Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC2233 — Programación Avanzada 2025-1

Publicación: 29 de Mayo de 2025
Actividad evaluada

Actividad 8
Networking 2

Entrega

- **Lugar:** Repositorio personal de GitHub — Carpeta: `Actividades/AC8`
- **Fecha máxima de entrega:** 29 de Mayo 17:20
- **Ejecución de actividad**: La Actividad será ejecutada **únicamente** desde la terminal del computador. **Los *paths* relativos utilizados en la Actividad deben ser coherentes con esta instrucción, y no pueden modificarse.**

Introducción

Asombrado por el gran éxito de **DCCuentasPendientes**, te propones crear una aplicación que te permita organizar y registrar tus gastos. Para esto, decides utilizar tus conocimientos de *Webservices* y así empezar a implementar **DCConsumos**.

Flujo del programa

El programa consiste en dos *scripts* independientes:

- Un **servidor tipo API** que administra la información de un usuario, guardando su información en un archivo JSON.
- Un **cliente** que se comunica con el servidor mediante la realización de consultas.

Esta actividad consta en completar 2 clases utilizando los contenidos de *Networking* II. Estas dos clases implementan una arquitectura cliente-servidor, donde una clase modela el cliente, mientras que la otra corresponde al servidor. Ambas clases serán corregidas exclusivamente mediante el uso de *tests*.

Finalmente, debes asegurarte de entregar -como mínimo- los archivos que tenga el *tag* de `Entregar` en la siguiente sección, asegurando que se mantenga la estructura de directorios original de la actividad. Los demás archivos no es necesario subir, pero tampoco se penalizará si se suben al repositorio personal.

Archivos y entidades

En el directorio de la actividad encontrarás los siguientes archivos:

- Entregar Modificar `cliente/main.py`: Archivo principal a ejecutar del cliente. Contiene la definición de la clases `ClienteApi`, que presenta los siguientes atributos:
  - `self.host` *String* correspondiente a la dirección IP del servidor.
  - `self.port` *Integer* correspondiente al puerto del servidor.

  Los métodos a completar serán explicados con mayor detalle en la Parte II.
- Entregar Modificar `servidor/main.py`: Archivo principal a ejecutar del servidor. En este archivo no se definen clases, pero se llaman las siguientes clases:
  - `app = Flask()` Instancia de `Flask` que permite implementar una API.
  - `dcconsumo = DCConsumo()` Clase personalizada que permite administrar los gastos e ingresos de un usuario.

  Este archivo cuenta con múltiples funciones a completar, estas serán explicadas con mayor detalle en la Parte I.
- No modificar `servidor/dcconsumo.py`: Archivo que contienen la clase `DCConsumo`. Esta clase permite manejar las distintas cuentas bancarias de un usuario, permitiéndole: obtener saldos, obtener transacciones, agregar gastos, ajustar saldos, y hacer transferencias.

  Esta clase presenta el atributo `self.data`, el cual corresponde a un diccionario que almacena las distintas cuentas bancarias del usuario, para esto utiliza la clase `Banco`.

  Los métodos de esta clase no serán explicados en el enunciado, pero cuentan con un *docstring* que permite comprender para qué sirve cada método.
- No modificar `servidor/banco.py`: Archivo que contiene las clases `Banco` y `Transaccion`, las cuales se utilizan únicamente para almacenar información:

  La clase `Banco` presenta los siguientes atributos:
  - `self.nombre_banco` *String* que permite identificar al banco.
  - `self.n_cuenta` *Integer* que indica el número de la cuenta bancaria.
  - `self.saldo` *Integer* que indica el saldo actual de la cuenta.
  - `self.transacciones` Lista de instancia `Transacciones`.

  Mientras que la clase `Transacción` presenta estos atributos:
  - `self.titulo` *String* que permite identificar la transacción.
  - `self.monto` *Integer* que indica el monto de dinero asociado a la transacción.
  - `self.tipo` *String* que indica el tipo de transacción: `'ingreso'` o `'gasto'`.

