Sistema Help Desk

Detalhamento das funções

Criação de usuário

O fluxo mostra como as 3 camadas da arquitetura trabalham juntas (Interface, Core e Infraestrutura)

INTERFACE

1. Tudo começa na camada de apresentação, especificamente na classe `UserCLI`. O papel dela é apenas conversar com o usuário.
2. Ao digitar a opção de criar usuário, o metodo `create_user_flow` solicita nome, email, senha e cargo. Ele nao faz nenhuma validação complexa e não toma nenhuma decisão de negócio.
3. De posse dos dados , a `UserCLI` empacota essas informações e entrega pro cérebro do sistema, que é o caso de uso `CreateUser`.

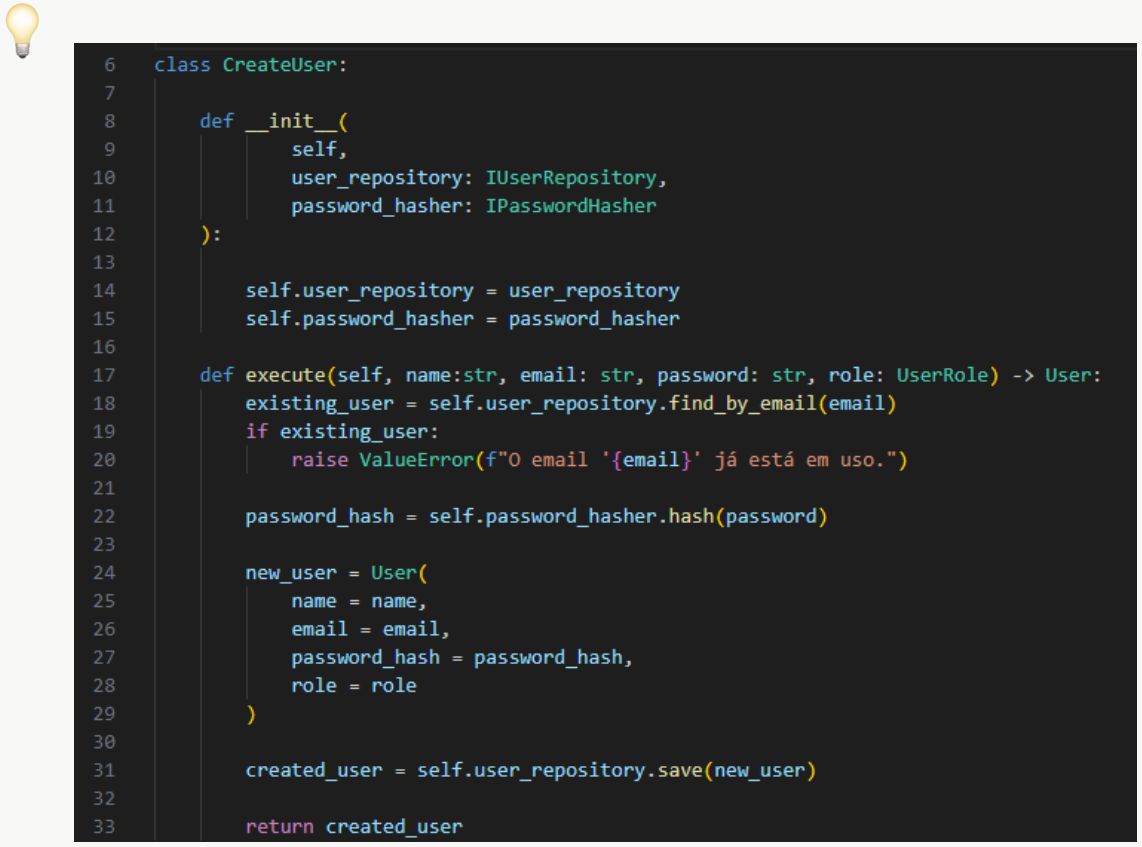
```python
class UserCLI:
    def create_user_flow(self):

        print("\n--- Cadastro de Novo Usuário ---")

        try:
            name = input("Nome: ")
            email = input("E-mail: ")
            password = input("Senha: ")

            print("Escolha o cargo: 1-Admin, 2-Tecnico, 3-Usuario")
            role_choice = input("> ")
            role_map = {
                "1": UserRole.ADMIN,
                "2": UserRole.TECHNICIAN,
                "3": UserRole.USER
            }
            role = role_map.get(role_choice, UserRole.USER)

            created_user = self.create_user_case.execute(
                name=name,
                email=email,
                password=password,
                role=role
            )
            print("\n✅ Usuário criado com sucesso!")
            print(f"ID: {created_user.id}, Nome: {created_user.name}. Email: {created_user.email}, Cargo: {created_user.role.value}")

        except ValueError as e:
            # Captura erros de negócio (ex: email duplicado) e mostra ao usuário
            print(f"\n❌ Erro ao criar usuário: {e}")
        except Exception as e:
            # Captura outros erros inesperados
            print(f"\n❌ Ocorreu um erro inesperado: {e}")
```

