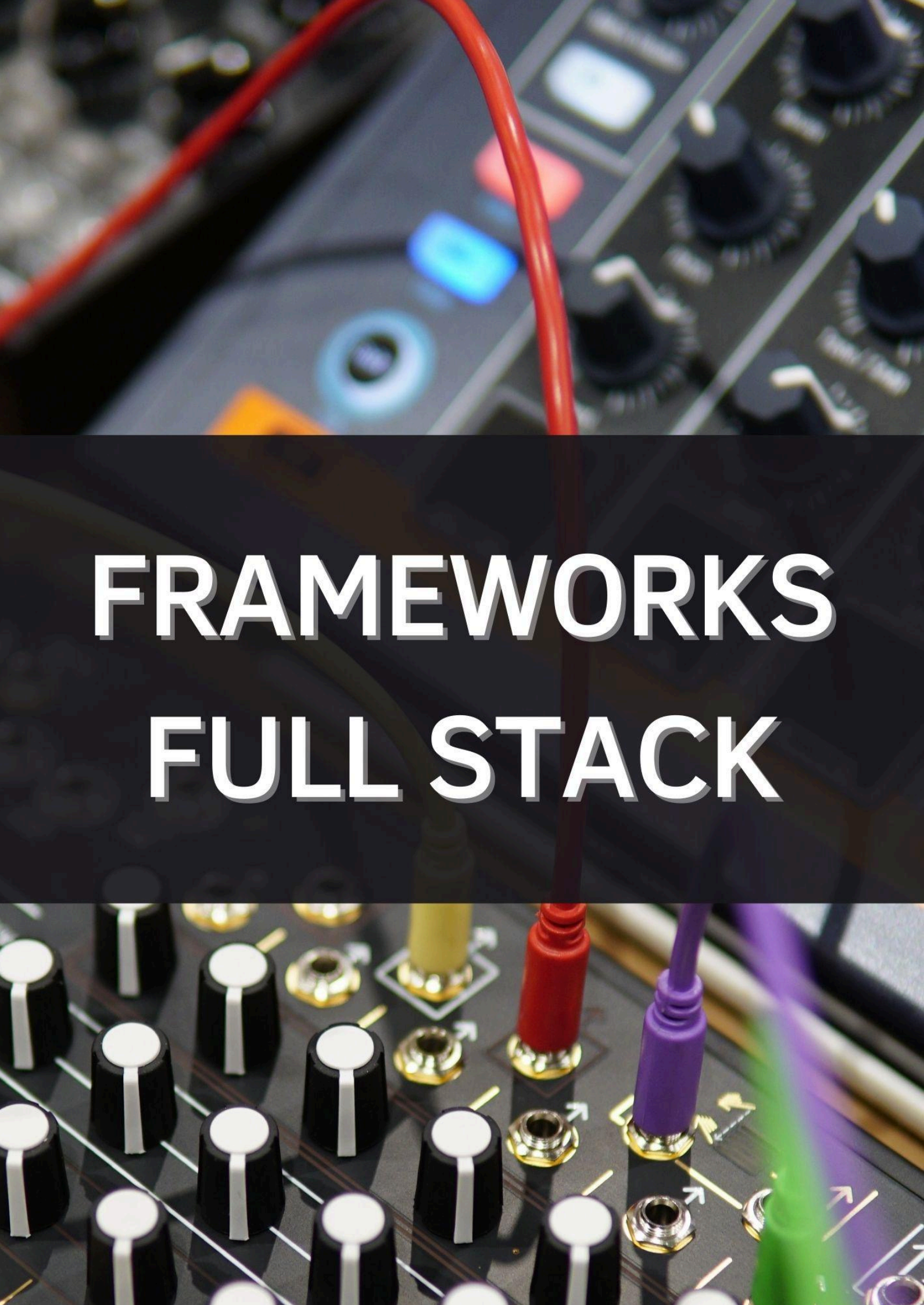
FRAMEWORKS
FULL STACK

Texto base

9

Execução de uma compra com persistência de dados

Prof. André Nascimento Maia
Prof. Caio Nascimento Maia

Resumo

Neste texto, são discutidas as formas de modularização de uma aplicação Flask, utilizando a estrutura funcional ou divisional. Também é aplicada a estruturação em um contexto de e-commerce fictício para a criação de um pedido de compra a partir de um carrinho de compras.

