Proyecto

CLIENTES BANCOS

Desarrollo Web en Entorno Servidor.

Christian Gregorio Asensio
Diego Novillo
Kelvin
Oscar
Tomás

ÍNDICE

2. Presentación del grupo y resumen de la aplicación

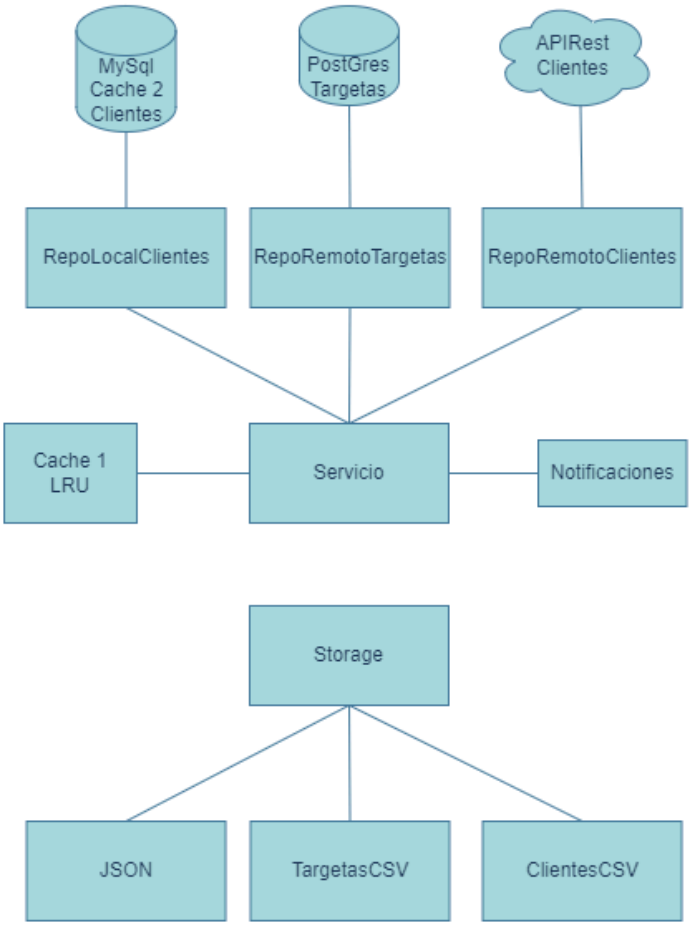
Arquitectura

- El proyecto está realizado usando una **arquitectura orientada al dominio**, ya que:

- Permite una mejor mantenibilidad y evolución del código, ya que el diseño está más cercano al modelo mental del dominio del problema.

- Facilita la implementación de cambios y mejoras en el sistema, ya que el modelo del dominio es más fácil de entender y modificar.

El proyecto está basado en este esquema que separa las distintas partes de la aplicación que son:



- **Repositorios**: se pueden distinguir tres repositorios que dependen en tres fuentes distintas:

    - Un repositorio local que almacena la información de ambos clientes y tarjetas de crédito y que funciona como una segunda caché en la que se almacena información por un periodo de tiempo limitado.

    - Un repositorio remoto que se conecta a una base de datos PostGres que en nuestra aplicación estará en un contenedor Docker.

    - Un repositorio remoto que se conectará al endpoint "https://jsonplaceholder.typicode.com/users" para gestionar los clientes sin sus tarjetas.

- **Servicios:** en nuestra aplicación solo habrá un servicio que se encargará de gestionar todos los repositorios y la caché al igual que es el encargado de pasar los clientes por validadores.

- **Caché**: una caché de tamaño limitado que cada minuto llama a un "cleaner" que se encarga de eliminar entradas que no han sido actualizadas en un minuto siguiendo el principio LRU ("least recently used").

- **Storage**: consiste en una clase que llama a funciones de diferentes storages que se especializan en diferentes tipos de archivos. Nuestro proyecto solo necesita poder importar y exportar tarjetas y clientes en ficheros CSV y JSON. Este storage engloba tres diferentes:

    - CardsStorageCSV: importa y exporta tarjetas en ficheros de formato CSV.

    - ClientesStorageCSV: importa y exporta clientes en ficheros de formato CSV.

    - ClientesStorageJSON: importa y exporta clientes y tarjetas juntos en un archivo JSON.

Rendimiento

- Para esta aplicación debemos tener en cuenta el rendimiento de la aplicación ya que hacemos uso de base de datos y un endpoint online al igual que, para eso todos los repositorios están hechos asíncronos con Completable Future y los storage están hechos de manera reactiva.

