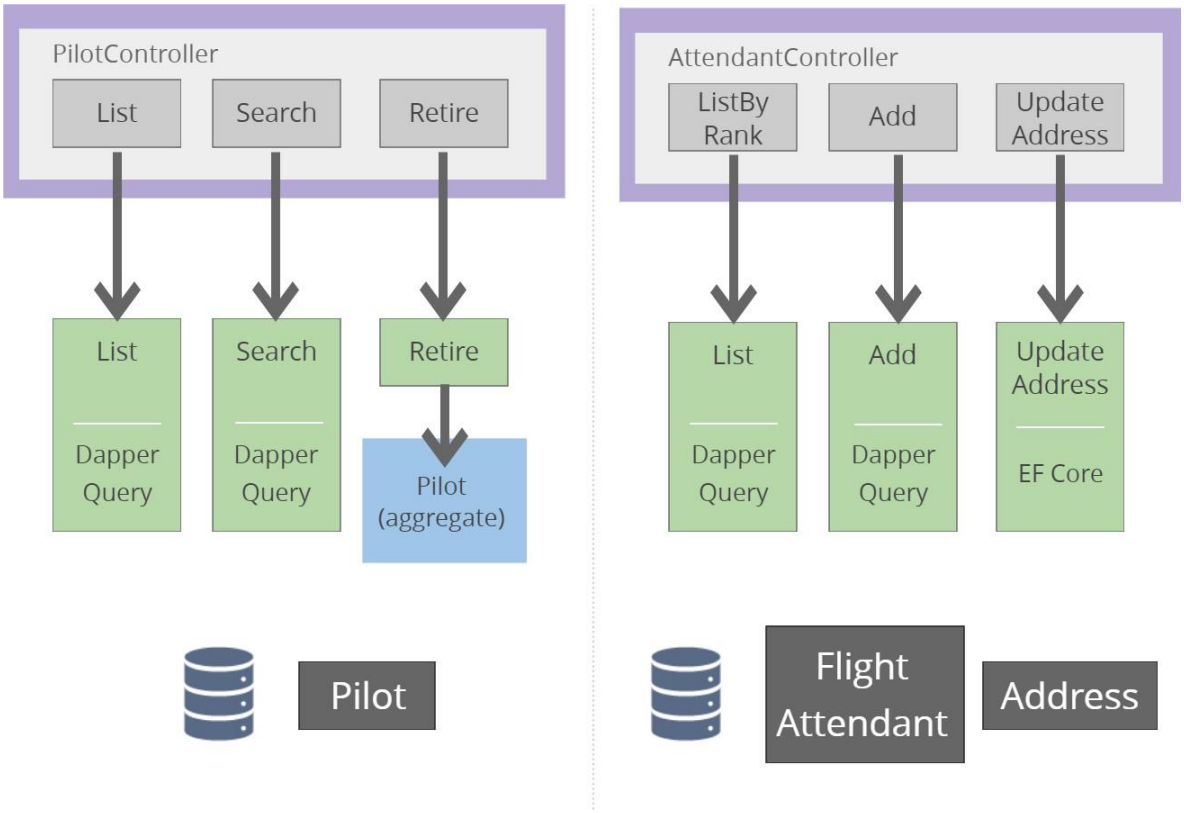
Y si un auxiliar de vuelo decide reciclarse y convertirse en piloto, simplemente archivaríamos su información en la parte del sistema dedicada a los auxiliares de vuelo y lo registraríamos como piloto en la parte del sistema dedicada a los pilotos; no hay ninguna **necesidad empresarial** imperiosa de compartir detalles (nombre, edad, etc.) entre ambos y, con toda probabilidad, querríamos ejecutar cualquier lógica empresarial que suele ejecutarse al registrar a los pilotos de todos modos.

