API Vuelos

Especificaciones de Requerimientos de Software

Versión 1

17/9/2024

**Requerimientos funcionales**

**Requerimiento funcional #1 Alta de cliente**

Descripción: A través del endpoint /clientes/alta, se podrá añadir un usuario a la base de datos.

Inputs (datos de entrada):

* DNI\_Cli
* Nombre\_Cli
* Email\_Cli

Proceso:

1. El sistema valida que los campos no estén vacíos
2. El sistema valida que no existan dos usuarios con el mismo DNI
3. Si los datos son correctos, se agrega el nuevo usuario con el campo de Estado\_Cli en “True” y el ID se asigna automáticamente con la estrategia “IDENTITY”. Caso contrario, se indicará el error.

Outputs:

* Mensaje de éxito/error

Error Handling:

* Si no se ingresaron todos los datos, se lanzará un mensaje de error “Faltan completar datos”
* Si el DNI ya existe, se lanzará un mensaje “El DNI ingresado ya fue registrado”

**Requerimiento funcional #2 Baja lógica de cliente**

Descripción: A través del endpoint /clientes/baja/{id}, se podrá dar de baja a un cliente de forma lógica (se cambiará su estado de true a false).

Inputs (datos de entrada):

* ID\_Cli

Proceso:

* El sistema valida que el id exista
* Si el id existe, se modificará el estado del cliente a false. Caso contrario, lanzará un mensaje de error

Outputs:

* Mensaje de éxito/error

Error Handling:

* Si el id no existe, se lanzará el siguiente mensaje “No existe el id ingresado. Por favor intente con otro.”

**Requerimiento funcional #3 Modificación de cliente**

Descripción: A través del endpoint /clientes/modificacion/{id}, se podrá modificar los datos de un cliente. No podrá cambiar su DNI o ID, pero si tendrá permiso para cambiar los demás datos.

Inputs (datos de entrada):

* Id\_Cli
* Nombre\_Cli
* Email\_Cli

Proceso:

* El sistema verificará que el id ingresado en el endpoint exista
* El sistema verificará que los campos a modificar no estén vacíos
* Si los datos son correctos, se modificará el cliente en la base de datos. Caso contrario, aparecerá un mensaje de error

Outputs:

* Mensaje de éxito/error

Error Handling:

* Si el id no existe, se lanzará el siguiente mensaje “No existe el id ingresado. Por favor intente con otro”
* Si no se completan todos los campos, se lanzará el siguiente mensaje “Completar todos los campos”

**Requerimiento funcional #4 Listado de clientes**

Descripción: A través del endpoint /clientes/, se listarán todos los clientes que haya en la base de datos en formato JSON.

Inputs (datos de entrada): No hay datos de entrada

Proceso:

* El sistema traerá todos los clientes de la base de datos en formato JSON

Outputs:

* ID\_Cli
* DNI\_Cli
* Nombre\_Cli
* Email\_Cli

Error Handling:

**Requerimiento funcional #5 Buscado de cliente**

Descripción: A través del endpoint /cliente/{id}, se traerá al cliente con el id especificado de la base de datos en formato JSON.

Inputs (datos de entrada):

* ID\_Cli

Proceso:

* El sistema verifica que el id del cliente ingresado exista
* El sistema trae el cliente de la base de datos en formato JSON

Outputs:

* ID\_Cli
* DNI\_Cli
* Nombre\_Cli
* Email\_Cli

Error Handling:

* Si el id no existe, se traerá un JSON vacío

**Requerimiento funcional #6 Alta de avión**

Descripción: A través del endpoint /aviones/alta, se podrá dar de alta un avión en la base de datos.

Inputs (datos de entrada):

* ID\_Modelo\_Av
* ID\_Tipo\_Av
* Capacidad\_Total\_Av

Proceso:

* Se verifica que los datos de entrada no estén vacíos y cumplan el formato deseado
* El sistema consume los servicios de tipo-service y modelo-service para corroborar que los ID de tipo y modelo de avión existan
* Si los datos son correctos, se agrega el avión en la base de datos. Se asigna el campo Estado\_Av con valor “True” y su ID se asigna de forma automática con la estrategia “IDENTITY”. Si los datos no son correctos, se informa con un mensaje.

Outputs:

* Mensaje de éxito/error

Error Handling:

* Si los datos no fueron completados, aparecerá un mensaje de “Debe completar todos los campos”
* Si hay algún campo que no cumple el formato deseado, aparecerá “Este campo no acepta números” o “Este campo no acepta letras” según sea el caso
* Si no existe el id del modelo de avión, se lanzará un mensaje de “No existe el modelo de avión ingresado”
* Si no existe el id del tipo de avión, se lanzará un mensaje de “No existe el tipo de avión ingresado”
* Si el sistema no se puede conectar a algún microservicio, lanzará un mensaje de “No se pudo conectar al microservicio [nombreMicroservicio]” gracias al patrón CircuitBreaker

**Requerimiento funcional #7 Baja lógica de avión**

Descripción: A través del endpoint /aviones/baja/{id}, se podrá dar de baja a un avión de forma lógica (se cambia el campo Estado\_Av a false).

