7. Construcción de Servicios Multi-tenant

Es aquí donde podemos comenzar a pensar en cómo aplicaremos el multi-tenancy al diseño e implementación de los servicios que darán vida a nuestra aplicación.

Aislamiento, noisy neighbor, particionamiento de datos: estos representan nuevos parámetros que deberás incorporar en el diseño de tus servicios.

Diseño de servicios multi-tenant

Servicios en entornos de software clásicos

Para comprender mejor esta dinámica, comencemos analizando una aplicación clásica en la que toda la huella de recursos de una aplicación se instala, despliega y gestiona de forma independiente.

Verás que los servicios están completamente dedicados a clientes individuales. Al diseñar servicios para estos entornos, tu enfoque se centra principalmente en encontrar un buen conjunto de servicios que pueda satisfacer las necesidades de escalabilidad, rendimiento y tolerancia a fallos de un único cliente.

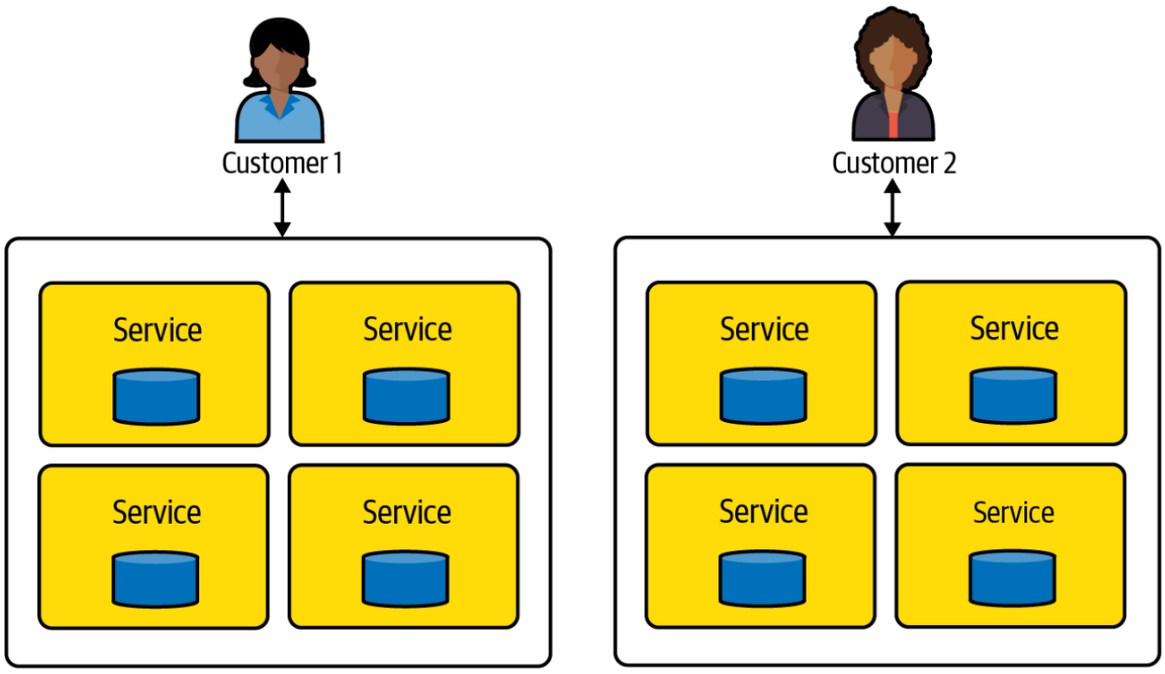

Figure 7-1. Services in a classic software environment

Servicios en entornos multi-tenant pooled

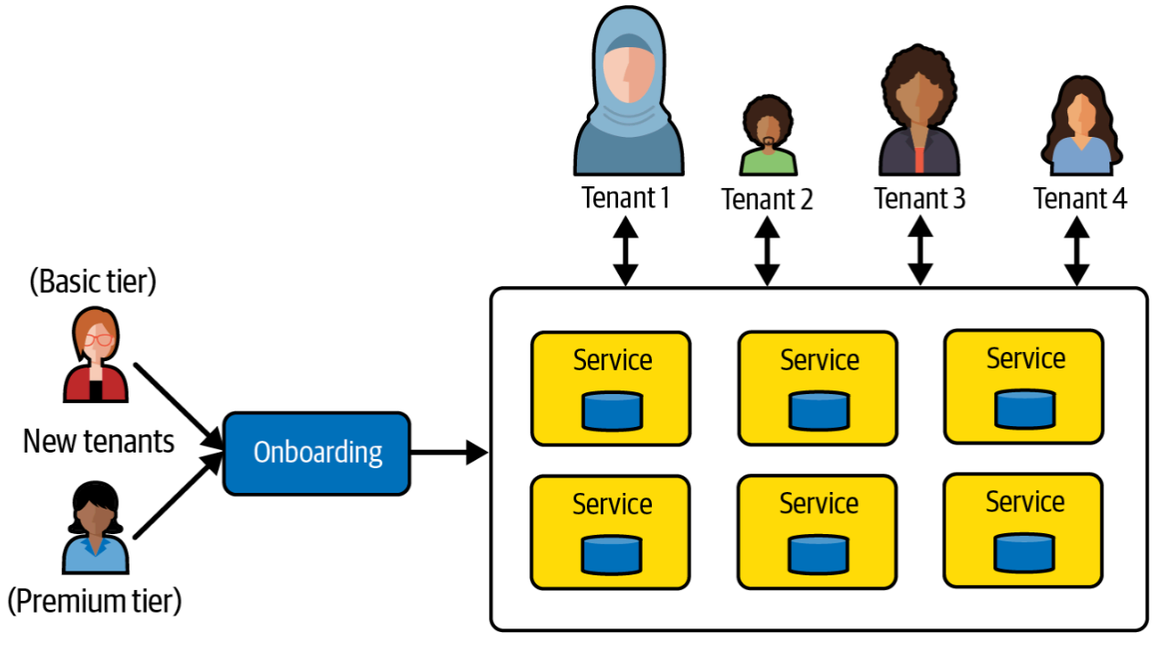

Figure 7-2. Services in a pooled multi-tenant environment

A primera vista, puede parecer que lo único que ha cambiado es el número de tenants que consumen estos servicios. Sin embargo, el hecho de que todos estos tenants estén utilizando estos servicios al mismo tiempo como recursos compartidos tiene

implicaciones significativas en cómo abordas el tamaño, la descomposición y la huella de recursos de cada uno de estos servicios.