Patrones de diseño

- La mayoría de las clases siguen el patrón Singleton evitando la creación de múltiples instancias de la misma clase, esto incluye los repositorios, la cache, el servicio y los managers de las bases de datos.

3. Especificación de requisitos

- **Especificación de Requisitos Funcionales**

1. **Gestionar clientes y sus tarjetas de crédito**:

    - Guardar un cliente: la aplicación debe almacenar los datos de un cliente y sus tarjetas de crédito.

    - Actualizar la información del cliente: se debe poder actualizar un cliente y las tarjetas que le pertenezcan.

    - Borrar clientes y tarjetas: la aplicación necesita poder borrar clientes y sus tarjetas.

    - Buscar clientes: tenemos que poder encontrar clientes por su id.

2. **Validación de clientes y tarjetas**:

    - La aplicación debe validar los clientes y las tarjetas de crédito antes de guardarlas para asegurarse de que no se guardan datos incorrectos.

3. **Manejo de errores**:

    - La aplicación debe devolver errores en el caso de que haya problemas en el funcionamiento de la aplicación o en el caso de que se intente guardar algo que no es válido.

4. **Leer y escribir diferentes tipos de ficheros**:

    - Debe poder importar clientes y exportarlos en ficheros JSON.

    -Debe poder importar y exportar clientes (sin sus respectivas tarjetas) a ficheros CSV.

    -Debe poder importar y exportar tarjetas a ficheros CSV.

5. Debe tener un **sistema de notificaciones** que informe cuando se guarda un nuevo cliente en la aplicación.

- **Especificación de requisitos no funcionales**

1. **Bases de datos:** la aplicación debe usar varias bases de datos para guardar los datos.

    - MySQL: para guardar los clientes y las tarjetas en una base de datos local.

    - PostGres: para guardar las tarjetas de crédito dentro de un contenedor Docker.

2. **Uso de una APIRest** para el almacenamiento de los clientes.

3. **Caché LRU**:

    - Implementar una política de caché **LRU** (Least Recently Used) con un tamaño máximo determinado para gestionar los clientes.

4. **Asincronía y reactividad**:

    - La gestión de entrada y salida de datos debe ser **asíncrona** y, en los casos necesarios, reactiva.

5. **Despliegue con Docker**:

    - Desplegar toda la infraestructura usando **Docker** y, para la aplicación, usar un **Docker multi-stage build**.

    - Publicar la app en **DockerHub**.

6. **Uso de TestContainers**:

    - Usar **TestContainers** para las pruebas de integración con bases de datos.

7. **Sistema de logs**:

    - Se debe usar un sistema de **logs** para registrar las operaciones del sistema.

8. **Gestión de errores y excepciones**:

    - Manejar correctamente los errores y excepciones en todas las operaciones del sistema.

9. **Versionado y gestión del código**:

    - Usar **GitFlow** para la gestión del código y realizar Pull Requests (PR) por cada tarea.

10. **Infraestructura en contenedores**:

    - Usar **TestContainers** para las pruebas con bases de datos.

    - Desplegar toda la infraestructura usando Docker.

11. **Build y despliegue continuo**:

    - Subir la aplicación a **DockerHub** a partir del repositorio de GitHub.

- **Especificación de requisitos de información**

- La aplicación debe almacenar la información de la siguiente manera:

    - **Clientes**:

        - **Id**: un número que nos permite identificarlos en el sistema

        - **Nombre**: el nombre del cliente;

        - **Nombre de usuario**: un nombre que debe ser único por cada cliente

        - **Email**: el email del cliente.

        -**Tarjetas** de crédito: las tarjetas de crédito que le corresponden.

        -**Created_At**: la fecha en la que se metió en la base de datos

        -**Updated_At**: la fecha de la última vez que se actualizó.

- **Tarjeta de crédito**:

    - **Número**: el número de la tarjeta que debe ser único.

    - **Titular**: el id del cliente al cual le pertenece la tarjeta.

    - **Fecha de caducidad**: la fecha en la que se caduca la tarjeta.

    - **Created_At**: la fecha en la que se metió en la base de datos

    -**Updated_At**: la fecha de la última vez que se actualizó.

4. Casos de uso

Caso de Uso: Guardar Cliente

Descripción: El sistema permite al usuario guardar un nuevo cliente con la información proporcionada.