CAMADA CORE - CASO DE USO

1. A requisição chega na classe `CreateUser` da camada `core`. Essa classe é a orquestradora da lógica de negócio.
2. O método `execute` aplica as regras de negócio:
   a. o email não pode ser repetido, e pra isso ele usa o `IUserRepository` para chamar o método `find_by_email`.
   b. a senha em texto puro não deve avançar, então é usado o `IPasswordHasher` pra transformar a senha em um hash seguro

> 💡 O caso de uso não depende das classes concretas, ele depende apenas dos contratos. (Principio de inversão de dependência).

1. Com o email validado e a senha hasheada o caso de uso cria o objeto `User` e delega a tarefa final de salvar os dados para o `IUserRepository`.



```python
class CreateUser:

    def __init__(
            self,
            user_repository: IUserRepository,
            password_hasher: IPasswordHasher
    ):

        self.user_repository = user_repository
        self.password_hasher = password_hasher

    def execute(self, name:str, email: str, password: str, role: UserRole) -> User:
        existing_user = self.user_repository.find_by_email(email)
        if existing_user:
            raise ValueError(f"O email '{email}' já está em uso.")

        password_hash = self.password_hasher.hash(password)

        new_user = User(
            name = name,
            email = email,
            password_hash = password_hash,
            role = role
        )

        created_user = self.user_repository.save(new_user)

        return created_user
```

CAMADA DE INFRAESTRUTURA

1. A chamada para `hash()` é direcionada para a `BcryptPasswordHasher`, que usa a biblioteca para fazer o trabalho de criptografia

```python
class BcryptPasswordHasher(IPasswordHasher):

    def hash(self, password: str) -> str:
        password_bytes = password.encode('utf-8')

        salt = bcrypt.gensalt()
        hashed_bytes = bcrypt.hashpw(password_bytes, salt)

        return hashed_bytes.decode('utf-8')

    def verify(self, plain_password: str, hashed_password: str) -> bool:
        plain_password_bytes = plain_password.encode('utf-8')
        hashed_password_bytes = hashed_password.encode('utf-8')

        return bcrypt.checkpw(plain_password_bytes, hashed_password_bytes)
```

1. A chamada para `save()` é direcionada para a `MySQLUserRepository`, que traduz a chamada em um `INSERT` no SQL, usando queries parametrizadas para prevenir ataques de SQL Injection.

```python
class MySQLUserRepository(IUserRepository):

    def __init__(self):
        self.table_name = "users"

    def save(self, user: User) -> User:
        query = f"""
            INSERT INTO {self.table_name} (name, email, password_hash, role)
            VALUES (%s, %s, %s, %s)
        """
        params = (user.name, user.email, user.password_hash, user.role.value)

        with db_handler.managed_cursor() as cursor:
            cursor.execute(query, params)
            user.id = cursor.lastrowid
        return user
```

1. O banco de dados salva o usuário e retorna o novo ID, então devolve a informação do repositório para o caso de uso e em seguida para a `CLI`, mostrando a mensagem de sucesso para o usuário.

Login do usuário

O login também funciona com separação de camadas pra garantir a segurança.

INTERFACE

1. Novamente no `UserCLI`, o metodo `login_flow` solicita email e senha do usuário e em seguida passa essas credenciais pro caso de uso `LoginUser` que fica no Core

CAMADA CORE - CASO DE USO

1. A classe `LoginUser` é reponsável pela lógica de autenticação, e seu processo é uma verificação em 2 etapas
   a. Primeiro ela usa o `IUserRepository` pra chamar o metodo `find_by_email`. Se o repositório retornar `None` significa que o usuário não existe, então o processo falha e retorna um erro de credenciais inválidas
   b. Segundo, a verificação da senha. Se o usuário foi encontrado (email) o caso de uso pega a senha pura que veio da interface e o hash que veio do banco de dados, então entrega os dois pro `IPasswordHasher`, chamando o metodo `verify`

```python
class LoginUser:

    def __init__(self, user_repository: IUserRepository, password_hasher: IPasswordHasher):
        self.user_repository = user_repository
        self.password_hasher = password_hasher

    def execute(self, email: str, password: str) -> User:

        user = self.user_repository.find_by_email(email)
        if not user:
            raise InvalidCredentialsError("Email ou senha inválidos.")

        is_password_valid = self.password_hasher.verify(plain_password = password, hashed_password = user.password_hash)

        if not is_password_valid:
            raise InvalidCredentialsError("Email ou senha inválidos.")

        return user
```

