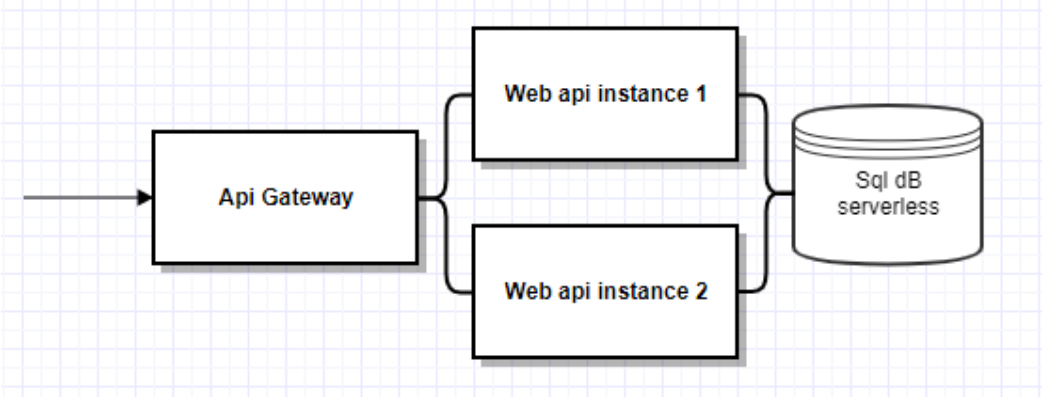
Facturación de MercadoLibre

Para el examen técnico de Meli se desarrolló una Api Rest siguiendo una arquitectura orientada a Domain Driven Design. De esta manera se separaron las capas en Dominio, Infraestructura y Aplicación (la web api).

Tecnología

El desarrollo está sobre Dot Net Core 3.0, el último de framework open source .Net. La tecnología se eligió por la experiencia sobre la misma y la agilidad que me permitió para alcanzar una primera versión productiva en un corto tiempo.

Infraestructura

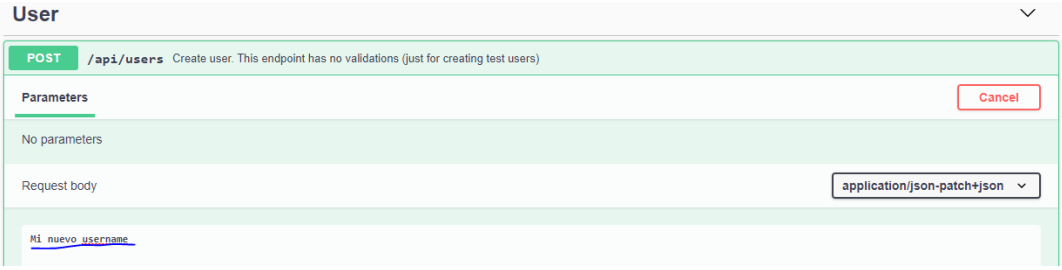


El proyecto se encuentra hosteado en Azure, utilizando 1 db SQL Server serverless, 2 app services que corresponden a 2 instancias de la Api Rest, y 1 Api gateway (**ocelot**) muy simple que funciona como load balancer.
Esta infraestructura con un gateway y 2 instancias de la api rest, apunta a satisfacer el requerimiento de miles de requests por minuto que se marca en la consigna. **Importante:** hay múltiples instancias de la web api, pero sigue habiendo una única db a la que acceden todas las apis. Este es un potencial cuello de botella.

Testing

Para testear la web api se recomienda hacerlo via postman a través de la url de la api-gateway(https://meli-tests-gateway.azurewebsites.net/), de manera que se aproveche el balanceo entre las 2 instancias hosteadas en Azure. Dejé el archivo "**Meli Tests.postman_collection.json**" en la raíz del repositorio.
Por otro lado, acceder directamente a las instancias permite alcanzar la url de Swagger, que contiene documentación de los endpoints y algunas facilidades para testear los llamados a la Api desde el navegador.
https://meli-api-1.azurewebsites.net/swagger/index.html

https://meli-api-2.azurewebsites.net/swagger/index.html
Nota: los eventos y pagos se generan con la fecha en la que son enviados, no puede indicarse una fecha ficticia por lo que no es posible testar datos anteriores. Para revisar un usuario con facturas más antiguas, probar con el userId 4 que contiene datos agregados manualmente a db.
Nota2: Agregué un endpoint para creación de usuarios que puede ser útil para realizar pruebas con un usuario "en blanco".



Moneda (ARS / US)

El documento menciona que el sistema maneja pesos y dolares, pero para este proyecto solo se utilizarán pesos. Cuando armé el modelado de datos y comencé a implementarlo, ya tenía bastante camino avanzado así que habilité el funcionamiento con las 2 monedas. En caso de querer restringirlo a pesos podría hacerse con mínimo esfuerzo.

Versionado

La implementación de esta api no incluye versionado.

