El futuro digital es de todos | MinTIC

UNIVERSIDAD EL BOSQUE | MisionTIC2022

Ciclo 4A

Semana 4

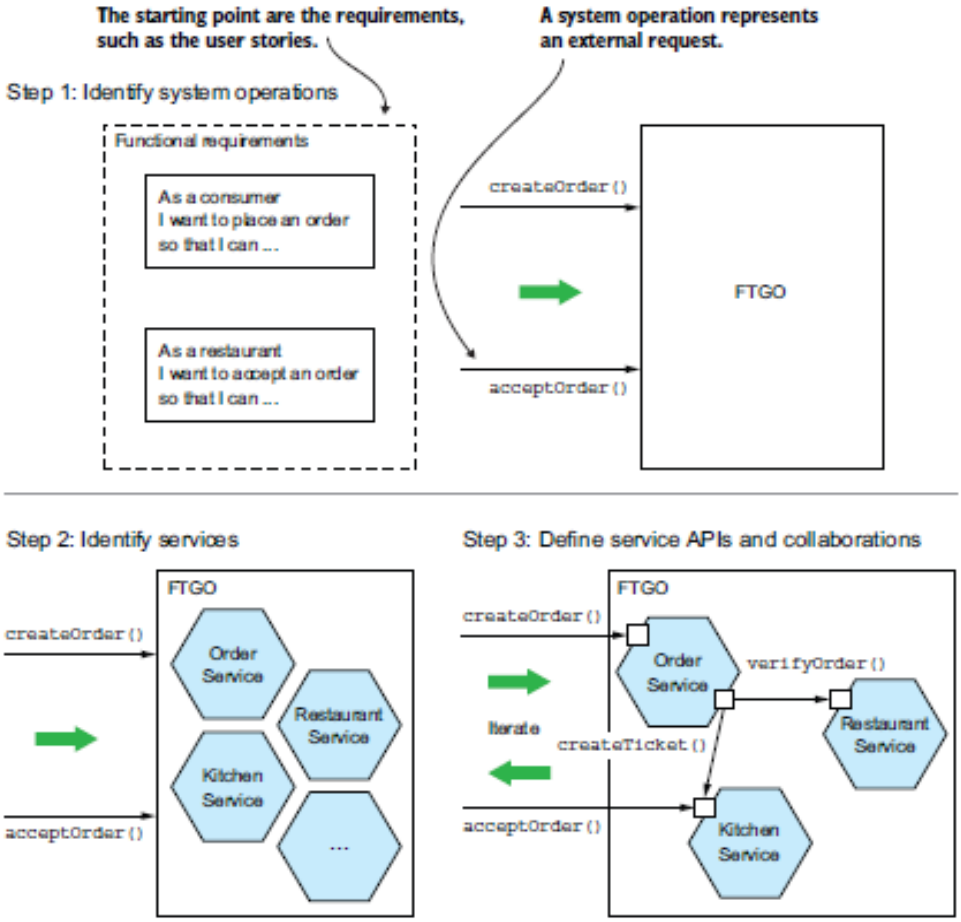
DAO, DTO, manejo de excepciones y documentación de código

Lectura 3 - Definiendo la arquitectura de microservicios de una aplicación

Convenio de asociación no. 777 de 2021 suscrito entre el Fondo Único de Tecnologías de la Información y las Comunicaciones y la Universidad el Bosque.

Definiendo la arquitectura de microservicios de una aplicación

Como gran parte del desarrollo de software, definir una arquitectura es más arte que ciencia. Aquí se describe un proceso simple de tres pasos para definir la arquitectura de una aplicación. Es importante, sin embargo, recordar que no es un proceso que se pueda seguir mecánicamente. Es probable que sea iterativo e implica mucha creatividad.



Una aplicación existe para manejar peticiones, así que el primer paso en definir la arquitectura es establecer los aspectos más importantes o claves de sus requerimientos. Pero en vez de describir los requerimientos en términos de tecnologías específicas de IPC tales como REST o mensajería, aquí se usa una noción más abstracta de operación de sistema. Una operación de sistema es una abstracción de un requerimiento que la aplicación debe manejar. Es un comando que actualiza datos, o una consulta que

recupera datos. El comportamiento de cada comando es definido en términos de un modelo de dominio abstracto, que es también derivado de los requerimientos. Las operaciones de sistema se convierten en escenarios de arquitectura que ilustran como colaboran los servicios.

El segundo paso del proceso es determinar la descomposición en servicios. Hay varias estrategias para elegir. Una estrategia, que tiene su origen en la disciplina de la arquitectura empresarial, es definir los servicios correspondientes a las capacidades del negocio. Otra estrategia consiste en organizar los servicios en torno a subdominios de acuerdo al diseño dirigido por dominios. El resultado final son servicios que se organizan en torno a conceptos comerciales en lugar de conceptos técnicos.