Precondición:

- El sistema está activo.
- El usuario tiene los datos requeridos del cliente (nombre, nombre de usuario y correo electrónico).

Secuencia Normal:

1. El usuario ingresa los datos del cliente a guardar.
2. El validador verifica que los campos obligatorios están correctos.
3. El servicio busca en el repositorio y la caché si el cliente ya tiene un identificador asignado.
4. Si no se encuentra un cliente con ese identificador, se guarda el nuevo cliente en el repositorio local, el remoto y la caché.
5. Se envía una notificación indicando que se ha creado un nuevo cliente.
6. Finalmente, se devuelve el cliente guardado con todos sus datos (incluyendo identificador, fecha de creación y última actualización).

Secuencia Alternativa:

- Si los datos del cliente no son válidos, el validador lanzará una `ClientExceptionBadRequest`.
- Si el identificador ya está ocupado, se lanzará una `ClientExceptionBadRequest` indicando que el cliente ya existe.

Postcondición: El cliente ha sido guardado correctamente en el sistema.

Caso de Uso: Buscar Cliente por Identificador

Descripción: El sistema permite al usuario buscar un cliente a partir de su identificador.

Precondición:

- El sistema está activo.
- El usuario tiene un identificador de tipo `Long`.

Secuencia Normal:

1. El sistema busca primero el cliente en la caché.
2. Si no lo encuentra en la caché, busca en el repositorio local.
3. Si tampoco está en el repositorio local, consulta el repositorio remoto (API REST).
4. Si se encuentra el cliente en algún repositorio, se devuelve y se almacena en la caché y el repositorio local (si es obtenido del remoto).

Secuencia Alternativa:

- Si el cliente no se encuentra en ninguno de los repositorios, se muestra un mensaje indicando que no existe un cliente con ese identificador.

Postcondición: El cliente ha sido encontrado o se ha indicado que no existe.

Caso de Uso: Buscar Todos los Clientes

Descripción: El sistema permite al usuario obtener una lista de todos los clientes almacenados en el repositorio remoto (API REST).

Precondición:

- El sistema está activo.

Secuencia Normal:

1. El sistema consulta el repositorio remoto para obtener todos los clientes.
2. Si se encuentran clientes, se devuelve una lista de ellos; de lo contrario, se devuelve una lista vacía.

Secuencia Alternativa:

- No aplica.

Postcondición: Se ha obtenido una lista de todos los clientes almacenados o una lista vacía si no hay registros.

Caso de Uso: Actualizar Cliente

Descripción: El sistema permite al usuario actualizar la información de un cliente existente.

Precondición:

- El sistema está activo.
- El usuario tiene un identificador válido y los datos actualizados del cliente.

Secuencia Normal:

1. El usuario ingresa el identificador del cliente y los datos a actualizar.
2. El validador verifica que los nuevos datos del cliente son correctos.
3. Si los datos son válidos, el sistema actualiza el cliente en el repositorio remoto, en el repositorio local y en la caché.
4. Se envía una notificación indicando que el cliente ha sido actualizado.

Secuencia Alternativa:

- Si los datos del cliente son incorrectos, se lanza una `ClientExceptionBadRequest`.
- Si falla la actualización en alguno de los repositorios, se registra el error en el log y se lanza una `RuntimeException`.

Postcondición: El cliente ha sido actualizado correctamente o se ha registrado el fallo.

Caso de Uso: Borrar Cliente

Descripción: El sistema permite al usuario eliminar un cliente a partir de su identificador.

Precondición:

- El sistema está activo.
- El usuario tiene un identificador válido.

Secuencia Normal:

1. El sistema elimina el cliente en la caché, el repositorio local y el repositorio remoto utilizando el identificador proporcionado.
2. Se envía una notificación indicando que el cliente ha sido eliminado.

Secuencia Alternativa:

- Si el cliente no se encuentra en ninguno de los repositorios, se registra el error en el log y se lanza una excepción indicando que el cliente no existe.

Postcondición: El cliente ha sido eliminado del sistema o se ha registrado el fallo.

-

b) Tarjetas bancarias

Caso de Uso: **Guardar Tarjeta Bancaria**

Descripción: El sistema permite al usuario guardar una nueva tarjeta bancaria asociada a un cliente.
Precondición:

- El sistema está activo.
- El usuario tiene los datos necesarios de la tarjeta.