Inputs (datos de entrada):

* ID\_Av

Proceso:

* El sistema corrobora que el id ingresado exista
* Si el id existe, se cambia el estado del avión a false. Caso contrario, saldrá un mensaje de error

Outputs:

* Mensaje de éxito/error

Error Handling:

* Si el id no existe, se lanzará el mensaje “El id ingresado no existe, por favor ingresar uno nuevo”

**Requerimiento funcional #8 Modificación de avión**

Descripción: A través del endpoint /aviones/modificacion/{id} se podrán modificar los datos de un avión ya ingresado

Inputs (datos de entrada):

* ID\_Av
* ID\_Modelo\_Av
* ID\_Tipo\_Av
* Capacidad\_Total\_Av

Proceso:

* El sistema verificará que el id ingresado en el endpoint exista
* El sistema verificará que los campos a modificar no estén vacíos
* El sistema llamará a los servicios de modelo-service y tipo-service y verificará que los ID de modelo y tipo de avión existan
* Si los datos son correctos, se modificará el avión en la base de datos. Caso contrario, aparecerá un mensaje de error

Outputs:

* Mensaje de éxito/error

Error Handling:

* Si el id del avión no existe, se lanzará el siguiente mensaje “No existe el id de avión ingresado. Por favor intente con otro”. Lo mismo pasará con los id de los modelos o tipos de avión
* Si no se completan todos los campos, se lanzará el siguiente mensaje “Completar todos los campos”
* Si no existe el id del modelo de avión, se lanzará un mensaje de “No existe el modelo de avión ingresado”
* Si no existe el id del tipo de avión, se lanzará un mensaje de “No existe el tipo de avión ingresado”
* Si el sistema no se puede conectar a algún microservicio, lanzará un mensaje de “No se pudo conectar al microservicio [nombreMicroservicio]” gracias al patrón CircuitBreaker

**Requerimiento funcional #9 Listado de aviones**

Descripción: A través del endpoint /aviones/ se listarán todos los aviones de la base de datos, llamando a los microservicios de modelo y tipo de avión para que el campo de estos tenga su respectivo nombre y no su ID. Los datos serán listados en formato JSON. Solo serán listados aquellos aviones con el estado en “true”.

Inputs (datos de entrada): No hay datos de entrada

Proceso:

* El sistema traerá todos los aviones con estado en true
* El sistema llamará a modelo-service y tipo-service para reemplazar el campo de ID de ambos por Nombre\_Modelo y Nombre\_Tipo respectivamente
* Se listarán todos los aviones que trajo el sistema en formato JSON

Outputs:

* ID\_Av
* Nombre\_Modelo
* Nombre\_Tipo
* Capacidad\_Av
* Estado\_Av

Error Handling:

* Si hay un error de conexión entre los microservicios, se devolverá un avión de tipo error (con id en 0 y todos los demás campos con valor “ERROR”) a través del patrón CircuitBreaker

**Requerimiento funcional #10 Buscado de avión**

Descripción: A través del endpoint /aviones/{id}, se podrá buscar un avión en específico. Este será listado en formato JSON y podrá tener el estado en “false”. También habrá un llamado a los microservicios de tipo y modelo

Inputs (datos de entrada):

* ID\_Av

Proceso:

* El sistema verifica que el ID exista en la base de datos
* El sistema consume los servicios de modelo-service y tipo-service para traer los datos necesarios
* Se lista el avión

Outputs:

* ID\_Av
* Nombre\_Modelo
* Nombre\_Tipo
* Capacidad\_Av
* Estado\_Av

Error Handling:

* Si no existe el ID, se traerá un JSON vacío
* Si hay un error de conexión entre los microservicios, se devolverá un avión de tipo error (con id en 0 y todos los demás campos con valor “ERROR”) a través del patrón CircuitBreaker

**Requerimiento funcional #11 Alta de modelo de avión**

Descripción: A través del endpoint /modelos/alta, se podrá dar de alta un modelo de avión.

Inputs (datos de entrada):

* Nombre\_Mo

Proceso:

* El sistema verifica que los campos no estén vacíos
* Si todo está correcto, se asignará el valor del campo Estado\_Mo a “true” y se le asignará un ID con la estrategia de generación de IDENTITY. Luego de esto, se carga en la base de datos

Outputs:

* Mensaje de éxito/error

Error Handling:

* Si hay campos vacíos, se devolverá un mensaje de “Completar todos los campos”

**Requerimiento funcional #12 Baja lógica de modelo de avión**

Descripción: A través del endpoint /modelos/baja/{id}, se podrá dar de baja a un modelo de avión de forma lógica (se cambia el campo Estado\_Mo a false).