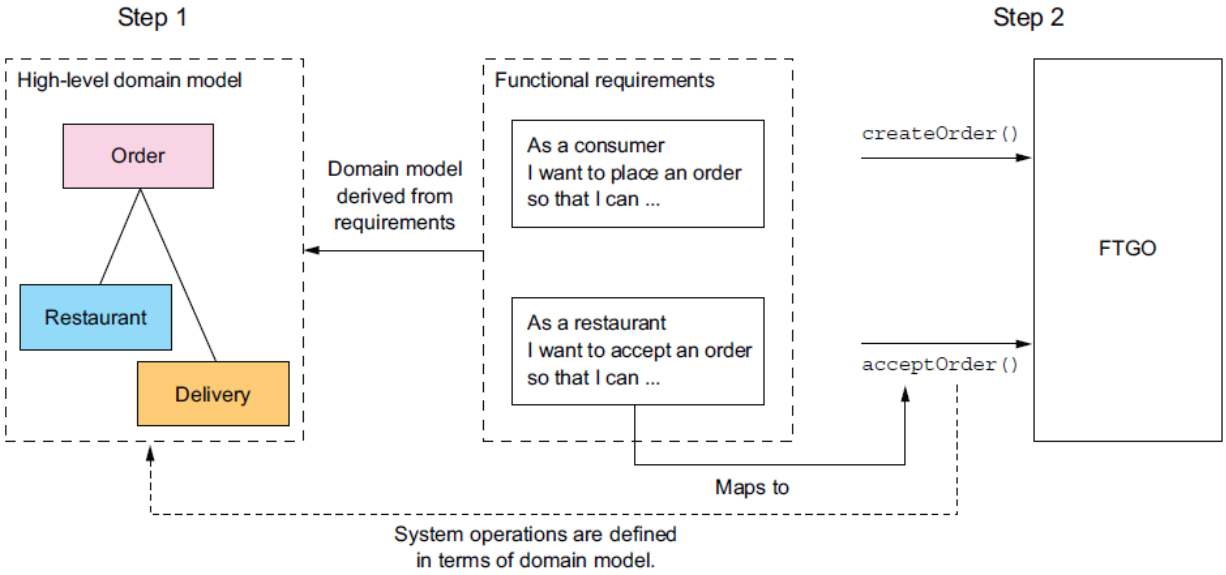
El tercer paso para definir la arquitectura de la aplicación es determinar el API de cada servicio. Para ello, se asigna cada operación de sistema identificada en el primer paso a un servicio. Un servicio puede implementar una operación completamente por sí mismo. Alternativamente, podría necesitar colaborar con otros servicios. En ese caso, se determina cómo los servicios colaboran, que normalmente requiere servicios para soportar operaciones adicionales. También se deberá decidir cuál de los mecanismos de IPC usar para implementar el API de cada servicio.

Hay varios obstáculos para la descomposición. El primero es la latencia de la red. Se podría descubrir que una descomposición en particular no sería práctica debido a demasiados viajes de ida y vuelta entre servicios. Otro obstáculo para la descomposición es que la comunicación sincrónica entre servicios reduce la disponibilidad.

Identificando las operaciones de sistema:

El primer paso para definir la arquitectura de una aplicación es definir las operaciones de sistema. El punto de partida son los requerimientos de la aplicación, incluidas las historias de usuario y sus escenarios de usuario asociados (hay que tener en cuenta que estos son diferentes de los escenarios de arquitectura). Las operaciones de sistema se identifican y definen mediante el proceso de dos pasos que se muestra en la siguiente figura. Este proceso está inspirado en el proceso de diseño orientado a objetos cubierto en el libro de Craig Larman “Applying UML and Patterns (Prentice Hall, 2004). El primer paso crea el modelo de dominio de alto nivel que consta de las clases clave que proporcionan un vocabulario para describir las operaciones de sistema. El segundo paso identifica las operaciones de sistema y describe el comportamiento de cada una en términos del modelo de dominio.

El modelo de dominio se deriva principalmente de los sustantivos de las historias de usuario, y las operaciones de sistema se derivan principalmente de los verbos. El comportamiento de cada operación de sistema se describe en términos de su efecto en uno o más objetos de dominio y las relaciones entre ellos. Una operación de sistema puede crear, actualizar o eliminar objetos de dominio, así como crear o destruir relaciones entre ellos.



Creando un modelo de dominio de alto nivel:

El primer paso en el proceso de definir las operaciones de sistema es esbozar un esquema de alto nivel del modelo de dominio para la aplicación. Se debe tener en cuenta que este modelo de dominio es mucho más simple de lo que finalmente se implementará. La aplicación ni siquiera tendrá un modelo de dominio único porque cada servicio tiene su propio modelo de dominio. A pesar de ser una simplificación drástica, un modelo de dominio de alto nivel es útil en esta etapa porque define el vocabulario para describir el comportamiento de las operaciones de sistema. Se crea un modelo de dominio utilizando técnicas estándar como analizar los sustantivos en las historias y escenarios y hablando con los expertos del dominio. Considere, por ejemplo, la historia de colocar orden, que se puede expandir en numerosos escenarios de usuario, incluido este:

Dado un consumidor

Y un restaurante

Y una dirección / hora de entrega que pueda ser atendida por ese restaurante

Y un pedido total que cumpla con el pedido mínimo del restaurante

Cuando el consumidor coloca una orden para el restaurante

Entonces se autoriza la tarjeta de crédito del consumidor

Y se crea una orden en el estado PENDING_ACCEPTANCE

Y la orden es asociada al consumidor

Y la orden es asociada al restaurante.

Los sustantivos en este escenario de usuario insinúan la existencia de varias clases, incluyendo Consumidor, Orden, Restaurante y TarjetaCredito.

De manera similar, la historia Aceptar orden se puede expandir a un escenario como este:

Dada una orden que está en estado PENDING_ACCEPTANCE

y un mensajero que esté disponible para entregarla

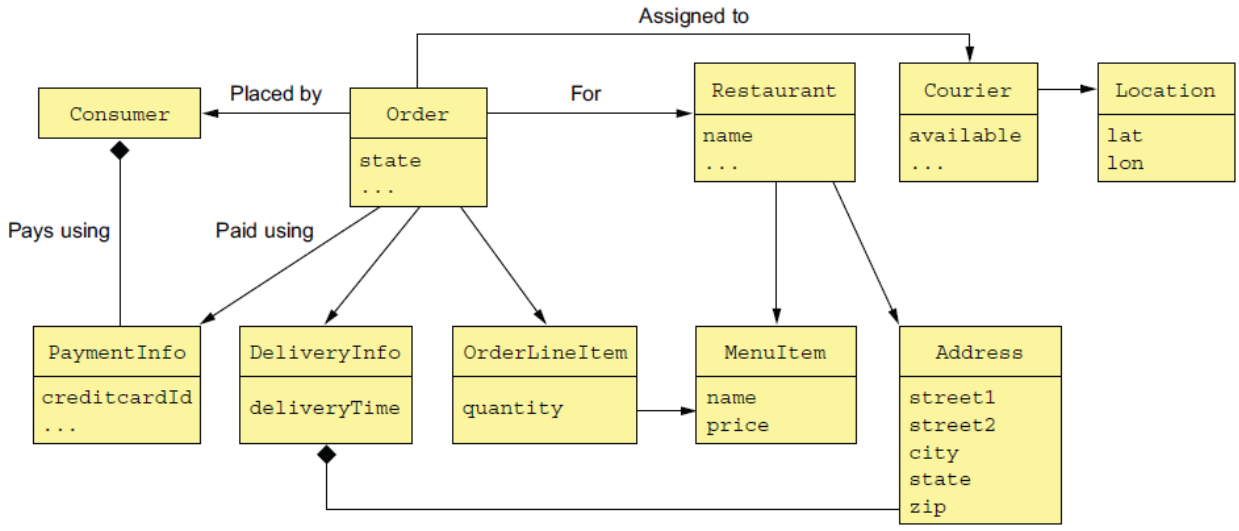
Cuando un restaurante acepta la orden con la promesa de prepararla para un tiempo particular

Entonces el estado de la orden se cambia a ACCEPTED

Y la promesa de tiempo de la orden se actualiza a la hora prometida

Y el mensajero es asignado para entregar la orden.

Este escenario sugiere la existencia de clases Courier y Delivery. El resultado final después de algunas iteraciones de análisis, será un modelo de dominio que consiste, como era de esperar, de esas clases y otras, como MenuItem y Address. Como se puede ver en el diagrama de clases:



Las responsabilidades de cada clase son las siguientes:

- Consumidor: un consumidor que realiza pedidos.
- Pedido: pedido realizado por un consumidor. Describe el pedido y rastrea su estado.
- OrderLineItem: un artículo de línea de un pedido.
- DeliveryInfo: la hora y el lugar para entregar un pedido.
- Restaurante: un restaurante que prepara pedidos para entregarlos a los consumidores.

- MenuItem: un elemento del menú del restaurante.
- Mensajero: un mensajero que entrega pedidos a los consumidores. Realiza un seguimiento de la disponibilidad del mensajero y su ubicación actual.
- Dirección: la dirección de un consumidor o un restaurante.
- Ubicación: latitud y longitud de un mensajero.

Un diagrama de clases como el presentado ilustra un aspecto de la arquitectura de la aplicación. Pero no es mucho más que una bonita imagen sin los escenarios para animarla. El siguiente paso es definir las operaciones de sistema, que corresponden a escenarios de la arquitectura.

Definiendo las operaciones de sistema:

Una vez que haya definido un modelo de dominio de alto nivel, el siguiente paso es identificar las solicitudes que debe manejar la aplicación. Los detalles de la interfaz de usuario no se cubrirán aquí, pero se puede imaginar que en cada escenario de usuario, la interfaz de usuario hará solicitudes a la lógica empresarial de backend para recuperar y actualizar datos. FTGO (la aplicación de ejemplo que está utilizando) es principalmente una aplicación web, lo que significa que la mayoría de las solicitudes están basadas en HTTP, pero es posible que algunos clientes podrían usar mensajería. En lugar de comprometerse con un protocolo específico, por lo tanto, tiene sentido utilizar la noción más abstracta de una operación de sistema para representar solicitudes.

Hay dos tipos de operaciones de sistema:

- Comandos: operaciones del sistema que crean, actualizan y eliminan datos
- Consultas: operaciones del sistema que leen (consultan) datos

En última instancia, estas operaciones de sistema corresponderán a REST, RPC o mensajería de puntos finales, pero por ahora es útil pensar en ellos de forma abstracta. Primero se identifican algunos comandos. Un buen punto de partida para identificar los comandos del sistema es analizar los verbos en las historias de usuario y escenarios. Considere, por ejemplo, la historia de Colocar Orden. Sugiere claramente que el sistema debe proporcionar una operación Crear orden. Muchas otras historias individualmente mapean directamente a comandos del sistema. La siguiente tabla enumera algunos de los comandos clave del sistema.

Actor	Historia	Comando	Descripción
Consumidor	Crear Orden	crearOrden()	Crea una orden
Restaurante	Aceptar Orden	aceptarOrden()	Indica que el restaurante ha aceptado la orden y está comprometido a prepararla en el tiempo indicado

Restaurante	Orden lista	ordenLista()	Indica que la orden está lista para tomar
Mensajero	Actualiza Localización	actualizaLocalizacion()	Actualiza la localización actual del mensajero
Mensajero	Entrega tomada	entregaTomada()	Indica que el mensajero ha tomado la orden
Mensajero	Entrega entregada	entregaEntregada()	Indica que el mensajero ha entregado la orden

Un comando tiene una especificación que define sus parámetros, valor de retorno y comportamiento en términos de las clases del modelo de dominio. La especificación de comportamiento consta de pre-condiciones que deben ser ciertas cuando se invoca la operación, y las pos-condiciones que son ciertas después de invocar la operación. Aquí, por ejemplo, se muestra la especificación de la operación de sistema createOrder ():

Operación	createOrder(idConsumer, metodoPago, direcEntrega, tiempoEntrega, idRestaurante, itemsOrden)
Retorno	orderId…
Pre-condiciones	El cliente existe y puede poner órdenes. Los ítems de la orden corresponden a ítems del menú del restaurante La dirección y tiempo de entrega pueden ser servidas por el restaurante.
Post-condiciones	La tarjeta de crédito del consumidor fue autorizada por el total de la orden. Una orden fue creada en estado PENDING_ACCEPTANCE

