

La Norma 0

Introducción

Esto es The Standard. Una colección de décadas de experiencia en la industria de la ingeniería. Lo he escrito para ayudarle a orientarse en el amplio océano del conocimiento. El Estándar no es perfecto y nunca lo será, y refleja la continua evolución de la industria de la ingeniería. Aunque la haya escrito una sola persona, es la recopilación de los pensamientos de cientos de ingenieros con los que he tenido el honor de interactuar y de los que he aprendido a lo largo de mi vida.

La norma contiene cientos de años de experiencias colectivas de muchos ingenieros diferentes. Como he viajado por todo el mundo y he trabajado en varios sectores, he tenido la oportunidad de trabajar con muchos tipos de ingenieros: algunos eran científicos locos que se preocupaban por los detalles más pequeños de cada rutina. Y otros han sido ingenieros de negocios que se preocupaban más por los resultados finales que por los medios para llegar a esos resultados. Además de otros, he aprendido de todos ellos: qué es lo que hace que una guía de ingeniería sencilla pueda iluminar el camino para que todos los demás ingenieros se inspiren en ella y, con suerte, la sigan. Por ello, he elaborado esta Norma, con la esperanza de que sea una brújula para que los ingenieros encuentren la mejor manera de diseñar soluciones que, con suerte, cambien el mundo.

Esta Norma es un llamamiento a los ingenieros de todo el mundo: léanla y hagan extractos de sus experiencias y conocimientos para enriquecer una Norma de ingeniería digna de la industria del software. Hoy vivimos conociendo los orígenes de la tierra, del hombre y de todos los animales. Sabemos lo caliente que está el agua hirviendo; lo larga que es una yarda. Los capitanes de nuestros barcos conocen las medidas precisas de latitud y longitud. Sin embargo, no tenemos ni carta ni brújula para guiarnos por el ancho mar del código. Ha llegado el momento de conceder a esta gran embarcación nuestra la misma dignidad y el mismo respeto que las demás normas definidas por la ciencia.

El valor de esta norma es inmenso para aquellos en la industria que todavía están encontrando su camino o incluso aquellos que han perdido su camino, y la norma puede guiarlos hacia un futuro mejor. Pero lo más importante es que la Norma está escrita para todos, por igual, para inspirar a todos los ingenieros o futuros ingenieros a centrarse en lo que más importa de la ingeniería: su propósito, no sus tecnicismos. Cuando los ingenieros tienen algún tipo de Norma, he observado que empiezan a centrarse más en lo que se puede lograr en nuestro mundo actual. Y cuando un equipo de ingenieros sigue algún tipo de Norma, su energía y su enfoque se centran más en lo que se puede lograr, no en cómo se debe lograr.

He recopilado, y luego redactado, esta Norma, con la esperanza de que elimine gran parte de la confusión y permita a los ingenieros centrarse en lo que más importa: utilizar la tecnología como medio para fines superiores y establecer sus objetivos equivalentes. El arte y la ciencia del diseño de software han recorrido un largo camino y han demostrado ser una de las herramientas más poderosas que una persona puede tener hoy en día. Merece una introducción adecuada, y la forma en que educamos a los jóvenes al respecto es importante.

En esencia, The Standard es mi interpretación de los principios SOLID y de muchas otras prácticas y patrones que siguen enriqueciendo nuestros diseños y desarrollos para conseguir sistemas verdaderamente sólidos. El objetivo de la norma es ayudar a todos los ingenieros a encontrar orientación en su trabajo diario. Pero, lo que es más importante, la Norma puede garantizar a todos los ingenieros que, cuando necesiten construir sistemas robustos que puedan aterrizar en la luna, resolver los problemas más complejos y garantizar la supervivencia de la humanidad y su evolución, dispondrán de la orientación necesaria.

La Norma es intencionadamente agnóstica desde el punto de vista tecnológico. Sus principios pueden aplicarse a cualquier lenguaje de programación, y su triple fundamento puede guiar cualquier decisión de desarrollo o diseño más allá del software. Utilizaré C# en el .NET para sólo materializar y realizar los conceptos de esta Norma. Pero sepan que en las primeras etapas de la formación de este Estándar, yo estaba usando fuertemente Scala como lenguaje de programación. El Estándar no estará atado a ninguna tecnología en particular, ni será una limitación para aquellos que quieran seguirlo, independientemente de su lenguaje de preferencia. Pero, ¿qué es lo más importante de La Norma? También se pretende que sirva de inspiración para que las generaciones venideras de ingenieros lo sigan, lo mejoren o inventen el suyo propio. La alternativa es construir software sin estándares, lo que supone un caos y una injusticia a la hora de invertir el mejor tiempo en los mejores efforts. Nuestra industria actual es un caos en términos de estandarización. Personas poco cualificadas pueden tener o tomar posiciones de liderazgo e influir en aquellos que están mucho más cualificados para tomar decisiones desafortunadas. La norma es la opción de establecer una medida de experiencia, influencia y profundidad de conocimientos antes de tomar cualquier decisión.

La Norma es también mi obra de amor para el resto del mundo. Está impulsada y escrita con la pasión de mejorar la experiencia de la ingeniería y producir sistemas eficientes, robustos, configables, enchufables y fiables que puedan soportar cualquier reto o cambio que se produzca casi a diario en nuestra industria.

[Introducción a The Standard]

[*] Preguntas sobre The Standard

0.0 La teoría

0.0.0 Introducción

Al diseñar cualquier sistema, es de suma importancia que el diseñador respalde su diseño con una determinada teoría. Las teorías desempeñan un papel enorme a la hora de garantizar que los propósitos, los modelos y las simulaciones de su diseño sean cohesivos y extensibles dentro de un determinado dominio.

Cualquier sistema, por muy caótico que parezca, está obligatoriamente influenciado por al menos una teoría, ya sea acuñada por el diseñador o heredada de diseñadores anteriores o de sus sistemas.

Independientemente de qué o quién sea el influenciador, es importante que el diseñador comprenda plenamente la teoría que sigue o tendrá un impacto negativo en sus futuras decisiones en términos de ampliar su diseño para mantenerse al día con un universo en constante cambio y expansión.

Me he dado cuenta muy pronto de que cuanto más simple es una teoría, más fácil es para otros diseñadores adaptarla y ampliar su alcance más allá de los sueños del diseñador original. Un universo construido sobre patrones más sencillos puede hacer que quienes se maravillen con su belleza lo entiendan mejor y lo aprecien mucho más que quienes se rinden ante el hecho de que es complejo más allá de su comprensión.

Una teoría sobre el universo podría hacer que la vida tuviera mucho más sentido, enriqueciéndose con todo tipo de historias sobre la supervivencia, la evolución y la plenitud.

0.0.1 Encontrar respuestas

Al principio de mi vida, tuve problemas con la escuela. Nada de lo que se enseñaba tenía sentido para mí. Parecía que todo el mundo en la escuela estaba más preocupado por memorizar y regurgitar lo que había memorizado durante el examen que por comprender realmente de forma consciente lo que se enseñaba y cuestionar sus orígenes y validar sus propósitos.

Me di cuenta a una edad temprana de que necesitaba alguna ecuación mágica que me ayudara a distinguir entre lo que es verdad y lo que no. Lo que está bien y lo que está mal.

Lo que es impulsado por un propósito y lo que es una imitación para aquellos que tienen verdaderos propósitos.

Así que empecé a buscar una teoría que me explicara el mundo. Durante mis años de escolaridad me pusieron todo tipo de nombres. Pero no me importaba mucho porque mi corazón, mi mente y mi cuerpo estaban totalmente volcados en encontrar la respuesta a todo.

Al buscar respuestas, es importante mantener el corazón y la mente abiertos a todas las opciones. No dejes que ninguna estructura social o tradicional limite tu mente a la hora de buscar la verdad sobre el universo y abrazar las respuestas de todas partes.

Después de años y años de búsqueda, me decidí por una teoría que simplificaba la comprensión de todo para una persona sencilla como yo. La llamé La Tri-Naturaleza del Todo.

0.0.2 Tri-Nature

La teoría de la triple naturaleza afirma que todo en este mundo está compuesto por 3 categorías principales. Dependencias, propósitos y exposiciones. Cada uno de estos componentes desempeña un papel crucial para la supervivencia de su sistema, su evolución y su cumplimiento.

Hablemos aquí de estos componentes.

0.0.2.0 Propósito

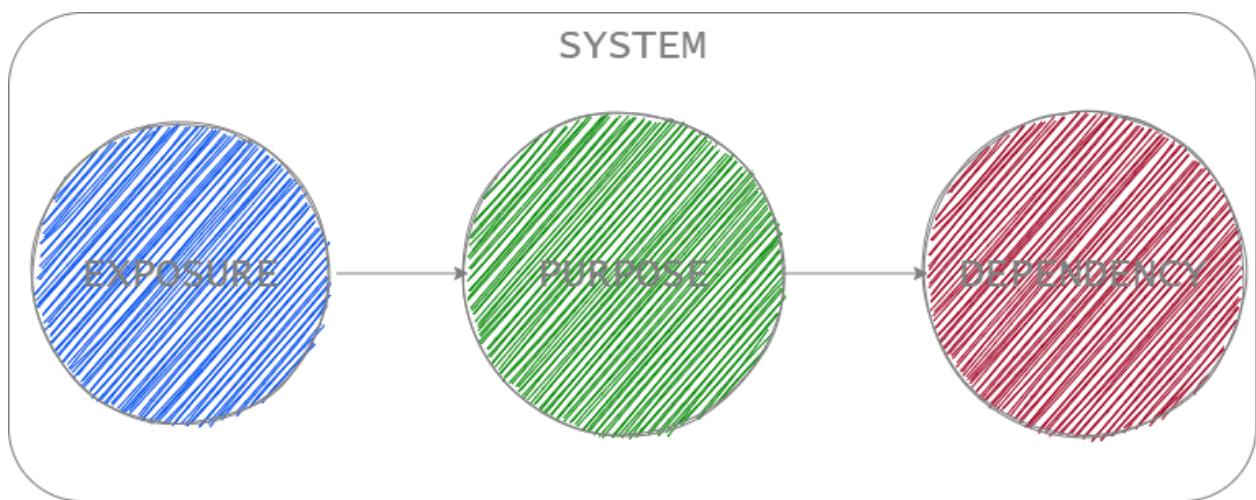
Todo lo que nos rodea tiene un propósito. Fue creado y diseñado con una razón determinada en la mente de su creador. Diseñamos coches para llevarnos del punto A al punto B. Diseñamos tazas para beber, platos para comer y zapatos para caminar. Todo lo que existe tiene un propósito central que rige su diseño y legitima su existencia.

0.0.2.1 Dependencia

Pero todos los sistemas que existen deben tener una dependencia de una forma u otra en orden. Por ejemplo, nosotros, como sistemas biológicos, dependemos de la comida y el agua para sobrevivir. Los coches dependen del petróleo o la electricidad. Los sistemas informáticos dependen de la energía y la electricidad, y así sucesivamente. Todos los sistemas, por pequeños o grandes que sean, e independientemente de su impacto e importancia, deben tener algún tipo de dependencia.

0.0.2.2 Exposición

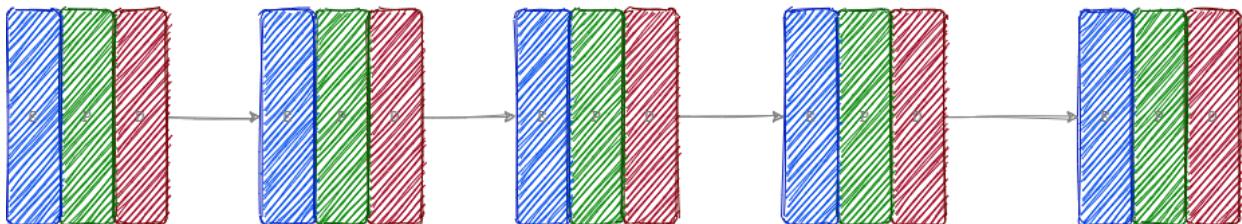
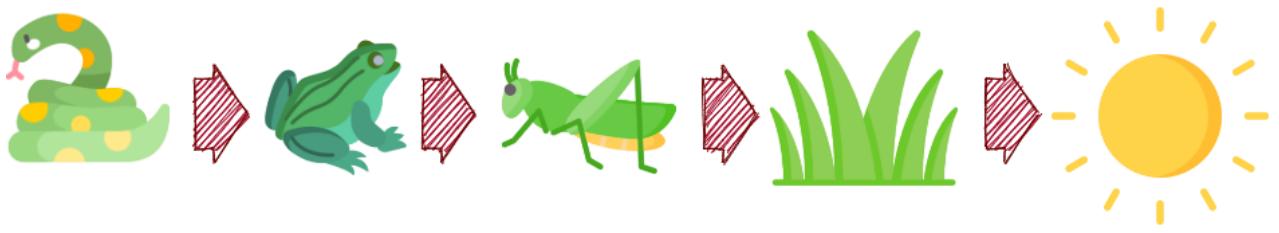
Pero para que una dependencia se convierta en tal, tiene que exponerse de alguna manera para que otros sistemas dependan de ella. Por ejemplo, las tomas de corriente son una capa de exposición para que las fuentes de energía permitan a otros sistemas enchufarse y consumir sus servicios. Las gasolineras son capas de exposición para que los tanques de petróleo enterrados almacenen ese petróleo. Y así con cada sistema que existe, necesita exponerse para permitir que otros sistemas se integren con él y consuman sus capacidades.



0.0.3 Todo está conectado

En el esquema más amplio de las cosas, todos los sistemas que existen están conectados entre sí. Un ejemplo sencillo de esto es la cadena alimentaria en la naturaleza. El sol depende de la hierba para crecer, los saltamontes son consumidores de la hierba, las ranas se alimentan de los saltamontes, las serpientes se alimentan de las ranas y así sucesivamente.

Cada miembro de la cadena alimentaria es un sistema que tiene dependencias, propósitos y exposición.

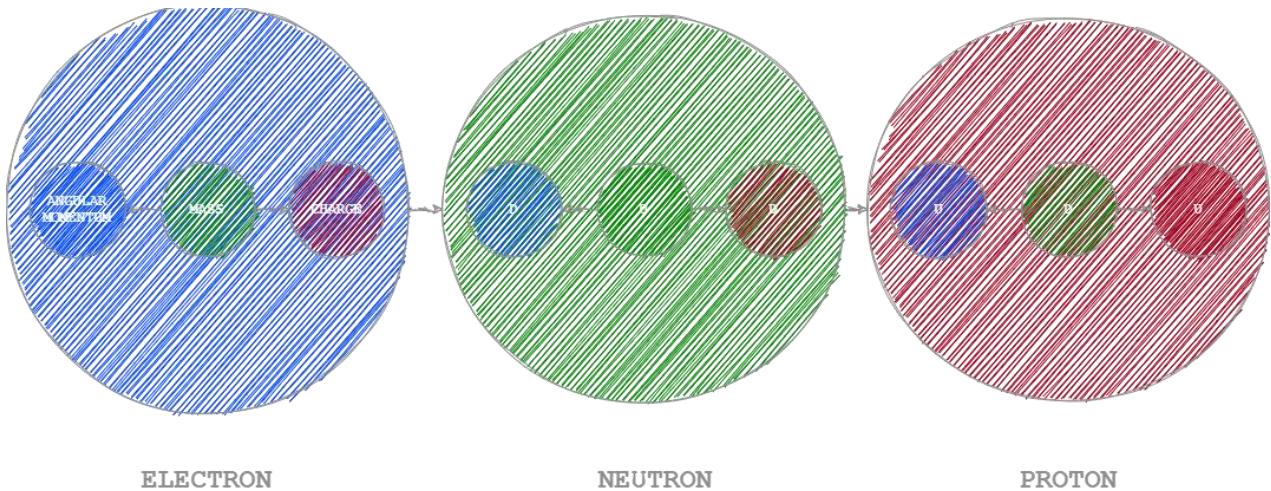


Dado que los sistemas informáticos no son más que una reflección de nuestra realidad. Estas integraciones de sistemas representan una cadena de dependencias infinitas en la que cada uno de estos sistemas depende de uno o más sistemas para cumplir su propósito. Una simple aplicación móvil podría depender de un sistema backend para persistir sus datos. Pero el sistema backend depende de un sistema basado en la nube para almacenar los datos. Y el sistema basado en la nube se apoya en el sistema de archivos para realizar operaciones básicas de persistencia, etc.

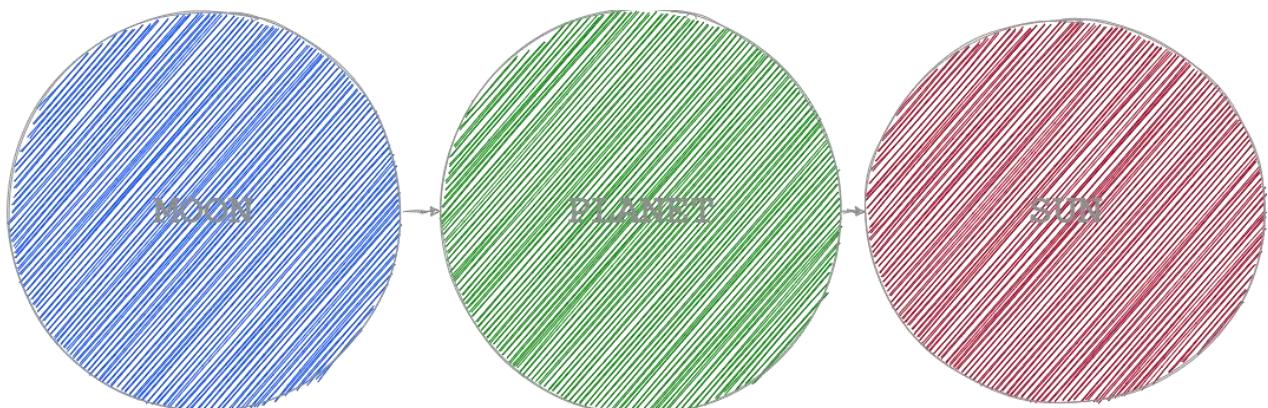
0.0.4 Patrón fractal

El patrón de la Tri-Naturaleza también podría percibirse en la escala más baja de cualquier sistema, así como en la más alta. Es lo que llamamos un patrón fractal. Cada sistema que existe está infinitamente compuesto por tres componentes, donde cada uno de ellos tiene también tres componentes y así sucesivamente.

Por ejemplo, el componente más pequeño conocido en el universo son los quarks dentro del neutrón de un átomo. Estos quarks son tres componentes, dos quarks down y un quark up. Pero si nos alejamos un poco, veremos que el sistema más grande en el que residen estos quarks también está formado por tres componentes que son los electrones, los protones y los neutrones.



Si nos alejamos del nivel subatómico hasta el sistema solar, el patrón continúa repitiéndose a escala masiva. Nuestro sistema solar se compone de un sol, planetas y lunas, y caen exactamente dentro de los patrones de propósito y exposición de la dependencia como los componentes en el nivel sub-atómico como sigue:



Y si ampliamos la escala, descubrimos que las galaxias están hechas de polvo, gas y materia oscura.

El patrón de la Tri-Naturaleza sigue repitiéndose en todos los aspectos de nuestra vida. Todos los componentes de nuestro universo, desde las partes subatómicas más pequeñas hasta las galaxias y los sistemas solares a escala masiva, siguen la misma regla.

0.0.5 Diseño y arquitectura de sistemas

Ahora es evidente que se ha descubierto una buena teoría que podemos seguir para diseñar sistemas. Ahora podemos desarrollar cada uno de los componentes de nuestro software de acuerdo con La Tri-Naturaleza del Todo. Las reglas y directrices que rigen el diseño de software de acuerdo con La Teoría se denominan El Estándar. Se refiere al estándar universal en el diseño de sistemas en todas las materias.

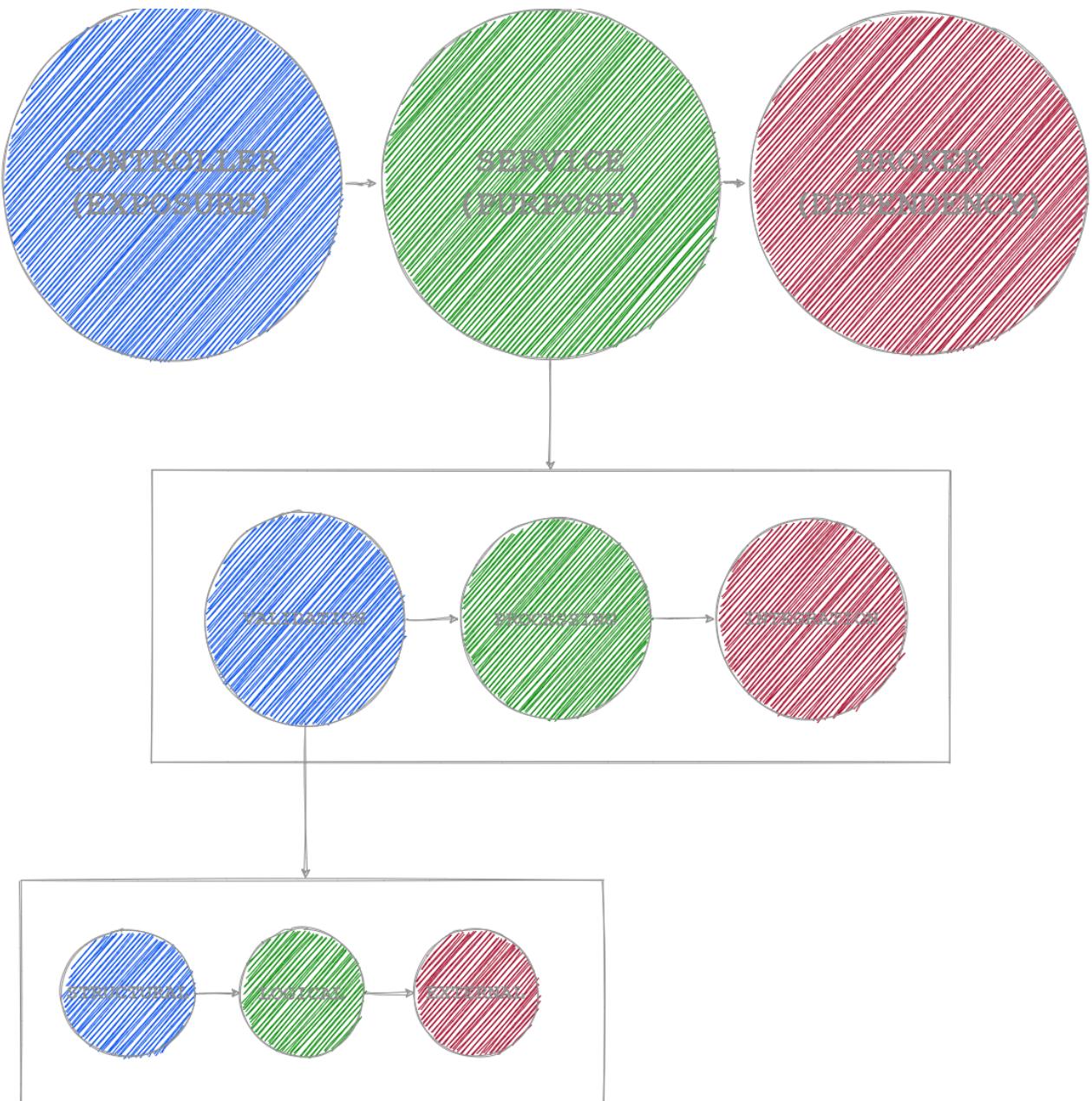
La Norma dicta en la arquitectura de bajo nivel que todo sistema debe estar compuesto por corredores (dependencias) y servicios (propósitos) y expositores (exposiciones).

Por ejemplo, al diseñar una simple API RESTful, puede que necesitemos integrarnos con un sistema de base de datos, luego validar los datos entrantes basándonos en ciertas reglas de negocio y después exponer estas capacidades al mundo exterior para que los consumidores de la API puedan integrarse con ella.

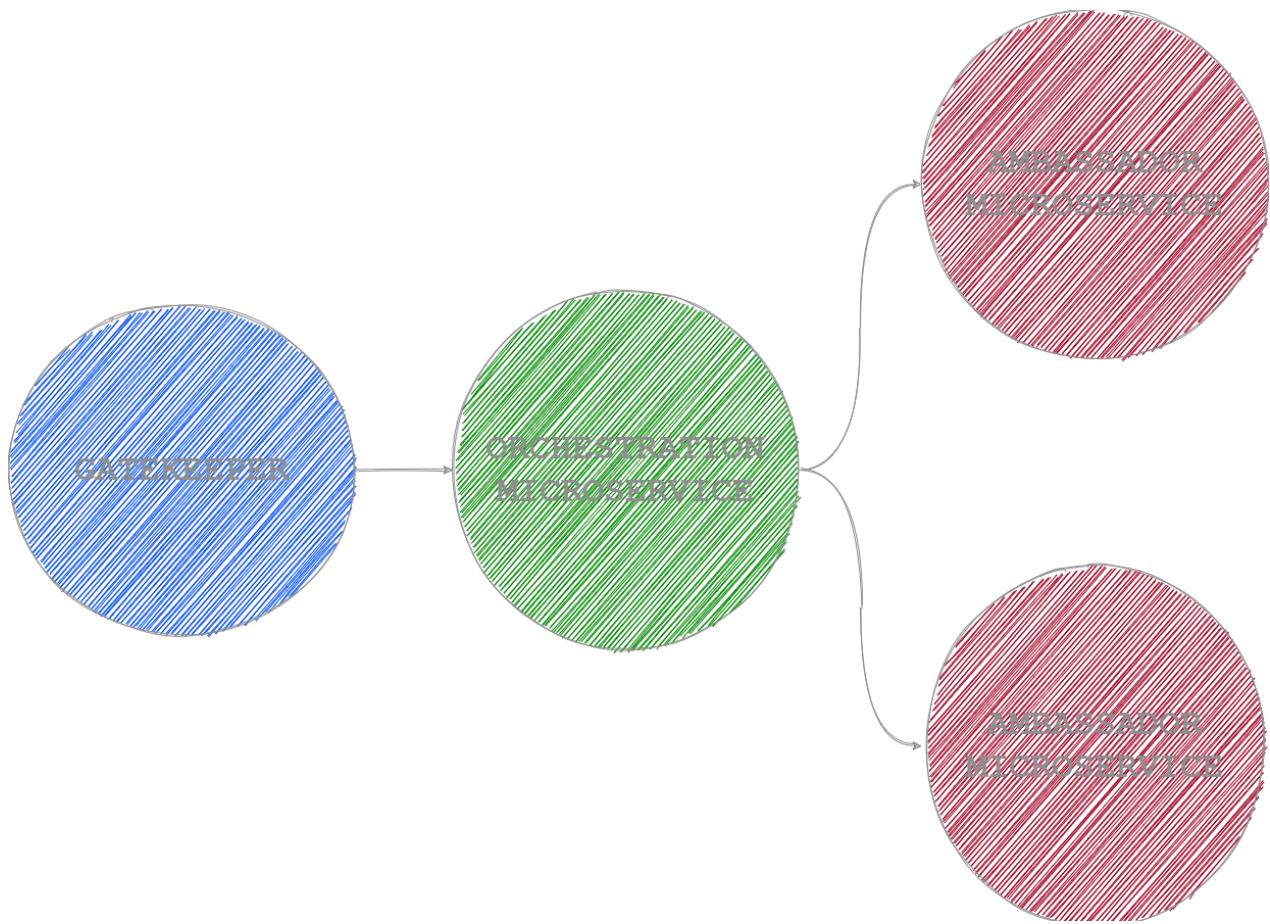
Según The Standard ese sistema quedaría así:



Al profundizar en cualquiera de estos componentes, se repite el mismo patrón. Por ejemplo, un servicio se compone de componentes de validación, componentes de procesamiento y componentes de integración. Y luego, si nos acercamos un poco más, estos mismos componentes de validación están formados por tres componentes fineros que son las validaciones estructurales, las validaciones lógicas y las validaciones externas. El patrón continúa hasta el nivel más bajo de nuestro diseño, como se muestra aquí:



El mismo patrón también se aplica a los sistemas más grandes si salimos del ámbito de un sistema y entramos en los sistemas modernos distribuidos, como las arquitecturas de microservicios:



En un sistema distribuido, algunos servicios desempeñan el papel de embajadores ante los recursos externos o locales, lo que equivale a un componente intermediario a nivel de servicio. Pero entonces debe entrar en juego un componente impulsado por el propósito de orquestar los flujos de negocio mediante la combinación de una o muchas operaciones primitivas de consumo de recursos de estos servicios de embajadores. La parte final que es la capa de exposición, una fina capa de gatekeeper que se convierte en el primer punto de contacto entre el mundo exterior y su arquitectura de microservicios.

El mismo patrón de la tríada seguirá repitiéndose en varios sistemas, ya sean grandes en varias organizaciones o pequeños en un solo servicio.

0.0.6 Conclusión

En conclusión, la Tri-Naturaleza del Todo es la teoría que impulsa El Estándar. Cada uno de los aspectos de las reglas y directrices de La Norma está fuertemente influenciado por la teoría de la trinaturaleza. Pero es importante entender que la teoría va mucho más allá del diseño de un sistema de software. Puede aplicarse a los estilos de gestión, a la escritura de libros, a la preparación de comidas, al establecimiento de relaciones y a cualquier otro aspecto de nuestras vidas que vaya más allá del propósito de La Norma aquí.

Después de tantos años de investigación y experimentación con la teoría de la trinaturaleza, es evidente que funciona. Ayuda a simplificar algunos de los sistemas más complejos que existen. Se adapta bien a nuestra intuición como seres humanos y simplifica aún más la tarea de los autómatas en el futuro para agilizar nuestros procesos de desarrollo de software y hardware y todo lo demás.

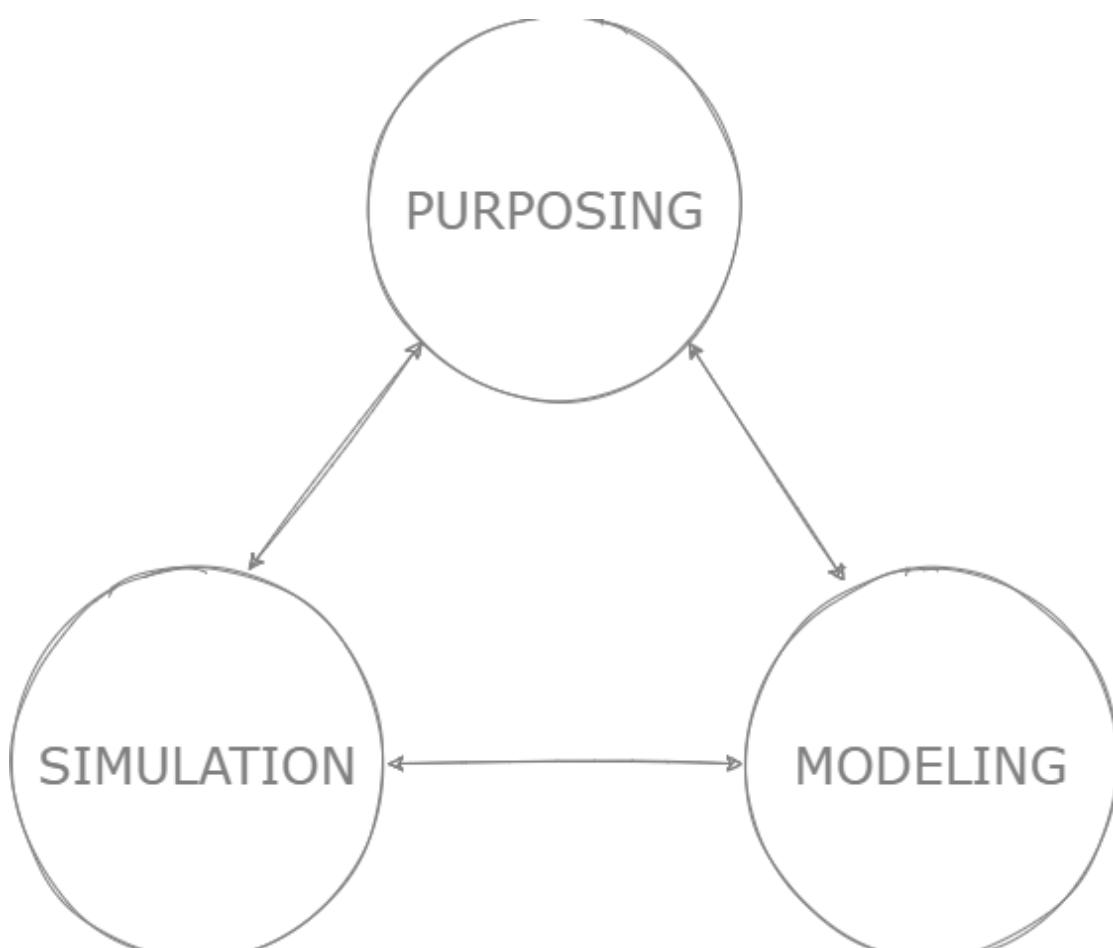
Por último, The Standard es un viaje continuo para seguir cuestionando la teoría de la triple naturaleza. Cuanto más nos adentramos en aguas inexploradas en términos de dominios empresariales, más descubrimos algunos territorios nuevos en los que mi teoría sigue siendo válida. Incluso para los sistemas más caóticos que existen, la teoría se aplica de cierta manera, aunque los componentes de dichos sistemas no se adhieran del todo a la forma de distinción de La Norma.

0.1 Propuestas, modelización y simulación

0.1.0 Introducción

La norma define el proceso de ingeniería de software en tres categorías principales. La propuesta, el modelado y la simulación. Cada uno de estos aspectos desempeña un papel crucial a la hora de guiar los recursos de ingeniería hacia la producción de una solución exitosa y el cumplimiento de un propósito particular.

El orden en que se siguen estos aspectos también es intencionado. Debe existir un propósito para dar forma al proceso de modelado. Y no se pueden simular interacciones sin modelos. Pero aunque ese orden al inicio del proceso de ingeniería es crucial, es importante entender que el proceso en sí es selectivamente iterativo. Un cambio en el propósito puede reflectar como un cambio en la simulación pero no necesariamente en el modelado. Un cambio en los modelos puede no requerir necesariamente un cambio ni en el propósito ni en la simulación.



0.1.1 Proponiendo

El proceso de creación de propósitos es nuestra capacidad para averiguar *por qué* necesitamos una solución. Por ejemplo, si tenemos un problema para saber cuántos artículos hay en el estante de una tienda de comestibles. Consideramos que el proceso manual de recuento es ineficaz y que es necesario implantar un sistema que garantice el recuento adecuado de los artículos.

Por lo tanto, proponer es encontrar una razón para llevar a cabo una acción. El razonamiento se basa en gran medida en la capacidad de observación. Nuestra capacidad de observar los problemas y, a continuación, ser capaces de articular el problema para idear mejor una solución que aborde el problema en cuestión.

Así, tenemos la observación, la articulación del razonamiento (el problema) y la intención de una solución. Todos estos aspectos constituyen la parte de la propuesta del software de ingeniería.

0.1.1.0 Observación

Vivimos en un mundo lleno de observables. Nuestra inspiración se dispara por nuestra ambición de conseguir más. Nuestros sueños revelan bloqueos en nuestro camino que tenemos que resolver para continuar nuestro viaje y cumplir nuestros sueños. Desde el momento en que un joven estudiante utiliza una calculadora para resolver una ecuación compleja hasta el momento en que ese mismo estudiante se convierte en astronauta, calculando la trayectoria de los satélites que orbitan nuestro planeta.

La observación es nuestra capacidad para detectar un problema que impide alcanzar un objetivo. Los problemas pueden ser tan sencillos como dar con el recuento adecuado de artículos en un estante de la tienda de comestibles. Hasta entender por qué no podemos captar imágenes de planetas que están a millones de años luz de nosotros. Todo esto es lo que los ingenieros describirían como un problema observable.

Cuanto mayor sea el propósito, más complejo será el problema. Pero estos propósitos más pequeños son la forma de entrenar nuestra mente para abordar los más grandes. Paso a paso, un problema cada vez.

0.1.1.1 Articulación

Describir lo observable es un arte en sí mismo. Porque describir correctamente un problema es la mitad del camino hacia su solución. Cuanto más clara sea la articulación del problema, más probabilidades habrá de que otros lo entiendan y nos ayuden a resolverlo.

La articulación no es siempre con las palabras. También lo es con figuras y formas. No es casualidad que algunas de las culturas antiguas más avanzadas hayan utilizado figuras y formas para describir su tiempo y su historia. Las figuras son un lenguaje universal, entendido e interpretado por cualquiera que pueda relacionarse con ellas mucho más rápido que el aprendizaje de un lenguaje hablado. De hecho, una figura o una forma podría ser la forma más óptima de ilustrar una idea, ya que sus imágenes son dignas de miles de palabras.

La articulación requiere la pasión para resolver el problema. Ya sea por escrito, dicho o ilustrado. Una mente apasionada transmitirá el mensaje oculto de la criticidad del problema a resolver. Articular un problema es una parte importante de la forma en que uno puede vender su solución. La capacidad de transmitir una idea a otros ingenieros y luego a los que van a invertir y utilizar esta solución es uno de los aspectos más importantes del software de ingeniería.

0.1.1.2 Solución

Una parte del propósito es la forma de cumplirlo. En la industria de la ingeniería, el cumplimiento de los objetivos no puede ser por cualquier medio. Un aspecto importante de por qué fracasan tantas piezas masivas de software en todo el mundo es porque el aspecto de la solución se pasó por alto como una parte trivial del propósito. Es posible que hayas oído hablar de aquellos que se vieron presionados por una fecha límite, por lo que decidieron recortar gastos para lograr el objetivo. En nuestra Norma, esto es una violación. Una solución no debe limitarse a cumplir un objetivo, sino que debe ser un propósito en sí mismo. Esto es en términos de optimización, legibilidad, configurabilidad y longevidad.

La solución como parte del propósito es la artesanía del software.

0.1.2 Modelado

El modelado es el segundo aspecto más importante de la ingeniería del software. Es nuestra capacidad para extraer modelos de los actores de cualquier problema. Ya sean estos actores seres vivos, objetos u otros. Por ejemplo, en un problema en el que intentamos contar los artículos de una estantería de un supermercado, un modelo sería para estos artículos. Extrayendo sólo los atributos que son relevantes para el problema que estamos tratando de resolver, y descartando todo lo demás.

Un ejemplo más sencillo sería detectar los artículos de una tienda de comestibles que son perecederos. El único atributo que nos interesa aquí es la fecha de caducidad del artículo. Todo lo demás, como la etiqueta, el color, el peso o cualquier otro detalle, queda fuera del alcance del proceso de

modelado y de la solución.

La modelización no puede existir sin un propósito. Ya que el propósito define el ámbito o el marco en el que debe producirse el modelado. Modelar sin un propósito deja la puerta abierta para atraer un número infinito de atributos que cada elemento del universo observable puede tener.

La relación entre los atributos de la finalidad y el modelado es proporcional. Cuanto más complejo sea el propósito, más probable será que el proceso de modelado requiera más atributos del mundo real para asemejarse en un prototipo.

Expresamos nuestros modelos en lenguajes de programación como una clase. El problema de los artículos perecederos antes mencionado puede representarse como sigue:

```
public class Artículo
{
    public DateTimeOffset ExpirationDate {get; set;}
}
```

El nombre del `clase` representa el *tipo* general del artículo. Dado que todos los artículos tienen exactamente el mismo atributo de tan genérico como puede ser. Fecha de caducidad entonces el nombre permanecerá

Ahora, imaginemos que nuestro propósito se hace un poco más complejo. Supongamos que el nuevo problema es poder identificar los artículos perecederos más caros para que la tienda pueda ponerlos a la venta antes que los artículos menos caros. En este nuestro modelo requeriría un nuevo atributo como `Precio` para que un ordenador programa o una solución puede determinar cuál es más valiosa. Así es como quedaría nuestro nuevo modelo:

```
public class Artículo
{
    public double Precio {get; set;}
    public DateTimeOffset ExpirationDate {get; set;}
}
```

0.1.2.0 Tipos de modelos

Los modelos rigen todo el proceso de simulación de un problema (y su solución). Los modelos en sí se dividen en tres categorías principales, Portadores de datos, Operativos y Configuraciones. Vamos a hablar de estos tipos en las siguientes secciones:

0.1.2.0.0 Modelos de soporte de datos

Los modelos de soporte de datos tienen un objetivo principal: transportar puntos de datos entre sistemas. Los modelos de soporte de datos pueden variar en función del tipo de datos que transportan. Algunos modelos portadores de datos transportarán otros modelos para representar un sistema complejo. Otros sólo representan referencias a los puntos de datos originales que representan.

Los modelos de soporte de datos de forma relacional pueden dividirse en tres categorías distintas. Estas categorías dejan mucho más claro cuáles son las áreas prioritarias en términos de desarrollo, diseño e ingeniería. Por ejemplo, no podemos empezar a desarrollar modelos secundarios/de apoyo si no tenemos primero nuestros modelos primarios. Hablemos de estas categorías en detalle:

0.1.2.0.0.0 Modelos primarios

Los modelos primarios son los pilares de todo sistema. Cualquier sistema no puede avanzar en términos de diseño e ingeniería sin una clara definición y una materialización de estos modelos primarios. Por ejemplo, si estamos construyendo un sistema de escolarización, modelos como el Estudiante, considerados modelos de Primaria.

Profeso Curso
y son

Llamamos a estos modelos primarios porque son autosuficientes. No dependen físicamente de algún otro modelo para poder existir. Lo que significa que

un determinado modelo de Primaria como Profeso puede seguir existiendo en un sistema de escolarización

si un registro existiera o no. Esto se llama dependencia física.

Los modelos primarios de un esquema de almacenamiento relacional no contienen claves externas ni referencias a ningún otro modelo físico.

Sin embargo, los modelos primarios pueden basarse conceptual o lógicamente en otros modelos.

Por ejemplo, un Estudia modelo tiene una relación lógica con un Profesor. Simplemente porque nunca puede haber un alumno sin un profesor y viceversa versa. A Estudia modelo también tiene una relación conceptual con su anfitrión y

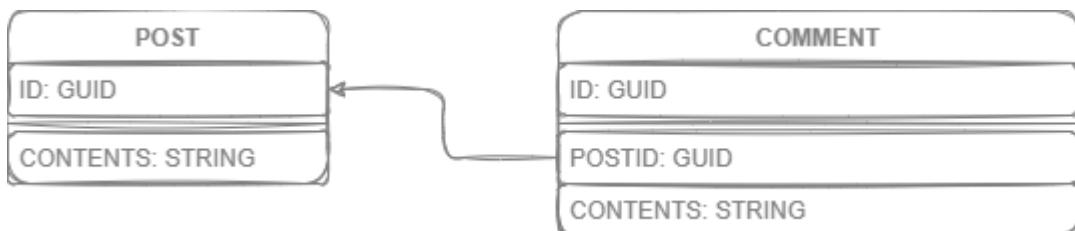
servicios de alojamiento vecinos. Por ejemplo, hay una relación conceptual entre un Estudia modelo y un Notificación modelo en términos de negocio flow. Cualquier alumno de cualquier centro escolar depende conceptualmente de las notificaciones para asistir a ciertas clases, realizar ciertas tareas o cualquier otro evento.

0.1.2.0.0.1 Modelos secundarios

Los modelos secundarios, por su parte, tienen una fuerte dependencia de los modelos primarios. En un modelo de base de datos relacional, los modelos secundarios suelen tener claves externas que hacen referencia a otro modelo en el esquema general de la base de datos. Pero incluso en los sistemas de almacenamiento no relacionales, los modelos secundarios pueden representarse como entidades anidadas dentro de una entidad mayor o tener una referencia suelta a otra entidad.

Hablemos de algunos ejemplos de modelos secundarios. ~~a~~omenta modelo en un plataforma de medios sociales no puede existir ~~sin un~~ ~~ri~~ modelo. Simplemente no se puede ~~icar~~

comentar algo que no existe. En una base de datos relacional, el modelo de comentarios sería algo así:



En el ejemplo anterior, un modelo secundario ~~Comentario~~ tiene un ~~PostId~~ foránea

haciendo referencia a la clave ~~primaria~~ en un modelo ~~Post~~. En un sistema no relacional,

Los modelos secundarios pueden identificarse fácilmente como objetos anidados dentro de una entidad determinada. He aquí un ejemplo:

```
{
  "id": "algún-id",
  "contenido": "alguna
entrada", "comentarios":
  [
    {
      "id": "comment-id",
      "content": "algún
comentario"
    }
  ]
}
```

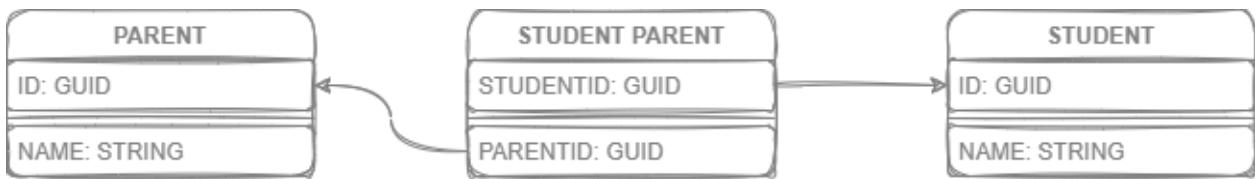
Los modelos secundarios en general pueden tener relaciones lógicas y conceptuales con otros modelos dentro de sus sistemas anfitriones, vecinos o externos. Sin embargo, sus posibilidades de tener estas relaciones conceptuales son mucho menores que las de los modelos primarios.

0.1.2.0.0.2 Modelos relacionales

Los modelos relacionales son conectores entre dos modelos primarios. Su principal responsabilidad es materializar una relación de muchos a muchos

entre dos entidades. Por ejemplo, un puede tener varios profesores; y un puede tener varios alumnos. En este caso, un modelo intermedio.

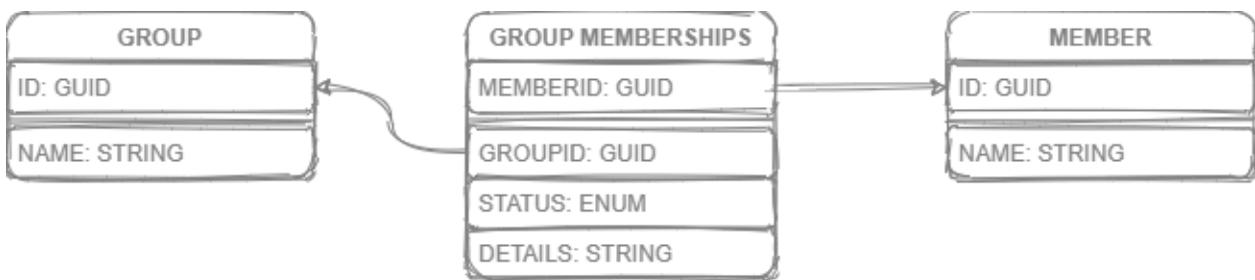
Se supone que los modelos relacionales no tienen ningún detalle. Sólo contienen referencias a otros modelos y esa es su clave primaria. Una clave compuesta que agrega dos o más claves foráneas dentro de ella. Veamos un ejemplo:



0.1.2.0.0.3 Modelos híbridos

Hay una situación en la que un modelo está conectando múltiples entidades pero también lleva sus propios datos. Desaconsejo encarecidamente seguir ese camino para mantener un nivel de pureza en el diseño de su sistema y controlar la complejidad de sus modelos. Sin embargo, a veces este enfoque es una opción necesaria para proceder con una determinada implementación o un flujo de negocio. En ese caso, podemos proponer un modelo híbrido que puede llevar ciertos detalles sobre la relación entre dos entidades independientes.

He aquí un ejemplo de modelo híbrido que puede darse en la realidad. En un escenario de supresión suave, un modelo híbrido puede describir la desvinculación entre dos entidades en una relación de muchos a muchos. Supongamos que un miembro del grupo no quiere seguir formando parte de un ~~determinado~~ grupo. Consideramos su pertenencia al grupo como ~~con~~ con un motivo adjunto sin llegar a eliminar completamente el récord. Así es como se vería:



En un modelo de datos no relacional, la integridad de las referencias puede ser un poco más floja, dada la naturaleza lineal de ese esquema. Los modelos híbridos combinan ambos modelos secundarios en la forma de referenciar los modelos primarios, e implementan una naturaleza relacional en la que permiten que múltiples entidades se relacionen entre sí sin exclusividad.

0.1.2.0.1 Modelos operativos

Los modelos operativos se dirigen principalmente al aspecto de simulación de cualquier sistema de software. Piense en todas las operaciones primitivas, complejas y de exposición que podría requerir un escenario sencillo para que se implemente una simulación con éxito. Supongamos que estamos tratando de resolver un problema en el que podemos simplificar las inscripciones de los estudiantes en alguna escuela. El proceso de registro requerirá alguna simulación para agregar la información de estos estudiantes en un sistema computarizado.