A la izquierda, también he representado el onboarding de nuevos tenants. Esto pretende transmitir la idea de que pueden incorporarse nuevos tenants a tu entorno en cualquier momento. Hay poco que puedas hacer para anticipar la carga de trabajo y los perfiles de estos nuevos tenants.

Los servicios en este entorno, compartidos por todos los tenants, deben anticipar de alguna manera todas las necesidades de escalabilidad, rendimiento y consumo de cada uno de los perfiles de tenant.

Deberás estar muy atento a asegurarte de que estos tenants no estén generando condiciones de noisy neighbor, donde un tenant afecta la experiencia de otro.

Aquí es esencialmente donde los beneficios de la infraestructura compartida chocan con las realidades de soportar un panorama de perfiles de consumo de clientes en constante cambio.

Para algunos, esto lleva al sobreaprovisionamiento de recursos para compensar las necesidades cambiantes y los perfiles de carga, lo que va exactamente en contra de los objetivos de eficiencia y economías de escala asociados con el modelo de negocio SaaS.

Extensión de las mejores prácticas existentes

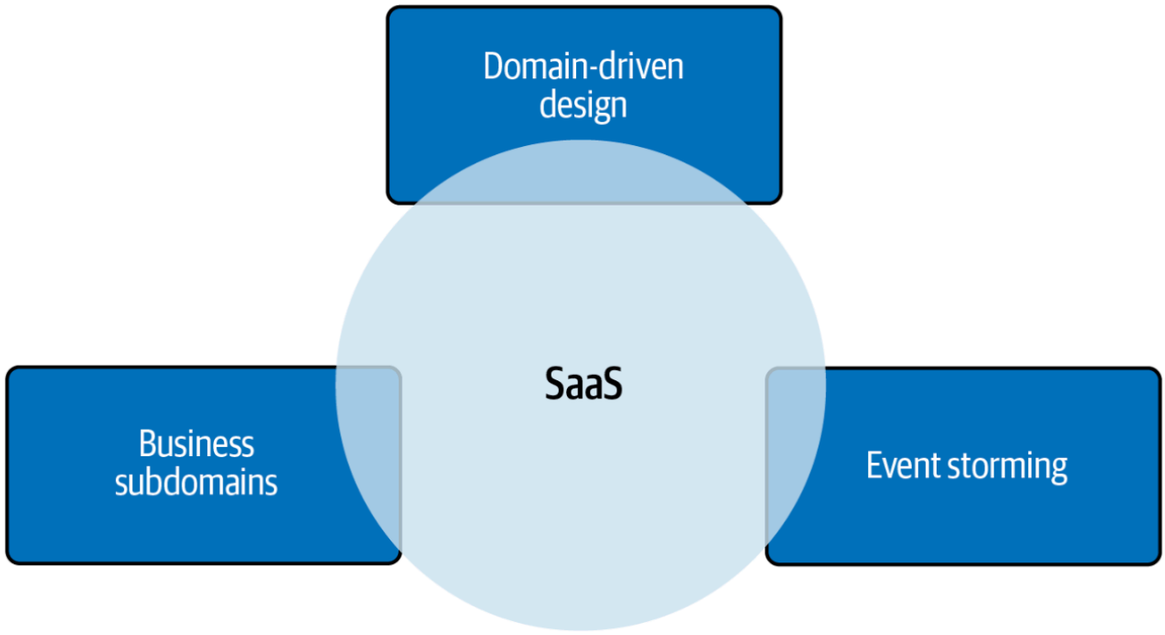

Figure 7-3. Blending service design methodologies

Si bien estas metodologías tienen un valor claro, no siempre incluyen análisis de las realidades multi-tenant que deben considerarse en el diseño de servicios multi-tenant.

Para ilustrar este punto, profundicemos en algunas de las áreas comunes donde el multi-tenancy podría influir en el diseño de tus servicios.

Mitigación del noisy neighbor

Los desarrolladores SaaS generalmente deben considerar cómo y dónde los usuarios podrían imponer carga en el sistema, lo que podría saturarlo o degradar su rendimiento.

Un noisy neighbor en un entorno SaaS tiene el potencial de tumbar todo el sistema o, en el mejor de los casos, degradar la experiencia del resto de los tenants en tu entorno multi-tenant.

El noisy neighbor puede manifestarse de múltiples formas en un entorno multi-tenant.

- Puede haber operaciones específicas en tu entorno que tengan alta latencia o que consuman recursos en patrones con alto potencial de generar cuellos de botella.
- Puede haber áreas donde ciertos perfiles de tenant tiendan a saturar un conjunto particular de servicios de tu sistema.

El desafío básico suele girar en torno al escalado. **Nuestro enfoque se centra en aquellos escenarios donde el escalado horizontal por sí solo puede no ser suficientemente eficaz o eficiente para abordar las realidades multi-tenant de tu entorno.**

Tu enfoque principal para resolver esto podría ser simplemente escalar el servicio horizontalmente, con el riesgo de sobreaprovisionarlo, esperando que esto limite cualquier impacto en cascada sobre tus tenants.