Motor de base de datos

Tuve algunas dudas respecto a implementarlo mediante una db SQL o no SQL. Me incliné por una db sql principalmente por el carácter transaccional del negocio, y la necesidad de relacionar entidades diferentes, como cargos y pagos, de forma ágil y eficiente.

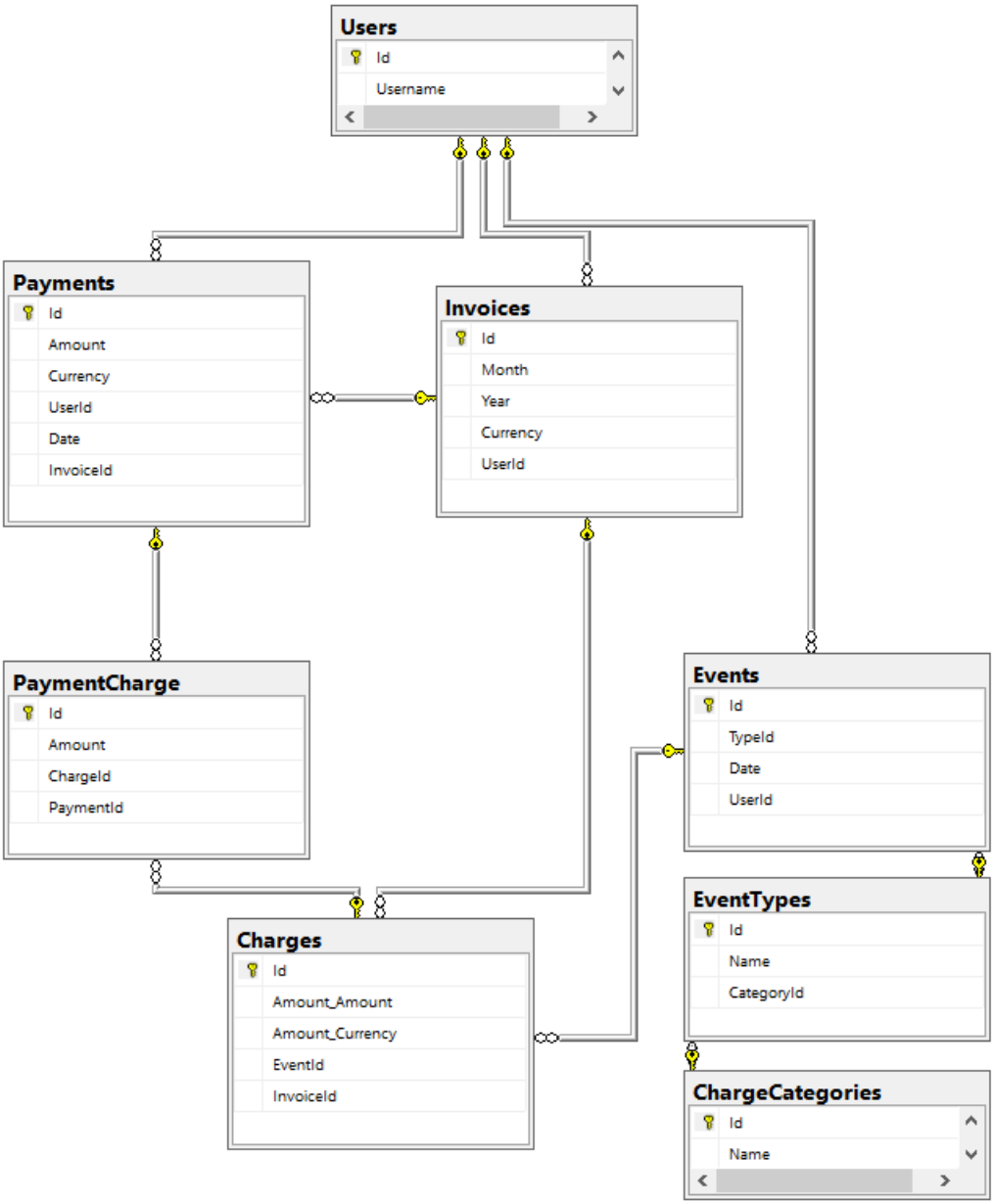
Deployment

Si se desea ejecutar el proyecto localmente, se debe tener instalado el SDK de dot net core 3.0. Descargar el repositorio, modificar el web config si se desea apuntar a una db sql local (utilizar la remota dará error por el firewall del servidor, si me pasan una ip de orígen puedo agregarla a las reglas de Azure para habilitarla), y parado en el proyecto de **NetCore3_api.WebApi** ejecutar:

1. dotnet ef database update --project NetCore3_api.Infrastructure --startup-project NetCore3_api.WebApi
   (en caso de estar apuntando a una nueva base de datos)
2. Dotnet run

Api + Modelo de datos

La Api está construida en torno a 5 recursos: Charges, Payments, Invoices, Debts y Events (podría considerarse User como un recurso también importante, pero no está desarrollado en profundidad para esta iteración).
El detalle de los endpoints y sus parámetros puede visualizarse en la página de swagger: https://meli-api-1.azurewebsites.net/swagger/index.html



Eventos

Los eventos inician el flujo de cargos/pagos. Al realizar un POST al endpoint *api/users/{userId}/events* se generará un nuevo **Cargo** al que se le asociará también una entidad de tipo **Evento**. La entidad evento almacena referencias al usuario que lo generó, así como la fecha y hora, y el tipo.