Los modelos operativos se encargarán de la exposición, el procesamiento y la integración de todo ese proceso, mediante la offeración de servicios que offers APIs/UIs para introducir, contabilizar, añadir e insertar/persistir la información de los estudiantes en algún sistema de escolarización.

Los modelos operativos pueden dividirse en tres categorías principales, que son: Integración, Procesamiento y Exposición. La Norma se centra en gran medida en los modelos operativos porque representan el núcleo de cualquier sistema en términos de flujos de negocio. Los modelos operativos son también el lugar al que se destinan la mayor parte de los recursos de desarrollo y diseño en cualquier eflorma de desarrollo de software.

Hablemos de los modelos operativos.

0.1.2.0.1.0 Modelos de integración (corredores)

La principal responsabilidad de los modelos operativos de integración es conectar cualquier sistema existente con recursos externos. Estos recursos pueden estar localizados en el entorno de ese sistema, como la lectura de la fecha o la hora actuales, o remotos, como la llamada a una API externa o la persistencia de datos en alguna base de datos.

A estos modelos de integración los llamamos Brokers. Desempeñan el papel de enlace entre los modelos operativos de procesamiento y los sistemas externos. He aquí un ejemplo:

```
public partial class ApiBroker
{
    public async ValueTask< Student> PostStudentAsync(Student student) => this.apiBroker.
        PostAsync< Student>(student, url);
}
```

El modelo de integración anterior, ofrece una capacidad para llamar a una API externa, mientras que abstrae los detalles de configuración lejos de los modelos operativos de procesamiento.

Al igual que cualquier otro tipo de modelo operativo, no contienen datos en su interior, sino que utilizan miembros de clase privados y constantes para

compartir datos internos
a través de sus métodos públicos y privados. El aquí como modelo
representa una simulación de integración con un sistema externo.

Hablaremos ampliamente de los corredores en los próximos capítulos para arrojar algo de luz sobre las reglas y directrices en el desarrollo de corredores con cualquier recurso o sistema externo.

0.1.2.0.1.1 Modelos de procesamiento (servicios)

Los modelos de procesamiento son los titulares de todas las simulaciones específicas del negocio. Cosas como las inscripciones de estudiantes, la solicitud de una nueva tarjeta de biblioteca o simplemente la recuperación de alguna información de los estudiantes basada en un determinado criterio. Los modelos de procesamiento pueden ser primitivos/fundamentales, de alto orden/procesamiento o avanzados/orquestadores.

Los modelos de procesamiento, en general, se apoyan en modelos de integración, o se apoyan en sí mismos, como los servicios de procesamiento computacional, o se apoyan entre sí.

Este es un ejemplo de un simple servicio fundacional/primitivo:

```
public partial class ServicioEstudiante : IStudentService
{
    private readonly IStorageBroker storageBroker;
    ...

    public async ValueTask< Student> AddStudentAsync(Student student) =>
        await this.storageBroker. InsertStudentAsync(student);
}
```

Un servicio de orden superior haría/tendría el siguiente aspecto:

```
public partial class ServicioDeTratamientoDeEstudiantes : IStudentProcessingService
{
    private readonly IStudentService studentService;
    ...

    public async ValueTask< Student> UpsertStudentAsync(Student student)
    {
        ...

        Student maybeStudent = await this.studentService
            .RetrieveStudentByIdAsync(estudiante.Id);

        return maybeStudent switch
        {
            null => await this.studentService. AddStudentAsync(student),
            _ => await this.studentService. ModifyStudentAsync(student)
        }
    }
}
```

Unos servicios más avanzados de tipo orquestado combinarían a partir de múltiples servicios de procesamiento o fundacionales de la siguiente manera:

```
public partial class ServicioDeEstudianteDeOrquestación : IStudentOrchestrationService
{
    private readonly IStudentProcessingService studentProcessingService;
    private readonly IStudentLibraryCardProcessingService studentLibraryCardProcessingService;
    ...

    public async ValueTask< Student> RegisterStudentAsync(Student student)
    {
        ...

        Student upsertedStudent = await this.studentProcessingService
            .UpsertStudentAsync(estudiante);

        ...
        await this.studentLibraryProcessingService. AddStudentLibraryCardAsync(studentLibraryCard);
    }
}
```

En general, los modelos operacionales sólo se preocupan por la naturaleza de la simulación o el procesamiento de ciertos modelos portadores de datos, no se preocupan por retener datos, ni por retener un estado. En general, los modelos operacionales son apátridas en el sentido de que no retienen ninguno de los detalles que pasaron por ellos, salvo la delegación del registro con fines de observabilidad y seguimiento.

0.1.2.0.1.2 Modelos de exposición (expositores)

Los modelos de exposición manejan la HMI en todos los escenarios en los que un humano y un sistema tienen que interactuar. Pueden ser simples APIs RESTful, pueden ser SDKs o simplemente UI como en aplicaciones web, móviles o de escritorio, incluyendo sistemas/terminales basados en líneas de comandos.

Los modelos operativos de exposición son como los modelos de integración, permiten que el mundo exterior interactúe con su sistema. Se sientan en el otro extremo de cualquier sistema y son responsables de enrutar cada solicitud, comunicación o llamada a los modelos operativos adecuados. Los modelos de exposición nunca se comunican directamente con los modelos de integración, y no tienen ninguna configuración dentro de ellos aparte de sus dependencias que se inyectan a través de sus constructores.

Los modelos de exposición pueden tener su propio lenguaje en términos de operaciones, por ejemplo, un modelo de integración podría utilizar un lenguaje como P
mientras que un modelo de exposición para un punto final de la API `ostStudent` utilizaría un lenguaje como para expresar la misma operación en un contexto de exposición.

He aquí un ejemplo para los modelos de exposición:

```
public class EstudiantesControlador
{
    private readonly IStudentOrchestrationService studentOrchestrationService;

    [HttpGet]
    public async ValueTask< ActionResult< Student>> PostStudentAsync(Student student)
    {
        Student registeredStudent = await this.studentOrchestrationService
            .RegisterStudentAsync(estudiante);

        devolver Ok(registeredStudent);
    }
}
```

El modelo anterior expone un punto final de la API para la comunicación RESTful para ofrecer una capacidad para registrar a los estudiantes en algún sistema de escolarización. Vamos a discutir más a fondo los tipos de modelos de exposición basados en el contexto y el sistema en el que se implementan.

0.1.2.0.2 Modelos de configuración

El último tipo de modelos en cualquier sistema son los modelos de configuración. Pueden representar el punto de entrada en cualquier sistema, o registrar dependencias para cualquier sistema o simplemente jugar el papel de middleware para enrutar URLs a su respectiva función dentro de un modelo de exposición.

Los modelos de configuración normalmente se sitúan al principio del lanzamiento de un sistema, o manejan las comunicaciones entrantes y en curso o simplemente manejan las operaciones subyacentes del sistema como el almacenamiento en caché de la memoria, la gestión de hilos, etc.

En una aplicación sencilla de la API puede ver modelos con este aspecto:

```
public class Inicio
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddTransient< IStorageBroker, StorageBroker>();
        services.AddOAuth();
    }
}
```

Como puede ver el fragmento de código anterior, el modelo de `Inicio` offers capacidades para manejar el registro basado en la inyección de dependencia de los contratos a sus implementaciones concretas. Pueden manejar la adición de seguridad o la configuración de una tubería de middleware. Los modelos de configuración son específicos de la tecnología. Pueden diferir de un marco de trabajo Play en Scala a un Spring o Flex en Python o Java. Esbozaremos reglas de alto nivel según The Standard para los modelos de configuración, pero no profundizaremos en los detalles de la implementación de ninguno de ellos.

0.1.3 Simulación

El aspecto de simulación de la ingeniería de software, es nuestra capacidad para asemejar las interacciones hacia y desde los modelos. Por ejemplo, en el ejemplo de la tienda de comestibles, una simulación sería el acto de *vender* el artículo. Vender el artículo requiere múltiples modificaciones en el artículo en términos de deducir el recuento de los artículos disponibles y reordenar los artículos que quedan en base al artículo disponible más valioso.

El proceso de simulación puede describirse como la ilustración de las relaciones entre modelos. se programan como funciones, métodos o rutinas y todos ellos significan lo mismo. Si tenemos un servicio de software que se encarga de la venta de artículos, un proceso de simulación sería así:

```
public class VentaServicio
{
    public void Sell(Item item) => Items.Remove(item);
}
```

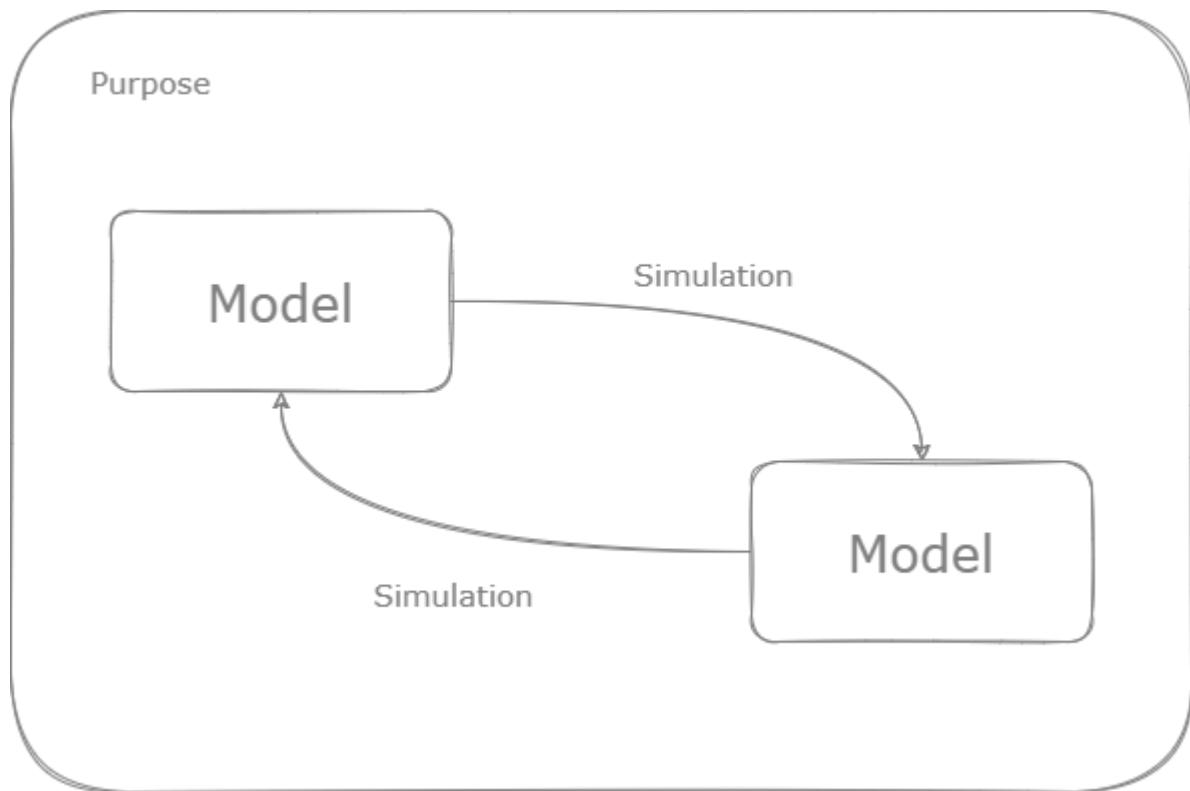
En el ejemplo anterior, tenemos un modelo llamado `Servicio` que offers un funcionalidad para simular el proceso de venta en el mundo real en un modelo de un artículo. Y así es como se describe todo en la programación orientada a objetos. Todo es un objeto (de un modelo) y estos objetos interactúan entre sí (simulación).

Los objetos que interactúan en general pueden observarse en tres tipos diferentes. Un modelo que realiza una acción sobre otro modelo. Por ejemplo, el `Vendedor` ejecutando una acción de `Artículo` en un modelo. Es un modelo que interactúa con otro modelo. En el mismo ejemplo, una simulación podría ser algo que le ocurra al modelo desde otro modelo como el `Artículo` ejecutando una acción de `Vendedor`. En el ejemplo anterior. Y el último tipo de simulación es un modelo interactuando consigo mismo. Modelos que se autodestruyen una vez que su propósito se cumple y ya no son necesarios, por ejemplo. Se autodestruirán.

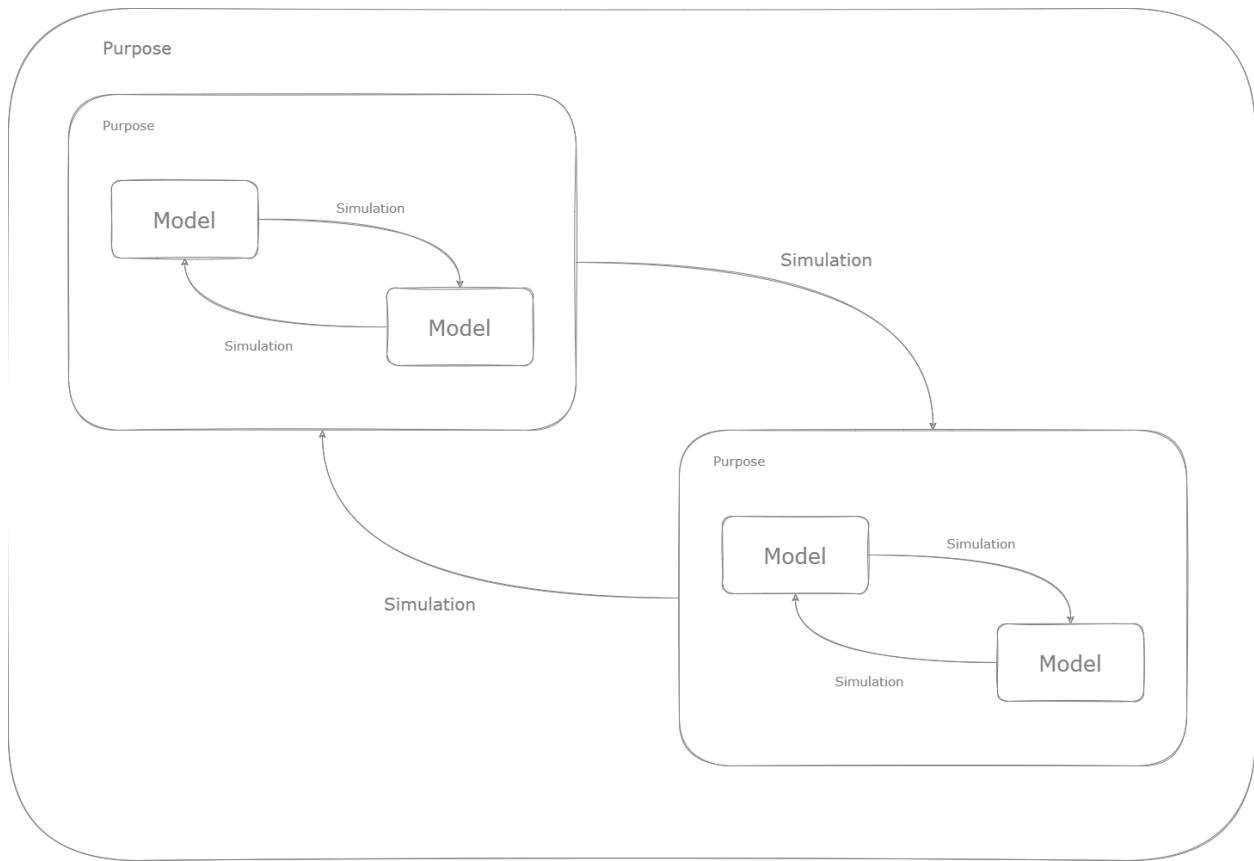
El proceso de simulación es el tercer y último aspecto de la ingeniería del software. En el que nos sumergiremos a fondo cuando hablamos de los correderos, los servicios y los expositores para ilustrar cómo ocurre el proceso de modelado y simulación en el software industrial.

0.1.4 Resumen

Si consideramos que el propósito es el dominio o el marco en el que interactúan los modelos. Entonces la siguiente ilustración debería simplificar y transmitir la imagen un poco más clara:



Es importante entender que un programa informático puede servir para múltiples propósitos. Los programas informáticos pueden interactuar con otros programas que comparten propósitos comunes. El propósito del software se convierte en el modelo y las integraciones se convierten en las simulaciones en ese aspecto. He aquí un ejemplo de 10.000 pies:



La complejidad de cualquier sistema grande puede descomponerse muy fácilmente en problemas más pequeños si se aplica el aspecto de propósito único o responsabilidad única para todos y cada uno de los subsistemas. Eso es lo que las arquitecturas de software modernas llamarían granularidad y modularización. De lo que hablaremos briefly a lo largo del aspecto de arquitectura de La Norma.

[*] Propuestas, modelización y simulación (parte 1)

0.2 Principios

En este capítulo, exploraremos los principios de la Norma. Estos principios se aplican a todos los componentes de un sistema que cumple con la Norma. Tanto si estos componentes son corredores, servicios o expositores.

0.2.0 People-First

La idea principal de este principio es construir y diseñar sistemas conformes a la norma pensando en las personas. No sólo las personas que van a utilizar el sistema, sino también las que van a mantenerlo y hacerlo evolucionar.

Para que un sistema siga el principio de "primero la gente", debe primar la simplicidad sobre la complejidad. La simplicidad conduce a la reescritura. También lleva a diseñar sistemas monolíticos que se construyen con una mentalidad modular para permitir una verdadera fractura en el patrón general del sistema.

La Norma también aplica los principios de medición de los conceptos avanzados de ingeniería frente a la comprensión de los ingenieros convencionales. Los nuevos ingenieros del sector son los líderes del mañana. Si no se creen cualquier sistema, acabarán por renunciar a él y reescribirlo una y otra vez.

0.2.0.0 Simplicidad

El código escrito según la Norma tiene que ser sencillo. Existen medidas para garantizar esta simplicidad, estas medidas son las siguientes:

0.2.0.0.0 Herencia excesiva

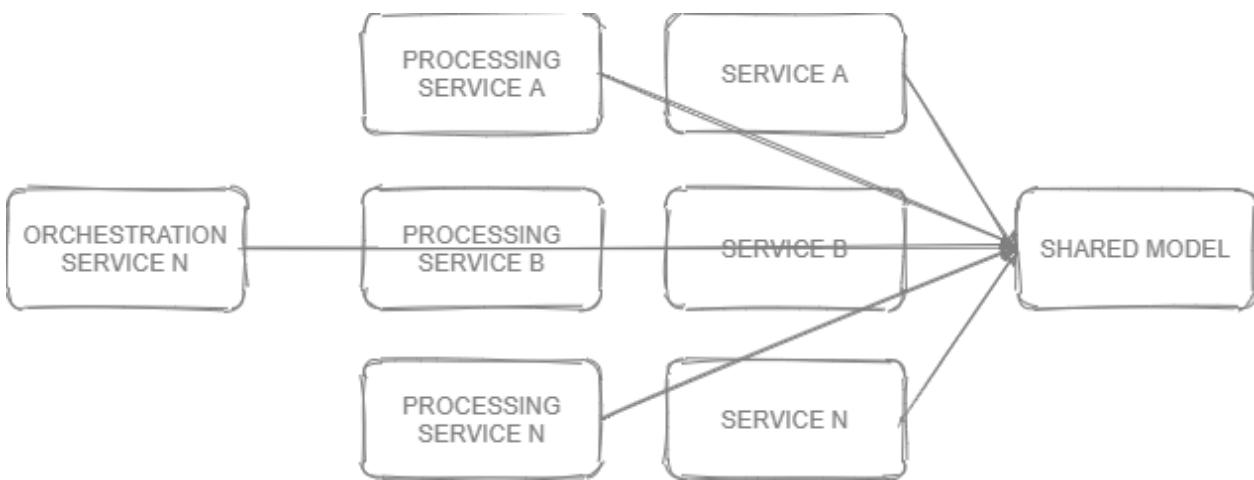
Cualquier software escrito de acuerdo con La Norma no deberá tener más de un nivel de herencia. Más de un nivel de herencia se considerará excesivo y estará prohibido. Excepto en los casos de versionado para el escalado vertical de flows. La herencia excesiva ha demostrado a lo largo de los años ser una fuente de confusión y dificultad en términos de legibilidad y mantenibilidad.

0.2.0.0.1 Enredo

0.2.0.0.1.0 Enredo horizontal

Construir componentes "comunes" en todos los sistemas con la promesa de simplificar el proceso de desarrollo es otra práctica prohibida en los sistemas que cumplen la norma. Esta práctica se manifiesta en componentes con nombres Comunes como Utils, o Helpers. Estas terminologías y lo que implican en términos de falsas simplificaciones prometidas no están permitidas. Cualquier sistema construido según La Norma debe estar compuesto por Corredores, Servicios o Expositores, ni más ni menos.

Otro ejemplo de enredos horizontales son los modelos compartidos entre múltiples flujos independientes: compartir excepciones, reglas de validación o cualquier otra forma de enredo entre múltiples flujos.



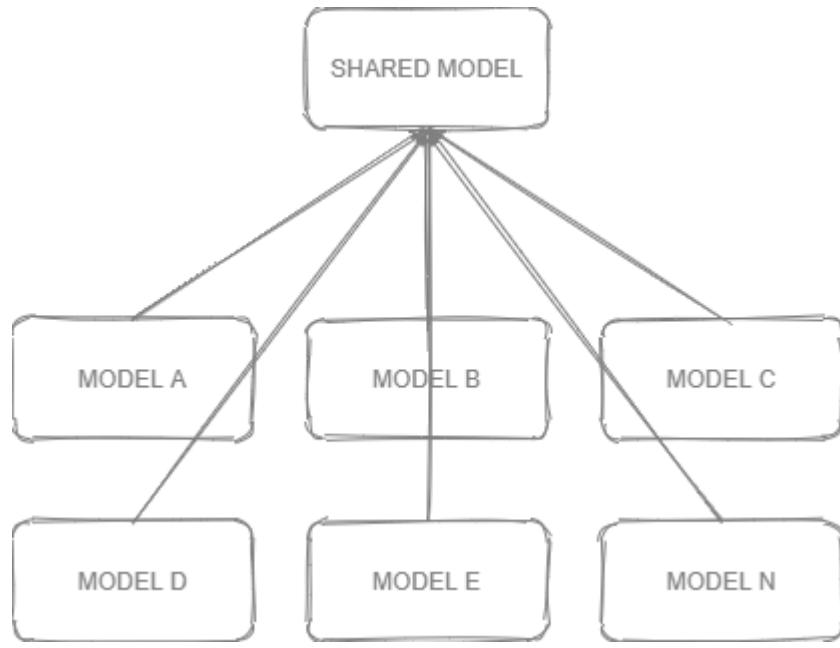
0.2.0.0.1.1 Enredo vertical

Este principio también se aplica a los escenarios en los que se utilizan componentes base. A menos que estos componentes base sean nativos o externos, no se permitirán en un sistema que cumpla con la Norma. Los componentes base locales crean un nivel de enredo vertical que perjudica la mantenibilidad y legibilidad del código. Comunes

Los enredos verticales son tan perjudiciales como la creación de puntos únicos de fallo en cualquier sistema.

Los enredos (verticales u horizontales) también impiden que los ingenieros de cualquier sistema (especialmente los recién llegados) comprendan plenamente la profundidad del sistema y se apropien de su funcionalidad. También impiden que los ingenieros tengan la oportunidad de construir flujos de extremo a extremo cuando la mitad de la funcionalidad está presumiblemente compuesta en aras de la rapidez y la simplicidad del

desarrollo.



0.2.0.0.2 Componentes autónomos

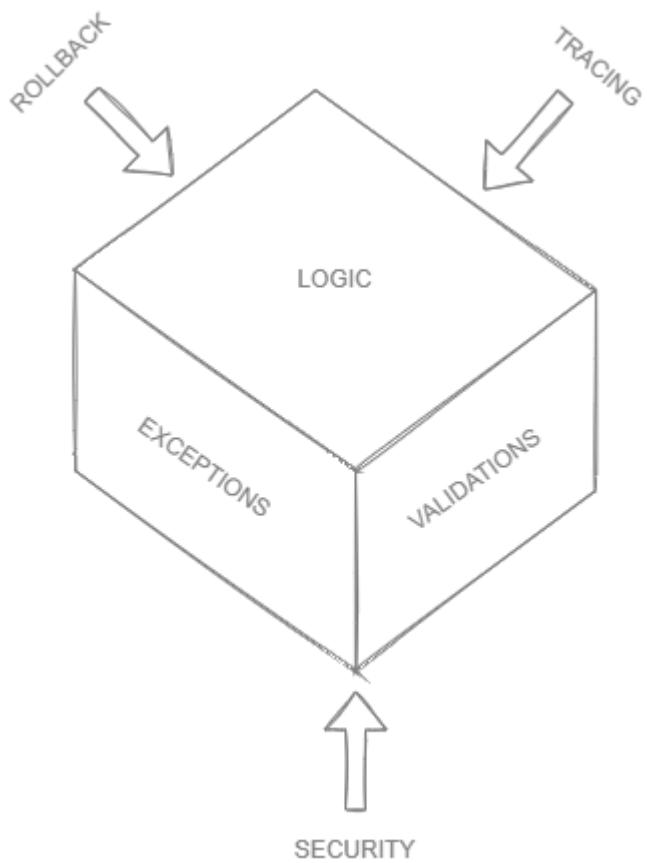
Este principio favorece la duplicación frente a la presunta simplificación mediante el entrelazamiento de códigos. Cada componente de cada sistema debe ser autosuficiente. Cada componente implementa sus propias validaciones, herramientas y utilidades en una de sus dimensiones sin depender de ningún otro componente externo, excepto a través de la inyección de dependencias.

Los componentes autónomos abrirán la oportunidad para que cada ingeniero de cada equipo sea dueño de todas las dependencias y herramientas que su componente pueda necesitar para cumplir su propósito. Esto puede causar un poco de duplicación en parte del código en aras de abrir una oportunidad igualitaria para que cada desarrollador aprenda completamente cómo construir y evolucionar un componente.

0.2.0.0.2.0 No hay magia

Los componentes autónomos ponen todas sus rutinas a la vista del ingeniero. No hay rutinas ocultas, bibliotecas compartidas o extensiones mágicas que requieran perseguir referencias una vez que comience a producirse una inevitable división del gran monolito.

Trataremos los objetos de la misma manera que en la naturaleza: componentes multidimensionales que se autocontienen como los átomos en la naturaleza. Estos componentes realizan sus propias validaciones, manejo de excepciones, rastreo, seguridad, localización y todo lo demás.



Los componentes construidos de acuerdo con El Estándar se adhieren estrictamente a la idea de *What You See Is What You Get* (WYSIWYG) - todo lo relativo a los componentes estará en el propio componente.

0.2.1 Reescribibilidad

Todo sistema debe desarrollarse teniendo en cuenta el principio de reescritura. El principio dicta que nuestros supuestos en los sistemas que desarrollamos tienen el alto potencial de ser reexaminados y probablemente reconsiderados. Todos los sistemas deberían ser fácilmente reescribibles como medida de adhesión a unos requisitos empresariales siempre crecientes y cambiantes.

El código reescribible es un código fácil de entender, modificar y reescribir completamente. El código reescribible es extremadamente modular y autónomo, lo que anima a los ingenieros a evolucionarlo con la menor cantidad de effort y riesgo posible.

El código reescribible no juega con el lector. No hay dependencias ocultas, rutinas inyectadas en tiempo de ejecución ni requisitos previos desconocidos. Debería ser plug-n-play - fork, clonar, construir y ejecutar todas sus pruebas con éxito sin problemas.

0.2.2 Mono-Micro

Construir sistemas monolíticos con mentalidad modular. Cada flujo debe construirse de forma totalmente independiente de otros flujos. Por ejemplo, podemos construir un sistema monolítico con mentalidad de microservicio. Lo que significa que cualquier flujo puede ser extraído del sistema y convertido en su propio microservicio o lambda con la menor cantidad de esfuerzo posible.

Este principio va de la mano con el concepto de componentes autónomos a un nivel superior en el que los flows también son autónomos de su flow vecino y de su sistema de alojamiento.

0.2.3 Nivel 0

El código debe ser comprensible para una persona que se inicie en el oficio de la ingeniería. Nuestra base de código sigue viviendo sólo en función de su facilidad de comprensión por parte de la mayoría de los ingenieros del sector. La mayoría de los ingenieros de nuestra industria siempre serán aquellos que son nuevos en el oficio.

Los ingenieros de nivel 0 son nuestra medida del éxito. Su capacidad para entender nuestro código es nuestra garantía de que este mismo código seguirá viviendo y evolucionando con la siguiente generación de ingenieros.

Este principio también obliga a todos los ingenieros de la industria a cruzar su código y a emparejarse con los juniors en el campo para ver si cumplen este principio.

0.2.4 Código abierto

El código abierto es un principio que dicta que todo lo que se escriba de acuerdo con la norma debe estar a disposición del público. El desarrollo de herramientas internas que no sean accesibles perjudicará inevitablemente la experiencia de ingeniería de quienes tratan de hacer evolucionar estas mismas herramientas. Esto se aplica a las prácticas de artefactos internos, a las bibliotecas de prueba y a cualquier otra forma de desarrollo de módulos que no permita a todos los ingenieros de cualquier lugar aprender y evolucionar un sistema determinado.

El principio también reconoce que, en circunstancias extremas, como cuando la seguridad o la protección están amenazadas o en virtud de alguna obligación contractual, no se puede poner a disposición del público el código, las herramientas, las plataformas y los patrones.

Sin embargo, The Standard no permite que la fuente sea de

propiedad exclusiva para beneficio personal o de la organización.

0.2.5 Modo avión (nube-extranjera)

La norma refuerza la idea del modo avión. Donde los ingenieros pueden de toda su infraestructura en su máquina local sin necesitar ni tener ningún tipo de conexión a la red. Este principio va muy en contra de principios como el de las aplicaciones Cloud-Native, en las que un determinado sistema no puede mantenerse en pie y funcionar sin una infraestructura en la nube.

El estándar también anima a sus adaptadores a desarrollar las herramientas adecuadas para salvar la brecha entre los componentes de la infraestructura de la nube y los componentes locales, como las colas, los concentradores de eventos y cualquier otra herramienta para que sea fácilmente comprobable y modificable.

0.2.6 Sin tostadoras

La Norma debe enseñarse de hombre a hombre, no de máquina a hombre. No debe haber estilistas ni analizadores que obliguen a la gente a seguir la Norma. Debe ser impulsada por la pasión en el corazón y la convicción en la mente. La Norma debe ser enseñada de persona a persona. Está pensada para fomentar una cultura de ingeniería de debates abiertos, convicción y comprensión.

0.2.7 Pasar al frente

La Norma se enseñará sin coste alguno. Así como llegó a usted sin costo alguno. Debe ser transmitida al siguiente ingeniero también sin costo alguno. Independientemente de su estatus financiero, social o educativo. La Norma es puro conocimiento dado por el selfless al selfless. No debe haber ningún profitecimiento off de ella, ni debe ser una razón para que alguien menosprecie a otros o los haga sentir menos. Enseñar El Estándar por profit viola El Estándar, y niega a su beneficiario cualquier otra guía del Autor.

1 Corredores de bolsa

1.0 Introducción

Los intermediarios desempeñan el papel de enlace entre la lógica empresarial y el mundo exterior. Son envolturas alrededor de cualquier biblioteca externa, recursos, servicios o APIs para satisfacer una interfaz local para que el negocio interactúe con estos recursos sin tener que estar estrechamente acoplado con cualquier recurso particular o implementación de biblioteca externa.

Los corredores, en general, están pensados para ser desecharables y reemplazables: se construyen teniendo en cuenta que la tecnología evoluciona y cambia constantemente y, por lo tanto, en algún momento del ciclo de vida de una aplicación determinada serán sustituidos por una tecnología moderna que haga el trabajo más rápido.

Pero los Brokers también garantizan que su negocio sea conectable al abstraer cualquier dependencia de recursos externos específicos de lo que su software está tratando de lograr realmente.

Por ejemplo, digamos que tienes una API que fue construida para consumir y servir datos de un servidor SQL. En algún momento, usted decidió que una opción mejor y más económica para su API es confiar en una tecnología NoSql en su lugar.

Disponer de un broker para abstraer la dependencia de SQL hará mucho más fácil la integración con NoSql con el menor tiempo y coste humanamente posible.

1.1 En el mapa

En cualquier aplicación, ya sea móvil, de escritorio, web o simplemente una API, los corredores suelen residir en la "cola" de cualquier aplicación, ya que son el último punto de contacto entre nuestro código personalizado y el mundo exterior.

Ya sea que el mundo exterior en este caso sea simplemente un almacenamiento local en la memoria, o un sistema totalmente independiente que reside detrás de una API, todos ellos tienen que residir detrás de los Brokers en cualquier aplicación.

En la siguiente arquitectura de bajo nivel para una determinada API - Los intermediarios residen entre nuestra lógica de negocio y el recurso externo:



1.2 Características

Hay algunas reglas simples que rigen la aplicación de cualquier corredor - estas reglas son:

1.2.0 Implementa una interfaz local

Los corredores tienen que satisfacer un contrato local. Tienen que implementar una interfaz local para permitir el desacoplamiento entre su implementación y los servicios que los consumen.

Por ejemplo, dado que tenemos un contrato local `IStorageBroker` que requiere una implementación para cualquier operación CRUD dada para un modelo local

Estudia - la operación del contrato sería la siguiente:

nte

```

interfaz parcial pública IStorageBroker
{
    IQueryable< Student> SelectAllStudents();
}

```

Una implementación para un agente de almacenamiento sería la siguiente:

```

public partial class StorageBroker
{
    public DbSet< Student> Students { get; set; }

    public IQueryable< Student> SelectAllStudents()
    {
        using var broker = new StorageBroker(this.configuration);

        return broker.Students;
    }
}

```

La implementación de un contrato local puede ser reemplazada en cualquier momento desde la utilización de Entity Framework como se muestra en el ejemplo anterior, hasta el uso de una tecnología completamente diferente como Dapper, o una infraestructura completamente diferente como una base de datos Oracle o PostgreSQL.

1.2.1 Sin control de flujo

Los corredores no deberían tener ninguna forma de control de flujo, como sentencias if, bucles while o casos de conmutación, simplemente porque el código de control de flujo se considera lógica de negocio, y encaja mejor en la capa de servicios, donde debería residir la lógica de negocio, no en los corredores.

Por ejemplo, un método de broker que recupere una lista de estudiantes de una base de datos tendría el siguiente aspecto:

```
public IQueryable< Student> SelectAllStudents()
{
    using var broker = new StorageBroker(this.configuration);
    return broker.Students;
}
```

Una simple función que llama al EntityFramework `DbSet<T>` y nativo devuelve un modelo local como `Student`.

1.2.2 Sin manejo de excepciones

El manejo de excepciones es una forma de control de flujo. Se supone que los corredores no deben manejar ninguna excepción, sino que deben dejar que la excepción se propague a los servicios vecinos al corredor, donde estas excepciones van a ser debidamente asignadas y localizadas.

1.2.3 Poseer sus Configuraciones

Los brokers también deben manejar sus propias configuraciones - pueden tener una inyección de dependencia de un objeto de configuración, para recuperar y configurar las configuraciones para cualquier recurso externo con el que se estén integrando.

Por ejemplo, las cadenas de conexión en las comunicaciones con las bases de datos deben ser recuperadas y pasadas al cliente de la base de datos para establecer una conexión exitosa, como se indica a continuación:

```
public partial class StorageBroker : EFExceptionsContext, IStorageBroker
{
    private readonly IConfiguration configuration;

    public StorageBroker(IConfiguration configuration)
    {
        this.configuration = configuration; this.Database.
        Migrate();
    }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        string connectionString = this.configuration.GetConnectionString("DefaultConnection"); optionsBuilder.
        UseSqlServer(connectionString);
    }
}
```

1.2.4 Los nativos de los primitivos

Los corredores pueden construir un objeto modelo externo basado en los tipos primitivos pasados desde los servicios vecinos al corredor.

Por ejemplo, en el corredor de notificación de correo electrónico, los `enviar()` parámetros de entrada de una función, por ejemplo, requieren los parámetros de entrada básicos como el tema, el contenido o la dirección, por ejemplo, he aquí un ejemplo:

```
public async ValueTask SendMailAsync(List< string> destinatarios, string asunto, string contenido)
{
    Message = BuildMessage(recipients, ccRecipients, subject, content); await
    SendEmailMessageAsync(message);
}
```

Los parámetros de entrada primitivos asegurarán que no haya fuertes dependencias entre los servicios vecinos al broker y los modelos externos. Incluso en situaciones en las que el broker es simplemente un punto de integración entre tu aplicación y una API RESTful externa, es muy recomendable que construyas tus propios modelos nativos para reflectar el mismo objeto JSON enviado o devuelto desde la API en lugar de depender de bibliotecas nuget, dlls o proyectos compartidos para lograr el mismo objetivo.

1.2.5 Convenciones de nomenclatura

Los contratos para los corredores serán lo más genéricos posible para indicar la funcionalidad general de un corredor, por ejemplo `IStorageBroker` en lugar de `SqlStorageBroker` para indicar una tecnología concreta o infraestructura

.

Pero en el caso de las implementaciones concretas de los brokers, todo depende de cuántos brokers tengas proporcionando una funcionalidad similar, en el caso de tener un único broker de almacenamiento, podría ser más conveniente mantener el mismo nombre que el contrato - en nuestra `StorageBroker` na implementación concreta de sería `StorageBroker`.

Sin embargo, si su aplicación admite múltiples colas, almacenes o proveedores de servicios de correo electrónico, es posible que tenga que empezar a especificar el objetivo general del componente, por ejemplo, un `IQueueBroker` tendría múltiples implementaciones como `PedidosQueueBroker`, `NotificaciónQueueBroker` y `o`.

Pero si las implementaciones concretas apuntan al mismo modelo y valor de negocio, entonces una desviación a la tecnología podría ser más befitting en este caso, por ejemplo en el caso de un `IStorageBroker` dos implementaciones concretas diferentes serían `SqlStorageBroker` y `FileStorageBroker`. Este caso es muy posible en situaciones en las que los costes del entorno se reducen en menor medida que la infraestructura de producción, por ejemplo.

1.2.6 Idioma

Los corredores hablan el idioma de las tecnologías que soportan. Por ejemplo, en un agente de almacenamiento, decimos `SelectById` para `Select`, la declaración coincide con el SQL y en un corredor de cola decimos `Enqueue` para que coincida con el lenguaje.

Si un agente admite un punto final de la API, deberá seguir la norma RESTful lenguaje de operaciones, como `POST`/`GET` o `PUT`, he aquí un ejemplo:

```
public async ValueTask< Student> PostStudentAsync(Student student)
=> await this.PostAsync(RelativeUrl, student);
```

1.2.7 Arriba y al lado

Los brokers no pueden llamar a otros brokers. eso es simplemente porque los brokers son el primer punto de abstracción, no requieren abstracciones adicionales ni dependencias adicionales más que un modelo de acceso a la configuración.

Los corredores no pueden tener también servicios como dependencias, ya que el flujo en un sistema determinado debe venir de los servicios a los corredores y no al revés.

Incluso en situaciones en las que un microservicio tiene que suscribirse a una cola, por ejemplo, los corredores pasarán un método de escucha para procesar los eventos entrantes, pero no llamarán a los servicios que proporcionan la lógica de procesamiento.

La regla general aquí sería, que los corredores sólo pueden ser llamados por los servicios, y sólo pueden llamar a las dependencias nativas externas.

1.3 Organización

Los corredores que soportan múltiples entidades, como los corredores de almacenamiento, deben aprovechar las clases parciales para desglosar las responsabilidades por entidades.

Por ejemplo, si tenemos un broker de almacenamiento que proporciona todas las operaciones RUD para manejar y consultar los datos, entonces la organización de los archivos debería ser la siguiente:

- IStorageBroker.cs
 - IStorageBroker.Students.cs
 - IStorageBroker.Teachers.cs
- StorageBroker.cs
 - StorageBroker.Students.cs
 - StorageBroker.Teachers.cs

El objetivo principal de esta organización particular que aprovecha las clases parciales es separar la preocupación por cada entidad hasta un nivel finer, lo que debería hacer que la mantenibilidad del software sea mucho mayor.

Pero la convención de nomenclatura de los corredores de archivos y carpetas se centra estrictamente en la pluralidad de las entidades que soportan y la singularidad para el recurso global que se soporta.

Por ejemplo, decimos `IStorageBroker.Students.cs`. y también decimos `IEmailBroker` o `IQueueBroker.Notifications.cs` - singular para el recurso y las entidades plurales.

El mismo concepto se aplica a las carpetas o espacios de nombres que contienen estos corredores.

Por ejemplo, decimos:

```
namespace Otriples.Web.Api.Brokers.Almacenes
{
    ...
}
```

Y nosotros decimos:

```
namespace Otriples.Web.Api.Brokers.Queues
{
    ...
}
```

1.4 Tipos de corredores

En la mayoría de las aplicaciones construidas hoy en día, hay algunos Brokers comunes que suelen ser necesarios para poner en marcha una aplicación empresarial - algunos de estos Brokers son como Almacenamiento, Tiempo, APIs, Logging y Colas.

Algunos de estos corredores interactúan con los recursos existentes en el sistema, como el tiempo, para permitir que los servicios vecinos al corredor traten el tiempo como una dependencia y controlen cómo se comportaría un servicio concreto en función del valor del tiempo en cualquier punto del

pasado, el presente o el futuro.

1.4.0 Corredores de bolsa

Los brokers de entidades son los que proporcionan puntos de integración con los recursos externos que el sistema necesita para cumplir con los requisitos del negocio.

Por ejemplo, los brokers de entidades son corredores que se integran con el almacenamiento, proporcionando capacidades para almacenar o recuperar registros de una base de datos.

Los brokers de entidades son también como los brokers de colas, proporcionando un punto de integración para empujar los mensajes a una cola para que otros servicios los consuman y procesen para cumplir con su lógica de negocio.

Los corredores de entidades sólo pueden ser llamados por los servicios vecinos al corredor, simplemente porque requieren un nivel de validación que debe realizarse en los datos que reciben o proporcionan antes de seguir adelante.

1.4.1 Apoyo a los corredores

Los brokers de soporte son brokers de propósito general, proporcionan una funcionalidad a los servicios de soporte pero no tienen ninguna característica que los haga diferentes de un sistema u otro.

Un buen ejemplo de corredores de apoyo es el `TimeBroker` - un corredor hecho específicamente para abstraer la fuerte dependencia de la capa de negocio de la fecha del sistema.

Los corredores de tiempo no se dirigen realmente a ninguna entidad específica, y son casi los mismos en muchos sistemas existentes.

Otro ejemplo de corredores de apoyo es el `LoggingBroker` - que proporcionan a los sistemas de registro y monitorización para que los ingenieros del sistema puedan visualizar el flujo global de datos en el sistema y ser notificados en caso de que se produzca algún problema.

A diferencia de los Entity Brokers - los brokers de soporte pueden ser llamados en toda la capa de negocio, pueden ser llamados en los servicios de fundación, procesamiento, orquestación, coordinación, gestión o agregación. eso es porque los brokers de registro son requeridos como un componente de soporte en el sistema para proporcionar todas las capacidades necesarias para que los servicios registren sus errores o calculen una fecha o cualquier otra funcionalidad de soporte.

Puede encontrar ejemplos reales de corredores en el proyecto OtripleS [aquí](#).

1.5 Aplicación

Esta es una implementación real de un broker de almacenamiento completo para todo el CRUD

operaciones para la entidad:

nte

Para IStorageBroker.cs:

```
namespace OtripleS.Web.Api.Brokers.Almacenamiento
{
    interfaz parcial pública IStorageBroker
    {
    }
}
```

Para StorageBroker.cs:

```
utilizando System;
usando EFxceptions. Identidad;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.
Configuration; using OtripleS.Web.Api.
Models. Usuarios;

namespace OtripleS.Web.Api.Brokers.Almacenamiento
{
    public partial class StorageBroker : EfxceptionsContext, IStorageBroker
    {
        private readonly IConfiguration configuration;

        public StorageBroker(IConfiguration configuration)
        {
            this.configuration = configuration; this.Database.
            Migrate();
        }

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            string connectionString = this.configuration. GetConnectionString("DefaultConnection"); optionsBuilder.
            UseSqlServer(connectionString);
        }
    }
}
```

Para IStorageBroker.Students.cs:

```
using System;
using System.Linq;
utilizando System.Threading.Tasks;
utilizando OtripleS.Web.Api.Modelos. Estudiantes;

namespace OtripleS.Web.Api.Brokers.Almacenamiento
{
    interfaz parcial pública IStorageBroker
    {
        public ValueTask< Student> InsertStudentAsync(Student student);
        public IQueryable< Student> SelectAllStudents();
        public ValueTask< Student> SelectStudentByIdAsync(Guid studentId);
        public ValueTask< Student> UpdateStudentAsync(Student student);
        public ValueTask< Student> DeleteStudentAsync(Student student);
    }
}
```

Para StorageBroker.Students.cs:

```
using System;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.ChangeTracking;
using OtripleS.Web.Api.Models;
using OtripleS.Web.Api.Brokers.Almacenamiento;

namespace OtripleS.Web.Api.Brokers
{
    public partial class StorageBroker
    {
        public DbSet<Student> Students { get; set; }

        public async ValueTask<Student> InsertStudentAsync(Student student)
        {
            using var broker = new StorageBroker(this.configuration);

            EntityEntry<Student> studentEntityEntry =
                await broker.Students.AddAsync(entity: student);

            await broker.SaveChangesAsync();

            return estudianteEntidadEntrada.Entidad;
        }

        public IQueryable<Student> SelectAllStudents()
        {
            using var broker = new StorageBroker(this.configuration);

            return broker.Students;
        }

        public async ValueTask<Student> SelectStudentByIdAsync(Guid studentId)
        {
            using var broker = new StorageBroker(this.configuration);

            broker.ChangeTracker.QueryTrackingBehavior =
                QueryTrackingBehavior.NoTracking;

            return await broker.Students.FindAsync(studentId);
        }

        public async ValueTask<Student> UpdateStudentAsync(Student student)
        {
            using var broker = new StorageBroker(this.configuration);

            EntityEntry<Student> studentEntityEntry =
                broker.Students.Update(entity: student);

            await broker.SaveChangesAsync();

            return estudianteEntidadEntrada.Entidad;
        }

        public async ValueTask<Student> DeleteStudentAsync(Student student)
        {
            using var broker = new StorageBroker(this.configuration);

            EntityEntry<Student> studentEntityEntry =
                broker.Students.Remove(entity: student);

            await broker.SaveChangesAsync();

            return estudianteEntidadEntrada.Entidad;
        }
    }
}
```

1.6 Resumen

Los brokers son la primera capa de abstracción entre tu lógica de negocio y el mundo exterior, pero no son la única capa de abstracción. simplemente porque todavía habrá pocos modelos nativos que se filtren a través de tus brokers a tus servicios vecinos al broker lo que es natural para evitar hacer cualquier mapeo fuera del ámbito de la lógica, en nuestro caso aquí los servicios de la fundación.

Por ejemplo, en un broker de almacenamiento, independientemente del ORM que estés utilizando, se producirán algunas excepciones nativas de tu ORM (~~Entity Framework por ejemplo~~, exception como o - en ese caso necesitamos otra capa de abstracción para desempeñar el papel de mapeador entre estas excepciones y nuestra lógica central para convertirlas en modelos de excepción locales.

Esta responsabilidad está en manos de los servicios de broker-neighboring, también los llamo servicios de fundación, estos servicios son el último punto de abstracción antes de su lógica central, en la que todo se convierte en nada más que modelos y contratos locales.

1.7 Preguntas frecuentes

A lo largo del tiempo, han surgido algunas preguntas comunes por parte de los ingenieros con los que he tenido la oportunidad de trabajar a lo largo de mi carrera - ya que algunas de estas preguntas se repitieron en varias ocasiones, pensé que podría ser útil agregarlas todas aquí para que todo el mundo conozca otras perspectivas en torno a los correderos.

1.7.0 ¿El patrón de los correderos es el mismo que el de los repositorios?

No exactamente, al menos desde un punto de vista operativo, los brokers parecen ser más genéricos que los repositorios.

Los repositorios suelen estar orientados a operaciones de almacenamiento, principalmente hacia las bases de datos. pero los brokers pueden ser un punto de integración con cualquier dependencia externa como servicios de correo electrónico, colas, otras APIs y demás.

Un patrón más similar para los correderos es el patrón de Unidad de Trabajo, se centra principalmente en la operación global sin tener que vincular la definición o el nombre con alguna operación en particular.

Todos estos patrones en general tratan de alcanzar el mismo objetivo de los principios SOLID, que es la separación de intereses, la inyección de dependencias y la responsabilidad única.

