UNIVERSIDADE FEDERAL DE MATO GROSSO DO SUL
Curso de Sistemas de Informação

JOSÉ DIEGO DALCECO MARTINS BONILHA

PUBLICAÇÃO DE APLICAÇÕES EM PYTHON COM DJANGO

Ponta Porã, MS
2025

JOSÉ DIEGO DALCECO MARTINS BONILHA

PUBLICAÇÃO DE APLICAÇÕES EM PYTHON COM DJANGO

Trabalho de Conclusão de Curso apresentado à Universidade Federal de Mato Grosso do Sul como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação.
Orientador: Profº Dr. Robson Soares Silva

Ponta Porã, MS
2025

DEDICATÓRIA

À minha tia e à minha irmã que me deram toda a base necessária. Aos amigos Alexandre, Aline, Antônio, Jaime e Kethelyn que me apoiaram durante todo esse processo.

AGRADECIMENTOS

Agradeço ao meu orientador, Prof. Dr. Robson Soares Silva, pela dedicação, orientação valiosa e apoio constante durante toda a elaboração deste trabalho. Sua expertise, paciência e comprometimento foram essenciais para a conclusão desta pesquisa.

RESUMO

Este trabalho aborda o processo de *deploy* (implantação) de aplicações desenvolvidas com o *framework* Django, destacando sua importância crítica para programadores iniciantes na construção de portfólios profissionais eficazes. A pesquisa parte da premissa de que, no competitivo mercado de tecnologia, a demonstração prática de habilidades por meio de projetos publicados e acessíveis online é um fator determinante para a inserção e progressão na carreira. Entretanto, muitos estudantes e desenvolvedores em início de carreira enfrentam uma barreira técnica significativa ao tentar disponibilizar suas aplicações na web. Entre os desafios mais comuns estão a configuração de servidores web, a gestão de bancos de dados em ambiente de produção, o gerenciamento de dependências e ambientes virtuais, e a implementação de medidas essenciais de segurança, como o uso de variáveis de ambiente para dados sensíveis e a disponibilidade de certificados SSL.

A metodologia adotada inclui o desenvolvimento de uma aplicação Django completa, servindo como estudo de caso, e sua implantação detalhada em diferentes plataformas de hospedagem, com foco no serviço de *Platform as a Service* (PaaS) Heroku, que simplifica a gestão de infraestrutura. Cada etapa do processo será documentada de forma a criar um guia prático e replicável. Os resultados evidenciam as principais dificuldades encontradas por iniciantes, como a gestão de arquivos estáticos, e propõem soluções claras e boas práticas para superá-las.

Conclui-se que o domínio do processo de *deploy* não é apenas uma habilidade técnica secundária, mas uma etapa que transforma um projeto local em uma prova de competência real, servindo como um diferencial competitivo no mercado de trabalho.

Palavras-chave: Python, Django, Mercado de Trabalho, Iniciantes, Portfólio.

ABSTRACT

This study examines the deployment process of applications developed with the Django framework, emphasizing its critical importance for beginner programmers in building effective professional portfolios. The research is based on the premise that in the competitive technology market, the practical demonstration of skills through published and accessible online projects is a decisive factor for career entry and advancement. However, many students and early-career developers face significant technical barriers when attempting to make their applications available on the web. Common challenges include configuring web servers, managing production databases, handling dependencies and virtual environments, and implementing essential security measures, such as using environment variables for sensitive data and enabling SSL certificates.

The methodology involves the development of a complete Django application as a case study, followed by its detailed deployment on different hosting platforms, with a focus on the Platform as a Service (PaaS) Heroku, which simplifies infrastructure management. Each step of the process is documented to create a practical and replicable guide. The results highlight the main difficulties encountered by beginners, such as static file management, and propose clear solutions and best practices to overcome them.

The study concludes that mastering the deployment process is not merely a secondary technical skill but a crucial step that transforms a local project into tangible proof of competence, serving as a competitive advantage in the job market.