> **¿Cómo se ve en el mundo real?**
>
> Cuando nos enfrentamos a decisiones como "¿cómo debemos modelar esto?" me parece súper importante considerar el dominio del mundo real.
>
> En realidad, cuando alguien se inscribe en esta aerolínea como piloto, sin duda rellena un formulario (quizá incluso en papel) con todos los datos necesarios (incluidas las cualificaciones, la experiencia, etc.).
>
> Es probable que haya algún proceso de investigación o comprobación antes de que se les permita volar para la compañía.
>
> Es este proceso el que estamos tratando de modelar, así que construye una Solicitud, una Respuesta y un manejador que actúe como el equivalente a rellenar ese formulario de papel, luego implementa todos los pasos que tienen que ocurrir para que ese proceso se complete.
>
> Utilice la terminología que se utiliza en la empresa al crear manejadores, objetos de dominio y (en última instancia) nombres de tablas, y le resultará mucho más fácil crear (y, en última instancia, mantener) sus funciones

Cortes verticales hasta el final

Obsérvese en el ejemplo de la aerolínea cómo el sistema se ha dividido en dos.

Esto no es en absoluto necesario, y para muchas aplicaciones sería una exageración, pero una de las ventajas de cortar la aplicación verticalmente es que se puede extender hasta los recursos como las bases de datos.



Así que en este caso podríamos tener potencialmente dos esquemas en la base de datos (para las tablas relativas a los pilotos y, por separado, a los auxiliares de vuelo) o incluso dos bases de datos distintas.

En sistemas más grandes puede ser ventajoso utilizar diferentes almacenes de datos para diferentes partes del sistema, por ejemplo, utilizando Postgres JSONB para almacenar datos para una parte de su sistema, pero una DB relacional más tradicional para otros.

La clave es que es mucho más fácil dividir el sistema cuando las divisiones se basan en características y grupos de características (contextos delimitados, o áreas de dominio) que en preocupaciones técnicas arbitrarias.

> **Pero, ¡mantenga la sencillez!**
>
> Vale la pena reiterar que lo simple suele ser lo mejor y que, sólo porque se pueda, no significa que se deba, especialmente cuando se trata de utilizar demasiadas tecnologías/enfoques diferentes en la aplicación
>
> Si hay una razón de peso para tener diferentes almacenes de datos, los cortes verticales lo hacen más fácil, pero para muchas aplicaciones esto sería una completa exageración y sólo haría las cosas innecesariamente más complicadas.

Actuar - Utilizar las carpetas de funciones

Puede que te imagines que puedes crear una carpeta llamada "Características" y seguir construyendo tu aplicación.

Esto es casi cierto, pero hay un pequeño paso extra si estás usando MVC.

MVC espera encontrar tus vistas en una carpeta Views, si quieres cambiar esto necesitas darle una pista de dónde buscar en su lugar.

Probablemente la forma más fácil de conseguir esto es simplemente añadir un paquete NuGet:

```
dotnet add package OdeToCode.AddFeatureFolders
```

Con esto añadido a tu proyecto MVC puedes configurar las carpetas de características en **Startup.cs**.

```
public void ConfigureServices(IServiceCollection services)
{
    servicios. AddControllersWithViews()
        . AddFeatureFolders();

    // otro código
}
```

Probablemente encontrará que la llamada `AddControllersWithViews` ya está ahí (si está usando ASP.NET Core 3.x o superior) por lo que sólo hay que añadirlo con `AddFeatureFolders` .

Bajo el capó esto actualiza las ViewLocations donde ASP.NET Core buscará las vistas.

Con esto en su lugar, ahora eres libre de poner tus vistas en una `Características` carpeta.

> ASP.NET buscará ahora en el mismo espacio de nombres que el controlador las vistas relevantes.
>
> Así que si tu controlador está en `Servicio de asistencia técnica.Funciones.Tickets` en tu(s) vista(s) utilizada(s) por ese controlador
> deben existir en el mismo espacio de nombres (básicamente, en la misma carpeta).

Si quieres saber más sobre las carpetas de características y algunos de los matices en torno a su uso, echa un vistazo al [repo de Github aquí](#).

Actúa - Construye un reportaje utilizando rodajas verticales con MediatR

Tendrás que empezar con una aplicación ASP.NET Core (Razor Pages, Web API, MVC, no debería importar cuál estés usando).

> MediatR también funciona con .NET Framework, y sí, ¡también encaja muy bien con Blazor!
>
> De hecho, ya lo he utilizado con WebForms, así que debería funcionar para prácticamente cualquier proyecto .NET que se te ocurra.