Inputs (datos de entrada):

* ID\_Mo

Proceso:

* El sistema corrobora que el id ingresado exista
* Si el id existe, se cambia el estado del modelo de avión a false. Caso contrario, saldrá un mensaje de error

Outputs:

* Mensaje de éxito/error

Error Handling:

* Si el id no existe, se lanzará el mensaje “El id ingresado no existe, por favor ingresar uno nuevo”

**Requerimiento funcional #13 Modificación de modelo de avión**

Descripción: A través del endpoint /modelos/modificacion/{id} se podrán modificar los datos de un modelo de avión ya ingresado

Inputs (datos de entrada):

* ID\_Mo
* Nombre\_Mo

Proceso:

* El sistema verificará que el id ingresado en el endpoint exista
* El sistema verificará que los campos a modificar no estén vacíos
* Si los datos son correctos, se modificará el modelo de avión en la base de datos. Caso contrario, aparecerá un mensaje de error

Outputs:

* Mensaje de éxito/error

Error Handling:

* Si el id del modelo de avión no existe, se lanzará el siguiente mensaje “No existe el id ingresado. Por favor intente con otro”.
* Si no se completan todos los campos, se lanzará el siguiente mensaje “Completar todos los campos”

**Requerimiento funcional #14 Listado de modelos de avión**

Descripción: A través del endpoint /modelos/ se listarán todos los modelos de avión de la base de datos. Los datos serán listados en formato JSON. Solo serán listados aquellos modelos de avión con el estado en “true”.

Inputs (datos de entrada): No hay datos de entrada

Proceso:

* El sistema traerá todos los modelos de avión con estado en true
* Se listarán todos los modelos de avión que trajo el sistema en formato JSON

Outputs:

* ID\_Mo
* Nombre\_Mo
* Estado\_Mo

Error Handling:

**Requerimiento funcional #15 Buscado de modelo de avión**

Descripción: A través del endpoint /modelos/{id}, se podrá buscar un modelo de avión en específico. Este será listado en formato JSON y podrá tener el estado en “false”.

Inputs (datos de entrada):

* ID\_Mo

Proceso:

* El sistema verifica que el ID exista en la base de datos
* Se lista el modelo

Outputs:

* ID\_Mo
* Nombre\_Mo
* Estado\_Mo

Error Handling:

* Si no existe el ID, se traerá un JSON vacío

**Requerimiento funcional #16 Alta de tipo de avión**

Descripción: A través del endpoint /tipos/alta, se podrá dar de alta un tipo de avión.

Inputs (datos de entrada):

* Nombre\_Ti

Proceso:

* El sistema verifica que los campos no estén vacíos
* Si todo está correcto, se asignará el valor del campo Estado\_Ti a “true” y se le asignará un ID con la estrategia de generación de IDENTITY. Luego de esto, se carga en la base de datos

Outputs:

* Mensaje de éxito/error

Error Handling:

* Si hay campos vacíos, se devolverá un mensaje de “Completar todos los campos”

**Requerimiento funcional #17 Baja lógica de tipo de avión**

Descripción: A través del endpoint /tipos/baja/{id}, se podrá dar de baja a un tipo de avión de forma lógica (se cambia el campo Estado\_Ti a false).

Inputs (datos de entrada):

* ID\_Ti

Proceso:

* El sistema corrobora que el id ingresado exista
* Si el id existe, se cambia el estado del tipo de avión a false. Caso contrario, saldrá un mensaje de error

Outputs:

* Mensaje de éxito/error

Error Handling:

* Si el id no existe, se lanzará el mensaje “El id ingresado no existe, por favor ingresar uno nuevo”

**Requerimiento funcional #18 Modificación de tipo de avión**

Descripción: A través del endpoint /tipos/modificacion/{id} se podrán modificar los datos de un tipo de avión ya ingresado

Inputs (datos de entrada):

* ID\_Ti
* Nombre\_Ti

Proceso:

* El sistema verificará que el id ingresado en el endpoint exista
* El sistema verificará que los campos a modificar no estén vacíos
* Si los datos son correctos, se modificará el tipo de avión en la base de datos. Caso contrario, aparecerá un mensaje de error

Outputs:

* Mensaje de éxito/error

Error Handling:

* Si el id del tipo de avión no existe, se lanzará el siguiente mensaje “No existe el id ingresado. Por favor intente con otro”.
* Si no se completan todos los campos, se lanzará el siguiente mensaje “Completar todos los campos”

**Requerimiento funcional #19 Listado de tipos de avión**

Descripción: A través del endpoint /tipos/ se listarán todos los tipos de avión de la base de datos. Los datos serán listados en formato JSON. Solo serán listados aquellos tipos de avión con el estado en “true”.

