
RELATÓRIO DO PROJETO FINAL - COUTURE HAVEN

Equipe:

- Augusto Faria Soares - Desenvolvedor (1710558)
- Dionisio Rodrigues Pequeno Neves - Desenvolvedor (2010789)
- Guthierre Marques Teixeira - Desenvolvedor e analista de requisitos (2127307)
- Victor Kauan Lima de Oliveira - Desenvolvedor e analista de requisitos (2213002)

Professor(a): Samuel Lincoln Magalhães Barrocas

Disciplina: Programação Funcional

1. **Repositório:** <https://github.com/Dionisio-Rodrigues/Couture-Haven>

2. Introdução

Este documento tem como objetivo documentar as principais funcionalidades e requisitos da API do e-commerce Couture Haven, projeto desenvolvido para a disciplina de programação funcional.

O sistema foi desenvolvido utilizando as tecnologias Python (sobretudo, as bibliotecas Flask API, SQL Alchemy e PyJWT), SQLite (banco de dados) e Postman (cliente para o envio de requisições de HTTP e testes).

3. Descrição Geral

Para o desenvolvimento do projeto foi utilizada a metodologia ágil SCRUM e a ferramenta Notion para a organização das sprints e atividades. O código do projeto está sendo armazenado em um repositório no GitHub.

Além disso, a utilização da programação funcional é um requisito obrigatório da disciplina, por isso, todos os scripts Python utilizam os conceitos de lambda, alta ordem, recursividade, list comprehension, currying, funtores, monad, etc.

4. Programação Funcional

4.1. Função alta ordem

O conceito de funções de alta ordem foi utilizado nos monads da aplicação, visto que os mesmos recebem uma função como parâmetro.



```
1 maybe_bind_id = lambda id, function: None if id is None else function(id)
```



```
1 view = lambda id: maybe_bind_id(id, view_flow)
```




```
1 update = lambda id: maybe_bind_id(id, update_flow)
```




```
1 destroy = lambda id: maybe_bind_id(id, destroy_response)
```



```
1 view = lambda id: maybe_bind_id(id, view_flow)
```




```
1 update = lambda id: maybe_bind_id(id, update_flow)
```



```
1 destroy = lambda id: maybe_bind_id(id, destroy_response)
```

4.2. Recursividade

O conceito de recursividade é utilizado na função “format_conditions”, uma função que formata um dicionário de condições e seus valores, caso sejam dicionários.



```
1 format_conditions = lambda conditions: "[{}]" .format(", ".join(
2     [
3         "{}: {}".format(key, f"({format_conditions(value)})" if isinstance(value, dict) else value)
4         for key, value in conditions.items()
5     ]
6 ))
7
```

4.3. Currying

O conceito de currying (chamadas sucessivas) é utilizado na função “check_password” para a autenticação do usuário do sistema.



```
1 check_password = (
2     lambda authorization: (
3         lambda username: (
4             lambda password: (
5                 {
6                     "message": "User authenticated successfully.",
7                     "token": jwt.encode(
8                         {"username": username, "exp": datetime.now() + timedelta(hours=12)}, SECRET_KEY
9                     ),
10                }
11                if check_password_hash(password, authorization["password"])
12                else unauthorized_response()
13            )
14        )
15    )
16 )
```

4.4. List comprehension

O conceito de list comprehension é utilizado para o populamento do banco de dados (seed).

```
1 insert_users = lambda: (  
2     print("INSERTING USER(S) INTO DATABASE..."),  
3     [post(url=f"{BASE_API_URL}/user", json=user) for user in USERS],  
4     print(f"[SUCCESS] {len(USERS)} USER(S) INSERTED."),  
5 )  
6  
7 insert_categories = lambda token: (  
8     print("INSERTING CATEGORY(IES) INTO DATABASE..."),  
9     [post(url=f"{BASE_API_URL}/category?token={token}", json=category) for category in CATEGORIES],  
10    print(f"[SUCCESS] {len(CATEGORIES)} CATEGORY(IES) INSERTED."),  
11 )  
12  
13 insert_products = lambda token: (  
14     print("INSERTING PRODUCT(S) INTO DATABASE..."),  
15     [post(url=f"{BASE_API_URL}/product?token={token}", json=product) for product in PRODUCTS],  
16     print(f"[SUCCESS] {len(PRODUCTS)} PRODUCT(S) INSERTED."),  
17 )
```