Cablear MediatR

El uso de MediatR tiene dos partes.

La primera es simplemente traer el paquete NuGet de MediatR para poder crear peticiones, respuestas y manejadores.

El segundo es configurar el framework de Inyección de Dependencia de tu elección para conectar estos manejadores. Es este segundo paso el que hace posible que MediatR localice y ejecute sus manejadores.

En primer lugar, tendrá que instalar el paquete NuGet de MediatR.

```
dotnet add package MediatR
```

El siguiente paso varía según el marco de trabajo de DI que estés utilizando.

Si utiliza la inyección de dependencias de Microsoft, deberá instalar el siguiente paquete NuGet.

```
dotnet add package MediatR.Extensions.Microsoft.DependencyInjection
```

Una vez que tengas eso puedes seguir adelante y cablear todo en **Startup.cs**

```
public void ConfigureServices(IServiceCollection services) {
    // otro código

    servicios. AddMediatR(Assembly. GetExecutingAssembly());
}
```

`AddMediatR` toma un ensamblaje como argumento; éste debe apuntar a la ubicación de su manipuladores.

En este caso todo (incluyendo los manejadores) está en un solo ensamblaje, así que lo he apuntado al ensamblaje "Ejecutando".

Si estás usando Microsoft Dependency Injection eso es todo, estás listo para ir.

Si está utilizando otro contenedor DI (como Lamar, SimpleInjector, Ninject, etc.) las instrucciones de configuración varían, por lo que su mejor opción es ir aquí y encontrar las instrucciones para su contenedor específico.

Cree su función

Construiremos una simple lista de "tickets" para nuestro sistema de asistencia técnica sólo para mostrar cómo funciona todo esto.

> **Haga más preguntas**
>
> En este caso se trata de un requisito bastante sencillo, mostrar una lista de entradas.
>
> Pero incluso con un requisito como este (con poca "lógica de negocio") es bueno hacer preguntas para entender completamente el requisito.
>
> A nivel superficial, hay consideraciones estándar como si los datos se paginarán, si sólo se devolverá el primer número x de entradas, si habrá que ordenarlas, etc.
>
> Pero siempre trato de profundizar más que eso. Dependiendo de cómo funcione tu equipo, y de lo estrechamente (o no) que estés involucrado en el diseño de una característica, no tengas miedo de hacer preguntas para entender realmente lo que estás construyendo.
>
> En este caso, me gustaría saber para qué sirve esta lista, ¿qué intenta conseguir el usuario cuando llega a esta función? ¿Busca una entrada específica? ¿Intenta tener una idea de lo que debe hacer primero? ¿Es importante ver los tickets "urgentes" antes que todo lo demás? Si es así, ¿qué se considera urgente?
>
> El cambio para centrarse en sus características no es sólo acerca de las abstracciones técnicas que usted utiliza o no, se trata de entender mejor lo que sus usuarios están tratando de lograr, para que pueda construir algo que realmente les ayuda a lograr ese resultado.
>
> Una vez que se tiene esa comprensión, el trabajo consiste en tomar medidas pragmáticas para escribir el código más simple que se pueda para hacer que esa característica cobre vida en el navegador.

Diríjase a una carpeta de su elección (si utiliza las carpetas de funciones, probablemente elegiría **Características/Tickets**).

Allí creamos una clase llamada `Lista` entonces sustituye la clase por defecto por esta:

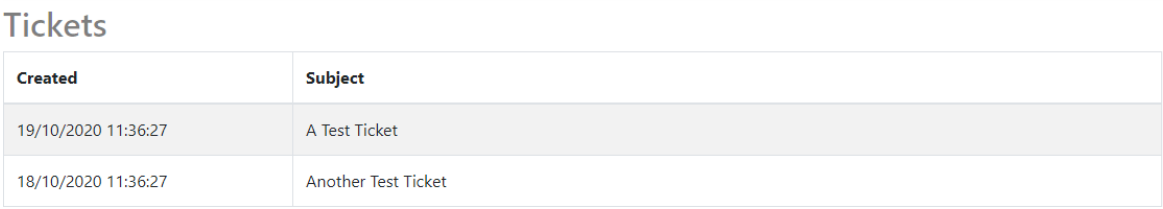
Lista.cs

```
public class Lista
{
    public class Consulta : IRequest< Modelo>
    {
    }

    public class Modelo
    {
    }

    public class Handler : IRequestHandler< Query, Model>
    {
        public Task< Model> Handle(Query request, CancellationToken
cancellationToken)
        {
            lanzar una nueva System. NotImplementedException();
        }
    }
}
```