Keywords: Python, Django, Job Market, Beginners, Portfolio

SUMÁRIO

1 Introdução

No dinâmico setor de tecnologia da informação, possuir um portfólio de projetos pode aumentar as chances de um indivíduo se destacar entre os inúmeros candidatos. O portfólio funciona como uma vitrine viva das habilidades de um profissional, mostrando de forma concreta sua capacidade de aplicar conhecimento para construir soluções funcionais e resolver problemas do mundo real. Para um desenvolvedor júnior, que frequentemente não possui experiência profissional formal, um portfólio com projetos online e acessíveis é uma grande ferramenta para comprovar sua competência, iniciativa e capacidade de entrega.

Contudo, observa-se uma lacuna curricular significativa em muitos cursos técnicos e de graduação: a etapa de publicação de uma aplicação. Enquanto o ciclo de desenvolvimento de software é amplamente ensinado, o processo de tornar esse software público e funcional na internet — o *deploy* — é frequentemente negligenciado ou abordado de maneira superficial. Essa omissão cria uma barreira considerável para o ingresso de novos profissionais na área. Este trabalho de conclusão de curso visa preencher exatamente essa lacuna, oferecendo um guia prático e detalhado sobre como realizar o *deploy* de uma aplicação desenvolvida com Python e Django, capacitando estudantes a transformarem seus projetos em peças centrais de um portfólio profissional e competitivo.

Neste trabalho, é demonstrado como funciona na prática a publicação de aplicações com o framework Django utilizando a plataforma Heroku, escolhida por prévio contato, atentando-se às configurações específicas que podem causar dúvidas, como parâmetros de segurança e otimização de recursos.

2 Tecnologias Empregadas e Termos Utilizados

Para o processo publicação de uma aplicação web, serão necessárias algumas ferramentas, dentre elas tem-se linguagens de programação, serviços de infra-estrutura, de banco de dados e entre outros. A seguir serão apresentadas as ferramentas utilizadas para alcançar o objetivo.

Para iniciar o desenvolvimento de acordo com esse material, é necessário ter já instalado em uma máquina, o Python, na versão 3.8 ou superior, Pip, que é o gerenciador de pacotes do Python e algum editor de código.

2.1 Python

Python é uma linguagem de programação versátil e popular, conhecida pela sua sintaxe simples e fácil de ler. É usada para uma grande variedade de aplicações, incluindo desenvolvimento web, ciência de dados, machine learning e automação. Foi escolhida por ser popular, o que aumenta a abrangência desse conteúdo.

2.2 Django Framework

Em desenvolvimento de software, um framework é uma estrutura abstrata que fornece um conjunto de funcionalidades e componentes reutilizáveis para acelerar o desenvolvimento de aplicações. É como uma moldura ou base sobre a qual um projeto de software pode ser construído, oferecendo estruturas, padrões e ferramentas que resolvem problemas comuns, permitindo que os desenvolvedores se concentrem nas partes específicas do projeto.

Django é um framework Python para desenvolvimento web que prioriza a produtividade, seguindo princípios de código organizado e soluções práticas. Sua arquitetura bem estruturada e documentação detalhada facilitam o aprendizado progressivo, tornando-o uma opção valiosa tanto para desenvolvedores em formação quanto para profissionais experientes.

> Com o Django, você pode levar aplicativos web do conceito ao lançamento em questão de horas. O Django cuida de grande parte da parte trabalhosa do desenvolvimento web, para que você possa se concentrar em escrever seu aplicativo sem precisar reinventar a roda. É gratuito e de código aberto (DJANGO SOFTWARE FOUNDATION, 2025, [s. p.], tradução nossa).

2.3 Banco de Dados PostgreSQL

PostgreSQL (também chamado de Postgres) é um sistema de gerenciamento de banco de dados relacional (SGBDR) open-source, conhecido por sua confiabilidade, robustez e recursos avançados. Ele suporta consultas complexas, transações ACID (Atomicidade, Consistência, Isolamento e Durabilidade) e é altamente extensível, permitindo a adição de novos tipos de dados, funções e até mesmo linguagens de programação.