  Adicionalmente, presenta la función `BancoDecoder` y la clase `BancoEncoder` las cuales se utilizan para manejar la información del archivo `data.json`.
- No modificar `servidor/data.json`: Archivo que almacena la información bancaria y transacciones del usuario.

Parte I. API

Antes de poder empezar a trabajar en el cliente, debes asegurar que la API funcione correctamente. Para esto, deberás utilizar tus conocimientos de *Webservices Server-side* y completar las siguientes funciones.

Recomendaciones:

1. Revisar la implementación de la función `hello_world()` y el *end-point* `"/"`, ya que esto mostrará como asociar las funciones a *end-points* y como generar las respuestas a las consultas.
2. Antes de completar las funciones, revisa el código de los métodos asociados de `DCConsumo`. Te pueden ayudar a entender el flujo del método y evaluar los posibles errores que saldrán durante la ejecución del programa.
3. Para manejar los distintos casos bordes, planteáte si utilizar un enfoque *"Look Before You Leap"* (uso de `if`/`else`) o *"Easier to Ask Forgiveness than Permission"* (uso de `try`/`except`). En caso de utilices el último enfoque, para obtener el mensaje de error asociado a una excepción utiliza el atributo `args` de una excepción.

Las funciones a completar son las siguientes:

- Modificar `def obtener_saldo_total() -> Response:`

  Función asociada al *end-point* `"/obtener_saldo"` y el método `GET`. Indica el salto total de todas las cuentas bancarias del usuario. Para esto utiliza el método `obtener_saldo()` de la clase `DCConsumo`.

  Retorna una instancia de `Response` con la siguiente información:

Caso	*Status Code*	Diccionario
El saldo se obtiene correctamente.	200	`{'result': int_saldo}`

- Modificar `def obtener_saldo_banco(banco: str) -> Response:`

  Función asociada al *end-point* dinámico `"/obtener_saldo/{banco}"` y el método `GET`. Indica el salto del banco entregado. Para esto utiliza el método `obtener_saldo()` de la clase `DCConsumo`.

  Dependiendo del comportamiento de la función se retorna una instancia de `Response` con la siguiente información:

Caso	*Status Code*	Diccionario
Banco indicado no forma parte de los bancos del usuario.	404	`{'error': 'Cuenta {banco} no encontrada'}`
El saldo se obtiene correctamente.	200	`{'result': int_saldo}`

- Modificar `def obtener_transacciones(banco: str) -> Response:`

  Función asociada al *end-point* dinámico `"/obtener_transacciones/{banco}"` y el método `GET`. Obtiene las últimas transacciones asociadas a la cuenta bancaria indicada. Para esto utiliza el método `obtener_transacciones()` de la clase `DCConsumo`.

  Además, este *end-point* cuenta con el parámetro opcional `cantidad`, el cual indica la cantidad de transacciones que se quieren obtener.

  Dependiendo del comportamiento de la función se retorna una instancia de `Response` con la siguiente información:

Caso	***Status Code***	**Diccionario**
Banco indicado no forma parte de los bancos del usuario.	404	`{'error': 'Cuenta {banco} no encontrada'}`
Las transacciones se obtienen correctamente.	200	`{'result': lista_transacciones}`

- Modificar `def agregar_gasto(banco: str) -> Response:`

  Función asociada al *end-point* dinámico `"/agregar_gasto/{banco}"` y el método `POST`. Agrega un gasto a la cuenta bancaria indicada y actualiza el saldo actual de la cuenta. Para esto utiliza el método `agregar_gasto()` de la clase `DCConsumo`.

  Además, este *end-point* utiliza los parámetros `titulo` y `monto`, para obtener la información necesaria para crear el gasto.

  Dependiendo del comportamiento de la función se retorna una instancia de `Response` con la siguiente información:

Caso	***Status Code***	**Diccionario**
Banco indicado no forma parte de los bancos del usuario	404	`{'error': 'Cuenta {banco} no encontrada'}`
Falta alguno de los parámetros.	400	`{'error': 'Faltan argumentos'}`
El gasto se agrega correctamente.	200	`{'msg': 'Gasto agregado'}`

- **Modificar** `def ajustar_saldo(banco: str) -> Response:`

  Función asociada al *end-point* dinámico `"/ajustar_saldo/{banco}"` y el método `PATCH`. Modifica el saldo de la cuenta bancaria indicada. Para esto utiliza el método `ajustar_saldo()` de la clase `DCConsumo`.