Pero como SOLID son principios y no directrices exactas, se espera ver todos los tipos diferentes de implementaciones y patrones para lograr ese principio.

1.7.1 ¿Por qué los correderos no pueden implementar un contrato para los métodos que devuelven una interfaz en lugar de un modelo concreto?

Esa sería una situación ideal, pero eso también requeriría que los correderos hicieran una conversión o mapeo entre los modelos nativos devueltos desde los SDKs o APIs de recursos externos y el modelo interno que se adhiere al contrato local.

Hacer eso en el nivel del corredor requerirá empujar la lógica de negocio en ese ámbito, que está fuera del propósito de ese componente por completo.

Los correderos no se someten a pruebas unitarias porque no tienen lógica de negocio, pueden ser parte de una prueba de aceptación o de integración, pero ciertamente no son parte de las pruebas a nivel de unidad - simplemente porque no contienen ninguna lógica de negocio en ellos. Definimos el código de lógica de negocio como cualquier código secuencial, selectivo o de iteración.

1.7.2 Si los brokers fueran realmente una capa de abstracción de la lógica de negocio, ¿cómo es que permitimos que las excepciones externas se filtren a través de ellos a la capa de servicios?

Los correderos son sólo *la primera* capa de abstracción, pero no la única - los servicios vecinos al corredor son responsables de convertir las excepciones nativas que ocurren desde un corredor en un modelo de excepción más local que puede ser manejado y procesado internamente dentro del ámbito de la lógica de negocio.

El código local puro comienza a producirse en las capas de procesamiento, orquestación, coordinación y agregación, donde todas las excepciones, todos los modelos devueltos y todas las operaciones se localizan en el sistema.

1.7.3 ¿Por qué utilizamos clases parciales para los correderos que manejan varias entidades?

Dado que los brokers deben poseer sus propias configuraciones, tenía más sentido parcializar cuando fuera posible para evitar reconfigurar cada broker de almacenamiento para cada entidad.

Esta es una característica de C# específicamente como lenguaje, pero debería ser posible implementarla a través de la herencia en otros lenguajes de programación.

1.7.4 ¿Son los correderos lo mismo que los proveedores (patrón de proveedores)?

No. Los proveedores desdibujan la línea entre los servicios (lógica

empresarial) y los intermediarios (capa de integración): los intermediarios se dirigen a componentes concretos del sistema que son desecharables. Los proveedores parecen incluir algo más que eso.

[*] Implementación de componentes abstractos (brokers)

[*] Implementación de componentes abstractos (Parte 2)

Generación de migraciones de modelos con EntityFramework

2 Servicios

2.0 Introducción

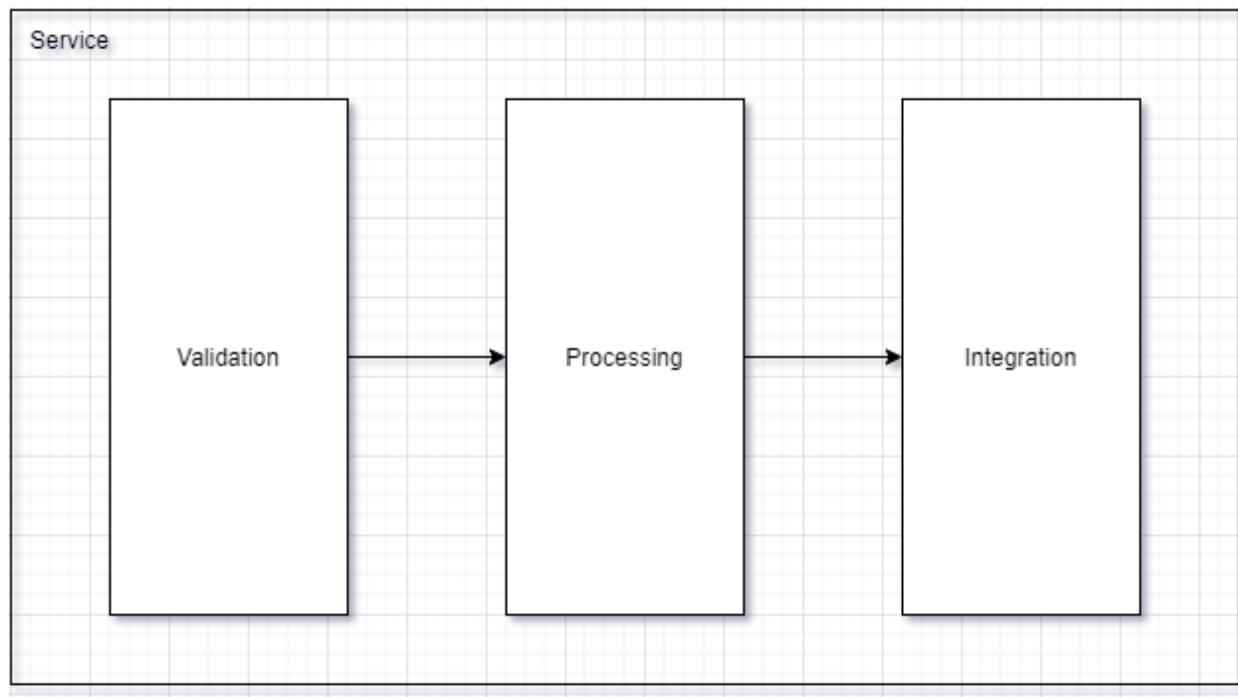
Los servicios en general son los contenedores de toda la lógica de negocio en cualquier software dado - son el componente central de cualquier sistema y el componente principal que hace que un sistema sea diferente de otro.

Nuestro principal objetivo con los servicios es que se mantengan completamente agnósticos de tecnologías específicas o dependencias externas.

Cualquier capa de negocio es más compatible con La Norma si se puede enchufar a cualquier otra dependencia y tecnologías de exposición con la menor cantidad de efforts de integración posibles.

2.0.0 Operaciones de servicios

Cuando decimos lógica empresarial, nos referimos principalmente a tres categorías principales de operaciones, que son la validación, el procesamiento y la integración.



Hablemos de estas categorías.

2.0.0.0 Validaciones

Las validaciones se centran en asegurar que los datos entrantes o salientes coinciden con un conjunto particular de reglas, que pueden ser validaciones estructurales, lógicas o externas, en ese orden exacto de prioridad. Entraremos en detalles sobre esto en las próximas secciones.

2.0.0.1 Procesamiento

El procesamiento se centra principalmente en el control del flujo, el mapeo y el cálculo para satisfacer una necesidad de negocio - las operaciones de procesamiento específicamente es lo que distingue un servicio de otro, y en general un software de otro.

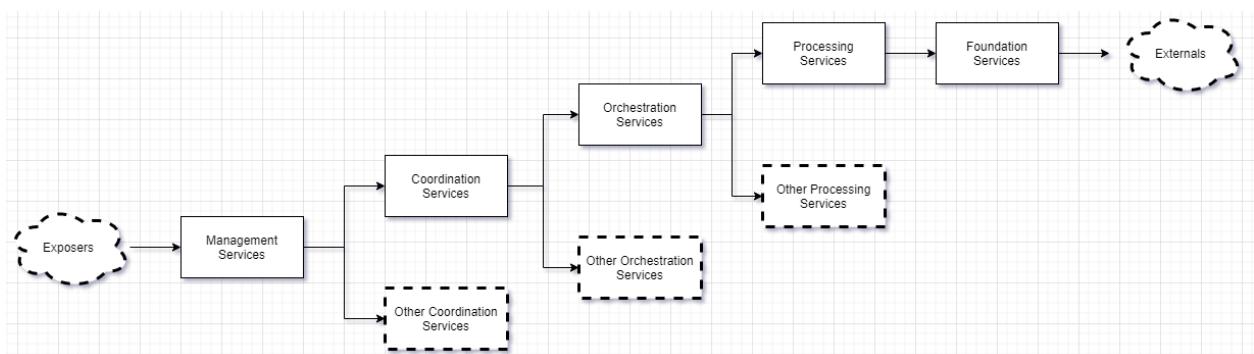
2.0.0.2 Integración

Por último, el proceso de integración se centra principalmente en la recuperación o el envío de datos desde o hacia cualquier dependencia del sistema integrado.

Cada uno de estos aspectos se discutirá en detalle en el próximo capítulo, pero lo principal que debe entenderse acerca de los servicios es que deben ser construidos con la intención de ser enchufables y configurables para que sean fácilmente integrados con cualquier tecnología desde un punto de vista de dependencia y también ser fácilmente enchufados en cualquier funcionalidad de exposición desde una perspectiva de API.

2.0.1 Tipos de servicios

Pero los servicios tienen varios tipos en función de su posición en una arquitectura determinada, y se dividen en tres categorías principales, que son: validadores, orquestadores y agregadores.



2.0.1.0 Validadores

Los servicios validadores son, principalmente, los servicios de intermediación y vecindad o los servicios de fundación.

La principal responsabilidad de estos servicios es añadir una capa de validación sobre las operaciones primitivas existentes, como las operaciones CRUD, para garantizar que los datos entrantes y salientes sean validados estructural, lógica y externamente antes de enviar los datos dentro o fuera del sistema.

2.0.1.1 Orquestadores

Los servicios orquestadores son el núcleo de la capa de lógica de negocio, pueden ser procesadores, orquestadores, coordinadores o servicios de gestión dependiendo del tipo de sus dependencias.

Los servicios de orquestación se centran principalmente en la combinación de múltiples operaciones primitivas, o de múltiples operaciones de lógica empresarial de alto orden para lograr un objetivo aún mayor.

Los servicios de orquestación son los que toman las decisiones dentro de cualquier arquitectura, son los dueños del control del flujo en cualquier sistema y son el componente principal que hace que una aplicación o software sea diferente de otro.

Los servicios orquestadores también están pensados para ser construidos y vivir más tiempo que cualquier otro tipo de servicios en el sistema.

2.0.1.2 Agregadores

La principal responsabilidad de los servicios agregadores es vincular el resultado de múltiples servicios de procesamiento, orquestación, coordinación o gestión para exponer una única API para que cualquier controlador de API o componente de interfaz de usuario interactúe con el resto del sistema.

Los agregadores son los guardianes de la capa de lógica de negocio, aseguran que los componentes de exposición de datos (como los controladores de la API) están interactuando con un solo punto de contacto para interactuar con el resto del sistema.

Los agregadores, en general, no se preocupan realmente por el orden en el que llaman a las operaciones que se les adjuntan, pero a veces se hace necesario ejecutar una operación concreta, como crear un registro de estudiante antes de asignarle un carné de biblioteca.

En los próximos capítulos hablaremos en detalle de todos y cada uno de estos servicios.

2.0.2 Normas generales

Hay varias reglas que rigen la arquitectura general y el diseño de los servicios en cualquier sistema.

Estas reglas garantizan la legibilidad, la mantenibilidad y la configurabilidad generales del sistema, en ese orden concreto.

2.0.2.0 Hacer o delegar

Cada servicio debe hacer el trabajo o delegarlo, pero no ambas cosas.

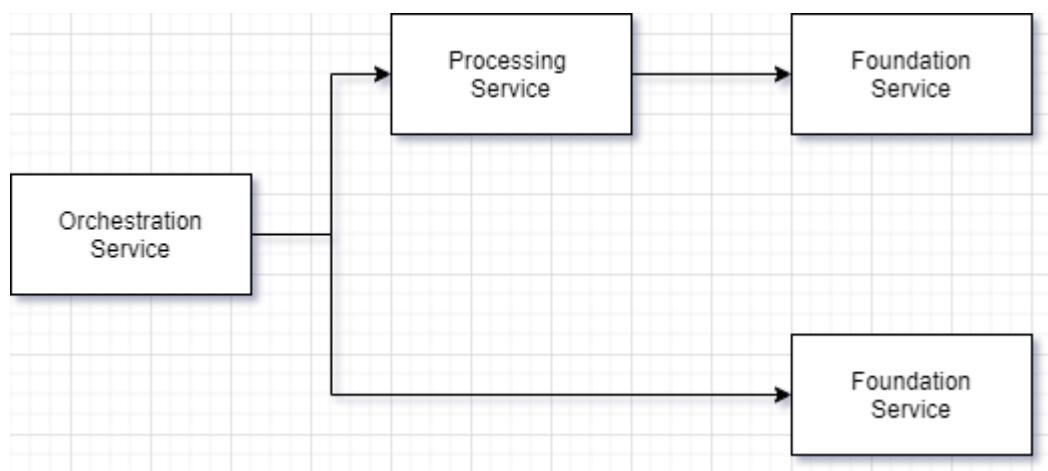
Por ejemplo, un servicio de procesamiento debería delegar el trabajo de persistencia de datos a un servicio de la fundación y no intentar hacer ese trabajo por sí mismo.

2.0.2.1 Dos-Tres (Patrón Florance)

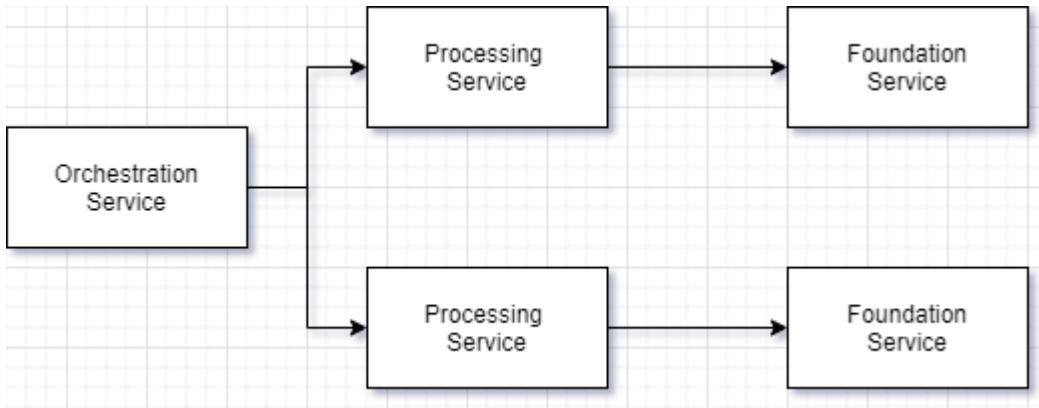
En el caso de los servicios del Orquestador, sus dependencias de servicios (no corredores) deben limitarse a 2 o 3, pero no a 1 ni a 4 o más.

La dependencia de un servicio niega la propia definición de orquestación. Esto se debe a que la orquestación, por definición, es la combinación entre múltiples operaciones diferentes de fuentes diferentes para lograr un orden superior de lógica empresarial.

Este patrón viola el patrón Florance



Este patrón sigue la simetría del Patrón



El patrón Florence también garantiza el equilibrio y la simetría de la arquitectura general.

Por ejemplo, no se puede orquestar entre una fundación y un servicio de procesamiento, provoca una forma de desequilibrio en su arquitectura, y una perturbación incómoda al tratar de combinar una declaración unificada con el lenguaje que habla cada servicio en función de su nivel y tipo.

El único tipo de servicios que se permite violar esta regla son los agregadores, donde la combinación y el orden de los servicios o sus llamadas no tienen ningún impacto real.

En las próximas secciones de The Standard hablaremos con más detalle del patrón Florence.

2.0.2.2 Punto único de exposición

Los controladores de la API, los componentes de la interfaz de usuario o cualquier otra forma de exposición de datos del sistema deben tener un único punto de contacto con la capa de lógica empresarial.

Por ejemplo, un punto final de la API que offer puntos finales para persistir y recuperar los datos de los estudiantes, no debe tener múltiples integraciones con múltiples servicios, sino un servicio que offer todas estas características.

A veces, un único servicio de orquestación, coordinación o gestión no offer todo lo relacionado con una entidad concreta, en cuyo caso es necesario un servicio agregador que combine todas estas características en un servicio listo para ser integrado por una tecnología de exposición.

2.0.2.3 Modelo de E/S igual o primitivo

Para todos los servicios, tienen que mantener un único contrato en cuanto a sus tipos de retorno y entrada, excepto si fueran primitivos.

Por ejemplo, un servicio que proporciona cualquier tipo de operaciones para un tipo de entidad

`Estudia` - no debe devolver desde ninguno de sus métodos ningún otro tipo de entidad.

Puede devolver una agregación de la misma entidad ya sea personalizada o nativo como `Lista<Estudia` o `Alumnos agregados` modelos, o un tipo primitivo ~~cómo~~ obtener el recuento de estudiantes, o un booleano que indique si un estudiante existe o no, pero no cualquier otro contrato no primitivo o no agregable.

Para los parámetros de entrada se aplica una regla similar: cualquier servicio puede recibir un parámetro de entrada del mismo contrato o de un contrato de agregación virtual o de un tipo primitivo, pero no cualquier otro contrato, que simplemente viola la regla.

Esta regla obliga a que cualquier servicio mantenga su responsabilidad en una sola entidad y en todas sus operaciones relacionadas.

Una vez que un servicio devuelve un contrato diferente, simplemente viola su propio nombre

convención como `StudentOrchestrationService` que regres`Lista<Teachr>` - y empieza a caer en la trampa de ser llamado por otros servicios desde una tubería de datos completamente diferente.

En el caso de los parámetros de entrada primitivos, si pertenecen a un modelo de entidad diferente, que no es necesariamente una referencia en la entidad principal, se plantea la cuestión de orquestar entre dos servicios de procesamiento o fundación para mantener un modelo unificado sin romper la regla de contratación pura.

Si se requiere la combinación entre múltiples contratos diferentes en un servicio de orquestación, entonces un nuevo modelo virtual unificado tiene que ser el nuevo contrato único para el servicio de orquestación con mapeos implementados por debajo en el nivel concreto de ese servicio para mantener la compatibilidad y la seguridad de la integración.

2.0.2.4 Cada servicio para sí mismo

Cada servicio es responsable de validar sus entradas y salidas. No debe depender de los servicios anteriores o posteriores para validar sus datos.

Se trata de un mecanismo de programación defensivo para garantizar que, en caso de intercambio de implementaciones detrás de los contratos, la responsabilidad de cualquier servicio dado no se vería afectada si los servicios descendentes o ascendentes decidieran pasar sus validaciones por cualquier motivo.

Dentro de cualquier sistema basado en una arquitectura monolítica, de microservicios o sin servidores, cada servicio se construye con la intención de que se separe del sistema en algún momento, y se convierta en el último punto de contacto antes de integrarse con algún intermediario de recursos externo.

Por ejemplo, en la siguiente arquitectura, los servicios son partes de mapeo de un modelo de entrada modelo en Tarjeta de la biblioteca

Estudiante

```
public class  
Estudiante  
{  
    public Guid Id {get; set;}  
    public string Name {get;  
        set;}  
}
```

Tarjeta de la biblioteca

```
public class Tarjeta de biblioteca  
{  
    public Guid Id {get; set;}  
    public Guid StudentId {get;  
        set;}  
}
```

Ahora, supongamos que nuestro servicio or `StudentOrchestrationService` es asegurar que cada nuevo estudiante que se inscriba tendrá que tener una tarjeta de biblioteca, por lo que nuestra lógica puede ser la siguiente:

```
public async ValueTask< Student> RegisterStudentAsync(Student student)  
{  
    Alumno registrado =  
        await this.studentProcessingService. RegisterStudentAsync(student);  
  
    await AssignStudentLibraryCardAsync(student);  
  
    return registeredStudent;  
}  
  
private async ValueTask< LibraryCard> AssignStudentLibraryCardAsync(Student student)  
{  
    LibraryCard studentLibraryCard = MapToLibraryCard(student);  
  
    return await this.libraryCardProcessingService. AddLibraryCardAsync(studentLibraryCard);  
}  
  
private LibraryCard MapToLibraryCard(Student student)  
{  
    devolver la nueva LibraryCard  
    {  
        Id = Guid. NewGuid(),  
        StudentId = student.Id  
    };  
}
```

Como puede ver arriba, se requiere una identificación de estudiante válida para asegurar una

a un Tarjeta de - y como el mapeo es del orquestador
responsabilidad, debemos asegurarnos de que el alumno de entrada y su id
están en buena forma antes de proceder al proceso de orquestación.

2.1 Servicios de la Fundación (corredor- vecino)

2.1.0 Introducción

Los servicios Foundation son el primer punto de contacto entre la lógica de su negocio y los corredores.

En general, los servicios de intermediación son un híbrido de lógica de negocio y una capa de abstracción para las operaciones de procesamiento en las que se produce la lógica de negocio de orden superior, de la que hablaremos más adelante cuando empecemos a explorar los servicios de procesamiento en la siguiente sección.

La principal responsabilidad de los servicios de intermediación es garantizar que los datos entrantes y salientes a través del sistema sean validados y examinados estructural, lógica y externamente.

También se puede pensar en los servicios de corredor-vecino como una capa de validación sobre las operaciones primitivas que los corredores ya offer.

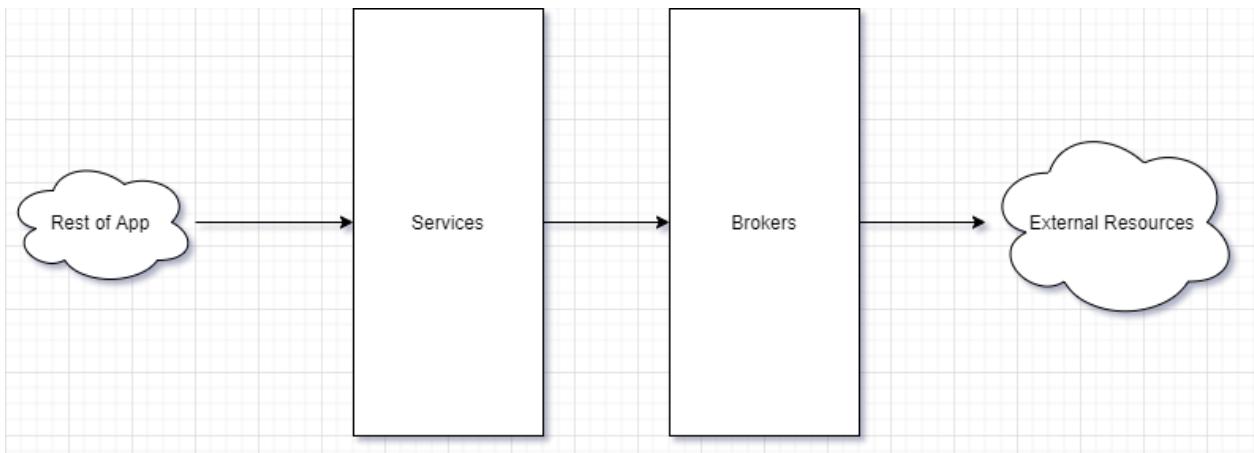
Por ejemplo, si un agente de almacenamiento InsertStudentAsync (Estudian estudian como método, entonces el servicio de corredor-vecino offer algo como lo siguiente:

```
public async ValueTask< Student> AddStudentAsync(Student student)
{
    ValidarEstudiante(estudiante);
    return await this.storageBroker. InsertStudentAsync(estudiante);
}
```

Esto hace que los servicios de "broker-neighboring" no sean más que una capa extra de validación sobre las operaciones primitivas existentes que los brokers ya offer.

2.1.1 En el mapa

Los servicios vecinos al broker residen entre sus brokers y el resto de su aplicación, en el lado izquierdo pueden vivir servicios de procesamiento de lógica de negocio de orden superior, servicios de orquestación, coordinación, agregación o gestión, o simplemente un controlador, un componente de UI o cualquier otra tecnología de exposición de datos.



2.1.2 Características

Los servicios Foundation o Broker-Neighboring en general tienen unas características muy específicas que rigen estrictamente su desarrollo e integración.

Los servicios de la Fundación, en general, se centran más en las validaciones que en cualquier otra cosa, simplemente porque ese es su propósito, asegurarse de que todos los datos entrantes y salientes a través del sistema están en un buen estado para que el sistema los procese de forma segura sin ningún problema.

Estas son las características y las normas que rigen los servicios de intermediación:

2.1.2.0 Puro-Primitivo

No se permite que los servicios vecinos al corredor combinen múltiples operaciones primitivas para lograr una operación de lógica empresarial de orden superior.

Por ejemplo, los servicios de corredor-vecino no pueden ofrecer una función de *upsert*, para combinar operaciones con una selección, actualización o inserción en el resultado para garantizar que una entidad existe y está actualizada en cualquier almacenamiento.

Pero ofrecer una validación y manejo de excepciones (y mapeo) envolver alrededor de las llamadas de dependencia, aquí es un ejemplo:

```

public ValueTask< Student> AddStudentAsync(Student student) =>
TryCatch(async () =>
{
    ValidarEstudiante(estudiante);

    return await this.storageBroker.InsertStudentAsync(estudiante);
});

```

En el método anterior, se puede ver la llamada a la función `ValidateStudent` precedida

por un `tryCatch` bloque. El bloque `TryCatch` es lo que yo llamo Ruido de Excepción

Patrón de cancelación, del que hablaremos próximamente en esta misma sección.

Pero la función de validación asegura que todas y cada una de las propiedades de los datos entrantes sean validadas antes de pasarlas a la operación de ~~intermediación~~^{apoyo}, que es el en este mismo caso.

2.1.2.1 Integración de la entidad única

Los servicios garantizan firmemente la aplicación del principio de responsabilidad única al no integrarse con ningún otro corredor de la entidad, excepto el que apoya.

Esta regla no se aplica necesariamente a los corredores de `DateTimeBroker` o `LoggingBroker` ya que no se dirigen específicamente a ninguna entidad comercial y son casi genéricos en todo el sistema.

Por ejemplo, un servicio de ~~estudiantes~~^{StorageBroker} puede integrarse con ~~un~~^{StorageBroker} como siempre y cuando ~~solo se dirija~~^{Estudiantes} a la funcionalidad offered por la clase parcial en el `StorageBroker.Students.cs` file.

Los servicios de la Fundación no deberían integrarse con más de un corredor de entidades de cualquier tipo, simplemente porque aumentará la complejidad de la validación y la orquestación, lo que va más allá del propósito principal del servicio, que es simplemente la validación. Esta responsabilidad la trasladamos a los servicios de tipo orquestación para que se encarguen de ella.

2.1.2.2 Lenguaje comercial

Los servicios vecinos al Broker hablan un lenguaje empresarial primitivo para sus operaciones. Por ejemplo, mientras que un Broker puede proporcionar un método con el nombre `InsertStudentAsync` - el equivalente de eso en la capa de servicio sería `AddStudentAsync`.

En general, la mayoría de las operaciones CRUD se convertirán de un lenguaje de almacenamiento a un lenguaje de negocio, y lo mismo ocurre con las operaciones que no son de ~~almacenamiento~~^{negocios} operaciones como las colas, `PostQueueMessage` para la capa de negocio `EnqueueMessage`. pero en diremos

Dado que las operaciones CRUD son las más comunes en todo sistema, nuestro mapeo a estas operaciones CRUD sería el siguiente:

Corredores de bolsa	Servicios
---------------------	-----------

Insertar	Añadir
Selección	Recuperar
Actualización	Modificar
Borrar	Eliminar

A medida que avanzamos hacia los servicios de lógica empresarial de orden superior, el lenguaje de los métodos utilizados se inclinará más hacia un lenguaje empresarial que hacia un lenguaje tecnológico, como veremos en las próximas secciones.

2.1.3 Responsabilidades

Los servicios de intermediación desempeñan tres funciones muy importantes en cualquier sistema. La primera función es abstraer las operaciones nativas del corredor del resto del sistema. Independientemente de si un broker es una comunicación entre un almacenamiento local o externo o una API - los servicios broker-neighboring siempre tendrán el mismo contrato/verbigracia para exponer a los servicios de flujo superior como el procesamiento, la orquestación o simplemente los expositores como los controladores o los componentes de la UI. La segunda función, y la más importante, es la de offer una capa de validación sobre las operaciones primitivas existentes que un broker ya offers para asegurar que los datos entrantes y salientes son válidos para ser procesados o persistidos por el sistema. El tercer papel es el de mapeador de todos los demás modelos y contratos nativos que puedan ser necesarios para completar cualquier operación dada mientras se interactúa con un corredor. Los servicios de la Fundación son el último punto de abstracción entre la lógica de negocio central de cualquier sistema y el resto del mundo, vamos a discutir estos roles en detalle.

2.1.3.0 Abstracción

La primera y más importante responsabilidad de los servicios vecinos a la fundación/corredor es asegurar que exista un nivel de abstracción entre los correderos y el resto de su sistema. Esta abstracción es necesaria para asegurar que la capa de lógica de negocio pura en cualquier sistema es verbalmente y funcionalmente agnóstica a cualquier dependencia en la que el sistema se basa para comunicarse con el mundo exterior.

Visualicemos un ejemplo concreto del principio anterior. Supongamos que tener `StudentProcessingService` que implementa una `UpsertStudent` un `funcionalidad`. En algún lugar de esa implementación habrá un `Async` dependencia de `AddStudentAsync` que es expuesta e implementada por algún `StudentService` como servicio de la fundación. Echa un vistazo a este fragmento:

```
public async ValueTask< Student> UpsertStudentAsync(Student student)
{
    ...
    return await this.studentService. AddStudentAsync(student);
}
```

El contrato entre un servicio de procesamiento o de orquestación y un servicio de fundación será siempre el mismo, independientemente del tipo de implementación o del tipo de corredores que utilice el servicio de fundación.

En AddStudentAsync
por ejemplo, podría ser una llamada a una base de datos o a un punto final de la API
o simplemente poner un mensaje en una cola. Aquí hay un ejemplo de tres implementaciones diferentes de un servicio de la fundación que no cambiaría nada en la implementación de sus servicios anteriores:

Con un agente de almacenamiento:

```
public async ValueTask< Student> AddStudentAsync(Student student)
{
    ...
    return await this.storageBroker. InsertStudentAsync(estudiante);
}
```

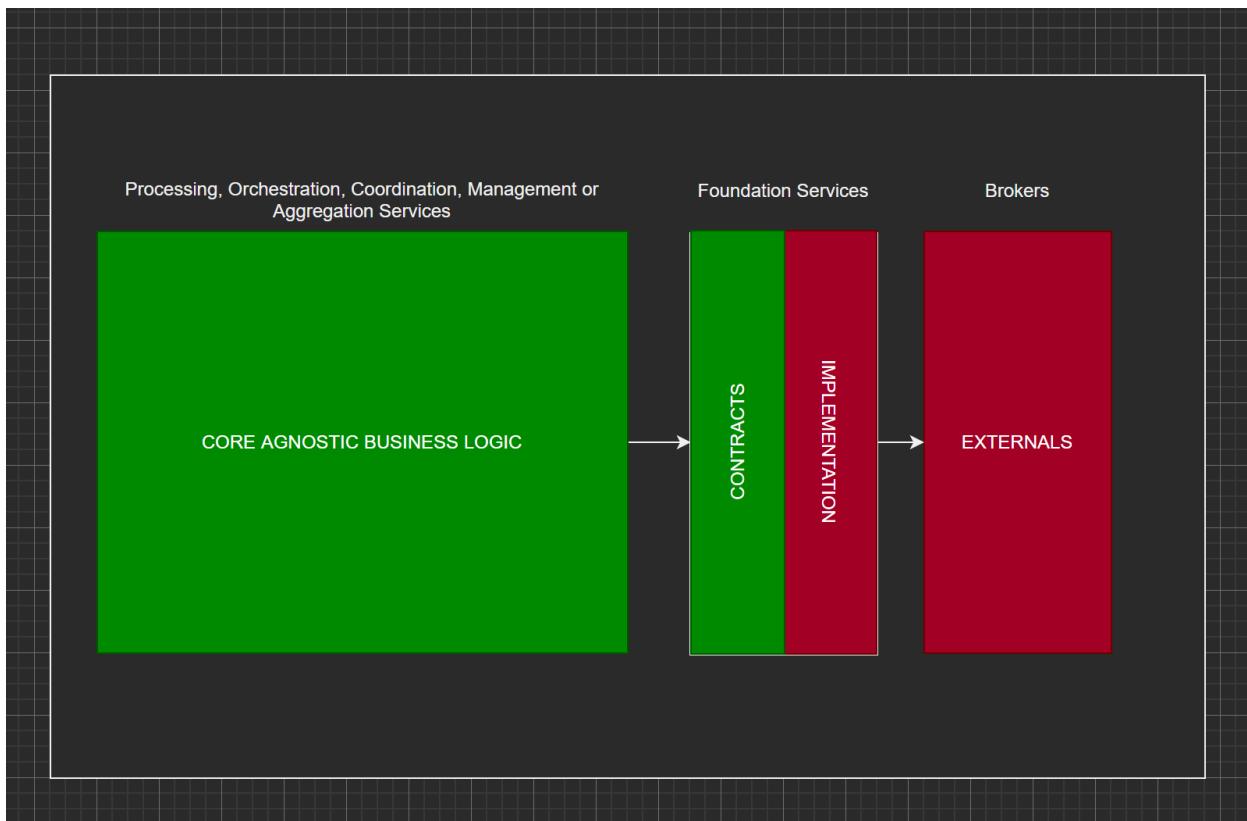
O con un corredor de colas:

```
public async ValueTask< Student> AddStudentAsync(Student student)
{
    ...
    return await this.queueBroker. EnqueueStudentAsync(student);
}
```

o con un agente de la API:

```
public async ValueTask< Student> AddStudentAsync(Student student)
{
    ...
    return await this.apiBroker. PostStudentAsync(student);
}
```

Aquí hay una visualización de ese concepto:



En todos estos casos, la implementación subyacente puede cambiar, pero el contrato expuesto siempre será el mismo para el resto del sistema. En capítulos posteriores discutiremos cómo la lógica de negocio central, agnóstica y abstracta de su sistema comienza con los servicios de Procesamiento y termina con los servicios de Gestión o Agregación.

2.1.3.0.1 Aplicación

Vamos a hablar de un ejemplo real de implementación de una ~~simple~~ en un servicio de fundación. Supongamos que tenemos el siguiente ~~contrato~~ ^{de} ~~ISt~~'a nuestro

udentService:

```
public IStudentService
{
    ValueTask< Student> AddStudentAsync(Student student);
}
```

Para empezar, vamos a escribir una prueba de fallo para nuestro servicio de la siguiente manera:

```

public async Task DeberíaAñadirEstudianteAsync()
{
    // dado
    Estudiante randomStudent = CreateRandomStudent();
    Estudiante inputStudent = randomStudent;
    Alumno de almacenamiento = Alumno de entrada;
    Estudiante esperadoEstudiante = storageStudent. DeepClone();

    this.storageBrokerMock. Setup(broker => broker.
        InsertStudentAsync(inputStudent)
        . DevuelveAsync(almacenamientoEstudiante);

    // cuando
    Estudiante realStudent =
        await this.studentService. AddStudentAsync(inputStudent);

    // entonces actualStudent. Debería().
    BeEquivalentTo(expectedStudent);

    this.storageBroker. Verify(broker => broker.
        InsertStudentAsync(inputStudent),
        Times.Once);

    this.storageBroker. VerifyNoOtherCalls(); this.loggingBroker.
    VerifyNoOtherCalls();
}

```

En la prueba anterior, hemos definido cuatro variables con el mismo valor. Cada variable contiene un nombre que se ajusta al contexto en el que se utilizará. Por ejemplo, `inputStudent` es la que mejor se ajusta a la posición de los parámetros de entrada, mientras que `randomStudent` es la que mejor se ajusta a lo que devuelve el broker de almacenamiento después de que un estudiante sea perseguido con éxito.

También observará que hemos clonado profundamente la variable `storageStudent` a la variable

asegurar que no se han producido modificaciones en el alumno originalmente introducido. Por ejemplo, supongamos que el valor de un estudiante de entrada ha cambiado en cualquiera de sus internamente dentro de la función `AddStudentAsync` función. Ese cambio no provocará un fallo en la prueba a no ser que desreferenciamos la variable de las variables de entrada y de retorno.

Simulamos la respuesta del broker de almacenamiento y ejecutamos nuestro sujeto de prueba `AddStudentAsync` luego verificamos el valor `storageStudent` devuelto `expectedStudent` coincide con la referencia `expectedStudent` independientemente de la del valor esperado.

Por último, verificamos que todas las llamadas se han realizado correctamente y que no se ha realizado ninguna llamada adicional a ninguna de las dependencias del servicio.

Hagamos que esa prueba pase escribiendo una implementación que sólo satisfaga los requisitos de la prueba mencionada:

```

public async ValueTask< Student> AddStudentAsync(Student student)
=> await this.storageBroker. InsertStudentAsync(student);

```

Esta simple implementación debería hacer que nuestra prueba pase con

éxito. Es importante entender que cualquier implementación debe ser sólo lo suficiente para pasar las pruebas que fallan. Nada más y nada menos.

2.1.3.1 Validación

Los servicios de la Fundación son necesarios para garantizar que los datos entrantes y salientes del sistema están en buen estado: desempeñan el papel de guardián entre el sistema y el mundo exterior para garantizar que los datos que pasan son estructuralmente, lógicamente y externamente válidos antes de realizar cualquier otra operación por parte de los servicios anteriores. El orden de las validaciones es muy intencionado. Las validaciones estructurales son las más baratas de los tres tipos. Garantizan que un atributo concreto o un dato en general no tenga un valor por defecto si es necesario. Lo contrario son las validaciones lógicas, en las que los atributos se comparan con otros atributos de la misma entidad o de cualquier otra. Las validaciones lógicas adicionales también pueden incluir una comparación con un valor constante, como la comparación de la edad de inscripción de un estudiante para que no sea menor de 5 años. Tanto las validaciones estructurales como las lógicas son anteriores a las externas. Como hemos dicho, es simplemente porque no queremos pagar el coste de la comunicación con un recurso externo, incluyendo la tasa de ~~Estudia si~~ nuestra petición no está en buena forma firme. Por ejemplo, ~~no~~deberíamos tratar de publicar algunos

a una API externa si el objeto es `null`. O si es inválido modelo estructural o lógicamente.

Para todos los tipos de validaciones, es importante entender que algunas validaciones son de ruptura de circuito o requieren una salida inmediata del flujo actual lanzando una excepción o devolviendo un valor en algunos casos. Y otras validaciones son continuas. Vamos a hablar primero de estas dos subcategorías de validaciones.

2.1.3.1.0 Validaciones para romper el circuito

Las validaciones para romper el circuito requieren una salida inmediata del flujo actual.

Por ejemplo, si un objeto que se pasa a una función es `null` - habrá no se requieren más operaciones a ese nivel que salir de la actual flow lanzando una excepción o devolviendo un valor de algún tipo. Aquí hay un ejemplo: En algún escenario de validación, supongamos ~~que~~ nuestra función tiene un alumno de valor `null` pasado en él de la siguiente manera:

```
Estudiante noStudent = null;  
await this.studentService.AddStudentAsync(noStudent);
```

Nuestra ~~función~~ `AddStudentAsync` función en este escenario se requiere ahora para validar si el parámetro pasado es `null` o no antes de seguir adelante con cualquier otro tipo de validaciones o la propia lógica de negocio. Algo así:

```

public EstudianteAñadirEstudianteAsync(Estudiante
estudiante) => TryCatch(async () =>
{
    ValidarEstudiante(estudiante);

    return await this.storageBroker.InsertStudentAsync(estudiante);
});

```

El enunciado en cuestión es `ValidarEstudiante` función y lo que hace. Este es un ejemplo de cómo se implementaría esa rutina:

```

private void ValidateStudent(Student student)
{
    if(student is null)
    {
        lanza una nueva NullStudentException();
    }
}

```

En la función de arriba, decidimos lanzar la excepción inmediatamente en lugar de ir más allá. Ese es un ejemplo de tipo de validación que rompe el circuito.

Pero con las validaciones, romper el circuito no siempre es lo más acertado. A veces queremos recopilar todos los problemas de una solicitud concreta antes de enviar el informe de errores al remitente de la solicitud. Vamos a hablar de eso en la siguiente sección.

2.1.3.1.1 Validaciones continuas

Las validaciones continuas son lo contrario de las validaciones de ruptura de circuitos. No detienen el flujo de validaciones, pero sí definen el flujo de lógica. En otras palabras, las validaciones continuas garantizan que no se ejecute ninguna lógica de negocio, pero también garantizan que otras validaciones del mismo tipo puedan seguir ejecutándose antes de romper el circuito. Materialicemos esta teoría con un ejemplo: Supongamos que nuestro modelo de estudiante tiene este aspecto:

```

public class Estudiante
{
    public Guid Id {get; set;}
    public string Name {get;
    set;}
}

```

Suponiendo que el modelo `pasado` no es nulo, sino que tiene un modelo por defecto

para todas sus propiedades. Queremos recopilar todas estas cuestiones para todos los atributos/propiedades que tenga este objeto y devolver un informe completo al solicitante. Esto es lo que hay que hacer.

2.1.3.1.1.0 Excepciones de la mesa superior

Un problema de este tipo requiere un tipo especial de excepciones que permitan recogiendo todos los errores en su propiedad. Todas las excepciones nativas que hay

contendrá la propiedad que es básicamente un diccionario para una clave/valor

para recoger más información sobre los problemas que causaron esa excepción. El problema con estas excepciones nativas es que no tienen soporte nativo para el upsertion. Es decir, la posibilidad de añadir a una lista de valores existente una clave determinada en cualquier momento. Aquí hay una implementación nativa de upserting valores en algún diccionario dado:

```
varException = new Exception(); if(someException.Data.  
Contains(someKey))  
{  
    (algunaExcepción.Datos[algunaClave] como Lista< cadena>) ? Add(someValue);  
}  
si no  
{  
    algunaExcepción.Datos. Add(someKey, new List< string>{ someValue });  
}
```

Esta implementación puede ser bastante desalentadora para que los ingenieros piensen y prueben en su implementación a nivel de servicio. Se consideró más apropiado Xeptions

introducir una simple biblioteca en algo tan sencillo como:

```
var someException = new Xeption();  
someException.UpsertData(someKey, someValue);
```

Ahora que tenemos esta biblioteca para utilizar, la preocupación de implementar excepciones updatable se ha abordado. Esto significa que tenemos lo necesario para recoger nuestros errores de validación. Pero eso no es suficiente si no tenemos un mecanismo para romper el circuito cuando creemos que es el momento adecuado para hacerlo. Podemos simplemente usar los offerings nativos para implementar la ruptura del circuito directamente como sigue:

```
if(algunaExcepción.Datos.Cuenta > 0)  
{  
    lanzar algunaExcepción;  
}
```

Y aunque esto puede ser fácilmente incorporado a cualquier implementación existente. Todavía no contribuyó mucho a la apariencia general del código.

Por lo tanto, tengo Xeptions

decidió hacerla parte de los siguientes:

biblioteca para ser simplificado a la

```
algunaExcepción.ThrowIfContainsErrors();
```

Eso haría que nuestras validaciones personalizadas tuvieran un aspecto similar

al siguiente:

```
public class InvalidStudentException : Xception
{
    public InvalidStudentException()
        : base ("El alumno no es válido. Por favor, corrija los errores e inténtelo de nuevo")
    { }
}
```

Pero con las validaciones continuas, el proceso de recogida de estos errores transmite algo más que una implementación de excepción especial. Vamos a hablar de ello en la siguiente sección.

2.1.3.1.1.1 Reglas dinámicas

Un proceso de validación continuo o que no rompa el circuito requerirá la capacidad de pasar reglas dinámicas en cualquier cuenta o capacidad para añadir estos errores de validación. Una regla de validación es una estructura dinámica que informa si la regla ha sido violada para su condición; y también el mensaje de error que debe ser reportado al usuario final para ayudarle a fixar ese problema.

En un escenario en el que queramos asegurarnos de que cualquier Id es válido, una regla de validación continua dinámica sería algo así:

```
private static dynamic IsValid(Guid id) => new
{
    Condición = id == Guid.Empty, Mensaje = "El
    id es obligatorio"
};
```

Ahora nuestra regla no sólo informa de si un determinado atributo es inválido o no. También tiene un mensaje significativo y legible para el ser humano que ayuda al consumidor del servicio a entender qué es lo que hace que ese mismo atributo sea inválido.

Es muy importante señalar el lenguaje que los ingenieros deben utilizar para los mensajes de validación. Todo dependerá de los consumidores potenciales de su sistema. Una persona que no sea ingeniero no entenderá ~~en mensaje~~ ser nulo, vacío o espacio en blanco - como término no es algo que se de uso muy común. Los ingenieros deben trabajar estrechamente con su meatware (las personas que utilizan el sistema) para asegurarse de que el lenguaje tiene sentido para ellos.

Las reglas dinámicas por diseño permitirán a los ingenieros modificar tanto sus entradas como sin romper ninguna funcionalidad existente siempre que se consideran de forma generalizada. Esta es otra manifestación de una regla de validación dinámica:

```
private static dynamic IsNotSame(
    Guid firstId,
    Guid secondId,
    string secondIdName) => new
{
    Condición = firstId != secondId,
    Message = $"Id no es el mismo que {secondIdName}.",
    HelpLink = "/help/code1234"
};
```

Nuestra regla dinámica ahora puede offer más parámetros de entrada y más

información útil en términos de un mensaje de excepción más detallado con enlaces a sitios de documentación útiles o referencias para los códigos de error.

2.1.3.1.1.2 Colección de reglas y validaciones

Ahora que tenemos las excepciones avanzadas y las reglas de validación dinámicas. Es hora de ponerlo todo junto en términos de aceptar un número infinito de reglas de validación, examinar los resultados de sus condiciones y finalmente romper el circuito cuando todas las validaciones continuas están hechas:

```
private void Validate(params (Regla dinámica, cadena Parámetro)[] validaciones)
{
    var invalidStudentException = new InvalidStudentException();

    foreach((regla dinámica, parámetro de cadena) en validaciones)
    {
        if(rule.Condition)
        {
            invalidStudentException. UpsertData(parámetro, regla.Mensaje);
        }
    }

    invalidStudentException. ThrowIfContainsErrors();
}
```

La función anterior ahora tomará cualquier número de reglas de validación, y los parámetros de la regla se está ejecutando contra entonces examinar las condiciones y upsert el informe de errores. Así es como podemos utilizar el método anterior:

```
private void ValidateStudent(Student student)
{
    Validar(
        (Regla: IsInvalid(estudiante.Id), Parámetro:
        nameof(estudiante.Id)), (Regla: IsInvalid(estudiante.nombre),
        Parámetro: nameof(estudiante.nombre)), (Regla:
        IsInvalid(estudiante.calificación), Parámetro:
        nameof(estudiante.calificación))
    );
}
```

Esta simplificación de la escritura de las reglas y las validaciones es el objetivo final de seguir aportando valor a los usuarios finales y, al mismo tiempo, hacer que el proceso de ingeniería de la solución sea agradable para los propios ingenieros.

Ahora, vamos a profundizar en los tipos de validaciones que nuestros sistemas pueden ofrecer y cómo manejarlos.

2.1.3.1.2 Validaciones estructurales

Las validaciones son tres capas distintas. la primera de estas capas son las validaciones estructurales. para asegurar que ciertas propiedades en cualquier modelo dado o un tipo primitivo no están en un estado estructural inválido.

Por ejemplo, una propiedad de tipo `cadena` no debe estar vacía, ser `nula` o ser blanca otro ejemplo sería para un parámetro de entrada de tipo `int`, es `no debería estar en por defecto` estado que es `0` al intentar introducir una edad su instancia.

Las validaciones estructurales aseguran que los datos están en buena forma antes de avanzar con cualquier otra validación - por ejemplo, no podemos validar que un estudiante tenga el número mínimo de caracteres (que es una validación lógica) en sus nombres si su primer nombre es estructuralmente inválido por ser nulo, vacío o con espacios en blanco.

Las validaciones estructurales juegan el papel de identificar las propiedades *requeridas* en cualquier modelo dado, y mientras que una gran cantidad de tecnologías ofrecen las anotaciones de validación, plugins o bibliotecas para hacer cumplir globalmente las reglas de validación de datos, elijo para llevar a cabo la validación de forma programática y manual para obtener más control de lo que sería necesario y lo que no de una manera TDD.

El problema con algunas de las implementaciones actuales de las validaciones estructurales y lógicas en los modelos de datos es que pueden cambiarse muy fácilmente bajo el radar sin que ninguna prueba de unidad haga saltar las alarmas. Mira este ejemplo por ejemplo:

```
público Estudiante
{
    [Requerido]
    public string Nombre {get; set;}
}
```

El ejemplo anterior puede ser muy atractivo a simple vista desde el punto de vista de la ingeniería. Todo lo que tienes que hacer es decorar tu atributo del modelo con una anotación mágica y de repente tus datos están siendo validados.

El problema aquí es que este patrón combina dos responsabilidades diferentes o más en el mismo modelo. Se supone que los modelos son sólo una representación de los objetos en la realidad, nada más y nada menos. Algunos ingenieros los llaman modelos anémicos, que centran la responsabilidad de cada modelo en representar únicamente los atributos del objeto del mundo real que intenta simular, sin ningún detalle adicional.