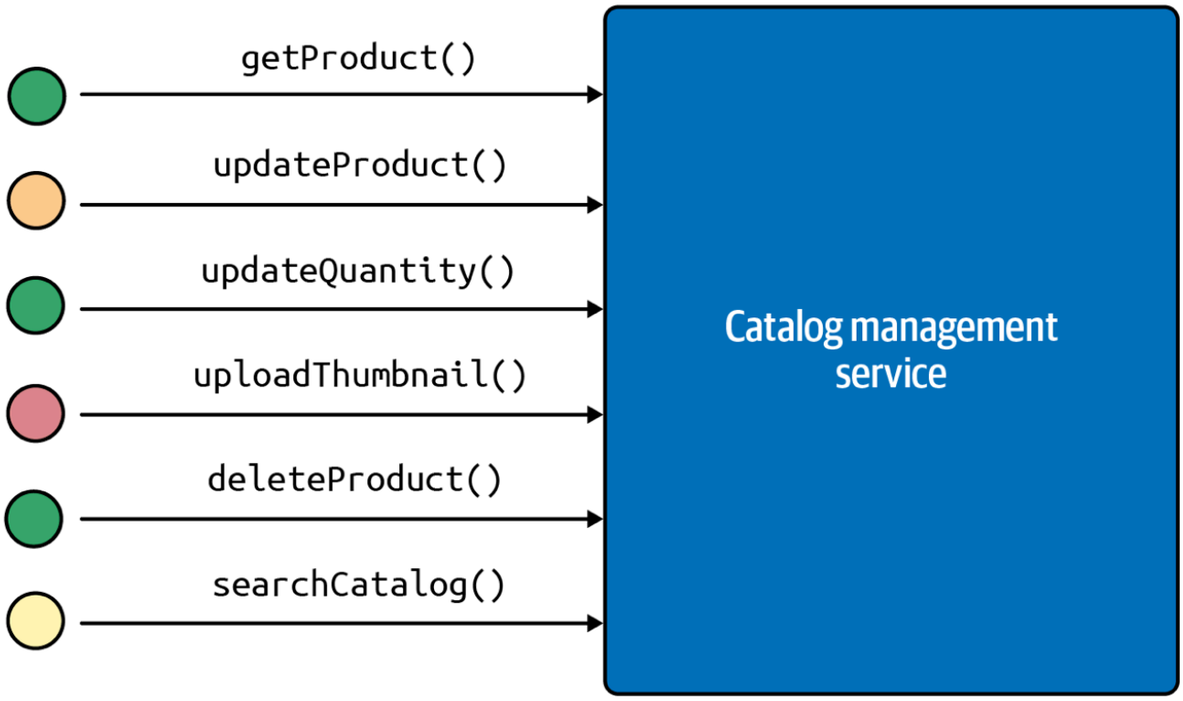

Figure 7-4. A noisy neighbor bottleneck

En esencia, podrías estar escalando todo el servicio cuando solo una operación del mismo necesita mayor rendimiento. **La mejor opción podría ser evaluar si esta operación podría extraerse y trasladarse a un servicio independiente** donde pueda escalar de forma más proporcional a la actividad del tenant, sin absorber las ineficiencias de escalar todo el servicio de gestión de catálogo para resolver los problemas de rendimiento.

A medida que identifiques tus servicios, busca aquellas áreas que se destaquen como posibles candidatos a generar noisy neighbor.

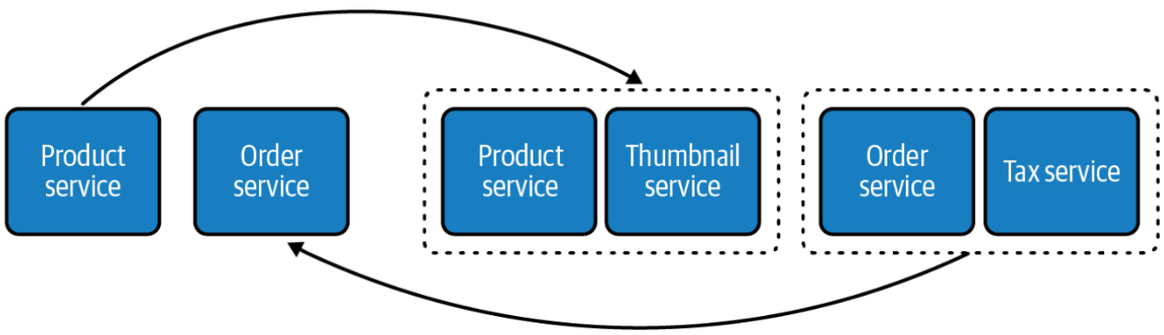

Figure 7-5. The noisy neighbor decomposition mindset

Es válido preguntarse si este enfoque es realmente exclusivo de SaaS. La respuesta es no. Como regla general, cualquier entorno debería buscar formas de mejorar el rendimiento y la escalabilidad mediante servicios más granulares. Un servicio con escalado ineficiente, cuellos de botella o sobreaprovisionado tiene mayor probabilidad de manifestarse en un entorno multi-tenant, lo que puede tener un impacto profundo en tus tenants y en el perfil operacional de tu entorno SaaS.

Identificación de servicios siloed

Cuando decides aislar un servicio en un modelo siloed, generalmente lo haces para satisfacer una necesidad específica del sistema o del tenant.

Por supuesto, cada vez que aislas un recurso, estás haciendo una concesión que puede impactar la complejidad operacional, de costos, de despliegue y de gestión de tu entorno. **Por lo tanto, si vas a tener recursos siloed, lo ideal es limitar el número de servicios que necesitan desplegarse en silos.**

Podrías, por ejemplo, aplicar alguna variación de esta misma mentalidad por razones de cumplimiento normativo, noisy neighbor o rendimiento general, dividiendo servicios más grandes para tener un control más granular sobre qué está en un modelo siloed y qué no.

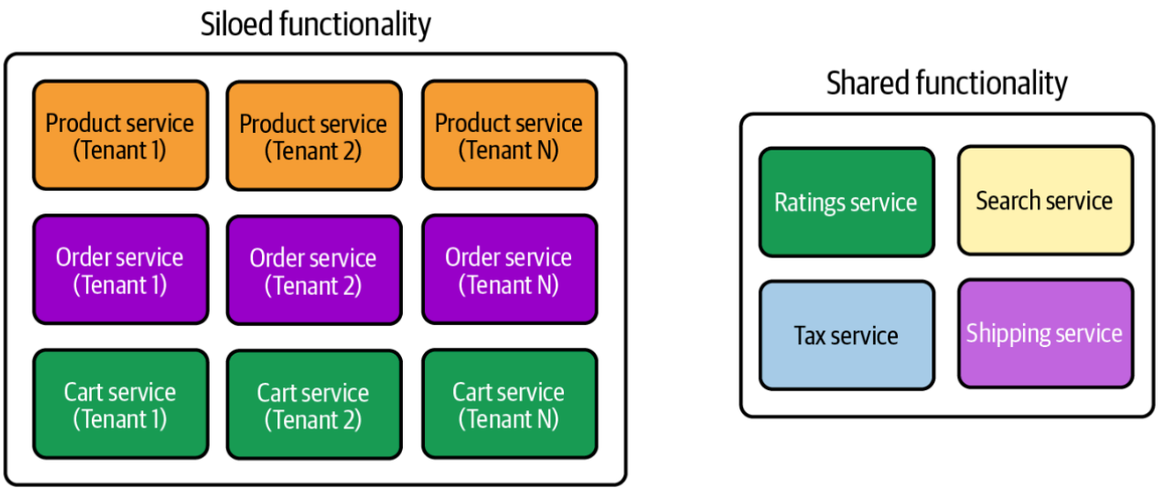

Figure 7-7. Aligning services with silo/pool requirements

Esta estrategia de siloed debería convertirse en una parte central de tu mentalidad de diseño de servicios, identificando los casos de uso y requisitos que podrían justificar el despliegue de un servicio en un modelo siloed. Esta lista generalmente incluye cumplimiento normativo, aislamiento, seguridad, niveles de servicio y rendimiento.