  Además, este *end-point* utiliza el parámetro `saldo`, el cual indica el nuevo saldo de la cuenta.

  Dependiendo del comportamiento de la función se retorna una instancia de `Response` con la siguiente información:

Caso	***Status Code***	**Diccionario**
Banco indicado no forma parte de los bancos del usuario.	404	`{'error': 'Cuenta {banco} no encontrada'}`
Falta el parámetro `saldo`.	400	`{'error': 'Faltan argumentos'}`
El saldo se ajusta correctamente.	200	`{'msg': 'Saldo ajustado'}`

- **Modificar** `def hacer_transferencia() -> Response:`

  Función asociada al *end-point* `"/hacer_transferencia/"` y el método `POST`. Realiza una transferencia por medio del método `hacer_transferencia()` de la clase `DCConsumo`.

  Recibe la cuenta bancaria desde la que se realiza la transferencia, la información del destinatario y el monto a partir del *body* de la *request*. Además, utiliza los *headers* de la *request* para recibir un *token* de autentificación. Recuerda que los *tokens* se asocian a la llave `"Authorization"`.

  Dependiendo del comportamiento de la función se retorna una instancia de `Response` con la siguiente información:

Caso	***Status Code***	**Diccionario**
Falta *token* de autentificación.	401	`{'error': 'Token invalido'}`
Banco indicado no forma parte de los bancos del usuario.	404	`{'error': 'Cuenta {banco} no encontrada'}`
Falta información en el *body* de la *request*.	400	`{'error': 'Faltan argumentos'}`
Monto indicado supera el saldo de la cuenta.	400	`{'error': 'Transferencia supera el saldo de la cuenta'}`
Transferencia realizada exitosamente.	200	`{'msg': 'Transferencia realizada'}`

Parte II. Cliente

Ahora que tenemos una API funcional, podemos centrarnos en la comunicación con el cliente. Para esto, deberás utilizar tus conocimientos de *webservices* y completar los siguientes métodos.

- Modificar `def obtener_saldo(self, banco: str | None = None) -> Any:`

  Pude recibir el nombre de una cuenta bancaria y se encarga de obtener el saldo del usuario.

  Realiza una *request* al *end-point* dinámico `"/obtener_saldo"`. Si recibe un banco, obtiene el saldo de la cuenta indicada; mientras que si no se recibe un banco, obtiene el salgo de todas las cuentas bancarias.

  Si la *request* termina exitosamente, retorna el saldo obtenido de la *request*. En caso contrario, retornar el error recibido.

- Modificar `def obtener_transacciones(self, banco: str, cantidad: int | None = None) -> Any:`

  Recibe el nombre de una cuenta bancaria y puede recibir una cantidad. Se encarga de obtener las últimas transacciones realizadas en la cuenta indicada.

  Realiza una *request* al *end-point* dinámico `"/obtener_transacciones"` y, en caso de que corresponda, pide la cantidad de transacciones indicadas en el *input*.

  Si la *request* termina exitosamente, retorna la lista de transacciones obtenida de la *request*. En caso contrario, retornar el error recibido.

- Modificar `def agregar_gasto(self, banco: str, titulo: str, monto: int) -> Any:`

  Recibe el nombre de una cuenta bancaria, el titulo de un gasto y el monto asociado. Agrega el gasto al listado de transacciones del banco y modifica el saldo actual del mismo.

  Realiza una *request* al *end-point* dinámico `"/agregar_gasto"`. Si la *request* termina exitosamente, retorna el mensaje recibido. En caso contrario, retornar el error recibido.

- Modificar `def ajustar_saldo(self, banco: str, saldo: int) -> Any:`

  Recibe el nombre de una cuenta bancaria y un saldo. Actualiza el saldo de la cuenta recibida.

