

***Propuestas de mejora para el trabajo VivesBank.Net***



Alba García Orduña  
Natalia González Álvarez  
Mario de Domingo Álvarez  
Jaime León Mulero  
Yahya El Hadri El Bakkali  
Kelvin J. Sánchez Barahona  
Javier Ruiz Serrano  
Óscar Encabo Nieto

# ÍNDICE

1. Mejora de Caché
2. Mejoras en el manejo de errores para las conexiones WebSocket
3. Mejoras en el servicio de Backup
4. Mejoras en GraphQL
5. Mejoras en el Manejo de Excepciones
6. Mejoras en la Configuración del OnModelCreating
7. Métodos añadidos ausentes
8. Mejoras en consistencia de datos/escalabilidad
9. Mejoras en la estructura de métodos

## 1. Uso de cache en memoria

Con el fin de optimizar el rendimiento y mejorar la escalabilidad del servicio de usuarios, se ha implementado una solución de caché en memoria. Esta implementación tiene como objetivo reducir la cantidad de consultas a la base de datos, mejorar el tiempo de respuesta de las peticiones repetitivas y minimizar la carga sobre el sistema de almacenamiento. La caché se almacena de forma temporal en la memoria para consultas que no requieran acceso a datos actualizados de forma constante, garantizando que los usuarios reciban las respuestas de forma más eficiente.

### Cambios realizados:

#### 1. Implementación de caché en memoria:

Se ha añadido una capa de caché en memoria utilizando una solución que almacena los datos de los usuarios. Este almacenamiento está diseñado para ser rápido y accesible, permitiendo que las solicitudes subsiguientes con los mismos datos no necesiten hacer una nueva consulta a la base de datos, sino que se sirvan directamente desde la caché.

#### 2. Invalidación de caché:

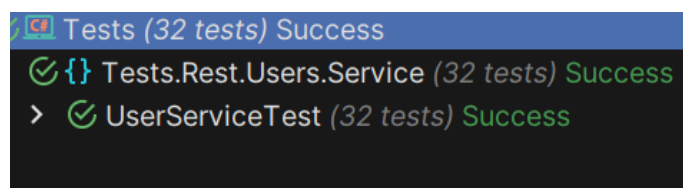
Se ha implementado un mecanismo que asegura que la caché se invalide correctamente cuando se realiza una actualización de los datos en la base de datos. De este modo, cuando un usuario es actualizado o cuando se hace un cambio importante, la caché se purga y se vuelve a generar la próxima vez que se necesite acceder a esos datos. Esto garantiza que los datos siempre sean consistentes y reflejen los cambios más recientes.

#### 3. Actualización de tests:

Se han modificado y añadido nuevos casos de prueba unitarios e integrados para comprobar el comportamiento correcto de la caché. Estos tests verifican tanto la inserción de los datos en la memoria como la validación de su correcta eliminación e invalidación cuando los datos cambian.

#### 4. Pruebas en Postman:

Se han realizado una serie de pruebas manuales en Postman para validar el funcionamiento de los endpoints. Durante estas pruebas, se verificó que los datos se devuelvan correctamente desde la caché en los casos en los que ya existía una entrada previa, y que los datos se actualicen correctamente cuando haya cambios en la base de datos.



## ✓ Beneficios:

### 1. Reducción del tiempo de respuesta:

Gracias a la caché en memoria, las consultas repetidas a los mismos datos no requieren acceso a la base de datos, lo que reduce considerablemente el tiempo de respuesta de los endpoints. Los usuarios que soliciten información que ya esté almacenada en la caché recibirán las respuestas mucho más rápido.

### 2. Menos carga en la base de datos:

Con la implementación de la caché, el número de consultas a la base de datos se reduce. Esto no solo mejora el rendimiento de la aplicación, sino que también reduce el uso de recursos en el servidor de base de datos, permitiendo una mayor disponibilidad para otras operaciones.

### 3. Mejora en la escalabilidad:

Al disminuir la carga de trabajo en la base de datos y mejorar el rendimiento general, el sistema es más escalable y capaz de manejar un mayor volumen de solicitudes sin que el rendimiento se vea afectado. La capacidad de almacenar en memoria y gestionar datos de manera eficiente permite que el servicio sea más robusto frente a un mayor número de usuarios concurrentes.

## Ejemplos de código actualizado

(Antes)

```
public async Task<UserResponse> UpdateUserAsync(string id, UserUpdateRequest user)
{
    if (user.Dni != null && !UserValidator.ValidateDni(user.Dni))
    {
        throw new InvalidDniException(user.Dni);
    }

    User? userToUpdate = await GetByIdAsync(id) ?? throw new UserNotFoundException(id);

    if (user.Dni != null)
    {
        User? userWithTheSameUsername = await GetByUsernameAsync(userToUpdate.Dni);
        if (userWithTheSameUsername != null && userWithTheSameUsername.Id != id)
        {
            throw new UserAlreadyExistsException(user.Dni);
        }
    }

    User updatedUser = user.UpdateUserFromInput(userToUpdate);
    await _userRepository.UpdateAsync(updatedUser);
    // Eliminar la entrada antigua de la caché
    await _cache.KeyDeleteAsync(id);
    await _cache.KeyDeleteAsync("users:" + userToUpdate.Dni.Trim().ToUpper());
    // Agregar la nueva entrada a la caché
    await _cache.SetStringAsync(id, JsonConvert.SerializeObject(updatedUser), expiry: TimeSpan.FromMinutes(10));
    return updatedUser.ToUserResponse();
}
```