Queremos mostrar todas las entradas (por ahora, de todos modos), por lo que probablemente no necesitemos añadir nada al `Solicite` (Query) todavía, en cuyo caso podemos centrarnos en modelar `Respuesta` Me resulta más

fácil si tengo una maqueta o algún tipo de diseño para trabajar. El (Modelo) clase.

`Respuesta` es esencialmente un viewmodel, así que lo que sea visible en el diseño querrá ser incluido en este viewmodel.



He aquí un primer intento.

Lista.cs

```
public class Modelo
{
    public IEnumerable< Details> Tickets { get; set; }

    public class Detalles
    {
        public string Subject { get; set; }
        public DateTime Created { get; set; }
    }
}
```

Ahora soy un gran fan de conseguir algo en funcionamiento "en la pantalla" tan pronto como sea posible cuando se trabaja con una característica. Con ese fin, probablemente cablearía algunos datos codificados en este punto, para tener algo con lo que trabajar mientras construyo la vista.

Lista.cs

```
public class Handler : IRequestHandler< Query, Model>
{
    public async Task< Model> Handle(Query request, CancellationToken
cancellationToken)
    {
        devolver el nuevo Modelo
        {
            Tickets = new List< Model. Detalles>
            {
                nuevo modelo. Detalles
                {
                    Creado = DateTime. Now,
                    Subject = "Un billete de
                    prueba"
                },
                nuevo modelo. Detalles
                {
                    Creado = DateTime. Now. AddDays(-1),
                    Subject = "Otro ticket de prueba"
                }
            }
        };
    }
}
```

Está claro que en este punto puedes devolver los datos que quieras para hacer pruebas.

Ahora tenemos un `Manipulador` podemos invocar y obtendremos una instancia de nuestro (o `Modelo`)

`Respuesta`

para que podamos seguir adelante e implementar nuestra vista MVC (o página Razor) para mostrar estos datos.

Me quedo con el MVC para este ejemplo:

Features\Tickets\TicketController.cs

```
public class TicketController : Controlador
{
    private readonly IMediator _mediator;

    public TicketController(IMediator mediator)
    {
        _mediador = mediador;
    }

    // GET
    public async Task< IActionResult> Index()
    {
        return View("Lista", await _mediator. Send(new List. Query());
    }
}
```

Esto utiliza `IMediador` para 'Enviar' la solicitud, y luego devuelve la respuesta para enlazarla con el MVC Ver.

CaracterísticasTickets\List.cshtml

```
@model Helpdesk.Slices.Features.Tickets.List.Model

@{
    ViewBag.Title = "Todas las
    entradas"; Layout = "_Layout";
}

<h2 class="text-black-50"> Entradas</h2>

<table class="table-striped table-bordered">
    <cabecera>
    <tr>
        <th class="w-25"> Creado</th>
        <th class="w-70"> Asunto</th>
    </tr>
    </head>
    <tbody>
    @foreach (var ticket in Model.Tickets)
    {
        <tr>
            <td> @ticket.Created</td>
            <td> @ticket.Subject</td>
        </tr>
    }
    </tbody>
</tabla>
```

Con Razor podemos mostrar fácilmente los valores del modelo. Debido a que el modelo ha sido creado específicamente para esta vista, este proceso es relativamente sencillo; se trata de crear el marcado para que coincida con la maqueta (o el diseño) y luego vincular la interfaz de usuario

a las propiedades pertinentes en el modelo.

Con esto tenemos una función en marcha, con un lugar claro al que acudir para seguir
it (`Ticket.List.Handler`
).
perfeccionando y ampliando

Próximos pasos

Una vez que tengas algo en marcha en la pantalla, podrás deshacerte de estos datos codificados, sustituyéndolos por datos de una fuente más realista.