4.5. Dicionário

Dicionários são utilizados no escopo de algumas funções do sistema, como a “to_dict” responsável por representar modelos em formato de dicionário e “contains_name_response”, “contains_message_response” ou “contains_username_response” que verifica se um dicionário possui determinadas chaves.

```
1 to_dict = lambda self: {"id": self.id, "name": self.name}
```

```
1 to_dict = lambda self: {"id": self.id, "message": self.message, "timestamp": self.timestamp}  
2
```

```
1 to_dict = lambda self: {"id": self.id, "name": self.name, "price": self.price, "category_id": self.category_id}  
2
```

```

1     to_dict = lambda self: {"id": self.id, "username": self.username}
2

```

```

1 contains_name_response = lambda body: (
2     None
3     if "name" in body.keys()
4     else ({ "error": "Invalid payload", "message": "Please provide the required category fields." }, 400)
5 )

```

```

1 contains_message_response = lambda body: (
2     None
3     if "message" in body.keys()
4     else ({ "error": "Invalid payload", "message": "Please provide the required log fields." }, 400)
5 )

```

```

1 contains_username_response = lambda body: (
2     None
3     if "username" in body.keys() and "password" in body.keys()
4     else ({ "error": "Invalid payload", "message": "Please provide the required user fields." }, 400)
5 )

```

4.6. Functor (map, filter, reduce...)

Os funtores são utilizados na função de validação “is_valid” e de conversão de modelos para dicionário “models_to_dict”.

```

1 is_valid = lambda body, obj_class: all([field in vars(obj_class).keys() for field in body.keys()])
2

```

```

1 models_to_dict = lambda models: list(map(lambda model: model.to_dict(), models))
2

```

4.7. Monad

```
1 maybe_bind_id = lambda id, function: None if id is None else function(id)
```

```
1 view = lambda id: maybe_bind_id(id, view_flow)
```

```
1 update = lambda id: maybe_bind_id(id, update_flow)
```

```
1 destroy = lambda id: maybe_bind_id(id, destroy_response)
```

```
1 view = lambda id: maybe_bind_id(id, view_flow)
```



5. Requisitos

5.1. Funcionais

ID	Funcionalidade
RF01	O sistema deve possuir um endpoint de criação de produtos.
RF02	O sistema deve possuir um endpoint de atualização parcial de produtos.
RF03	O sistema deve possuir um endpoint de atualização integral de produtos.
RF04	O sistema deve possuir um endpoint de listagem de produtos.
RF05	O sistema deve possuir um endpoint de listagem individual de produtos.
RF06	O sistema deve possuir um endpoint de exclusão de produtos.
RF07	O sistema deve possuir um endpoint de criação de categorias.
RF08	O sistema deve possuir um endpoint de atualização parcial de categorias.
RF09	O sistema deve possuir um endpoint de atualização integral de categorias.
RF10	O sistema deve possuir um endpoint de listagem de categorias.
RF11	O sistema deve possuir um endpoint de listagem individual de categorias.
RF12	O sistema deve possuir um endpoint de exclusão de categorias.

RF13	O sistema deve possuir um endpoint de criação de usuários.
RF14	O sistema deve possuir um endpoint de autenticação de usuários.

O arquivo “product.py”, em “src/routes”, contém a aplicação dos primeiros 6 requisitos funcionais (01 - 06), visto que possui todos os endpoints relacionados à entidade produto, realizando operações no banco de dados através de seus serviços.

O arquivo “category.py”, em “src/routes”, contém a aplicação dos próximos 6 requisitos funcionais (07 - 12), nele estão os endpoints relacionados à entidade categoria.

Por último, os requisitos funcionais 13 e 14 estão nos arquivos “user.py” e “auth.py”, em “src/routes”, respectivamente.

5.2. Não-funcionais

ID	Funcionalidade
RNF01	O sistema deve utilizar a linguagem de programação Python e as bibliotecas Flask API e SQL Alchemy para o servidor web.
RNF02	O sistema deve utilizar o banco de dados SQLite.
RNF03	O sistema deve utilizar os conceitos da programação funcional (lambda, currying, recursividade, monad...) nos seus endpoints e serviços.
RNF04	O sistema deve utilizar JWT e criptografia de dados para o sistema de autenticação.

O sistema utiliza a linguagem de programação Python e as bibliotecas Flask API (para o servidor web) e SQL Alchemy (para conexão com o banco de dados SQLite).

Todos os endpoints e serviços utilizam o conceito de programação funcional (lambda, currying, recursividade, monad, etc).

Além disso, o sistema de autenticação do sistema utiliza a biblioteca PyJWT para criação de JWT e criptografia de dados da biblioteca Werkzeug.