Al momento de realizar el diseño de datos, tenía algunas dudas sobre si era necesario generar una entidad de este tipo, ya que la relación con Cargo siempre es 1 a 1, y uno no existe sin el otro (podría considerarse al evento como propiedades del cargo). Sin embargo, por el énfasis que se le pone en la consigna a este tipo de objeto, y para abrir la posibilidad a eventos que no generen cargos, se creó la clase/entidad Evento.

Por último, consideré que no es importante consultar eventos en esta aplicación, por lo que solo está permitida la creación de Eventos (POST).

Cargos
Esta entidad se genera a raíz de un POST al endpoint de eventos, es decir, no se puede crear un cargo sin un evento. Al generarse el cargo, se valida que el monto, el tipo de moneda, usuario y tipo de evento sean válidos. Si el cargo es válido, se incluirá en una **factura** y se persistirá en db.
La consulta de cargos via GET a */api/users/{userId}/charges* retorna todos los cargos del usuario, incluyendo el detalle de los pagos asociados y el monto asignado por el pago a este cargo. Se creó una entidad intermedia que relaciona **Cargos** con **Pagos** llamada **PaymentCharge** que registra cuál es el monto del pago vinculado a saldar este cargo. Esto es porque un pago puede usarse para saldar múltiples cargos, y también un cargo puede tener asociados múltiples pagos parciales hasta saldar su total.

Pagos
¡Los pagos no pueden exceder la deuda del usuario! Esa es la ley primera, y se valida eso al generar cada pago. Es importante marcar que la deuda es respecto a una moneda, un usuario puede tener deuda en ARS pero no en U$, por lo que se aceptarán pagos por montos correctos en pesos pero ninguno en dólares.
Al generar el pago sucede lo siguiente:

1. Se valida lo ya mencionado, + que el pago contenga un monto mayor a 0 y de una moneda aceptada en nuestro sistema.
2. Se buscan los cargos con saldo impago, ordenados de más antíguo a más reciente(según el ID) y se van generando **PaymentCharges** hasta que se agota el monto que tenía el pago.
3. Se asocia el pago a una **Factura**.

Facturas (Invoices)
Las facturas se generan cuando se crea un cargo y no existía una factura para este usuario, en este período y con la moneda del cargo en cuestión. Una vez que ya existe una factura para un usuario - período - moneda, los cargos y pagos nuevos generados en dicho período, se asignarán al mismo.
La consulta de facturas (GET) es bastante completa, ya que detalla los cargos con sus correspondientes pagos asociados (y el detalle de cuando saldó dicho pago) además del total de los pagos realizados en ese período. Es decir, la factura se puede reducir a un listado de los cargos del período (con detalles de sus pagos asociados) + un listado de los pagos del período.
Un detalle importante de esta implementación es que si un pago de noviembre salda un cargo del mes de octubre, dicho pago no figurará en el listado de

pagos de la factura de octubre PERO sí aparecerá en el detalle de pagos asociado a cada cargo (como una entidad anidada).

Status Usuario / Deuda (UserDebt)
La deuda del usuario se mostrará separada por moneda, es decir que, al consultar un usuario que acredita eventos en pesos y en dolares, mostrará un listado de 2 deudas (con totales por moneda) al solicitar un GET a */api/users/{userId}/debts*
En esta implementación, al consultar la deuda se recorren todos los cargos, sus pagos asociados, y se sumariza el total de saldo adeudado por moneda. Si la consulta de deuda fuese algo habitual, esto puede volverse pesado para procesar, y podría evaluarse almacenar el total de deuda desnormalizada, quizás como una columna de usuario o una nueva entidad.

Solución Bonus 1

Para esta iteración no desarrollé el feature de aceptar pagos por montos mayores a la deuda del usuario. Pero, para implementarlo, tendría que:

1. Modificar la lógica de UserDebtService.IsValidPayment(payment)
   Este método concentra esta regla de validación. Luego tendría que modificar los Tests que evalúan esta regla.
2. Esto permitiría asociar pagos a un usuario que no están asociados a ningún cargo. Por lo que habría que agregar lógica a ChargeService.CreateChargeWithEvent() de manera que al generar un cargo, se revise si no hay pagos “libres” para asociarlos inmediatamente a estos cargos. Similar a como sucede ahora cuando se crea un pago y se buscan cargos disponibles para asociar.

Gracias por leer =)