Inputs (datos de entrada): No hay datos de entrada

Proceso:

* El sistema traerá todos los tipos de avión con estado en true
* Se listarán todos los tipos de avión que trajo el sistema en formato JSON

Outputs:

* ID\_Ti
* Nombre\_Ti
* Estado\_Ti

Error Handling:

**Requerimiento funcional #20 Buscado de tipo de avión**

Descripción: A través del endpoint /tipos/{id}, se podrá buscar un tipo de avión en específico. Este será listado en formato JSON y podrá tener el estado en “false”.

Inputs (datos de entrada):

* ID\_Ti

Proceso:

* El sistema verifica que el ID exista en la base de datos
* Se lista el tipo de avión

Outputs:

* ID\_Ti
* Nombre\_Ti
* Estado\_Ti

Error Handling:

* Si no existe el ID, se traerá un JSON vacío

**Requerimiento funcional #21 Alta de país**

Descripción: A través del endpoint /paises/alta, se podrá dar de alta un país.

Inputs (datos de entrada):

* Nombre\_Pa

Proceso:

* El sistema verifica que los campos no estén vacíos
* Si todo está correcto, se asignará el valor del campo Estado\_Pa a “true” y se le asignará un ID con la estrategia de generación de IDENTITY. Luego de esto, se carga en la base de datos

Outputs:

* Mensaje de éxito/error

Error Handling:

* Si hay campos vacíos, se devolverá un mensaje de “Completar todos los campos”

**Requerimiento funcional #22 Baja lógica de país**

Descripción: A través del endpoint /paises/baja/{id}, se podrá dar de baja un país de forma lógica (se cambia el campo Estado\_Pa a false).

Inputs (datos de entrada):

* ID\_Pa

Proceso:

* El sistema corrobora que el id ingresado exista
* Si el id existe, se cambia el estado del país a false. Caso contrario, saldrá un mensaje de error

Outputs:

* Mensaje de éxito/error

Error Handling:

* Si el id no existe, se lanzará el mensaje “El id ingresado no existe, por favor ingresar uno nuevo”

**Requerimiento funcional #23 Modificación de país**

Descripción: A través del endpoint /paises/modificacion/{id} se podrán modificar los datos de un país ya ingresado

Inputs (datos de entrada):

* ID\_Pa
* Nombre\_Pa

Proceso:

* El sistema verificará que el id ingresado en el endpoint exista
* El sistema verificará que los campos a modificar no estén vacíos
* Si los datos son correctos, se modificará el país en la base de datos. Caso contrario, aparecerá un mensaje de error

Outputs:

* Mensaje de éxito/error

Error Handling:

* Si el id del país no existe, se lanzará el siguiente mensaje “No existe el id ingresado. Por favor intente con otro”.
* Si no se completan todos los campos, se lanzará el siguiente mensaje “Completar todos los campos”

**Requerimiento funcional #24 Listado de países**

Descripción: A través del endpoint /paises/ se listarán todos los países de la base de datos. Los datos serán listados en formato JSON. Solo serán listados aquellos países con el estado en “true”.

Inputs (datos de entrada): No hay datos de entrada

Proceso:

* El sistema traerá todos los países con estado en true
* Se listarán todos los países que trajo el sistema en formato JSON

Outputs:

* ID\_Pa
* Nombre\_Pa
* Estado\_Pa

Error Handling:

**Requerimiento funcional #25 Buscado de país**

Descripción: A través del endpoint /paises/{id}, se podrá buscar un país en específico. Este será listado en formato JSON y podrá tener el estado en “false”.

Inputs (datos de entrada):

* ID\_Pa

Proceso:

* El sistema verifica que el ID exista en la base de datos
* Se lista el país

Outputs:

* ID\_Pa
* Nombre\_Pa
* Estado\_Pa

Error Handling:

* Si no existe el ID, se traerá un JSON vacío

**Requerimiento funcional #26 Alta de ciudad**

Descripción: A través del endpoint /ciudades/alta, se podrá dar de alta una ciudad.

Inputs (datos de entrada):

* Nombre\_Ciu
* ID\_Pa\_Ciu

Proceso:

* El sistema verifica que los campos no estén vacíos
* El sistema consume el microservicio de pais-service y comprueba que exista el ID de país ingresado
* Si todo está correcto, se asignará el valor del campo Estado\_Ciu a “true” y se le asignará un ID con la estrategia de generación de IDENTITY. Luego de esto, se carga en la base de datos

Outputs:

* Mensaje de éxito/error

Error Handling:

* Si hay campos vacíos, se devolverá un mensaje de “Completar todos los campos”
* Si el ID del país no existe, se devolverá un mensaje de “El id del país ingresado no existe”
* Si el sistema no se puede conectar a algún microservicio, lanzará un mensaje de “No se pudo conectar al microservicio [nombreMicroservicio]” gracias al patrón CircuitBreaker

**Requerimiento funcional #27 Baja lógica de ciudad**

Descripción: A través del endpoint /ciudades/baja/{id}, se podrá dar de baja una ciudad de forma lógica (se cambia el campo Estado\_Ciu a false).

