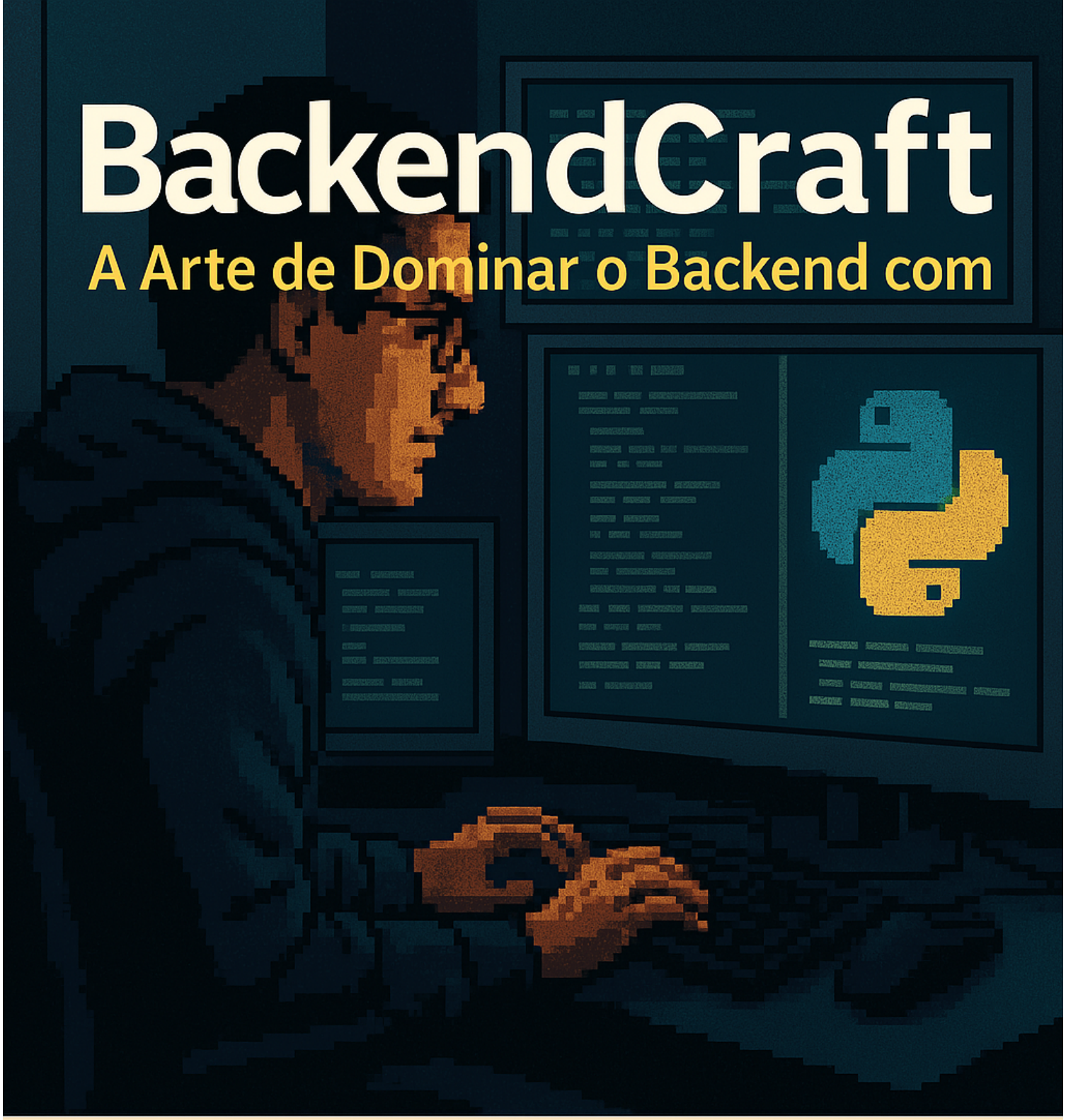

Capítulo 1: Introdução ao Mundo Backend

O que é Backend?

Que você vi precesar

Capítulo 1: Introdução ao Mundo Backend

O backend é o "cérebro" de qualquer aplicação. Ele roda em um servidor, recebe requisições vindas do frontend ■ normalmente via HTTP ■, processa regras de negócio, acessa bancos de dados e devolve respostas. • **Cliente■Servidor**: navegadores ou apps móveis (clientes) enviam requisições ao servidor (backend). • **Protocolos**: HTTP/HTTPS são os mensageiros; cada requisição contém método (GET, POST…), cabeçalhos e corpo. • **Responsabilidades**: autenticação, autorização, validação de dados, cálculos, armazenamento, integrações externas. • **Escalabilidade**: boas práticas de código + infraestrutura (cache, load balancer) garantem que o backend aguente picos de tráfego. **Por que Python?** - Sintaxe legível ➜ curva de aprendizagem suave. - Ecossistema gigantesco (Flask, Django, FastAPI, SQLAlchemy). - Produtividade elevada: menos código para entregar mais rápido. - Comunidade ativa, milhares de pacotes prontos.