(Después)

```
public async Task<UserResponse> UpdateUserAsync(string id, UserUpdateRequest user) 5+8 usages 1 Diego NL +4
{
    if (user.Dni != null && !UserValidator.ValidateDni(user.Dni))
    {
        throw new InvalidDniException(user.Dni);
    }

    // Obtener el usuario a actualizar
    User? userToUpdate = await GetByIdAsync(id) ?? throw new UserNotFoundException(id);

    if (user.Dni != null)
    {
        // Verificar si hay otro usuario con el mismo DNI
        User? userWithTheSameUsername = await GetByUsernameAsync(userToUpdate.Dni);
        if (userWithTheSameUsername != null && userWithTheSameUsername.Id != id)
        {
            throw new UserAlreadyExistsException(user.Dni);
        }
    }

    // Actualizar los detalles del usuario
    User updatedUser = user.UpdateUserFromInput(userToUpdate);
    await _userRepository.UpdateAsync(updatedUser);

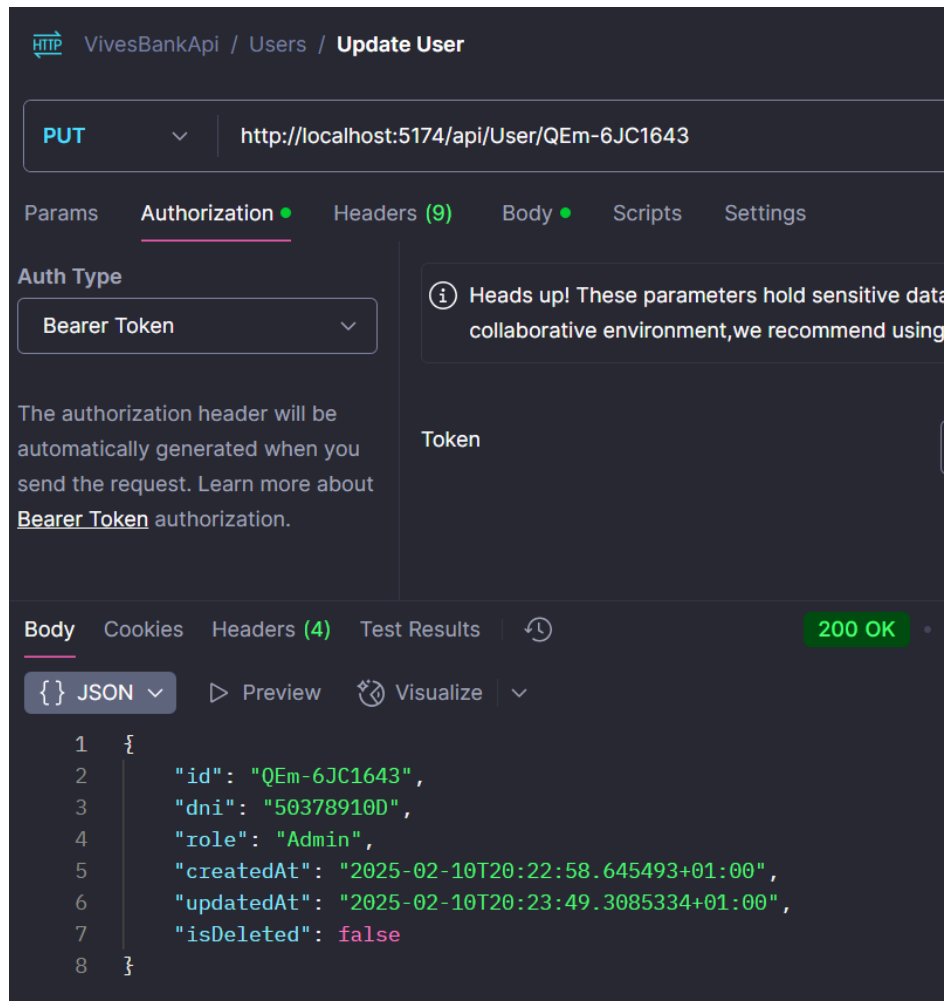
    // Eliminar la entrada antigua de la caché (tanto en Redis como en la memoria)
    await _cache.KeyDeleteAsync(id);
    await _cache.KeyDeleteAsync("users:" + userToUpdate.Dni.Trim().ToUpper());

    _memoryCache.Remove(id);
    _memoryCache.Remove(key: "users:" + userToUpdate.Dni.Trim().ToUpper());

    await _cache.SetStringAsync(id, JsonConvert.SerializeObject(updatedUser), expiry: TimeSpan.FromMinutes(10));
    _memoryCache.Set(id, updatedUser, absoluteExpirationRelativeToNow: TimeSpan.FromMinutes(10));
    _memoryCache.Set("users:" + updatedUser.Dni.Trim().ToUpper(), updatedUser, absoluteExpirationRelativeToNow: TimeSpan.FromMinutes(10));

    return updatedUser.ToUserResponse();
}
```

## Ejemplo de prueba en Postman



## 2. Mejoras en el manejo de errores al enviar notificaciones por WebSocket

La propuesta de cambio en el servicio de WebSocket optimiza la gestión de conexiones al incluir un manejo adecuado de excepciones, lo que mejora la estabilidad y resiliencia del servicio. Se han añadido verificaciones para asegurar que las notificaciones enviadas a los clientes sean efectivamente transmitidas y, en caso contrario, se lance una excepción.

### 1. Manejo de excepciones en NotifyUserAsync:

Anteriormente, el método NotifyUserAsync intentaba enviar notificaciones sin verificar si la operación de envío se completaba correctamente. Ahora, se ha implementado un bloque try-catch que captura posibles errores durante la ejecución de SendAsync. En caso de fallo, se lanza explícitamente una WebSocketException, asegurando que los errores sean manejados adecuadamente y registrados en los logs.

### 2. Manejo de excepciones en NotifyAllAsync:

El método NotifyAllAsync también ha sido mejorado para manejar correctamente las excepciones durante el envío de notificaciones a múltiples clientes. Ahora, si alguna de las llamadas a SendAsync falla, la excepción se propagará, lo que permite una mejor identificación de problemas en la comunicación con los clientes conectados.

### 3. Optimización de la estabilidad y monitoreo:

La implementación de estas mejoras garantiza una mayor robustez en la comunicación WebSocket, evitando silencios en la notificación de errores y facilitando la depuración mediante logs detallados. De esta manera, se mejora la estabilidad del servicio y se reduce el riesgo de inconsistencias en la entrega de mensajes.