También es importante señalar que puedes aislar servicios en un modelo siloed basándote exclusivamente en tus propias necesidades operacionales internas. Por ejemplo, puede haber algunos servicios que simplemente no puedan satisfacer las demandas de los tenants en un modelo pooled.

Para mí, lo principal que busco evitar es simplemente mover servicios a un silo sin cuestionarme si existen formas más creativas de descomponer el servicio.

Para sistemas con una población más pequeña de tenants, esta podría ser una buena estrategia. **Sin embargo, para sistemas con un mayor número de tenants, esto se volvería difícil de escalar y soportar.**

La influencia de las tecnologías de cómputo

También hay escenarios en los que la tecnología de cómputo que utilizas puede influir en la huella de recursos de tus servicios. Los contenedores, el modelo serverless y otros elementos de cómputo pueden introducir sus propias consideraciones en tu diseño multi-tenant.

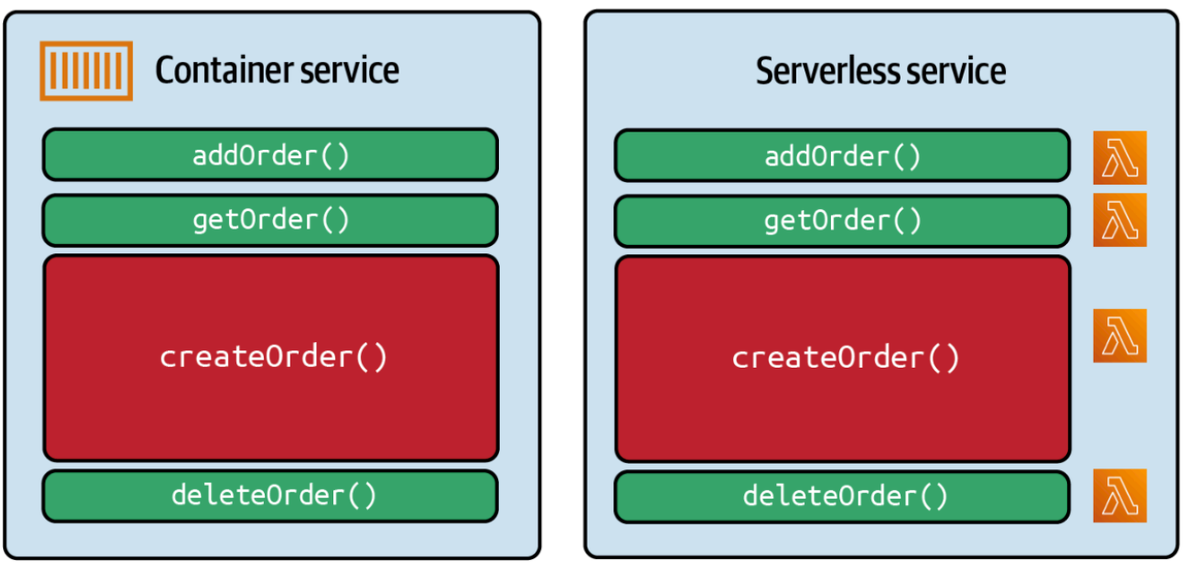

Figure 7-8. Compute and service design

Al analizar el despliegue basado en contenedores, podríamos considerar cómo refactorizar este servicio para mejorar la eficiencia de escalado general de nuestro entorno. Con contenedores, toda esta funcionalidad se empaqueta, escala y despliega de forma colectiva, de modo que el contenedor se convierte en nuestra unidad de escalado.

Con el modelo serverless, en cambio, cada una de las operaciones que se despliegan como parte de nuestro servicio representa una función independiente que puede desplegarse, gestionarse y escalarse de forma autónoma. **Así, si createOrder() o cualquier otra operación recibe un nivel de carga desproporcionado, esa función escala por sí sola.** Esto representa una de las ventajas significativas del modelo serverless, donde el escalado es más granular y pasa a ser responsabilidad de otro gestionar.

La influencia de los aspectos de almacenamiento

Al diseñar servicios, también solemos pensar en el alcance y la naturaleza de los datos a los que accederán y que gestionarán estos servicios. Dado que se espera que cada servicio encapsule los datos que gestiona, debes considerar cómo se consumirán

estos datos.

Ahora, al analizar los servicios multi-tenant, tenemos nuevos factores que añadir a nuestras consideraciones de diseño. Los datos multi-tenant pueden almacenarse en un modelo siloed, donde cada tenant tiene su propia estructura de almacenamiento dedicada, o podrían almacenarse en un modelo pooled, donde los datos de los tenants se mezclan dentro de un elemento de almacenamiento compartido.

También puede ser necesario pensar en cómo las diferentes operaciones sobre tus datos escalarán eficazmente cuando los clientes compiten por un recurso de almacenamiento compartido. Imagina miles de tenants consultando simultáneamente alguna base de datos relacional que almacena todos sus datos en una tabla compartida.

En muchos aspectos, las necesidades de almacenamiento de tu servicio también deben considerar muchos de los factores que dieron forma a la discusión sobre noisy neighbor y siloed expuesta anteriormente. Con el almacenamiento, entramos dentro del servicio y pensamos en cómo los requisitos multi-tenant, los perfiles de tenant y las cargas de trabajo podrían moldear la huella de recursos de tu servicio.

La conclusión principal es que el almacenamiento puede —y con frecuencia lo hace— desempeñar un papel significativo en la descomposición de tus servicios, por lo que al sentarte a identificar tus servicios y su alcance y granularidad, querrás asegurarte de darle al almacenamiento la atención que merece.

Uso de métricas para analizar el diseño

El diseño de tus servicios estará en constante evolución. Nuevas funcionalidades, nuevos tenants, nuevos niveles de servicio y nuevas cargas de trabajo harán que tu equipo evalúe continuamente el rendimiento, la escalabilidad y la eficiencia de tu arquitectura multi-tenant.