Las pre-condiciones reflejan lo que en el escenario de usuario ColocarOrden aparecía como “Dado o Dada” descrito anteriormente. Las post-condiciones reflejan las “Entonces” del escenario. Cuando una operación del sistema se invoca, verificará las condiciones previas y realizará las acciones necesarias para hacer verdaderas las post-condiciones. Aquí se muestra la especificación de la operación de sistema acceptOrder ():

Operación	acceptOrder(idRestaurante, idOrden,tiempoEntrega)
Retorno	…
Pre-condiciones	El estado de la orden es PENDING_ACCEPTANCE. Un mensajero está disponible para entregar la orden.
Post-condiciones	El estado de la orden fue cambiado a ACCEPTED. El tiempo de la orden fue cambiado a tiempo de entrega. El mensajero fue asignado a entregar la orden.

La mayoría de las operaciones de sistema arquitectónicamente relevantes son comandos. Algunas veces, sin embargo, las consultas, que recuperan datos, también son importantes.

Además de implementar comandos, una aplicación también debe implementar consultas. Las consultas proporcionan a la interfaz de usuario la información que un usuario necesita para tomar decisiones. En esta etapa, no tenemos en mente un diseño de interfaz de usuario en particular para la aplicación FTGO, pero considere, por ejemplo, el flujo cuando un consumidor realiza un pedido:

1. El usuario ingresa la dirección y la hora de entrega.
2. El sistema muestra los restaurantes disponibles.
3. El usuario selecciona el restaurante.
4. El sistema muestra el menú.
5. El usuario selecciona el artículo y lo paga.
6. El sistema crea orden.

Este escenario de usuario sugiere las siguientes consultas:

- findAvailableRestaurants (deliveryAddress, deliveryTime): recupera los restaurantes que pueden realizar entregas en la dirección de entrega especificada a la hora especificada.
- findRestaurantMenu (id): recupera información sobre un restaurante, incluyendo el menú.

De las dos consultas, findAvailableRestaurants () es probablemente la más significativa arquitectónicamente. Es una consulta compleja que involucra geobúsqueda. El componente de geobúsqueda de la consulta consiste en encontrar todos los puntos (restaurantes) que están cerca de una ubicación, la dirección de entrega. También filtra aquellos restaurantes que están cerrados cuando el pedido necesita estar preparado y recogido. Además, el rendimiento es fundamental, porque la

consulta se ejecuta cada vez que un consumidor desea realizar un pedido. El modelo de dominio de alto nivel y las operaciones de sistema capturan lo que la aplicación hace. Ayudan a impulsar la definición de la arquitectura de la aplicación. El comportamiento de cada operación de sistema se describe en términos del modelo de dominio. Cada operación de sistema importante representa un escenario arquitectónicamente significativo que es parte de la descripción de la arquitectura. Una vez definidas las operaciones de sistema, el siguiente paso es identificar los servicios de la aplicación. Como se mencionó anteriormente, no hay un proceso mecánico a seguir. Sin embargo, existen varias estrategias de descomposición que se pueden utilizar. Cada una ataca el problema desde una perspectiva diferente y utiliza su propia terminología. Pero con todas las estrategias, el resultado final es el mismo: una arquitectura que consta de servicios que se organizan principalmente en torno a conceptos comerciales más que técnicos.

Definición de servicios aplicando el patrón Descomponer por capacidad del negocio:

Una estrategia para crear una arquitectura de microservicios es descomponer por la capacidad del negocio. Un concepto de modelado de arquitectura empresarial, una capacidad empresarial es algo que hace una empresa para generar valor. El conjunto de capacidades para un determinado negocio depende del tipo de negocio. Por ejemplo, las capacidades de una empresa de seguros suelen incluir Suscripción, Gestión de reclamaciones, Facturación, Cumplimiento, etcétera. Las capacidades de una tienda en línea incluyen gestión de pedidos, gestión de inventario, envío, etc.

Ver *http://microservices.io/patterns/decomposition/decompose-by-business-capability.html.*

Las capacidades empresariales definen lo que hace una organización:

Las capacidades comerciales de una organización capturan lo que es el negocio de una organización. Por lo general, son estables, a diferencia de la forma en que una organización lleva a cabo sus negocios, que cambia con el tiempo, a veces dramáticamente. Eso es especialmente cierto hoy, con el rápido y creciente uso de la tecnología para automatizar muchos procesos comerciales. Por ejemplo, no fue hace mucho tiempo que depositaba cheques en su banco entregándoselos a un cajero. Entonces se hizo posible depositar cheques usando un cajero automático. Hoy puedes depositar cómodamente la mayoría de los cheques con su teléfono inteligente. Como puede ver, la capacidad empresarial de Depósito de cheques se ha mantenido estable, pero la forma en que se ha hecho ha cambiado drásticamente.

Identificación de las capacidades empresariales:

Las capacidades comerciales de una organización se identifican analizando el propósito, estructura y procesos comerciales. Se puede pensar en cada capacidad empresarial como un servicio, excepto que está orientado a los negocios en lugar de ser técnico. Su especificación consiste de varios componentes, incluidos insumos, productos y acuerdos de nivel de servicio. Por ejemplo, la entrada a una capacidad de suscripción de seguros es la del consumidor de la aplicación, y las salidas incluyen aprobación y precio. Una capacidad empresarial a menudo se centra en un objeto empresarial particular. Por ejemplo, el objeto de negocio de Reclamaciones es el foco de la capacidad de gestión de Reclamaciones. Una capacidad a menudo se puede descomponer en subcapacidades. Por ejemplo, la gestión de reclamaciones tiene varias subcapacidades, incluida la gestión de información de reclamaciones, revisión y gestión de pagos de reclamaciones. No es difícil imaginar que las capacidades comerciales de FTGO incluyen las siguientes:

- Gestión de proveedores

  - Gestión de mensajería: gestión de la información de mensajería

  - Gestión de información de restaurantes: gestión de menús de restaurantes y otra información, incluida la ubicación y el horario de atención.
- Gestión de consumidores: gestión de información sobre consumidores
- Toma y cumplimiento de pedidos

  - Gestión de pedidos: permite a los consumidores crear y gestionar pedidos.