Secuencia Normal:

1. El usuario ingresa los datos de la tarjeta bancaria a guardar.
2. El validador se asegura de que los campos están bien.
3. El servicio verifica en la caché y en el repositorio si el número de tarjeta ya existe.
4. Si no hay nada con ese número, se guarda la tarjeta en el repositorio de PostgreSQL y en la caché.
5. Se lanza una notificación de que una nueva tarjeta ha sido creada.
6. Finalmente, se devuelve la tarjeta guardada con sus datos.

Secuencia Alternativa:

- Si el número de la tarjeta está mal estructurado, el validador enviará un error de `BankCardException`.
- Si el número de tarjeta ya existe, devolverá un error `BankCardException` indicando que ya existe.

Postcondición:

- La tarjeta bancaria ha sido guardada correctamente en el sistema.

Caso de Uso: Buscar Tarjeta Bancaria por Número

Descripción: El sistema permite al usuario buscar una tarjeta bancaria a partir de su número.
Precondición:

- El sistema está activo.

Secuencia Normal:

1. El usuario ingresa el número de la tarjeta bancaria a buscar.
2. El sistema busca primero en la caché.
3. Si no encuentra la tarjeta en la caché, busca en el repositorio de PostgreSQL.

4. Si encuentra la tarjeta, la devuelve y la almacena en la caché para futuras búsquedas.

Secuencia Alternativa:

- Si no encuentra la tarjeta en ningún repositorio, muestra un mensaje de que no existe la tarjeta con ese número.

Postcondición:

- La tarjeta bancaria ha sido encontrada y devuelta al usuario.

Caso de Uso: Buscar Todas las Tarjetas Bancarias

Descripción: El sistema permite al usuario buscar todas las tarjetas bancarias almacenadas.

Precondición:

- El sistema está activo.

Secuencia Normal:

1. El usuario elige buscar todas las tarjetas bancarias.
2. El sistema busca todas las tarjetas en el repositorio de PostgreSQL.
3. Si encuentra tarjetas, las devuelve como una lista al usuario.
4. Si no encuentra ninguna tarjeta, devuelve una lista vacía.

Secuencia Alternativa:

- No hay.

Postcondición:

- Todas las tarjetas bancarias han sido encontradas y devueltas al usuario.

Caso de Uso: Actualizar Tarjeta Bancaria

Descripción: El sistema permite al usuario actualizar los datos de una tarjeta bancaria existente.

Precondición:

- El sistema está activo.
- El usuario tiene los datos actualizados de la tarjeta bancaria.

Secuencia Normal:

1. El usuario ingresa los datos actualizados de la tarjeta bancaria y el número de la tarjeta a actualizar.
2. Se valida la tarjeta a actualizar.
3. Si los datos son correctos, se actualiza la tarjeta en el repositorio de PostgreSQL y en la caché.
4. Se envía una notificación avisando de que se ha actualizado una tarjeta bancaria.

Secuencia Alternativa:

- Si los datos de la tarjeta son incorrectos, se lanza una excepción `BankCardException`.
- Si falla la actualización en cualquier repositorio, se muestra un error en el log y se lanza una `RuntimeException`.

Postcondición:

- Los datos de la tarjeta bancaria han sido actualizados correctamente.

Caso de Uso: Borrar Tarjeta Bancaria

Descripción: El sistema permite al usuario borrar una tarjeta bancaria a partir de su número.

Precondición:

- El sistema está activo.

Secuencia Normal:

1. El usuario ingresa el número de la tarjeta bancaria a borrar.
2. El sistema borra la tarjeta a partir del número en la caché y en el repositorio de PostgreSQL.
3. Se envía una notificación avisando de que una tarjeta bancaria ha sido eliminada.

Secuencia Alternativa:

- En caso de no encontrar la tarjeta en los repositorios, se muestra un mensaje de error y se lanza una excepción.

Postcondición:

- La tarjeta bancaria ha sido eliminada correctamente del sistema.

Caso de Uso: Importar Tarjetas Bancarias desde CSV

Descripción: El sistema permite al usuario importar tarjetas bancarias desde un archivo CSV.

Precondición:

- El sistema está activo.
- El usuario tiene credenciales para acceder a la funcionalidad de importación.

Secuencia Normal:

1. El usuario elige importar tarjetas bancarias.
2. El sistema solicita al usuario que seleccione el archivo CSV a importar.
3. El usuario selecciona el archivo CSV.
4. El sistema procesa el archivo CSV.
5. El usuario confirma la importación de las tarjetas.
6. El sistema verifica los datos de las tarjetas a importar.
7. Si los datos son correctos, el sistema añade las tarjetas al repositorio de PostgreSQL y a la caché.