9.1. Introdução

Uma solução completa baseada em *software* geralmente contém uma aplicação de *front-end* e um aplicação de *back-end*. A aplicação *front-end*, quando é uma *web application* contém diversas visões, rotas e formas de manipulação da informação pelo usuário, de forma restrita ou pública. Essa aplicação *front-end* utiliza diversas APIs de *back-end*. Por exemplo, considerando o contexto de um e-commerce fictício, é necessário termos diversas APIs para manipulação e exibição de dados. Alguns exemplos de APIs são: Cart, para manter um carrinho de compras para um determinado usuário; Product, para disponibilizar de forma eficiente uma lista de produtos disponíveis para compra; Order, que permite coordenar o processo de compra através de um carrinho de compras formado pelo usuário, e também, consultar históricos de pedidos e compras de um determinado usuário.

A implementação de um conjunto de APIs em uma única aplicação utilizando Python e Flask, deve ser feito de forma organizada e clara para permitir facilidade de manutenção e evolução. Considerando estes aspectos, então, discutiremos neste texto qual seria a configuração adequada para uma API REST utilizando o *framework* Flask que implementa todas as APIs mínimas para uma e-commerce e persiste os dados em um banco de dados SQLite em memória, focando na funcionalidade de executar uma compra.

9.2. Organização da API REST

A organização de uma aplicação web é importante para garantirmos flexibilidade, manutenibilidade e facilidade na evolução das APIs e funcionalidades.

Para um e-commerce fictício, considere criar uma aplicação web utilizando Python, Flask e SQLAlchemy para integração com um banco de dados SQLite em memória. Esta aplicação, deve conter todas as APIs necessárias para que uma *web application* da camada de *front-end* seja capaz de exibir todas as informações relevantes para o usuário e também aplicar as ações necessárias no contexto de negócio para executar processos como executar uma compra a partir de um carrinho de compras.

Existem duas abordagens bem conhecidas para estruturar os arquivos e diretórios em uma aplicação utilizando o *framework* Flask, que são: Funcional e Divisional (Pallets Projects, 2022).

A estrutura funcional tende a organizar as partes da aplicação de acordo com o que ela faz. O Código 9.1 mostra um exemplo de uma API REST utilizando a estruturação funcional.

Código 9.1: Exemplo web API utilizando a estrutura funcional

```
yourapp/
    __init__.py
    controller/
      carts_controller.py
      products_controller.py
    services/
      __init__.py
      cart_services.py
      product_services.py
    model/
        __init__.py
        cart_model.py
        product_model.py
```

Fonte: autores, 2022.

A estrutura divisional tem os mesmo arquivos, porém a estrutura de diretórios é diferente. O Código 9.2 mostra a mesma aplicação do código 9.1, mas estruturado de forma divisional.

Código 9.2: Exemplo web API utilizando a estrutura divisional

```
yourapp/
    __init__.py
    carts/
      carts_controller.py
      cart_services.py
      cart_model.py
    products/
      __init__.py
      products_controller.py
      product_services.py
      product_model.py
```

Fonte: autores, 2022.

Na forma divisional, temos vários diretórios com as partes da aplicação agrupadas com o contexto que ela contribui.

Em ambos os casos é utilizado o contexto de Blueprints do *framework* Flask (Pallets Projects, 2022). Uma Blueprint é a recomendação do *framework* para agrupar todas as APIs e configurá-las de forma eficiente. O Código 9.3, mostra como devemos utilizar uma Blueprint em um projeto com uma estrutura funcional.

Código 9.3: Utilizando Blueprint com Flask

```
00. yourapp/controller/hello_controller.py
01. from flask import Blueprint
02.
03. hello_api = Blueprint('hello', __name__)
04.
05. @hello_api.route("/hello")
06. def hello_world():
07.     return "<p>Olá, Mundo!</p>"
```

Fonte: autores, 2022.