Capítulo 2: Ambiente de Desenvolvimento

Antes de codar, precisamos de um ambiente limpo e reprodutível: • **Python 3.10+**: baixe do site oficial ou via gerenciadores (apt, brew, choco). • **Virtualenv**: isola dependências por projeto. ```bash python -m venv venv source venv/bin/activate # Linux / Mac venv\Scripts\activate # Windows ``` • **Editor**: VS Code com extensões Python (IntelliSense, linting, debugging). • **Gerenciamento de pacotes**: `pip` + `requirements.txt` ou `pip-tools/poetry`. • **Controle de versão**: Git + GitHub/GitLab para backups e colaboração. Após ativar o ambiente, instale Flask: ```bash pip install flask ```

Capítulo 3: Primeiro Projeto com Flask

A estrutura mínima de um app Flask: 1. **Instância do app**: ```python from flask import Flask app = Flask(__name__) ``` 2. **Rotas**: URLs que mapeiam para funções. ```python @app.route("/") def home(): return "Bem■vindo ao BackendCraft!" ``` 3. **Bootstrap**: ```python if __name__ == "__main__": app.run(debug=True) ``` • `debug=True` recarrega o servidor a cada alteração. • A aplicação roda em `localhost:5000`. **Ciclo Request■Response** Navegador ➜ `/` ➜ Flask cria `Request` ➜ executa `home()` ➜ devolve `Response` (HTML/JSON).

Capítulo 4: Trabalhando com Rotas e Parâmetros

Rotas dinâmicas recebem partes da URL como argumentos: ```python @app.route("/user/<nome>") def user(nome): return f"Olá, {nome}!" ``` • Visite `/user/Ana` → resposta personalizada. • Para tipos específicos (`int`, `float`, `path`): `/post/<int:id>`. **Query■strings** (`?page=2`) chegam em `request.args`. **HTTP methods**: use `methods=['POST']` para criar ou atualizar recursos. Boas práticas: • Separar rotas em Blueprints. • Validar entrada com pydantic ou Marshmallow. • Retornar códigos de status corretos (200, 201, 400, 404…).

Capítulo 5: Introdução ao Banco de Dados

SQLite rápido ```python import sqlite3 conn = sqlite3.connect('usuarios.db') cursor = conn.cursor() cursor.execute('CREATE TABLE IF NOT EXISTS usuarios (id INTEGER PRIMARY KEY, nome TEXT)') conn.commit() conn.close() ``` **Quando escalar** escolha PostgreSQL ou MySQL. Utilize **SQLAlchemy ORM** para mapear tabelas em classes Python: ```python from sqlalchemy import Column, Integer, String class Usuario(Base): __tablename__ = "usuarios" id = Column(Integer, primary_key=True) nome = Column(String) ``` Conceitos■chave: • ACID, transações, índices. • Migrations (Alembic) para versionar esquema. • Pool de conexões para performance.

Capítulo 6: APIs com Flask e JSON

Para APIs, retornamos objetos `dict` ou `list`, convertidos em JSON: ```python from flask import jsonify @app.route("/api") def api(): dados = {"nome": "BackendCraft", "versao": 1.0} return jsonify(dados), 200 ``` **REST Principles** • Recursos nomeados no plural (`/users`). • Métodos: GET (listar), POST (criar), PUT/PATCH (atualizar), DELETE (remover). • Status codes semânticos (201 Created, 204 No Content…). • Versionamento na URL (`/v1/`). **Documentação**: Swagger/OpenAPI via Flask■RESTX ou FastAPI (gerada automaticamente). **Testes**: `pytest` + `requests` ou `httpx` para validar endpoints.

Capítulo 7: Próximos Passos

Agora que a base está sólida: • **FastAPI**: performance assíncrona com `async/await` + tipagem padrão Pydantic. • **Banco não■relacional**: MongoDB, Redis para caching. • **Containers**: Docker para empacotar e rodar em qualquer lugar. • **CI/CD**: GitHub Actions para testes automáticos e deploy. • **Segurança**: autenticação JWT, HTTPS, rate■limit, CORS. • **Escalabilidade**: Gunicorn + Nginx, filas Celery/RabbitMQ, monitoramento Prometheus/Grafana.

Conclusão

Você agora entende a engrenagem que movimenta aplicações modernas. O caminho para maestria envolve: • **Prática contínua**: construa APIs, pequenos serviços e experimente padrões. • **Colaboração**: contribua em projetos open source, participe de comunidades. • **Estudo constante**: arquitetura de software, design patterns, observabilidade. _“Com grandes poderes (do backend) vêm grandes responsabilidades (de não quebrar produção).”_ Boa codificação!