  - Gestión de pedidos del restaurante: gestión de la preparación de pedidos en un restaurante

  - Logística

  - Gestión de disponibilidad de mensajería: gestión de la disponibilidad de mensajería en tiempo real a los pedidos de entrega

  - Gestión de entregas: entrega de pedidos a los consumidores
- Contabilidad

  - Contabilidad del consumidor: gestión de la facturación de los consumidores

  - Contabilidad de restaurantes: gestión de pagos a restaurantes

  - Contabilidad de mensajería: gestión de pagos a mensajería

Las capacidades de nivel superior incluyen gestión de proveedores, gestión de consumidores, toma y cumplimiento de pedidos y Contabilidad. Probablemente habrá muchos otros niveles de capacidades superiores, incluidas las capacidades relacionadas con el marketing. La mayoría de las capacidades de primer nivel se descomponen en subcapacidades. Por ejemplo, la toma y el cumplimiento de pedidos es descompuesto en cinco subcapacidades.

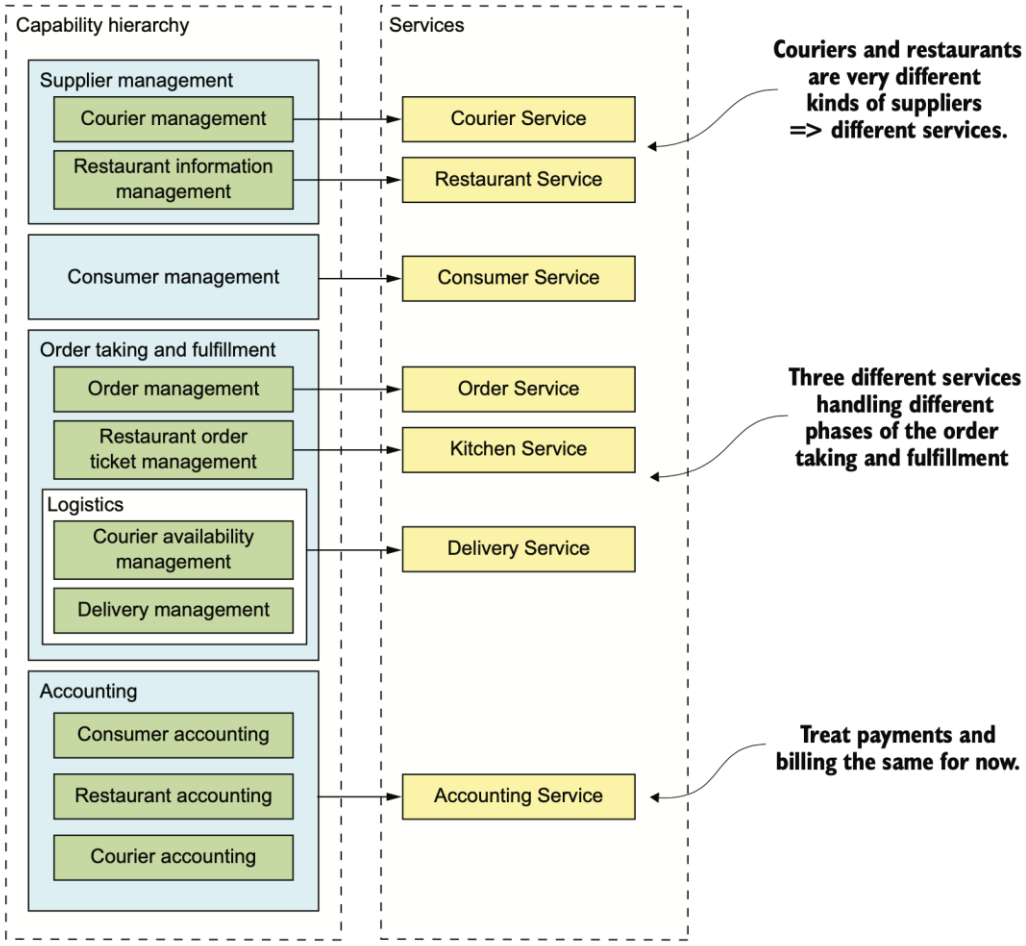
Un aspecto interesante de esta jerarquía de capacidades es que hay tres capacidades de restaurantes relacionadas: gestión de información de restaurantes, gestión de pedidos de restaurantes y contabilidad de restaurantes. Eso es porque representan tres muy diferentes aspectos de las operaciones del restaurante.

De las capacidades empresariales a los servicios:

Una vez que se ha identificado las capacidades comerciales, se define un servicio para cada capacidad o grupo de capacidades relacionadas. La figura siguiente muestra el mapeo de capacidades a los servicios para la aplicación FTGO. Algunas capacidades de alto nivel, como la capacidad de contabilidad se asigna a los servicios. En otros casos, las subcapacidades son mapeadas a servicios.

La decisión de qué nivel de la jerarquía de capacidades se asigna a los servicios, es algo subjetivo. La justificación para este mapeo en particular es la siguiente:

- Se mapearon las subcapacidades de la gestión de proveedores a dos servicios, porque Los restaurantes y los mensajeros son tipos muy diferentes de proveedores.
- Se mapeó la capacidad de toma y cumplimiento de pedidos a tres servicios que son cada uno responsable de las diferentes fases del proceso. Se combinaron las capacidades del mensajero de gestión de disponibilidad y gestión de entrega y se asignaron a un solo servicio porque están profundamente entrelazadas.
- Se mapeó la capacidad de Contabilidad a su propio servicio, porque los diferentes tipos de contabilidad parecen similares.