Es posible que puedas usar datos básicos de monitoreo para sacar conclusiones generales sobre el comportamiento de tu sistema, pero estos datos normalmente no te permitirán evaluar los patrones de consumo y actividad de tenants y niveles de servicio individuales.

Obtener estas métricas implica añadir instrumentación a tus servicios que expongan los datos necesarios para analizar el perfil operacional de tus servicios.

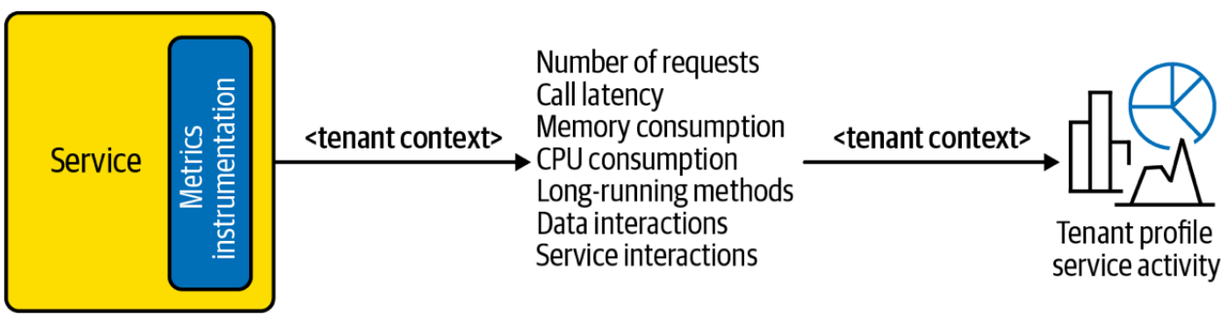

Figure 7-9. Surfacing tenant-aware service metrics

Lo que elijas instrumentar aquí dependerá de la naturaleza de los servicios y de los datos que mejor caractericen su actividad. **Todos los datos registrados incluirán como mínimo el contexto del tenant y el nivel de servicio del tenant (si estás usando niveles de servicio).**

Finalmente, a la derecha hay un espacio reservado para la agregación y el análisis de estos datos. Es aquí donde usarás la herramienta de tu elección para analizar los datos de métricas y evaluar el perfil de ejecución de tus servicios.

Dentro de los servicios multi-tenant

Como regla general, **les digo a los equipos que deben hacer todo lo posible para limitar el conocimiento que tiene un desarrollador sobre el multi-tenancy cuando está implementando las capacidades de negocio de su aplicación SaaS.**

Nuestro enfoque, entonces, se centra en las estrategias y técnicas que puedes introducir para limitar la sobrecarga que los desarrolladores SaaS absorberán al implementar servicios multi-tenant.

Como arquitecto multi-tenant, debes encontrar una forma de introducir el contexto de tenant y el soporte para los elementos multi-tenant sin añadir complejidad innecesaria a la experiencia del desarrollador.

Extracción del contexto de tenant

Antes de poder siquiera pensar en aplicar el contexto de tenant, debemos pensar en cómo ese contexto de tenant llega a nuestros servicios. Esto comienza revisando los temas de identidad y autenticación que se analizaron anteriormente.

Como recordarás, el JWT se incrusta como encabezado dentro de cada solicitud HTTP que se envía a tus servicios. Este token se pasa como lo que se conoce como "bearer token". Para tu servicio, indica que estás autorizando al sistema a realizar una operación en nombre del tenant asociado con ese bearer token.

Es importante señalar que este token está codificado y firmado, por lo que necesitaremos desempaquetarlo para acceder a los claims que nos interesan.

```
def query_orders(self, status):
  # get tenant context
  auth_header = request.headers.get('Authorization')
  token = auth_header.split(" ")
  if (token[0] != "Bearer")
    raise Exception('No bearer token in request')
  bearer_token = token[1]
  decoded_jwt = jwt.decode(bearer_token, "secret",
                  algorithms=["HS256"])
  tenant_id = decoded_jwt['tenantId']
  tenant_tier = decoded_jwt['tenantTier']

  # query for orders with a specific status
  logger.info("Finding orders with the status of %s", status)
  ...
```

En este ejemplo particular, asumo que tu servicio sería responsable de decodificar cada token. Sin embargo, existen otras opciones. Podrías, por ejemplo, tener un API gateway que se sitúe frente a todos tus servicios, procesando cada solicitud entrante. **Este gateway podría desempaquetar estos JWTs, acceder al contexto de tenant e inyectarlo en cada servicio.**

La clave es que en algún punto al inicio de estas solicitudes, necesitarás código que pueda encargarse de adquirir este contexto de tenant para cada solicitud (ya sea que esté en caché o extraído del JWT cada vez).

Logs y métricas con contexto de tenant

El logging representa uno de esos mecanismos fundamentales que usará cada servicio, emitiendo mensajes que crean un rastro de auditoría informativo y de depuración esencial para solucionar problemas y analizar la actividad en tu sistema.

Por defecto, si no hicieras nada con tus logs, contendrían una colección heterogénea de información sin correlación con ningún tenant específico. Esto haría casi imposible construir una visión de la actividad de cualquier tenant en particular.

La buena noticia es que ahora que tenemos el contexto de tenant a nuestro alcance, podemos inyectarlo en nuestros mensajes de log.

```
def query_orders(self, status):
  # get tenant context
  auth_header = request.headers.get('Authorization')
  token = auth_header.split(" ")
  if (token[0] != "Bearer")
    raise Exception('No bearer token in request')
  bearer_token = token[1]
  decoded_jwt = jwt.decode(bearer_token, "secret",
                  algorithms=["HS256"])
  tenant_id = decoded_jwt['tenantId']
  tenant_tier = decoded_jwt['tenantTier']

  # query for orders with a specific status
  logger.info("Tenant: %s, Tier: %s, Find orders with status %s
              tenant_id, tenant_tier, status);
  ...
```

Este contexto se añadiría a todos los mensajes de logging dentro de tus servicios, incorporando los datos que brindan a los equipos perspectivas mucho más ricas sobre el comportamiento específico de sus tenants.

Esta misma mentalidad de logging también debe aplicarse a la instrumentación de métricas de tu arquitectura multi-tenant. Sí, queremos logs para construir una visión forense de la actividad del tenant, pero también necesitamos datos utilizados por el negocio para perfilar el consumo y la actividad de los tenants, que no encajan del todo en el perfil operacional de los mensajes de log.