Así que, dependiendo de tu proceso, aquí es donde podrías empezar a crear entidades de dominio, conectando EF Core o cualquier ORM que desees utilizar.

La buena noticia es que, una vez que los tengas instalados, puedes actualizar tu manejador para obtener tus datos y asignarlos a la respuesta (modelo), momento en el que tu función comenzará a mostrar datos reales.

Como siempre, es importante atenerse a lo más sencillo que funcione.

Si estás usando EF (Core) entonces esto es probablemente para inyectar una instancia de tu DBContext y usarlo para obtener tus datos (ver ejemplos anteriores).

La clave es evitar crear muchas abstracciones técnicas en este punto. Si se trata de una simple consulta puede ir directamente a este manejador.

Este enfoque aporta una serie de ventajas:

- El código de sus características es más fácil de localizar (cuando vuelva más tarde)
- El código es mucho más fácil de comprender cuando se vuelve a él (porque no está fragmentado y disperso en muchos lugares diferentes, o compartido con otras funciones)
- Puede realizar cambios en esta característica sin correr el riesgo de que esos cambios se extiendan a otras partes de la aplicación

Aplicar el enfoque de los cortes verticales a sus propios proyectos

Sé lo que estás pensando: todo esto está muy bien para estos proyectos de demostración, pero ¿cómo se supone que voy a aplicar esto a mis propias aplicaciones del mundo real?

Después de haber pasado por este camino en varias ocasiones, sé lo desalentador que puede ser intentar cambiar una aplicación existente en una nueva dirección.

He aquí algunos consejos para orientarle en la dirección correcta.

Empieza con algo sencillo

Las funciones más fáciles de implementar con MediatR son las consultas.

Si estás trabajando activamente en una aplicación y llegas a una etapa en la que necesitas recuperar y mostrar algunos datos, ¡es el momento perfecto para darle una vuelta a MediatR!

Piensa en la entrada (solicitud) y en la salida (el modelo que debe devolver esta consulta) y luego escribe el código más sencillo que puedas en el manejador para que todo funcione.

Migrar las consultas existentes

La implementación de una nueva función como ésta suele ser bastante sencilla, pero también es posible migrar el código existente utilizando un enfoque similar.

Migrar el código existente definitivamente trae más complicaciones y tiene el potencial de salirse de control (dependiendo del estado del código existente).

Pero las recompensas pueden cambiar el juego de sus aplicaciones heredadas... He aquí un enfoque práctico que reduce los riesgos:

Modelar la petición (consulta)

Averigüe en qué entradas (o parámetros, o argumentos) se basa la consulta existente (por ejemplo, si es para recuperar los detalles del producto, `podría` requerir un `Id de` producto).

Crear `Consulta` una clase con propiedades para representar esas entradas.

Modelar la respuesta (modelo)

Averiguar qué forma tienen los datos devueltos por esta consulta (para nuestro ejemplo de detalles del producto: el nombre del producto, el precio, etc.)

Crea `Modelo` una clase con propiedades para representar estos datos.

Crear el Manejador (inicialmente vacío)

Crear un Handler que implemente `IRequestHandler<Query, Model>` para su nuevo `Consulta` y `Modelo` y, a continuación, se pone a localizar todo el código de la consulta existente (¡puede estar repartido en varias clases!)

A menudo se inicia en un controlador, o incluso en un code-behind de WebForms, pero debería ser posible reunir todo el código para esta consulta.

En este punto, usted quiere mover o copiar todo el código relevante en (o junto a) su manejador.

Digo copia porque, si te encuentras con métodos que tienen múltiples referencias (son usados por más de una función de llamada) puede ser mejor tomar una copia del método para separar tu nueva función de propósito único (consulta) de todo lo demás.

El objetivo es aislar todo el código que hace funcionar esta consulta en un solo lugar, invocado por su manejador, y eliminar la estructura existente (servicios, repositorios, etc.).

Ahora tienes que hacer que tu Handler invoque lo que sea necesario para devolver los mismos datos que esta consulta devolvía antes. Esencialmente el manejador se convierte en el pegamento que mantiene todo esto junto.