Más adelante, puede tener sentido separar los pagos (de restaurantes y mensajería) y facturación (de consumidores). Un beneficio clave de organizar los servicios en torno a las capacidades es que, debido a que son estables, la arquitectura resultante también será relativamente estable. Los componentes individuales de la arquitectura pueden evolucionar a medida que cambia el aspecto del negocio, pero la arquitectura permanece sin cambios.

Dicho esto, es importante recordar que los servicios que se muestran en la figura anterior son simplemente el primer intento de definir la arquitectura. Pueden evolucionar con el tiempo a medida que se aprenda más sobre el dominio de la aplicación. En particular, un paso importante en el proceso de definición de la arquitectura es investigar cómo los servicios colaboran en cada uno de los servicios arquitectónicos clave. Se podría, por ejemplo, descubrir que una descomposición en particular es ineficiente debido a una excesiva comunicación entre procesos y que se debería combinar servicios. Por

el contrario, un servicio podría crecer en complejidad al punto en el que valdría la pena dividirlo en varios servicios.

Definición de servicios aplicando el patrón Descomponer por subdominios:

DDD, como se describe en el excelente libro Diseño Dirigido por Dominios de Eric Evans (Addison-Wesley Professional, 2003), es un enfoque para crear aplicaciones de software complejas que se centran en el desarrollo de un modelo de dominio orientado a objetos. Un modo de dominio captura el conocimiento sobre un dominio en una forma que se puede usar para resolver problemas dentro de ese dominio. Define el vocabulario utilizado por el equipo, lo que DDD llama el lenguaje ubicuo. El modelo de dominio se refleja de cerca en el diseño e implementación de la aplicación. DDD tiene dos conceptos que son increíblemente útiles al aplicar la arquitectura de microservicios: subdominios y contextos acotados.

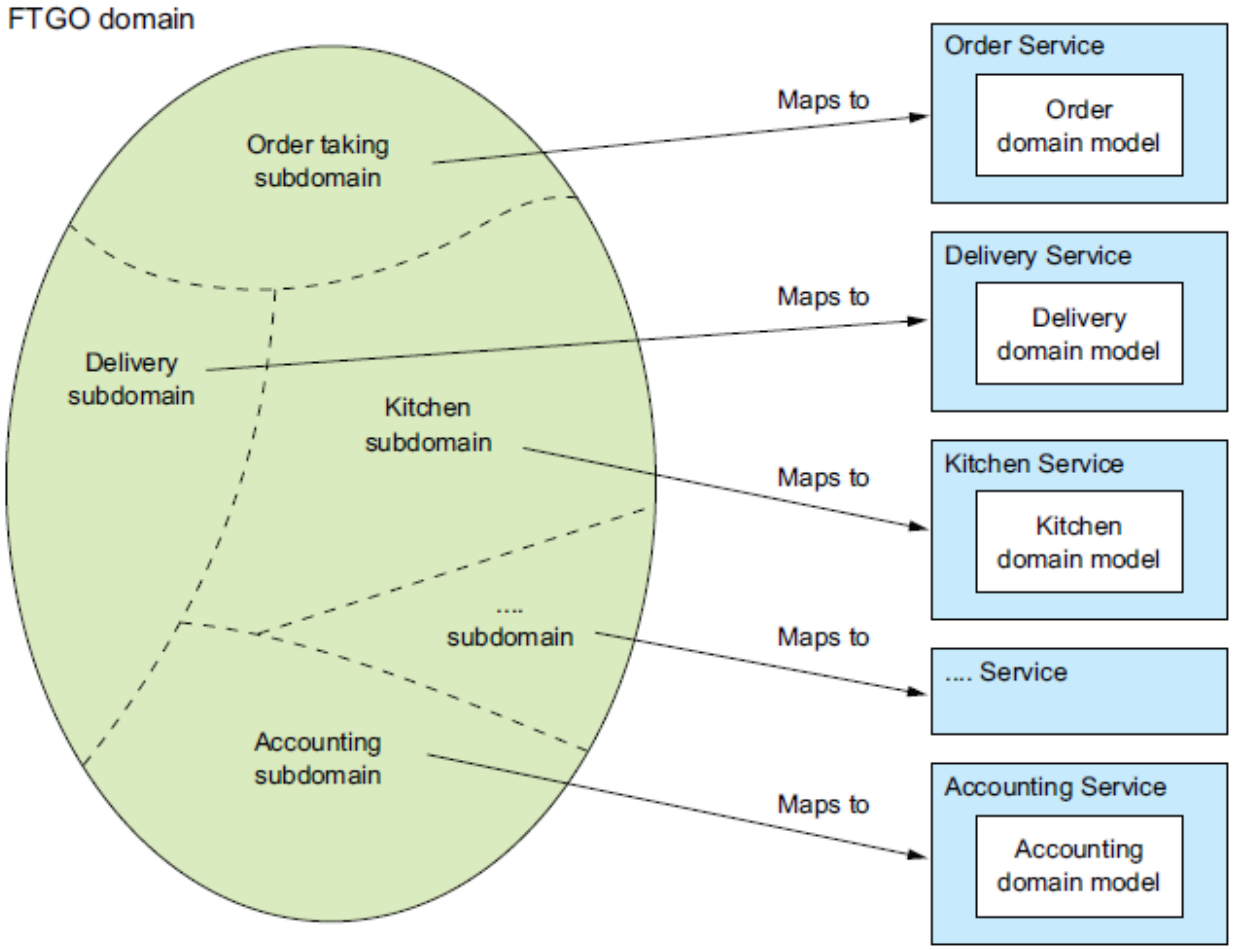
Ver *http://microservices.io/patterns/decomposition/decompose-by-subdomain.html.*