Inputs (datos de entrada):

* ID\_Ciu

Proceso:

* El sistema corrobora que el id ingresado exista
* Si el id existe, se cambia el estado de la ciudad a false. Caso contrario, saldrá un mensaje de error

Outputs:

* Mensaje de éxito/error

Error Handling:

* Si el id no existe, se lanzará el mensaje “El id ingresado no existe, por favor ingresar uno nuevo”

**Requerimiento funcional #28 Modificación de ciudad**

Descripción: A través del endpoint /ciudades/modificacion/{id} se podrán modificar los datos de una ciudad ya ingresada

Inputs (datos de entrada):

* ID\_Ciud
* Nombre\_Ciu
* ID\_Pa\_Ciu

Proceso:

* El sistema verificará que el id ingresado en el endpoint exista
* El sistema verificará que los campos a modificar no estén vacíos
* El sistema consume el microservicio de pais-service y comprueba que exista el ID de país ingresado
* Si los datos son correctos, se modificará la ciudad en la base de datos. Caso contrario, aparecerá un mensaje de error

Outputs:

* Mensaje de éxito/error

Error Handling:

* Si el id de la ciudad no existe, se lanzará el siguiente mensaje “No existe el id ingresado. Por favor intente con otro”.
* Si no se completan todos los campos, se lanzará el siguiente mensaje “Completar todos los campos”
* Si el id del país no existe, se lanzará el mensaje “No existe el país ingresado”
* Si el sistema no se puede conectar a algún microservicio, lanzará un mensaje de “No se pudo conectar al microservicio [nombreMicroservicio]” gracias al patrón CircuitBreaker

**Requerimiento funcional #29 Listado de ciudades**

Descripción: A través del endpoint /ciudades/ se listarán todas las ciudades de la base de datos. Los datos serán listados en formato JSON. Solo serán listados aquellas ciudades con el estado en “true”.

Inputs (datos de entrada): No hay datos de entrada

Proceso:

* El sistema traerá todas las ciudades con estado en true
* Se listarán todas las ciudades que trajo el sistema en formato JSON

Outputs:

* ID\_Ciu
* Nombre\_Ciu
* ID\_Pa\_Ciu
* Estado\_Ciu

Error Handling:

* Si el sistema no se puede conectar a algún microservicio, mostrará un List con un elemento Ciudad de tipo Error (con los id en 0 y los demás campos con valor “Error”) gracias al patrón CircuitBreaker

**Requerimiento funcional #30 Buscado de ciudad**

Descripción: A través del endpoint /ciudades/{id}, se podrá buscar una ciudad en específico. Este será listado en formato JSON y podrá tener el estado en “false”.

Inputs (datos de entrada):

* ID\_Ciu

Proceso:

* El sistema verifica que el ID exista en la base de datos
* Se lista la ciudad

Outputs:

* ID\_Ciu
* Nombre\_Ciu
* ID\_Pa\_Ciu
* Estado\_Ciu

Error Handling:

* Si no existe el ID, se traerá un JSON vacío
* Si el sistema no se puede conectar a algún microservicio, mostrará un objeto Ciudad de tipo Error (con los id en 0 y los demás campos con valor “Error”) gracias al patrón CircuitBreaker

**Requerimiento funcional #31 Alta de Vuelo**

Descripción: A través del endpoint /vuelos/alta, se podrá dar de alta un vuelo.

Inputs (datos de entrada):

* ID\_Av\_Vu
* Destino\_Vu
* Origen\_Vu
* Precio\_Vu
* Fecha\_Compra\_Vu
* Hora\_Vu

Proceso:

* El sistema verifica que los campos no estén vacíos
* El sistema consume los microservicios de avion-service y ciudad-service y comprueba que existan los ID ingresados (ID\_Av\_Vu, Destino\_Vu y Origen\_Vu)
* Si todo está correcto, se asignará el valor del campo Estado\_Vu a “true”, el campo de Capacidad\_Disponible\_Vu igual que el campo Capacidad\_Total\_Av del avión seleccionado y se le asignará un ID con la estrategia de generación de IDENTITY.
* Se carga en la base de datos

Outputs:

* Mensaje de éxito/error

Error Handling:

* Si hay campos vacíos, se devolverá un mensaje de “Completar todos los campos”
* Si el ID del avión no existe, se devolverá un mensaje de “El id del avión ingresado no existe”
* Si el ID de la ciudad de destino no existe, se devolverá un mensaje de “El id de la ciudad de destino ingresada no existe”
* Si el ID de la ciudad de origen no existe, se devolverá un mensaje de “El id de la ciudad de origen ingresada no existe”
* Si el sistema no se puede conectar a algún microservicio, lanzará un mensaje de “No se pudo conectar al microservicio [nombreMicroservicio]” gracias al patrón CircuitBreaker

**Requerimiento funcional #32 Baja lógica de Vuelo**

Descripción: A través del endpoint /vuelos/baja/{id}, se podrá dar de baja una ciudad de forma lógica (se cambia el campo Estado\_Vu a false).