El código antes:

```
public async Task NotifyAllAsync<T>(Notification<T> notification)
{
    // Serialize notification to JSON while ignoring null values
    var jsonSettings = new JsonSerializerSettings
    {
        NullValueHandling = NullValueHandling.Ignore,
        Formatting = Formatting.Indented
    };
    var jsonString = JsonConvert.SerializeObject(notification, jsonSettings);
    _logger.LogInformation($"Notifying all clients: {json}");

    var buffer :byte[] = Encoding.UTF8.GetBytes(json);

    var tasks :Task[] = _sockets.Select(socket =>
        socket.SendAsync( buffer, new ArraySegment<byte>(buffer), WebSocketMessageType.Text, endOfMessage: true, CancellationToken.None))
        .ToArray();
    await Task.WhenAll(tasks);
}
```

El código después:

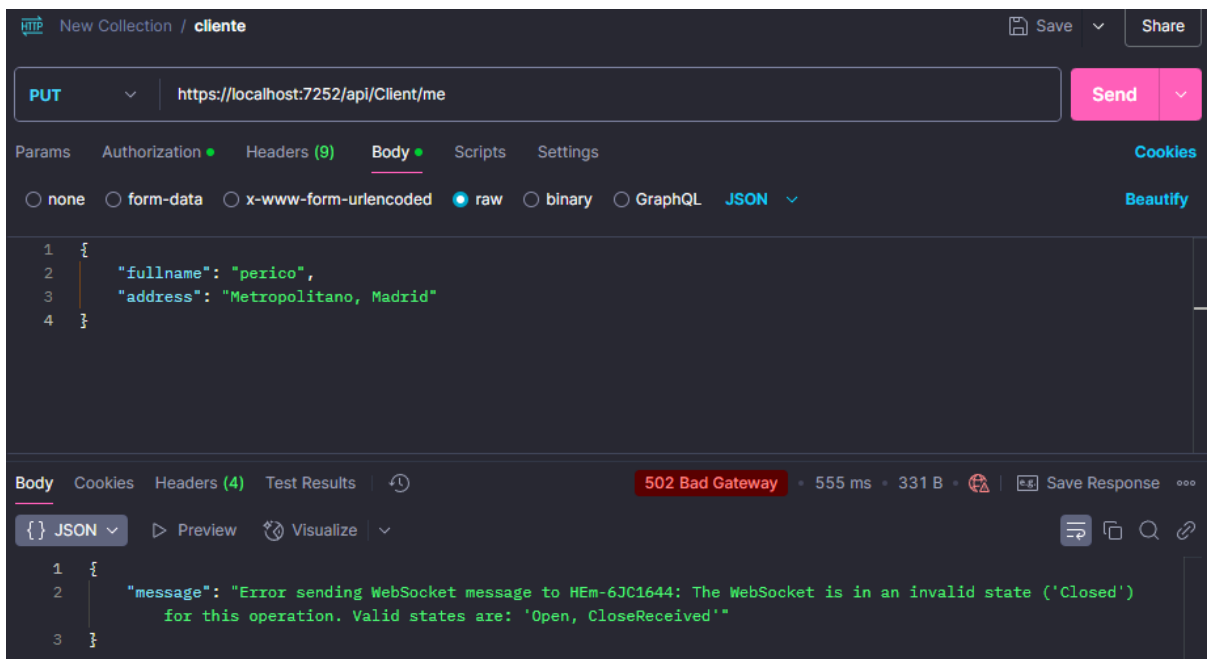
```
public async Task NotifyAllAsync<T>(Notification<T> notification)
{
    var jsonSettings = new JsonSerializerSettings
    {
        NullValueHandling = NullValueHandling.Ignore,
        Formatting = Formatting.Indented
    };
    var jsonString = JsonConvert.SerializeObject(notification, jsonSettings);
    _logger.LogInformation($"Notifying all clients: {json}");

    var buffer :byte[] = Encoding.UTF8.GetBytes(json);

    var tasks :Task[] = _sockets.Select(async socket =>
    {
        try
        {
            await socket.SendAsync( buffer, new ArraySegment<byte>(buffer), WebSocketMessageType.Text, endOfMessage: true, CancellationToken.None);
        }
        catch (WebSocketException ex)
        {
            _logger.LogError($"Error sending WebSocket message: {ex.Message}");
            throw new WebSocketException($"Failed to send notification: {ex.Message}", ex);
        }
    }).ToArray();

    await Task.WhenAll(tasks);
}
```

El cambio se ha probado en un entorno controlado utilizando Postman para simular conexiones WebSocket y verificar que las excepciones se manejan correctamente. Las pruebas confirmaron que los WebSockets se cierran de manera adecuada y que el sistema responde correctamente ante errores, asegurando su funcionamiento al 100%.





### 3. Mejoras en el servicio de Backup

#### Corrección de exportación/importación y uso de repositorios para acceso a datos

##### 1. Exportación a ZIP corregida:

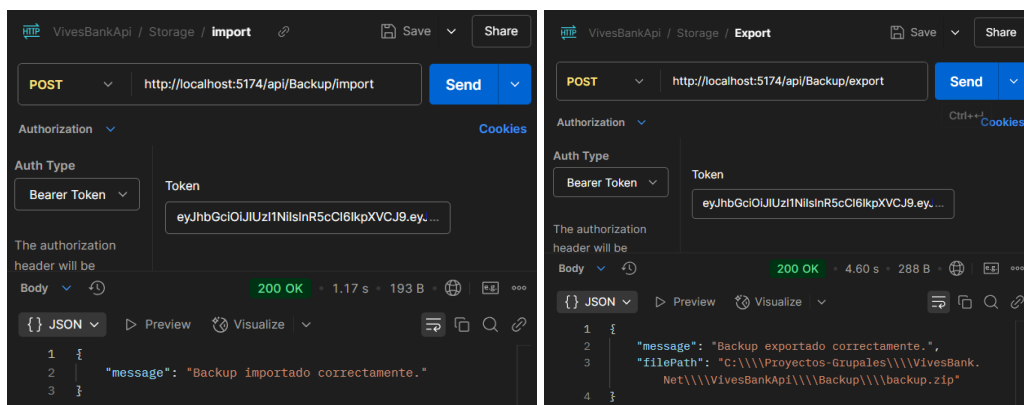
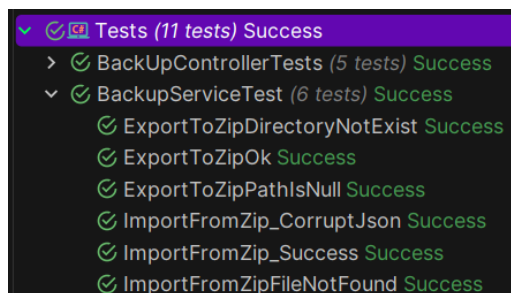
- **Problema:** El método ExportToZip generaba archivos JSON vacíos, por lo que no se exportaban los datos correctamente.
- **Solución:** Ahora se exportan los datos a archivos JSON antes de comprimirlos en un ZIP. Los datos se obtienen directamente de los repositorios y se serializan correctamente.

