

Relatório 3

Prática: Validação de dados com Pydantic (I)

Kaíque Medeiros Lima

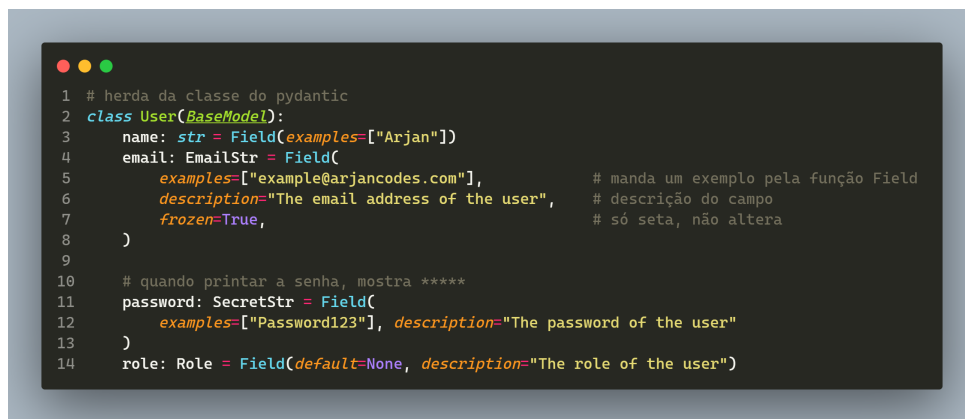
1 Introdução

Este relatório apresenta uma análise detalhada dos exercícios práticos realizados com a biblioteca **Pydantic** em Python, abordando conceitos fundamentais de validação de dados, serialização e integração com **FastAPI**. O estudo foi desenvolvido através de uma série de notebooks Jupyter que exploram progressivamente as funcionalidades da biblioteca, desde conceitos básicos até implementações mais avançadas com APIs REST.

2 Descrição da atividade

2.1 Example 1

2.1.1 Aula



```
1 # herda da classe do pydantic
2 class User(BaseModel):
3     name: str = Field(examples=["Arjan"])
4     email: EmailStr = Field(
5         examples=["example@arjancodes.com"],      # manda um exemplo pela função Field
6         description="The email address of the user", # descrição do campo
7         frozen=True,                               # só seta, não altera
8     )
9
10 # quando printar a senha, mostra *****
11 password: SecretStr = Field(
12     examples=["Password123"], description="The password of the user"
13 )
14 role: Role = Field(default=None, description="The role of the user")
```

Figura 1: example_1_aula.ipynb

- Implementação básica de uma classe **User** herdando de **BaseModel**.
- Uso de tipos especiais como **EmailStr** e **SecretStr**, apresentando o conceito de **Field()** para a configuração de campos.
- Configuração de campos com **Field()**, incluindo exemplos e descrições.
- Tratamento de erros de validação com **ValidationError**.

2.1.2 Prática

- Extensão do modelo base com campos adicionais: espécie, gênero e idade.
- Implementação de validação básica de dados.
- Teste com dados válidos e inválidos.

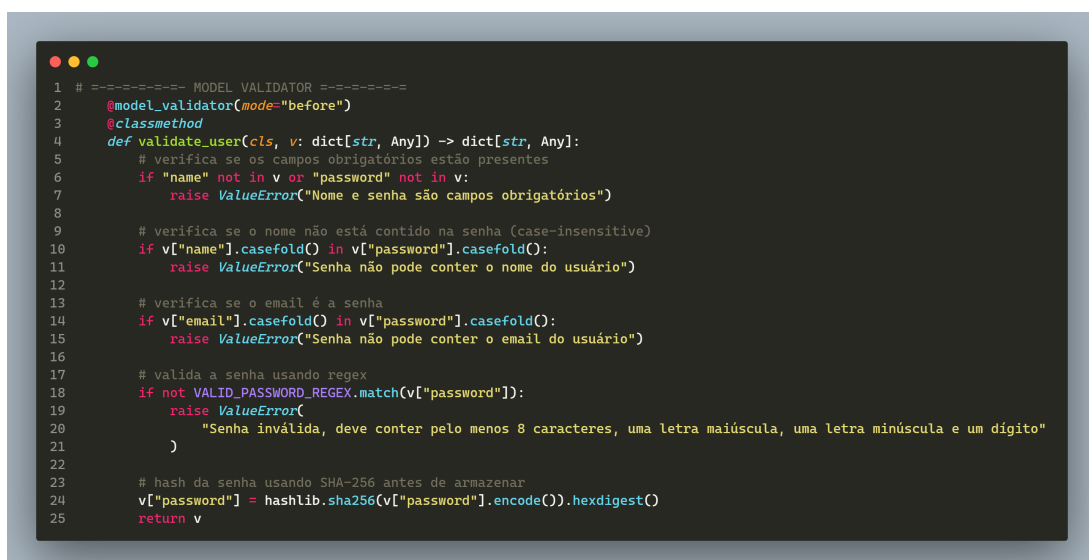
2.2 Example 2

2.2.1 Aula

- Implementação de expressões regulares para validação de senhas e nomes usando `VALID_PASSWORD_REGEX` e `VALID_NAME_REGEX`.
- Uso de `@field_validator` para validação individual de campos.
- Uso de `@model_validator` para validação cruzada entre campos.
- Hash de senhas com SHA-256 antes do armazenamento.
- Foi apresentado também o conceito de `before` e `after` validator, sendo o `before` validator executado antes da validação do modelo e o `after` validator executado após a validação do modelo.