O Código 9.3 mostra a utilização de Blueprint para o registro de uma única rota chamada "/hello" para retornar um HTML com um parágrafo e a frase "Olá, Mundo!". Perceba que na linha 05, não registramos a rota diretamente na aplicação Flask, como geralmente é feito, mas registramos a rota na Blueprint que declaramos, chamada hello_api.

Após criar a Blueprint, é necessário registrá-la na aplicação. O Código 9.4 mostra que o processo de registro de Blueprints deve ser feito utilizando a Blueprint criada no Código 9.3.

Código 9.4: Registro de Blueprint na aplicação Flask.

```
01. # yourapp/__init__.py
02.
03. from flask import Flask
04. from .controller.hello_controller import hello_api
05.
06. app = Flask(__name__)
07. app.register_blueprint(hello_api)
```

Fonte: autores, 2022.

É necessário executar a função *register_blueprint* da aplicação Flask para registrar uma Blueprint, como mostrado na linha 07 do Código 9.4. Seguindo a estrutura funcional, a recomendação é criar um arquivo no diretório controller para cada uma das APIs disponíveis na web API.

9.3. API Order

Seguindo o contexto de um e-commerce fictício, quando precisamos executar uma compra é necessário ao menos ter selecionado alguns produtos disponíveis para compra, os adicionar em um carrinho de compras, e então, criar um pedido de compra a partir de um carrinho de compras preenchido. Internamente, o e-commerce deve se integrar com um *Payment Service Provider* (PSP) (EBANX, n.d.) para submeter os dados do comprador para pagamento.

Um exemplo de API REST para executar um pedido é demonstrado no Código 9.5. Nesta aplicação, estamos utilizando uma estrutura funcional para modularizar a aplicação, com os diretórios de nomes e funcionalidades conforme a seguir:

- **Controller**: o diretório *controller* é responsável por conter todas as Blueprints de cada uma das APIs disponíveis na aplicação. Dentro de cada arquivo, deve existir uma rota para cada *endpoint* de API. Esta é a camada mais externa, responsável por receber as requisições dos usuários, interpretá-las e direcioná-las internamente para a camada Service.
- **Service**: o diretório *service* representa a camada de regras de negócios. Todas as regras e manipulações do modelo devem estar nesta camada. Ela é responsável por expor as funcionalidades do negócio.
- **Model**: este diretório representa o nosso modelo de domínio do negócio. Ela reflete a estrutura do banco de dados e o caminho direto para integração com qualquer tipo de persistência de dados que seja utilizada no projeto.

Código 9.5: Exemplo de API REST Order.

```
01. # api/controller/orders_controller.py
02.
03. from flask import Blueprint, request
04. from api.service import order_service
05. from flask import jsonify
06.
07. order_api = Blueprint('orders', __name__)
08.
09. @order_api.route("/orders/", methods=['POST'])
10. def post():
11.     """Cria um pedido, a partir de um código de Cart informado."""
12.     payload = request.get_json()
13.     cart_code = payload["cart_code"]
14.
15.     order = order_service.create_from(cart_code)
16.
17.     return jsonify(order.serialized)
```

Fonte: autores, 2022.

No Código 9.5 mostra um *endpoint POST /orders/* que aceita um objeto JSON com um único atributo chamado *cart_code* (linha 13). O código do carrinho é passado para o *order_service* criar um novo pedido de compra a partir desse código.

Código 9.6: Criar um pedido a partir do código de um carrinho

```
01. def create_from(cart_code):
02.     with db.session.begin():
03.         cart = cart_service.get_by_code(cart_code)
04.         now = datetime.utcnow()
05.         order = Order(products=cart.content,
06.                       total_amount=sum_total(cart.content),
07.                         created_at=now, updated_at=now, status='DONE')
08.
09.         db.session.add(order)
10.         db.session.delete(cart)
11.
12.         # Commit a transaction do banco de dados.
13.         db.session.commit()
14.
15.         return order
```

Fonte: autores, 2022.