##### 2. Importación desde ZIP corregida:

- **Problema:** El método ImportFromZip solo verifica si existían archivos JSON, pero no comprobaba si tenían contenido válido.
- **Solución:** Ahora se valida el contenido de los archivos JSON y se verifica si los datos ya existen en la base de datos antes de importarlos, evitando duplicados.

##### 3. Uso de repositorios:

- **Problema:** El servicio dependía de la capa de servicios (IClientService, IUserService, etc.), lo que impedía la importación necesaria debido a cómo está hecho.
  - **Solución:** Se ha modificado el servicio para usar directamente los repositorios (IClientRepository, IUserRepository, etc.), lo que simplifica el acceso a la base de datos y mejora la eficiencia.
- **Mejora:** Ahora se elimina el directorio temporal después de su uso, evitando la acumulación de archivos innecesarios.
  - **Test:** Actualizados los test al nuevo servicio.



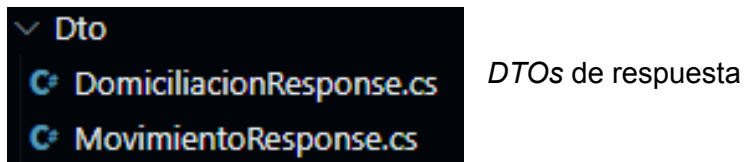
## 4. Mejoras en GraphQL

En esta propuesta de mejora se logran dos objetivos. En primer lugar, reducir la memoria utilizada evitando la creación de nuevos objetos, y en segundo lugar, evitar devolver campos nulos no necesarios.

Para lograr el objetivo, se han definido dos tipos de *DTOs* de respuesta quitando así del modelo principal el campo ID. Además, para poder trabajar entre los *DTOs* y los *Models* se ha definido un *Mapper* de forma estática para poder llamarlo desde cualquier lugar del proyecto.

### Cambios realizados:

Se han implementado dos *DTOs* de respuesta para quitar el campo ID del *Model* principal de estos:



```
public class MovimientoResponse
{
    public string Guid { get; set; }

    public string ClienteGuid { get; set; }
```

Se ha ignorado el primer atributo: ID

```
public class DomiciliacionResponse
{
    public string Guid { get; set; }

    public string ClienteGuid { get; set; }
```

Se ha ignorado el primer atributo: ID

Por otro lado, para el manejo de ambos *DTOs*, se ha implementado un *Mapper* con dos funciones. Ambas realizan la misma acción pero cada una será para un *Model - DTO*.

```
public static class MovimientosMapper
{
    public static DomiciliacionResponse ToDomiciliacionResponseFromModel(this Domiciliacion domiciliacion)
    {
        return new DomiciliacionResponse()
        {
            Guid = domiciliacion.Guid,
            ClienteGuid = domiciliacion.ClienteGuid,
            IbanOrigen = domiciliacion.IbanOrigen,
            IbanDestino = domiciliacion.IbanDestino,
            Cantidad = domiciliacion.Cantidad,
            NombreAcreedor = domiciliacion.NombreAcreedor,
            FechaInicio = domiciliacion.FechaInicio.ToString(),
            Periodicidad = domiciliacion.Periodicidad.ToString(),
            Activa = domiciliacion.Activa,
            UltimaEjecucion = domiciliacion.UltimaEjecucion.ToString()
        };
    }

    public static MovimientoResponse ToMovimientoResponseFromModel(this Movimiento movimientoResponse)
    {
        return new MovimientoResponse()
        {
            Guid = movimientoResponse.Guid,
            ClienteGuid = movimientoResponse.ClienteGuid,
            Domiciliacion = movimientoResponse.Domiciliacion?.ToDomiciliacionResponseFromModel(),
            IngresoDeNomina = movimientoResponse.IngresoDeNomina,
            PagoConTarjeta = movimientoResponse.PagoConTarjeta,
            Transferencia = movimientoResponse.Transferencia,
            CreatedAt = movimientoResponse.CreatedAt.ToString(),
            UpdatedAt = movimientoResponse.UpdatedAt.ToString(),
            IsDeleted = movimientoResponse.IsDeleted,
        };
    }
}
```

Podemos observar que ambas funciones están declaradas *static* para poder usarlas en cualquier lugar de nuestro proyecto.

Además, en *ToMovimientoResponseFromModel* se hace uso de la función *ToDomiciliacionResponseFromModel* para evitar devolver el campo ID de domiciliación dentro de esta.

Finalmente, se ha modificado el uso de *GraphQL* en el archivo *MovimientoQuery*.

Mediante el uso de los *Models* de respuesta y el *Mapper* se evita generar nuevos objetos reduciendo así la memoria utilizada y simplificando la función.

A continuación, se muestra el cambio en una de las funciones pertenecientes a *MovimientoQuery*.