Inputs (datos de entrada):

* ID\_Vu

Proceso:

* El sistema corrobora que el id ingresado exista
* Si el id existe, se cambia el estado del vuelo a false. Caso contrario, saldrá un mensaje de error

Outputs:

* Mensaje de éxito/error

Error Handling:

* Si el id no existe, se lanzará el mensaje “El id ingresado no existe, por favor ingresar uno nuevo”

**Requerimiento funcional #33 Modificación de vuelo**

Descripción: A través del endpoint /vuelos/modificacion/{id} se podrán modificar los datos de una ciudad ya ingresada

Inputs (datos de entrada):

* ID\_Vu
* ID\_Av\_Vu
* Destino\_Vu
* Origen\_Vu
* Precio\_Vu
* Capacidad\_Disponible\_Vu
* Fecha\_Compra\_Vu
* Hora\_Vu

Proceso:

* El sistema verificará que el id ingresado en el endpoint exista
* El sistema verificará que los campos a modificar no estén vacíos
* El sistema consume los microservicios de ciudad-service y avion-service y comprueba que exista el ID de la ciudad de origen, la ciudad de destino y el avión
* Si los datos son correctos, se modificará la ciudad en la base de datos. Caso contrario, aparecerá un mensaje de error

Outputs:

* Mensaje de éxito/error

Error Handling:

* Si hay campos vacíos, se devolverá un mensaje de “Completar todos los campos”
* Si el ID del avión no existe, se devolverá un mensaje de “El id del avión ingresado no existe”
* Si el ID de la ciudad de destino no existe, se devolverá un mensaje de “El id de la ciudad de destino ingresada no existe”
* Si el ID de la ciudad de origen no existe, se devolverá un mensaje de “El id de la ciudad de origen ingresada no existe”
* Si el sistema no se puede conectar a algún microservicio, lanzará un mensaje de “No se pudo conectar al microservicio [nombreMicroservicio]” gracias al patrón CircuitBreaker

**Requerimiento funcional #34 Listado de Vuelos**

Descripción: A través del endpoint /vuelos/ se listarán todos los vuelos de la base de datos. Los datos serán listados en formato JSON. Solo serán listados aquellos vuelos con el estado en “true”.

Inputs (datos de entrada): No hay datos de entrada

Proceso:

* El sistema traerá todos los vuelos con estado en true
* Se listarán todos los vuelos que trajo el sistema en formato JSON

Outputs:

* ID\_Vu
* ID\_Av\_Vu
* Destino\_Vu
* Origen\_Vu
* Precio\_Vu
* Capacidad\_Disponible\_Vu
* Fecha\_Compra\_Vu
* Hora\_Vu
* Estado\_Vu

Error Handling:

* Si el sistema no se puede conectar a algún microservicio, mostrará un List con un elemento Vuelo de tipo Error (con los id en 0 y los demás campos con valor “Error”) gracias al patrón CircuitBreaker

**Requerimiento funcional #35 Buscado de Vuelo**

Descripción: A través del endpoint /vuelos/{id}, se podrá buscar un vuelo en específico. Este será listado en formato JSON y podrá tener el estado en “false”.

Inputs (datos de entrada):

* ID\_Vu

Proceso:

* El sistema verifica que el ID exista en la base de datos
* Se lista el vuelo

Outputs:

* ID\_Vu
* ID\_Av\_Vu
* Destino\_Vu
* Origen\_Vu
* Precio\_Vu
* Capacidad\_Disponible\_Vu
* Fecha\_Compra\_Vu
* Hora\_Vu
* Estado\_Vu

Error Handling:

* Si no existe el ID, se traerá un JSON vacío
* Si el sistema no se puede conectar a algún microservicio, mostrará un objeto Vuelo de tipo Error (con los id en 0 y los demás campos con valor “Error”) gracias al patrón CircuitBreaker

**Requerimiento funcional #36 Comprar Pasaje**

Descripción: A través del endpoint /pasajes/comprar, el cliente va a ser capaz de comprar un pasaje y, de esta forma, se haga la alta en la base de datos con los datos ingresados

Inputs (datos de entrada):

* ID\_Vu\_Pa
* ID\_Cli\_Pa
* Fecha\_Compra\_Pa

Proceso:

* El sistema verifica que los campos no estén vacíos
* El sistema consume los microservicios de vuelo-service y cliente-service. Comprueba que existan los ID ingresados (ID\_Vu\_Pa y ID\_Cli\_Pa) y que el campo de Cantidad\_Disponible\_Vu de la tabla Vuelo sea mayor a 0
* Si todo está correcto, se asignará el valor del campo Estado\_Pa a “true” y el campo de “Capacidad\_Disponible\_Vu” de la tabla de Vuelo se reducirá en uno. Se le asignará un ID con la estrategia de generación de IDENTITY.
* Se envía un mail al cliente con la fecha del vuelo, la hora, el precio, el país de origen y destino, y la ciudad de origen y destino
* Se carga en la base de datos

Outputs:

* Mensaje de éxito/error

Error Handling:

* Si hay campos vacíos, se devolverá un mensaje de “Completar todos los campos”
* Si el ID del vuelo no existe, se devolverá un mensaje de “El id del vuelo ingresado no existe”
* Si el ID del cliente no existe, se devolverá un mensaje de “El id del cliente ingresado no existe”
* Si el campo de Capacidad\_Disponible\_Vu de la tabla Vuelo no es mayor a 0, aparecerá un mensaje de “No hay más pasajes para este vuelo”
* Si el sistema no se puede conectar a algún microservicio, lanzará un mensaje de “No se pudo conectar al microservicio [nombreMicroservicio]” gracias al patrón CircuitBreaker

**Requerimiento funcional #37 Baja lógica de pasaje**

Descripción: A través del endpoint /pasajes/baja/{id}, el cliente va a ser capaz de dar de baja un pasaje que ha comprado de forma lógica (se cambia el estado de true a false en la base de datos).

Inputs (datos de entrada):

* ID\_Pa

Proceso:

* El sistema corrobora que el id ingresado exista
* El sistema corrobora que el microservicio de vuelo-service esté disponible
* Se cambia el estado del pasaje a false
* Se aumenta el valor de Capacidad\_Disponible\_Vu de la tabla Vuelo en uno

Outputs:

* Mensaje de éxito/error

Error Handling:

* Si el id no existe, se lanzará el mensaje “El id ingresado no existe, por favor ingresar uno nuevo”
* Si no se puede acceder al microservicio de vuelo-service, se lanzará el mensaje de “No se pudo acceder al microservicio de vuelo-service” a través del patrón CircuitBreaker

**Requerimiento funcional #38 Listado de todos los Pasajes**

Descripción: A través del endpoint /pasajes/ se listarán todos los pasajes con la información relevante de estos. Además de la información del pasaje, también traerá el nombre del país y ciudad de destino y el nombre del cliente. Solo serán listados aquellos pasajes con el estado en “true”.

Inputs (datos de entrada): No hay datos de entrada

Proceso:

* El sistema intentará consumir los servicios de vuelo-service y cliente-service
* Se listarán todos los pasajes que trajo el sistema, además de la información extra que necesita de los servicios consumidos

Outputs:

* ID\_Pa
* ID\_Vu\_Pa
* Nombre\_Pa\_Pa
* Nombre\_Ciu\_Pa
* Nombre\_Cli\_Pa
* Fecha\_Compra\_Pa

Error Handling:

* Si el sistema no se puede conectar a algún microservicio, mostrará un List con un elemento PasajeDTO de tipo Error (con los id en 0 y los demás campos con valor “Error”) gracias al patrón CircuitBreaker

**Requerimiento funcional #39 Buscado de Pasaje**

Descripción: A través del endpoint /pasajes/{id}, se podrá buscar un pasaje en específico. Este será listado en formato JSON y podrá tener el estado en “false”.

Inputs (datos de entrada):

* ID\_Pa

Proceso:

* El sistema intentará consumir los servicios de vuelo-service y cliente-service
* El sistema verifica que el ID exista en la base de datos
* Se lista el pasaje

Outputs:

* ID\_Pa
* ID\_Vu\_Pa
* Nombre\_Pa\_Pa
* Nombre\_Ciu\_Pa
* Nombre\_Cli\_Pa
* Fecha\_Compra\_Pa

Error Handling:

* Si no existe el ID, se traerá un JSON vacío
* Si el sistema no se puede conectar a algún microservicio, mostrará un objeto PasajeDTO de tipo Error (con los id en 0 y los demás campos con valor “Error”) gracias al patrón CircuitBreaker

**Requerimiento funcional #40 Listado de todos los Pasajes de un Cliente**

Descripción: A través del endpoint /pasajes/cliente/{id} se listarán todos los pasajes comprados por un cliente específico con la información relevante de estos. Además de la información del pasaje, también traerá el nombre del país y ciudad de destino y el nombre del cliente. Solo serán listados aquellos pasajes con el estado en “true”.