No Código 9.6, temos a operação de criar um pedido de compra a partir de um carrinho de compras. Uma das primeiras operações é abrir um contexto transacional com o banco de dados relacional para garantirmos que a operação será atômica (linha 02). Em seguida, obtemos o carrinho do banco de dados, e criamos um novo pedido a partir do conteúdo do carrinho (linha 05). Os próximos passos são persistir o pedido (linha 09) e apagar o carrinho de compra que foi utilizado (linha 10). Se tudo ocorreu conforme o esperado, finalizamos o contexto transacional no banco de dados (linha 13) e retornamos o pedido que foi criado.

Observe que o Código 9.6 cria um pedido de compra com o estado DONE, que significa "finalizado" em nosso domínio de exemplo. Obviamente, existem diversos outros passos a serem executados antes que o pedido seja considerado como "finalizado", como integração com um PSP, validação e cálculo de frete, entre outras necessidades de um e-commerce. Porém, para o nosso exemplo, isso é o suficiente.

9.3. Executando uma compra a partir de um carrinho de compras

Com a API Order pronta, em uma *web application* utilizando Next.js e React, podemos executar uma compra toda vez que clicarmos no botão "Fechar pedido", baseando-se no protótipo proposto na Figura 9.1.

Figura 9.1: Protótipo de tela de carrinho de compras.

Fonte: autores, 2022.

Na página de carrinho de compras podemos capturar o clique do botão "Fechar pedido" e fazermos uma chamada remota diretamente para a API Order utilizando o *endpoint POST /orders*. O Código 9.7, mostra essa integração do *front-end* com a API REST.

Código 9.7: Código *front-end* para criar um pedido de compra.

```
01. import { useRouter } from "next/router";
02.
03. import Layout from "../components/Layout";
04. import Cart from "../components/ShoppingCart/Cart";
05. import { closeOrder } from "../libs/api";
06. const FIXED_CART_CODE = "fixed-cart-code";
07.
08. export default function ShoppingCartPage() {
09.   const router = useRouter();
10.
11.   function close() {
12.     closeOrder(FIXED_CART_CODE);
13.     router.push("/orders");
14.   }
15.   return (
16.     <Layout>
17.       <h1 className="text-center mt-5">Carrinho de Compras</h1>
18.       <Cart />
19.       <button onClick={close} className="btn btn-primary">
20.         Fechar pedido
21.       </button>
22.     </Layout>
23.   );
24. }
```

Fonte: autores, 2022.

O Código 9.7, mostra a integração da página de carrinho de compra (*front-end*) com a API REST (*back-end*). O evento de clique do botão de fechar pedido é capturado e uma chamada para o *endpoint POST /orders* é executada. Nesse caso, para o exemplo, é informado um código fixo de carrinho de compras. Em seguida, o cliente é redirecionado para outra rota chamada /orders. Essa deve ser a página que lista todos os pedidos de compras disponíveis, e seus estados, para o usuário.

9.4. Conclusões

Assim como nas aplicações de *front-end*, pensar na estrutura correta para a aplicação de *back-end* é importante. Esta estrutura nos garantirá grande produtividade, facilidades de manutenção e adição de novas funcionalidades no futuro.

É importante analisar com atenção e aderir, quando possível, às recomendações de organização e modularização proposta pelo *framework*. Na maioria das vezes, existirão facilidades de configuração para essas recomendações.

Uma boa estratégia para desenvolvimento de software é sempre trabalhar camada a camada, mantendo as APIs claras e limpas para interligá-las depois. Isso fica claro quando se desenvolve uma API clara e em seguida, é necessário apenas, ligar o *front-end* utilizando os seus padrões de desenvolvimento de *software*.

Referências

EBANX. (n.d.). **Payment Service Provider | Payments Explained**. EBANX. Disponível em: <https://business.ebanx.com/en/resources/payments-explained/payment-service-providers>. Acesso em : 09 fev. 2022.

Pallets Projects. (2022). **Blueprints — Explore Flask 1.0 documentation**. Explore Flask. Disponível em: <http://exploreflask.com/en/latest/blueprints.html>. Acesso em: 09 fev. 2022.