Antes del cambio:

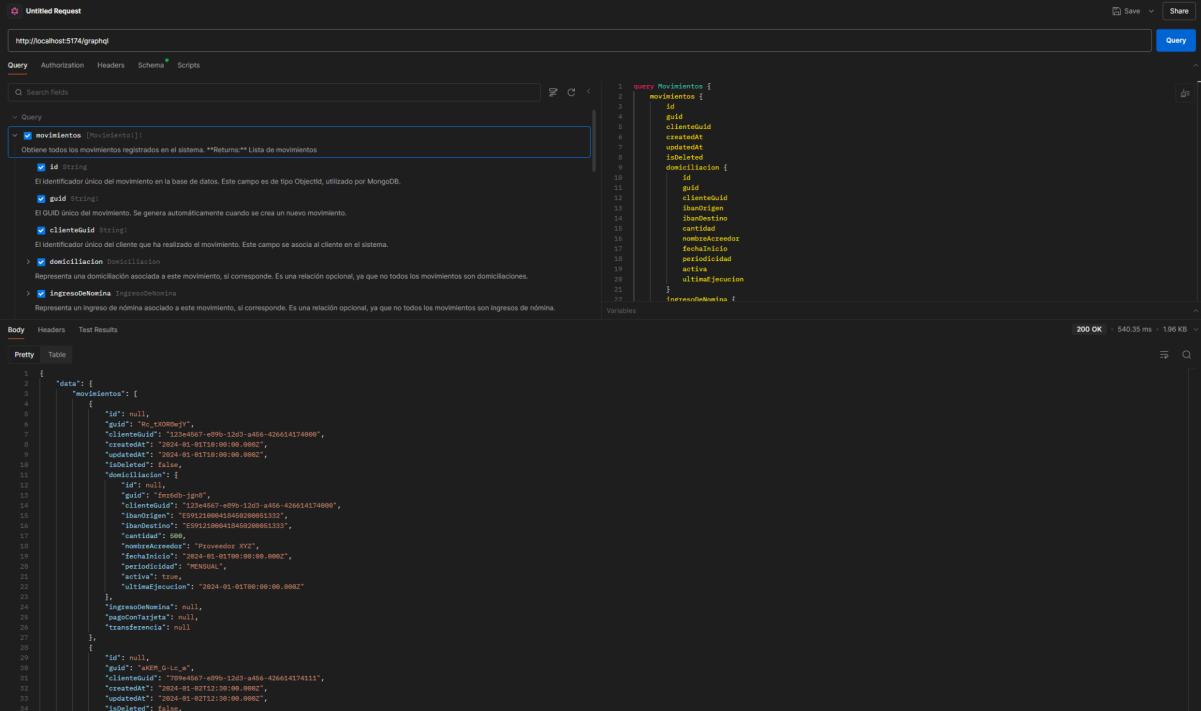
```
public async Task<IQueryable<Movimiento>> GetMovimientos()
{
    var movimientosList = await movimientoService.FindAllMovimientosAsync();
    return movimientosList.Select(movimiento => new Movimiento
    {
        Guid = movimiento.Guid,
        ClienteGuid = movimiento.ClienteGuid,
        Domiciliacion = movimiento.Domiciliacion,
        IngresoDeNomina = movimiento.IngresoDeNomina,
        PagoConTarjeta = movimiento.PagoConTarjeta,
        Transferencia = movimiento.Transferencia,
        CreatedAt = movimiento.CreatedAt,
        UpdatedAt = movimiento.UpdatedAt
    }).AsQueryable();
}
```

Después del cambio:

```
public async Task<IQueryable<MovimientoResponse>> GetMovimientos()
{
    var movimientosList = await movimientoService.FindAllMovimientosAsync();
    return movimientosList.Select(
        movimiento => movimiento.ToMovimientoResponseFromModel()
    ).AsQueryable();
}
```

## ✓ Resultados y comprobaciones en Postman:

Antes:



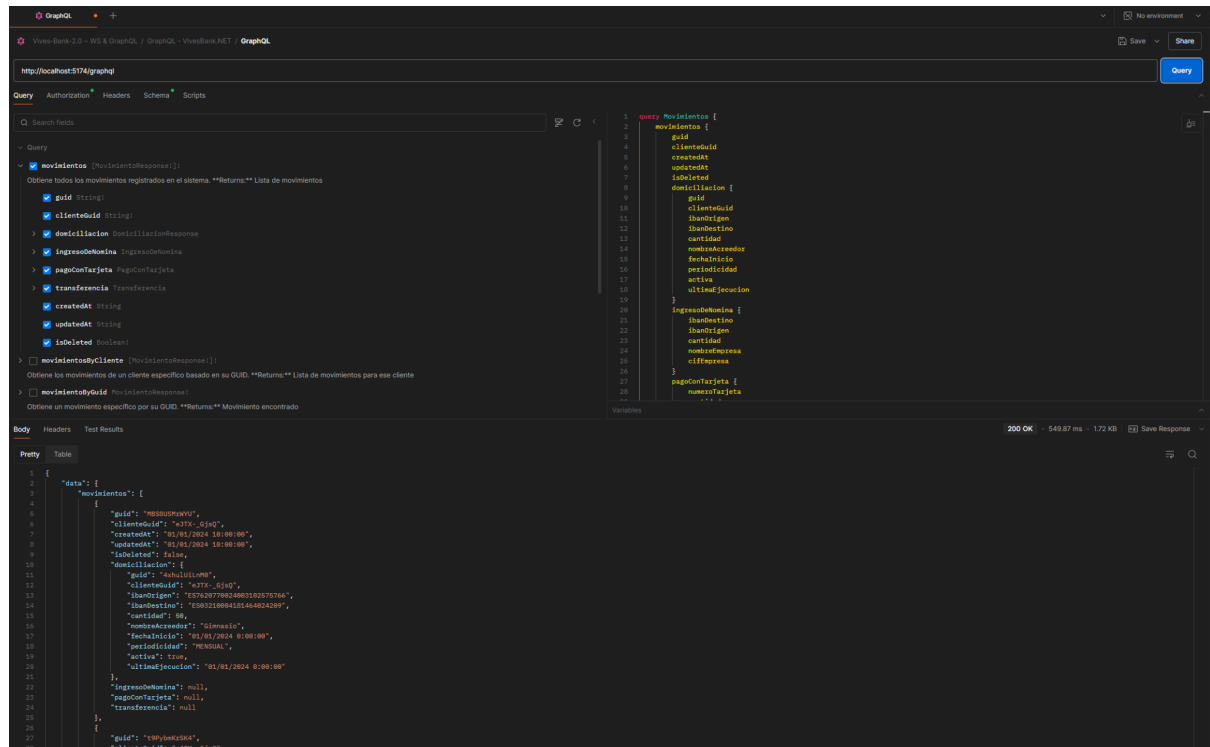
Postman interface showing a REST client request to `http://localhost:5174/graphql`. The query is:

```
query Movimientos {
  movimientos {
    id
    clienteGuid
    domiciliacion {
      id
      guid
      clienteGuid
      ibanOrigen
      ibanDestino
      cantidad
      numeroAcreditor
      fechaInicio
      periodicidad
      activa
      ultimaEjecucion
    }
    ingresoDeNomina
    pagoConTarjeta
    transferencia
  }
}
```