  Realiza una *request* al *end-point* dinámico `"/ajustar_saldo"`. Si la *request* termina exitosamente, retorna el mensaje recibido. En caso contrario, retornar el error recibido.

- Modificar `def hacer_transferencia(self, banco: str, dest_nombre: str, dest_cuenta: int, monto: int, token: str) -> Any:`

  Recibe el nombre de una cuenta bancaria, el nombre y cuenta de un destinatario, el monto de la transferencia y un *token*. Realiza una transferencias al destinatario.

  Realiza una *request* al *end-point* `"/hacer_transferencia"`. El *token* debe entregarse por medio de los *headers*, y la información de la transferencia debe entregarse en el *body* en el siguiente formato:

```
{
    'banco': banco,
    'destinatario': {
        'nombre': dest_nombre,
        'n_cuenta': dest_cuenta
    },
    'monto': monto
}
```

  Importante: Recomendamos transformar el diccionario a un *string* mediante el uso de JSON, para asegurar que el *body* se envíe correctamente.

  Si la *request* termina exitosamente, retorna el mensaje recibido. En caso contrario, retornar el error recibido.

Notas

- **No puedes hacer *import* de otras librerías externas a las entregadas en el archivo.**
- Recuerda que la ubicación de tu entrega es en **tu repositorio de Git**. En la rama (*branch*) por defecto del repositorio: `main`.
- Se recomienda completar la actividad en el orden del enunciado.
- Recuerda que esta evaluación presenta corrección **automatizada**. Si entregas un código que se cae al momento de correr los *tests*, será evaluado con 0 puntos.
- Si aparece un error inesperado, ¡léelo y revisa el código del *test*! Intenta interpretarlo y/o buscarlo en Google.
- Se recomienda probar tu código con los *tests* y ejecutando `main.py`, este último se ofrece un pequeño código donde se prueba la carrera con 3 jugadores.

Objetivo de la actividad

- Aplicar conceptos de la arquitectura cliente-servidor.
- Utilizar *webservices* para comunicar un cliente y servidor.
- Utilizar `Flask` para implementar una API.
- Probar código mediante la ejecución de *test* y de los archivos `main.py`.

Ejecución de código

Por lo general -en este curso- cuando trabajamos en una arquitectura cliente-servidor, se separamos el código del cliente y servidor en carpetas independientes, para así simular computadores independientes. Por lo que, para ejecutar el código, se ubica la terminal en cada una de las carpetas y se ejecuta el código.

Dado que en el caso de esta Actividad utilizamos tests para ejecutar el código, para ejecutar el código del servidor y el cliente debes ubicarte `Actividades/AC8` y ejecutar los siguientes comandos:

- `python3 servidor.py XXXX`
- `python3 cliente.py XXXX`

donde `XXXX` corresponde al puerto utilizado por el servidor.

Importante: recuerda que si `python3` no funciona, probar con el comando específico de tu computador. Este puede ser `py`, `python`, `py3` o `python3.11`.

Ejecución de *tests*

En esta actividad se provee de varios archivos `.py` los cuáles contiene diferentes *tests* que ayudan a validar el desarrollo de la actividad. Para ejecutar estos *tests*, **primero debes posicionar tu terminal/consola en la carpeta de la actividad (`Actividades/AC8`)**. Luego, desde esta misma, debes escribir el siguiente comando para ejecutar todos los *tests* de la actividad:

- `python3 -m unittest discover tests_publicos -v -b`

En cambio, si deseas ejecutar un subconjunto de *tests*, puedes hacerlo si escribes lo siguiente en la terminal/consola:

- `python3 -m unittest -v -b tests_publicos.test_api`
  Para ejecutar solo el subconjunto de *tests* de la Parte I.
- `python3 -m unittest -v -b tests_publicos.test_cliente`
  Para ejecutar solo el subconjunto de *tests* de la Parte II.

Importante: recuerda que si `python3` no funciona, probar con el comando específico de tu computador. Este puede ser `py`, `python`, `py3` o `python3.11`.