# Hola,

# ¡Llegó el momento de ver la magia como Dev! 🤩

### 📣 Introducción

A continuación, se presenta un ejercicio simple que debe ser realizado en lenguaje Rust (v1.71 o superior) con el objetivo de evaluar las siguientes cualidades del participante

- Conocimiento general del lenguaje.
- Prolijidad.
- Buenas prácticas de programación.
- Capacidad de investigación.
- Pensamiento orientado a concurrencia y sistemas informáticos.

El challenge consiste en realizar un “mini procesador de pagos” con la capacidad de llevar el saldo de los clientes en memoria y persistirlos (cuando se solicite) en un archivo.

Se deberá implementar un micro servicio que exponga una API REST al usuario*, a través de la cual pueda llevar un registro del saldo de sus clientes.

** Al referirnos a un procesador de pagos, el usuario del mismo es un emisor de tarjetas y/o servicios de pago, no así el cliente final.*

**Definición de requerimientos** 👉

El servicio recibirá instrucciones a través de su API REST para acreditar o debitar saldo a los clientes. Cada cliente deberá ser creado inicialmente mediante el servicio "*new_client*" y luego podrá recibir tanto débitos (resta al saldo) como créditos (suma al saldo).

Se podrá consultar el saldo e información de los clientes mediante el servicio "*client_balance*".

El saldo y toda la base de clientes deberá poder persistirse en un archivo mediante el servicio "*store_balances*". El servicio deberá implementar los siguientes endpoints:

- **POST *"new_client"***

  El servicio debe recibir el siguiente body:

```
{
    "client_name": <STRING>,
    "birth_date": <NaiveDate>,
    "document_number": <STRING>,
    "country": <STRING>

}
```

  El número de documento no se puede repetir en la base de clientes.

  El servicio debe devolver como respuesta, el ID del cliente creado (se debe generar internamente y ser único).

  Realizar las validaciones que se crean necesarias.

- **POST *"new_credit_transaction"***

  El servicio debe recibir el siguiente body (JSON):

```
{
    "client_id": <INT>,
    "credit_amount": <DECIMAL>
}
```

  Debe poder encontrar al cliente (o devolver un error adecuado) mediante su

ID e incrementar su saldo en el monto especificado.

El servicio debe devolver como respuesta el nuevo saldo del cliente. Realizar las validaciones que se crean necesarias.

- **POST *"new_debit_transaction"***

El servicio debe recibir el siguiente body (JSON):

```
{
    "client_id": <INT>,
    "debit_amount": <DECIMAL>
}
```

Debe poder encontrar al cliente (o devolver un error adecuado) mediante su ID y decrementar su saldo en el monto especificado.

El servicio debe devolver como respuesta el nuevo saldo del cliente. Realizar las validaciones que se crean necesarias.

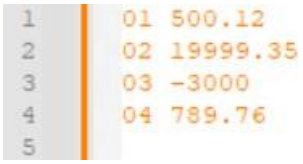
- **POST *"store_balances"***

Este servicio no recibe ningún input de información.

Debe persistir en archivo todos los IDs de clientes y sus balances. Este servicio debe limpiar los balances de los clientes (sólo su balance), de forma que todos los clientes queden con balance 0 en memoria, pero sus balances previos almacenados en el archivo.

El formato del nombre del archivo deberá ser "DDMMYYYY_FILE COUNTER.DAT", por ejemplo: "01122023_10.DAT" donde "01122023" corresponde a la fecha (01/12/2023) y el "10" identifica a que es el archivo n° 10 generado (desde el inicio del servicio).

Se presenta a continuación un ejemplo de formato del archivo:




ID_CLIENTE<espacio>BALANCE<Salto de línea>
ID_CLIENTE<espacio>BALANCE<Salto de línea>
ID_CLIENTE<espacio>BALANCE<Salto de línea>
ID_CLIENTE<espacio>BALANCE<Salto de línea>

- **GET *"client_balance"***

  El servicio debe recibir el ID de cliente mediante un parámetro de URL (user_id) y devolver un JSON conteniendo la información del cliente y su balance. El formato del mismo queda a criterio del lector.

## 💪 Recomendaciones

Se describen a continuación recomendaciones generales alineadas con la forma de trabajo interna de PrexCORE:

- Utilizar Actix Web para la realización del web server.
- Utilizar formato Decimal para los saldos de clientes.
- Utilizar asincronismo mediante runtime de Tokio.
- Utilizar Serde para serialización y deserialización de payloads.
- Evitar la repetición de código mediante el uso de funciones y/o macros.
- Buenas prácticas de diseño: KISS, DRY y SOLID.

## 💪 Entregables

La entrega debe realizarse mediante un repositorio en la nube (GitHub o Bitbucket) que contenga:

- Código fuente.
- Colección de Postman para la prueba de la API.
- Documentación que el desarrollador considere necesaria (si la hay)

### 💪 Criterio de revisión

Durante la revisión del challenge se pondrá énfasis en las siguientes características:

- Prolijidad.
- Buenas prácticas de programación.
- Documentación del código.
- Calidad del código.
- Capacidad de comprensión del participante.
- Proactividad del participante.
- Organización del código.
- Buen criterio y sentido común.

**Ahora sí: ¡llegó el momento de ver tu magia!** ✨

¡Buena suerte! 😊