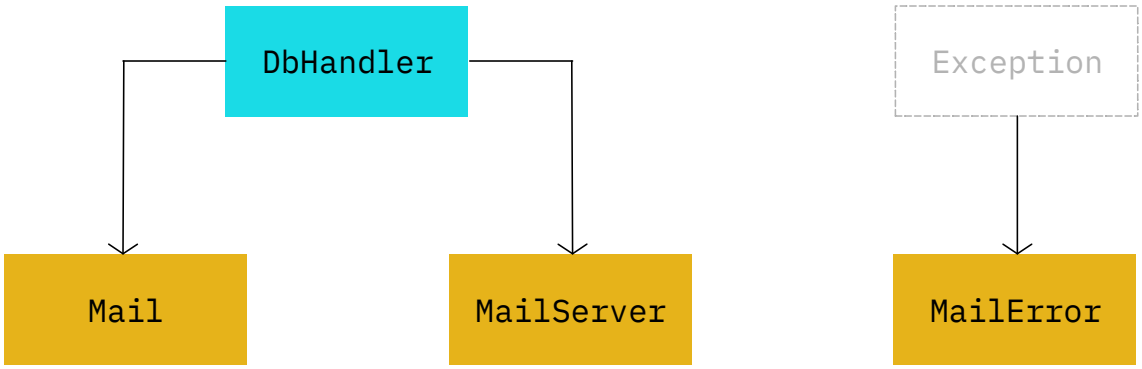
Mail

Este ejercicio gira en torno a la gestión de **correos electrónicos** desarrollando un programa en **Python** mediante el paradigma de **programación orientada a objetos** junto con **acceso a datos**.

La idea es simular el **envío** y **recepción** de *emails* a traves de una pequeña base de datos.

1. Diagrama de clases

A continuación se presentan las clases que deben ser implementadas (junto con su **herencia**):



2. Base de datos

A continuación se especifican las tablas que componen la base de datos y su estructura:

Tabla `login`

Representa las credenciales de acceso al servidor de correo.

Columna	Tipo	Descripción
`username`	Texto (PK)	Nombre de usuario
`password`	Texto	Contraseña
`domain`	Texto	Dominio de correo

Ejemplo de registro:
`|'guido'|'hh34y78'|'python.org'|`

Tabla `activity`

Representa la actividad de envío de correos.

Columna	Tipo	Descripción
`id`	Entero (PK)	Clave primaria
`sender`	Texto	Correo del remitente
`recipient`	Texto	Correo del destinatario
`subject`	Texto	Asunto
`body`	Texto	Cuerpo del correo

Ejemplo de registro:
`|3|'guido@python.org'|'pablo@python.org'|'Python 3.14'|'This is awesome'|`

3. Clase `DbHandler`

Representa una entidad abstracta que contiene funcionalidades para manejo de bases de datos y de la que heredan otras clases.

Implementa, *al menos*, los siguientes **métodos**:

```
def __init__(self, db_path: str = DB_PATH):
```

- Constructor de la clase.
- Crea el atributo `con` ⇒ conexión a la base de datos (especificando factoría `Row`), conectando a `db_path`.
- Crea el atributo `cur` ⇒ cursor a la base de datos.
- En este punto **no hay que almacenar nada en la base de datos**.

```
def create_db(self) -> None:
```

- Crea la base de datos y sus correspondientes tablas.
- **No cierres** la conexión a la base de datos.

4. Clase `Mail`

Representa un correo electrónico.

Implementa, *al menos*, los siguientes **métodos**:

```
def __init__(self, sender: str, recipient: str, subject: str, body: str):
```

- Constructor de la clase.
- Esta clase hereda de `DbHandler`...
- Crea los atributos homónimos a los parámetros.
- En este punto **no hay que almacenar nada en la base de datos**.

```
def send(self) -> None:
```

- Simula el envío del correo `self` mediante una operación en la base de datos.
- Inserta todos los campos del correo en la tabla `activity`.

```
def __str__(self):
```

- Representa un objeto de tipo `Mail` de la siguiente forma:

```
From: <remitente>
To: <destinatario>
---
<asunto pasado a mayúsculas>
<cuerpo del correo>
```

5. Clase `MailServer`

Representa un servidor de correo.

Implementa, *al menos*, los siguientes **métodos**:

```
def __init__(self, username: str):
```

- Constructor de la clase.
- Esta clase hereda de `DbHandler`...
- Crea el atributo `username` desde el argumento.
- Crea el atritubo `logged` (`bool`) que indique se el usuario se ha logeado en el servidor de correo.
- **No crees** aún el atributo `domain`. Se hará más adelante.
- En este punto **no hay que almacenar nada en la base de datos**.

```
def login(self, password: str): -> None
```

- Intenta hacer el login del usuario comprobando la contraseña que se pasa (mediante una consulta a la base de datos).
- Hay que **actualizar** los atributos `domain` y `logged` de la siguiente manera:
  - Si el usuario se logea correctamente, su dominio será el que está almacenado en la base de datos.
  - Si el usuario no se logea correctamente, su dominio será la cadena vacía.


```python
@staticmethod
def login_required(method):
```

- Decorador para comprobar si un usuario está logeado.
- En caso de no estarlo, habrá que lanzar una excepción de tipo `MailError` con el mensaje User is not logged in
- Recuerda que la excepción recibe en su constructor tanto el mensaje de error como el objeto actual de tipo `MailServer`.

```python
@property
def sender(self) -> str:
```

- Devuelve el correo del remitente en formato: `<nombre-de-usuario>@<dominio>`
- No hay que aplicar ningún decorador aquí pero debes saber que esta propiedad sólo va a funcionar si se ha hecho "login" previamente, ya que en otro caso no disponemos del dominio.

```python
@login_required
def send_mail(self, *, recipient: str, subject: str, body: str) -> None:
```

- Realiza (simula) en envío de un correo.
- Habrá que construir un objeto `Mail` y enviar el correo mediante los argumentos recibidos.
- Si `recipient` no tiene un formato válido de *email* habrá que lanzar una excepción de tipo `MailError` con el mensaje Recipient has invalid mail format
- Recuerda que la excepción recibe en su constructor tanto el mensaje de error como el objeto actual de tipo `MailServer`.

```python
@login_required
def get_emails(self, sent: bool = True):
```

- **Función generadora** que devuelve objetos de tipo `Mail`.
- El comportamiento es el siguiente:
  - Si `sent` es verdadero, se devuelven los correos enviados por el usuario actualmente logeado.
  - Si `sent` es falso, se devuelven los correos recibidos por el usuario actualmente logeado.

6. Clase `MailError`

Representa un error en la gestión de correos electrónicos.

Implementa, *al menos*, los siguientes **métodos**:


```
def __init__(self, message: str, db_handler: DbHandler):
```

- Constructor de la clase.
- El argumento `message` es el mensaje que debe procesar la excepción.
- Esta clase hereda de `Exception`...
- Cierra la conexión a la base de datos.