DDD es bastante diferente al enfoque tradicional de modelado empresarial, que crea un modelo único para toda la empresa. En tal modelo habría, por ejemplo, una única definición de cada entidad de negocios, como cliente, pedido, etc. sobre. El problema con este tipo de modelado es lograr que diferentes partes de una organización se pongan de acuerdo sobre un modelo único es una tarea monumental. Además, significa que, desde la perspectiva de una parte determinada de la organización, el modelo es demasiado complejo para sus necesidades. Además, el modelo de dominio puede ser confuso porque diferentes partes de la organización podrían usar el mismo término para diferentes conceptos o diferentes términos para el mismo concepto. DDD evita estos problemas mediante la definición de múltiples modelos de dominio, cada uno con un alcance explícito.

DDD define un modelo de dominio separado para cada subdominio. Un subdominio es una parte del dominio, término de DDD para el espacio del problema de la aplicación. Los subdominios son identificados utilizando el mismo enfoque para identificar las capacidades del negocio: analizar el negocio e identificar las diferentes áreas de especialización. Es muy probable que el resultado final sean subdominios que son similares a las capacidades del negocio. Los ejemplos de subdominios en FTGO incluyen Toma de pedidos, Gestión de pedidos, Gestión de cocina, Entrega, y Finanzas. Como se puede ver, estos subdominios son muy similares a las capacidades de negocio descritas anteriormente.

DDD llama al alcance de un modelo de dominio un contexto limitado. Un contexto limitado incluye los artefactos de código que implementan el modelo. Cuando se usa la arquitectura de microservicios, cada contexto delimitado es un servicio o posiblemente un conjunto de servicios. Se puede crear una

arquitectura de microservicios aplicando DDD y definiendo un servicio para cada subdominio. La siguiente figura muestra cómo los subdominios se asignan a los servicios, cada uno con su propio modelo de dominio.



DDD y la arquitectura de microservicios están en una alineación casi perfecta. El concepto DDD de subdominios y contextos delimitados se asigna muy bien a los servicios dentro de una arquitectura de microservicios. También, el concepto de arquitectura de microservicios de equipos autónomos que poseen servicios está completamente alineado con el concepto de DDD de que cada modelo de dominio le pertenece y es desarrollado por un solo equipo.

Descomponer por subdominios y Descomponer por capacidad empresarial son los dos principales patrones para definir la arquitectura de microservicios de una aplicación. Sin embargo, hay algunas pautas útiles para la descomposición que tienen sus raíces en el diseño orientado a objetos.

Pautas para la descomposición:

Hasta ahora se han analizado las principales formas de definir una arquitectura de microservicios. Pero, también se pueden adaptar y utilizar un par de principios del diseño orientado a objetos al aplicar el patrón de arquitectura de microservicios. Estos principios fueron creados por Robert C. Martin y descrito en su libro clásico Designing Object Oriented C++ Applications Using The Booch Method (Prentice Hall, 1995). El primer principio es el Single Responsibility Principle (SRP) principio de responsabilidad única, para definir las responsabilidades de una clase. El segundo principio es el Common Closure Principle (CCP) principio de agrupación común, para organizar clases en paquetes.

Single Responsibility Principle (SRP) principio de responsabilidad única:

Uno de los principales objetivos de la arquitectura y el diseño de software es determinar las responsabilidades de cada elemento de software. El principio de responsabilidad única es el siguiente: Una clase debe tener solo una razón para cambiar. (Robert C. Martin).

Cada responsabilidad que tiene una clase es una razón potencial para que esa clase cambie. Si una clase tiene múltiples responsabilidades que cambian de forma independiente, la clase no será estable. Al seguir el SRP, se definen clases en las que cada una tiene una sola responsabilidad y de ahí una única razón para el cambio.

Se puede aplicar SRP al definir una arquitectura de microservicios y crear pequeños servicios cohesivos en los que cada uno tiene una única responsabilidad. Esto reducirá el tamaño del servicio y aumentará su estabilidad. La nueva arquitectura FTGO es un ejemplo de SRP en acción. Cada aspecto de llevar comida a un consumidor: toma de pedidos, preparación de pedidos, y entrega — es responsabilidad de un servicio separado.

Common Closure Principle (CCP) principio de agrupación común:

Este principio es el siguiente: Las clases en un paquete deben agruparse juntas frente al mismo tipo de cambios. Un cambio que afecta a un paquete afecta a todas las clases de ese paquete. (Robert C. Martin). La idea es que si dos clases cambian al mismo tiempo debido a la misma razón subyacente, entonces pertenecen al mismo paquete. Quizás, por ejemplo, esas clases implementan un aspecto diferente de una regla de negocio en particular. El objetivo es que cuando esas reglas de negocio cambien, los desarrolladores solo necesitan cambiar el código en una pequeña cantidad de paquetes (idealmente solo uno). Adherirse al CCP mejora significativamente la mantenibilidad de una aplicación.

Se puede aplicar CCP al crear una arquitectura de microservicios y empaquetar componentes que cambian por el mismo motivo en el mismo servicio. Hacer esto minimizará la cantidad de servicios que deben cambiarse y desplegarse cuando algún requerimiento cambie. Idealmente, un cambio solo afectará a un solo equipo y a un solo servicio. CCP es el antídoto contra el antipatrón monolítico distribuido.

SRP y CCP son 2 de los 11 principios desarrollados por Bob Martin. Son particularmente útiles al desarrollar una arquitectura de microservicios. Los nueve principios restantes se utilizan al diseñar clases y paquetes. Para obtener más información sobre SRP, CCP y los demás principios de OOD, consulte el artículo (http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod). Descomposición por capacidad empresarial y por subdominio junto con SRP y CCP son buenas técnicas para descomponer una aplicación en servicios.

