UNIVERSIDAD
NACIONAL
DE COLOMBIA

Ciclo III

Desarrollo de Software

Capa Lógica: Entidades de Datos

8

Jeisson Andrés Vergara Vargas
Departamento de Ingeniería de Sistemas e Industrial
http://colswe.unal.edu.co/~javergarav/
javergarav@unal.edu.co

Objetivo de Aprendizaje

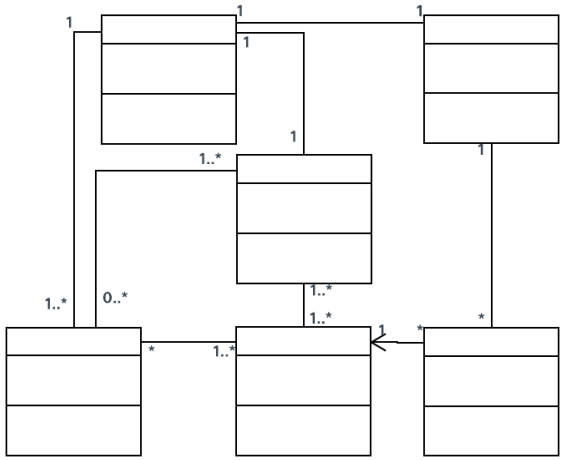
Reconocer los diferentes tipos de entidades de datos y sus utilidades, en el contexto de la capa lógica y el uso de FastAPI.

Introducción

Entidades de Datos

¿Qué es una **Entidad de Datos**?

Una **entidad de datos** es un **modelo** o **clase** que agrupa una serie de **información**, con el fin de representar un objeto o un estado de un objeto.



Tipos de Entidades de Datos

En la práctica, en la **capa lógica**, una **entidad de datos** puede **representar** los siguientes abstraciones:

- Un esquema de base de datos
- Datos de autenticación
- Entradas de una petición
- Respuesta de una petición

Campos en Entidades de Datos

Los campos en las entidades de datos se caracterizan por las siguientes propiedades:

- Nombre descriptivo
- Tipos de datos
- Requerido u opcional
- Valor por defecto

Entidades de Datos en FastAPI

¿Qué es FastAPI?

FastAPI es un **Framework** que facilita la creación **de API's REST** usando el lenguaje **Python**.

- Rápido y Versatil
- Documentación Automática
- Tipado de Datos

Definición de Entidades en FastAPI

Salvo las **entidades** que representan un **esquema de base de datos**, todos las demás **entidades de datos** en FastAPI se definen con ayuda de **2 librerías**:

Pydantic
(validación)

Typing
(tipado de datos)