Aquí estás perfilando cómo tu servicio influye en la experiencia del tenant y rastreando tu capacidad para medir un conjunto de métricas clave que los equipos técnicos y de negocio pueden usar para evaluar la eficacia, agilidad, eficiencia del sistema, entre otros aspectos.

```
def query_orders(self, status):
  # get tenant context
  ...
  tenant_id = decoded_jwt['tenantId']
  tenant_tier = decoded_jwt['tenantTier']

  # query for orders with a specific status
  logger.info("Tenant: %s, Role: %s, Finding orders with status
              tenant_id, tenant_role, status);
  try:
    start_time = time.time()
    response = ddb.query(
                TableName = "order_table",
                KeyConditionExpression = Key(status).eq(status
    duration = (time.time() - start_time)
    message = {
                "tenantId": tenant_id,
                "tier": tenant_tier,
                "service": "order",
                "operation": "query_orders",
                "duration": duration
              }
    firehose = boto3.client('firehose')
    firehose.put_record(
      DeliveryStreamName = "saas_metrics",
      Record = message
    )
  except ClientError as err:
    logger.error(
      "Tenant: %s, Find order error, status: %s. Info: %s: %s",
      tenant_id, status,
      err.response['Error']['Code'],
      err.response['Error']['Message'])
    raise
  else:
    return response['Items']
```

Ahora necesito publicar esta métrica en algún servicio que pueda ingerirla y agregarla. Para este ejemplo, usé un pipeline de datos en streaming de AWS (Amazon Kinesis Data Firehose) para ingerir mis datos de métricas, construyendo el cliente de Firehose y llamando al método put_record() para enviar el evento de métrica al servicio.

La mayor parte del esfuerzo se destinará a determinar qué quieres capturar y cómo introducirás el código que publica tus datos de métricas. La inversión es pequeña si es adoptada ampliamente por tus equipos, pero el retorno puede ser sustancial.

El desafío de contar la historia de las métricas es que no existe un enfoque único y universal que todos deban aplicar a sus servicios. El valor es claro, pero los detalles específicos son difíciles de precisar. **Esto a menudo debe ser impulsado por tu propia necesidad de identificar las métricas que añadirán más valor para tu negocio.**

También diré que algunas de las empresas SaaS más eficaces son aquellas que priorizan las métricas y trabajan para identificar los insights que mejor informarán su capacidad de evaluar la experiencia interna y externa de sus sistemas.

Acceso a datos con contexto de tenant

La forma más natural y sencilla de aplicar el contexto de tenant es agregar el tenant a los parámetros que forman parte de tu búsqueda.

Asumamos, por un momento, que tienes un modelo de base de datos pooled donde los datos de tus tenants están mezclados en la misma tabla. Cuando los datos están en modo pooled, simplemente podemos añadir una clave TenantId a nuestra tabla de pedidos que asociará cada pedido con un tenant específico.

Este identificador de tenant se convertirá en la clave de nuestra tabla. Esto significa que el estado que estábamos usando ahora se convertirá en un parámetro de búsqueda secundario que devuelve todos los pedidos de un tenant que coincidan con el estado indicado.

Empecé con el caso de uso más simple. Donde la discusión sobre el acceso a datos se vuelve más interesante es cuando comenzamos a pensar en las diversas combinaciones de estrategias de almacenamiento que tu servicio podría necesitar soportar. Supongamos, por ejemplo, que tu sistema ofrece almacenamiento diferenciado para distintos niveles de servicio.

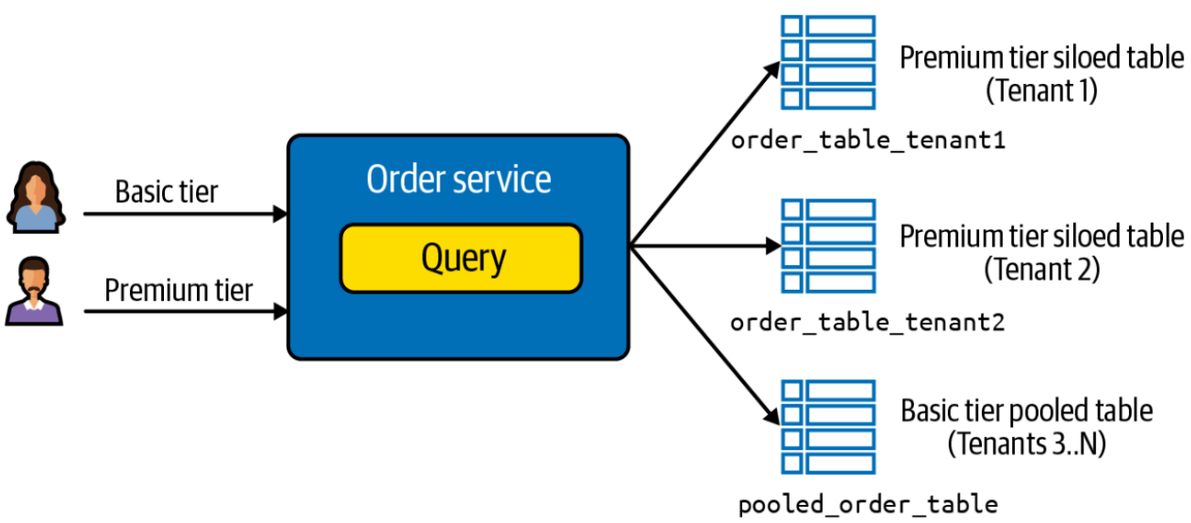

Figure 7-10. Supporting tiered storage models

En algún punto dentro del código de tu servicio, debes tener lógica que examine el nivel de servicio de cada tenant para determinar qué tabla se usará para procesar sus solicitudes. Y, dependiendo de cómo estén almacenados sus datos, puede que necesites múltiples rutas de ejecución dentro de tu servicio para soportar la identificación e interacción con los datos de pedidos del tenant.