Definiendo la API del servicio:

Hasta ahora, se tiene una lista de operaciones de sistema y una lista de servicios potenciales. El siguiente paso es definir la API de cada servicio: sus operaciones y eventos. Una operación de API de servicio existe por una de dos razones: algunas operaciones corresponden a operaciones de sistema. Ellas son invocadas por clientes externos y quizás por otros servicios. Las otras operaciones existen para apoyar la colaboración entre servicios. Estas operaciones solo son invocadas por otros servicios.

Un servicio publica eventos principalmente para permitirle colaborar con otros servicios. Una aplicación también puede usar eventos para notificar a clientes externos. Por ejemplo, podría usar WebSockets para entregar eventos a un navegador.

El punto de partida para definir las APIs de los servicios es mapear cada operación de sistema a un servicio. Después de eso, se decide si un servicio necesita colaborar con otros para implementar una operación de sistema. Si se requiere colaboración, se determina aquellas API que otros servicios deben proveer para soportar la colaboración.

Asignando operaciones de sistema a servicios:

El primer paso es decidir qué servicio es el punto de entrada inicial para una solicitud. Muchas operaciones de sistema se asignan claramente a un servicio, pero a veces la asignación es menos obvia. Considere, por ejemplo, la operación noteUpdatedLocation (), que actualiza la ubicación del mensajero. Por un lado, debido a que está relacionado con los mensajeros, esta operación debería ser asignada al servicio de mensajería. Por otro lado, es el Servicio de Entrega el que necesita la ubicación del mensajero. En este caso, asignar una operación a un servicio que necesita la información proporcionada por la operación es una mejor opción. En otras situaciones, podría tener sentido asignar una operación al servicio que tenga la información necesaria para manejarlo.

Servicio	Operaciones
Servicio al cliente	createConsumer()
Servicio de órdenes	createOrder()
Servicio de restaurante	findAvailableRestaurants()
Servicio de cocina	- accepOrder() - noteOrderReadyForPickup()
Servicio de entrega	- noteUpdateLocation() - note DeliveryPickedUp() - noteDeliveryDelivered()

Después de haber asignado las operaciones a servicios, el siguiente paso es decidir cómo los servicios colaboran para manejar cada operación de sistema.

Determinar las APIs requeridas para soportar la colaboración entre servicios:

Algunas operaciones de sistema son manejadas completamente por un solo servicio. Por ejemplo, en la aplicación FTGO, el Servicio al cliente maneja la operación createConsumer() completamente por sí mismo. Pero otras operaciones del sistema abarcan múltiples servicios. Los datos necesarios para manejar una de esas solicitudes podría, por ejemplo, estar disperso en varios servicios. Por ejemplo, para implementar la operación createOrder(), el servicio de órdenes debe invocar los siguientes servicios para verificar sus pre-condiciones previas y hacer que las post-condiciones se hagan realidad:

- Servicio al cliente: verificar que el consumidor pueda realizar un pedido y obtener su Información de pago.

- Servicio de restaurante: validar los ítems del pedido, verificar que la dirección/hora está dentro del área de servicio del restaurante, verificar que cumple el pedido mínimo y obtener precios para los ítems del pedido.
- Servicio de cocina: crear el ticket.
- Servicio de contabilidad: autorizar la tarjeta de crédito del consumidor.

De manera similar, para implementar la operación de sistema acceptOrder(), el servicio de cocina debe invocar al servicio de entrega para programar un mensajero para entregar el pedido. Con el fin de definir completamente las API de servicio, se necesita analizar cada operación de sistema y determinar qué colaboración se requiere.

Servicio	Operaciones	Colaboradores
Servicio al cliente	`verifyConsumerDetails()`	
Servicio de órdenes	`createOrder()`	☐ Servicio al cliente `verifyConsumerDetails()` ☐ Servicio de restaurante `verifyOrderDetails()` ☐ Servicio de cocina `createTicket()` ☐ Servicio de contabilidad `authorizeCard()`
Servicio de restaurante	☐ `findAvailableRestaurants()` ☐ `verifyOrderDetails()`	
Servicio de cocina	☐ `createTicket()` ☐ `acceptOrder()` ☐ `noteOrderReadyForPickup()`	☐ Servicio de entrega `scheduleDelivery()`
Servicio de entrega	☐ `scheduleDelivery()` ☐ `noteUpdatedLocation()` ☐ `noteDeliveryPickedUp()` ☐ `noteDeliveryDelivered()`	
Servicio de contabilidad	☐ `authorizeCard()`	

Hasta ahora, se han identificado los servicios y las operaciones que implementa cada servicio. Pero es importante recordar que la arquitectura que se ha esbozado es muy abstracta. No se ha seleccionado ninguna tecnología IPC específica. Además, aunque el término operación sugiere algún tipo de mecanismo sincrónico de IPC basado en solicitud/respuesta, se deberá tener en cuenta que la mensajería asincrónica juega un papel importante.

Tomado y traducido de: Richardson Chris, Microservices Patterns with examples in java, Shelter Island NY, Manning Publications Co, 2019.

API REST

https://www.redhat.com/es/topics/api/what-is-a-rest-api

https://www.bbvaapimarket.com/es/mundo-api/api-rest-que-es-y-cuales-son-sus-ventajas-en-el-desarrollo-de-proyectos/

https://geekytheory.com/que-es-una-api-rest-y-para-que-se-utiliza

GraphQL

https://graphql.org/

https://www.graphql.com/

https://www.redhat.com/es/topics/api/what-is-graphql

https://www.paradigmadigital.com/dev/graphql-todos-uno-uno-todos/