2.2.2 Prática

- Validações customizadas para campos específicos (espécie, gênero, idade).
- Regras de negócio como verificação de idade mínima (18 anos).
- Validação de valores permitidos para campos categóricos.



```
1 # ----- MODEL VALIDATOR -----
2 @model_validator(mode="before")
3 @classmethod
4 def validate_user(cls, v: dict[str, Any]) -> dict[str, Any]:
5     # verifica se os campos obrigatórios estão presentes
6     if "name" not in v or "password" not in v:
7         raise ValueError("Nome e senha são campos obrigatórios")
8
9     # verifica se o nome não está contido na senha (case-insensitive)
10    if v["name"].casefold() in v["password"].casefold():
11        raise ValueError("Senha não pode conter o nome do usuário")
12
13    # verifica se o email é a senha
14    if v["email"].casefold() in v["password"].casefold():
15        raise ValueError("Senha não pode conter o email do usuário")
16
17    # valida a senha usando regex
18    if not VALID_PASSWORD_REGEX.match(v["password"]):
19        raise ValueError(
20            "Senha inválida, deve conter pelo menos 8 caracteres, uma letra maiúscula, uma letra minúscula e um dígito"
21        )
22
23    # hash da senha usando SHA-256 antes de armazenar
24    v["password"] = hashlib.sha256(v["password"].encode()).hexdigest()
25    return v
```

Figura 2: example_2_pratica.ipynb

2.3 Example 3

2.3.1 Aula

- Uso de `@field_serializer` para personalizar a serialização de campos específicos.
- Uso de `@model_serializer` com modo “wrap” para o controle da serialização do usuário.
- Diferentes modos de serialização `dict` e `JSON`.
- Exclusão de campos sensíveis da serialização.

2.3.2 Prática

- Implementação de serialização personalizada para todos os campos customizados.
- Validação pós-processamento com `@model_validator(mode="after")`.
- Regras de negócio específicas apenas um pode ser Admin.
- Conversão de dados para diferentes formatos de saída.

```
1 # ===== SERIALIZER =====
2 # o field_serializer permite que você serialize o campo de uma maneira personalizada
3 @field_serializer("role", when_used="json")
4 @classmethod
5 def serialize_role(cls, v) -> str:
6     return v.name # vai serializar o campo role como uma string
7
8 @field_serializer("especie", when_used="json")
9 @classmethod
10 def serialize_especie(cls, v: str) -> str:
11     return v.lower() # capitaliza a primeira letra da espécie
12
13 @field_serializer("genero", when_used="json")
14 @classmethod
15 def serialize_genero(cls, v: str) -> str:
16     return v.lower() # capitaliza a primeira letra do gênero
17
18 @field_serializer("idade", when_used="json")
19 @classmethod
20 def serialize_idade(cls, v: int) -> str:
21     return str(v) # converte a idade para string
22
23 # como os usuários podem ser serializados
24 @model_serializer(mode="wrap", when_used="json")
25 def serialize_user(self, serializer, info) -> dict[str, Any]:
26     if not info.include and not info.exclude: # se não houver campos incluídos ou excluídos
27         return {
28             "name": self.name,
29             "role": self.role.name,
30             "email": self.email,
31             "especie": self.especie.lower()
32         } # retorna um dicionário com os campos
33     return serializer(self)
```

Figura 3: example_3_pratica.ipynb

2.4 Example 4

- Modelo `User` simplificado com campos `UUID4` para identificação única.
- Implementação da `FastAPI` com endpoints `GET` e `POST`.
- Uso de `TestClient` para testes automatizados.
- Validação automática de dados de entrada via `FastAPI`.
- Tratamento de erros HTTP com respostas adequadas.
- Gerenciamento de estado com lista de usuários em memória.

3 Dificuldades

- **Serialização Customizada:** O controle fino sobre a serialização, especialmente com `@model_serializer` em modo “wrap”, exige entendimento detalhado de como o `Pydantic` processa os dados.
- **Integração FastAPI:** A transição do `Pydantic` para `FastAPI` introduz complexidades adicionais relacionadas ao tratamento de *requests* HTTP e respostas adequadas.
- **Tratamento de Erros:** Implementar mensagens de erro consistentes, mantendo a compatibilidade com os padrões do `Pydantic`.

4 Conclusões

Eu realmente achei que trabalhar com o `Pydantic` valeu a pena: ele mostrou ser bem flexível e eficiente na hora de validar e transformar dados em Python. Começamos do básico só definir classes e deixar o `Pydantic` cuidar da parte chata de conferir tipos e, aos poucos, chegamos a cenários bem mais complexos, integrando tudo isso ao `FastAPI`. No fim das contas, dá para ver claramente como a biblioteca economiza tempo e espaço no código. Outro ponto que não dá para ignorar é como o `Pydantic` ajuda a deixar as APIs mais seguras e fáceis de manter. Em vez de escrever um monte de checagens manuais, a biblioteca faz quase tudo sozinha: exibe erros claros em português, garante que os dados batam com o que esperamos e ainda permite personalizar validações avançadas quando necessário. Isso reduz muito o famoso “código boilerplate” e deixa a vida do desenvolvedor mais tranquila.

A progressão didática dos exercícios, começando com conceitos fundamentais e evoluindo para implementações práticas, proporcionou uma base sólida para aplicação dos conhecimentos em projetos reais.

5 Referências