Para que esto funcione, necesitaremos esencialmente añadir alguna operación de mapeo a nuestra consulta para resolver el nombre de la tabla que se usará (basándose en el nivel de servicio de un tenant).

```
response = ddb.query(
              TableName = getTenantOrderTable(tenant_id, tenant
              KeyConditionExpression = Key('TenantId').eq(tenar
              FilterExpression=Attr('status').eq(status))

# helper function to get generate tier-based table name
def getTenantOrderTableName(tenant_id, tenant_tier):
  if tenant_tier == BASIC_TIER:
        table_name = "pooled_order_table"
  elif tenant_tier == PREMIUM_TIER:
        table_name = "order_table_" + tenantId
  return table_name
```

Este enfoque, sin embargo, presupone que las tablas para tenants de nivel básico y premium serán idénticas. En su mayor parte, serían iguales; sin embargo, nuestros tenants pooled dependen de una clave TenantId que se usa para acceder a los pedidos de tenants individuales. Esta clave no tiene valor ni significado en las tablas siloed. Muchos equipos mantendrán esta clave en sus tablas siloed simplemente para evitar tener que soportar comportamientos adicionales específicos.

Soporte del aislamiento de tenant

El ejemplo de acceso a datos que acabamos de cubrir se basa en insertar el contexto de tenant en nuestra consulta para limitar los datos a un tenant determinado. Sin embargo, en entornos multi-tenant —donde el aislamiento de tenant es esencial para la confianza de tus tenants— filtrar el acceso a datos por tenant realmente no es suficiente.

Es fundamental que tracemos una línea clara entre las estrategias que se usan para particionar y acceder a los datos y las estrategias que se usan para hacer cumplir el aislamiento de tenant.

- Cómo se almacenan y acceden los datos es lo que consideraríamos tu estrategia de "particionamiento de datos".
- Cómo protegemos los recursos (incluidos los datos) del acceso entre tenants se denomina "aislamiento de tenant".

> Cuando hablamos de aislar los recursos del tenant, estamos hablando de las medidas que utilizamos para rodear el código dentro de nuestros servicios y asegurarnos de que los desarrolladores no crucen los límites del tenant, ya sea intencional o involuntariamente.

El objetivo es que tu código adquiera de alguna manera un contexto de aislamiento antes de acceder a cualquier recurso, y que use ese contexto para limitar el acceso a recursos al tenant actual. Con este contexto aplicado, cualquier intento de interactuar con un recurso quedará restringido únicamente a aquellos recursos que pertenezcan al tenant actual.

Para este ejemplo particular donde accedemos a DynamoDB, podemos alcanzar nuestros objetivos de aislamiento configurando nuestra sesión con un conjunto de credenciales que limitarán el acceso a datos según el contexto del tenant.

Nuestro objetivo, entonces, es restringir el acceso de este cliente para cada solicitud que intente obtener pedidos, inicializando el cliente con un scope que incluya el contexto del tenant solicitante.

```
def query_orders(self, status):
  # get database client (DynamoDB) with tenant scoped credentia
  sts = boto3.client('sts')

  # get credentials based on tenant scope policy
  tenant_credentials = sts.assume_role(
    RoleArn = os.environ.get('IDENTITY_ROLE'),
    RoleSessionName = tenant_id,
    Policy = scoped_policy,
    DurationSeconds = 1000
  )

  # get a scoped session using assumed role credentials
  tenant_scoped_session = boto3.Session(
    aws_access_key_id =
      tenant_credentials['Credentials']['AccessKeyId'],
    aws_secret_access_key =
      tenant_credentials['Credentials']['SecretAccessKey'],
    aws_session_token =
      tenant_credentials['Credentials']['SessionToken']
  )
  # get database client with tenant scoped credentials
  ddb = tenant_scoped_session.client('dynamodb')
    ...
```

- Primero, observa que nuestro primer bloque de código se enfoca en obtener un conjunto más restringido de credenciales con scope basado en el identificador del tenant actual. En este ejemplo particular, nos mantenemos dentro de la familia de servicios de AWS, usando el AWS Security Token Service (STS) para facilitar este proceso de restricción de scope.
- No profundizaremos en los detalles de esta política aquí, pero solo debes saber que esencialmente restringe el acceso únicamente a aquellos elementos en la base de datos que coincidan con un ID de tenant determinado. Entonces, cuando llamo a la función assume_role() y proporciono mi política e identificador de tenant (extraído del JWT), este servicio devolverá un conjunto de credenciales que limitará el acceso únicamente a los elementos que pertenecen al tenant actual. Estas credenciales se almacenan en la variable tenant_credentials.
- Una vez que tenemos estas credenciales, podemos declarar e inicializar una sesión con los valores de credenciales específicos devueltos por nuestra llamada a assume_role().
- Solo queda declarar nuestro cliente de DynamoDB (como lo hicimos antes). Sin embargo, el cliente ahora se crea usando la variable tenant_scoped_session. Ahora, cuando invocamos el comando query usando este cliente, heredará las políticas de scope y las aplicará a cualquier llamada realizada con este cliente.

Ahora, sin importar qué valor o configuración ponga un desarrollador en su consulta, el sistema impedirá que el servicio acceda a datos que no sean válidos para el tenant actual.

Las políticas y el enfoque de aislamiento que hemos cubierto están pensados para aplicarse en cualquier recurso que pueda estar gestionando o tocando elementos específicos del tenant. Si tienes colas, por ejemplo, esas colas pueden requerir alguna forma de aislamiento.

Ocultamiento y centralización de los detalles multi-tenant

También puedes imaginar cómo tener este código en cada servicio sería ineficiente, distribuyendo conceptos y elementos comunes a cada servicio en mi sistema.

Aquí es donde ponemos en práctica nuestras habilidades básicas de desarrollo y buscamos las oportunidades naturales para mover estos conceptos fuera de nuestros servicios y hacia bibliotecas que puedan ocultar gran parte del detalle que hemos estado cubriendo.

```
def get_tenant_context(request):
  auth_header = request.headers.get('Authorization')
  token = auth_header.split(" ")
  if (token[0] != "Bearer")
    raise Exception('No bearer token in request')
  bearer_token = token[1]
  decoded_jwt = jwt.decode(bearer_token, "secret",
                           algorithms=["HS256"])
  tenant_context = {
                     "TenantId": decoded_jwt['tenantId'],
                     "Tier": decoded_jwt['tenantTier']
                   }
  return tenant_context
```

La clave, sin embargo, es que esta biblioteca ahora significa que cualquier servicio que necesite extraer el contexto de tenant puede hacer una única llamada a esta biblioteca y reducir la cantidad de código que aterriza en tus servicios. También te permite alterar las políticas JWT sin que estos cambios se propaguen por toda tu solución.