> 💡 Independente da informação errada (email ou senha) a mensagem de erro será a mesma. Isso é feito propositalmente para casos de uma tentativa de invasão o atacante não saber qual dado está incorreto.

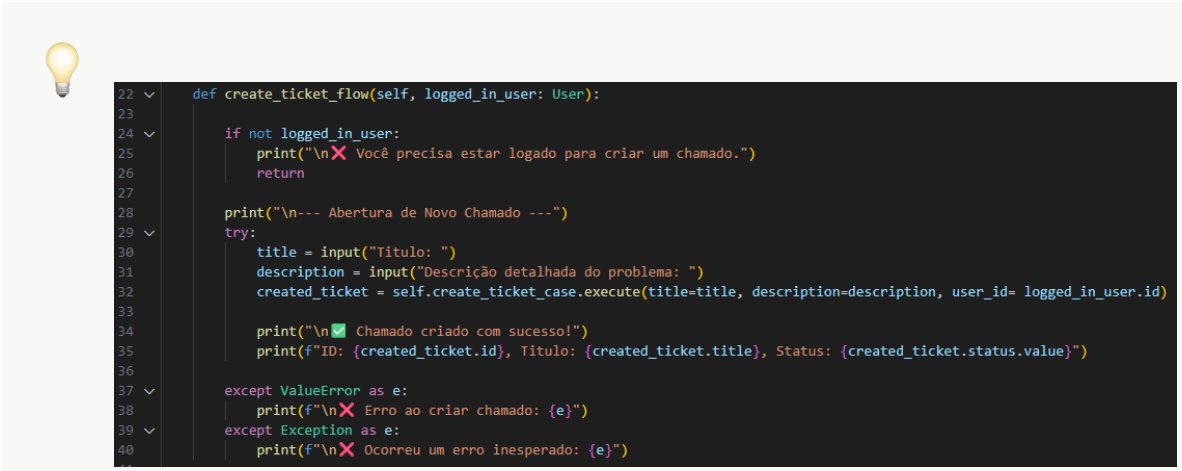
CAMADA DE INFRAESTRUTURA

1. Aqui a `MySQLUserRepository` executa um select para encontrar o usuário, seguida pela `BcryptPasswordHasher` que usa a função `checkpw` do `bcrypt` para comparar a senha inserida com um hash
2. Ambas as etapas bem sucedidas, o objeto `User` completo é retornado. Ele vai para o `UserCLI` e em seguida para o `main.py`
3. No `main.py`, a variável `logged_in_user` deixa de ser `None` e passa a conter o objeto do usuário
4. Na proxima iteração do loop principal o sistema detecta que agora existe um usuário logado e exibe o menu de opções para usuários logados (dependendo do cargo mais ou menos opções são exibidas)