The response is a JSON object with a `data` field containing an array of `movimientos`.

```
{
  "data": {
    "movimientos": [
      {
        "id": null,
        "guid": "76c4308b-17c",
        "clienteGuid": "123e4567-e89b-12d3-a456-426614174000",
        "createdAt": "2024-01-01T00:00:00.000Z",
        "updatedAt": "2024-01-01T00:00:00.000Z",
        "deleted": false,
        "domiciliacion": {
          "id": null,
          "guid": "76c4308b-17c",
          "clienteGuid": "123e4567-e89b-12d3-a456-426614174000",
          "ibanOrigen": "ES9121000413400200001333",
          "ibanDestino": "ES9121000413400200001333",
          "cantidad": 0.00,
          "numeroAcreditor": "Proveedor XYZ",
          "fechaInicio": "2024-01-01T00:00:00.000Z",
          "periodicidad": "Mensual",
          "activa": true,
          "ultimaEjecucion": "2024-01-01T00:00:00.000Z"
        },
        "ingresoDeNomina": null,
        "pagoConTarjeta": null,
        "transferencia": null
      },
      {
        "id": null,
        "guid": "76c4308b-17c",
        "clienteGuid": "123e4567-e89b-12d3-a456-426614174000",
        "createdAt": "2024-01-01T00:00:00.000Z",
        "updatedAt": "2024-01-01T00:00:00.000Z",
        "deleted": false,
        "domiciliacion": {
          "id": null,
          "guid": "76c4308b-17c",
          "clienteGuid": "123e4567-e89b-12d3-a456-426614174000",
          "ibanOrigen": "ES9121000413400200001333",
          "ibanDestino": "ES9121000413400200001333",
          "cantidad": 0.00,
          "numeroAcreditor": "Proveedor XYZ",
          "fechaInicio": "2024-01-01T00:00:00.000Z",
          "periodicidad": "Mensual",
          "activa": true,
          "ultimaEjecucion": "2024-01-01T00:00:00.000Z"
        },
        "ingresoDeNomina": null,
        "pagoConTarjeta": null,
        "transferencia": null
      }
    ]
  }
}
```

Después:



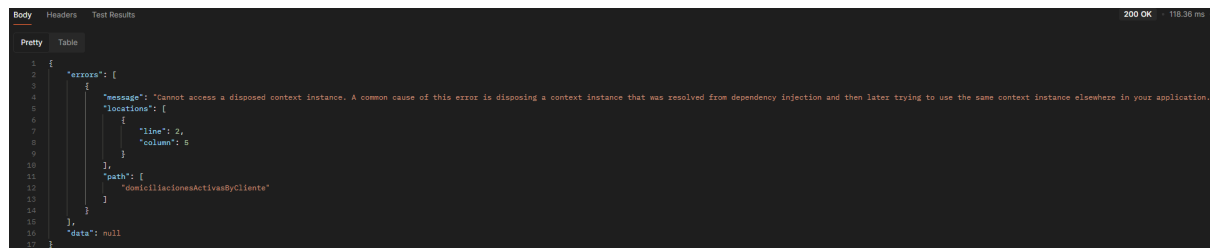
**⚠ Importante:**

Para realizar pruebas una vez subidos los cambios a la rama dev se deberá modificar el script *initMongo.js* añadiendo un *GUID* fijo a cada movimiento para así poder buscar por este. Actualmente, al no tener *GUID* en el archivo de datos iniciales no podremos buscar por *GUID* porque los genera de forma aleatoria cada vez que buscamos, sin añadir este a la base de datos.

## ✗ Errores:

Queda pendiente solucionar un error previo a los cambios que sigue pasando independientemente de estos.

Este error aparece al realizar cualquier consulta de autorización en *GraphQL* después de haber hecho una consulta previa. Sin embargo, si la consulta de autorización es la primera en hacerse, esta funcionará sin problema.



## 5. Mejoras en el Manejo de Excepciones

### Problema:

En el sistema anterior, se lanzaban excepciones directamente sin un manejo adecuado. Esto generaba errores no controlados que no proporcionaban suficiente contexto sobre el origen del problema, y no se gestionaban correctamente en el middleware de excepciones.

```
if (_httpContextAccessor.HttpContext == null)
    throw new Exception("HttpContext is null");

var user = _httpContextAccessor.HttpContext.User;

var id = user.FindFirst(ClaimTypes.NameIdentifier)?.Value;
if (string.IsNullOrEmpty(id))
    throw new Exception("User ID claim is missing");

var userForFound = await _userService.GetUserByIdAsync(id)
    ?? throw new UserNotFoundException(id);
var client = await _clientRepository.GetByIdAsync(id)
    ?? throw new ClientExceptions.ClientNotFoundException(id);

var product = await _productRepository.GetByNameAsync(request.ProductName);
if (product == null)
    throw new AccountsExceptions.AccountNotCreatedException();

var iban = await _ibanGenerator.GenerateUniqueIbanAsync();
if (string.IsNullOrEmpty(iban))
    throw new Exception("IBAN generation failed");
```

### Solución:

Se han implementado excepciones personalizadas para un manejo más preciso y claro de los errores. Las nuevas excepciones que se han añadido son las siguientes:

1. **HttpContextNull**: se lanza cuando el Http Context es null, lo cual impide realizar ciertas operaciones dependientes del contexto HTTP.
2. **UserIdMissing**: Se lanza cuando no se encuentra el User Id en el contexto, lo que impide identificar al usuario.
3. **IbanGeneratorFail**: Se lanza cuando el generador de Iban no puede generar un valor válido.
4. **ClientUnprocessable**: Se lanza cuando se intenta serializar a json para exportar los datos del cliente.
5. **ClientStorageException**: Excepción general que captura errores como espacio de almacenamiento insuficiente, el directorio es inaccesible o ya existe un documento con el mismo nombre al intentar exportar los datos del cliente.