Pero los modelos anotados intentan ahora inyectar lógica de negocio en sus propias definiciones. Esta lógica de negocio puede o no ser necesaria en todos los servicios, corredores o componentes de exposición que los utilizan.

Las validaciones estructurales en los modelos pueden parecer un trabajo extra que se puede evitar con decoraciones mágicas. Pero en el caso de intentar desviarse ligeramente de estas validaciones hacia unas validaciones más personalizadas, ahora verás surgir un nuevo antipatrón como son las anotaciones personalizadas que pueden o no ser detectables a través de las pruebas unitarias.

Vamos a hablar de cómo probar una rutina de validación estructural:

2.1.3.1.2.0 Pruebas de validación estructural

Porque realmente creo en la importancia de TDD, voy a empezar a mostrar la implementación de las validaciones estructurales escribiendo primero una prueba que falla.

Supongamos que tenemos un modelo de estudiante, con los siguientes detalles:

```
public class Estudiante
{
    public Guid Id {get; set;}
}
```

Queremos validar que el Id del estudiante no es un Id estructuralmente inválido

- como `Guid`

como una `moda vacía:` - por lo que escribiríamos una prueba unitaria de la siguiente manera

```
[Hecho]
public async void ShouldThrowValidationExceptionOnRegisterWhenIdIsInvalidAndLogItAsync()
{
    // dado
    Estudiante randomStudent = CreateRandomStudent();
    Estudiante inputStudent = randomStudent;
    inputStudent.Id = Guid.Empty;

    var invalidStudentException = new InvalidStudentException();

    invalidStudentException.AddData(
        clave: nameof(Estudiante.Id),
        valor: "La identificación es necesaria"
    );

    var expectedStudentValidationException =
        new StudentValidationException(invalidStudentException);

    // cuando
    ValueTask< Student> registerStudentTask = this.studentService.
        RegisterStudentAsync(inputStudent);

    StudentValidationException actualStudentValidationException =
        await Assert.ThrowsAsync< StudentValidationException>(
            registerStudentTask.AsTask);

    // entonces actualStudentValidationException. Debería().
    BeEquivalentTo(
        expectedStudentValidationException);

    this.loggingBrokerMock.Verify(broker => broker.LogError(It.
        Is(expectedStudentValidationException)),
        Times.Once);

    this.storageBrokerMock.Verify(broker => broker.InsertStudentAsync(It. IsAny<
        Student>()),
        Times.Never);

    this.dateTimeBrokerMock.VerifyNoOtherCalls(); this.loggingBrokerMock.
    VerifyNoOtherCalls(); this.storageBrokerMock. VerifyNoOtherCalls();
}
```

En la prueba anterior, creamos un objeto estudiante al azar y luego asignamos el objeto an

valor de Id inválido `Guid.Empty` al Id. del estudiante.

Cuando la lógica de validación estructural en nuestro servicio de la fundaciónd examina la propiedad, debe lanzar una propiedad de excepción que describa el problema de validación en nuestro modelo de estudiante. en este caso lanzamos `InvalidStudentException`.

La excepción es necesaria para describir brevemente el qué, el dónde y el por qué de la operación de validación. en nuestro caso, el qué sería el problema de validación que se está produciendo, el dónde sería el servicio Student y el por qué sería el valor de la propiedad.

Así es como un `InvalidStudentException` sería:

```
public class InvalidStudentException : Exception
{
    public InvalidStudentException()
        :base ("El estudiante no es válido. Por favor, corrija los errores e inténtelo de nuevo")
    { }
}
```

La prueba unitaria anterior, sin embargo, requiere que nuestra `InvalidStudentException` esté envuelta en una excepción más genérica a nivel de sistema, que es `StudentValidationException`.

- estas excepciones es lo que yo llamo excepciones externas, encapsulan todas las situaciones diferentes de las validaciones independientemente de su categoría y comunica el error a los servicios o controladores ascendentes para que puedan mapear eso al código de error apropiado para el consumidor de estos servicios.

Nuestro `StudentValidationException` se aplicaría de la siguiente manera:

```
public class AlumnoValidaciónExcepción : Excepción
{
    public StudentValidationException(Exception innerException)
        : base("Se ha producido un error de validación del estudiante, por favor compruebe su entrada y vuelva a intentarlo.", innerException) { }
}
```

El mensaje en la validación externa anterior indica que el problema está en la entrada, y por lo tanto requiere que el remitente de la entrada lo intente de nuevo ya que no hay acciones requeridas desde el lado del sistema para ser ajustado.

2.1.3.1.2.1 Aplicación de las validaciones estructurales

Ahora, veamos el otro lado del proceso de validación, que es la implementación. Las validaciones estructurales siempre van por delante de todos y cada uno de los otros tipos de validaciones. Esto se debe simplemente a que las validaciones estructurales son las más baratas desde el punto de vista de la ejecución y el tiempo asintótico. Por ejemplo, Es mucho más barato validar un `string` que no es válido estructuralmente, que enviar un

API para obtener exactamente la misma respuesta, más el coste de la latencia. Todo esto se acumula cuando empiezan a llegar varios millones de peticiones por segundo. Las validaciones estructurales y lógicas en general viven en su propia clase parcial para ejecutar es `StudentService.cs` validaciones, por ejemplo, si nuestro servicio se llama nuevo file debe crearse con el nombre `StudentService.Validations.cs` entonces para encapsular y abstraer visualmente

las reglas de validación para asegurar que los datos entran y salen limpios.
Así es como se vería una validación de Id:

StudentService.Validations.cs

```
private void ValidateStudent(Student student)
{
    Validate((Regla: IsInvalid(estudiante.Id), Parámetro: nameof(estudiante.Id));
}

private static dynamic IsInvalid(Guid id) => new
{
    Condición = id ==
    Guid.Empty, Mensaje = "El
    id es obligatorio"
};

private void Validate(params (Regla dinámica, cadena Parámetro)[] validaciones)
{
    var invalidStudentException = new InvalidStudentException();

    foreach((regla dinámica, parámetro de cadena) en validaciones)
    {
        if(rule.Condition)
        {
            invalidStudentException. UpsertData(parámetro, regla.Mensaje);
        }
    }

    invalidStudentException. ThrowIfContainsErrors();
}
```

Hemos implementado un método para validar el objeto estudiante completo, con una compilación de todas las reglas que necesitamos configurar para validar estructural y lógicamente el objeto de entrada del estudiante. La parte más importante a notar sobre el fragmento de código anterior es asegurar la encapsulación de cualquier detalle finer más lejos del objetivo principal de un método particular.

Esta es la razón por la que decidimos implementar un método estático privado `IsInvalid` para abstraer los detalles de lo que determina que una propiedad de tipo `Guid` sea inválida o no. Y a medida que avancemos en la implementación, vamos a tener que implementar múltiples sobrecargas del mismo método para validar otros tipos de valores estructural y lógicamente.

El propósito del método `ValidateStudent` es simplemente establecer las reglas

y tomar una acción lanzando una excepción si se viola alguna de estas reglas. Siempre hay una oportunidad para agregar los errores de violación en lugar de lanzar demasiado pronto a la primera señal de problema de validación estructural o lógica que se detecte.

Ahora, con la implementación anterior, necesitamos llamar a ese método para validar estructural y lógicamente nuestra entrada. Hagamos esa `RegisterStudentAsync`

método de la siguiente manera:

StudentService.cs

```
public ValueTask< Student> RegisterStudentAsync(Student student) =>
TryCatch(async () =>
{
    ValidarEstudiante(estudiante);

    return await this.storageBroker. InsertStudentAsync(estudiante);
});
```

A simple vista, notarás que nuestro método aquí no maneja necesariamente ningún tipo de excepciones a nivel lógico. Eso es porque todo el ruido de las excepciones también se abstrae en un método llamado TryCatch.

TryCatch es un concepto que inventé para permitir a los ingenieros centrarse en lo que más importa en función del aspecto del servicio que están mirando sin tener que tomar ningún atajo con el manejo de excepciones para hacer el código un poco más legible.

TryCatch métodos en general viven en otra clase parcial, y un nuevo archivo ~~Service~~.Exceptions.cs - que es donde se encuentran todos los el manejo de las excepciones y el reporte de errores ocurre como lo mostraré en el siguiente ejemplo.

Veamos cómo es un método TryCatch método:

StudentService.Exceptions.cs

```
private delegate TareaValor< Alumno> DevoluciónFunciónAlumno();

private async ValueTask< Student> TryCatch(ReturningStudentFunction returningStudentFunction)
{
    pruebe con
    {
        return await returningStudentFunction();
    }
    catch (InvalidStudentException invalidStudentInputException)
    {
        Lanzar CreateAndLogValidationException(invalidStudentInputException);
    }
}

private StudentValidationException CreateAndLogValidationException(Excepción)
{
    var studentValidationException = new StudentValidationException(exception);
    this.loggingBroker.LogError(studentValidationException);

    return studentValidationException;
}
```

El TryCatch patrón de cancelación de ruido de excepción toma maravillosamente cualquier que devuelve un tipo particular como delegado y maneja cualquier excepción lanzada off de esa función o sus dependencias.

La principal responsabilidad de TryCatch es envolver un servicio interno Hay subtipos de estas excepciones, como las Excepciones de Validación de Dependencia, pero normalmente se incluyen en la categoría de Excepción de Validación, como veremos en las próximas secciones de la Norma.

En un `tryCatch` método, podemos añadir tantas excepciones internas y externas como queremos y los mapeamos en excepciones locales para que los servicios ascendentes no tengan una fuerte dependencia de ninguna biblioteca en particular o de modelos de recursos externos, de lo que hablaremos en detalle cuando pasemos a la responsabilidad de mapeo de los servicios (de la fundación) del corredor.

2.1.3.1.3 Validaciones lógicas

Las validaciones lógicas son las segundas en orden a las validaciones estructurales. su principal responsabilidad por definición es validar lógicamente si un dato estructuralmente válido es lógicamente válido. Por ejemplo, la fecha de nacimiento de un estudiante podría ser estructuralmente válido al tener un `v1/1/1800` pero lógicamente, un estudiante de más de 200 años es un imposible.

Las validaciones lógicas más comunes son las validaciones de los campos de auditoría como

Creando y Actualiza - es lógicamente imposible que un nuevo registro por puedaPor con dos valores diferentes para los autores de ese nuevo registro - simplemente porque los datos sólo pueden ser insertados por una persona a la vez.

Hablemos de cómo podemos probar e implementar las validaciones lógicas:

2.1.3.1.3.0 Prueba de validaciones lógicas

En el caso común de probar las validaciones lógicas para los campos de auditoría, queremos que el `UpdatedBy` lance una excepción de validación porque no coincide con el Campo "CreatedBy". no es válido simplemente

Supongamos que nuestro modelo de estudiante es el siguiente:

```
public class Estudiante {  
    Guid CreatedBy {get;  
        set;} Guid UpdatedBy  
    {get; set;}  
}
```

Nuestra prueba para validar estos valores lógicamente sería la siguiente:

```

[Hecho]
public async Task ShouldThrowValidationExceptionOnRegisterIfUpdatedByNotSameAsCreatedByAndLogItAsync()
{
    // dado
    Student randomStudent = CreateRandomStudent();
    Student inputStudent = randomStudent;
    inputStudent.UpdatedBy = Guid.NewGuid();

    var invalidStudentException = new InvalidStudentException();

    invalidStudentException.AddData(
        clave: nameof(Student.UpdatedBy),
        value: $"El Id no es el mismo que {nameof(Student.CreatedBy)}.");

    var expectedStudentValidationException =
        new StudentValidationException(invalidStudentException);

    // cuando
    ValueTask< Student> registerStudentTask = this.studentService.
        RegisterStudentAsync(inputStudent);

    StudentValidationException actualStudentValidationException =
        await Assert.ThrowsAsync< StudentValidationException>(
            registerStudentTask.AsTask);

    // entonces actualStudentValidationException. Debería().
    BeEquivalentTo(
        expectedStudentValidationException);

    this.loggingBrokerMock.Verify(broker => broker.LogError(It.
        IsSameExceptionAs(expectedStudentValidationException)),
        Times.Once);

    this.storageBrokerMock.Verify(broker => broker.InsertStudentAsync(It. IsAny<
        Student>()),
        Times.Never);

    this.loggingBrokerMock.VerifyNoOtherCalls(); this.dateTimeBrokerMock.
    VerifyNoOtherCalls(); this.storageBrokerMock.VerifyNoOtherCalls();
}

```

En la prueba anterior, hemos cambiado el valor del campo `Actualiza` campo a asegurar que difiere completamente del campo `CreatedBy`. Ahora esperamos que se produzca esta excepción de validación.

Sigamos adelante y escribamos una implementación para esta prueba que falla.

2.1.3.1.3.1 Implementación de validaciones lógicas

Al igual que hicimos en la sección de validaciones estructurales, vamos a añadir más reglas para nuestra validación de cambio de la siguiente manera:

StudentService.Validations.cs

```
private void ValidateStudent(Student student)
{
    Validar(
        Regla: IsNotSame(
            firstId: student.UpdatedBy,
            secondId: student.CreatedBy,
            secondIdName: nameof(student.CreatedBy)),
        Parameter: nameof(Student.UpdatedBy))
};

private static dynamic IsNotSame(
    Guid firstId,
    Guid secondId,
    string secondIdName) => new
{
    Condición = firstId != secondId,
    Mensaje = $"Id no es el mismo que {secondIdName}".
};

private void validate(params (dynamic Rule, string Parameter)[] validaciones)
{
    var invalidStudentException = new Exception();

    foreach((regla dinámica, parámetro de cadena) en validaciones)
    {
        if(rule.Condition)
        {
            invalidStudentException. UpsertData(
                clave: parámetro,
                valor: regla.Mensaje);
        }
    }
}
```

Todo lo demás en ambos `studentService.cs` y `Servicio de Estudiantes.cs` sigue siendo exactamente lo mismo que hemos hecho anteriormente en las validaciones estructurales.

Las excepciones de las validaciones lógicas, al igual que cualquier otra excepción que pueda ocurrir, no suelen ser críticas. Sin embargo, todo depende de su caso de negocio para determinar si una validación lógica, estructural o incluso una dependencia en particular son críticas o no, esto es cuando podría necesitar crear una clase especial de excepciones, algo así como

y luego registrarla como corresponde.

2.1.3.1.4 Validaciones externas

El último tipo de validaciones que suelen realizar los servicios de la fundación son las validaciones externas. Defino las validaciones externas como cualquier forma de validación que requiera llamar a un recurso externo para validar si un servicio de la fundación debe continuar con el procesamiento de los datos entrantes o detenerse con una excepción.

Un buen ejemplo de validaciones de dependencia es cuando llamamos a un agente para recuperar una entidad concreta por su id. Si la entidad devuelta no se encuentra, o el `NotFound`

El agente de la API devuelve un error - el servicio de la fundación es necesario para envolver ese error en un proceso siguiente. `ValidationException` ene todos los

Las excepciones de validación externa pueden producirse si el valor devuelto no se ajusta a las expectativas, como una lista vacía devuelta por una llamada a la API al intentar insertar un nuevo entrenador de un equipo: si no hay miembros del equipo, no puede haber entrenador, por ejemplo. En este caso, el servicio de la fundación deberá lanzar una excepción local para `CompoundException` ema, algo así como o algo por el estilo.

Escribamos una prueba que falle para un ejemplo de validación externa:

2.1.3.1.4.0 Pruebas de validaciones externas

Supongamos que intentamos recuperar un alumno con `uid` que no tiene coincidencia con algún registro de la base de datos. Así es como haríamos la prueba este escenario. Primero off, vamos a `NotFoundStudentException` modelo como a definir lo siguiente:

```
utilizando Xception;

public class NotFoundStudentException : Xception
{
    public NotFoundStudentException(Guid id)
        : base (mensaje: $"No se pudo encontrar un estudiante con el id: {id} .")
    {}
}
```

El modelo anterior es el aspecto de localización de la gestión del problema. Ahora vamos a escribir una prueba que falla de la siguiente manera:

```
public async Task ShouldThrowValidationExceptionOnRetrieveByIdIfStudentNotFoundAndLogItAsync()
{
    // dado
    Guid randomStudentId = Guid.
    NewGuid(); Guid inputStudentId =
    randomStudentId; Student noStudent =
    null;

    var notFoundStudentException =
        new NotFoundStudentException(inputStudentId);

    var expectedStudentValidationException =
        new StudentValidationException(notFoundStudentException);

    this.storageBrokerMock. Setup(broker => broker.
        SelectStudentByIdAsync(inputStudentId))
        . DevuelveAsync(noStudent);

    // cuando
    ValueTask< Student> retrieveStudentByIdTask = this.studentService.
    RetrieveStudentByIdAsync(inputStudentId);

    StudentValidationException actualStudentValidationException =
        await Assert.ThrowsAsync< StudentValidationException>(
            retrieveStudentByIdTask.AsTask);

    // entonces actualStudentValidationException. Debería().
    BeEquivalentTo(
        expectedStudentValidationException);

    this.storageBrokerMock. Verify(broker => broker.
        SelectStudentByIdAsync(inputStudentId),
        Times.Once);

    this.loggingBrokerMock. Verify(broker => broker. LogError(It.
        Is(expectedStudentValidationException)),
        Times.Once);

    this.storageBrokerMock. VerifyNoOtherCalls(); this.loggingBrokerMock.
    VerifyNoOtherCalls(); this.dateTimeBrokerMock. VerifyNoOtherCalls();
}
```

La prueba anterior requiere que lancemos una excepción localizada como `NotFoundStudentException` cuando el agente de almacenamiento no devuelve valores para el `studentId` dado y luego envolver o categorizar esto en `StudentValidationException`.

Elegimos envolver la excepción localizada en una excepción de validación y no en una excepción de validación de dependencia porque el inicio del error se originó en nuestro servicio y no en el recurso externo. Si el recurso externo es la fuente del error, tendríamos que categorizarlo como una `DependencyValidationException` de la que hablaremos en breve.

Ahora vamos a la parte de la implementación de esta sección para hacer que nuestra prueba pase.

2.1.3.1.4.1 Implementación de validaciones externas

Para implementar una validación externa necesitaremos tocar todos los diferentes aspectos de nuestro servicio. El núcleo de la lógica, la validación y los aspectos de gestión de excepciones son los siguientes.

Primero off, vamos a construir una función de validación `NotFoundStudentException` que lanzará un si el parámetro pasado es nulo.

StudentService.Validations.cs

```
private static void VerifyStudentExists(Student maybeStudent, Guid studentId)
{
    si (maybeStudent es null)
    {
        lanzar una nueva NotFoundStudentException(studentId);
    }
}
```

Esta implementación se encargará de detectar un problema y emitir una excepción local `NotFoundStudentException`. Ahora pasemos al aspecto del manejo de excepciones de nuestro servicio.

StudentService.Exceptions.cs

```
private async ValueTask< Student> TryCatch(ReturningStudentFunction returningStudentFunction)
{
    pruebe con
    {
        return await returningStudentFunction();
    }
    ..
    catch (NotFoundStudentException notFoundStudentException)
    {
        Lanzar CreateAndLogValidationException(notFoundStudentException);
    }
}

private StudentValidationException CreateAndLogValidationException(Excepción)
{
    var studentValidationException = new StudentValidationException(exception);
    this.loggingBroker.LogError(studentValidationException);

    return studentValidationException;
}
```

La implementación anterior se encargará de categorizar un `NotFoundStudentException` a `StudentValidationException`. La última parte es unir las piezas de la siguiente manera.

StudentService.cs

```
public ValueTask< Student> RetrieveStudentByIdAsync(Guid studentId) =>
TryCatch(async () =>
{
    ValidarIdDeEstudiante(studentId);

    Estudiante maybeStudent =
        await this.storageBroker. SelectStudentByIdAsync(studentId);

    ValidateStudentExists(maybeStudent, studentId);

    devolver maybeStudent;
});
```

La implementación anterior se asegurará de que el id es válido, pero lo más importante es que se comprobará si lo que devuelve el `storageBroker` es un objeto o es `null`. Entonces emite la excepción.

Hay situaciones en las que intentar recuperar una entidad y luego descubrir que no existe no es necesariamente erróneo. Aquí es donde entran los Servicios de Procesamiento para aprovechar una lógica de negocio de orden superior para hacer frente a este escenario más complejo.

2.1.3.1.5 Validaciones de dependencia

Las excepciones de validación de dependencia pueden ocurrir porque se ha llamado a un recurso externo y éste ha devuelto un error, o ha devuelto un valor que justifica el lanzamiento de 404 un error. Por ejemplo, una llamada a la API puede devolver un código código, y eso es se interpreta como una excepción si la entrada debía corresponder a un objeto existente.

Un ejemplo más común es cuando una entidad de entrada particular está usando el mismo id que una entidad existente en el sistema. En el mundo de las bases de datos relacionales, se lanzaría una excepción de clave duplicada. En un escenario de API RESTful, la aplicación programática del mismo concepto también logra el mismo objetivo para las validaciones de la API, asumiendo que la granularidad del sistema que se llama debilita la integridad referencial de los datos del sistema en general.

Hay situaciones en las que la respuesta defectuosa puede expresarse de forma distinta a las excepciones, pero tocaremos ese tema en un capítulo más avanzado de esta Norma.

Escribamos una prueba que falle para verificar si estamos lanza una excepción si el modelo de estudiante ya existe en el almacenamiento y el agente de almacenamiento lanza una DuplicateKeyException como resultado nativo de la operación.

2.1.3.1.5.0 Prueba de validaciones de dependencia

Supongamos que nuestro modelo de estudiante utiliza un Id con el tipo Guid de la siguiente manera:

```
public class Estudiante
{
    public Guid Id {get; set;}
    public string Name {get;
    set;}
}
```

nuestra prueba unitaria para validar que se lance una excepción DependencyValidation en una situación de DuplicateKey sería la siguiente:

```
[Hecho]
public async void ShouldThrowDependencyValidationExceptionOnRegisterIfStudentAlreadyExistsAndLogItAsync()
{
    // dado
    Estudiante algúnEstudiante =
    CreateRandomStudent(); cadena algúnMensaje =
    GetRandomMessage ();
    var duplicateKeyException = new DuplicateKeyException(exceptionMessage);

    var alreadyExistsStudentException =
        new AlreadyExistsStudentException(duplicateKeyException);

    var expectedStudentDependencyValidationException =
        new StudentDependencyValidationException(alreadyExistsStudentException);

    this.storageBrokerMock. Setup(broker => broker. InsertStudentAsync<It. IsAny<
        Student>())
        . ThrowsAsync(duplicateKeyException);

    // cuando
    ValueTask< Student> registerStudentTask = this.studentService.
    RegisterStudentAsync(inputStudent);

    StudentDependencyValidationException actualStudentDependencyValidationException =
        await Assert. ThrowsAsync< StudentDependencyValidationException>(
            registerStudentTask.AsTask);

    // entonces actualStudentDependencyValidationException. Debería().
    BeEquivalentTo(
        expectedStudentDependencyValidationException);

    this.storageBrokerMock. Verify(broker => broker. InsertStudentAsync<It. IsAny<
        Student>()),
        Times.Once);

    this.loggingBrokerMock. Verify(broker => broker. LogError(It.
        Is
```

En la prueba anterior, validamos que envolvemos una DuplicateKeyException nativa en un modelo local al caso del modelo specific que es el Ya adaptado en nuestro ejemplo aquí. luego envolvemos eso de nuevo con un modelo de excepción de categoría genérica que es la ValidationException.

Hay un par de reglas que rigen la construcción de validaciones de dependencia, que son las siguientes:

- Regla 1: Si una validación de dependencia está manejando otra validación de dependencia de un servicio descendente, entonces la excepción interna de la excepción descendente debe ser la misma para la validación de dependencia en el nivel actual.

En otras palabras, si algún `StudentService` está lanzando una `StudentDependencyValidationException` a un servicio ascendente como `StudentProcessingService` - entonces es importante que la `StudentProcessingDependencyValidationException` contenga la misma excepción interna que la `StudentDependencyValidationException`. Esto se debe a que una vez que estas excepciones se asignan a los componentes de exposición, como el controlador de la API o los componentes de la interfaz de usuario, el mensaje de validación original debe propagarse a través del sistema y presentarse al usuario final, independientemente de su origen.

Además, el mantenimiento de la excepción interna original garantiza la capacidad de comunicar diferentes mensajes de error a través de los puntos finales de la API. Por ejemplo, `AlreadyExistsStudentException` puede comunicarse como

o `409` en un nivel de controlador de API - esto difiere de otra excepción de validación de dependencia como `InvalidStudentReferenceException` que se comunicaría como error `FailedDependency` o `424`.

- Regla 2: Si una excepción de validación de dependencia está manejando una excepción de validación que no es de dependencia, debe tomar esa excepción como su excepción interna y no otra.

Estas reglas aseguran que sólo la excepción de validación local es lo que se propaga y no la excepción nativa de un sistema de almacenamiento o una API o cualquier otra dependencia externa.

Que es el caso que tenemos aquí con nuestro `AlreadyExistsStudentException` y es `StudentDependencyValidationException` - la excepción nativa está completamente oculta a la vista, y el mapeo de esa excepción nativa y su mensaje interno es lo que se comunica al usuario final. Esto da a los ingenieros el poder de controlar lo que se comunica desde el otro extremo de su sistema en lugar de dejar que el mensaje nativo (que está sujeto a cambios) se propague a los usuarios finales.

2.1.3.0.5.1 Implementación de validaciones de dependencia

Dependiendo de dónde se origine el error de validación, la implementación de las validaciones de dependencia puede o no contener ninguna lógica de negocio. Como hemos mencionado anteriormente, si el error se origina en el recurso externo (que es el caso aquí) - entonces todo lo que tenemos que hacer es simplemente envolver ese error en una excepción local y luego categorizarlo con una excepción externa bajo la validación de dependencia.

Para asegurar que la mencionada prueba se ha superado, vamos a necesitar unos cuantos modelos.

Por lo `AlreadyExistsStudentException` la aplicación sería como siguiente:

```
public class AlreadyExistsStudentException : Excepción
{
    public AlreadyExistsStudentException(Excepción innerException)
        : base($"Ya existe un alumno con el mismo Id", innerException){ }
}
```

También `StudentDependencyValidationException` que debería necesitamos el ser como sigue:

```
public class EstudianteDependenciaValidaciónExcepción :
Excepción
{
    public StudentDependencyValidationException(Exception innerException)
        : base($"Se ha producido un error de validación de la dependencia del estudiante, inténtelo de nuevo.", innerException){ }
}
```

Ahora, vayamos al lado de la implementación, comenzemos con la lógica de manejo de excepciones:

StudentService.Exceptions.cs

```
private delegate TareaValor< Alumno> DevoluciónFunciónAlumno();

private async ValueTask< Student> TryCatch(ReturningStudentFunction returningStudentFunction)
{
    pruebe con
    {
        return await returningStudentFunction();
    }
    ...
    catch (DuplicateKeyException duplicateKeyException)
    {
        var alreadyExistsStudentException = new AlreadyExistsStudentException(duplicateKeyException);
        throw CreateAndLogDependencyValidationException(alreadyExistsStudentException);
    }
}

...
private StudentDependencyValidationException CreateAndLogDependencyValidationException(Excepción)
{
    var studentDependencyValidationException = new StudentDependencyValidationException(exception); this.loggingBroker.
    LogError(studentDependencyValidationException);

    return studentDependencyValidationException;
}
```

Creamos la excepción interna local en el bloque catch de nuestro proceso de manejo de excepciones para permitir la reutilización de nuestro método de excepción de validación de dependencias para otras situaciones que requieran ese mismo nivel de excepciones externas.

Todo lo demás permanece igual para la referenciación del `tryCatch` método en el `StudentService.cs` file.

2.1.3.2 Cartografía

La segunda responsabilidad de un servicio de fundación es desempeñar el papel de mapeador en ambos sentidos entre los modelos locales y los modelos no locales. Por ejemplo, si estás aprovechando un servicio de correo electrónico que proporciona sus propios SDKs para integrarse, y tus brokers ya están envolviendo y exponiendo las APIs para ese servicio, tu servicio de fundación está obligado a mapear las entradas y salidas de los métodos del broker en modelos locales. La misma situación y más comúnmente entre excepciones nativas no locales como las que mencionamos anteriormente con la situación de validación de dependencia, el mismo aspecto se aplica a sólo errores de dependencia o errores de servicio como discutiremos en breve.

2.1.3.2.0 Modelos no locales

Es muy común que las aplicaciones modernas requieran la integración en algún momento con servicios externos. Estos servicios pueden ser locales a la arquitectura general o al sistema distribuido donde vive la aplicación, o puede ser un proveedor de terceros, como algunos de los servicios de correo electrónico populares, por ejemplo. Los proveedores de servicios externos invierten mucho esfuerzo en el desarrollo de APIs, SDKs y librerías en todos los lenguajes de programación habituales para facilitar a los ingenieros la integración de sus aplicaciones con ese servicio de terceros. Por ejemplo, supongamos que un proveedor de servicios de correo electrónico de terceros ofrece la siguiente API a través de sus SDK:

```
interfaz pública IEmailServiceProvider
{
    ValueTask< EmailMessage> SendEmailAsync(EmailMessage message);
}
```

Consideremos que el modelo `EmailMessage` es un modelo nativo, viene con el SDK del proveedor de servicios de correo electrónico. Sus correidores podrían ofrecer una envoltura alrededor de esta API mediante la construcción de un contrato para abstraer la *funcionalidad*, pero no puede hacer mucho con los modelos nativos que se pasan en o devueltos de estas funcionalidades. Por lo tanto, nuestra interfaz de correidores se vería algo así:

```
interfaz pública IEmailBroker
{
    ValueTask< EmailMessage> SendEmailMessageAsync(EmailMessage message);
}
```

Entonces la implementación sería algo así:

```
public class EmailBroker : IEmailBroker
{
```

```
public async ValueTask< EmailMessage> SendEmailMessageAsync(EmailMessage message) =>
    await this.emailServiceProvider. SendEmailAsync(mensaje);
}
```

Como hemos dicho antes, los corredores han hecho su parte de abstracción alejando la implementación real y las dependencias de los nativos

E

mailServiceProvider de nuestros servicios básicos. Pero eso es sólo el 50% del trabajo, la abstracción no está completa hasta que no haya

pistas del modelo native EmailMessage modelo. Aquí es donde la base

Los servicios de la fundación vienen a hacer una operación de mapeo entre los modelos nativos no locales y los modelos locales de su aplicación. por lo tanto, es muy posible ver una función de mapeo en un servicio de la fundación para abstraer el modelo nativo del resto de sus servicios de la capa de negocio.

Su servicio de fundación entonces será requerido para soportar un nuevo modelo local, llamémoslo Email. la propiedad de su modelo local puede ser idéntica al modelo externo EmailMessage - especialmente en un nivel de tipo de datos primitivo. Pero el nuevo modelo sería el único contrato entre su capa de lógica de negocio pura (servicios de procesamiento, orquestación, coordinación y gestión) y su capa de lógica híbrida como los servicios de la fundación. Aquí hay un fragmento de código para esta operación:

```
public async ValueTask< Email> SendEmailMessageAsync(Email email)
{
    EmailMessage inputEmailMessage = MapToEmailMessage(email);
    EmailMessage sentEmailMessage = await this.emailBroker. SendEmailMessageAsync(inputEmailMessage);

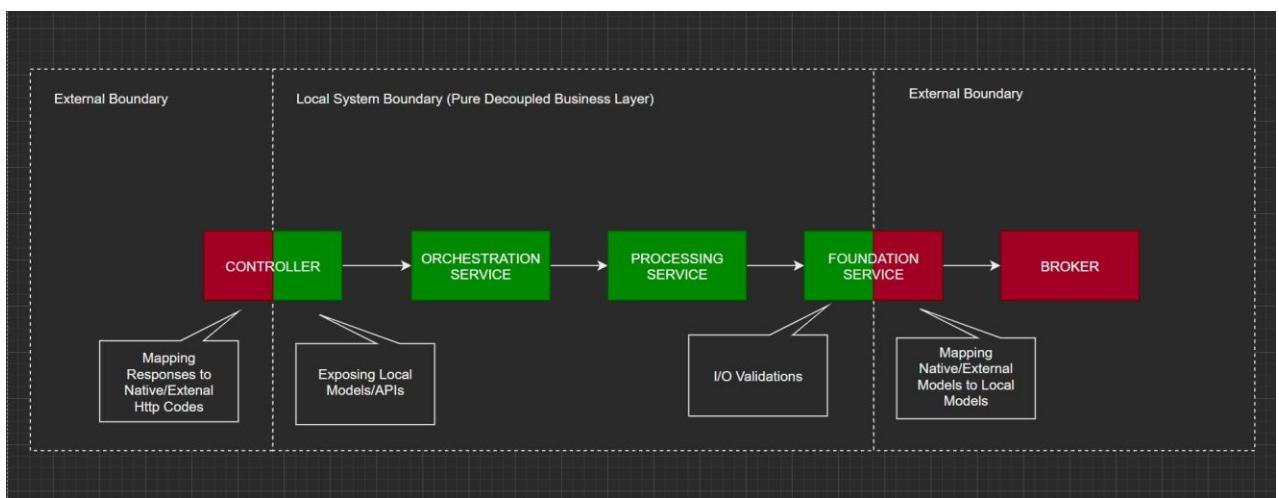
    return MapToEmail(sentEmailMessage);
}
```

Dependiendo de si el mensaje devuelto tiene un estado o quieres devolver el mensaje de entrada como señal de una operación exitosa, ambas prácticas son válidas en mi Norma. Todo depende de lo que tiene más sentido para la operación que está tratando de ejecutar. el fragmento de código anterior es un escenario ideal en el que su código tratará de permanecer fiel al valor pasado en, así como el valor devuelto con todo el mapeo necesario incluido.

2.1.3.2.1 Asignación de excepciones

Al igual que los modelos no locales, las excepciones que se producen API externa como los modelos de EntityFramework `UpdateException` o cualquier otras tienen que ser mapeadas en modelos de excepción locales. El manejo de estas excepciones no locales antes de entrar en los componentes de la capa de negocio puro evitará cualquier posible acoplamiento estrecho o dependencia de cualquier modelo externo. ya que puede ser muy común, que las excepciones pueden ser manejadas differentemente basado en el tipo de excepción y cómo queremos tratar con él internamente en el sistema.

Por ejemplo, si tratamos de manejar un `UserNotFoundException` ser lanzada desde el uso de Microsoft Graph, por ejemplo, podríamos no querer necesariamente salir de todo el procedimiento. podríamos querer continuar añadiendo un usuario en algún otro almacenamiento para el futuro procesamiento de presentación de Graph. Las APIs externas no deben influir en si su operación interna debe detenerse o no. y por lo tanto el manejo de excepciones en la capa Foundation es la garantía de que esta influencia se limita dentro de las fronteras de nuestra área de manejo de recursos externos de nuestra aplicación y no tiene ningún impacto en nuestros procesos de negocio principales. La siguiente ilustración debería dibujar la imagen un poco más clara desde esa perspectiva:



Estos son algunos escenarios comunes para mapear excepciones locales nativas o internas a excepciones externas:

Excepción	Envolver la excepción interior con	Envolver con	Nivel de regis tro
NullStudentException	-	StudentValidationException	Error
InvalidStudentException	-	StudentValidationException	Error
SqlException	FailedStudentStorageException	StudentDependencyException	Crítica
HttpResponseUrlNotFoundException	FailedStudentApiException	StudentDependencyException	Crítica
HttpResponseUnauthorizedException	FailedStudentApiException	StudentDependencyException	Crítica
NotFoundStudentException	-	StudentValidationException	Error
HttpResponseNotFoundException	NotFoundStudentException	StudentDependencyValidationException	Error
DuplicateKeyException	AlreadyExistsStudentException	StudentDependencyValidationException	Error
HttpResponseConflictException	AlreadyExistsStudentException	StudentDependencyValidationException	Error
ForeignKeyConstraintConflictException	InvalidStudentReferenceException	StudentDependencyValidationException	Error
DbUpdateConcurrencyException	LockedStudentException	StudentDependencyValidationException	Error
DbUpdateException	FailedStudentStorageException	StudentDependencyException	Error
HttpResponseException	FailedStudentApiException	StudentDependencyException	Error
Excepción	FailedStudentServiceException	StudentServiceException	Error

[*] Estandarización de validaciones y excepciones

[*] Validaciones de prueba sin ruptura de circuito

2.2 Servicios de Procesamiento (Lógica Empresarial de Orden Superior)

2.2.0 Introducción

Los servicios de procesamiento son la capa en la que se implementa un orden superior de lógica empresarial. Pueden combinar (u orquestar) dos funciones de nivel primitivo de su correspondiente servicio base para introducir una funcionalidad más novedosa. También pueden llamar a una función primitiva y cambiar el resultado con un poco de lógica empresarial añadida. Y, a veces, los servicios de procesamiento están ahí como paso para introducir equilibrio en la arquitectura general.

Los servicios de procesamiento son opcionales, dependiendo de su necesidad de negocio - en una simple API de operaciones CRUD, los servicios de procesamiento y todas las demás categorías de servicios más allá de ese punto dejarán de existir ya que no hay necesidad de un orden superior de lógica de negocio en ese punto.

Este es un ejemplo de cómo sería una función de servicio de procesamiento:

```
public ValueTask< Student> UpsertStudentAsync(Student student) =>
TryCatch(async () =>
{
    ValidarEstudiante(estudiante);

    IQueryable< Student> allStudents = this.studentService.
        RetrieveAllStudents();

    bool alumnoExiste = allStudents. Any(recuperadoEstudiante =>
        recuperadoEstudiante.Id == estudiante.Id);

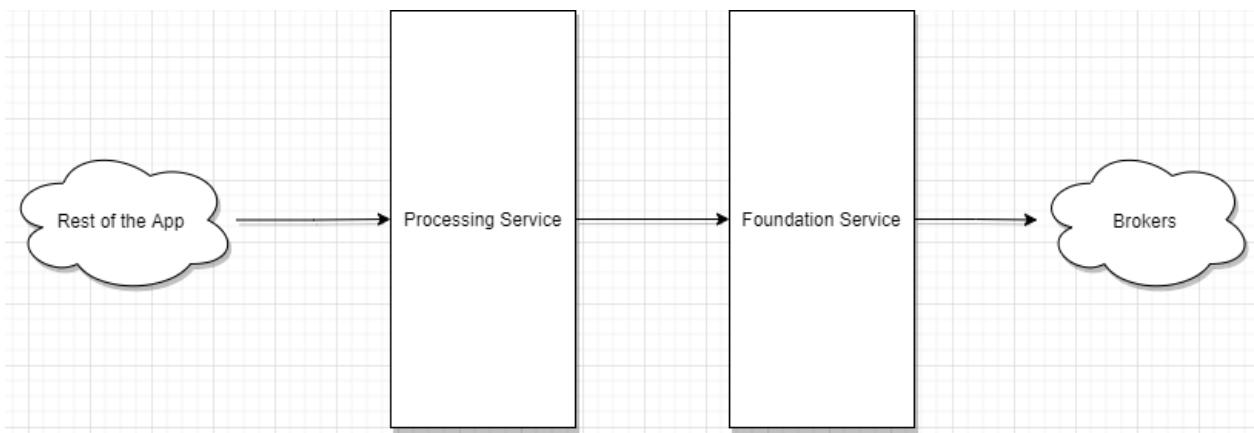
    return estudianteExiste switch {
        false => await this.studentService. RegisterStudentAsync(student),
        _ => await this.studentService. ModifyStudentAsync(student.Id)
    };
});
```

Los servicios de procesamiento hacen que los servicios de la Fundación no sean más que una capa de validación sobre las operaciones primitivas existentes. Lo que significa que las funciones de los servicios de procesamiento están más allá de las primitivas, y sólo se ocupan de los modelos locales, como discutiremos en las próximas secciones.

2.2.1 En el mapa

Cuando se utilizan, los servicios de procesamiento viven entre los servicios de la fundación y el resto de la aplicación. no pueden llamar a los brokers de entidad o negocio, pero pueden llamar a los brokers de utilidad como los brokers de registro, los brokers de tiempo y cualquier otro broker que offer funcionalidad de apoyo y no specific a ninguna lógica de negocio en particular. Aquí hay una visualización de dónde se encuentran los servicios

de procesamiento en el mapa de nuestra arquitectura:



En el lado derecho de un servicio de Procesamiento se encuentran todos los modelos y la funcionalidad no locales, ya sea a través de los corredores, o los modelos que el servicio de la fundación está tratando de mapear en modelos locales. En el lado izquierdo de los servicios de Procesamiento se encuentra la funcionalidad, los modelos y la arquitectura locales puros. A partir de los propios servicios de Procesamiento, no debería haber ningún rastro o pista de ningún modelo nativo o no local en el sistema.

2.2.2 Características

Los servicios de procesamiento en general son combinadores de múltiples funciones de nivel primitivo para producir una lógica de negocio de orden superior. pero tienen muchas más características que eso, hablemos de ellas aquí.

2.2.2.0 Idioma

El lenguaje utilizado en los servicios de procesamiento define el nivel de complejidad y las capacidades que ofrece. Normalmente, los servicios de procesamiento combinan dos o más operaciones primitivas de la capa base para crear un nuevo valor.

2.2.2.0.0 Funciones Idioma

De un vistazo, el lenguaje de los servicios de procesamiento cambia de las operaciones primitivas

como AddStudent o EliminarAlumn a EnsureStudentExists o Ups suelen offer unas operaciones de lógica empresarial más avanzadas para soportar una funcionalidad de orden superior. Aquí hay algunos ejemplos de las combinaciones más comunes que un servicio de procesamiento puede offer:

Operación de procesamiento	Funciones primitivas
EnsureStudentExistsAsync	RetrieveAllStudents + AddStudentAsync
UpsertStudentAsync	RetrieveStudentById + AddStudentAsync + ModifyStudentAsync
VerifyStudentExists	RecuperarTodosLosEstudiantes
TryRemoveStudentAsync	RetrieveStudentById + RemoveStudentByIdAsync

Como se puede ver, la combinación de funciones primitivas que hacen los servicios de procesamiento también podría incluir la adición de una capa adicional de lógica en la parte superior de las existentes operación primitiva. Por ejemplo, ~~la función primitiva~~ se aprovecha de la función primitiva ~~LosEstudiantes~~ función primitiva, y luego añade una lógica booleana para ~~verificar~~ que el estudiante devuelto por ~~y el Id~~ de una consulta realmente existe o no antes de devolver un booleano.

2.2.2.0.1 Pasar a través de

Los servicios de procesamiento pueden tomar prestada parte de la terminología que puede utilizar un servicio de fundación. por ejemplo, en un escenario de paso, un servicio de procesamiento puede ser tan simple como `AddStudentAsync`. Discutiremos los escenarios de equilibrio de la arquitectura más adelante en este capítulo. A diferencia de los servicios Foundation, los servicios de procesamiento deben tener el identifier `Processing` en sus nombres. por ejemplo, decimos `StudentProcessingService`.

2.2.2.0.2 Lenguaje a nivel de clase

Lo que es más importante, los servicios de procesamiento deben incluir el nombre de la entidad que es apoyada por su correspondiente servicio de la Fundación. Por ejemplo, si un servicio de procesamiento depende de un servicio de profesor, el nombre del servicio de procesamiento debe ser `TeacherProcessingService`.

2.2.2.1 Dependencias

Los servicios de procesamiento sólo pueden tener dos tipos de dependencias: un servicio Foundation correspondiente o un broker de utilidades. Esto se debe simplemente a que los servicios de procesamiento no son más que un nivel extra de lógica empresarial de orden superior, orquestado por operaciones primitivas combinadas en el nivel de la Fundación. Los servicios de tramitación también pueden utilizar intermediarios de servicios públicos como `gingBroker` para apoyar su aspecto de información. pero nunca deberá interactuar con un corredor de Entidades o Negocios.

2.2.2.2 Una fundación

Los servicios de Processing pueden interactuar con uno y sólo un servicio de Foundation. De hecho, sin un servicio de la Fundación nunca puede haber una capa de Procesamiento. y al igual que mencionamos anteriormente sobre el lenguaje y la nomenclatura, los servicios de Procesamiento adoptan exactamente el mismo nombre de entidad que su dependencia de la Fundación. Por ejemplo, un servicio de procesamiento que maneja la lógica de negocio de orden superior para los estudiantes se comunicará con nada más que su capa de fundación, que

sería por ejemplo. Eso significa que los servicios de procesamiento

tendrá uno y sólo un servicio como dependencia en su construcción o iniciación como sigue:

```
Public Class StudentProcessingService
{
    private readonly IStudentService studentService;

    public StudentProcessingService(IStudentService studentService) =>
        this.studentService = studentService;
}
```

Sin embargo, los servicios de procesamiento pueden requerir dependencias de múltiples servicios públicos

corredores como `DateBroker` o `LoggingBroker` ... etc.

2.2.2.3 Validaciones de datos usados

A diferencia de los servicios de la capa Foundation, los servicios Processing sólo validan lo que necesitan de su entrada. Por ejemplo, si un servicio de procesamiento debe validar la existencia de una entidad de estudiante, y su modelo de entrada resulta ser un

entidad completa del `Estudiante`, sólo validará que la entidad no es `null` y que

el `Id` de esa entidad es válido. el resto de la entidad está fuera de la preocupación del servicio de Procesamiento. Los servicios de procesamiento delegan las validaciones completas a la capa de servicios que se ocupa de eso que es la capa de la Fundación. he aquí un ejemplo:

```
public ValueTask< Student> UpsertStudentAsync(Student student) =>
TryCatch(async () =>
{
    ValidarEstudiante(estudiante);

    IQueryable< Student> allStudents = this.studentService.
        RetrieveAllStudents();

    bool isStudentExists = allStudents. Any(retrievedStudent =>
        retrievedStudent.Id == student.Id);

    return isStudentExists switch {
        false => await this.studentService. RegisterStudentAsync(student),
        _ => await this.studentService. ModifyStudentAsync(student.Id)
    };
});
```

Los servicios de procesamiento tampoco se preocupan mucho por las validaciones salientes, excepto por lo que va a utilizar dentro de la misma rutina. Por ejemplo, si un servicio de procesamiento está recuperando un modelo, y va a utilizar este modelo para pasarlo a otra función de nivel primitivo en la capa de la Fundación, el servicio de procesamiento deberá validar que el modelo recuperado es válido dependiendo de los atributos del modelo que utilice. Sin embargo, para los escenarios de Pass-through, los servicios de procesamiento delegarán la validación de salida a la capa Foundation.

2.2.3 Responsabilidades

La principal responsabilidad de los servicios de procesamiento es proporcionar una lógica de negocio de orden superior. Esto sucede junto con el mapeo regular de firmas y varias validaciones de uso que discutiremos en detalle en esta sección.

2.2.3.0 Lógica de orden superior

La lógica empresarial de orden superior son funciones que están por encima de las primitivas. Para

instancia, `AddStudentAsync` es una función primitiva que hace una cosa y sólo una cosa. Pero la lógica de orden superior es cuando tratamos de proporcionar una función que cambie el resultado de una sola función primitiva como `StudentExists` que devuelve un valor booleano en lugar de todo el objeto del Estudiante, o una combinación de múltiples funciones primitivas como `reStudentExistsAsync` que es una función que sólo Stud añadirá un determinado ent si y sólo si dicho objeto no existe ya en el almacén. he aquí algunos ejemplos:

2.2.3.0.0 Cambiadores

El patrón shifter en una lógica de negocio de orden superior es cuando el resultado de una función primitiva particular se cambia de un valor a otro.

Idealmente un tipo primitivo como un `bool` o `int` no un tipo completamente diferente como el que

violaría el principio de pureza. Por ejemplo, en un patrón de desplazamiento, queremos verificar si un alumno existe o no. En realidad no queremos el objeto completo, sino sólo si existe o no en un sistema concreto. Ahora bien, este parece un caso en el que sólo necesitamos interactuar con uno y sólo un servicio de la fundación y estamos cambiando el valor del resultado a otra cosa. Que debería fit perfectamente en el ámbito de los servicios de procesamiento. Aquí hay un ejemplo:

```

public ValueTask< bool> VerifyStudentExists (Guid studentId) =>
TryCatch(async () =>
{
    IQueryable< Student> allStudents = this.studentService.
        RetrieveAllStudents();

    ValidarEstudiantes(todosEstudiantes);

    return allStudents. Any(student => student.Id == studentId);
});

```

En el fragmento anterior, proporcionamos una lógica de negocio de orden superior, devolviendo

un valor booleano de si un estudiante particular con un determinado existe en el

sistema o no. Hay casos en los que su capa de orquestación de servicios no se preocupa realmente de todos los detalles de una entidad concreta, sino simplemente de saber si existe o no como parte de una lógica de negocio superior o lo que llamamos orquestación.