8. El sistema muestra un mensaje de confirmación de que las tarjetas han sido importadas correctamente.

Secuencia Alternativa:

- Si hay errores en los datos de las tarjetas en el archivo CSV, el sistema muestra un mensaje de error y permite al usuario corregir los datos o seleccionar otro archivo.

Postcondición:

- Las tarjetas del archivo CSV habrán sido importadas correctamente al sistema.

Caso de Uso: Exportar Tarjetas Bancarias a CSV

Descripción:

El sistema permite al usuario exportar los datos de las tarjetas bancarias a un archivo CSV.

Precondición:

- El sistema está activo.
- El usuario tiene credenciales y permisos para exportar datos.

Secuencia Normal:

1. El usuario selecciona la opción para exportar tarjetas bancarias.
2. El sistema solicita al usuario que elija la ubicación y el nombre del archivo CSV a exportar.
3. El usuario proporciona la ubicación y el nombre del archivo.
4. El sistema recupera todas las tarjetas bancarias del repositorio.
5. El sistema crea un archivo CSV en la ubicación especificada.
6. El sistema escribe los datos de cada tarjeta bancaria en el archivo CSV.
    a. Si los datos se escriben correctamente, el sistema continúa.
7. El sistema muestra un mensaje de confirmación indicando que la exportación se ha completado con éxito.

Secuencia Alternativa:

6.2. Si ocurre un error al escribir los datos en el archivo CSV, el sistema muestra un mensaje de error, registra el problema en los logs y permite al usuario intentar la operación nuevamente.

Postcondición:

- Los datos de las tarjetas bancarias se habrán exportado correctamente al archivo CSV en la ubicación especificada por el usuario.

5. Presupuesto

Informe de Desglose de Costes Asociados al Desarrollo del Proyecto “Clientes Bancos”:

Salario de los Desarrolladores:
5 desarrolladores × 13.36€/hora × 8 horas/día × 5 días/semana × 2 semanas = **5,344€**

Coste Herramientas de Desarrollo:

- IntelliJ IDEA Ultimate: Suscripción 599€ al año por usuario. Coste para 2 semanas (aproximadamente 1/26 de un año): 5 desarrolladores × 599€/año × 1/26 = **115.19€**
- GitHub Team: Suscripción 48€/Año. Coste para 2 semanas (aproximadamente 1/26 de un año): 5 desarrolladores × 48€/año × 1/26 = **9.23€**

Total coste herramientas desarrollo para 5 desarrolladores: 115.19€ + 9.23€ = **124.42€**

Coste Amortización Herramienta Equipos Portátiles:
Con una vida aproximada de 3 años (36 meses). Calculo el coste de amortización mensual de los equipos portátiles valorados cada uno en 1500€, dando como resultado:
5 equipos portátiles × 1500€/equipo / 36 meses = 208.33€ /mes

Para un proyecto de 2 semanas (aproximadamente 0.5 meses), el costo de amortización resultante es: 208.33€/mes × 0.5 meses = **104.17**

Coste de Pruebas:
Porcentaje de tiempo dedicado a test para garantizar eficiencia, estabilidad y seguridad de la aplicación:

20% de salario desarrolladores. 20% de salario desarrolladores. 5,344€ × 20% = **1,068.80€**

Coste de Mantenimiento:
15% del costo total del proyecto por año. Costo de mantenimiento para el primer año de servicio de la aplicación sería:

(5,344€ + 124.42€ + 1,068.80€) × 15% = **937.61€**

Coste del Servidor:
Estimado en 50€ al mes para un servicio del primer año:

50€/mes × 12 = **600€**

Por lo tanto, el costo total estimado del proyecto sería:

5,344€ (salario de los desarrolladores) +
124.42€ (costo de las herramientas de desarrollo) +
1,068.80€ (costo de pruebas) +
937.61€ (costo de mantenimiento) +
104.17€ (Amortización equipos) +
600€ (servidor)
= **8,179.00€**

Margen de Beneficio:
25% del costo total del proyecto: 8,179.00€ × 0.25 = **2,044.75€**

Coste final para cliente:

8,179.00€ (costo total del proyecto) + 2,044.75€ (25% beneficio) = **10,223.75€** sin IVA

10,223.75€ (precio total antes del IVA) * 21% (IVA) = **2,147.99€** de IVA

Precio total cobrado al cliente:
10,223.75€ + 2,147.99€ IVA = **TOTAL IVA INCLUIDO 12,371.74€**