Apostila 7: Banco de Dados PostgreSQL e Deploy no Render

Objetivos de Aprendizagem

Ao final desta apostila, você será capaz de:

1. **Arquitetura de Produção:** Compreender a diferença entre ambientes locais (SQLite) e de produção (PostgreSQL).
2. **Infraestrutura na Nuvem:** Criar e gerenciar um banco de dados PostgreSQL no Render.com.
3. **Segurança:** Separar configurações sensíveis do código usando Variáveis de Ambiente (.env).
4. **Configuração Django:** Adaptar o settings.py para ser dinâmico (funcionar tanto no local quanto na nuvem).
5. **Arquivos Estáticos:** Configurar o WhiteNoise para servir CSS/JS em produção.
6. **Deploy:** Publicar sua API na internet acessível globalmente.

Capítulo 1: O Fim do SQLite

1.1. Por que abandonar o SQLite?

Nas apostilas anteriores, usamos o **SQLite**. Ele é excelente para desenvolvimento porque é apenas um arquivo (db.sqlite3) no seu computador. Porém, ele não serve para produção profissional:

- **Concorrência:** Ele trava se muitos usuários tentarem escrever ao mesmo tempo.
- **Perda de Dados:** Em plataformas modernas (como Render ou Heroku), o sistema de arquivos é **efêmero**. Toda vez que você faz deploy, o arquivo db.sqlite3 é apagado e resetado.

1.2. A Nova Arquitetura

Nesta apostila, mudaremos nossa arquitetura de conexão:

1. **Ambiente Local:** Seu Django rodará no seu PC, mas se conectará a um banco PostgreSQL hospedado na nuvem (Render).
2. **Ambiente de Produção:** O Django rodará nos servidores do Render, conectando-se internamente ao mesmo banco PostgreSQL.

Capítulo 2: Criando o Banco de Dados no Render

O Render.com é uma plataforma PaaS (Platform as a Service) moderna que simplifica muito o deploy.

2.1. Passo a Passo

1. Crie uma conta em dashboard.render.com.
2. Clique no botão **New +** e selecione **PostgreSQL**.
3. Configure os detalhes:
   - **Name:** django-postgres (ou o nome do seu projeto)
   - **Database:** postgres (padrão)
   - **User:** postgres (padrão)
   - **Region:** Escolha a mais próxima (ex: Ohio ou Frankfurt) - *Importante: O Web Service deve ficar na mesma região.*
   - **Instance Type:** Free (Starter)
4. Clique em **Create Database**.

2.2. Obtendo as Credenciais

Após criar, o Render exibirá duas URLs de conexão. É crucial entender a diferença:

- **Internal Database URL:** Usada APENAS dentro da rede do Render (quando seu site estiver no ar).
- **External Database URL:** Usada para conectar do SEU COMPUTADOR ao banco na nuvem.

  Copie a "External Database URL" para usarmos no próximo passo. Ela se parece com:
  postgres://usuario:senha@host-oregon.render.com/nome_banco_123

Capítulo 3: Preparando o Projeto Local

Vamos instalar as ferramentas necessárias para o Django conversar com o PostgreSQL e ler variáveis de ambiente.

3.1. Instalando Dependências

No seu terminal, com o ambiente virtual ativado:

```
pip install psycopg2-binary dj-database-url python-dotenv whitenoise gunicorn
```

O que cada pacote faz?

- psycopg2-binary: O driver que permite o Python falar com o PostgreSQL.
- dj-database-url: Traduz a string de conexão (URL do banco) para o formato de dicionário DATABASES do Django.
- python-dotenv: Lê arquivos .env localmente.
- whitenoise: Serve arquivos estáticos (CSS, imagens do admin) de forma eficiente.
- gunicorn: Servidor HTTP de produção (o runserver não aguenta tráfego real).

3.2. Congelando as Dependências

Gere o arquivo que informa ao Render o que instalar:

```
pip freeze > requirements.txt
```

Capítulo 4: Segurança com Variáveis de Ambiente (.env)

Jamais coloque senhas ou chaves secretas diretamente no código (settings.py). Vamos usar um arquivo .env que **não** será enviado para o GitHub.

4.1. Criando o arquivo .env

Na raiz do projeto (mesma pasta do manage.py), crie um arquivo chamado .env (sem nome, só a extensão):

```
# Arquivo: .env
DEBUG=True
SECRET_KEY=django-insecure-chave-super-secreta-local-123
DATABASE_URL=cole_aqui_a_EXTERNAL_database_url_do_render
ALLOWED_HOSTS=localhost,127.0.0.1
```

> **Atenção:** Substitua o valor de DATABASE_URL pela URL Externa que você copiou do painel do Render no Capítulo 2.

4.2. Ignorando no Git

Certifique-se de que o arquivo .gitignore contém:

```
.env
venv/
__pycache__/
db.sqlite3
```

Capítulo 5: Configurando o settings.py

Esta é a parte mais crítica. Vamos alterar o settings.py para ele ler as configurações do .env quando estiver local, e das variáveis do sistema quando estiver no Render.