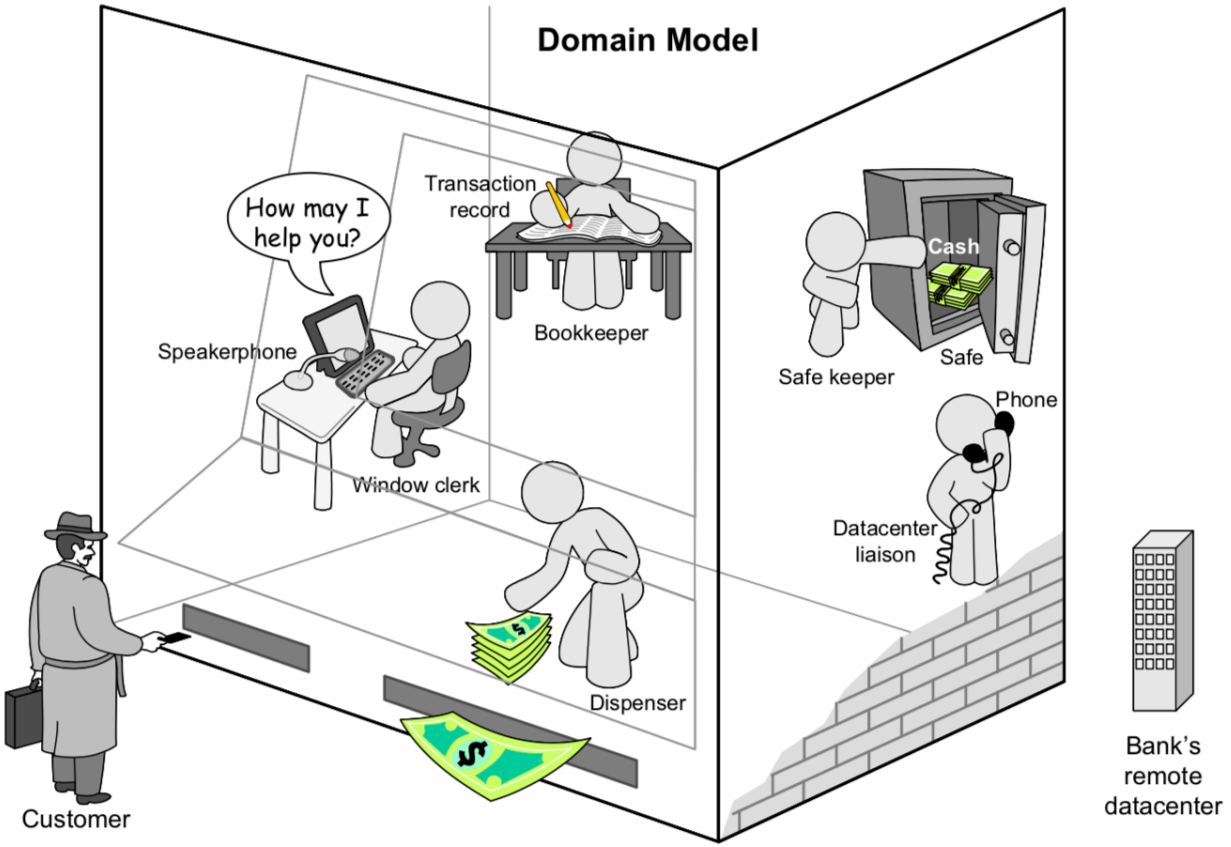
Definición de Entidades en FastAPI

Las **entidades** que representan un **esquema de base de datos** son definidas con la librería **SQL Alchemy**:

NOTA: las entidades se definirán con **pydantic** y **typing**, ya que se usará una base de datos ficticia.

Ejemplo

Software para un «ATM»



Entidades de Datos

Se definirán **diferentes modelos de datos** que serán usados en la implementación de las **operaciones** para la **API-REST** (esta sesión).

API-REST

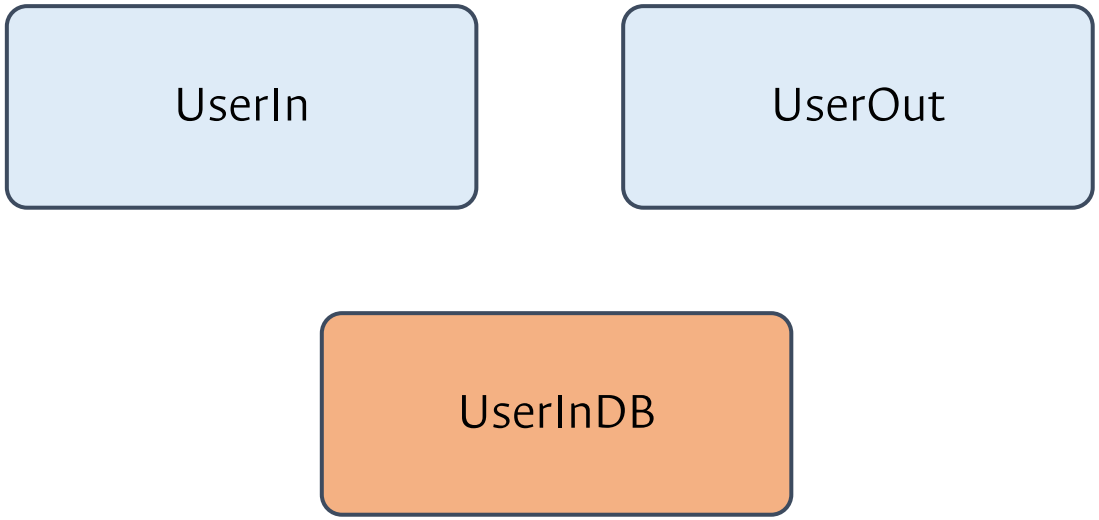
Usando los **mecanismos** que ofrece **FastAPI** y las **entidades de datos creadas**, se llevará a cabo la implementación de las operaciones para la **API-REST** (próxima sesión).