O PostgreSQL oferece diversos recursos para ajudar desenvolvedores a criar aplicativos, administradores a proteger a integridade dos dados e criar ambientes tolerantes a falhas [...] (POSTGRESQL GLOBAL DEVELOPMENT GROUP, 2025, [s. p.], tradução nossa).

2.4 CRUD (Create, Read, Update, Delete)

Acrônimo em inglês para, respectivamente, criar, ler, atualizar e excluir as quais são as operações básicas feitas com dado recurso.

3 Arquitetura do Django

Segundo Pinkham, “a estrutura do Django é frequentemente descrita com base na arquitetura Model-View-Controller (MVC) porque isso facilita o aprendizado do framework” (PINKHAM, 2016, p. 8, tradução nossa). Essa abordagem sugere a organização do software dividida entre as seguintes camadas:

3.1 Model (Modelo)

Controla a organização e o armazenamento dos dados e pode também definir comportamentos específicos relacionados a esses dados. (Exemplo: Definição de tabelas no banco de dados, validações, regras de negócio.)

3.2 View (Visualização)

Controla como os dados são exibidos e gera a saída que será apresentada ao usuário. *(Exemplo: Templates HTML que renderizam posts de um blog.)*

3.3 Controller (Controlador)

Age como intermediário entre o Model e a View (e o Usuário). O Controller determina o que o usuário deseja, retorna os dados solicitados e pode selecionar quais dados buscar no Model e usar a View para formatar os dados para visualização.

O Django, especialmente, usa outra nomenclatura quanto a essa arquitetura. Aqui, a camada de Controller se chama View e a camada View se chama Template, sendo comumente descrita como MVT, Model, View, Template. Isso não deve causar mais desentendimentos, visto que é apenas a forma como são intituladas as camadas que muda diante do modelo tradicional de MVC (**Figura 1**).

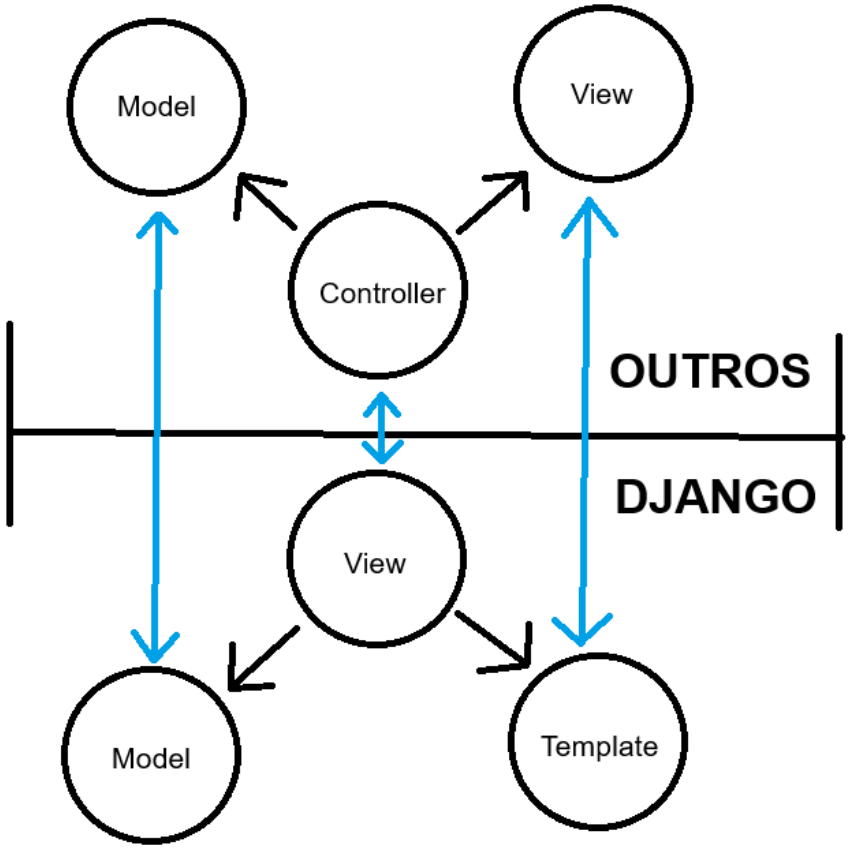

Figura 1: Comparação entre MVC e MVT
Fonte: Elaborada pelo autor (2025)