Criação de um novo chamado

INTERFACE

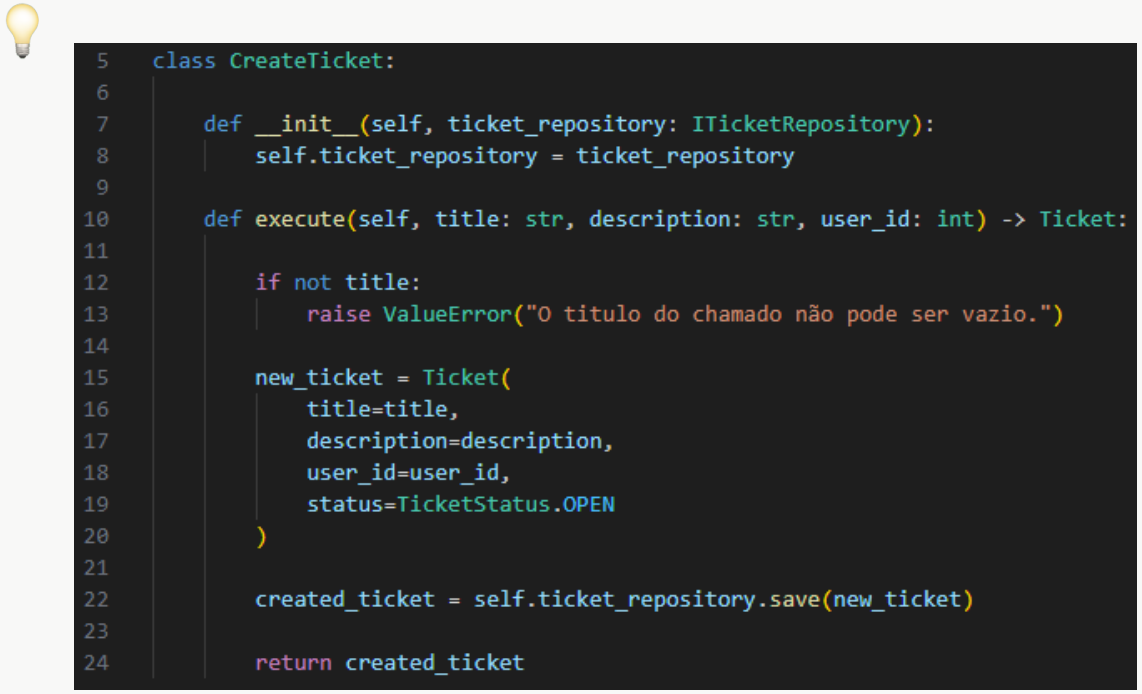
1. Com o usuário logado, o menu agora exibe novas opções. Isso é controlado pela variavel `logged_in_user` no `main.py`
2. Quando o usuário escolhe “Abrir novo chamado”, o `main.py` delega a tarefa para o `TicletCLI`.
3. O método `create_ticket_flow` recebe como parâmetro o objeto `logged_in_user`, isso é essencial pra interface saber quem está criando o chamado pra vincular corretamente.
4. Assim como nas outras interfaces, a `TicketCLI` apenas pede as informações pro usuário (título e descrição) e então passa esses dados junto com o ID do usuário logado para o caso de uso `CreateTicket` no `Core`.



```python
def create_ticket_flow(self, logged_in_user: User):

    if not logged_in_user:
        print("\n❌ Você precisa estar logado para criar um chamado.")
        return

    print("\n--- Abertura de Novo Chamado ---")
    try:
        title = input("Titulo: ")
        description = input("Descrição detalhada do problema: ")
        created_ticket = self.create_ticket_case.execute(title=title, description=description, user_id= logged_in_user.id)

        print("\n✅ Chamado criado com sucesso!")
        print(f"ID: {created_ticket.id}, Titulo: {created_ticket.title}, Status: {created_ticket.status.value}")

    except ValueError as e:
        print(f"\n❌ Erro ao criar chamado: {e}")
    except Exception as e:
        print(f"\n❌ Ocorreu um erro inesperado: {e}")
```

CAMADA CORE - CASO DE USO

1. A requisição chega na classe `CreateTicket` na camada `Core`. No seu construtor, ela recebe a dependência `ITicketRepository`, que é o contrato para todas as operações de banco de dados relacionadas a chamados.
2. O método `execute` aplica uma regra de negócio, onde falida se o título do chamado não está vazio, e se estiver, lança um erro e para a operação
3. Se a validação passar, o caso de uso cria uma nova instância da entidade `Ticket`, preenche os dados recebidos e então define o inicial do chamado como `OPEN` de forma explícita. Garantindo que a lógica de negócios, e não o banco de dados, seja a informação verdadeira para o estado inicial do chamado.
4. Com o objeto `Ticket` pronto, o caso de uso delega a tarefa de persistência ao seu repositório, chamando o método `save`.



```python
class CreateTicket:

    def __init__(self, ticket_repository: ITicketRepository):
        self.ticket_repository = ticket_repository

    def execute(self, title: str, description: str, user_id: int) -> Ticket:

        if not title:
            raise ValueError("O titulo do chamado não pode ser vazio.")

        new_ticket = Ticket(
            title=title,
            description=description,
            user_id=user_id,
            status=TicketStatus.OPEN
        )

        created_ticket = self.ticket_repository.save(new_ticket)

        return created_ticket
```

CAMADA DE INFRAESTRUTURA

1. A chamada `save` é recebida pela `MySQLTicketRepository` na camada de infraestrutura.
2. Essa classe implementa o contrato e traduz a ação em um `INSERT` no SQL para a tabela `tickets` utilizando o `DBConnectionHandler` para garantir que a transação com o banco seja segura e confiável.