Objeto Usuario

El objeto **usuario** representa la cuenta que el **cliente** tiene en el banco. Los atributos correspondientes son:

Objeto Usuario

Las **entidades de datos** que representan los **diferentes estados** del objeto **usuario** son:



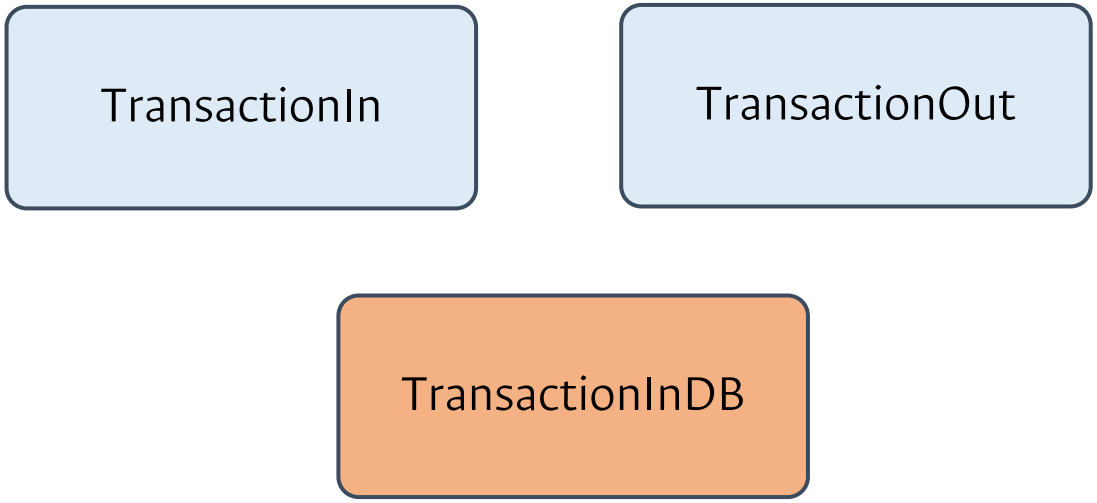
Objeto Transacción

El objeto **transacción** representa un **movimiento de dinero** asociado a un usuario. Los atributos correspondientes son:

- id_transaction
- username
- date
- value
- actual_balance

Objeto Transacción

Las **entidades de datos** que representan los **diferentes estados** del objeto **transacción** son:



Implementación

Instalación de Paquetes

Para la **ejecución** de la **API** se necesitan los siguientes **paquetes**:

- fastapi
- uvicorn
- pydantic

Instalación de Paqutes

Para la instalación de los **paquetes**, se deben ejecutar los siguientes **comandos** en consola:

pip install **fastapi**
pip install **uvicorn**
pip install **pydantic**

Para la **ejecución** de los **comandos** es necesario tener instalado **Python.**

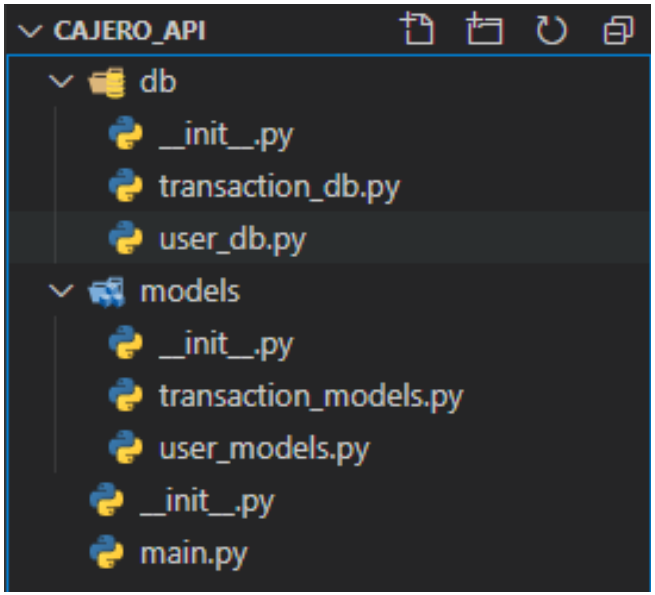
Estructura del Proyecto

Para la **estructura** del **proyecto** se crean los siguientes directorios y archivos:

- Se crea una carpeta principal llamada **cajero_api**
- Dentro **cajero_api** se crea un archivo llamado **main.py**
- Dentro **cajero_api** se crea una carpeta llamada **db**
  - Dentro de **db** se crean: **user_db.py** y **transaction_db.py**
- Dentro **cajero_api** se crea una carpeta llamada **models**
  - Dentro de **models** se crean: **user_models.py** y **transaction_models.py**
- De manera adicional, en cada uno de laos directorios se debe crear un archivo **__init__.py**, con el fin de que estos sean reconocidos como módulos.

Estructura del Proyecto

Al finalizar los pasos, la estructura del proyecto debe verse de la siguiente manera:



Revisión de la Implementación

La **implementación** para esta **sección** consistirá en implementar los siguientes elementos:

- user_models.py
- transaction_models.py
- user_db.py
- transaction_db.py

Implementación – user_db.py

Para la **implementación** de este elemento, se deben copiar los siguientes **3 bloques de código**:

Bloque 1: definición de UserInDB

```
from typing import  Dict
from pydantic import BaseModel

class UserInDB(BaseModel):
    username: str
    password: str
    balance: int
```

Implementación – user_db.py

Bloque 2: definición de la base de datos ficticia

```
database_users = Dict[str, UserInDB]

database_users = {
    "camilo24": UserInDB(**{"username":"camilo24",
                            "password":"root",
                            "balance":12000}),

    "andres18": UserInDB(**{"username":"andres18",
                            "password":"hola",
                            "balance":34000}),
}
```

Implementación – user_db.py

Bloque 3: definición de funciones sobre la base de datos fictica

```
def get_user(username: str):
    if username in database_users.keys():
        return database_users[username]
    else:
        return None

def update_user(user_in_db: UserInDB):
    database_users[user_in_db.username] = user_in_db
    return user_in_db
```

Implementación – transaction_db.py

Para la **implementación** de este elemento se deben copiar los siguientes **2 bloques de código**:

Bloque 1: definición de TransactionInDB

```
from datetime import datetime
from pydantic import BaseModel


class TransactionInDB(BaseModel):
    id_transaction: int = 0
    username: str
    date: datetime = datetime.now()
    value: int
    actual_balance: int
```

Implementación – transaction_db.py

Bloque 2: definición de la base de datos ficticia y una función

```
database_transactions = []
generator = {"id":0}


def save_transaction(transaction_in_db: TransactionInDB):
    generator["id"] = generator["id"] + 1
    transaction_in_db.id_transaction = generator["id"]
    database_transactions.append(transaction_in_db)
    return transaction_in_db
```

Implementación – user_models.py

Para la **implementación** de este elemento se debe copiar el siguiente **bloque de código**, que permitirá la definición de los **modelos de estado**:

```
from pydantic import BaseModel

class UserIn(BaseModel):
    username: str
    password: str

class UserOut(BaseModel):
    username: str
    balance: int
```

Implementación – transaction_models.py

Para la **implementación** de este elemento se deben copiar los siguientes **bloques de código**, que permitirán la definición de los **modelos de estado**:

Parte 1:

```
from pydantic import BaseModel
from datetime import datetime

class TransactionIn(BaseModel):
    username: str
    value: int
```

Implementación – transaction_models.py

Para la **implementación** de este elemento se deben copiar los siguientes **bloques de código**, que permitirán la definición de los **modelos de estado**:

Parte 2:

```
class TransactionOut(BaseModel):
    id_transaction: int
    username: str
    date: datetime
    value: int
    actual_balance: int
```

Referencias

- **[FASTAPI]** Comunidad FastAPI. (2020, noviembre). *FastAPI*. FastAPI. https://fastapi.tiangolo.com/