Este es otro ejemplo popular para el patrón de cambio de los servicios de procesamiento:

```

public int RecuperarCuentoDeEstudiantes()
=> TryCatch(() =>
{
    IQueryable< Student> allStudents = this.studentService.
        RetrieveAllStudents();

    ValidarEstudiantes(todosEstudiantes);

    return allStudents. Count();
});

```

En el ejemplo anterior, proporcionamos una función para recuperar el recuento de todos los estudiantes en un sistema determinado. Son los diseñadores del sistema los que deben determinar

si se debe interpretar un valor recuperado para todos los estudiantes como un

caso de excepción que no se esperaba que ocurriera o devolver un 0; todo depende de cómo gestionen el resultado. En nuestro caso, validamos los datos salientes tanto como los entrantes, especialmente si se van a utilizar dentro de la función de procesamiento para asegurar que no se produzcan más fallos en los servicios ascendentes.

2.2.3.0.1 Combinaciones

La combinación de múltiples funciones primitivas de la capa base para conseguir una lógica de negocio de orden superior es una de las principales responsabilidades de un servicio de procesamiento. Como mencionamos anteriormente, algunos de los ejemplos más populares es para asegurar que exista un modelo de estudiante particular como el siguiente:

```

public async ValueTask< Student> EnsureStudentExistsAsync(Student student) =>
TryCatch(async () =>
{
    ValidarEstudiante(estudiante);

    IQueryable< Student> allStudents = this.studentService.
        RetrieveAllStudents();

    Estudiante tal vezEstudiante = allStudents. FirstOrDefault(retrievedStudent =>
        retrievedStudent.Id == student.Id);

    return maybeStudent switch
    {
        {} => quizásEstudiante,
        _ => await this.studentService. AddStudentAsync(student)
    };
});

```

En el fragmento de código anterior, combinamos `RetrieveAll` con `AddAsync` para lograr una operación de lógica de negocio de orden superior. La operación `EnsureAsync` que necesita verificar que algo o entidad existe primero antes de intentar persistirlo. La terminología alrededor de estas rutinas de lógica de negocio de orden superior es muy importante. Su importancia radica principalmente en el control de las expectativas del resultado y la funcionalidad interna. Pero también asegura que se requieran menos recursos cognitivos de los ingenieros para entender las capacidades subyacentes de una rutina particular. El lenguaje convencional utilizado en todos estos servicios también garantiza que no se creen capacidades redundantes por error. Por ejemplo, un equipo de ingenieros sin ninguna forma de estándar podría crear `TryAddStudentAsync` mientras ya tiene una funcionalidad existente como `EnsureStudentExistsAsync` que hace exactamente lo mismo. La convención aquí con la limitación del tamaño de las capacidades que un servicio particular puede tener asegura que el trabajo redundante nunca se producirá en cualquier ocasión. Hay muchas diferentes instancias de combinaciones que pueden producir una lógica de negocio de orden superior, por ejemplo podemos necesitar implementar una funcionalidad que asegure que un estudiante es eliminado. Usamos `EnsureS`

`tudentRemovedByIdAsync` para combinar `RetrieveById` y un `RemoveById` en la misma rutina. Todo depende del nivel de capacidades que pueda necesitar un servicio ascendente para implementar dicha funcionalidad.

2.2.3.1 Mapeo de firmas

Aunque los servicios de procesamiento operan completamente en modelos locales y contratos locales, todavía se requiere que mapeen los modelos de los servicios de nivel de fundación a sus propios modelos locales. Por ejemplo, si un servicio de la fundación lanza una `dentValidationException`, los servicios de procesamiento asignarán esa excepción a `StudentProcessingDependencyValidationException`. Vamos a hablar de la asignación en esta sección.

2.2.3.1.0 Modelos locales sin excepción

En general, los servicios de procesamiento deben asignar cualquier objeto entrante o saliente con un modelo específico al suyo propio. Pero esa regla no siempre se aplica a los modelos que no son de excepción. Por ejemplo, si un servicio funciona en base a un modelo, y no hay necesidad de un para este servicio, entonces se puede permitir que el servicio de procesamiento utilice exactamente el mismo modelo de la capa base.

2.2.3.1.1 Modelos de excepción

Cuando se trata de procesar servicios que manejan excepciones de la capa de base, es importante entender que las excepciones en nuestro Estándar son más expresivas en sus convenciones de nomenclatura y su papel más que cualquier otro modelo. Las excepciones aquí definen el qué, dónde y por qué cada vez que se lanzan. Por ejemplo, una excepción que se llama StudentProcessingServiceException indica la entidad de la excepción que es la entidad Student. Luego indica la ubicación de la excepción que es el StudentProcessingService. Y por último indica la razón de esa excepción que es ServiceException indicando un error interno al servicio que no es una validación o una dependencia de naturaleza que ocurrió. Al igual que la capa base, los servicios de procesamiento harán el siguiente mapeo para que ocurran las excepciones de sus dependencias:

Excepción	Envolver la excepción interior con	Envolver con	Nivel de registro
StudentDependencyValidationException	Cualquier excepción interna	StudentProcessingDependencyValidationException	Error
StudentValidationException	Cualquier excepción interna	StudentProcessingDependencyValidationException	Error
StudentDependencyException	-	StudentProcessingDependencyException	Error
StudentServiceException	-	StudentProcessingDependencyException	Error
Excepción	-	StudentProcessingServiceException	Error

[*] Servicios de procesamiento en acción (Parte 1)

[*] Servicios de procesamiento en acción (Parte 2)

[*] Servicios de procesamiento en acción (Parte 3)

[*] Servicios de procesamiento en acción (Parte 4)

2.3 Servicios de orquestación (lógica compleja de orden superior)

2.3.0 Introducción

Los servicios de orquestación son los combinadores entre múltiples servicios de fundación o procesamiento para realizar una operación lógica compleja. La principal responsabilidad de los servicios de orquestación es realizar una operación lógica de varias entidades y delegar las dependencias de dicha operación a los servicios de procesamiento o de base posteriores. La principal responsabilidad de los servicios de orquestación es encapsular las operaciones que requieren dos o tres entidades de negocio.

```
public async ValueTask< LibraryCard> CreateStudentLibraryCardAsync(LibraryCard libraryCard) =>
TryCatch(async () =>
{
    ValidarTarjetaDeBiblioteca(TarjetaDeBiblioteca);

    await this.studentProcessingService
        .VerifyEnrolledStudentExistsAsync(libraryCard.StudentId);

    return await this.libraryCardProcessingService. CreateLibraryCardAsync(libraryCard);
});
```

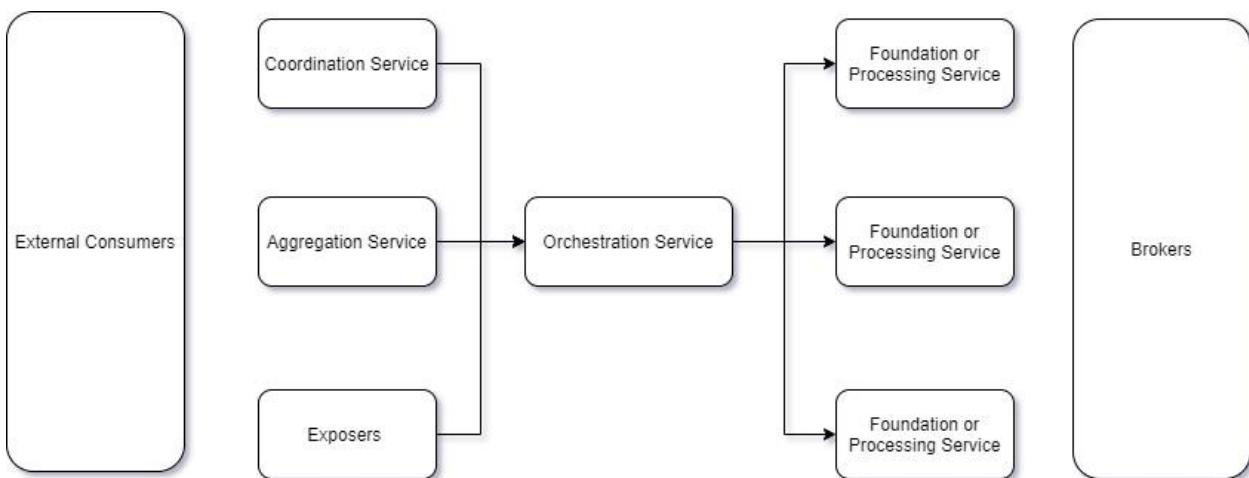
En el ejemplo anterior, el `LibraryCardOrchestrationService` llama tanto al `StudentProcessingService` como al `LibraryCardProcessingService` para realizar una operación compleja. En primer lugar, verifica que el estudiante para el que estamos creando un carné de biblioteca existe y que está matriculado. Luego, crear el carné de biblioteca.

La operación de creación de un carné de biblioteca para un alumno determinado no puede realizarse simplemente llamando al servicio de carnés de biblioteca. Esto se debe a que el servicio de carné de biblioteca (procesamiento o fundación) no tiene acceso a todos los detalles sobre el estudiante. Por lo tanto, aquí es necesario implementar una lógica de combinación para asegurar un flow adecuado.

Es importante entender que los servicios de orquestación sólo son necesarios si, y sólo si, necesitamos combinar operaciones de múltiples entidades primitivas o de orden superior. En algunas arquitecturas, los servicios de orquestación podrían ni siquiera existir. Eso es simplemente porque algunos microservicios podrían ser simplemente responsables de aplicar la lógica de validación y persistir y recuperar los datos del almacenamiento, ni más ni menos.

2.3.1 En el mapa

Los servicios de orquestación son uno de los componentes de lógica de negocio más importantes de cualquier sistema. Se sitúan entre los servicios de entidad única (como el procesamiento o la fundación) y los servicios de lógica avanzada, como los servicios de coordinación, los servicios de agregación o simplemente los expositores, como los controladores, los componentes web o cualquier otro. He aquí una visión general de alto nivel de dónde pueden vivir los servicios de orquestación:



Como se ha mostrado anteriormente, los servicios de orquestación tienen bastantes dependencias y consumidores. Son el motor central de cualquier software. En la parte derecha, puedes ver las dependencias que puede tener un servicio de orquestación. Dado que un servicio de procesamiento es opcional en función de si se necesita o no una lógica de negocio de orden superior, los servicios de orquestación pueden combinar también varios servicios de base.

La existencia de un servicio de orquestación garantiza la existencia de un servicio de procesamiento. Pero no siempre es así. Hay situaciones en las que todos los servicios de orquestación necesitan para finalizar un negocio flow es interactuar con la funcionalidad de nivel primitivo.

Sin embargo, desde el punto de vista del consumidor, el servicio de orquestación podría tener varios consumidores. Estos consumidores pueden ser desde servicios de coordinación (orquestadores de orquestadores) o servicios de agregación o simplemente un expositor. Los expositores son como los controladores, el servicio de vista, los componentes de la interfaz de usuario o simplemente otra base o servicio de procesamiento en el caso de

poner los mensajes en una cola - que discutiremos más adelante en nuestro Estándar.

2.3.2 Características

En general, los servicios de orquestación se ocupan de combinar operaciones primitivas de entidad única o de lógica de negocio de orden superior para ejecutar un flujo exitoso. Pero también se puede pensar en ellos como el pegamento que une múltiples operaciones de entidad única.

2.3.2.0 Idioma

Al igual que los servicios de procesamiento, el lenguaje utilizado en los servicios de orquestación define el nivel de complejidad y las capacidades que ofrece. Normalmente, los servicios de orquestación combinan dos o más operaciones primitivas o de orden superior de varios servicios de entidad única para ejecutar una operación con éxito.

2.3.2.0.0 Funciones Idioma

Los servicios de orquestación tienen una característica muy común cuando se trata del lenguaje de sus funciones. Los servicios de orquestación son holísticos en la mayor parte del lenguaje de sus funciones, verás funciones como

donde el servicio extrae todos los usuarios con tipo de administrador y luego llama a un servicio de notificación para notificar a todos y cada uno de ellos.

Resulta muy obvio que los servicios de orquestación ofrecen una funcionalidad que se acerca cada vez más a un lenguaje empresarial que a una operación técnica perimida. Se puede ver una expresión casi idéntica en un requisito empresarial no técnico que coincide uno por uno con un nombre de función en un servicio de orquestación. El mismo patrón continúa a medida que se pasa a categorías más altas y avanzadas de servicios dentro de ese ámbito de la lógica empresarial.

2.3.2.0.1 Pasar a través de

Los servicios de orquestación también pueden ser un paso para algunas operaciones. Para

ejemplo, un servicio de orquestación podría permitir que una operación sea propagada a través del servicio para unificar el origen de las interacciones con el sistema a nivel de los expositores. En este caso, los servicios de orquestación utilizarán la misma terminología que puede utilizar un servicio de procesamiento o una fundación para propagar la operación.

2.3.2.0.2 Lenguaje a nivel de clase

Los servicios de orquestación combinan principalmente múltiples operaciones para apoyar un entidad particular. Así, si la entidad principal de apoyo es `Estudiante` y el resto de las entidades son sólo para apoyar una operación dirigida principalmente a un Estudiante entonces el nombre del servicio de orquestación sería `StudentOrchestrationService`.

Este nivel de aplicación de las convenciones de nomenclatura garantiza que cualquier servicio de orquestación se mantenga centrado en la responsabilidad de una sola entidad con respecto a otras múltiples entidades de apoyo.

Por ejemplo, si la creación de un carné de biblioteca requiere asegurar que el estudiante al que se hace referencia en ese carné de biblioteca debe estar matriculado en una escuela. Entonces un servicio de orquestación tendrá el nombre de su entidad principal, que en este caso es el carnet de biblioteca. El nombre de nuestro servicio de orquestación sería entonces `LibraryCardOrchestrationService`.

Lo contrario también es cierto. Si la inscripción de un estudiante en una escuela tiene una operación de acompañamiento como la creación de una tarjeta de biblioteca, entonces en este caso se debe crear un `StudentOrchestrationService` con el propósito principal de crear un estudiante y luego todas las demás entidades relacionadas una vez que lo anterior tiene éxito.

La misma idea se aplica a todas las excepciones creadas en un servicio de orquestación

como `StudentOrchestrationValidationException` y `StudentOrchestrationDependencyException`, etc.

2.3.2.1 Dependencias

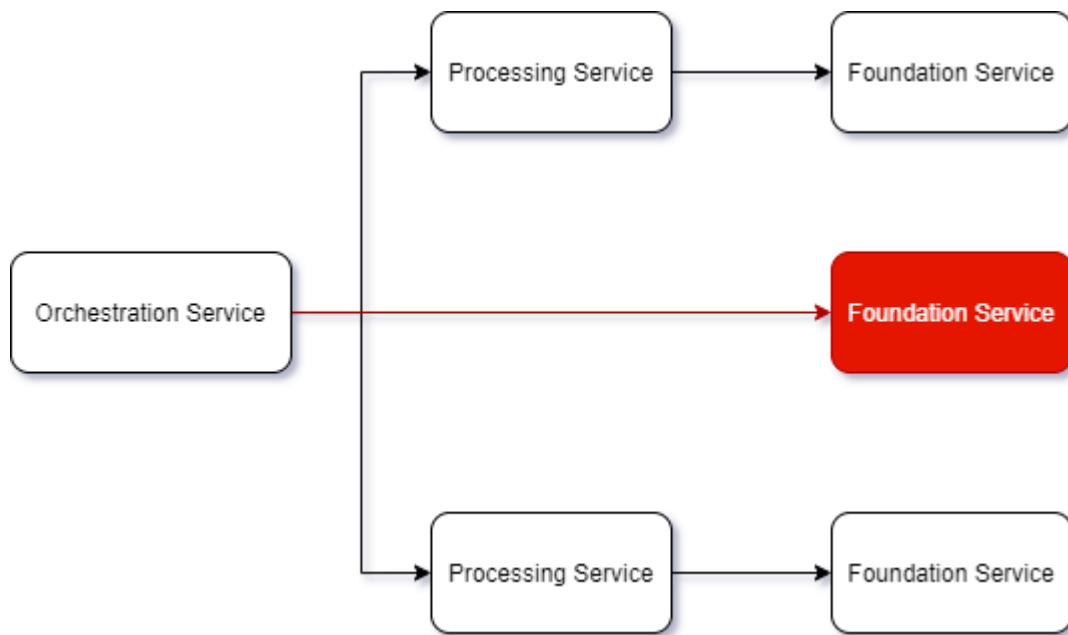
Como hemos mencionado anteriormente, los servicios de orquestación pueden tener una gama un poco más amplia de tipos de dependencias a diferencia de los servicios de procesamiento y fundación. Esto se debe únicamente a la opcionalidad de los servicios de procesamiento. Por lo tanto, los servicios de orquestación pueden tener dependencias que van desde los servicios de fundación o los servicios de procesamiento y, opcionalmente y por lo general, el registro o cualquier otro corredor de utilidad.

2.3.2.1.0 Equilibrio de la dependencia (patrón Florance)

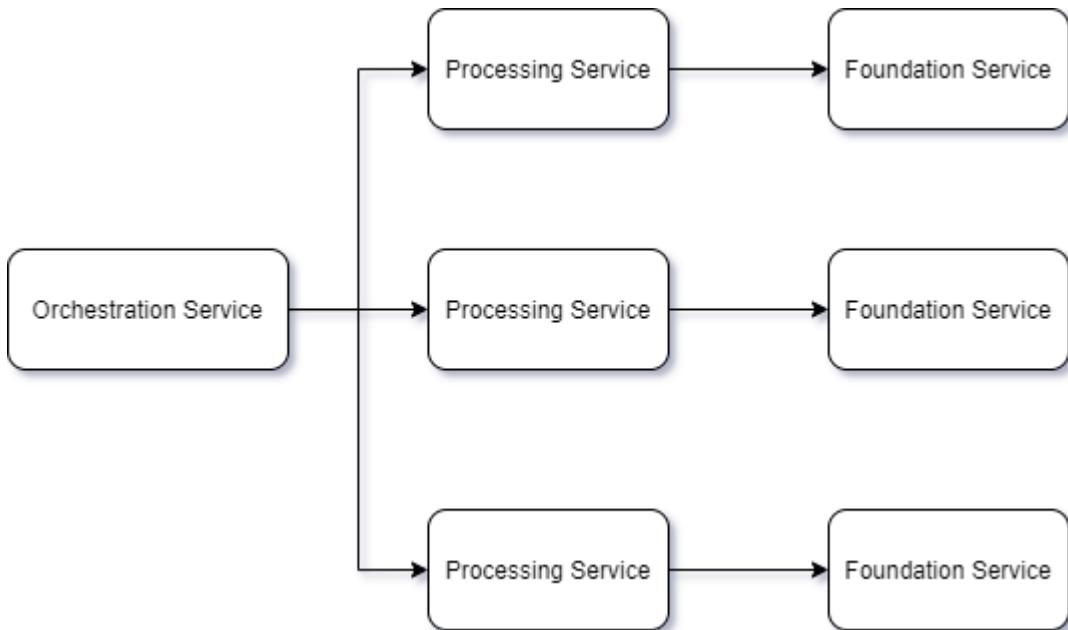
Hay una regla muy importante que rige la coherencia y el equilibrio de los servicios de orquestación, que es el "Patrón Florance". La regla dicta que cualquier servicio de orquestación no puede combinar dependencias de diferentes categorías de operación.

Lo que esto significa es que un servicio de orquestación no puede tener una fundación y un servicio de procesamiento combinados. Las dependencias tienen que ser o bien todos los procesamientos o bien todas las fundaciones. Sin embargo, esa regla no se aplica a las dependencias de los agentes de servicios públicos.

Este es un ejemplo de un desequilibrio en las dependencias del servicio de orquestación:



Se requiere un servicio de procesamiento adicional para dar un paso a un servicio de base de nivel inferior para equilibrar la arquitectura - aplicando el 'Patrón Florance' para la simetría convertiría nuestra arquitectura en la siguiente:



La aplicación del "patrón Florance" puede ser muy costosa al principio, ya que incluye la creación de un servicio de procesamiento completamente nuevo (o varios) sólo para equilibrar la arquitectura. Sin embargo, sus beneficios compensan el esfuerzo realizado desde el punto de vista de la mantenibilidad, la legibilidad y la conectividad.

2.3.2.1.1 Dos-tres

La regla "Dos-Tres" es una regla de control de la complejidad. Esta regla dicta que un servicio de orquestación no puede tener más de tres ni menos de dos servicios de procesamiento o fundación para ejecutar la orquestación. Esta regla, sin embargo, no se aplica a los servicios de orquestación. Y el servicio de orquestación puede tener a una o más interacciones con otros servicios sin ningún problema. Pero un servicio de orquestación puede no tener un corredor de entidad StorageBroker ni un QueueBroker que alimenta directamente la capa de negocio principal de cualquier servicio.

La regla 'Dos-Tres' puede requerir una capa de normalización a la función de negocio categórica que se requiere cumplir. Hablemos de los diferentes mecanismos de normalización de los servicios de orquestación.

2.3.2.1.1.0 Normalización completa

A menudo, hay situaciones en las que la arquitectura actual de cualquier servicio de orquestación termina con un servicio de orquestación que tiene tres dependencias. Y se requiere un nuevo servicio de procesamiento de

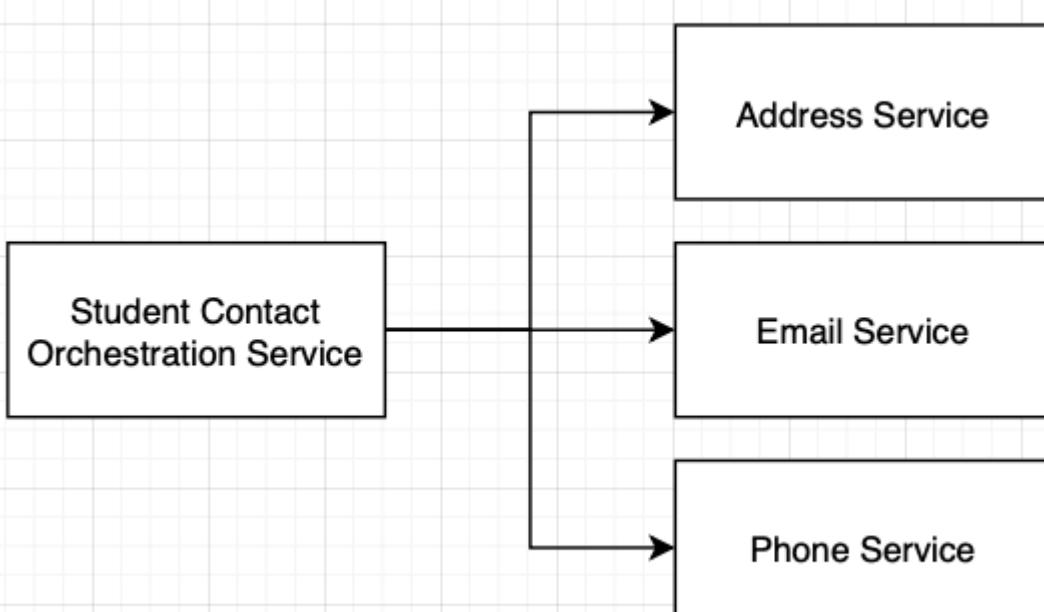
entidades o de fundación para completar un proceso existente.

Por ejemplo, digamos que tenemos `StudentContactOrchestrationService` y ese servicio tiene dependencias que proporcionan funcionalidad a nivel primitivo

Envíe Teléfono para cada estudiante. Esta es una visualización de

para la dirección, correo electrónico o teléfono a

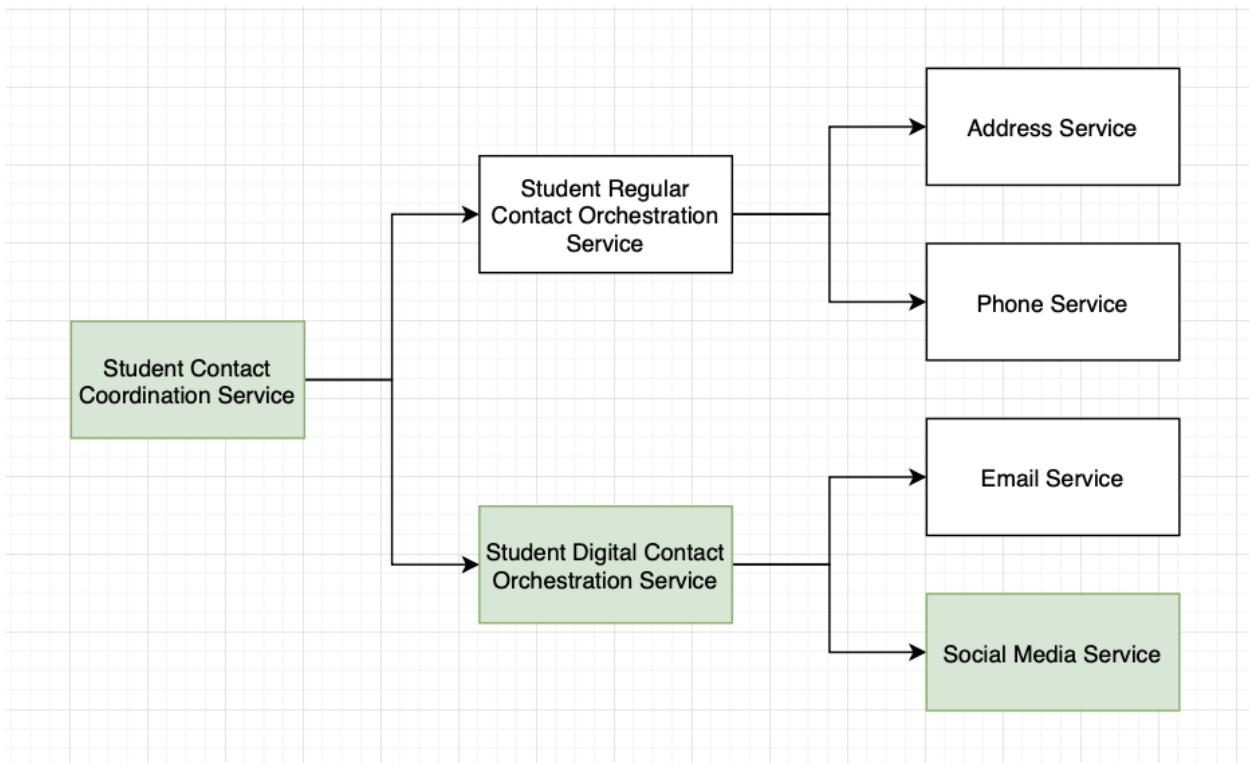
ese estado:



Ahora, un nuevo requisito viene a añadir un estudiante `SocialMedia` para reunir más información de contacto sobre cómo llegar a un determinado estudiante. Podemos pasar al modo de normalización completa simplemente encontrando el punto común que divide por igual las entidades de información de contacto. Por ejemplo, podemos dividir la información de contacto en `PhysicalAddress` y `DigitalAddress`. De este modo, dividimos cuatro dependencias en dos, cada una para sus propios servicios de orquestación, como sigue:

Envíe Teléfono para cada estudiante. Esta es una visualización de

para la dirección, correo electrónico o teléfono a



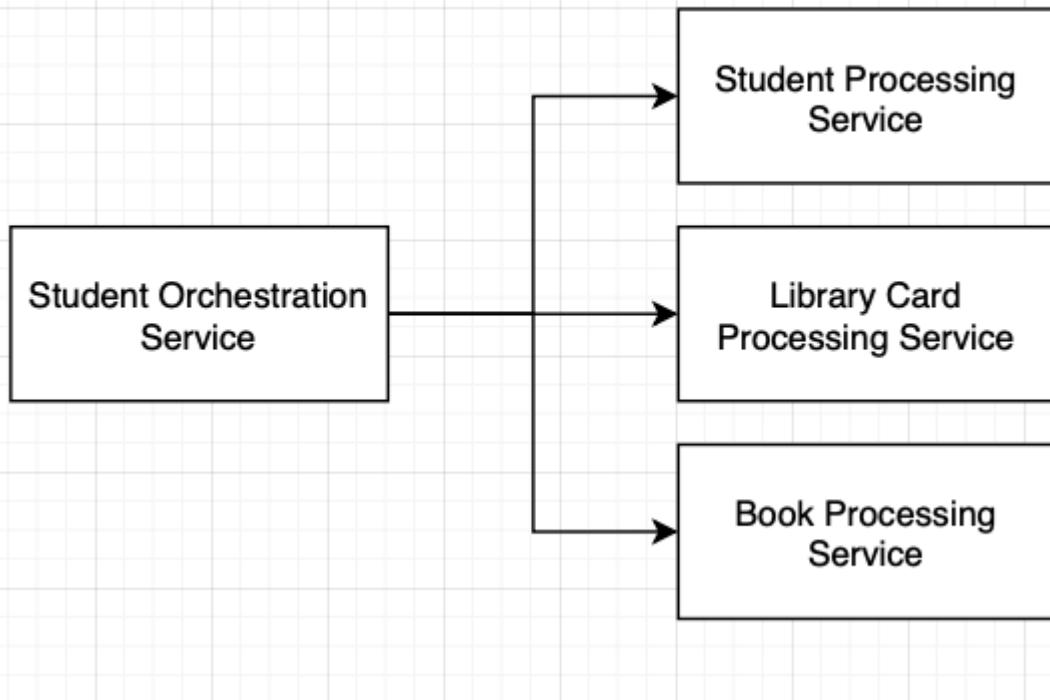
Como puede ver en la figura anterior, hemos modificado el `StudentContactOrchestrationService` existente en `StudentRegularContactOrchestrationService`, y luego hemos eliminado una de sus dependencias en el `EmailService`.

Además, creamos un nuevo `StudentDigitalContactOrchestrationService` para tener dos dependencias del `EmailService` existente además del nuevo `SocialMediaService`. Pero ahora que la normalización ha terminado, necesitamos una capa de lógica de negocio de orden avanzado, como un servicio de coordinación para proporcionar información de contacto de los estudiantes a los consumidores anteriores.

2.3.2.1.1.1 Seminormalización

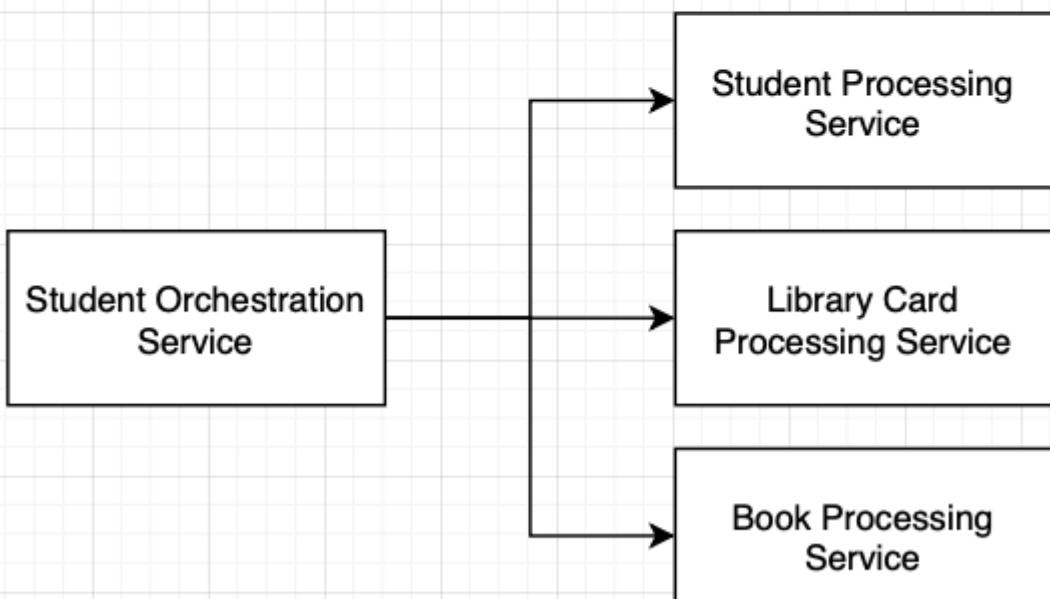
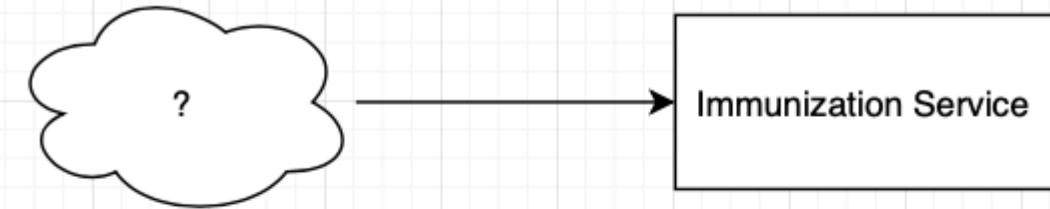
La normalización no siempre es tan sencilla como el ejemplo anterior. Especialmente en situaciones en las que debe existir una entidad principal antes de crear o escribir información adicional hacia entidades relacionadas con esa misma entidad.

Por ejemplo, digamos que tenemos un servicio que `StudentRegistrationOrchestrationService` que depende de `StudentProcessingService`, `LibraryCardProcessingService` y `BookProcessingService` de la siguiente manera:



Pero ahora, necesitamos un nuevo servicio para manejar los registros de inmunización de los estudiantes como

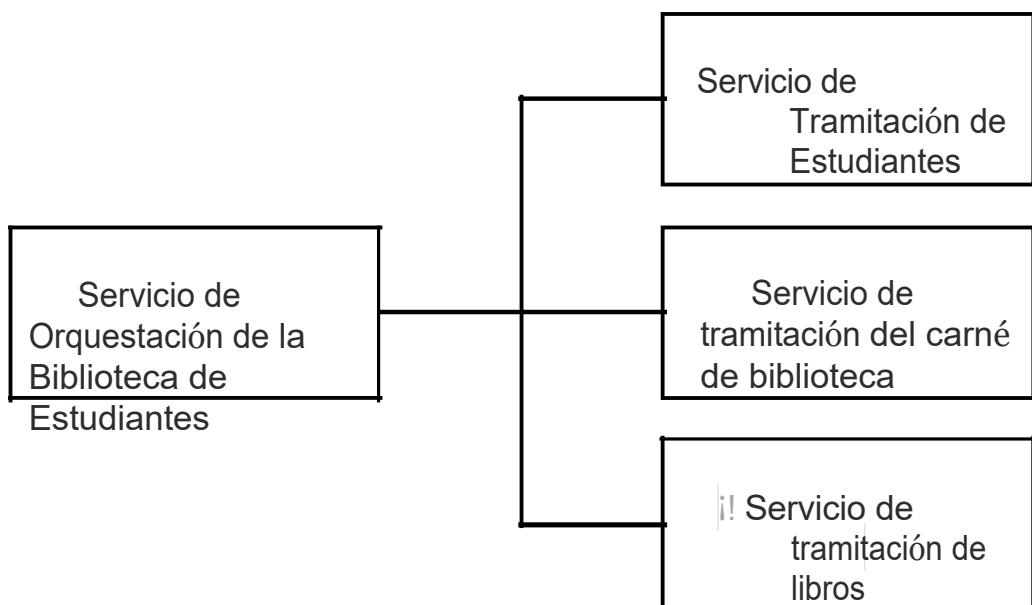
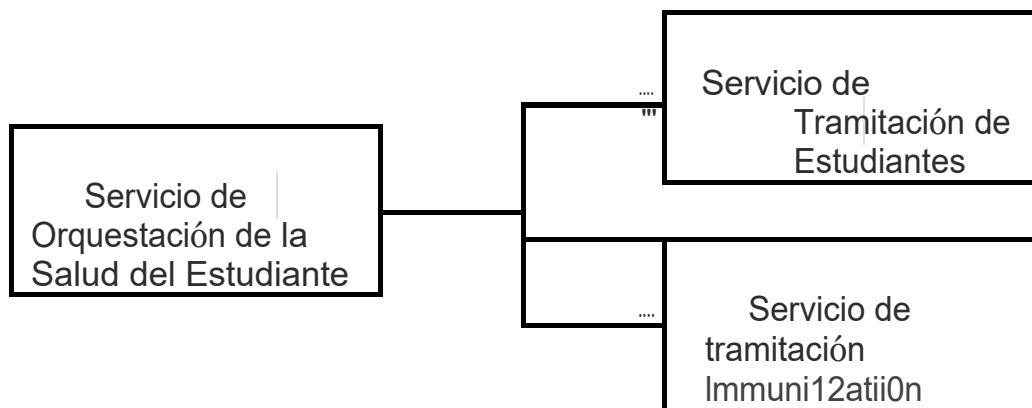
ImmunizationProcessingService. Necesitamos los cuatro servicios pero ya tenemos un StudentRegistrationOrchestrationService que tiene tres dependencias. En este punto se requiere una semi-normalización para el re-balanceo de la arquitectura para honrar la regla 'Dos-Tres' y eventualmente para controlar la complejidad.



En este caso se requiere una mayor normalización o una división para reequilibrar la arquitectura. Tenemos que pensar conceptualmente en una base común entre las entidades primitivas de un proceso de registro de estudiantes. Los requisitos de un estudiante contienen identidad, salud y materiales (tarjeta de estudiante, libro, etc.).

El servicio de orquestación combina la biblioteca y el libro bajo el mismo servicio de orquestación que rige la biblioteca.

Los libros y las bibliotecas están algo relacionados. Así que tenemos `StudentLibraryOrchestrationService` y para el otro servicio tendríamos `StudentHealthOrchestrationService` como sigue:



Para completar el flujo de registro con un nuevo modelo, se requiere un servicio de coordinación para pasar una lógica de negocio avanzada para combinar todas estas entidades juntas. Pero lo más importante es que notarás que cada orquestación tiene ahora una dependencia redundante de `StudentProcessingService`

p

ara asegurar que ninguna dependencia virtual de ningún otro servicio de orquestación cree o asegure la existencia de un registro de estudiante.

Las dependencias virtuales son muy delicadas. Se trata de una conexión oculta entre dos servicios de cualquier categoría en la que un servicio asume implícitamente que se creará y estará presente una entidad determinada. Las dependencias virtuales son muy peligrosas y amenazan la verdadera autonomía de cualquier servicio. Detectar las dependencias virtuales en una fase temprana del proceso de diseño y desarrollo podría ser una tarea desalentadora pero necesaria para garantizar una arquitectura limpia y estandarizada.

Al igual que los cambios de modelo, ya que pueden requerir migraciones, y lógica y validaciones adicionales, un nuevo requisito para una entidad adicional completamente nueva podría requerir una reestructuración de una arquitectura existente o ampliarla a una nueva versión, dependiendo de la etapa en la que el sistema está recibiendo estos nuevos requisitos.

Puede ser muy tentador añadir una dependencia adicional a un servicio de orquestación existente, pero ahí es donde el sistema empieza a desviarse de la "norma". Y es entonces cuando el sistema empieza a recorrer el camino de ser un sistema heredado imposible de mantener. Pero, lo que es más importante, es cuando los ingenieros que participan en el diseño y el desarrollo del sistema se enfrentan a sus propios principios y a su capacidad artesanal.

2.3.2.1.1.2 Sin normalización

Hay escenarios en los que cualquier nivel de normalización es un reto a alcanzar. Aunque creo que todo, en todas partes, de alguna manera está conectado. A veces puede ser incomprendible para la mente agrupar múltiples servicios bajo un servicio de orquestación.

Porque a mi mente le resulta bastante difícil pensar en un ejemplo de entidades múltiples que no tengan conexión entre sí, ya que creo sinceramente que no podría existir. Voy a basarme en algunas entidades finales para visualizar un problema. Así que vamos a suponer que hay `AService` y `BService` orquestados junto con un `XService`. La existencia de `XService` es importante para asegurar que ambos

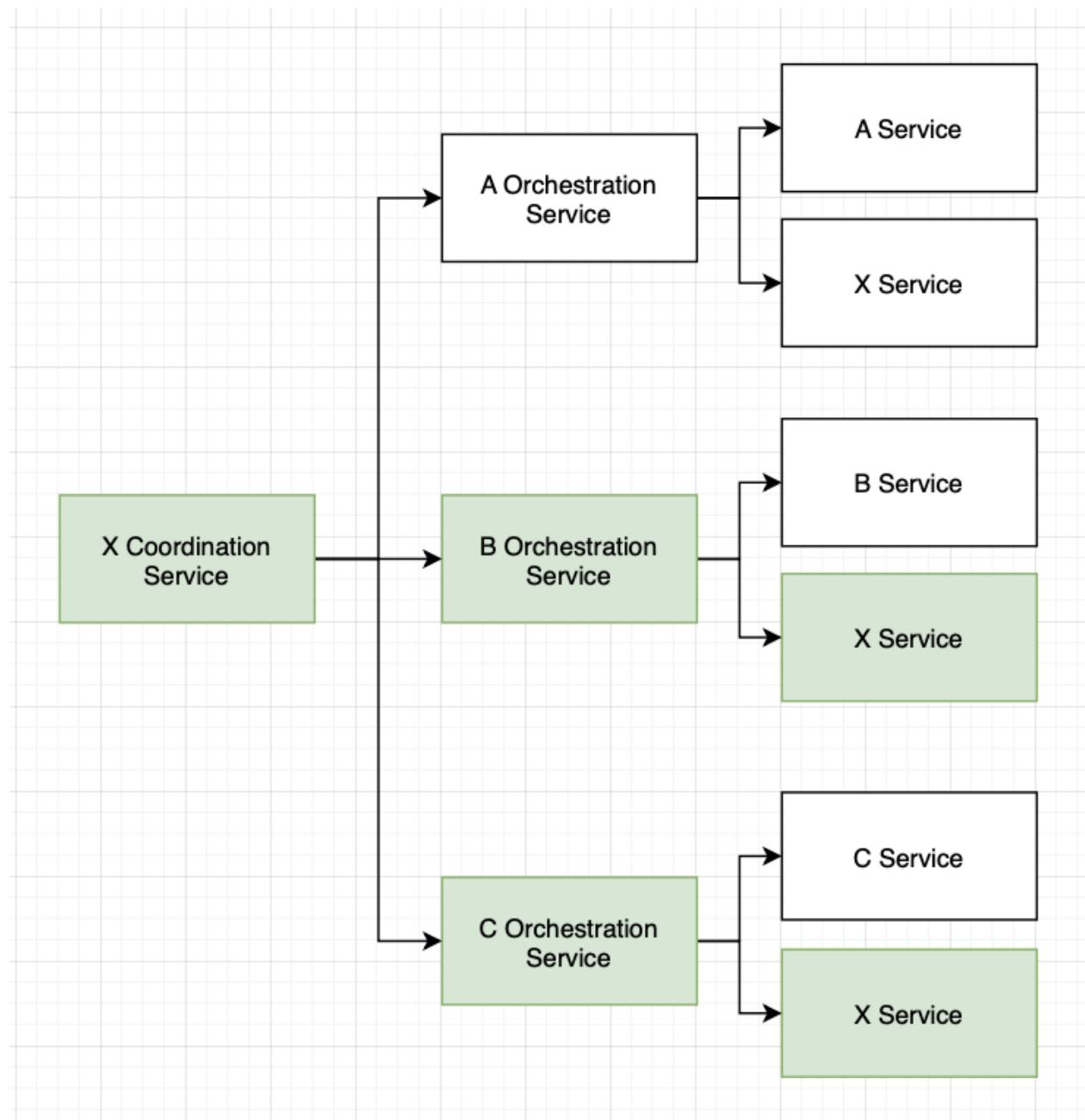
A

y B puede crearse con la seguridad de que una entidad central existe.

Ahora, digamos que un nuevo servicio es necesario añadirlo a la mezcla para completar el flujo existente. Por lo tanto, ahora tenemos cuatro dependencias diferentes bajo un servicio de orquestación, y una división es obligatoria.

Como no hay

relación alguna entre A, B y C, un enfoque de "No-Normalización" se convierte en la única opción para realizar un nuevo diseño como el siguiente:



Cada uno de los servicios primitivos anteriores se orquestará con un núcleo servicio X y luego se reúnen de nuevo bajo un servicio de coordinación. Este caso anterior

es el peor escenario, en el que una normalización de cualquier tamaño es imposible. Tenga en cuenta que el autor de esta norma no ha podido presentar un ejemplo realista, a diferencia de otros, para mostrarle lo raro que es encontrarse con esa situación, así que el enfoque de "No-Normalización" será su última solución si realmente se queda sin opciones.

2.3.2.1.1.3 Desglose significativo

Independientemente del tipo de normalización que tenga que seguir. Tiene que asegurarse de que sus servicios agrupados representan un significado común. Para

StudentProcessingService

LibraryPro

por ejemplo, reuniendo un

y

cessingService debe requerir una función común entre ambos. Un buen ejemplo de ello sería Servicio de Registro de Estudiantes hie por ejemplo. El proceso de registro requiere añadir un nuevo registro de lo estudiante y crear una tarjeta de biblioteca para ese mismo estudiante.

La implementación de los servicios de orquestación sin la intersección entre dos o tres entidades por operación, anula el propósito de tener un servicio de orquestación. Esta condición se cumple si se produce al menos una intersección entre dos entidades. Un servicio de orquestación puede entonces tener otras operaciones que llamamos "Pass-Through" en las que propagamos ciertas rutinas desde sus orígenes de procesamiento o fundación si coinciden con el mismo contrato.

He aquí un ejemplo:

```
public class Servicio de organización de estudiantes
{
    public async ValueTask< Student> RegisterStudentAsync(Student student)
    {
        Estudiante añadidoEstudiante =
            await this.studentProcessingService. AddStudentAsync(student);

        LibraryCard libraryCard =
            await this.libraryCardPorcessingService. AddLibraryCardAsync(
                addedStudent.Id);

        devolver addedStudent;
    }

    public async ValueTask< Student> ModifyStudentAsync(Student student) =>
        await this.studentProcessingService. ModifyStudentAsync(student);
}
```

En el ejemplo anterior, nuestro `studentOrchestrationService` tenía un rutina de orquestación en la que se combina la adición de un estudiante y la creación de una tarjeta de biblioteca para ese mismo estudiante. Pero, además, también ofrece una función "Pass- Through" para que una rutina de servicio de procesamiento de bajo nivel modifique a un estudiante.

Las rutinas 'Pass-Through' deben tener exactamente el mismo contrato que el resto de las rutinas de cualquier servicio de orquestación. Este es nuestro principio de 'Pure Contract' que dicta que cualquier servicio debe permitir el mismo contrato que los tipos de entrada y salida o primitivos.

2.3.2.2 Contratos

Los servicios de orquestación pueden combinar dos o tres entidades distintas y sus operaciones para lograr una lógica empresarial avanzada. Hay dos situaciones cuando se trata de contratos/modelos para los servicios de orquestación. Una que se mantiene fiel a la entidad principal de propósito. Y una que es compleja - un servicio de orquestación combinador que trata de exponer explícitamente sus entidades internas de destino.

Hablemos de estos dos escenarios en detalle.

2.3.2.2.0 Contratos físicos

Algunos servicios de orquestación siguen siendo de propósito único aunque combinen otras dos o tres rutinas de orden superior de múltiples entidades. Por ejemplo, un servicio de orquestación que reacciona a los mensajes de alguna cola y luego persigue estos mensajes son servicios de orquestación de una sola entidad y con una sola finalidad.

Veamos este fragmento de código:

```
public class Servicio de organización de estudiantes
{
    private readonly IStudentEventProcessingService studentEventProcessingService;
    private readonly IStudentProcessingService studentProcessingService;

    public StudentOrchestrationService(
        IStudentEventProcessingService studentEventProcessingService,
        IStudentProcessingService studentProcessingService)
    {
        this.studentEventProcessingService = studentEventProcessingService;
        this.studentProcessingService = studentProcessingService;
        ListenToEvents();
    }

    public void ListenToEvents () => this.studentEventService.
        ListenToEvent(UpserStudentAsync);

    public async ValueTask< Student> UpserStudentAsync(Student student)
    {
        ...
        await this.studentProcessingService. UpserStudentAsync(student);
        ...
    }
}
```

En el ejemplo anterior, el servicio de orquestación sigue exponiendo una funcionalidad

que honra el modelo físico Estudia y se comunica internamente con varios servicios que pueden ofrecer modelos completamente distintos. Estos son los escenarios en los que hay una entidad única con un propósito principal y todos los demás servicios son servicios de apoyo para garantizar el éxito de esa entidad.

En nuestro ejemplo, los servicios de orquestación *escuchan* una cola en la que se colocarán los nuevos mensajes de los estudiantes, y luego utilizan ese evento para persistir cualquier entrada

Estudia

nuevos estudiantes en el sistema. Así que el contrato *físico* ^{pte.} es el mismo lenguaje que el servicio de orquestación utiliza explícitamente como modelo para comunicarse con los servicios/expositores de flujo superior u otros.

Pero hay otros escenarios en los que una sola entidad no es el único propósito/objetivo de un servicio de orquestación. Hablemos de ello en detalle.