Incluso puede ser útil en este punto alinear todo para que todo el código termine en el manejador. Eso no significa que tenga que quedarse ahí (ver la discusión sobre la refactorización en un capítulo anterior), pero eliminar toda la estructura para empezar puede hacer que sea mucho más claro lo que estás tratando.

Llámelo desde su página de controlador/webform/razor

Una vez que tenga eso en su lugar puede inyectar `mediator` en el controlador/WebForms existente código-behind/página Razor o lo que sea y llamar al `.Send(new Query())` para invocar su nuevo manejador.

Ya he visto este enfoque aplicado a grandes proyectos de WebForms heredados y el resultado final fue un código de WebForms mucho más sencillo que simplemente invocaba a Mediator para recuperar los datos, dejando el código de WebForms centrado únicamente en la presentación.

Esta separación entre la consulta y el código de presentación hizo mucho, mucho más sencillo migrar eventualmente ese sistema fuera de WebForms porque podíamos simplemente usar todas nuestras características (representadas como manejadores de MediatR) con un marco de trabajo de interfaz de usuario más moderno.

Migrar los comandos existentes

Si no es una consulta, lo más probable es que sea un comando.

Mientras que las consultas devuelven datos, los comandos los modifican de alguna manera (insertar, modificar, eliminar).

El enfoque es prácticamente el mismo que para las consultas, con la principal diferencia de que es en estos comandos donde a menudo se encuentra la lógica de negocio, y puede requerir una atención cuidadosa cuando se trata de utilizar cortes verticales.

Los principios generales siguen siendo los mismos y pueden simplificarse:

1. Crea una petición (el comando), una respuesta (opcional, sólo necesaria si necesitas devolver algo) y un manejador
2. Reunir toda la lógica existente
3. Conéctalo en el manipulador

Aunque parezca sencillo, es sin duda un reto mayor que la implementación de nuevas

funciones, pero incluso la migración de una que otra función se compone con el tiempo, y si se acostumbra a refactorizar con regularidad, se sorprenderá de lo mucho que se puede avanzar en un espacio de tiempo relativamente corto.

Lecturas adicionales (incluyendo ideas sobre la comprobación de las rebanadas)

Si quiere profundizar en los cortes verticales, aquí tiene algunos enlaces útiles que puede consultar.

Jimmy Bogard tiene varios proyectos de ejemplo en los que ha construido el ejemplo de la Universidad Contoso de Microsoft utilizando cortes verticales.

- Contoso University Razor Pages
- Contoso University MVC

Unas palabras sobre la prueba de los cortes verticales

Recomiendo encarecidamente explorar estos ejemplos, entre otras cosas porque demuestran un enfoque para probar sus rebanadas.

Jimmy emplea pruebas de integración que ejercitan toda la función (incluyendo el uso de una base de datos real para escribir/recuperar datos).

> Las pruebas de integración son un poco más lentas que las pruebas unitarias (que sólo ejercitan el código), pero hay que tener en cuenta que nunca se sabe realmente si una función funciona hasta que se ejecuta en una base de datos real.
>
> En general, he comprobado que este tipo de pruebas son las que más valor ofrecen y que las compensaciones merecen la pena.

Jimmy tiene sus pruebas de integración configuradas para que restablezcan el inicio de la base de datos antes de ejecutarse (utilizando una práctica biblioteca llamada Respawn). Esto significa que puede repetir la misma prueba una y otra vez sin preocuparse por los registros duplicados, pero también ejecutar varios pasos para configurar sus datos de prueba antes de ejercitar su función.

Si echas un vistazo a una de las pruebas, como esta de aquí, te darás cuenta de que heredan de una clase llamada `IntegrationTestBase` y también hacer uso de otra clase (estática) llamada `SliceFixture` .

Usando `SliceFixture` clase que es capaz de 'enviar' sus consultas MediatR directamente que desde la prueba.

```
var adminId = await SendAsync(new ContosoUniversity. Características. Instructores.
CreateEdit. Command
            {
                FirstMidName = "George",
                LastName = "Costanza",
                HireDate = DateTime. Hoy,
            });
```

`SliceFixture` expone otros métodos útiles para hacer cosas como la inserción de registros en la base de datos, por lo que puede insertar sus datos de prueba, ejecutar sus manejadores MediatR, y luego consultar la base de datos para comprobar que todo es como se espera.