Inputs (datos de entrada): ID\_Cli\_Pa

Proceso:

* El sistema intentará consumir los servicios de vuelo-service y cliente-service
* El sistema verificará que exista el ID del cliente ingresado
* Se listarán todos los pasajes del cliente ingresado, además de la información extra que necesita de los servicios consumidos

Outputs:

* ID\_Pa
* ID\_Vu\_Pa
* Nombre\_Pa\_Pa
* Nombre\_Ciu\_Pa
* Nombre\_Cli\_Pa
* Fecha\_Compra\_Pa

Error Handling:

* Si el sistema no se puede conectar a algún microservicio, mostrará un List con un elemento PasajeDTO de tipo Error (con los id en 0 y los demás campos con valor “Error”) gracias al patrón CircuitBreaker

**Requerimiento funcional #41 Listado de Pasajes de un vuelo**

Descripción: A través del endpoint /pasajes/vuelo/{id}, se podrán buscar los pasajes de un vuelo en específico. Estos serán listados en formato JSON.

Inputs (datos de entrada):

* ID\_Vu\_Pa

Proceso:

* El sistema intentará consumir los servicios de vuelo-service y cliente-service
* El sistema verifica que el ID del vuelo exista en la base de datos
* Se listan los pasajes

Outputs:

* ID\_Pa
* ID\_Vu\_Pa
* Nombre\_Pa\_Pa
* Nombre\_Ciu\_Pa
* Nombre\_Cli\_Pa
* Fecha\_Compra\_Pa

Error Handling:

* Si no existe el ID, se traerá un JSON vacío
* Si el sistema no se puede conectar a algún microservicio, mostrará un objeto PasajeDTO de tipo Error (con los id en 0 y los demás campos con valor “Error”) gracias al patrón CircuitBreaker

**Requerimiento funcional #42 Generar reporte**

Descripción: A través del endpoint /reportes/{fechaInicio}/{fechaFinal}, el administrador podrá descargar un Excel en el cual estarán plasmados todos los pasajes comprados desde la fecha inicio hasta la fecha final

Inputs (datos de entrada):

* Fecha inicio
* Fecha final

Proceso:

* El sistema intentará consumir los servicios de vuelo-service y cliente-service
* Se insertarán todos los pasajes que trajo el sistema en un excel, además de la información extra que necesita de los servicios consumidos.
* Se descarga el Excel en la pc del administrador con el nombre “Reportes compra de pasajes [fechaInicio] – [fechaFinal]”

Outputs:

* Excel con los siguientes datos
  + ID\_Pa
  + ID\_Vu\_Pa
  + Nombre\_Pa\_Pa
  + Nombre\_Ciu\_Pa
  + Nombre\_Cli\_Pa
  + Fecha\_Compra\_Pa

Error Handling:

* Si el sistema no se puede conectar a algún microservicio, lanzará una excepción

**Requerimientos no funcionales**

1. Rendimiento

* El sistema debe ser capaz de manejar al menos 20 solicitudes concurrentes de forma eficiente, dado que es un proyecto personal y no estará en un entorno de producción.
* Las respuestas a las solicitudes deben ser procesadas en menos de 2 segundos bajo una carga ligera de usuarios.

2. Escalabilidad

* El sistema debe permitir escalar a nivel local en caso de ser necesario, pero no está orientado a entornos de nube ni a un gran volumen de usuarios.
* Los microservicios deben estar organizados para permitir una fácil expansión futura en caso de añadir nuevas funcionalidades.

3. Mantenibilidad

* El código debe ser modular y seguir principios de buenas prácticas en cuanto a organización y legibilidad, facilitando futuras actualizaciones.
* Se debe garantizar que los módulos de los microservicios sean independientes, de modo que puedas modificar uno sin afectar a los demás.

4. Compatibilidad

* El sistema debe ser compatible con diferentes entornos de desarrollo local, como Windows (que es tu sistema operativo actual) y sistemas basados en Docker para contenerizar la base de datos MySQL y los servicios de la API.
* Se debe garantizar que la API siga los principios básicos de REST para facilitar futuras integraciones o consultas desde herramientas como Postman.

5. Usabilidad

* La API debe estar bien documentada para que puedas realizar pruebas e integraciones fácilmente.
* Las operaciones deben ser simples y fáciles de probar, priorizando la facilidad de uso en tu entorno local.

**Restricciones**

1. Dependencias Tecnológicas

* El sistema utilizará Java y Spring Boot como base para los microservicios.
* Spring Cloud se utilizará para la gestión de la arquitectura de microservicios.
* La base de datos será MySQL, y se manejará mediante Docker en un entorno local.

2. Plazos y Cronograma

* El proyecto debe completarse en alrededor de 200 horas de desarrollo, distribuidas en un período entre 4 y 5 semanas.

3. Recursos de Infraestructura

* Se debe usar Docker para contenerizar la base de datos, asegurando que el entorno de desarrollo sea consistente.

4. Recursos Humanos

* Al ser un proyecto personal, el único desarrollador responsable del diseño, desarrollo y mantenimiento del sistema será Leonardo Gil.