2.3.2.2.1 Contratos virtuales

En algunos escenarios, un servicio de orquestación puede ser requerido para crear sus propios contratos no físicos para completar una determinada operación. Por ejemplo, considere un servicio de orquestación que se requiere para persistir una publicación de medios sociales con una imagen adjunta. El requisito aquí es persistir la imagen en una tabla de base de datos, y la publicación real (comentarios, autores y otros) en una tabla de base de datos diferente en un modelo relacional.

Ahora, el modelo entrante podría ser significativamente diferente de cómo serían los modelos físicos reales. Veamos cómo se vería eso en el mundo real.

Considera tener este modelo:

```
public class MediaPost
{
    public Guid Id {get; set;}
    public string Content {get;
    set;}
    public DateTimeOffset Date {get; set;}
    public IEnumerable< string> Base64Images {get; set;}
}
```

El contrato anterior `MediaPost` contiene dos entidades físicas diferentes combinado. El primero es el puesto real, incluyendo el `IConteni` y `Fech` y la segunda es la lista de imágenes adjunta a ese mismo `post`.

Así es como un servicio de orquestación reaccionaría a este modelo virtual entrante:

```

public async ValueTask< MediaPost> SubmitMediaPostAsync(MediaPost mediaPost)
{
    ...

    Post = MapToPost(mediaPost); List< Media>
    medias = MapToMedias(mediaPost);

    Puesto añadidoPuesto =
        await this.postProcessingService. AddPostAsync(post);

    List< Medias> addedMedias =
        await this.mediaProcessingService. AddMediasAsync(medias);

    return MapToMediaPost(addedPost, addedMedias);
}

public Post MapToPost(MediaPost mediaPost)
{
    devolver el nuevo Post
    {
        Id = mediaPost.Id,
        Content = mediaPost.Content,
        CreatedDate = mediaPost.Date,
        UpdatedDate = mediaPost.Date
    };
}

public List< Media> MapToMedias(MediaPost mediaPost)
{
    return mediaPost.Base64Images. Select(image
        => new Media
        {
            Id = Guid.
            NewGuid(), PostId =
            mediaPost.Id, Image =
            image,
            CreatedDate = mediaPost.Date,
            UpdatedDate = mediaPost.Date
        });
}

```

El fragmento de código anterior muestra el servicio de orquestación deconstruyendo un modelo/contrato virtual `MediaPost` en dos modelos físicos, cada uno de ellos tiene su propio servicio de procesamiento separado que maneja su persistencia. También hay situaciones en las que el modelo virtual se descompone en un único modelo con detalles adicionales que se utilizan para la validación y la verificación con los servicios de procesamiento posteriores o de la fundación.

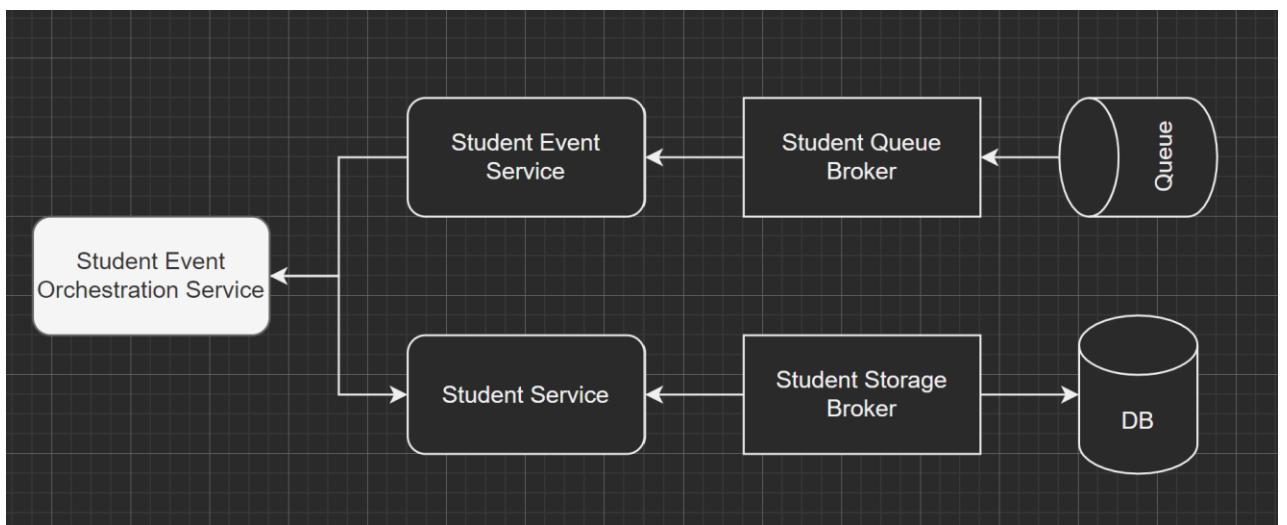
También hay situaciones híbridas en las que el modelo virtual entrante puede tener modelos físicos anidados en él. Lo cual es algo que sólo podemos permitir con los modelos virtuales. Los modelos físicos deben permanecer anémicos (no contienen rutinas ni constructores) y flat (no contienen modelos anidados) en todo momento para controlar la complejidad y centrar la responsabilidad.

En resumen, los servicios de orquestación pueden crear sus propios contratos. Estos contratos pueden ser físicos o virtuales. Y un contrato virtual puede ser una combinación de uno o varios contratos físicos (o virtuales anidados) o simplemente tener su propio diseño en términos de propiedades.

2.3.2.2 Callejón sin salida

Hay veces que los servicios de orquestación y sus equivalentes (coordinación, gestión... etc.) pueden no necesitar un componente expositor (controlador por ejemplo). Esto se debe a que estos servicios pueden ser oyentes de ciertos eventos y comunicar el evento de vuelta a un servicio de procesamiento o de fundación en el mismo nivel en el que el evento comenzó o fue recibido.

Por ejemplo, imagina que construyes una aplicación simple en la que se notifica con mensajes de una cola y luego mapea estos mensajes en algún modelo local para persistirlo en el almacenamiento. En este caso, la entrada de estos servicios no es necesariamente a través de un componente expositor. Los mensajes entrantes pueden ser recibidos desde una suscripción a un servicio de eventos o una cola. En este caso el servicio de orquestación tendría un aspecto similar al siguiente:



El `StudentEventOrchestrationService` en este caso, escucha el para los nuevos estudiantes que llegan y lo convierte inmediatamente en modelos que pueden persistir en la base de datos.

He aquí un ejemplo:

Comencemos con una prueba unitaria para este patrón de la siguiente manera:

```

[Hecho]
public void DeberíaEscucharLosEventosDelPerfil()
{
    // dado . cuando this.profileEventOrchestrationService.
    ListenToProfileEvents();

    // entonces this.profileEventServiceMock.
    Verify(service =>
        servicio. ListenToProfileEvent(
            this.profileEventOrchestrationService.ProcessProfileEventAsync),
        Times.Once);

    this.profileEventService.VerifyNoOtherCalls(); this.profileServiceMock.
    VerifyNoOtherCalls(); this.loggingBrokerMock. VerifyNoOtherCalls();
}

[Hecho]
public async Task ShouldAddProfileAsync()
{
    // dado
    ProfileEvent randomProfileEvent =
        CreateRandomProfileEvent();

    ProfileEvent inputProfileEvent =
        randomProfileEvent;

    this.profileServiceMock. Setup(service => service.
        AddProfileAsync(inputProfileEvent.Profile));

    // cuando
    await this.profileEventOrchestrationService
        . ProcessProfileEventAsync(inputProfileEvent);

    // entonces this.profileServiceMock.
    Verify(service =>
        servicio. AddProfileAsync(inputProfileEvent.Profile),
        Times.Once);

    this.profileServiceMock.VerifyNoOtherCalls(); this.loggingBrokerMock.
    VerifyNoOtherCalls(); this.profileEventService.VerifyNoOtherCalls();
}

```

La prueba aquí indica que primero tiene que ocurrir una escucha de eventos, luego una persistencia en el servicio de estudiantes debe coincidir con el resultado de la asignación de un mensaje entrante a un estudiante determinado.

Intentemos que esta prueba sea superada.

```

public partial class ServicioDeOrquestaciónDeEventosDePerfil : IProfileEventOrchestrationService
{
    private readonly IProfileEventService profileEventService;
    private readonly IProfileService profileService;
    private readonly ILoggingBroker loggingBroker;

    public ProfileEventOrchestrationService(
        IProfileEventProcessingService profileEventService,
        IProfileProcessingService profileService,
        ILoggingBroker loggingBroker)
    {
        this.profileEventService = profileEventService;
        this.profileService = profileService;
        this.loggingBroker = loggingBroker;
    }

    public void ListenToProfileEvents() =>
    TryCatch(() =>
    {
        this.profileEventService. ListenToProfileEvent(
            ProcessProfileEventAsync);
    });

    public ValueTask ProcessProfileEventAsync(ProfileEvent profileEvent) =>
    TryCatch(async () =>
    {
        ...
        await this.profileService. AddProfileAsync(profileEvent.Profile);
    });
}

```

En el ejemplo anterior, el constructor del servicio de orquestación se suscribe a los eventos que vendrían del `StudentEventService`, cuando se produce un evento, el servicio de orquestación llamará al `ProcessingIncomingStudentMessageAsync` para persistir el estudiante entrante en la base de datos a través de una fundación o un servicio de procesamiento al mismo nivel que el servicio de eventos.

Este patrón o característica se denomina Cul-De-Sac. Donde un mensaje entrante será un giro y se dirigirá en una dirección diferente para una dependencia diferente. Este patrón es muy común en grandes aplicaciones de nivel empresarial donde se incorpora la consistencia eventual para asegurar que el sistema puede escalar y ser resistente bajo un gran consumo. Este patrón también previene los ataques maliciosos contra sus puntos finales de la API, ya que permite procesar los mensajes o eventos de la cola siempre que el servicio esté listo para procesarlos. Discutiremos los detalles en "La arquitectura estándar".

2.3.3 Responsabilidades

Los servicios de orquestación proporcionan una lógica empresarial avanzada. Orquestan múltiples flujos para múltiples entidades/modelos para completar un único flujo. Vamos a discutir en detalle cuáles son estas responsabilidades:

2.3.3.0 Lógica avanzada

Los servicios de orquestación no pueden existir sin combinar múltiples rutinas de varias entidades. Estas entidades pueden ser de naturaleza distinta, pero comparten un propósito común. Por ejemplo, un modelo de tarjeta de biblioteca como modelo es fundamentalmente diferente de un modelo de estudiante. Sin embargo, ambos comparten propósito común cuando se trata del proceso de registro de estudiantes. Es necesario añadir un registro de estudiante para inscribir a un estudiante, pero también asignar una tarjeta de biblioteca a ese mismo estudiante es un requisito para que el proceso de inscripción de estudiantes sea exitoso.

Los servicios de orquestación garantizan que se integren las rutinas correctas para cada entidad, pero también garantizan que estas rutinas se llamen en el orden correcto. Además, los servicios de orquestación se encargan de retroceder en caso de que una operación falle, si es necesario. Estos tres aspectos son los que constituyen un effort de orquestación a través de múltiples rutinas, entidades o contratos.

Hablemos de ellos en detalle.

2.3.3.0.0 Combinaciones de flujo

Hemos hablado antes de los servicios de orquestación que combinan múltiples rutinas para lograr un propósito común o un único flujo. Este aspecto de los servicios de orquestación puede servir como característica fundamental pero también como responsabilidad. Y un servicio de orquestación sin al menos una rutina que combine dos o tres entidades no se considera verdaderamente una orquestación.

La integración con múltiples servicios sin un propósito común es una mejor fit definición para los servicios de agregación que trataremos más adelante en este capítulo de servicios.

Pero dentro de la combinación flow viene la unificación del contrato. Lo llamo mapeo y ramificación. Mapear un modelo entrante en múltiples modelos de servicios de flujo inferior y luego ramificar la responsabilidad entre estos servicios.

Al igual que los servicios anteriores, los servicios de orquestación son responsables durante su combinación de flow de asegurar la pureza de los contratos de entrada y salida expuestos. Lo cual se vuelve un poco más complejo cuando se combinan múltiples modelos. Los servicios de orquestación seguirán siendo responsables de mapear los contratos entrantes a sus respectivos servicios descendentes, pero también se aseguran de mapear de nuevo los resultados devueltos por estos servicios en el modelo unificado.

Volvamos a un fragmento de código anterior para ilustrar este aspecto:

```

public async ValueTask< MediaPost> SubmitMediaPostAsync(MediaPost mediaPost)
{
    ...

    Post = MapToPost(mediaPost); List< Media>
    medias = MapToMedias(mediaPost);

    Puesto añadidoPuesto =
        await this.postProcessingService. AddPostAsync(post);

    List< Medias> addedMedias =
        await this.mediaProcessingService. AddMediasAsync(medias);

    return MapToMediaPost(addedPost, addedMedias);
}

private Post MapToPost(MediaPost mediaPost)
{
    devolver el nuevo Post
    {
        Id = mediaPost.Id,
        Content = mediaPost.Content,
        CreatedDate = mediaPost.Date,
        UpdatedDate = mediaPost.Date
    };
}

private List< Media> MapToMedias(MediaPost mediaPost)
{
    return mediaPost.Base64Images. Select(image
        => new Media
        {
            Id = Guid.
            NewGuid(), PostId =
            mediaPost.Id, Image =
            image,
            CreatedDate = mediaPost.Date,
            UpdatedDate = mediaPost.Date
        });
}

private MediaPost MapToMediaPost(Post post, List< Media> medias)
{
    devolver el nuevo MediaPost
    {
        Id = post.Id,
        Content = post.Content,
        Date = post.CreatedDate,
        Base64Images = medias. Select(media => media.Image)
    }
}

```

Como se puede ver en el ejemplo anterior, el mapeo y la bifurcación no sólo se producen en la entrada. Sino que hay que realizar una acción inversa en la salida.

Es una violación de la norma devolver el mismo objeto de entrada que se pasó. Eso elimina cualquier visibilidad sobre cualquier cambio potencial que haya ocurrido a la solicitud entrante durante la persistencia. El mapeo dúplex debería sustituir la necesidad de desreferenciar la petición entrante para asegurar que no se han producido cambios internos inesperados.

Tenga en cuenta que también se recomienda romper la lógica de mapeo en su propio aspecto / clase parcial file. Algo así como de organización de `appings.cs` para asegurarse de que lo único que queda allí es la lógica de negocio de orquestación.

2.3.3.0.1 Orden de llamada

Llamar a las rutinas en el orden correcto puede ser crucial para cualquier proceso de orquestación. Por ejemplo, no se puede crear un carné de biblioteca si no se crea primero un registro de estudiante. Hacer cumplir el orden aquí puede dividirse en dos tipos diferentes. Vamos a hablar de ellos

aquí un poco.

2.3.3.0.1.0 Orden natural

El orden natural aquí se refiere a ciertos flujos que no pueden ser ejecutados a menos que un prerequisito de parámetros de entrada sea recuperado o persistido. Por ejemplo, imaginemos una situación en la que no se puede crear un carné de biblioteca a menos que se recupere primero el identificador único del estudiante. En este caso, no tenemos que preocuparnos por comprobar que ciertas rutinas se han llamado en el orden correcto, ya que esto viene de forma natural con el flujo.

Aquí hay un ejemplo de código de esta situación:

```
public async ValueTask< LibraryCard> CreateLibraryCardAsync(LibraryCard libraryCard)
{
    Student student = await this.studentProcessingService
        .RetrieveStudentByIdAsync(libraryCard.StudentId);

    return await this.libraryCardProcessingService
        .CreateLibraryCardAsync(libraryCard, student.Name);
}
```

En el ejemplo anterior, tener un alumno `Nombre` es un requisito para crear una tarjeta de la biblioteca. Por lo tanto, la `questación` del orden aquí viene naturalmente como parte del flujo sin ningún `effort` adicional.

Hablemos del segundo tipo de orden: la orden forzosa.

2.3.3.0.1.1 Orden de ejecución

Imagina el mismo ejemplo anterior, pero en lugar del carné de la biblioteca que requiera un nombre de estudiante, en su lugar sólo `Id` que ya es necesario el estudiante incluido en el modelo de solicitud entrante. Algo así:

```
public async ValueTask< LibraryCard> CreateLibraryCardAsync(LibraryCard libraryCard)
{
    await this.studentProcessingService.VerifyEnlistedStudentExistAsync(
        libraryCard.StudentId);

    return await this.libraryCardProcessingService.CreateLibraryCardAsync(libraryCard);
}
```

Garantizar que la función `VerifyEnlistedStudentExistAsync` haya sucedido antes de crear una tarjeta de biblioteca podría ser un desafío para lograr de forma natural porque no hay dependencia entre el valor de retorno de una rutina y los parámetros de entrada de la siguiente. En otras palabras, no hay nada que la función `VerifyEnlistedStudentExistAsync` devuelva que a la función `CreateLibraryCardAsync` le importe en términos de parámetros de entrada.

En este caso, un tipo de orden forzado debe ser implementado a través de pruebas unitarias. Una prueba unitaria para esta rutina requeriría verificar no sólo que la dependencia ha sido llamada con los parámetros correctos, sino también que son llamados en el *orden correcto* echemos un vistazo a cómo se implementaría:

```

[Hecho]
public async Task DeberíaCrearTarjetaDeBibliotecaAsync()
{
    // dado
    Estudiante algúnEstudiante = CreateRandomStudent();
    LibraryCard randomLibraryCard = CreateRandomLibraryCard();
    LibraryCard inputLibraryCard = randomLibraryCard;
    LibraryCard createdLibraryCard = inputLibraryCard;
    LibraryCard expectedLibraryCard = inputLibraryCard;
    DeepClone(); Guid studentId = inputLibraryCard.StudentId;
    var mockSequence = new MockSequence();

    this.studentProcessingServiceMock. InSequence(mockSequence). Setup(service =>
        service. VerifyEnlistedStudentExistAsync(studentId)
            . Devuelve(algúnEstudiante);

    this.libraryCardProcessingServiceMock. InSequence(mockSequence). Setup(service =>
        service. CreateLibraryCardAsync(inputLibraryCard)
            . DevuelveAsync(createdLibraryCard);

    // cuando
    LibraryCard actualLibraryCard = await this.libraryCardOrchestrationService
        . CreateLibraryCardAsync(inputLibraryCard);

    // entonces actualLibraryCard. Debería().
    BeEquivalentTo(expectedLibraryCard);

    this.studentProcessingServiceMock. Verify(service => service.
        VerifyEnlistedStudentExistAsync(studentId),
        Times.Once);

    this.libraryCardProcessingServiceMock. Verify(service => service.
        CreateLibraryCardAsync(inputLibraryCard),
        Times.Once);

    this.studentProcessingServiceMock. VerifyNoOtherCalls();
    this.libraryCardProcessingServiceMock. VerifyNoOtherCalls();
    this.loggingBrokerMock. VerifyNoOtherCalls();
}

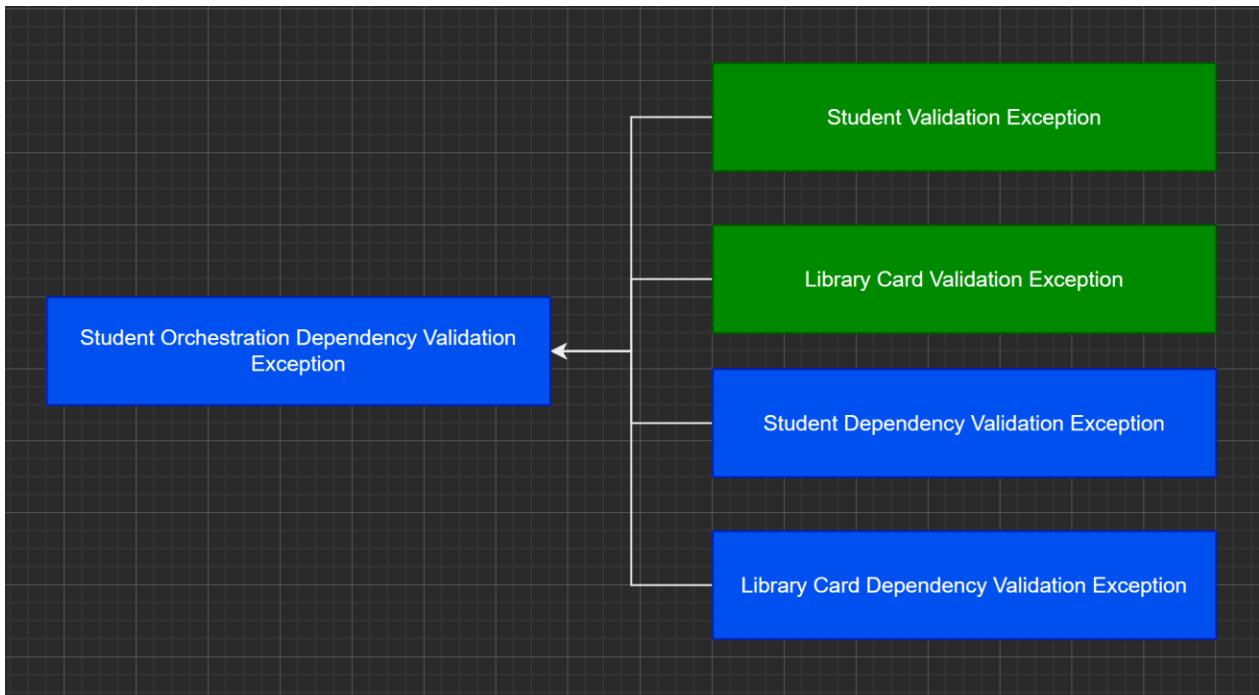
```

En el ejemplo anterior, el marco de trabajo mock aquí se utiliza para garantizar un cierto orden se está aplicando cuando se trata de llamar a estas dependencias. De este modo, se impone una implementación determinada dentro de cualquier método para garantizar que las dependencias no conectadas de forma natural se llamen de forma secuencial en el orden exacto.

Es más probable que la forma de ordenación se incline más hacia lo forzado que hacia lo natural cuando los servicios de orquestación alcanzan el número máximo de dependencias que pueden tener en un momento dado.

2.3.3.0.2 Mapeo de excepciones (Wrapping & Unwrapping)

Esta responsabilidad es muy similar a la de las combinaciones de flujo. Excepto que en este caso los servicios de orquestación unifican todas las excepciones que pueden ocurrir fuera de cualquiera de sus dependencias en un modelo de excepción categórico unificado. Empecemos con una ilustración de cómo puede ser ese mapeo:



En la ilustración anterior, observará que las excepciones de validación y validación de dependencia lanzadas desde los servicios de dependencia descendentes se están mapeando en una excepción de dependencia unificada en el nivel de orquestación. Esta práctica permite a los consumidores de aguas arriba de ese mismo servicio de orquestación determinar el siguiente curso de acción basado en un tipo de excepción categórica en lugar de cuatro o, en el caso de tres dependencias, serían seis dependencias categóricas.

Empecemos con una prueba fallida para materializar nuestra idea aquí:

```

public static TheoryData DependencyValidationExceptions()
{
    string exceptionMessage = GetRandomMessage();
    var innerException = new Xception(exceptionMessage);

    var studentValidationException =
        new StudentValidationException(innerException);

    var studentDependencyValidationException =
        new StudentDependencyValidationException(innerException);

    var libraryCardValidationException =
        new LibraryCardValidationException(innerException);

    var libraryCardDependencyValidationException =
        new LibraryCardDependencyValidationException(innerException);

    return new TheoryData< Xception>
    {
        studentValidationException,
        studentDependencyValidationException,
        libraryCardValidationException,
        libraryCardDependencyValidationException
    };
}

[Teoria]
[MemberData(nameof(DependencyValidationExceptions))]
public async Task ShouldThrowDependencyValidationExceptionOnCreateIfDependencyValidationErrorOccursAndLogItAsync(
    Xception dependencyValidationException)
{
    // dado
    Estudiante algúnEstudiante = CreateRandomStudent();

    var expectedStudentOrchestrationDependencyValidationException =
        new StudentOrchestrationDependencyValidationException(
            dependencyValidationException.InnerException como Xception);

    this.studentServiceMock. Setup(service => service.AddStudentAsync(It.IsAny<
        Student>()))
        . ThrowsAsync(dependencyValidationException);

    // cuando
    ValueTask< Student> addStudentTask =
        await this.studentOrchestrationService. AddStudentAsync(someStudent);

    StudentOrchestrationDependencyValidationException
        actualStudentOrchestrationDependencyValidationException =
        await Assert.ThrowsAsync< StudentOrchestrationDependencyValidationException>(
            addStudentTask.AsTask);

    // entonces
    actualStudentOrchestrationDependencyValidationException. Debería()
        . BeEquivalentTo(expectedStudentOrchestrationDependencyValidationException);

    this.studentServiceMock. Verify(service => service.AddStudentAsync(It.IsAny<
        Student>()),
        Times.Once);

    this.loggingBrokerMock. Verify(broker => broker.LogError(It.
        Is(SameExceptionAs(
            expectedStudentOrchestrationDependencyValidationException)),
        Times.Once));

    this.libraryCardServiceMock. Verify(service => service.AddLibraryCard(It.IsAny<
        Guid>()),
        Times.Once);

    this.studentServiceMock. VerifyNoOtherCalls(); this.loggingBrokerMock.
    VerifyNoOtherCalls(); this.libraryCardServiceMock. VerifyNoOtherCalls();
}

```

En la prueba anterior, estamos verificando que cualquiera de las cuatro excepciones mencionadas al ocurrir, se mapearían en una `StudentOrchestrationDependencyValidationException`. Mantenemos la excepción localizada original como una excepción interna. Pero desenvolvemos la excepción categórica en este nivel para mantener la cuestión original a medida que avanzamos.

Estas excepciones se mapean bajo una excepción de validación de dependencia porque se originan en una dependencia o en una dependencia de una dependencia posterior. Por ejemplo, si un agente de almacenamiento lanza una excepción que se considera una validación de dependencia (algo como `DuplicateKeyException`). El servicio vecino al broker mapearía esa excepción en una `StudentAlreadyExistException` localizada y luego envolvería esa excepción en una excepción categórica de tipo `StudentDependencyValidationException`. Cuando esa excepción se propaga a un servicio de procesamiento o de orquestación, perdemos la excepción categórica porque ya la hemos capturado en el ámbito correcto del mapeo. Entonces continuamos incrustando esa excepción muy localizada bajo la excepción de validación de dependencia de servicio actual.

Intentemos que esta prueba sea superada:

```
public partial class Servicio de organización de estudiantes
{
    private delegate Task< Alumno> DevoluciónFunciónAlumno();

    private async ValueTask< Student> TryCatch(ReturningStudentFunction returningStudentFunction)
    {
        pru
        ebe
        con return await returningStudentFunction();
        {

        }
        catch (StudentValidationException studentValidationException)
        {
            Lanzar CreateAndLogDependencyValidationException(studentValidationException);
        }
        catch (StudentDependencyValidationException studentDependencyValidationException)
        {
            lanzar CreateAndLogDependencyValidationException(studentDependencyValidationException);
        }
        catch (LibraryCardValidationException libraryCardValidationException)
        {
            lanzar CreateAndLogDependencyValidationException(libraryCardValidationException);
        }
        catch (LibraryCardDependencyValidationException libraryCardDependencyValidationException)
        {
            lanzar CreateAndLogDependencyValidationException(libraryCardDependencyValidationException);
        }
    }

    private StudentOrchestrationDependencyValidationException CreateAndLogDependencyValidationException(Xception exception)
    {
        var studentOrchestrationDependencyValidationException =
            new StudentOrchestrationDependencyValidationException(exception.innerException as Xception); this.loggingBroker.

        LogError(studentOrchestrationDependencyValidationException);

        lanzar studentOrchestrationDependencyValidationException;
    }
}
```

Ahora podemos utilizar `TryCatch` de la siguiente manera:

```
public async ValueTask< Student> AddStudentAsync(Student student) =>
TryCatch(async () =>
{
    ...
    Student addedStudent = await this.studentService.AddStudentAsync(student);
    LibraryCard libraryCard = await this.libraryCard.AddLibraryCard(addedStudent.Id);

    devolver addedStudent;
});
```

Puedes ver ahora en la implementación que hemos mapeado los cuatro tipos diferentes de excepciones de validación de servicios externos descendentes en una excepción categórica y luego hemos mantenido la excepción interna para cada uno de ellos.

La misma regla se aplica a las excepciones de dependencia. Las excepciones de dependencia pueden ser tanto excepciones de servicio como de dependencia de servicios posteriores. Por ejemplo, en el ejemplo anterior, la llamada a un servicio de estudiantes puede producir

y `StudentServiceException`. Ambas excepciones categóricas serán desenvueltas de su capa categórica y tendrán su capa local envuelta en una nueva excepción categórica unificada a nivel de orquestación bajo

`StudentOrchestrationDependencyException`. Lo mismo se aplica a todas las demás excepciones categóricas de dependencia como

`LibraryCardDependencyException`

y `LibraryCardServiceException`.

Es extremadamente importante desenvolver y envolver las excepciones localizadas de los servicios descendentes con excepciones categóricas en la capa de servicio actual para asegurar que en la capa de expositores estas excepciones puedan ser fácilmente manejadas y mapeadas en lo que dicte la naturaleza del componente expositor. En el caso de un componente Exposer de tipo Controlador API, el mapeo produciría Códigos de Estado Http. En el caso de los componentes UI Exposer se mapearía a un texto significativo para los usuarios finales.

Más adelante en esta Norma discutiremos cuándo determinar la exposición de detalles de excepciones internas localizadas en las que los usuarios finales no deben realizar ninguna acción. Esto es exclusivo de las excepciones de dependencia y de nivel de servicio.

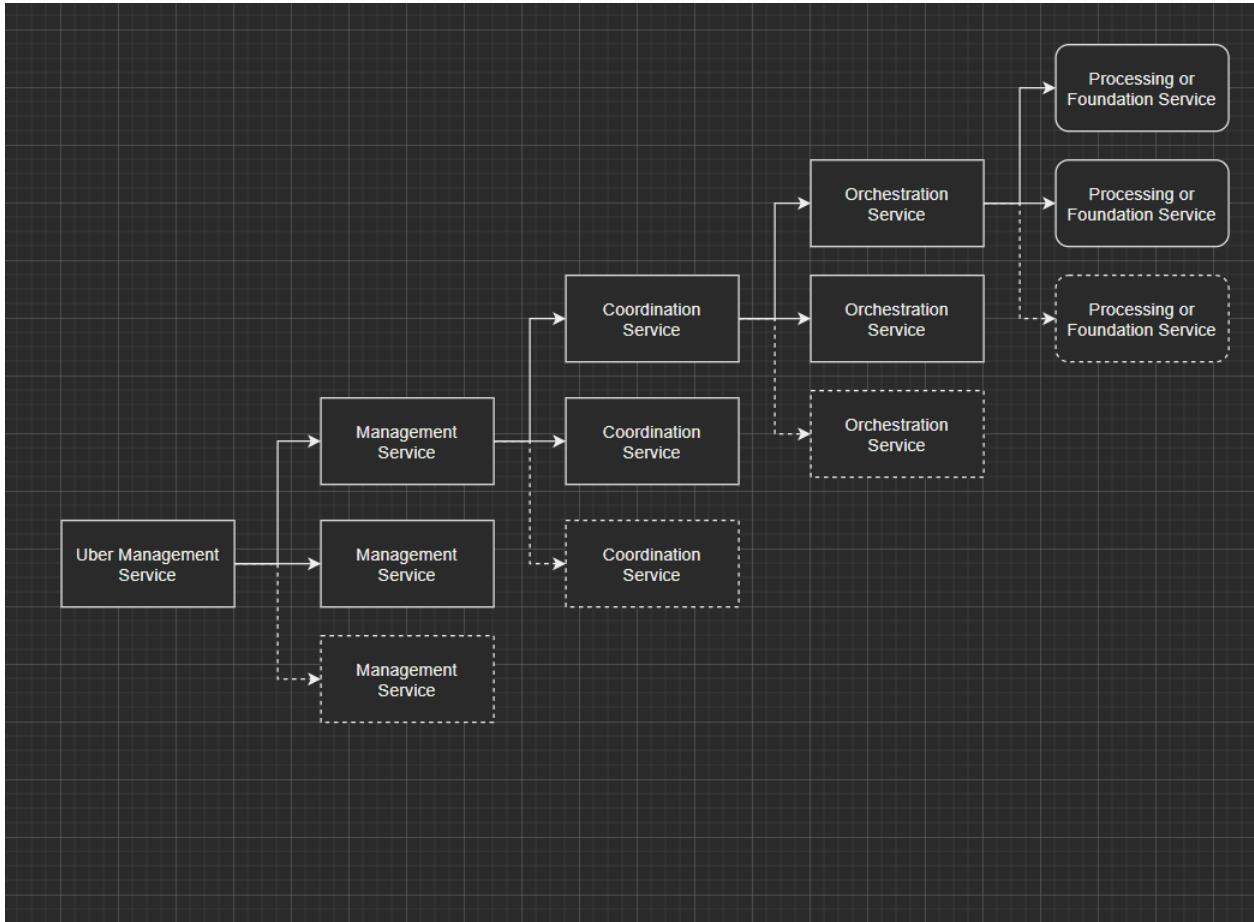
2.3.4 Variaciones

Los servicios de orquestación tienen varias variaciones dependiendo de su posición en la arquitectura general de bajo nivel. Por ejemplo, un servicio de orquestación que depende de servicios de orquestación descendentes se denomina Servicio de Coordinación. Un servicio de orquestación que trabaja con múltiples servicios de coordinación como dependencias se denomina Servicio de Gestión. Todas estas variantes son, en esencia, un servicio de orquestación con una lógica empresarial de nivel superior.

2.3.4.0 Variantes Niveles

Veamos las posibles variantes de los servicios de orquestación y su

ubicación:



En mi experiencia personal, rara vez he tenido que resolver a un servicio de gestión de Uber. La idea de la limitación aquí en términos de dependencias y variaciones de servicios similares a la orquestación es ayudar a los ingenieros a repensar la complejidad de su lógica. Pero hay que admitir que hay situaciones en las que la complejidad es una necesidad absoluta, por lo que los servicios de gestión de Uber existen como una opción.

La siguiente tabla debería guiar el proceso de desarrollo de variantes de servicios de orquestación en función del nivel:

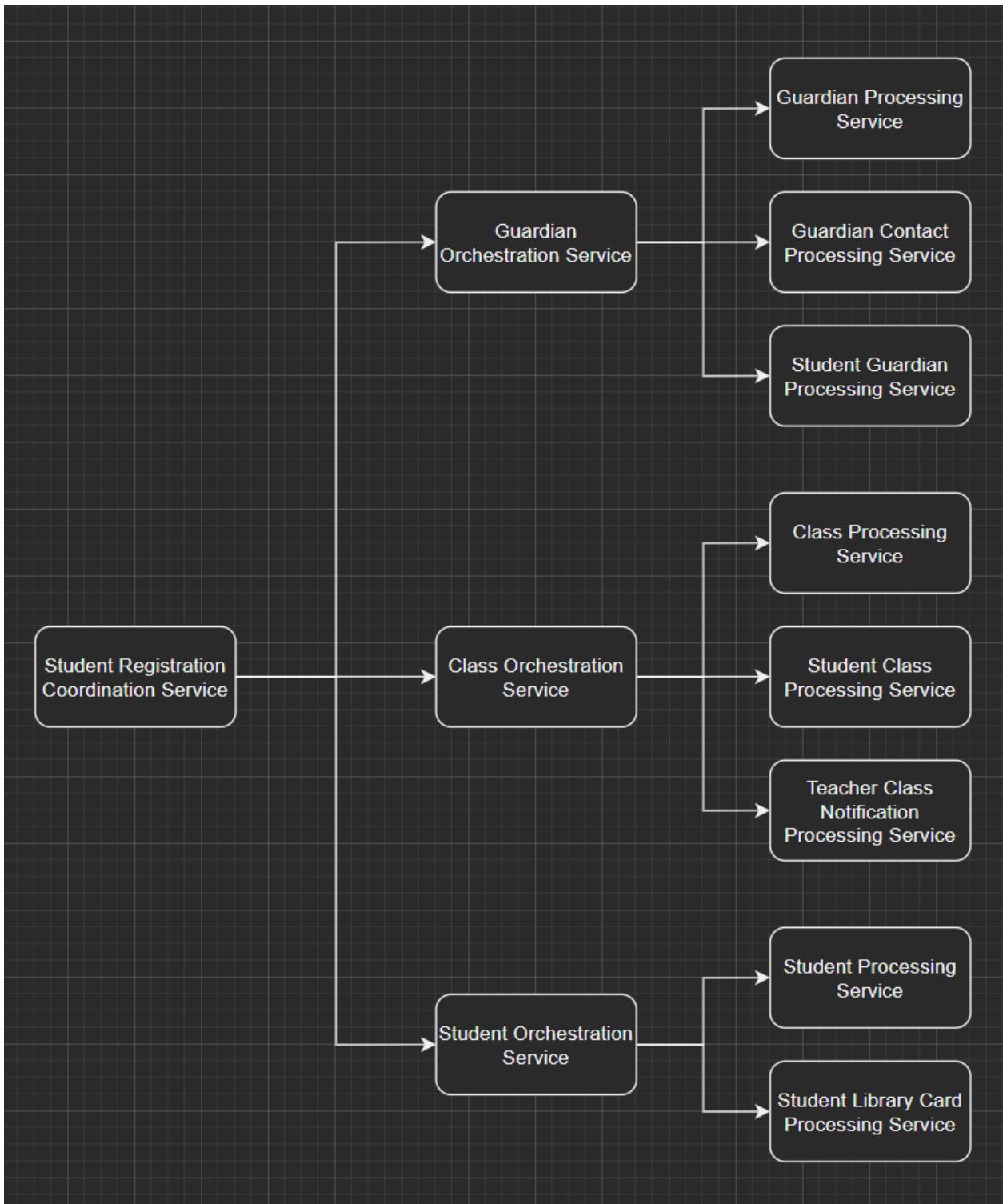
Variante	Dependencias	Consumidores	Complejidad
Servicios de orquestación	Servicios de fundación o tramitación	Servicios de coordinación	Bajo
Servicios de coordinación	Servicios de orquestación	Servicios de gestión	Medio
Servicios de gestión	Servicios de coordinación	Servicios de gestión de Uber	Alta
Servicios de gestión de Uber	Servicios de gestión	Componentes de agregación, vistas o expositores	Muy alta

Trabajar más allá de los servicios de gestión de Uber de manera orquestada requeriría una discusión más profunda y una consideración seria de la arquitectura general. Algunas versiones futuras de The Standard podrían abordar esta cuestión en lo que yo llamo "La Casa del Lago", pero eso está fuera del alcance de esta versión de The Standard.

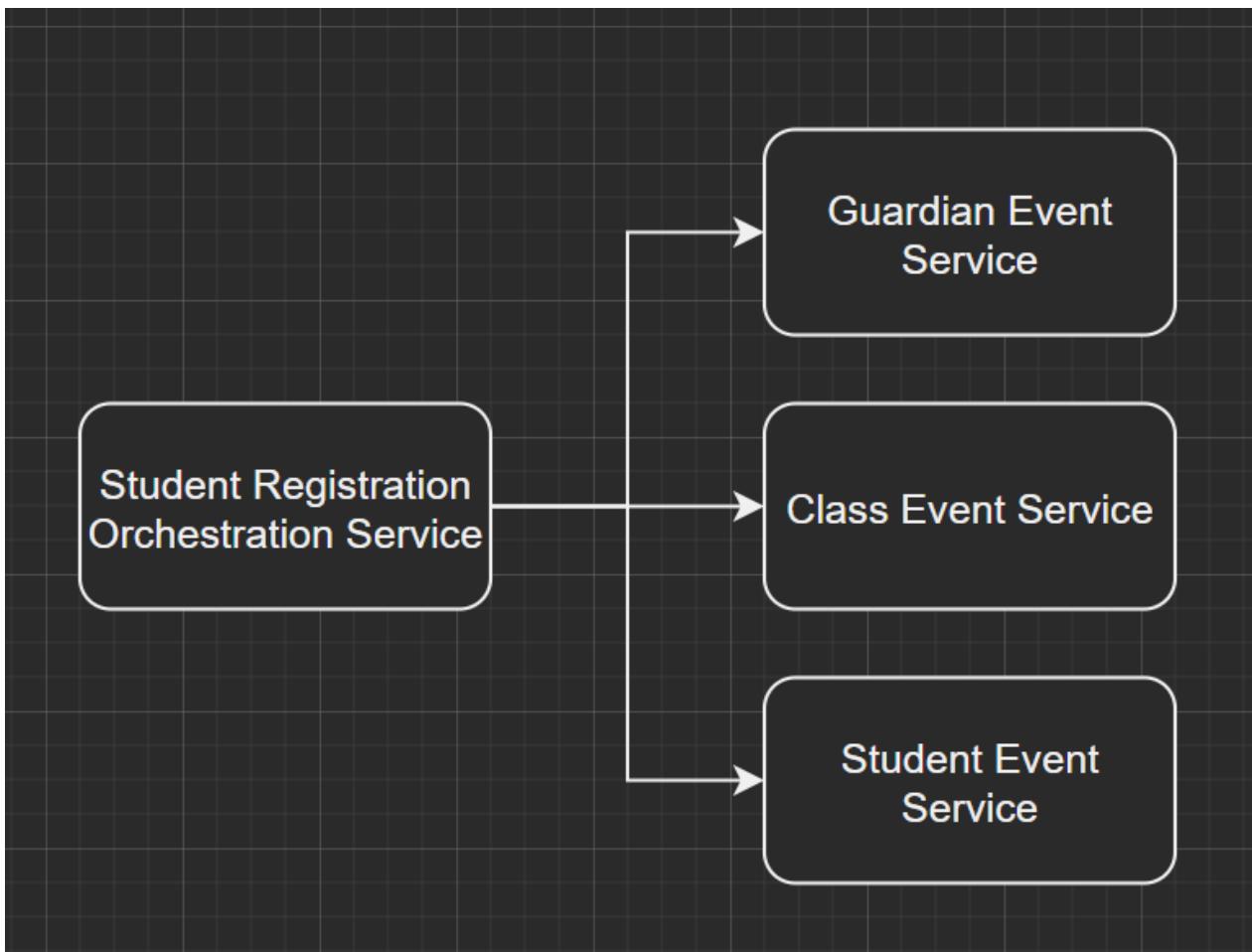
2.3.4.1 Unidad de trabajo

Con las variaciones de los servicios de orquestación, recomiendo encarecidamente mantenerse fiel al concepto de unidad de trabajo. Donde cada solicitud puede hacer una cosa y sólo una cosa incluyendo sus requisitos. Por ejemplo, si necesitas inscribir a un estudiante en alguna escuela, puedes requerir también añadir un tutor, información de contacto y algunos otros detalles. Evitar estas acciones puede disminuir en gran medida la complejidad del flujo y reducir el riesgo de cualquier fallo en los servicios posteriores.

Aquí hay una visualización para un enfoque complejo de un solo hilo:



La solución anterior es una solución que funciona para registrar a un estudiante. Necesitábamos incluir la información del tutor, las tarjetas de la biblioteca, las clases ... etc. Estas dependencias se pueden desglosar en términos de eventing y permitir que otros servicios recojan donde los servicios de un solo hilo dejan off para continuar el proceso de registro. Algo así:



La solicitud entrante en el ejemplo anterior se convertiría en eventos, donde cada uno de estos eventos estaría notificando a sus propios servicios de orquestación de una manera de callejón sin salida como hemos discutido anteriormente en la sección 2.3.2.2. Lo que esto significa es que un único hilo ya no es responsable del éxito de todas y cada una de las dependencias del sistema. En su lugar, cada corredor de escucha de eventos manejaría su propio proceso de una manera mucho, mucho más simplificada.

Este enfoque no garantiza una respuesta inmediata de éxito o fracaso al solicitante. Es un patrón de consistencia eventual donde el cliente obtendría un `Aceptado` mensaje o su equivalente basado en el protocolo de comunicación para hacerles saber que se ha iniciado un proceso, pero que aún no hay garantía de resultados hasta que se termine.

También es importante mencionar que se puede añadir una capa extra de resistencia a estos eventos almacenándolos temporalmente en componentes tipo cola o en almacenes temporales basados en la memoria; dependiendo de la criticidad del negocio.

Pero un enfoque de consistencia eventual no siempre es una buena solución si el cliente del otro lado está esperando una respuesta. Especialmente en situaciones muy críticas en las que se requiere una respuesta inmediata. Este problema puede resolverse mediante las colas Fire-n-Observe, de las que hablaremos en una futura versión de The Standard.

[Introducción a los servicios de orquestación]

[Patrón de Cul-De-Sac para los servicios de orquestación]

[*] [Patrón de callejón sin salida para los servicios de coordinación]

2.4 Servicios de agregación (The Knot)

2.4.0 Introducción

La principal responsabilidad de los servicios de agregación es exponer un único punto de contacto entre la capa lógica de negocio principal y cualquier capa de exposición. Garantiza, si varios servicios de cualquier variación comparten el mismo contrato, que se agreguen y expongan a un componente expositor a través de una capa lógica.

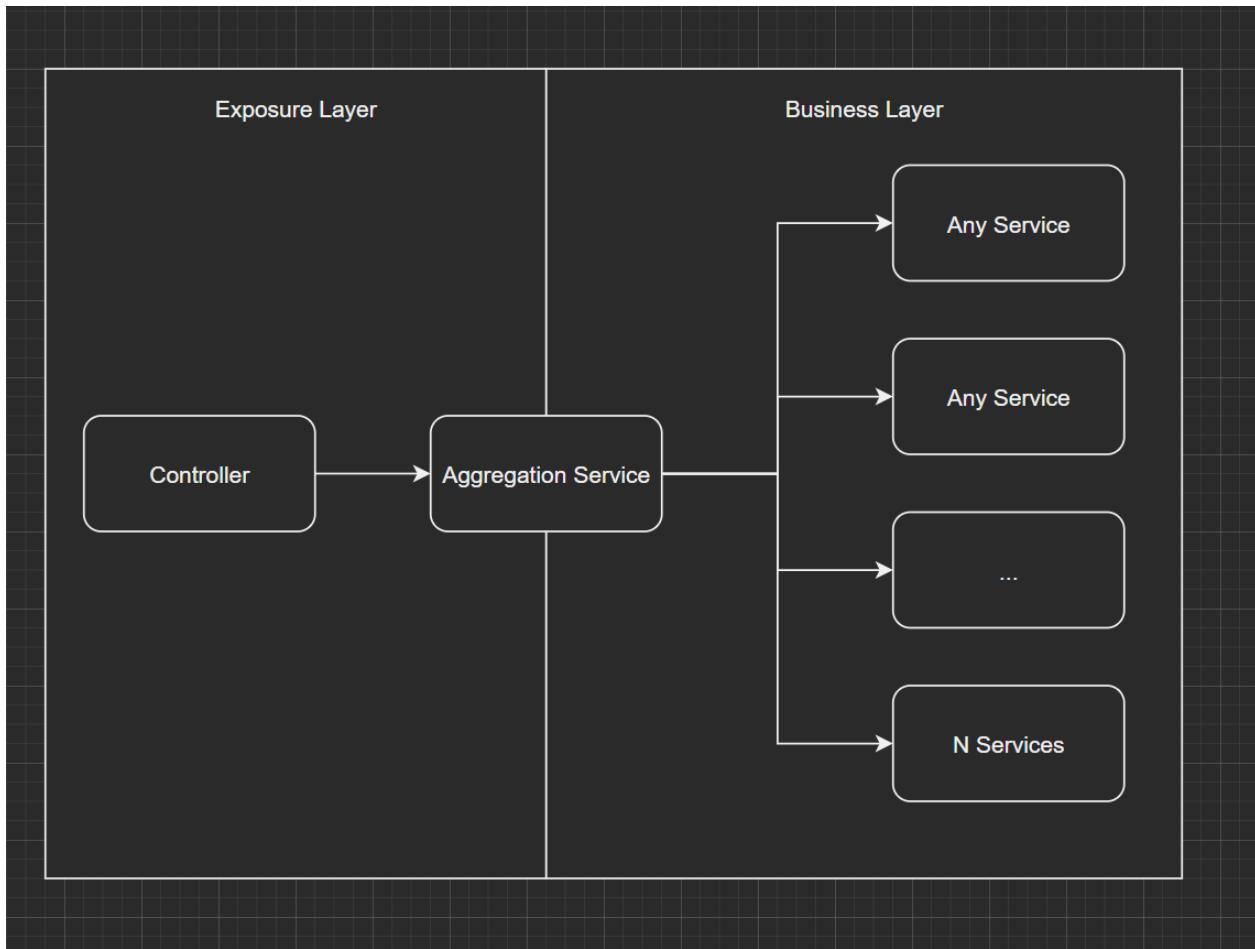
Los servicios de agregación no contienen ninguna lógica de negocio en sí mismos. Son simplemente un nudo que une múltiples servicios de cualquier número. Pueden tener cualquier capa de servicios como dependencias y principalmente expone la llamada a estos servicios en consecuencia. Aquí hay un ejemplo de código de un servicio de agregación:

```
public async ValueTask ProcessStudentAsync(Student student)
{
    await this.studentRegistrationCoordinationService. RegisterStudentAsync(student);
    await this.studentRecordsCoordinationService. AddStudentRecordAsync(student);
    ...
    ...
    await this.anyOtherStudentRelatedCoordinationService. DoSomethingWithStudentAsync(student);
}
```

Como muestra el fragmento anterior, un servicio de agregación puede tener cualquier número de llamadas en cualquier orden sin limitación. Y puede haber ocasiones en las que puede o no necesitar devolver un valor a sus capas de exposición dependiendo del flujo y la arquitectura general, que discutiremos en breve en este capítulo. Pero lo más importante es que los servicios de agregación no deben confundirse con un servicio de orquestación o cualquiera de sus variantes.

