Resumo de Assuntos Avançados em Desenvolvimento Web com Python - Explicação

Exemplo de Script de Deploy com Docker

```
FROM python:3.9-slim
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
CMD ["python", "app.py"]
```

Explicação:

- `FROM python:3.9-slim`: Define a imagem base do Docker, que é uma versão mínima do Python 3.9, para economizar espaço.
- `WORKDIR /app`: Define o diretório de trabalho dentro do contêiner como `/app`.
- `COPY . /app`: Copia todos os arquivos do diretório atual do host para o diretório `/app` no contêiner.
- `RUN pip install -r requirements.txt`: Instala as dependências listadas no arquivo `requirements.txt` usando o pip.
- `CMD ["python", "app.py"]`: Especifica o comando que será executado quando o contêiner for iniciado, que neste caso é rodar o aplicativo Python `app.py`.

Exemplo de Uso de MVC (Model-View-Controller) em Flask

app.py (Controller)

```
from flask import Flask, render_template
from models import get_data
app = Flask(__name__)

@app.route('/')
def home():
    data = get_data()
    return render_template('index.html', data=data)

if __name__ == '__main__':
    app.run()
```

Explicação:

- `from flask import Flask, render_template`: Importa o framework Flask e a função `render_template` para renderizar templates HTML.
- `from models import get_data`: Importa a função `get_data` do módulo `models`.
- `app = Flask(__name__)`: Cria uma instância do aplicativo Flask.
- `@app.route('/')`: Define uma rota para a URL base (`/`), associando-a à função `home`.
- `def home()`: Define a função `home`, que age como um controlador.
- `data = get_data()`: Chama a função `get_data` do modelo para obter dados.
- `return render_template('index.html', data=data)`: Renderiza o template `index.html`, passando os dados obtidos como contexto.
- `if __name__ == '__main__': app.run()`: Inicia o servidor Flask se o script for executado diretamente.

models.py (Model)

```
def get_data():
    return {"key": "value"}
```

Explicação:

- `def get_data()`: Define uma função `get_data` que simula a obtenção de dados do modelo.
- `return {"key": "value"}`: Retorna um dicionário com dados que serão usados na view.

index.html (View)

```
<html>
<body>
 <h1>{{ data['key'] }}</h1>
</body>
</html>
```

Explicação:

- `{{ data['key'] }}`: Utiliza a sintaxe de template do Jinja2 para exibir o valor associado à chave `key` do dicionário `data` passado pelo controlador.

Exemplo de Uso de React para um Componente de Botão

Button.js

```
import React from 'react';
```

```
function Button() {
    return (
        <button>Clique Aqui</button>
    );
}

export default Button;
```

Explicação:

- `import React from 'react';`: Importa a biblioteca React necessária para criar componentes.
- `function Button()`: Define um componente funcional chamado `Button`.
- `return (<button>Clique Aqui</button>);`: O componente retorna um botão HTML com o texto "Clique Aqui".
- `export default Button;`: Exporta o componente `Button` para que possa ser usado em outros arquivos.

Exemplo de Rota com Node.js e Express

server.js

```
const express = require('express');
const app = express();

app.get('/api/data', (req, res) => {
    res.json({ key: 'value' });
});

app.listen(3000, () => {
    console.log('Server running on port 3000');
});
```

Explicação:

- `const express = require('express');`: Importa o framework Express.
- `const app = express();`: Cria uma instância do aplicativo Express.
- `app.get('/api/data', (req, res) => { ... })`: Define uma rota GET para `/api/data`, que envia um objeto JSON como resposta.
- `app.listen(3000, () => { ... });`: Inicia o servidor na porta 3000 e exibe uma mensagem de log.

Exemplo de Consulta SQL para União de Tabelas

Consulta SQL com JOIN

```
SELECT users.name, orders.total
FROM users
JOIN orders ON users.id = orders.user_id
WHERE orders.total > 100;
```

Explicação:

- `SELECT users.name, orders.total`: Seleciona as colunas `name` da tabela `users` e `total` da tabela `orders`.
- `FROM users`: Especifica a tabela `users` como a tabela principal da consulta.
- `JOIN orders ON users.id = orders.user_id`: Realiza um JOIN (união) entre as tabelas `users` e `orders` onde `users.id` corresponde a `orders.user_id`.
- `WHERE orders.total > 100`: Filtra os resultados para incluir apenas as linhas onde o valor total dos pedidos (`orders.total`) é maior que 100.

Exemplo de Hashing de Senha com Bcrypt em Python

password_hashing.py

```
import bcrypt

password = 'mysecretpassword'
hashed = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())

# Verificação de senha
if bcrypt.checkpw(password.encode('utf-8'), hashed):
    print("A senha está correta")
else:
    print("A senha está incorreta")
```

Explicação:

- `import bcrypt`: Importa a biblioteca Bcrypt, usada para hashing de senhas.
- `password = 'mysecretpassword'`: Define a senha que será hashada.
- `hashed = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())`: Gera o hash da senha usando Bcrypt, com um salt aleatório.
- `if bcrypt.checkpw(password.encode('utf-8'), hashed):`: Verifica se a senha fornecida corresponde ao hash gerado.
- `print("A senha está correta")`: Imprime uma mensagem confirmando que a senha está correta.