4 Preparação do Ambiente de Desenvolvimento

Para iniciar um projeto em Django em ambiente local, primeiramente é necessário configurar um ambiente virtual Python para que os pacotes de outros

projetos previamente configurados na máquina não entrem em conflito com os que serão utilizados neste. Os comandos são apresentados nas **Listagens 1 a 3**:

```
python -m venv .venv
```

Listagem 1: Comando para criação de um ambiente virtual
Fonte: Elaborada pelo autor (2025)

```
.venv\Scripts\activate.bat
```

Listagem 2: Comando para ativação de um ambiente virtual em ambiente Windows
Fonte: Elaborada pelo autor (2025)

```
source .venv/bin/activate
```

Listagem 3: Comando para ativação de um ambiente virtual em ambiente Linux/macOS
Fonte: Elaborada pelo autor (2025)

Após a ativação, será possível ver, precedendo a linha de comando, o texto "(venv)", indicando que os comandos que sucedem, serão executados nesse ambiente virtual (**Listagem 1**).

```
C:\Users\user\projeto> python -m venv .venv
C:\Users\user\projeto> .\venv\Scripts\activate.bat
(venv) C:\Users\user\pasta_do_projeto>
```

Listagem 4: Criação e ativação do ambiente virtual Python
Fonte: Elaborada pelo autor (2025)

Agora que se tem um ambiente isolado para o novo projeto, é preciso instalar o Django. Isso se faz utilizando o Pip, o gerenciador de pacotes do Python com o comando da **Listagem 5**.

```
pip install django
```

Listagem 5: Comando para instalar o Django com o gerenciador de pacotes Pip
Fonte: Elaborada pelo autor (2025)

5 Projeto de Exemplo

Para este caso, será desenvolvida uma aplicação de gerenciamento de usuários, com as principais funcionalidades sendo cadastro, listagem, edição e exclusão dos recursos. Para iniciar um projeto, usa-se o comando da **Listagem 6,** alterando o nome do projeto conforme necessidade.

```
django-admin startproject nomeDoProjeto
```

Listagem 6: Comando para criar um projeto Django
Fonte: Elaborada pelo autor (2025)

Note que foi criado um diretório com o nome do projeto, nesse caso "meuprojeto". Dentro do projeto, há novamente uma pasta com o mesmo nome do

projeto. Na pasta raiz ficam as configurações gerais do projeto. O Django trabalha orientado a aplicativos e quando se cria um, ele fica alocado na mesma pasta raiz. Com o comando da **Listagem 7** é possível criar um aplicativo e na **Figura 2** é possível ver como ficam organizados os arquivos. Nesse material, o aplicativo criado tem o nome "*crud*".

```
python manage.py startapp crud
```

Listagem 7: Comando para criar um aplicativo Django
Fonte: Elaborada pelo autor (2025)

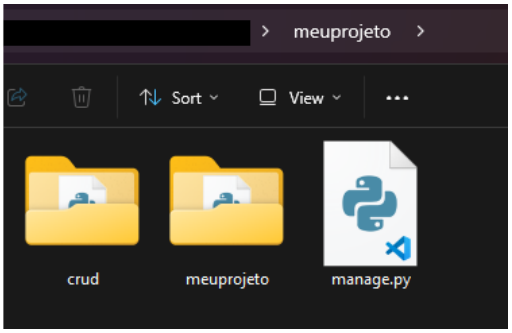

Figura 2: Organização da pasta do projeto com o aplicativo
Fonte: Elaborada pelo autor (2025)