2.4.1 En el mapa

Los servicios de agregación siempre se sitúan en el otro extremo de una capa lógica de negocio central. Son el último punto de contacto entre las capas de exposición y las capas lógicas. He aquí una visualización de dónde se ubicarían los servicios de agregación en una arquitectura global:



Vamos a hablar de las características de los servicios de agregación.

2.4.2 Características

Los servicios de agregación existen principalmente cuando hay múltiples servicios, que comparten el mismo contrato o comparten tipos primitivos del mismo contrato, que requieren un único punto de exposición. Existen principalmente en aplicaciones hipercomplejas en las que múltiples servicios (normalmente de orquestación o superiores, pero pueden ser inferiores) requieren un único punto de contacto a través de capas de exposición. Hablemos en detalle de las principales características de los servicios de agregación.

2.4.2.0 No hay limitación de la dependencia

A diferencia de cualquier otro servicio, los servicios de Agregación pueden tener cualquier número de dependencias siempre que estos servicios sean de la misma variación. Por ejemplo, un servicio de Agregación no puede agregarse entre un servicio de Orquestación y un servicio de Coordinación. Se trata de un patrón parcial similar al de Florence, en el que los servicios deben ser de la misma variación, pero no necesariamente limitados por el número.

La razón de la falta de limitación de las dependencias para los servicios de agregación es porque el servicio en sí no realiza ningún nivel de lógica de negocio entre estos servicios. No le importa lo que estos servicios hacen o requieren. Sólo se centra en exponer estos servicios independientemente de lo que se haya llamado antes o después de ellos.

Este es el aspecto de una prueba de servicio de agregación:

```
[Hecho]
public async Task DeberíaProcesarEstudianteAsync()
{
    // dado
    Student randomStudent = CreatedRandomStudent();
    Student inputStudent = randomStudent;

    // cuando
    await this.studentAggregationService. ProcessStudentAsync(inputStudent);

    // entonces this.studentRegistrationCoordinationServiceMock.
    Verify(service =>
        servicio. RegisterStudentAsync(student),
        Times.Once);

    this.studentRecordsCoordinationServiceMock. Verify(service => service.
        AddStudentRecordAsync(student),
        Times.Once);
    ...

    ...
    this.anyOtherStudentRelatedCoordinationServiceMock. Verify(service => service.
        DoSomethingWithStudentAsync(student),
        Times.Once);

    this.studentRegistrationCoordinationServiceMock. VerifyNoOtherCalls();
    this.studentRecordsCoordinationServiceMock. VerifyNoOtherCalls();
    ...
    ...
    this.anyOtherStudentRelatedCoordinationServiceMock. VerifyNoOtherCalls();
}
```

Como puede ver arriba, sólo estamos verificando y probando el aspecto de agregación de la llamada a estos servicios. No se requiere ningún tipo de retorno en este escenario, pero podría haber en los escenarios de pass-through que discutiremos en breve.

Una aplicación de la prueba anterior sería la siguiente:

```
public async ValueTask ProcessStudentAsync(Student student)
{
    await this.studentRegistrationCoordinationService. RegisterStudentAsync(student);
    await this.studentRecordsCoordinationService. AddStudentRecordAsync(student);
    ...
    ...
    await this.anyOtherStudentRelatedCoordinationService. DoSomethingWithStudentAsync(student);
}
```

2.4.2.1 No hay validación de pedidos

Por definición, los servicios de agregación deben llamar naturalmente a varias dependencias sin ninguna limitación. El orden de llamada a estas dependencias tampoco es una preocupación o una responsabilidad para los servicios de agregación. Esto se debe simplemente a que la verificación del orden de llamada se considera una lógica de negocio central. que cae fuera de las responsabilidades de un servicio de Agregación. Eso, por supuesto, incluye tanto el orden natural de verificación como el orden forzado de verificación, como explicamos en la sección 2.3.3.0.1 del capítulo anterior.

Es una violación de la Norma intentar utilizar técnicas simples como una secuencia simulada para probar un servicio de Agregación. Es más probable que estas responsabilidades recaigan en la siguiente capa inferior de un servicio de Agregación para cualquier servicio de tipo orquestación. También es una violación verificar la dependencia del valor de retorno de una llamada de servicio para iniciar una llamada al siguiente.

2.4.2.2 Validaciones básicas

Los servicios de agregación aún deben validar si los datos entrantes son estructuralmente válidos o no. Por ejemplo, un servicio de agregación que toma un objeto `Estudiante` como parámetro de entrada sólo validará si ~~el~~ `estudiante` es nulo o no. Pero ahí se acaba todo.

Puede haber una ocasión en la que una dependencia requiera que se pase una propiedad de un parámetro de entrada, en cuyo caso también se permite validar el valor de esa propiedad estructuralmente. Por ejemplo, si una dependencia descendente requiere que se pase el nombre de un estudiante. Un servicio de agregación sigue `Nombre` que se requiere para validar si `re` es nulo, vacío o sólo un espacio en blanco. `el`

2.4.2.3 Pasar a través de

Los servicios de agregación no están obligados a implementar su agregación realizando múltiples llamadas desde un método. También pueden agregarse ofreciendo métodos de paso para múltiples servicios. Por ejemplo, `temporadas`, `studentCoordinationService` y `studentRecordsService`, `anyOtherStudentRelatedCoordinationService` y cada uno de estos y

Los servicios son independientes en términos de flujo de negocio. Así que una agregación aquí es sólo a nivel de exposición pero no necesariamente a nivel de ejecución.

He aquí un ejemplo de código:

```

public partial class ServicioDeAgregaciónDeEstudiantes
{
    ...
    public async ValueTask< Student> RegisterStudentAsync(Student student)
    {
        ...
        return await this.studentCoordinationService. RegisterStudentAsync(student);
    }

    public async ValueTask< Student> AddStudentRecordAsync(Student student)
    {
        ...
        return await this.studentRecordsCoordinationService. AddStudentRecordAsync(student);
    }

    ...
    ...

    public async ValueTask< Student> DoSomethingWithStudentAsync(Student student)
    {
        ...
        return await this.anyOtherStudentRelatedCoordinationService. DoSomethingWithStudentAsync(student);
    }
}

```

Como se puede ver arriba, cada servicio está utilizando el servicio de agregación como un pass-through. En este escenario no hay necesidad alguna de una llamada de rutina agregada. Este sigue siendo un escenario muy válido para los servicios de agregación.

2.4.2.4 Opcionalidad

Es importante mencionar aquí que los servicios de agregación son opcionales. A diferencia de los servicios de base, los servicios de agregación pueden existir o no en cualquier arquitectura. Los servicios de agregación están ahí para resolver un problema de abstracción. Este problema puede existir o no en función de si la arquitectura requiere un único punto de exposición en la frontera de la capa de lógica de negocio principal o no. Esta responsabilidad única de los servicios de agregación hace que sea mucho más sencillo implementar su tarea y realizar su función fácilmente.

Es más probable que los servicios de agregación sean opcionales que cualquier otro servicio de nivel inferior. Incluso en las aplicaciones más complejas que existen.

2.4.2.5 Agregación a nivel de rutina

Si un servicio de agregación tiene que hacer dos llamadas diferentes de la misma dependencia entre otras llamadas, se recomienda agregar para cada rutina de dependencia. Pero eso es sólo desde la perspectiva de la limpieza del código y no necesariamente afecta a la arquitectura o el resultado final de ninguna manera.

He aquí un ejemplo:

```

public async ValueTask ProcesarEstudiante(Estudiante estudiante)
{
    await this.studentCoordinationService.
        AddStudentAsync(student); await
        ProcessStudentRecordAsync(student);
}

private async ValueTask ProcessStudentRecordAsync(Student student)
{
    await this.studentRecordCoordinationService. AddStudentRecordAsync(student);
    await this.studentRecordCoordinationService. NotifyStudentRecordAdminsAsync(student);
}

```

Esta acción organizativa no justifica ningún tipo de cambio en cuanto a las pruebas o el resultado final, como se ha mencionado anteriormente.

2.4.2.6 Contratos de dependencia puros

La regla/característica más importante de un servicio de agregación es que sus dependencias (a diferencia de los servicios de orquestación) deben compartir el mismo contrato. El parámetro de entrada de una rutina pública en cualquier servicio de agregación debe ser el mismo para todas sus dependencias. Puede haber ocasiones en las que una dependencia pueda requerir un identificador de alumno en lugar de todo el alumno, lo que se permite con precaución siempre que el contrato parcial no sea un tipo de retorno de otra llamada dentro de la misma rutina.

2.4.3 Responsabilidades

La principal responsabilidad de los servicios de agregación es ofrecer un único punto de contacto entre los componentes de exposición y el resto de la lógica empresarial principal. Pero en esencia, la abstracción es el verdadero valor que ofrecen los servicios de agregación para garantizar que cualquier componente empresarial en su conjunto sea enchufable en cualquier sistema, independientemente del estilo de exposición que este mismo sistema pueda necesitar.

Hablemos de estas responsabilidades en detalle.

2.4.3.0 Abstracción

Un servicio de agregación desempeña su responsabilidad con éxito cuando sus clientes o consumidores no tienen idea de lo que hay más allá de las líneas de su implementación. Un servicio de agregación podría combinar 10 servicios diferentes y exponer una única rutina en un escenario de "reutilización".

Pero incluso en los escenarios de paso, los servicios de agregación abstractan cualquier identificación de la dependencia subyacente de los expositores a toda costa. No siempre sucede especialmente en términos de excepciones localizadas, pero lo suficientemente cerca para hacer que la integración parezca como si fuera con un solo servicio que está ofreciendo todas las opciones de forma nativa.

2.4.3.1 Agregación de excepciones

Los servicios de agregación también son similares a los servicios de orquestación en términos de mapeo y agregación de excepciones de dependencias descendentes. Por ejemplo, si `studentCoordinationService` lanza una `StudentCoordinationValidationException`, un servicio de agregación la convertiría en una `StudentAggregationDependencyValidationException`. Esto vuelve a caer en el concepto de desenvolver la excepción y luego envolver las excepciones localizadas de las que hablamos en detalle en la sección 2.3.3.0.2 de este estándar.

3 Expositores

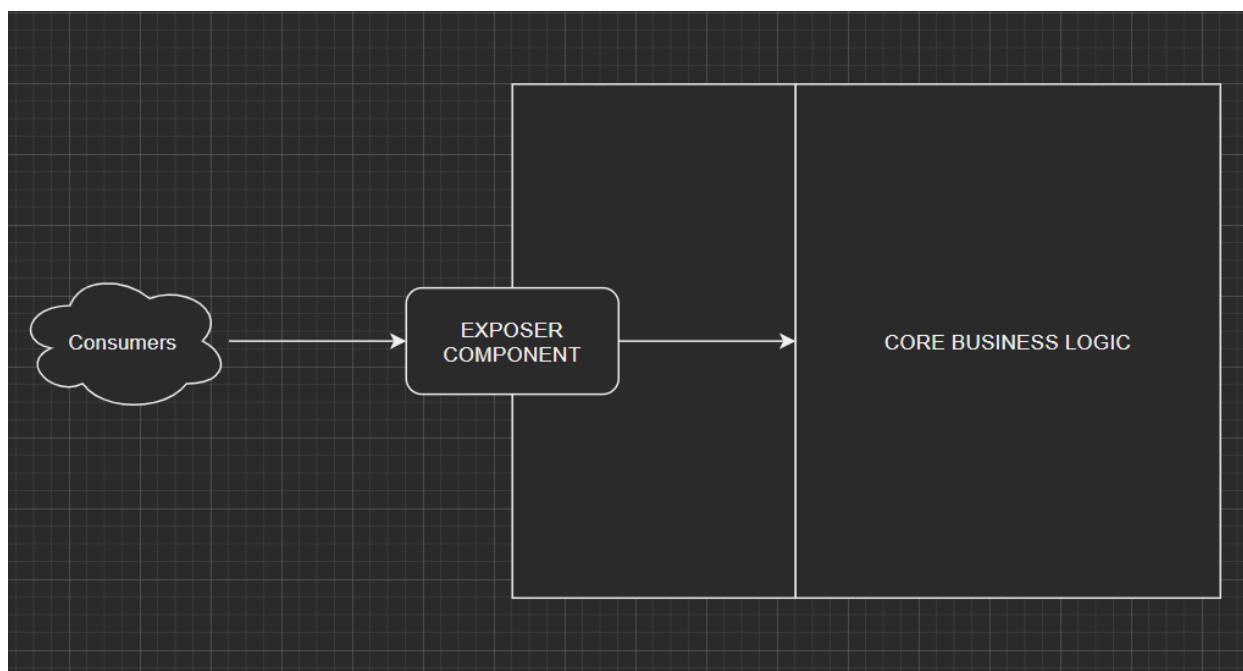
3.0 Introducción

Los expositores son componentes desechables en cualquier sistema que tiene la única responsabilidad de exponer la funcionalidad de su lógica de negocio principal mapeando sus respuestas a un determinado protocolo. Por ejemplo, en las comunicaciones RESTful,

200

un controlador de la API sería responsable de devolver un código para una respuesta satisfactoria. Lo mismo ocurre con otros protocolos como gRPC o SOAP o cualquier otro protocolo de comunicación entre sistemas distribuidos.

Los componentes Exposer son similares a los Brokers. Son el último punto de contacto entre la lógica empresarial central y el mundo exterior. Se construyen con la intención de que se separen del sistema actual en algún momento para permitir que la misma lógica central se integre con sistemas, protocolos o interfaces modernos.



3.0.0 Propósito

En general, la responsabilidad principal de los componentes de exposición es permitir que alguien o algo interactúe con tu lógica de negocio. En ese propósito principal se debe comunicar cautelosamente un mapeo preciso, bit a bit, de cada posible respuesta de tu lógica de negocio principal con el consumidor de esa lógica. Digo cautelosamente porque a veces ciertas cuestiones internas del sistema no deben ser expuestas al mundo exterior. Este mapeo de respuestas normalmente puede ser un effort de cero siempre y cuando el protocolo y el lenguaje con el que se comunica tu lógica de negocio del código sean los mismos como es el caso de las librerías producidas para ejecutarse en el sistema que utilizan las mismas tecnologías o lenguajes de programación.

Pero hay ocasiones en las que el protocolo sin estado del mundo exterior no coincide necesariamente con el mismo valor de una respuesta. En ese caso, se convierte en una responsabilidad del componente expositor hacer un mapeo exitoso en ambos sentidos dentro y fuera del sistema. Los controladores ~~de la~~ API son un gran ejemplo de ello. Ellos comunicar un problema cuando hay una excepción de validación de algún tipo y devuelve un valor JSON deserializado si la comunicación fue exitosa. Pero también hay más detalles en torno a los detalles del problema, los códigos de error y otros niveles de mapeo y comunicación que discutiremos en los próximos capítulos dentro de esta sección.

3.0.0.0 Cartografía pura

El aspecto más importante de los componentes de exposición es que no se les permite comunicarse con corredores de ningún tipo. Y no se les permite contener ninguna forma de lógica de negocio dentro de ellos. Por lógica de negocio me refiero a ninguna secuencia de llamadas de rutina, ninguna iteración o selección/toma de decisiones. Lo mismo ocurre con los brokers, sólo enlazan un reino existente con el exterior para conseguir un determinado valor.

3.0.1 Tipos de componentes de la exposición

Los componentes de exposición tienen tres tipos diferentes. Los cuales son protocolos de comunicación, interfaces de usuario o simplemente una rutina IO. Vamos a hablar breifly de ellos.

3.0.1.0 Protocolos de comunicación

Un componente de exposición que es un protocolo de comunicación puede variar desde simples APIs RESTful, hasta comunicación SOAP o gRPC.

También pueden ser un simple cliente en una biblioteca donde los consumidores simplemente instalarían la biblioteca en sus proyectos y consumirían su lógica principal a través de las APIs del cliente. Todos estos ejemplos son del mismo tipo de componentes de exposición.

El diferenciador aquí es que un protocolo de comunicación suele estar basado en eventos. Se desencadena por una comunicación entrante y se trata con una respuesta de cualquier tipo. Los protocolos de comunicación suelen ser para las integraciones de sistema a sistema, pero pueden ser accesibles y comprensibles por los seres humanos para fines de prueba y depuración.

3.0.1.1 Interfaces de usuario

Otro tipo de componentes del expositor son las interfaces de usuario. Pueden ser desde aplicaciones web, móviles o de escritorio hasta simples líneas de comando. Se dirigen principalmente a los usuarios finales para comunicarse, pero pueden ser automatizadas por otros sistemas. En la actualidad, las interfaces de usuario también pueden incluir realidades virtuales y aumentadas, metaversos y cualquier otra forma de software.

Hay ocasiones en las que las interfaces hombre-máquina (HMI) también pueden entrar en ese nivel de componentes de exposición. Por ejemplo, los botones de un teléfono móvil, los teclados que utilizamos a diario y cualquier forma de hardware que pueda interactuar directamente con las interfaces de la lógica empresarial principal como componente de exposición. La misma teoría se aplica a los componentes del Internet de las Cosas (IoT) y a muchos otros en los que un humano tiene que utilizar un componente para aprovechar una determinada capacidad en su propio beneficio de alguna manera.

3.0.1.2 Componentes de E/S

Algunos componentes de la exposición no son necesariamente un sistema de interfaz con otro sistema. Tampoco están pensados para comunicarse con los humanos. Son daemons o componentes basados en IO que hacen algo en segundo plano sin un disparador. normalmente estos componentes se basan en el tiempo y pueden aprovechar los protocolos existentes o simplemente interactuar directamente con la lógica empresarial principal, que son ambas opciones viables.

3.0.2 Punto de contacto único

Los componentes de exposición sólo pueden comunicarse con uno y sólo un servicio. La integración con múltiples servicios convertiría a un componente de exposición en servicios de orquestación o agregación, que no pueden existir como lógica central en ese ámbito de exposición.

La regla del punto de contacto único también garantiza la facilidad de disposición del propio componente de exposición. Garantiza que la integración sea lo suficientemente sencilla y con dependencias controladas (sólo una) como para que pueda reconectarse a prácticamente cualquier protocolo en cualquier momento con el menor coste posible.

3.0.3 Ejemplos

Tomemos como ejemplo los controladores de la API para un componente de exposición del mundo real en cualquier sistema.

```
[HttpPost]
public async ValueTask< ActionResult< Student>> PostStudentAsync(Student student)
{
    pru
    ebe
    con Alumno registrado =
    {
        await this.studentService. RegisterStudentAsync(student);

        return Created(registeredStudent);

    }
    catch (StudentValidationException studentValidationException)
        cuando (studentValidationException.InnerException es AlreadyExistsStudentException)
    {
        return Conflict(studentValidationException.InnerException);
    }
    catch (StudentValidationException studentValidationException)
    {
        return BadRequest(studentValidationException.InnerException);
    }
    catch (StudentDependencyException studentDependencyException)
    {
        return InternalServerError(studentDependencyException);
    }
    catch (StudentServiceException studentServiceException)
    {
        return InternalServerError(studentServiceException);
    }
}
```

El fragmento de código anterior es para un método de la API que ~~de~~ un modelo de estudiante en la lógica de negocio central de un sistema de escolarización (OtripleS). En una tecnología como ASP.NET, los controladores se encargan de gestionar el mapeo de la petición JSON entrante en el modelo ~~del~~ modelo para que el controlador pueda utilizar ese modelo con un sistema integrado.

Sin embargo, también verá que el código del controlador intenta mapear cada posible excepción categórica en su respectivo protocolo REST. Esto es sólo un simple fragmento para mostrar cómo puede ser un componente de exposición. Pero hablaremos más sobre las reglas y condiciones de los

controladores en el próximo capítulo de La Norma.

3.0.4 Resumen

En resumen, los componentes de exposición son una capa muy fina que no contiene ninguna inteligencia o lógica en ella. no está pensada para orquestar, o llamar a múltiples servicios de lógica de negocio. Y sólo se centra en el aspecto de mapeo dúplex de la comunicación entre un sistema y otro.

3.1 Protocolos de comunicación

3.1.0 Introducción

En el ámbito de la exposición, una de las metodologías más comunes para construir una estructura de comunicación entre varios sistemas es utilizar un protocolo de comunicación. Estos protocolos han evolucionado a lo largo de los años, desde SOAP hasta REST, pasando por tantos otros protocolos y principios de comunicación que han manifestado sus propias tecnologías para lograr la exposición de APIs a sistemas distribuidos.

En el mundo .NET, la tecnología ha evolucionado con la evolución de la arquitectura desde SOA con WCF hasta Microservicios con REST. La evolución continúa pero los principios cambian con menos frecuencia. En estos próximos capítulos hablaremos de los protocolos de comunicación más comunes con una forma estandarizada de implementarlos para una aplicación de nivel empresarial.

3.1.0.0 Principios y normas

Los protocolos de comunicación son necesarios para lograr dos cosas cuando se integran con la lógica empresarial principal. La comunicación de resultados y la notificación de errores. Vamos a hablar brevemente de ellos:

3.1.0.0.0 Comunicación de resultados

Cualquier protocolo de comunicación debe satisfacer el principio de devolver el resultado de la lógica del negocio. Este resultado puede ser serializado en algún tipo de JSON

lenguaje como o simplemente comunicarse tal cual. En el caso de la API no suele ser necesario serializar y deserializar los datos. Pero esto conlleva la limitación de que solo las tecnologías que pueden integrarse con estas bibliotecas pueden beneficiarse de ellas.

La comunicación de los resultados también puede encapsularse con algún tipo de estado.

En el caso de las comunicaciones de la API RESTful, un puede acompañar al código

devuelve el resultado serializado resultado. Estos códigos permiten a los consumidores

entender el siguiente curso de acción. Algunos resultados 2xx pueden requerir una acción retardada si la respuesta es apenas Aceptada pero no necesariamente Creada.

3.1.0.0.1 Informes de errores

Si se espera que la lógica de negocio central proporcione un informe detallado de todos los errores de validación, por ejemplo, que se produjeron en una solicitud particular. Es responsabilidad de los protocolos de comunicación representar estos informes de error, ya sea en su forma original, o serializar el informe en un lenguaje que sea fácilmente deserializable y convertible de nuevo en la forma original de Excepción en el lado del cliente.

Los informes de error también deberán tener sus propios códigos en caso de que se serialicen para que el cliente sepa cuál es el siguiente curso de acción. Es muy recomendable seguir una forma estandarizada de comunicar los errores con documentación, preferiblemente para ayudar a guiar a los consumidores a desarrollar los mejores clientes para estas APIs.

3.1.0.1 Tipos comunes

Exploraremos algunos de los tipos más comunes de protocolos de comunicación en esta sección.

3.1.0.1.0 REST

En el momento de redactar esta norma, las APIs RESTful son la forma más común de comunicación entre sistemas distribuidos. REST es un protocolo de transferencia de estado representativo con ciertas restricciones que definen específicamente la forma de comunicación, la notificación de errores y su propia naturaleza sin estado. Las APIs RESTful son agnósticas a la tecnología. Pueden ser implementadas por cualquier tecnología o lenguaje de programación, pero permiten que estas tecnologías se comuniquen entre sí sin ningún tipo de dependencia del servidor o de la tecnología elegida por el cliente.

3.1.0.1.1 Bibliotecas

El otro protocolo de comunicación más común son las APIs implementadas dentro de las bibliotecas. Por ejemplo, los paquetes nuget son bibliotecas publicadas y distribuidas que permiten a los desarrolladores aprovechar una lógica empresarial central localizada o comunicarse con un recurso externo para lograr un determinado objetivo.

3.1.0.1.2 Otros tipos

Existen otros tipos de protocolos de comunicación. Algunos de ellos son más antiguos y otros están a punto de presentarse en la industria del software. Estos tipos son como SOAP con manifestaciones como WCF y gRPC, GraphQL y varios otros protocolos.

Nos centraremos principalmente en las APIs RESTful como protocolos de comunicación más comunes con un breve toque en las Bibliotecas. Y a medida que evolucionemos y aprendamos más, también lo hará nuestro Estándar, que incluirá más y más protocolos de comunicación diferentes y evolucionará también en términos de patrones.

Empecemos con las APIs RESTful como protocolo de comunicación y profundicemos en el diferente aspecto de ese componente expositor.

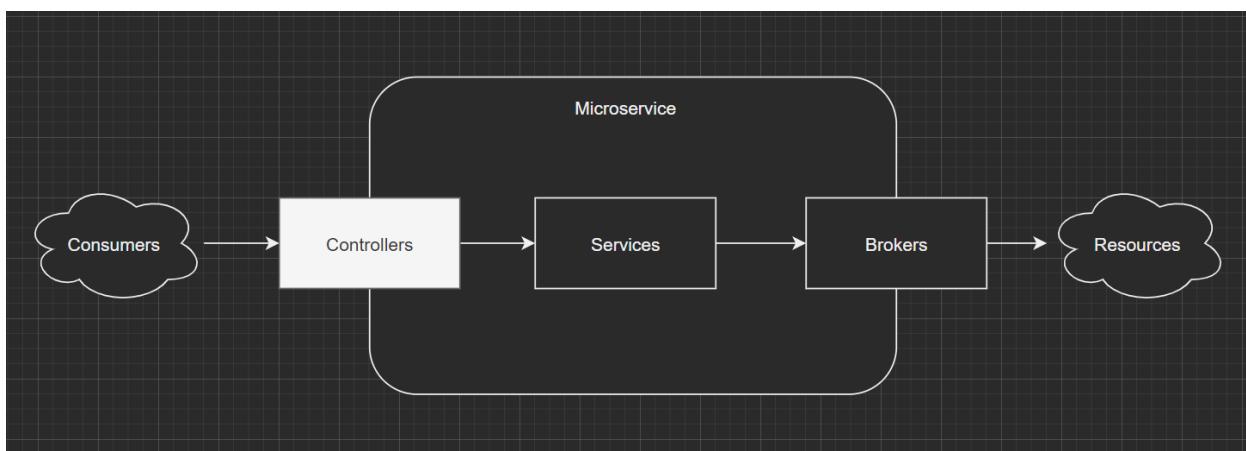
3.1.1 APIs RESTful

3.1.1.0 Introducción

Los controladores de la API RESTful son un enlace entre la capa lógica del negocio principal y el mundo exterior. Se sitúan al otro lado del ámbito del negocio principal de cualquier aplicación. En cierto modo, los controladores de API son como los corredores. Aseguran una integración exitosa entre nuestra lógica central y el resto del mundo.

3.1.1.1 En el mapa

Los controladores se sitúan en el extremo de cualquier sistema. Independientemente de que este sistema sea una plataforma monolítica o un simple microservicio. Los controladores de API hoy en día se aplican incluso a lambdas más pequeñas o funciones en la nube. Desempeñan el papel de activador para acceder a estos recursos en cualquier sistema a través de REST.



El lado del consumidor de los controladores puede variar. En los sistemas de producción, estos consumidores pueden ser otros servicios que requieren datos de un punto final de la API en particular. Pueden ser bibliotecas construidas como envolturas alrededor de las APIs del controlador para proporcionar un recurso local con datos externos. Pero los consumidores también pueden ser simplemente ingenieros probando puntos finales, validando sus comportamientos a través de documentos swagger.

3.1.1.2 Características

Hay varias reglas y principios que rigen la implementación de los puntos finales de la API RESTful. Vamos a discutirlos aquí.

3.1.1.2.0 Idioma

Los controladores hablan un lenguaje diferente cuando se trata de implementar sus métodos en comparación con los servicios y corredores. Por ejemplo, si un broker que interactúa con un almacenamiento utiliza un lenguaje como `InsertStudentAsync`, y su correspondiente implementación de servicio utiliza algo como `AddStudentAsync`, el controlador equivalente utilizará un lenguaje RESTful como `PostStudentAsync`.

Sólo hay un puñado de terminologías que un controlador utilizaría para expresar una determinada operación. Vamos a dibujar el mapa aquí para mayor claridad:

Controladores	Servicios	Corredores de bolsa
Publicar	Añadir	Insertar
Obtenga	Recuperar	Selección
Poner	Modificar	Actualización
Borrar	Eliminar	Borrar

El lenguaje que hablan los controladores se llama Http Verbs. Su alcance es más amplio que las operaciones CRUD antes mencionadas. Por ejemplo, existe PATCH que permite a los consumidores de la API actualizar sólo partes de un documento concreto. PATCH es raramente usado hoy en día desde mi experiencia en aplicaciones producidas. Pero puede que especialice una sección especial para ellos en algún momento en futuras versiones de The Standard.

3.1.1.2.0.0 Más allá de las rutinas CRUD

Pero como hemos mencionado antes, los controladores pueden interactuar con algo más que un servicio de la fundación. Pueden interactuar con la lógica de negocio de orden superior función. Por ejemplo, un servicio de procesamiento puede offer una `Upsert` rutina. En en cuyo caso un Verbo Http típico no podría satisfacer una rutina combinatoria como un `Upsert`. En cuyo caso resolvemos al estado inicial `Post` asumiendo que el recurso no existe.

Puede llegar a ser útil notificar a nuestros consumidores si decidimos modificar en lugar de añadir la operación por la que nos decidimos. Pero eso es una implementación caso por caso y, más a menudo que nunca, a los consumidores no les interesa aprender esa pieza de información. La misma idea se aplica a otros lenguajes los servicios que no son de la fundación pueden utilizar. Por ejemplo `Proceso` o `Calcular` o cualquier

otros servicios de orden superior o hiperavanzado específicos de la empresa pueden elegir.

3.1.1.2.0.1 Verbos similares

A veces, especialmente con operaciones CRUD básicas, necesitarás el mismo verbo Http para describir dos rutinas diferentes. Por ejemplo, la integración con RetrieveById y RecuperarTo ambos se resuelven en un obteng a la operación en el ámbito RESTful. En este caso, cada función tendrá un nombre diferente, pero mantendrá el mismo verbo, como se indica a continuación:

```
[HttpGet]
public ActionResult< IQueryable< Student>> GetAllStudents()
{
    ...
}

[HttpGet("{studentId}")]
public async ValueTask< ActionResult< Student>> GetStudentByIdAsync(Guid studentId)
{
    ...
}
```

Como puedes ver arriba, el diferenciador aquí es tanto el nombre de la GetAllStudents que apunta a GetStudentByIdAsync como la Ruta al mismo tiempo.

También la capacidad de implementar múltiples rutinas con nombres diferentes aunque resuelvan al mismo Verbo Http.

3.1.1.2.0.2 Convenciones de las rutas

Los controladores de la API RESTful son accesibles a través de rutas. Una ruta es simplemente una url que se utiliza combinada con un verbo Http para que el sistema sepa a qué rutina debe llamar para que coincida con esa ruta. Por ejemplo, si necesito recuperar un estudiante

con Id entonces mi ruta api sería la siguiente: api/estudiantes/123. Y si quiero recuperar todos los estudiantes en algún sistema, podría simplemente llamar a api/

estudiantes con GET verbo.

3.1.1.2.0.2.0 Rutas del controlador

La clase controladora en una aplicación ASP.NET simple puede ser simplemente configurada en la parte superior de la declaración de la clase controladora con una decoración como la siguiente:

```
[ApiController]
[Route("api/[controller]")]
public class
EstudiantesController
{
    ...
}
```

La ruta allí es una plantilla que define el punto final para comenzar con api y se traza omitiendo el término "Controlador" del nombre de la clase. Así que EstudiantesC

acabaría siendo `api/estudiantes`. Es importante que todos los controladores tengan una versión plural del contrato que están sirviendo. A diferencia de los servicios donde decimos `StudentService` los controladores serían la versión plural con `StudentsController`.

3.1.1.2.0.2.1 Rutas de rutina

La misma idea se aplica a los métodos dentro de la clase del controlador. Como decimos en el

fragmento de código anterior, hemos decorado `GetByIdAsync` con un `HttpGet` con una ruta particular identificada para añadirla a la ruta general del controlador existente. Por ejemplo, si la ruta del controlador es `api/students` una rutina con `HttpGet("{studentId}")` daría como resultado una ruta con el siguiente aspecto: `api/students/{studentId}`.

El `studentId` entonces se mapearía como una variable de parámetro de entrada

que *debe* coincidir con la variable definida en la ruta como sigue:

```
[HttpGet("{studentId}")]
public async ValueTask< ActionResult< Student>> GetStudentByIdAsync(Guid studentId)
{
    ...
}
```

Pero a veces estas rutas no son sólo parámetros de url. A veces contienen una petición dentro de ellas. Por ejemplo, digamos que queremos contabilizar una tarjeta de la biblioteca contra un registro de estudiante en particular. Nuestro punto final sería `api/students/{studentId}/librarycards` PO algo así:

con un ST verbo. En este caso tenemos que distinguir entre estos dos parámetros de entrada con una nomenclatura adecuada como la siguiente:

```
[HttpPost("{studentId}/librarycards")]
public async ValueTask< ActionResult< LibraryCard>> PostLibraryCardAsync(Guid studentId, LibraryCard libraryCard)
{
    ...
}
```

3.1.1.2.0.2.2 Plural Singular Plural

Al definir rutas en una API RESTful, es importante seguir las convenciones de nomenclatura global para estas rutas. La regla general es acceder a una colección de recursos, luego apuntar a una entidad en particular, luego nuevamente acceder a una colección de recursos dentro de esa entidad y así sucesivamente. Por ejemplo, en el ejemplo del carné de biblioteca `api/students/{studentId}/librarycards/{librarycardId}` se puede ver que empezamos accediendo a todos los estudiantes, luego nos dirigimos a un estudiante con un id particular, luego queremos acceder a todos los carnés de biblioteca asociados a ese estudiante y luego nos dirigimos a un carné muy particular haciendo referencia a su id.

Esta convención funciona perfectamente en las relaciones de uno a muchos. ¿Pero qué pasa con las relaciones uno a uno? Supongamos que un estudiante puede tener una y sólo una tarjeta de biblioteca en todo momento. En ese caso, nuestra ruta seguiría siendo algo así:

`api/students/{studentId}/librarycards` con un

verbo, y se produciría un error como CONFLICTO si ya existe una tarjeta independientemente de que las identidades coincidan o no.

3.1.1.2.0.2.2 Parámetros de consulta y OData

Pero la ruta que recomiendo es la ruta flat-model. Donde cada recurso vive por sí mismo con sus propias rutas únicas. En nuestro caso aquí tirando de una tarjeta de la biblioteca para un estudiante en particular sería como sigue: studentId={studentId} o simplemente utilizar una tecnología global un poco avanzada como OData donde la consulta sería simplemente

?\$filter=studentId eq '123'.

Este es un ejemplo de implementación de parámetros de consulta básicos:

```
[HttpPost()]
public async ValueTask< ActionResult< LibraryCard>> PostLibraryCardAsync(Guid studentId, LibraryCard libraryCard)
{
    ...
}
```

En el lado de OData, una implementación sería la siguiente:

```
[HttpGet]
[EnableQuery]
public async ValueTask< IQueryable< LibraryCard>> GetAllLibraryCards()
{
    ...
}
```

La misma idea se aplica a POST para un modelo. en lugar de publicar `api:/students/{studentId}/librarycards` - podemos aprovechar el propio contrato para publicar contra `api/librarycards` con un modelo que contenga el id del estudiante dentro. Esta idea de flat-route puede simplificar la implementación y se alinea perfectamente con el tema general de The Standard. Mantener las cosas simples.

3.1.1.2.1 Códigos y respuestas

Las respuestas de un controlador de la API deben ser mapeadas hacia códigos y respuestas. Por ejemplo, si intentamos añadir un nuevo alumno a una escolarización sistema. Vamos a estudiante y en la reejecución recibimos el mismo cuerpo que enviamos con un código 201 lo que significa que la resolución tiene de estado Creado.

Hay tres categorías principales en las que pueden caer las respuestas. La primera es la categoría de éxito. Donde tanto el usuario como el servidor han hecho su parte y la petición ha tenido éxito. La segunda categoría es la de los códigos de error del usuario, cuando la solicitud del usuario tiene un problema de cualquier tipo. En este caso, un código será devuelto con un mensaje de error detallado para ayudar a los usuarios a fijar sus solicitudes para realizar una operación exitosa. El tercer caso es el de los Códigos de Error del Sistema, donde el sistema se ha encontrado con

un problema de cualquier tipo interno o externo y necesita comunicar un 5xx código para indicar al usuario que algo internamente han ido mal con el sistema y tienen que ponerse en contacto con el servicio de asistencia.

Hablemos aquí de esos códigos y de sus escenarios en detalle.

3.1.1.2.1.0 Códigos de éxito (2xx)

Los códigos de éxito indican que un recurso ha sido creado, actualizado, eliminado o recuperado. Y en algunos casos indica que una solicitud ha sido enviada con éxito de forma eventual que puede o no tener éxito en el futuro. Aquí están los detalles de cada uno:

Código	Método	Detalles
200	Ok	Se utiliza para las operaciones GET, PUT y DELETE exitosas.
201	Creado	Se utiliza para las operaciones POST con éxito
202	Aceptado	Se utiliza para la solicitud que fue delegada pero puede o no tener éxito

He aquí algunos ejemplos de cada uno de ellos:

En un escenario de recuperación sin post, es más befitting devolver un como sigue: Ok código de situación

```
[HttpGet("{studentId}")]
public async ValueTask< ActionResult< Student>> GetStudentByIdAsync(Guid studentId)
{
    Estudiante recuperadoEstudiante =
        await this.studentService. RetrieveStudentByIdAsync(studentId);

    devolver Ok(recuperadoEstudiante);
}
```

Pero en un escenario donde tenemos que crear un recurso, un befitting para este caso como el siguiente: Creado es más

```
[HttpPost]
public async ValueTask< ActionResult< Student>> PostStudentAsync(Student student)
{
    Estudiante recuperadoEstudiante =
        await this.studentService. AddStudentAsync(student);

    devolver Creado(alumno);
}
```

En los casos de consistencia eventual, en los que un recurso contabilizado no es realmente persistente todavía, ponemos en cola la solicitud y devolvemos un proceso se iniciará: Aceptado para indicar un

```
[HttpPost]
public async ValueTask< ActionResult< Student>> PostStudentAsync(Student student)
{
    Estudiante recuperadoEstudiante =
        await this.studentEventService. EnqueueStudentEventAsync(student);

    devolver Aceptado(estudiante);
}
```

La regla estándar para eventuales escenarios de consistencia es asegurar que el remitente tenga un token de algún tipo para que los solicitantes puedan consultar el estado de su solicitud con una llamada diferente a la API. Discutiremos estos patrones en un libro diferente llamado La Arquitectura Estándar.

3.1.1.2.1.1 Códigos de error del usuario (4xx)

Esta es la segunda categoría de respuestas de la API. Cuando una solicitud de usuario tiene un problema y el sistema debe ayudar al usuario a entender por qué su solicitud no ha tenido éxito. Por ejemplo, supongamos que un cliente está enviando una nueva estudiante a un sistema de escolarización. Si la identificación del estudiante no es válida a

400 Bad
o
Solicitud debe ser devuelto con un detalle del problema que explique cuál es exactamente el motivo del fallo de la solicitud.

Los controladores son responsables de asignar las excepciones categóricas de la capa central a los códigos de estado adecuados. He aquí un ejemplo:

```
[HttpGet("{studentId}")]
public async ValueTask< ActionResult>> GetStudentByIdAsync(Guid studentId)
{
    pru
    ebe
    con ...
    {

    }
    catch (StudentValidationException studentValidationException)
    {
        return BadRequest(studentValidationException.InnerException)
    }
}
```

Así que, como se muestra en este fragmento de código, capturamos una validación categórica

y lo ha convertido en un código de error que es BadRequest. El acceso a la excepción interna aquí es con el fin de extraer un detalle de la propiedad InnerException de la excepción interna que contiene toda la información del informe de errores.

Pero a veces los controladores tienen que profundizar. Atrapar una excepción local particular, no sólo la categórica. Por ejemplo, digamos que queremos manejar StudentException con un código de 404 o NotFound. Así es como error lo lograríamos:

```
[HttpGet("{studentId}")]
public async ValueTask< ActionResult>> GetStudentByIdAsync(Guid studentId)
{
    pru
    ebe
    con ...
    {

    }
    catch (StudentValidationException studentValidationException)
        (cuando studentValidationException.InnerException es NotFoundStudentException)
    {
        return NotFound(studentValidationException.InnerException)
    }
}
```

En el fragmento de código anterior, tuvimos que examinar el tipo de excepción interior para validar la excepción localizada desde dentro. Esta es la ventaja del proceso de desenvolver y envolver que discutimos en la sección 2.3.3.0.2 de La Norma. El controlador puede examinar múltiples tipos

dentro del mismo bloque, como se indica a continuación:

```

...
    catch (StudentCoordinationDependencyValidationException studentCoordinationDependencyValidationException)
        (cuando studentValidationException. InnerException
            es NotFoundStudentException
            o NotFoundLibraryCardException
            o NotFoundStudentContactException)
    {
        return NotFound(studentValidationException. InnerException)
    }
...

```

Teniendo esto en cuenta, vamos a detallar las correspondencias más comunes de las excepciones a los códigos:

Código	Método	Excepción
400	BadRequest	ValidationException o DependencyValidationException
404	NotFound	NotFoundException
409	Conflict	AlreadyExistException
423	Bloqueado	LockedException
424	FailedDependency	InvalidReferenceException

Hay más 4xx códigos de estado por ahí. Pero a partir de este momento puede ser generado automáticamente por el marco de trabajo de la web, como en ASP.NET

o todavía no hay escenarios útiles para ellos. Por ejemplo, un 401 o Unauthorized puede generarse automáticamente si el punto final del controlador está decorado con el requisito de autorización.

3.1.1.2.1.2 Códigos de error del sistema (5xx)

Los códigos de error del sistema son el tercer y último tipo posible de códigos que pueden ocurrir o ser devueltos desde un punto final de la API. Su principal responsabilidad es indicar en general que el consumidor del punto final de la API no tiene ningún fallo.

Algo malo ha sucedido en el sistema, y el equipo de ingeniería debe involucrarse para resolver el problema. Por eso registramos nuestras excepciones con un nivel de gravedad en la capa de lógica empresarial principal, para saber lo urgente que puede ser el asunto.

El código http más común que se puede comunicar en un asunto del lado del servidor

es el 500 o InternalServerError código. Veamos un código que trata de esta situación:

```

[HttpGet("{studentId}")]
public async ValueTask< ActionResult< Student>> GetStudentByIdAsync(Guid studentId)
{
    pru
    ebe
    con ...
    {

    }
    ...
    catch (StudentDependencyException studentDependencyException)
    {
        return InternalServerError(studentValidationException)
    }
}

```

En el fragmento anterior ignoramos completamente la excepción interna y nos centramos principalmente en la excepción categórica por razones de seguridad. Principalmente para no permitir que la información interna del servidor sea expuesta ~~en una respuesta de la API que no sea algo tan simple como~~
~~póngase en contacto con el servicio de asistencia.~~
Dado que el consumidor de la API no tiene que realizar ninguna acción más que crear un ticket para el equipo de soporte.

Idealmente, estos problemas deberían ser capturados fuera de las Pruebas de Aceptación que discutiremos en breve en este capítulo. Pero hay veces en las que hay un fallo en el servidor que puede causar una fuga de memoria de algún tipo o cualquier otro problema de infraestructura interna que no será capturado por las pruebas de extremo a extremo de ninguna manera.

En términos de tipos de excepciones que pueden ser manejados, es un poco más pequeño cuando se trata de error del servidor aquí está el detalle:

Código	Método	Excepción
500	InternalServerError	DependencyException o ServiceException
507	NotFound	InsufficientStorageException (sólo interno)

También hay un caso interesante en el que dos equipos se ponen de acuerdo en un determinado documento swagger, y el equipo de desarrollo de la API back-end decide construir los correspondientes puntos finales de la API con métodos que aún no están implementados para comunicar al otro equipo que el trabajo aún no ha comenzado. En este caso ~~NotImplemen~~ caso utilizando el código de error ~~es suficientes que es sólo un código para~~ ~~ted.~~

También es importante mencionar que el código de error nativo 500 puede ser comunicado en las aplicaciones ASP.NET a través del método ~~Problem~~. Estamos

~~confiar en una biblioteca FullSense para proporcionar más códigos que el nativo implementación puede offer, pero lo más importante es proporcionar una opción de serialización y deserialización de detalles de problemas en el lado del cliente.~~

3.1.1.2.1.3 Todos los códigos

Aparte de los mencionados en las secciones anteriores, y para la documentación propósitos, aquí está el todo ~~4xx y 5xx~~ códigos que una API podría de la ~~comunicar~~ de acuerdo con las últimas directrices estandarizadas de la API:

Estatus	Código
BadRequest	400
No autorizado	401
PagoRequerido	402
Prohibido	403
NotFound	404
NotFound	404
MethodNotAllowed	405
NotAcceptable	406
ProxyAuthenticationRequired	407
RequestTimeout	408
Conflict	409
Gone	410
LongitudRequerida	411
PreconditionFailed	412
RequestEntityTooLarge	413
RequestUriTooLong	414
UnsupportedMediaType	415
RequestedRangeNotSatisfiable	416
ExpectativaFallida	417
MisdirectedRequest	421
UnprocessableEntity	422
Bloqueado	423
FailedDependency	424
ActualizaciónRequerida	426
PrecondiciónRequerida	428
TooManyRequests	429
RequestHeaderFieldsTooLarge	431
No disponible por razones legales	451
InternalServerError	500
NotImplemented	501
BadGateway	502
Servicio no disponible	503
GatewayTimeout	504
HttpVersionNotSupported	505
VariantAlsoNegotiates	506
InsufficientStorage	507
LoopDetected	508
NotExtended	510
NetworkAuthenticationRequired	511

Estudiaremos la posibilidad de incorporar algunos de estos códigos en futuras revisiones de la Norma, según sea necesario.

3.1.1.2.2 Dependencia única

Los componentes del Exposer pueden tener una y sólo una dependencia. Esta dependencia debe ser un componente de Servicio. No puede ser un Broker o cualquier otra dependencia nativa que los Brokers puedan utilizar para extraer configuraciones o cualquier otro tipo de dependencias.

Cuando se implementa un controlador, el constructor puede ser implementado como sigue:

```
[ApiController]
[Route("api/{controller}")]
public class EstudiantesControlador : RESTfulControlador
{
    private readonly IStudentService studentService;

    public StudentsController(IStudentService studentService) =>
        this.studentService = studentService;

    ...
}
```

3.1.1.2.3 Contrato único

Esta característica viene de fábrica con la regla de dependencia única. Si los servicios sólo pueden servir y recibir un contrato, la misma regla se aplicará a los controladores. Pueden devolver un contrato, una lista de objetos con el mismo contrato o una parte del contrato al pasarles Ids o consultas.

3.1.1.3 Organización

Los controladores deben estar ubicados en la carpeta `src/Controllers` y pertenecer a una carpeta `Controlador` espacio de nombres. Los controladores no necesitan tener sus propias carpetas o espacios de nombres ya que realizan una simple tarea de exposición y eso es todo.

Este es un ejemplo de un espacio de nombres de controlador:

```
namespace GitFyle.Core.Api.Controllers
{
    [ApiController]
    [Route("api/{controller}")]
    public class ContributionsController : RESTfulController
    {
        ...
    }
}
```

3.1.1.4 Controlador de casa

Cada sistema debe implementar un punto final de la API que lanza el controlador. La única responsabilidad del controlador es devolver un mensaje simple para indicar que la API sigue viva. He aquí un ejemplo:

```

utilizando Microsoft.AspNetCore.Mvc;

namespace OtripleS.Web.Api.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class HomeController : ControllerBase
    {
        [HttpGet]
        public ActionResult< string> Get() =>
            Ok("¡Hola Mario, la princesa está en otro castillo!");
    }
}

```

Los controladores domésticos no están obligados a tener ninguna seguridad en ellos. Abren una puerta para que las pruebas de latidos del corazón aseguren que el sistema como entidad está funcionando sin comprobar ninguna dependencia externa. Esta práctica es muy importante para ayudar a los ingenieros a saber cuándo se cae el sistema y actuar rápidamente en consecuencia.