Um ponto importante aqui é a integridade referencial dos dados. A tabela `tickets` foi criada com uma chave estrangeira que aponta para a `user_id` na tabela `users`. Isso faz com que o próprio banco de dados garanta que seja impossível criar um chamado pra um usuário que não existe.

1. Após o bd confirmar a inserção e retornar o novo ID do chamado, o objeto `Ticket` completo e atualizado passa do repositório para o caso de uso, e em seguida para a CLI (`TicketCLI`) que por fim exibe a mensagem confirmando que o chamado foi aberto.

```python
class CreateUser:

    def __init__(
            self,
            user_repository: IUserRepository,
            password_hasher: IPasswordHasher
    ):

        self.user_repository = user_repository
        self.password_hasher = password_hasher

    def execute(self, name:str, email: str, password: str, role: UserRole) -> User:
        existing_user = self.user_repository.find_by_email(email)
        if existing_user:
            raise ValueError(f"O email '{email}' já está em uso.")

        password_hash = self.password_hasher.hash(password)

        new_user = User(
            name = name,
            email = email,
            password_hash = password_hash,
            role = role
        )

        created_user = self.user_repository.save(new_user)

        return created_user
```

EXEMPLOS DE APLICAÇÃO DE POO

- Abstração e Encapsulamento
  - A classe `BcryptPasswordHasher`.
    Quando um caso de uso precisa de um hash ele simplesmente chama o método `hash("senha123")`. Ele não precisa saber nada sobre "salts" ou conversão para bytes, pois tudo está encapsulado dentro da classe que abstrai a tarefa de criar um hash em um único método simples.
- Herança
  - A implementação de contratos.
    A classe `MySQLUserRepository` herda da classe abstrata `IUserRepository`. Ao fazer isso ela não herda só a forma dos métodos, mas também a obrigação de implementa-los.
- Polimorfismo

- O Caso de Uso `CreateUser`. Ele tem uma variável, `self.user_repository`, que espera um objeto do tipo `IUserRepository`. Ele chama `self.user_repository.save(user)`. O `CreateUser` não sabe se o objeto real é um `MySQLUserRepository` (que executa uma query `INSERT`) ou se, no futuro, teríamos um `PostgresUserRepository` (que executaria uma query similar) ou até um `InMemoryTestRepository` (que apenas adicionaria o usuário a uma lista em memória para testes). Cada um desses objetos responderia à chamada `.save()` de sua própria maneira.

EXEMPLOS D EAPLICAÇÃO DE SOLID

- S - Principio de responsabilidade única
  - A `UserCLI` tem a única responsabilidade de interagir com o usuário.
  - O `CreateUser` (Caso de Uso) tem a única responsabilidade de orquestrar a lógica de criação de um usuário.
  - O `MySQLUserRepository` tem a única responsabilidade de traduzir objetos `User` para comandos SQL.
  - Cada classe é especialista em uma única tarefa.
- O - Princípio de Aberto/Fechado
  - Adicionar suporte a um novo banco de dados.
    Se for necessário suportar o PostgreSQL não será necessário mudar nenhuma linha de código no `Core`. Em vez disso bastaria criar uma nova classe, por exemplo, `PostgreUserRepository` na camada de infraestrutura
- L - Principio da Substituição de Liskov
  - O uso de injeção de dependência.
    O `CreateUser` espera um `IUserRepository`. É passado um `MYSQLUserRepository`, que é um “filho” de `IUserRepository`. A aplicação funciona perfeitamente porque a classe filha cumpre todas as promessas de contrato da classe mãe.
- I - Princípio da Segregação de Interfaces
  - Não foi criada uma única interface gigante com todos os métodos pra usuários e chamados. Em vez disso, elas foram segregadas: foi criada

uma `IUserRepository` e uma `ITicketRepository`. O `CreateUser` depende apenas da `IUSerRepository` e não sabe nada sobre os métodos para manipular chamados. Isso impede que as classes dependam de métodos que não usam.

- D - Princípio de Inversão de Dependências
  - O `Core` (alto nível) não importa nada da `Infraestrutura` (baixo nível). Em vez disso, o `core` define os contratos (interfaces abstratas)
    Tanto o `Core` (que usa o contrato) quanto a `Infraestrutura` (que implementa o contrato) dependem dessa abstração. A dependencia que normalmente iria do Core para a `infraestrutura` foi invertida, e agora a `infraestrutura` depende de uma regra definida pelo `Core`.