He descubierto que este tipo de pruebas son muy útiles. Prueban la función a través de su API pública y no dependen de los detalles de la implementación en la prueba.

Digamos que, por ejemplo, usted tenía una prueba que utilizaba EF Core para recuperar una lista de tickets de nuestro sistema ficticio de asistencia técnica, pero esa consulta se ejecutaba muy lentamente y entonces decidió cambiarla por Dapper.

Una prueba de integración como esta no se preocupa de cómo funciona el manejador. Se trata de configurar algunos tickets de prueba en el sistema y luego invocar el manejador (usando `Send`) y comprobar que los datos correctos volvieron.

Podrías arrancar todo el código de EF Core, sustituirlo por Dapper y volver a ejecutar la prueba y (suponiendo que la consulta sea correcta) la prueba seguiría pasando.

En conclusión: no hay balas de plata

No hay ningún patrón, técnica o marco mágico que haga que su aplicación sea fácil de construir, escalar y mantener.

Pero hay algunos principios fundamentales que afectan directamente a lo fácil o difícil que es para usted transformar las ideas y las peticiones de características en un software que funcione.

Una vez que se aleja de las abstracciones técnicas arbitrarias que dividen conceptos que de otro modo estarían cohesionados en múltiples piezas y los dispersan por toda la aplicación, puede centrarse en lo que realmente importa;

Comprender, modelar y perfeccionar los requisitos empresariales.

Tomar cada requisito como viene y tratar de entenderlo realmente.

Sé curioso, haz preguntas, aclara cualquier cosa que no sea obvia (para ti) hasta que sepas lo suficiente para
explicarlo a otra persona.

Necesitas un punto de partida, para empezar a convertir tu comprensión en código.

Los cortes verticales se apoyan en una única abstracción ligera para dar ese punto de partida a sus características.

Cuando escriba su código, sea pragmático, preocúpese menos de la "duplicación" y más de ser específico: específico en los nombres que utiliza y en los datos que devuelve.

Haz lo más sencillo y prepárate para refactorizar a medida que tu código se amplíe y aumente su complejidad.

Comience su solicitud con el pie derecho y puede que descubra que la solicitud de cambio del lunes por la mañana es un trabajo de 5 minutos después de todo.

Feliz codificación...

Descargue la demo de la aplicación Helpdesk

Antes de que te vayas, he creado un pequeño ejemplo de aplicación de Helpdesk para demostrar los conceptos que hemos cubierto en este libro.

Encontrará una versión construida con un enfoque de capas más "tradicional" y otra versión construida con rodajas verticales y MediatR.

Descargue el código aquí

Consulta el documento incluido `Léame.md` para obtener más detalles sobre cómo ponerlos en marcha.

Cuando eches un vistazo, aquí tienes algunas cosas en las que fijarte:

Versión de capas

- Observe cuántas carpetas más hay en la carpeta raíz
- Obsérvese también que las carpetas representan conceptos técnicos (repositorios, servicios, modelos, etc.) y no características
- Echa un vistazo a **startup.cs** y observa cuántos servicios tienes que registrar (prácticamente todos los aspectos del sistema tienen al menos un servicio y un repositorio que se registran aquí)
- Echa un vistazo al método `List a` en `UserRepository` . Este es todavía un proyecto pequeño pero ya estamos en una lógica condicional para que este método funcione (porque se comparte entre diferentes servicios)

Versión en rodajas

- Hay menos carpetas en el nivel raíz, y la más importante es la carpeta **Features** Echa
- un vistazo a **startup.cs** y fíjate en que no hemos tenido que registrar ningún servicio adicional porque la lógica de esta aplicación era lo suficientemente sencilla como para mantenerla en los manejadores de MediatR
- Imagínese que tiene que hacer un ajuste en cualquiera de estas funciones (por ejemplo, la lista de entradas). ¿Cómo de fácil es localizar el código que necesita actualizar (digamos, para mostrar un campo extra en la Lista)? Ahora intente encontrar exactamente el mismo código para actualizarlo en la versión de Layers y vea cuál parece más sencillo.