3.1.1.5 Pruebas

Los controladores pueden ser potencialmente probados por la unidad para verificar el mapeo de las excepciones a los códigos de error están en su lugar. Pero ese no es un patrón que yo haya seguido hasta ahora. Sin embargo, lo que es más importante son las pruebas de aceptación. Las cuales verifican que todos los componentes del sistema están completamente y exitosamente integrados entre sí.

Este es un ejemplo de prueba de aceptación:

```

[Hecho]
public async Task DeberíaPublicarEstudianteAsync()
{
    // dado
    Estudiante randomStudent = CreateRandomStudent();
    Estudiante inputStudent = randomStudent;
    Estudiante esperadoEstudiante = Estudiante de entrada;

    // cuando
    await this.oTRIPLESApiBroker. PostStudentAsync(inputStudent);

    Estudiante realStudent =
        await this.oTRIPLESApiBroker. GetStudentByIdAsync(inputStudent.Id);

    // entonces actualStudent. Debería().
    BeEquivalentTo(expectedStudent);
    await this.oTRIPLESApiBroker. DeleteStudentByIdAsync(actualStudent.Id);
}

```

Las pruebas de aceptación deben cubrir todos los puntos finales disponibles en un controlador. También son responsables de la limpieza de los datos de la prueba después de que ésta se haya completado. También es importante mencionar que los recursos que no son propiedad del microservicio, como la base de datos, deben ser emulados con aplicaciones como y muchos otros.

Las pruebas de aceptación también se implementan a posteriori, a diferencia de las pruebas unitarias. Un punto final tiene que estar totalmente integrado y ser funcional antes de escribir una prueba para garantizar el éxito de la implementación.

[Pruebas de aceptación (Parte 1)]

[Pruebas de aceptación (parte 2)]

3.2 Interfaces de usuario

3.2.0 Introducción

Las interfaces de usuario o UI son un tipo de componentes de exposición que se dirigen principalmente a los seres humanos para la interacción con la capa de negocio principal. A diferencia de los protocolos de comunicación, que son principalmente para los sistemas distribuidos. Las interfaces de usuario evolucionan constantemente en términos de tecnologías y metodologías con las que los humanos pueden interactuar con cualquier sistema. Esto va desde las aplicaciones web hasta las realidades virtuales/aumentadas. Sistemas activados por voz y, más recientemente, por ondas cerebrales.

El desarrollo de interfaces de usuario puede ser mucho más difícil en términos de experiencias. Hoy en día no existe un estándar global sobre lo que es una experiencia intuitiva. Depende en gran medida de la cultura, de los puntos comunes y de muchas otras variables que cambian constantemente en el mundo actual. Este estándar esbozará los principios y reglas para construir componentes de interfaz de usuario modulares, mantenibles y conectables. Pero habrá una norma diferente para definir las experiencias de los usuarios, las interacciones humanas y la teoría de la inutilidad.

Esta Norma también breifly ciertas directrices en cuanto a las opciones de renderizado, servidor, cliente o híbrido como es el caso de la tríada de todo. Profundicemos en los principios y reglas que rigen la construcción de componentes de interfaz de usuario.

3.2.0.0 Principios y normas

Al igual que cualquier otro tipo de componente expositor. Se requiere que las UIs sean capaces de asignar procesos, resultados y errores a sus consumidores. Algunos de estos componentes de interfaz de usuario requerirán un enfoque basado en pruebas. Otros son más parecidos a los correderos, donde sólo son envoltorios de componentes de interfaz de usuario nativos o de terceros. Vamos a hablar de estos principios aquí.

3.2.0.0.0 Progreso (Carga)

El principio más importante en la construcción de componentes de interfaz de usuario es desarrollar inteligencia para mantener al usuario comprometido mientras un determinado proceso está en marcha. Es posible que hayas visto algunos con un simple spinner o una barra de progreso para mantener a los usuarios informados en todo momento de lo que ocurre entre bastidores en el sistema.

Es una violación de la Norma indicar un progreso de cualquier tipo si realmente no está pasando nada en el fondo. Cae en la práctica de hacer perder el tiempo a los usuarios finales y básicamente mentirles sobre el estado real del sistema. Pero asumiendo que el sistema está realmente ocupado trabajando en una determinada petición, hay tres niveles de comunicación que pueden ocurrir en un componente expositor para comunicar un progreso. Vamos a discutirlos en detalle:

3.2.0.0.0.0 Progreso básico

El enfoque básico de progreso es donde se presenta un estado con una etiqueta como "Esperando..." o un spinner sin más indicaciones. Este es el mínimo de indicación de progreso y ninguna interfaz de usuario debería congelarse o dejar de funcionar mientras se procesan las solicitudes en segundo plano, asumiendo que un patrón de consistencia eventual no es alcanzable para la necesidad actual del negocio.

Algunas aplicaciones web optan por mostrar una barra de progreso permanente en la parte superior de la página para indicar que se está produciendo un progreso. Desde el punto de vista de la experiencia y dependiendo del nivel de visibilidad de estas barras de progreso, los usuarios finales pueden o no pasárlas por alto. Algunos otros equipos de ingeniería han optado por reproducir una simple animación para mantener a los usuarios comprometidos con el progreso visual sin ninguna indicación de los detalles de ese progreso.

3.2.0.0.0.1 Progreso restante

Un poco más allá de lo mínimo, hay indicaciones sobre el tiempo restante o el progreso que queda por completar antes de que se procese la solicitud. Una indicación como "40% restante" o algo más específico como "5 minutos restantes..." para ayudar a los usuarios finales a entender o estimar cuánto tiempo o effort queda. Hay patrones en los que los ingenieros indicarían cómo quedan las tareas sin ninguna indicación de cuáles son estas tareas.

A veces, una actualización del progreso del reamining es lo más detallado que pueden conseguir los ingenieros de la interfaz de usuario. Por ejemplo, si estás descargando un archivo de Internet. No se puede ser más detallado que decir que queda un x por ciento de bits por descargar sin más detalles. Algunos desarrolladores de juegos optan por visualizar también las velocidades de Internet y el espacio disponible en el disco, todo ello para mantener al usuario final interesado en el sistema. Y todos estos son patrones aceptables en esta Norma.

3.2.0.0.0.2 Progreso detallado

El nivel más alto de información sobre el progreso es el tipo de progreso detallado. El componente de la interfaz de usuario es totalmente transparente con sus consumidores, ya que informa de cada paso del progreso. Este tipo de progreso es más común en aplicaciones científicas. Los ingenieros en modo de depuración pueden habilitar una función en la que toda la actividad subyacente en el sistema se visualiza a través de la UI.

Este tipo de sistema ayuda a los usuarios finales a entender lo que ocurre entre bastidores, pero también les permite comunicar mejor los detalles a los ingenieros de soporte para ayudarles a solucionar un problema si el proceso falla. Pero este proceso no siempre es el preferido en términos de experiencia, ya que algunos detalles deben ocultarse por razones de seguridad.

En resumen, la selección del tipo correcto de progreso en la UI depende principalmente del flow del negocio, del tipo de usuarios que interactuarán con el sistema y de varias otras variables que discutiremos en The Experience Standard.

3.2.0.0.1 Resultados

Los componentes del expositor de UI informarán de un resultado para indicar la finalización de una determinada solicitud por parte de los usuarios finales. Considere la posibilidad de registrar a un nuevo estudiante en un sistema escolar. Hay varias formas de indicar que el proceso de registro se ha completado con éxito. Vamos a discutir ese tipo de visualización de resultados en detalle aquí.

3.2.0.0.1.0 Simple

La simple indicación de éxito es cuando la UI informa de que el proceso se ha completado con éxito sin más detalles. Es posible que hayas visto algunas de estas implementaciones para este tipo como "Gracias, solicitud enviada" o algo tan simple como una marca de verificación con una visualización de color verde que indica el éxito de alguna manera.

Las indicaciones de resultados simples, especialmente con las solicitudes enviadas que los datos recuperados, pueden añadir algunos detalles más en términos del siguiente curso de acción.

3.2.0.0.1.1 Detalles parciales

El otro tipo de resultados o indicación de éxito es presentar a los usuarios finales detalles parciales. Una visión general de la naturaleza de la solicitud, en qué punto se encuentra en términos de estado y marcas de tiempo. Los detalles parciales suelen ser útiles cuando se trata de proporcionar al usuario final un "número de ticket" para ayudar a los usuarios finales a realizar un seguimiento de sus solicitudes más tarde para preguntar sobre el estado. Este patrón es muy común en las aplicaciones de comercio electrónico, donde cada solicitud de compra puede ser devuelta con un número de seguimiento para ayudar a los clientes y al personal de atención al cliente a evaluar con las solicitudes.

Los resultados parciales detallados también pueden ser muy útiles para la visualización del proceso de éxito. Especialmente con las solicitudes que contienen múltiples partes. Las solicitudes de mayor envergadura, como la presentación de una solicitud para ingresar en una universidad o similares, pueden contener archivos adjuntos, varias páginas de detalles e información confidencial, como datos de pago o números de la seguridad social.

3.2.0.0.1.2 Detalles completos

En algunos casos, también puede ser preferible informar de todos los detalles de la solicitud presentada. Especialmente en el caso de las solicitudes más pequeñas, en las que puede ser útil ayudar a los usuarios finales a revisar sus solicitudes. Algunos ingenieros prefieren mostrar los detalles completos como un paso extra de confirmación antes de enviar la solicitud. Pero los detalles completos también pueden incluir algo más que los detalles de la solicitud, podría incluir la actualización del estado del servidor un largo con el punto de contacto asignado o un officer de los equipos de mantenimiento y soporte.

Es una violación de la Norma redirigir a los usuarios finales en el momento de presentar sus solicitudes sin indicar lo que ha ocurrido.

3.2.0.0.2 Informes de errores

La principal responsabilidad de los informes de error es informar a los usuarios finales de lo que ha sucedido, por qué ha sucedido y cuál es el siguiente curso de acción. Algunos tipos de informes de error no indican necesariamente ningún curso de acción, lo que puede ser una experiencia pobre dependiendo del flujo del negocio. Pero lo mínimo en la notificación de errores es la indicación básica del error en sí con detalles básicos.

Vamos a hablar de esos tipos aquí.

3.2.0.0.2.0 Información

El mínimo de informes de error es el de tipo informativo. Indicar que se ha producido un error y por qué se ha producido. Algo así como: "Solicitud fallida. Inténtelo de nuevo" o "Solicitud fallida, contrate asistencia". También hay errores informativos que se basan en el tiempo. Algo como: "Nuestros servidores están experimentando actualmente un alto volumen de solicitudes. Por favor, inténtelo de nuevo más tarde". Estos informes de errores informativos son necesarios para que el usuario final siga consumiendo el sistema de alguna manera.

Los informes de errores informativos se rigen por el contexto y el tipo de usuarios que los reciben. En una aplicación científica, cuantos más detalles, mejor. Para algunos otros sistemas, es importante cambiar el lenguaje técnico de los errores a un lenguaje menos técnico. Por ejemplo, no podemos comunicar: "El Id. del estudiante no puede ser nulo, vacío o con espacio". Deberíamos seleccionar un lenguaje más legible como: "Por favor, proporcione una identificación de estudiante válida".

3.2.0.0.2.1 Acciones referenciales/implícitas

El segundo tipo de informes de error son los de tipo referencial. Cuando se produce un error, automáticamente se toma la acción de informar al equipo de soporte y devuelve una referencia de un ticket de soporte a los usuarios finales para que puedan hacer un seguimiento. Puede ver esto mucho cuando los videojuegos no se inician, o ciertas aplicaciones no pueden inicializarse. Los informes de errores referenciales son los mejores cuando se trata de ciertos flows de negocios, ya que se encarga de todas las acciones, envía un correo electrónico al usuario final con el número de referencia y luego sólo hace un seguimiento en un par de días para informar del estado.

En general, cuantas menos acciones requiera un sistema a sus usuarios después de que se produzca un determinado fallo, mejor. Dado que los usuarios finales ya han realizado sus tareas en términos de envío de solicitudes. Resulta aún más conveniente si la solicitud original se pone en cola, como en el caso de los sistemas empresariales de gran volumen, para que los usuarios finales no tengan que volver a enviar los mismos datos.

3.2.0.0.2.2 Actuable

El segundo tipo de informes de error son los informes procesables. Se trata de errores que se producen y proporcionan una acción adicional para que los usuarios vayan más allá en su solicitud. Por ejemplo, hay informes de error que proporcionan un botón para volver a intentarlo. O enviar una solicitud de detalles adicionales a los equipos de ingeniería y soporte.

También hay informes que proporcionan una ruta diferente para realizar la misma tarea en aplicaciones más híbridas, tanto heredadas como modernizadas. Estos informes procesables son más cómodos que los informes informativos, pero siguen exigiendo a sus usuarios finales más acciones, más pulsaciones de teclas, lo que conlleva un cierto nivel de incomodidad.

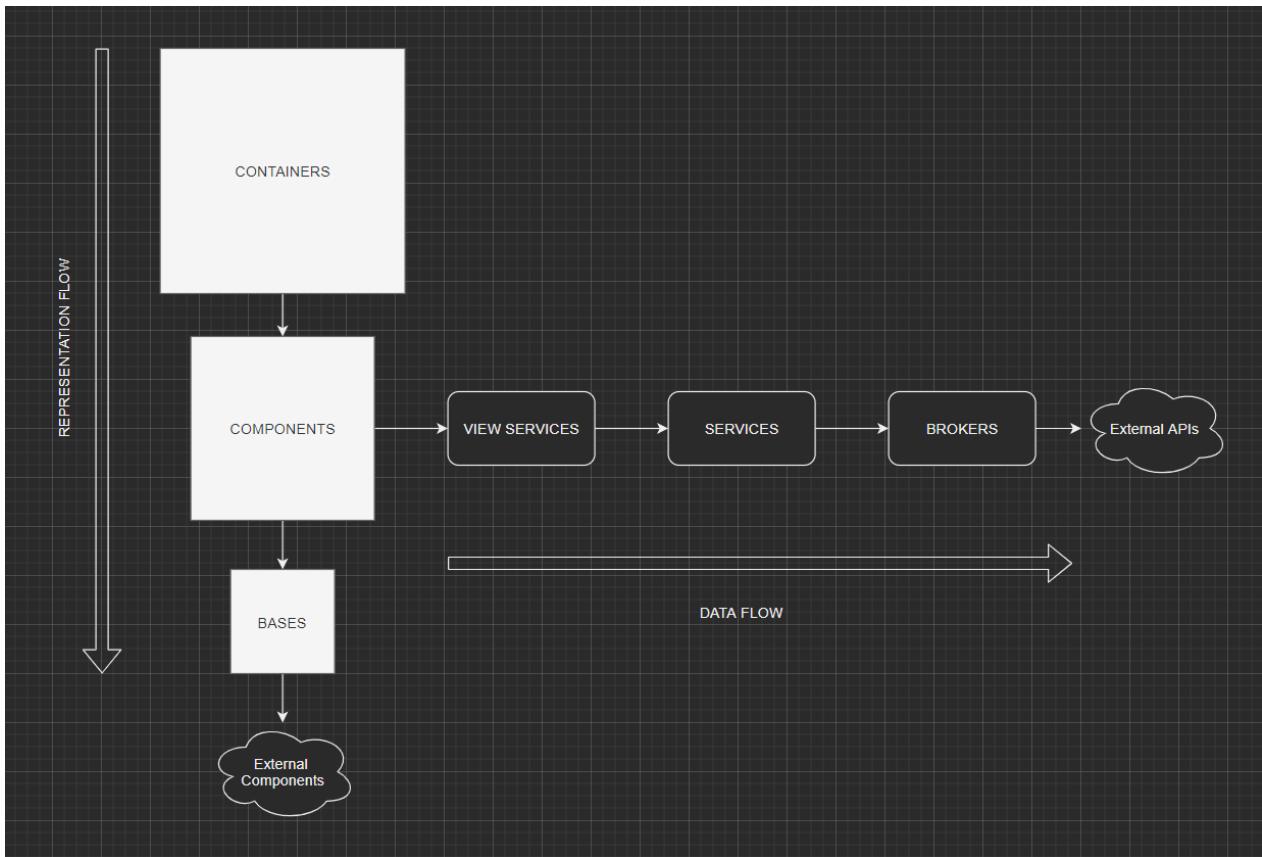
3.2.0.0.3 Dependencia única

Como es el caso de cualquier componente expositor, sólo puede integrarse con una única dependencia en todo momento. Sin embargo, en el caso de los componentes de interfaz de usuario, siempre existe el caso de pureza de contrato en el que se supone que la interfaz de usuario no debe recibir más de lo que necesita en términos de datos. Aquí es donde se implementa un nuevo tipo de servicios de tipo fundacional para garantizar que se cumpla este patrón y que todos los demás detalles, como los fields de auditoría, las marcas de tiempo y demás, se cuiden fuera de la vista del componente de interfaz de usuario.

Hablaremos en detalle de los servicios de vista en breve cuando avancemos hablando de los expositores de UI.

3.2.0.0.4 Anatomía

Al igual que los datos flow en cualquier servicio. Tenemos corredores -> Servicios -> Expositores. Los componentes de la interfaz de usuario también forman su propio flujo de datos en términos de representación:



Los componentes del expositor de la interfaz de usuario, como se ha indicado anteriormente, pueden ser Bases, Componentes o Contenedores. Cada uno de estos tipos tiene una responsabilidad específica para asegurar que la mantenibilidad y la conectividad del sistema estén en su punto más alto de acuerdo con el Estándar. Vamos a discutir estos tres tipos aquí:

3.2.0.0.4.0 Bases

Los Componentes Base o de Base son como los Brokers en el flujo de datos. Son simples envoltorios finos alrededor de componentes nativos o de terceros. Su principal responsabilidad es abstraer la dependencia dura de los componentes no locales para permitir la configurabilidad del sistema para cambiar a cualquier otro componente externo o nativo de la UI con la menor effort posible.

Los componentes base también facilitan la simulación del comportamiento de cualquier componente externo o nativo, y centran la atención en asegurar que el componente local se comporta de la manera esperada. En el próximo capítulo hablaremos de los componentes base para aplicaciones web en Blazor y otras tecnologías.

3.2.0.0.4.1 Componentes

Los componentes de interfaz de usuario son un híbrido entre un servicio y un controlador en la cadena de datos. En cierto modo, los componentes contienen *cierta* lógica de negocio en términos de manejo de las interacciones con ciertos componentes base. Pero también están limitados por la integración con uno y sólo un servicio de vista. Los componentes son dirigidos por pruebas, requieren escribir pruebas para asegurar que se comportan como se espera. Pero también no contienen casi ninguna lógica de iteración, selección o secuenciación de datos dentro de ellos.

El aspecto más importante sobre los componentes de interfaz de usuario es que están en la intersección entre el flujo de interfaz de usuario y el flujo de datos. Se encargan de aprovechar su dependencia de los datos (servicios de vista) y sus componentes base para ser fácilmente enchufables en componentes contenedores (como las páginas con rutas en las aplicaciones web).

3.2.0.0.4.2 Contenedores

Los componentes del contenedor son orquestadores/agregadores de componentes. Son la ruta real o la página con la que interactúan los usuarios finales. Los contenedores no pueden tener ningún nivel de lógica de interfaz de usuario en ellos. No pueden aprovechar los componentes base. Y pueden tener cualquier número de componentes de interfaz de usuario como el flujo de negocio requiere.

Como ocurre con todas las categorías de componentes, los contenedores no pueden integrarse con otros contenedores. La regla se aplica de forma generalizada a todos los componentes de datos o de interfaz de usuario de cualquier tipo.

3.2.0.0.5 Tipos de componentes de interfaz de usuario

Los componentes de la interfaz de usuario tienen diferentes formas y tamaños. El entorno de alojamiento y el tipo de dispositivos que sirven a estos componentes desempeñan un papel importante a la hora de determinar las tecnologías y las capacidades que puede tener un determinado componente de interfaz de usuario. En esta sección hablaremos de los distintos tipos de componentes de interfaz de usuario.

3.2.0.0.5.0 Aplicaciones web

El tipo más popular de aplicaciones de interfaz de usuario son las aplicaciones web. Las aplicaciones web son las más populares y dominantes debido a su facilidad de uso. No requieren ningún tipo de instalación. No dependen del sistema operativo que las ejecute ni del tipo de dispositivos

que utilicen los usuarios. Pueden funcionar en ordenadores, tabletas, teléfonos móviles e incluso en televisores y relojes que admitan la navegación web.

En los últimos años, los frameworks web han evolucionado mucho debido a la mencionada popularidad. Existen frameworks que permiten a los ingenieros escribir aplicaciones web en muchos lenguajes de programación hoy en día. La evolución del ensamblaje web también ha abierto la puerta para que los ingenieros desarrollen marcos aún más escalables con sus tecnologías y lenguajes preferidos.

Las aplicaciones web se desarrollan en dos tipos diferentes en términos de renderización. Aplicaciones del lado del servidor y aplicaciones del lado del cliente. En los próximos capítulos de La Norma hablaremos de las ventajas y desventajas de cada tipo, además del modelo híbrido.

3.2.0.0.5.1 Aplicaciones móviles

La segunda plataforma más popular hoy en día para desarrollar interfaces de usuario es el mundo móvil. El desarrollo de aplicaciones móviles conlleva sus propios retos, ya que dependen en gran medida del sistema operativo, del tamaño del teléfono en términos de resolución y de los controles nativos disponibles. Además, las aplicaciones móviles son siempre aplicaciones del lado del cliente. Al igual que las aplicaciones de escritorio, requieren ser compiladas, aprovisionadas y publicadas en una tienda de aplicaciones para que los consumidores puedan descargarlas, instalarlas y aprovecharlas en sus actividades diarias.

La mayor ventaja de las aplicaciones móviles es que permiten interacciones offline. Como los juegos móviles, las aplicaciones de edición y los servicios de streaming con capacidades offline. Pero la construcción de aplicaciones móviles con frameworks web es cada vez más popular. El ecosistema universal que permite a los usuarios finales experimentar el software de la misma manera en sus PC, navegadores y aplicaciones móviles de la misma manera. Esta tendencia acabará permitiendo a los ingenieros desarrollar sistemas para todos los ecosistemas diferentes con el menor coste posible.

3.2.0.0.5.2 Otros tipos

Existen otros tipos de componentes de interfaz de usuario que no se incluyen en nuestra norma. Estos tipos son, por ejemplo, las aplicaciones de consola/terminal, las aplicaciones de escritorio, los videojuegos y el software de realidad virtual/aumentada, además de los dispositivos portátiles y los sistemas activados por voz. El mundo de la interfaz hombre-máquina HMI está evolucionando tan rápidamente en la era del metaverso que es posible que tengamos que crear en algún momento capítulos especiales para estos tipos diferentes.

3.2.1 Aplicaciones web

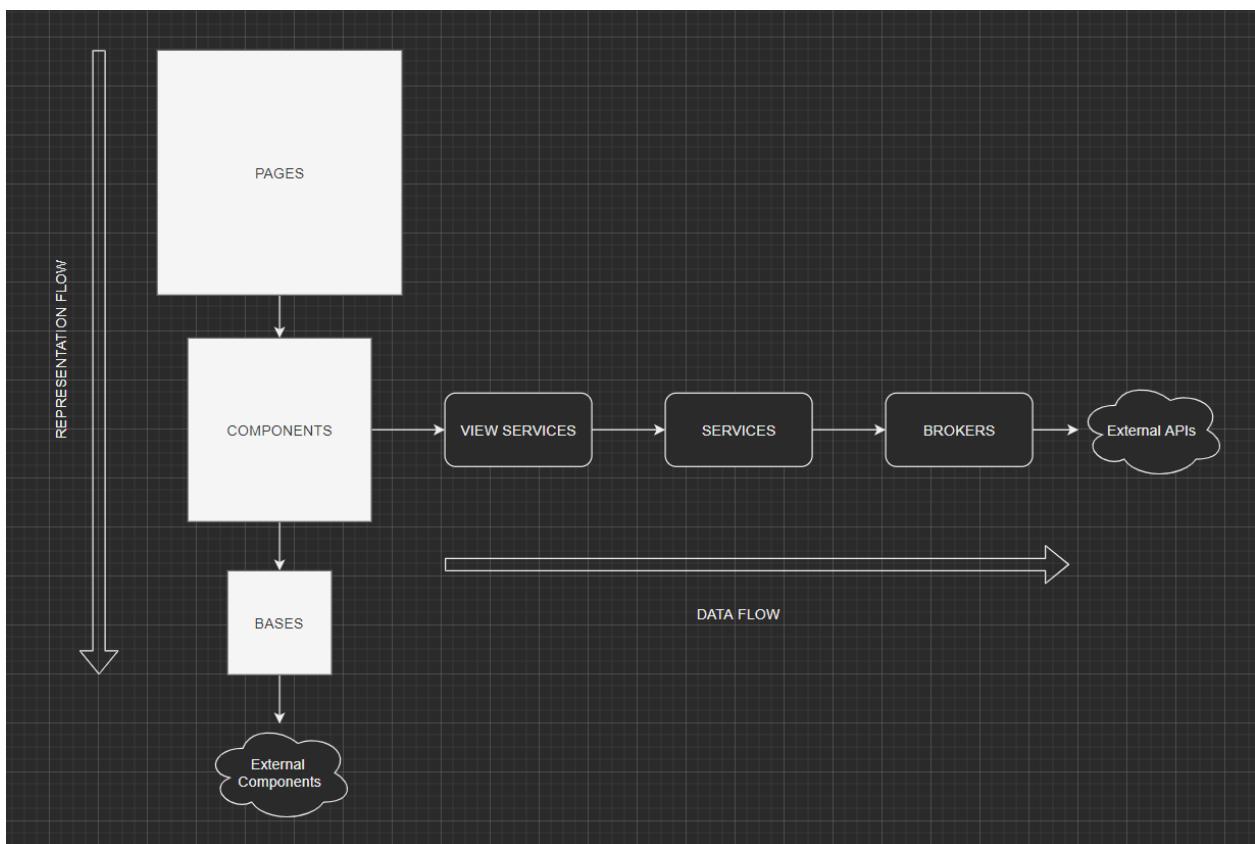
3.2.1.0 Introducción

Las aplicaciones web son el tipo más común de componentes expositores hoy en día. Son mucho más fáciles de usar que cualquier otro componente UI expositor conocido en la industria del software. Pero lo más importante es que las aplicaciones web tienen un conjunto de tecnologías mucho más diversas que las aplicaciones móviles. El mercado de software web es también mucho más fácil para el ingeniero para publicar y actualizar que las aplicaciones móviles que hace que sea bastante atractivo para los nuevos ingenieros en general.

En este capítulo, utilizaremos la tecnología Blazor para demostrar la implementación de los principios de El Estándar en las aplicaciones web. Pero como he mencionado anteriormente, El Estándar es agnóstico a la tecnología. Lo que significa que se puede aplicar a cualquier tecnología web sin ningún problema.

3.2.1.1 En el mapa

Las aplicaciones web suelen situarse en el otro extremo de cualquier sistema. Son los terminales que los humanos utilizan para interactuar con el sistema. Veamos dónde se encuentran en el mapa:



Como se muestra arriba, las aplicaciones web son algo similar a las APIs centrales, excepto que tienen un grupo diferente de componentes en términos de visualización como Páginas, Componentes y Bases. Hay una intersección entre dos flows principales en cada aplicación web. El flujo de presentación y el flujo de datos/negocios. Dependiendo de dónde viva una aplicación web en términos de arquitectura de alto nivel su ubicación determina si su backend (BFF o Backend de Frontend) es un flow de negocio o sólo un flow de datos. Vamos a discutir estos detalles en la sección de características en este capítulo.

3.2.1.2 Características

Las aplicaciones web suelen tener 6 componentes básicos. Corredores, Servicios, Servicios de Vista, Bases, Componentes y Páginas. Como ya hemos discutido los componentes de flujo de datos en la parte de Servicios de El Estándar. En esta sección, discutiremos el aspecto UI (Bases, Componentes y Páginas) con un ligero detalle sobre los servicios de vista.

Hablemos aquí de estas características.

3.2.1.2.0 Anatomía

Los componentes de la interfaz de usuario consisten en la base, los componentes y las páginas. Todos ellos desempeñan el papel de separar la responsabilidad de la integración, la representación y el enrutamiento de los usuarios a una funcionalidad particular de la interfaz de usuario.

Hablemos de estos tipos en detalle.

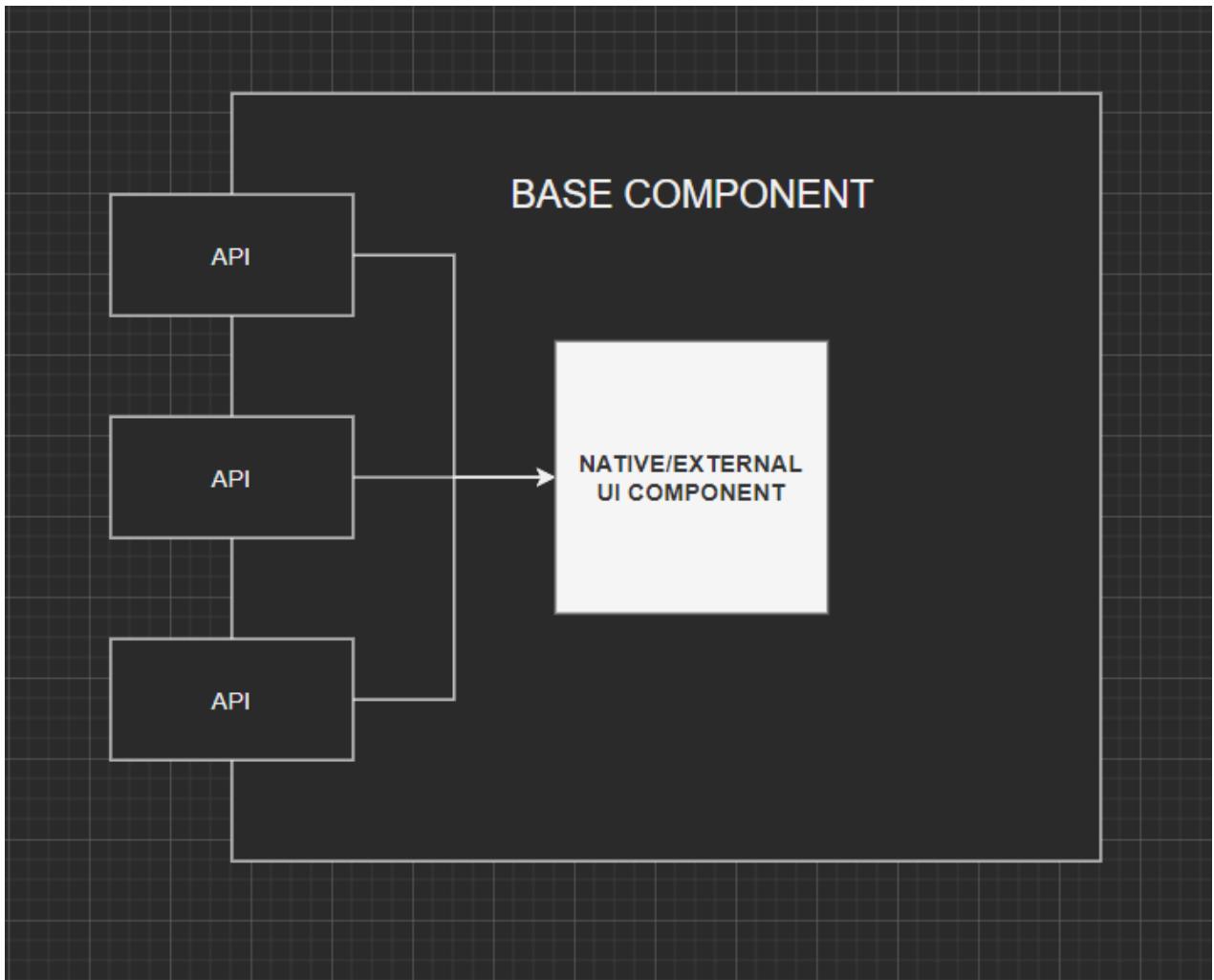
3.2.1.2.0.0 Componente base

Los componentes base son como los brokers, son envoltorios de componentes de interfaz de usuario nativos o externos. Su principal responsabilidad es abstraer cualquier dependencia de la capacidad de la interfaz de usuario no local. Por ejemplo, digamos que queremos offer la `<input>` de crear cajas de texto para la inserción/captura de datos. La interfaz nativa

podría ofrecer esta capacidad. Pero exponer o aprovechar esta etiqueta en

nuestros componentes principales de la interfaz de usuario es peligroso. Porque crea una fuerte dependencia de componentes de interfaz de usuario no abstractos. Si en algún momento decidimos utilizar algún `<input>` componente de interfaz de usuario de terceros, tendríamos que cambiar estrategias a través de todos los componentes que los utilizan. Esa no es una estrategia

óptima. Echemos un vistazo a una visualización de la funcionalidad del `tags` componente base:



Como se ha visto en el ejemplo anterior, los componentes base envolverán un componente de interfaz de usuario externo o nativo y luego expondrán APIs para permitir la interacción con ese componente sin problemas y de forma programática. Hay ocasiones en las que estas APIs representarán parámetros, funciones o delegados para interactuar con el componente basado en el flujo de negocio.

3.2.1.2.0.0.0 Aplicación

Veamos un sencillo componente Base para resolver este problema:

```

<input @bind-value=Value />

public partial class TextBoxBase : ComponentBase
{
    [Parámetro]
    public string Valor {get; set;}

    public void SetValue(string value) =>
        this.Value = value;
}

```

En el código anterior, hemos envuelto la etiqueta `base` con nuestra propia etiqueta `TextBoxBase`. El componente `TextBoxBase` ofrece un parámetro de entrada `Valor` para ser pasado a ese componente para que pueda pasarlo al elemento nativo de la UI. Además, también proporcionamos una función `publica` para permitir imitando programáticamente el comportamiento de los usuarios para probar el componente de consumo de este elemento base.

3.2.1.2.0.0.1 Utilización

Ahora, cuando tratamos de aprovechar este componente base en el nivel de los componentes principales, podemos simplemente llamarlo de la siguiente manera:

```
< TextBoxBase @ref=MyTextBox />
```

El aspecto permitirá que el código del backend interactúe con la base programáticamente entre bastidores para llamar a cualquier funcionalidad existente.

3.2.1.2.0.0.2 Restricciones

Los componentes base sólo pueden ser utilizados por los componentes del núcleo o simplemente componentes para abreviar. No pueden ser utilizados por páginas y no pueden ser utilizados por otros componentes Base. Pero lo más importante es que se prefiere que los componentes base sólo se envuelvan en uno y sólo un componente no local.

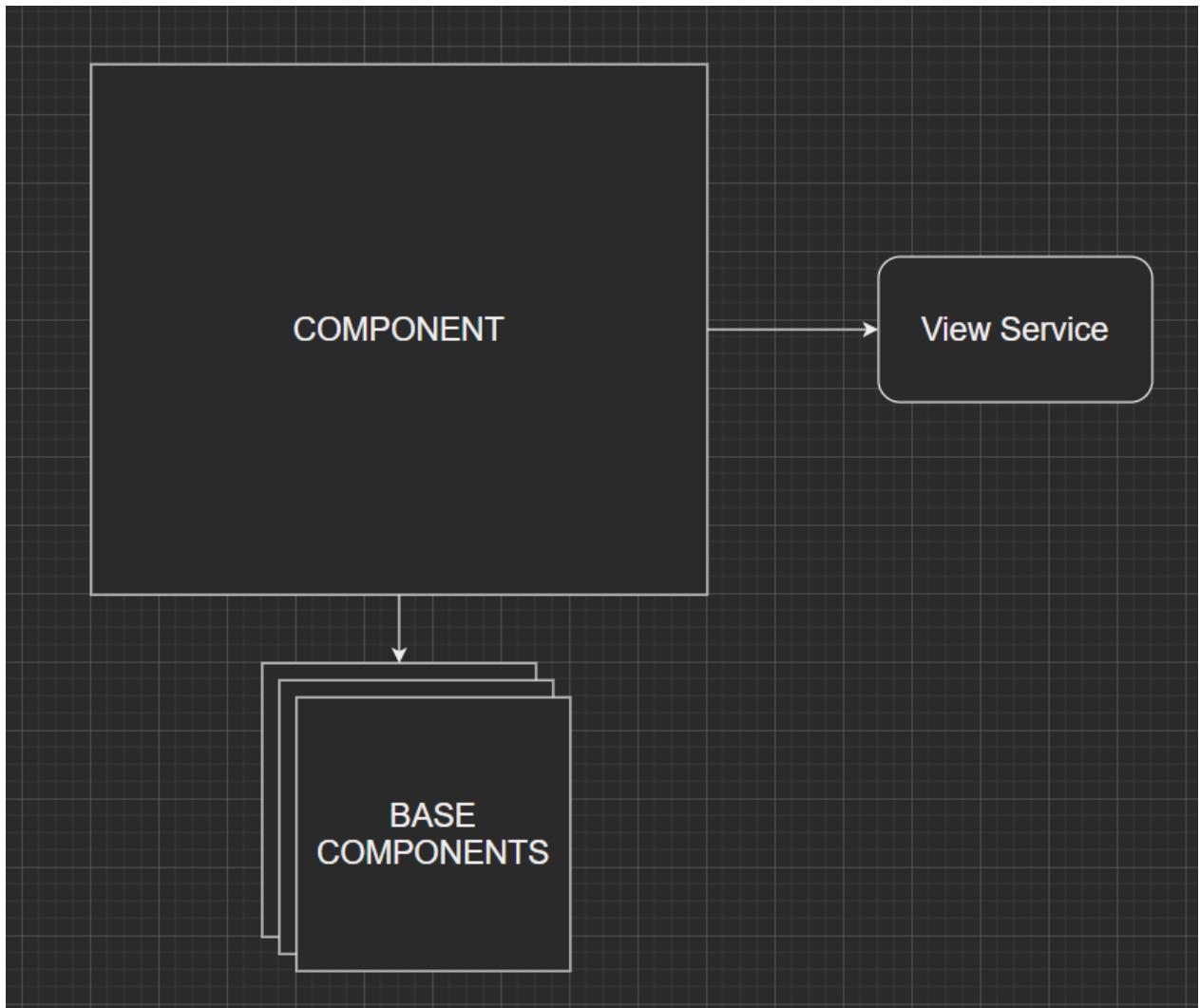
Y al igual que los Brokers, los Componentes Base no tienen ninguna lógica en ellos. No manejan excepciones, no hacen cálculos ni ninguna forma de operaciones lógicas de negocio secuenciales, iterativas o selectivas. Estas operaciones se basan en los datos, cuando pertenecen a los servicios de visualización y a las APIs posteriores, o se basan en la interfaz de usuario, cuando pertenecen a los componentes básicos.

Los componentes base tampoco manejan excepciones, no lanzan sus propias excepciones y no realizan ningún tipo de validaciones.

3.2.1.2.0.1 Componente principal

Los componentes básicos son como los servicios en el flujo de datos. Están orientados a las pruebas, pero también están restringidos a una y sólo una dependencia en todo momento. Los componentes básicos aprovechan los componentes base para realizar un flujo específico de negocio. Son menos genéricos que los componentes Base porque orquestan y se comunican con el flujo de datos.

Aquí hay una visualización de la arquitectura de los componentes principales:



Los componentes básicos son, en cierto modo, orquestadores de los componentes de interfaz de usuario y de datos. Aprovecharán uno o varios componentes Base para construir un flujo específico de negocio, como un formulario de inscripción de estudiantes, y luego enviarán la señal a los servicios de visualización para que persistan los datos y devuelvan las respuestas o informen de los errores.

3.2.1.2.0.1.0 Aplicación y pruebas

Veamos la implementación de un componente central.

```

public partial class MatriculaEstudianteComponente : ComponentBase
{
    [Injectar]
    public IStudentViewService StudentViewService {get; set; }

    public StudentRegistrationComponentState State {get; set;}
    public StudentView StudentView {get; set;}
    public TextBoxBase StudentNameTextBox {get; set;}
    public ButtonBase SubmitButton {get; set;}
    public LabelBase StatusLabel {get; set;}

    public void OnInitialized() =>
        this.State == StudentRegistrationComponentState.Content;

    public async Task SubmitStudentAsync()
    {
        pru
        ebe
        con this.StudentViewService.AddStudentViewAsync(this.StudentView);
    }

    catch (Exception exception)
    {
        this.State = StudentRegistrationComponentState.Error;
    }
}
}

```

El código anterior muestra los diferentes tipos de propiedades dentro de cualquier componente, el servicio de vista de dependencia que mapea los modelos/datos de la API en bruto en modelos de interfaz de usuario consumibles. Y el Estado que determina si un componente debe estar Cargando, Contenido o Error. El modelo de vista de datos para vincular la entrada a un modelo unificado StudentView. Y los últimos tres son componentes de nivel base que se utilizan para construir el formulario de registro.

Echemos un vistazo a la parte de marcado del componente principal:

```

<Evaluación de la condición=IsLoading>
    <Fósforo>
        < LabelBase @ref=StatusLabel Value="Cargando..." />
    </Match>
</Condición>

<Condición Evaluación=IsContent>
    <Fósforo>
        < TextBoxBase @ref=StudentNameTextBox @bind-value=StudentView.Name />
        <ButtonBase @ref=SubmitButton Label="Submit" OnClick=SubmitStudentAsync />
    </Match>
</Condición>

<Condición Evaluación=IsError>
    <Fósforo>
        < LabelBase @ref=Etiqueta de estado Value="Se ha producido un error" />
    </Match>
</Condición>

```

Vinculamos las referencias de las propiedades de los componentes de registro de los estudiantes a los componentes de la interfaz de usuario para garantizar que la representación de estos componentes se ha producido realmente y el envío de datos se ha ejecutado.

Veamos un par de pruebas para verificar estos estados. un componente ya ha cargado el estado. Y los estados posteriores al envío.

```

[Hecho]
public void ShouldRenderComponent()
{
    // dado
    StudentRegistrationComponentState expectedComponentState =
        StudentRegistrationComponentState.Content;

    // cuando
    this.renderedStudentRegistrationComponent =
        RenderComponent< StudentRegistrationComponent>();

    // entonces
    this.renderedStudentRegistrationComponent.Instance.StudentView
        .Debería().NotBeNull();

    this.renderedStudentRegistrationComponent.Instance.State
        .Debería().Be(expectedComponentState);

    this.renderedStudentRegistrationComponent.Instance.StudentNameTextBox
        .Debería().NotBeNull();

    this.renderedStudentRegistrationComponent.Instance.SubmitButton
        .Debería().NotBeNull();

    this.renderedStudentRegistrationComponent.InstanceStatusLabel.Value
        .Debería().BeNull();

    this.studentViewServiceMock.VerifyNoOtherCalls();
}

```

La prueba anterior verificará que todos los componentes tienen asignada una propiedad de referencia y que no se han realizado llamadas a dependencias externas. `validacionesOnIntialized`

el código en el nivel de los componentes se valida y funcionando como se esperaba.

Ahora, echemos un vistazo a las validaciones del código de presentación:

```

[Hecho]
public void DeberíaSubirEstudianteAsync()
{
    // dado
    StudentRegistrationComponentState expectedComponentState =
        StudentRegistrationComponentState.Content;

    var inputStudentView = new StudentView
    {
        Nombre = "Hassan Habib"
    };

    StudentView expectedStudentView = inputStudentView;

    // cuando
    this.renderedStudentRegistrationComponent =
        RenderComponent< StudentRegistrationComponent>();

    this.renderedStudentRegistrationComponent.Instance.StudentName
        .SetValue(inputStudentView.Name);

    this.renderedStudentRegistrationComponent.Instance.SubmitButton. Click();

    // entonces
    this.renderedStudentRegistrationComponent.Instance.StudentView
        .Debería(). NotBeNull();

    this.renderedStudentRegistrationComponent.Instance.StudentView
        .Debería(). BeEquivalentTo(expectedStudentView);

    this.renderedStudentRegistrationComponent.Instance.State
        .Debería(). Be(expectedComponentState);

    this.renderedStudentRegistrationComponent.Instance.StudentNameTextBox
        .Debería(). NotBeNull();

    this.renderedStudentRegistrationComponent.Instance.StudentNameTextBox.Value
        .Debería(). BeEquivalentTo(studentView.Name);

    this.renderedStudentRegistrationComponent.Instance.SubmitButton
        .Debería(). NotBeNull();

    this.renderedStudentRegistrationComponent.InstanceStatusLabel.Value
        .Debería(). BeNull();

    this.studentViewServiceMock. Verify(service => service.
        AddStudentAsync(inputStudentView),
        Times.Once);

    this.studentViewServiceMock. VerifyNoOtherCalls();
}

```

La prueba anterior valida que en el momento del envío, el modelo de estudiante se rellena con el conjunto de datos de forma programada a través de la instancia del componente base, pero también verifica que todos estos componentes se renderizan realmente en la pantalla ante los usuarios finales validando que cada componente base tiene una instancia asignada en tiempo de ejecución o de renderización.

3.2.1.2.0.1.1 Restricciones

Los componentes Core tienen restricciones similares a las de los componentes Base en el sentido de que no pueden llamarse entre sí a ese nivel. Hay un nivel de componentes básicos de orquestación que pueden combinar varios componentes para intercambiar mensajes, pero no renderizan nada por sí mismos de la misma manera que los servicios de orquestación delegan todo el trabajo a sus dependencias.

Pero los componentes del núcleo tampoco pueden llamar a más de un servicio de vista. Y en eso, se mantienen fieles al modelo de vista en todo momento. Un servicio de vistas corresponde a un componente central que renderiza un solo modelo de vista.

Los servicios de vista pueden hacer su propio trabajo a nivel de orquestación también, en un modo extremadamente complejo, pero es muy recomendable mantener las cosas a un nivel sencillo. Estos mismos servicios de vista no realizan nada más que el mapeo y la adición de campos de auditoría, además de las validaciones estructurales básicas.

3.2.1.2.0.2 Páginas

En toda aplicación web, las páginas son un componente contenedor obligatorio muy fundamental que debe existir para que los usuarios finales puedan navegar hacia ellas. Las páginas contienen principalmente una ruta, comunican un parámetro de esa ruta y combinan componentes de nivel básico para representar un valor comercial.

Un buen ejemplo de páginas son los cuadros de mando. Las páginas de tableros son un contenedor de múltiples componentes como mosaicos, notificaciones, cabeceras y barras laterales con referencias a otras páginas. Las páginas no contienen ninguna lógica de negocio en sí mismas, pero delegan todas las operaciones relacionadas con las rutas a sus componentes hijos.

Veamos la implementación de una página sencilla:

```
@página '/registro'  
<CabeceraComponente />  
<Componente de registro de estudiantes />  
<FooterComponent />
```

Las páginas son mucho más sencillas que el núcleo o los componentes base. No requieren pruebas unitarias, y no necesitan necesariamente un código backend. Simplemente hacen referencia a sus componentes sin referencia (a menos que sea necesario) y ayudan a servir ese contenido cuando se navega a una ruta.

3.2.1.2.0.3 Discreción

Para todos los componentes de la interfaz de usuario, es una violación incluir código de múltiples tecnologías/lenguajes en la misma página. Por ejemplo, el código de estilo CSS, el código C# y el marcado HTML no pueden existir en el mismo archivo. Deben estar separados en sus propios archivos.

La regla de la discreción ayuda a evitar la contaminación cognitiva de los ingenieros que construyen componentes de interfaz de usuario, pero también hace que el sistema sea mucho más fácil de mantener. Por eso, cada componente puede anidar sus archivos debajo de él si el IDE/entorno utilizado para el desarrollo permite implementaciones parciales de la siguiente manera:

- StudentRegistrationComponent.razor

- StudentRegistrationComponent.razor.cs
- StudentRegistrationComponent.razor.css

El nodo file aquí .navaj file tiene todo el marcado necesario para poner en marcha el a de inicialización del componente. Mientras que los dos files anidados son files de soporte para el código lógico de la interfaz de usuario y el estilo. Con este nivel de organización (especialmente en Blazor) no se requiere ninguna referencia para estos files anidados/de soporte. Este puede no ser el caso de otras tecnologías, por lo que insto a los ingenieros a hacer todo lo posible para adaptarse a ese modelo/estándar.

3.2.1.2.0.4 Organización

Todos los componentes de la interfaz de usuario aparecen en la carpeta Views de la solución:

- Vistas
 - Bases
 - Componentes
 - Páginas

Esta organización conforme a la tri-naturaleza debería facilitar el desplazamiento de los componentes reutilizables y facilitar también la localización de estos componentes en función de sus categorías. Dejaré a la preferencia de los ingenieros la decisión de desglosar aún más estos componentes por carpetas/espacios de nombres o dejarlos todos en el mismo nivel, dado que el anidamiento está en marcha.