Estas excepciones permiten manejar los errores de manera más controlada y proporcionar un mensaje específico sobre la causa del problema.

### Implementación de las Excepciones Personalizadas:

```

if (_httpContextAccessor.HttpContext == null)
    throw new ContextExceptions.HttpContextNull();

var user :ClaimsPrincipal = _httpContextAccessor.HttpContext.User;

var id :string? = user.FindFirst( type: ClaimTypes.NameIdentifier)?.Value;
if (string.IsNullOrEmpty(id))
    throw new ContextExceptions.UserIdMissing();

var userForFound :UserResponse = await _userService.GetUserByIdAsync(id)
    ?? throw new UserNotFoundException(id);
var client = await _clientRepository.GetByIdAsync(id)
    ?? throw new ClientExceptions.ClientNotFoundException(id);

var product = await _productRepository.GetByNameAsync(request.ProductName);
if (product == null)
    throw new AccountsExceptions.AccountNotCreatedException();

var iban :string = await _ibanGenerator.GenerateUniqueIbanAsync();
if (string.IsNullOrEmpty(iban))
    throw new IbanGeneratorExceptions.IbanGeneratorFail();

```

### Test de las Excepciones:

Se han creado pruebas unitarias para verificar que las excepciones se lanzan correctamente en los siguientes escenarios

## Middleware de Manejo de Excepciones:

Se ha actualizado el middleware de manejo de excepciones para capturar estas nuevas excepciones personalizadas y devolver respuestas adecuadas al cliente con los detalles del error.

```

/***** CONTEXT EXCEPTIONS *****/
case ContextExceptions.HttpContextNull:
    statusCode = HttpStatusCode.BadRequest;
    errorResponse = new { message = exception.Message };
    logger.LogWarning(exception, exception.Message);
    break;

case ContextExceptions.UserIdMissing:
    statusCode = HttpStatusCode.NotFound;
    errorResponse = new { message = exception.Message };
    logger.LogWarning(exception, exception.Message);
    break;

/***** IBANGENERATOR EXCEPTIONS *****/
case IbanGeneratorExceptions.IbanGeneratorFail:
    statusCode = HttpStatusCode.BadRequest;
    errorResponse = new { message = exception.Message };
    logger.LogWarning(exception, exception.Message);
    break;
```

```

case ClientExceptions.ClientUnprocessable:
    statusCode = HttpStatusCode.UnprocessableEntity;
    errorResponse = new { message = exception.Message };
    logger.LogWarning(exception, exception.Message);
    break;

case ClientExceptions.ClientStorageException:
    statusCode = HttpStatusCode.ServiceUnavailable;
    errorResponse = new { message = exception.Message };
    logger.LogWarning(exception, exception.Message);
    break;
```

## Mejora:

El sistema ahora maneja de manera centralizada las excepciones personalizadas, lo que mejora la trazabilidad de los errores y permite una mejor experiencia de usuario al proporcionar mensajes específicos sobre los problemas que ocurren en el flujo.

## Test:

Se han actualizado los tests para verificar que las excepciones se lanzan correctamente y que el middleware maneja las excepciones adecuadamente, devolviendo los códigos de estado y los mensajes correctos.



## 6. Mejoras en la Configuración del OnModelCreating

### Problema:

En el sistema anterior, las propiedades CreateAt y UpdateAt no se configuraban de manera automática en las entidades, lo que requería que se asignaran manualmente cada vez que se creaba o actualizaba una entidad. Esto podría generar errores y problemas de consistencia, ya que no se garantizaba un valor correcto o consistente para estas propiedades en todos los casos.

### Solución:

Se ha modificado el método OnModelCreating para que las propiedades CreateAt y UpdateAt se gestionen automáticamente durante el ciclo de vida de las entidades. Esta configuración garantiza que:

- CreateAt se asigne automáticamente cuando se crea una entidad.
- UpdateAt se actualice automáticamente cuando se actualiza una entidad.

Se ha usado la configuración de ValueGeneratedOnAdd() para la propiedad CreatedAt, y ValueGeneratedOnUpdate() para la propiedad UpdatedAt. Esto asegura que los valores se generen automáticamente al añadir o actualizar registros en la base de datos.

### Implementación:

El método OnModelCreating del DbContext ha sido actualizado de la siguiente manera:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    // Configuración de la entidad Account
    modelBuilder.Entity<Account>(entity =>
    {
        entity.Property(e => e.CreatedAt)
            .IsRequired()
            .ValueGeneratedOnAdd();

        entity.Property(e => e.UpdatedAt)
            .IsRequired()
            .ValueGeneratedOnUpdate();
    });
}
```

### Explicación de la Configuración:

- **ValueGeneratedOnAdd()**: Este método asegura que el valor de CreatedAt se genere automáticamente cuando se añada una nueva entidad en la base de datos.
- **ValueGeneratedOnUpdate()**: Este método asegura que el valor de UpdatedAt se actualice automáticamente cuando se modifique una entidad existente en la base de datos.

### Mejora:

Con esta configuración, no es necesario asignar manualmente los valores de CreatedAt y UpdatedAt cada vez que se crea o actualiza una entidad. Esto reduce la posibilidad de errores y garantiza la consistencia de los valores.

## 7. Métodos añadidos ausentes y refactorización

Destaca la ausencia de algunos métodos necesarios y pedidos en el proyecto como la capacidad de un cliente de eliminar los datos personales que existen en la base de datos, se han añadido y testeado.

También se ha refactorizado código para evitar la duplicación innecesaria de líneas, creando nuevos métodos privados reutilizables, facilitando la legibilidad y mantenimiento del proyecto. A su vez se ha ajustado el uso de la caché en Redis, eliminando de la caché los clientes eliminados e insertarlos cuando se buscan.

(Antes)