Este mismo principio también puede aplicarse al código de logging, métricas, acceso a datos y aislamiento de tenant que cubrimos anteriormente. Cada una de estas áreas puede abordarse mediante la introducción de bibliotecas que estandaricen el manejo de estos conceptos multi-tenant.

El acceso a datos es un área que puede ser un poco menos genérica y que puede requerir helpers locales a un servicio específico.

En este escenario, podrías apoyarte en el patrón tradicional de data access library (DAL) o repositorio para crear un elemento orientado que abstrae los detalles de interacción con el almacenamiento de nuestro servicio. Aquí, este DAL podría encapsular todos los requisitos multi-tenant, incluida la aplicación del aislamiento dentro de esa capa (completamente fuera de la vista de los desarrolladores del servicio).

Herramientas y estrategias de intercepción

La idea básica es que queremos analizar cómo podríamos aprovechar las capacidades integradas de un determinado elemento tecnológico para soportar estas necesidades multi-tenant transversales, permitiéndote introducir y configurar operaciones y políticas multi-tenant con mínima cooperación de tus desarrolladores de servicios.

Si bien sería contraproducente revisar todo el espectro de posibilidades, quiero destacar algunas estrategias de ejemplo para darte una mejor comprensión de los diferentes tipos de mecanismos que podrían adaptarse a esta mentalidad.

Aspectos

Los aspectos generalmente se introducen como un elemento del lenguaje o del framework. Permiten tejer mecanismos transversales en tu código que te habilitan para inyectar lógica de pre- y post-procesamiento en la huella de recursos de tus servicios.

Esto te permite introducir políticas y estrategias globales en tus servicios que pueden alinearse bien con algunos de los mecanismos multi-tenant que forman parte de tu entorno.

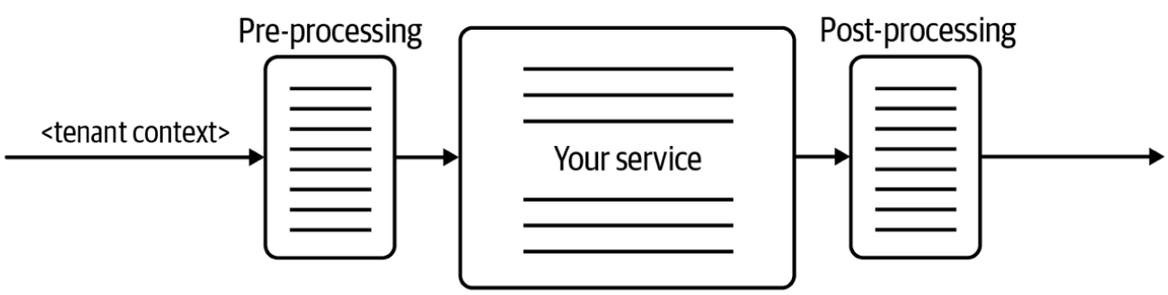

Figure 7-11. Using aspects to apply tenant context

Con la programación orientada a aspectos, puedo agregar lógica de manejo adicional a mi servicio que se ejecutará cuando las solicitudes de tenant entren y salgan.

Puedes imaginar cómo esto podría ser una excelente solución para manejar, procesar y aplicar operaciones contextuales de tenant. Por ejemplo, podrías usar un aspecto para interceptar cada solicitud que llega a tu servicio, añadiendo pre-procesamiento que extraiga el JWT del encabezado HTTP, lo decodifique e inicialice el contexto de tenant para el resto de tu solicitud. También

podrías considerar implementar elementos de tu modelo de aislamiento de tenant, adquiriendo e inyectando las credenciales con scope de tenant necesarias para hacer cumplir tus políticas de aislamiento.

Sidecars

Un sidecar se ejecuta en un contenedor independiente dentro de tu entorno de Kubernetes y tiene la capacidad de situarse entre tu servicio y otros recursos y servicios.

Lo bueno de estos sidecars es que están completamente fuera de la vista de tu servicio. Esto te permite aplicar cualquier política multi-tenant global de una manera que puede no requerir la cooperación de tu servicio.

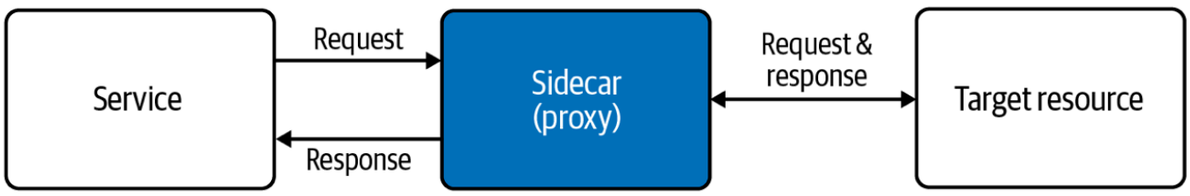

Figure 7-12. Using sidecars for horizontal concepts

Esto permite al sidecar interceptar y aplicar el contexto de tenant fuera de la vista de mi servicio, extrayendo el contexto y aplicando cualquier política asociada con mi servicio y el recurso que está consumiendo.

Poder desplegar y configurar este sidecar de forma independiente me permite crear una historia de cumplimiento multi-tenant mucho más robusta, permitiéndome tener mayor control sobre las interacciones de mi servicio con otros recursos.

Middleware

Algunos frameworks de desarrollo soportan la noción de middleware. La idea es que puedes introducir código que se sitúe entre tu solicitud entrante y tu operación objetivo. Esto te permite interceptar y aplicar cualquier política global que se aplicaría en todo tu servicio.

AWS Lambda Layers/Extensions

Si estás construyendo un entorno SaaS serverless en AWS, tienes la oportunidad de usar Lambda Layers o Lambda Extensions para mover tus bibliotecas compartidas a un mecanismo independiente.

Con Lambda Layers, puedes esencialmente mover todos tus helpers a una biblioteca compartida que luego se despliega de forma independiente. Cada una de las funciones Lambda que forman parte de tu servicio puede referenciar el código en esta biblioteca compartida, permitiéndoles acceder a las diferentes funciones helper que tienes sin que ese código forme parte de cada servicio.

Lambda Extensions, por otro lado, se adaptan más al patrón de aspectos que discutimos anteriormente, permitiéndote asociar código personalizado con el ciclo de vida de una función Lambda.