Exemplo de Micro Serviço com Flask

microservice.py

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/api/service1')
def service1():
    return jsonify({'response': 'Service 1 Response'})

if __name__ == '__main__':
    app.run(port=5001)
```

Explicação:

- `from flask import Flask, jsonify`: Importa o framework Flask e a função `jsonify` para retornar respostas JSON.
- `app = Flask(__name__)`: Cria uma instância do aplicativo Flask.
- `@app.route('/api/service1')`: Define uma rota para o endpoint `/api/service1`.
- `def service1()`: Define a função `service1` que será chamada quando o endpoint for acessado.
- `return jsonify({'response': 'Service 1 Response'})`: Retorna uma resposta JSON com uma chave `response` e o valor `'Service 1 Response'`.
- `if __name__ == '__main__': app.run(port=5001)`: Inicia o servidor Flask na porta 5001 se o script for executado diretamente.

Exemplo de Chamada de API com Fetch em JavaScript

```
<script>
fetch('https://api.exemplo.com/data')
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error('Erro:', error));
</script>
```

Explicação:

- `fetch('https://api.exemplo.com/data')`: Realiza uma requisição GET para o endpoint da API fornecido.
- `.then(response => response.json())`: Converte a resposta da API para JSON.
- `.then(data => console.log(data))`: Imprime os dados da API no console.

- `.catch(error => console.error('Erro:', error))`: Captura e exibe qualquer erro que ocorrer durante a requisição.

Exemplo de Pipeline CI/CD com YAML no GitLab

```
# .gitlab-ci.yml
```

Explicação:

- `.gitlab-ci.yml`: Este arquivo define o pipeline de CI/CD para automação de testes, builds e deploys em um projeto hospedado no GitLab.

Exemplo de Arquivo .github/workflows/ci.yml

Aqui está um exemplo de um arquivo `.yml` para configurar um pipeline de CI/CD no GitHub Actions. Esse pipeline executa um conjunto básico de etapas, como build, testes e deploy.

```
name: CI/CD Pipeline

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout source code
        uses: actions/checkout@v3

      - name: Set up Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '14'

      - name: Install dependencies
        run: npm install

      - name: Build the application
        run: npm run build
```

```
    - name: Upload build artifacts
      uses: actions/upload-artifact@v3
      with:
        name: build
        path: dist/

  test:
    runs-on: ubuntu-latest

    needs: build

    steps:
    - name: Checkout source code
      uses: actions/checkout@v3

    - name: Set up Node.js
      uses: actions/setup-node@v3
      with:
        node-version: '14'

    - name: Install dependencies
      run: npm install

    - name: Run tests
      run: npm run test
      env:
        CI: true

    - name: Upload test results
      if: always()
      uses: actions/upload-artifact@v3
      with:
        name: test-results
        path: test-results.xml

  deploy:
    runs-on: ubuntu-latest

    needs: test

    if: github.ref == 'refs/heads/main'

    steps:
    - name: Checkout source code
      uses: actions/checkout@v3

    - name: Deploy to production
      run: ./deploy.sh
```

```
    env:
      DEPLOY_ENV: production
```

Estrutura e Explicação

- **Nome do Pipeline:**
  - `name: CI/CD Pipeline`: Nomeia o pipeline para fácil identificação no GitHub Actions.
- **Eventos que Disparam o Pipeline:**
  - `on`: Especifica quando o pipeline deve ser acionado.
  - `push`: O pipeline é disparado quando há um push para a branch `main`.
  - `pull_request`: Também é disparado ao abrir ou atualizar um pull request na branch `main`.
- **Jobs:**
  - **Build:**
    - `runs-on: ubuntu-latest`: Define o ambiente de execução como uma máquina virtual Ubuntu.
    - Passos:
      - `actions/checkout@v3`: Faz checkout do código-fonte do repositório.
      - `actions/setup-node@v3`: Configura o Node.js na versão especificada (neste caso, 14).
      - `npm install`: Instala as dependências do projeto.
      - `npm run build`: Compila a aplicação.
      - `actions/upload-artifact@v3`: Faz upload dos artefatos gerados no diretório `dist/`.
  - **Test:**
    - `needs: build`: Depende do job de build ser executado com sucesso antes de rodar.
    - Passos:
      - Parecidos com os do job de build, mas focam em rodar os testes com `npm run test`.
      - `env: CI: true`: Configura a variável de ambiente para rodar testes em modo contínuo.
      - Upload dos resultados dos testes como artefato.
  - **Deploy:**
    - `needs: test`: Só roda se os testes passarem.
    - `if: github.ref == 'refs/heads/main'`: Só é executado se a branch for `main`.
    - Passos:
      - Faz o checkout do código e executa o script de deploy (`./deploy.sh`), que é responsável por implantar a aplicação em produção.

Informações Importantes

- **Controlando o Fluxo com needs:** O uso de `needs` entre jobs permite criar dependências, garantindo que certos jobs rodem somente se outros forem concluídos com sucesso.
- **Condicionais com if:** `if: github.ref == 'refs/heads/main'` permite que o job de deploy seja executado apenas quando os commits são feitos na branch principal, ajudando a proteger contra deploys acidentais de branches de feature.
- **Uso de Artefatos:** Artefatos são usados para armazenar os resultados do build e testes. Eles podem ser compartilhados entre jobs ou baixados para análise.
- **Configuração do Ambiente:** Usar o `actions/setup-node@v3` permite configurar o Node.js de forma fácil. Isso pode ser estendido para outras linguagens e ambientes.
- **Script de Deploy:** O script `deploy.sh` é chamado durante o deploy. Este script pode conter qualquer lógica necessária para a implantação da aplicação, como transferir arquivos para um servidor ou acionar uma API de deploy.

Esse exemplo cobre um cenário típico de CI/CD para aplicações Node.js, mas pode ser adaptado para outras linguagens e frameworks conforme necessário.