```
public async Task<ClientResponse> GettingMyClientData()
{
    var user = _httpContextAccessor.HttpContext!.User;
    var id = user.FindFirst(ClaimTypes.NameIdentifier)?.Value;
    var userForFound = await _userService.GetUserByIdAsync(id);
    if (userForFound == null)
        throw new UserNotFoundException(id);
    var client = await _clientRepository.getByUserIdAsync(userForFound.Id);
    if (client == null)
        throw new ClientExceptions.ClientNotFoundException(userForFound.Id);
    return client.ToResponse();
}
```

(Después)

```
5 usages KevinSanchez5
private async Task<string?> ValidateLoggedUser()
{
    var user :ClaimsPrincipal = _httpContextAccessor.HttpContext!.User;
    var id :string? = user.FindFirst(type: ClaimTypes.NameIdentifier)?.Value;
    var userForFound :UserResponse = await _userService.GetUserByIdAsync(id);

    if (userForFound == null)
        throw new UserNotFoundException(id);
    return id;
}

4 usages KevinSanchez5
private async Task<Client?> ValidateClientOfUser(string id)
{
    var client = await _clientRepository.getByUserIdAsync(id);
    if (client == null)
        throw new ClientExceptions.ClientNotFoundException(id);
    return client;
}

3 usages KevinSanchez5
private async Task<Client?> FindClient(string id)
{
    return await _clientRepository.GetByIdAsync(id)? throw new ClientExceptions.ClientNotFoundException(id);
}
```

```
public async Task<ClientResponse> GettingMyClientData()
{
    var userId :string? = await ValidateLoggedUser();
    var client = await ValidateClientOfUser(userId!);
    return client!.ToResponse();
}
```

## 8. Mejoras en la consistencia de datos/escalabilidad.

En el proyecto presentado se ha observado un problema referente a los datos, específicamente a aquellos generados automáticamente por el sistema y con valor único, principalmente referente a los números de las tarjetas de crédito e Iban de cuenta.

Aunque la posibilidad es mínima siempre se debe pensar en una alta escalabilidad del producto ya que en cualquier momento puede ser necesario una ampliación de los servicios por demanda del cliente, en este caso si esperamos poca afluencia en nuestro negocio este problema es apenas perceptible ya que se trata de una posibilidad ínfima de que ocurra, pero si no tenemos un control de los mismos puede llegar a ocurrir que el sistema genere dos números idénticos en campos que nunca deben ser idénticos.

Por eso para solucionar el problema en este caso hemos añadido una pequeña validación al servicio de tarjeta para que en el momento que se genere el número de la tarjeta sea dentro de un bucle el cual solo podrá terminar una vez que hayamos obtenido ese número único que buscamos, en el caso de las cuentas podríamos hacer la misma acción para los Iban.

Estas son las líneas que deberemos añadir al servicio de tarjeta para realizar nuestra comprobación:

```
var numeroTarjeta="";
do{
    numeroTarjeta = _numberGenerator.GenerateCreditCard();
} while (!(await GetByCardNumber(numeroTarjeta) == null));

creditCardModel.CardNumber = numeroTarjeta;
creditCardModel.ExpirationDate = _expirationDateGenerator.GenerateRandomDate();
creditCardModel.Cvc = _cvcGenerator.Generate();
```

## 9. Mejoras en la estructura de métodos

En el proyecto presentado se ha detectado una forma inusual en la estructuración de los métodos, métodos con muchos condicionales “if,else” , que pueden dificultar la lectura y entendimiento del propio método, por lo que hemos optado en la implementación de métodos auxiliares que puedan validar distintos campos sin la utilización de un condicional.

¿Por qué hacemos esto?.Según los principios SOLID de la programación, uno de sus principios es el llamado Principio de Responsabilidad Única o Single Responsibility Principle. El cual establece que un único método debe tener una única funcionalidad y una única razón para cambiar

Los beneficios que nos aporta la utilización de este principio son:

- Reducción de la extensión del método
- Más flexibilidad y maniobrabilidad a la hora de modificar el método
- Hace más accesible y sencillo su utilización

Hemos implementado este principio en métodos como los GetById y el AddUser, que utilizan 2 condicionales para su funcionamiento

Estos son los cambios que hemos implementado:

Antes:

```

public async Task<UserResponse> AddUserAsync(CreateUserRequest userRequest)
{
    if (!UserValidator.ValidateDni(userRequest.Dni))
    {
        throw new InvalidDniException(userRequest.Dni);
    }
    User newUser = userRequest.toUser();
    User? userWithTheSameUsername = await GetByUsernameAsync(userRequest.Dni);
    if (userWithTheSameUsername != null)
    {
        throw new UserAlreadyExistsException(userRequest.Dni);
    }
    await _userRepository.AddAsync(newUser);
    var notificacion = new Notification<UserResponse>
    {
        Type = Notification<UserResponse>.NotificationType.Create.ToString(),
        CreatedAt = DateTime.Now,
        Data = newUser.ToUserResponse()
    };
    var user = _httpContextAccessor.HttpContext!.User;
    var id = user.FindFirst(ClaimTypes.NameIdentifier)?.Value;
    await _webSocketHandler.NotifyUserAsync(id, notificacion);
    return newUser.ToUserResponse();
}

```

Ahora:

```

public async Task<UserResponse> AddUserAsync(CreateUserRequest userRequest)
{
    ValidateUserRequest(userRequest);

    User newUser = userRequest.toUser();
    await EnsureUserDoesNotExist(userRequest.Dni);

    await _userRepository.AddAsync(newUser);
    await NotifyUserCreation(newUser);

    return newUser.ToUserResponse();
}

```

Como se puede ver es un método más compacto y fácil de entender y con sus métodos auxiliares creados únicamente para desempeñar la función asignada