Abra config/settings.py e altere o seguinte:

5.1. Importações Iniciais

No topo do arquivo, adicione:

```
from pathlib import Path
import os
import dj_database_url
from dotenv import load_dotenv
```

5.2. Carregamento do .env

Logo abaixo de BASE_DIR, adicione a lógica para carregar o arquivo .env apenas se ele existir (desenvolvimento local):

```
BASE_DIR = Path(__file__).resolve().parent.parent

# Carrega variáveis de ambiente do arquivo .env
load_dotenv(BASE_DIR / ".env")
```

5.3. Chaves e Debug

Substitua as configurações antigas hardcoded por:

```
# Lê do ambiente. Se não achar, usa uma chave insegura (fallback)
SECRET_KEY = os.getenv("SECRET_KEY", "chave-insegura-fallback")

# Lê do ambiente. Retorna 'True' se o valor for "True", senão False.
DEBUG = os.getenv("DEBUG", "False") == "True"

# Hosts permitidos
ALLOWED_HOSTS = os.getenv("ALLOWED_HOSTS", "").split(",")

# O Render define a variável RENDER_EXTERNAL_HOSTNAME automaticamente
render_host = os.getenv("RENDER_EXTERNAL_HOSTNAME")
if render_host:
    ALLOWED_HOSTS.append(render_host)
```

5.4. Banco de Dados (PostgreSQL)

Substitua todo o bloco DATABASES antigo (SQLite) por este:

```
DATABASES = {
    "default": dj_database_url.config(
        default=os.getenv("DATABASE_URL"),
        conn_max_age=600,
        ssl_require=True, # Importante para Render
    )
}
```

5.5. Arquivos Estáticos (WhiteNoise)

O Django não serve arquivos estáticos (CSS/JS) em produção por padrão. O WhiteNoise resolve isso.

1. Adicione ao MIDDLEWARE (logo após SecurityMiddleware):

```
MIDDLEWARE = [
    "django.middleware.security.SecurityMiddleware",
    "whitenoise.middleware.WhiteNoiseMiddleware", # <--- ADICIONE AQUI
    "django.contrib.sessions.middleware.SessionMiddleware",
    # ...
]
```

2. Configure os caminhos no final do arquivo:

```
STATIC_URL = "/static/"
STATIC_ROOT = BASE_DIR / "staticfiles"

# Compactação e cacheamento otimizado
STATICFILES_STORAGE = "whitenoise.storage.CompressedManifestStaticFilesStorage"
```

Capítulo 6: Testando a Conexão Local

Agora vem a mágica. Vamos rodar as migrações. Como configuramos a DATABASE_URL no .env apontando para o Render, essas tabelas serão criadas **na nuvem**, mesmo rodando o comando do seu PC.

1. Rode as migrações:
   python manage.py migrate

   Se demorar um pouco mais que o normal, é porque está conectando na internet.
2. Crie um superusuário (agora ele existirá no PostgreSQL):
   python manage.py createsuperuser

3. Rode o servidor localmente:
   python manage.py runserver

Acesse http://127.0.0.1:8000/admin. Se você conseguir logar, parabéns! Seu Django local está salvando dados no PostgreSQL do Render.

Capítulo 7: Deploy do Web Service

Agora que o banco está pronto, vamos colocar o código Python (Django) para rodar no servidor do Render.

7.1. Preparar o Repositório

Garanta que todo o seu código (incluindo requirements.txt atualizado e settings.py modificado) foi enviado para o GitHub.

```
git add .
git commit -m "Configuração de produção para Render"
git push origin main
```

7.2. Criar Web Service no Render

1. No dashboard do Render, clique em **New +** -> **Web Service**.
2. Conecte sua conta do GitHub e selecione o repositório do projeto.
3. **Configurações:**
   - **Name:** minha-api-django
   - **Region:** A MESMA região onde você criou o Banco de Dados.
   - **Branch:** main
   - **Runtime:** Python 3
   - **Build Command:**
     pip install -r requirements.txt && python manage.py collectstatic --noinput && python manage.py migrate

     (Isso instala libs, prepara estáticos e roda migrações a cada deploy)
   - **Start Command:**
     gunicorn config.wsgi:application

7.3. Variáveis de Ambiente no Render (Environment Variables)

Role para baixo até a seção **Environment Variables**. Aqui não usamos arquivo .env, definimos chave-valor na interface. Adicione:

Key	Value
DEBUG	False
SECRET_KEY	uma-chave-aleatoria-muito-segura-e-longa-para-prod
WEB_CONCURRENCY	4
DATABASE_URL	**Use a Internal Database URL** (copie do painel do banco)

> **Dica Pro:** Ao usar a *Internal Database URL*, a conexão é ultrarrápida e não passa pela internet pública.

7.4. Finalizar

Clique em **Create Web Service**.

O Render iniciará o processo de build. Você verá logs no console:

1. Clonando repo...
2. Instalando requirements.txt...
3. collectstatic...
4. migrate...
5. Iniciando Gunicorn...

Se aparecer **"Live"** em verde, sua API está no ar! 🚀

Resumo da Apostila

O que aprendemos:

- ☑ **Arquitetura Híbrida:** Desenvolver localmente conectado a banco na nuvem.
- ☑ **Segurança:** Uso de .env (local) e Environment Variables (Render) para proteger credenciais.
- ☑ **Settings Dinâmico:** Um único settings.py que se adapta ao ambiente.
- ☑ **WhiteNoise:** Servir arquivos estáticos sem precisar de S3 ou CDN complexo.
- ☑ **Deploy:** API rodando em HTTPS com certificado SSL automático (fornecido pelo Render).

Próximos Passos (Opcionais)

Agora que sua API está online, você entrou no mundo DevOps. Futuramente, você pode explorar:

- **CI/CD:** Rodar testes automatizados (da Apostila 6) antes de cada deploy.
- **Docker:** Criar containers para garantir que o ambiente local seja idêntico ao de produção.
- **Monitoramento:** Usar ferramentas como Sentry para capturar erros em tempo real.

Parabéns! Você concluiu a série de Apostilas de Django REST Framework. Você saiu do "Hello World" e agora tem uma API Autenticada, Testada e Publicada em Produção.