Com o aplicativo criado, deve-se registrá-lo no projeto através do arquivo de configuração do projeto, inserindo o nome do mesmo na listagem de aplicativos instalados. Este passo é descrito na **Listagem 8**.

```
# /projeto/settings.py
INSTALLED_APPS = [
      [...]
      'crud'
]
```

Listagem 8: Registrando aplicativo no projeto Django.
Fonte: Elaborada pelo autor (2025)

Para este caso, é necessário criar a entidade do usuário. Isso é feito no arquivo de entidades do aplicativo. Uma *Model* com os atributos nome, email e data de nascimento pode ser definida de acordo com a **Listagem 9**.

```
# /projeto/aplicativo/models.py
class Usuario(models.Model):
  nome=models.CharField("Nome", max_length=100)
  email=models.CharField("E-mail", max_length=100)
  dataNascimento=models.DateField("Data de Nascimento")
```

Listagem 9: Definição da entidade "Usuário"
Fonte: Elaborada pelo autor (2025)

Django utiliza o sistema de Migrações para o versionamento da estrutura do banco de dados, assegurando que alterações nos modelos (definidos em código

Python) sejam traduzidas de forma consistente para esquemas de banco de dados. Quando um desenvolvedor modifica um modelo (ex.: adiciona um campo telefone à entidade Usuário), o Django gera automaticamente um arquivo de *migration* contendo instruções SQL para aplicar essa mudança. Esse arquivo, armazenado no projeto, documenta a evolução do esquema de forma incremental e reversível. O próximo passo é aplicar a *Migration* que irá criar a respectiva tabela no banco de dados para esse modelo. Para isso se usa o comando da **Listagem 10**.

```
python manage.py migrate
```

Listagem 10: Comando para aplicar migração com base nos modelos
Fonte: Elaborada pelo autor (2025)

Na **Listagem 11** é mostrado o código da *View*, que é responsável por definir quais dados vão ser exibidos e como serão exibidos, funcionando aqui como um controlador.

```
# /projeto/aplicativo/views.py
from django.shortcuts import render, redirect,
get_object_or_404
from .models import Usuario
from .forms import UsuarioForm

def lista_usuarios(request):
  usuarios = Usuario.objects.all()
  return render(request, 'usuarios/lista_usuarios.html',
{'usuarios': usuarios, 'titulo': 'Usuários'})

def visualizar_usuario(request, pk):
  usuario = get_object_or_404(Usuario, pk=pk)
  return render(request,
'usuarios/visualizar_usuario.html', {'usuario': usuario,
'titulo': 'Visualizar Usuário'})

def criar_usuario(request):
  if request.method == 'POST':
    form = UsuarioForm(request.POST)
    if form.is_valid():
      form.save()
      redirect('lista_usuarios')
    else:
      form = UsuarioForm()
    return render(request, 'usuarios/form_usuario.html',
{'form': form, 'titulo': 'Novo Usuário'})

def editar_usuario(request, pk):
  usuario = get_object_or_404(Usuario, pk=pk)
  if request.method == 'POST':
    form = UsuarioForm(request.POST, instance=usuario)
  if form.is_valid():
    form.save()
```

```
    return redirect('lista_usuarios')
  else:
    form = UsuarioForm(instance=usuario)
  return render(request, 'usuarios/form_usuario.html',
{'form': form, 'titulo': 'Editar Usuário'})

def excluir_usuario(request, pk):
  usuario = get_object_or_404(Usuario, pk=pk)
  if request.method == 'POST':
    usuario.delete()
  return redirect('lista_usuarios')
```

Listagem 11: Controlador de requisições para operações com um usuário
Fonte: Elaborada pelo autor (2025)

É possível observar que cada método, agrupados por "*def*" define as operações possíveis com os usuários, sendo, visualizar um em específico, listar todos, criar, editar e excluir os registros. Em alguns casos, essa view retorna a renderização de um template, que é o código HTML que de fato exibe os dados.

As templates devem ser criadas no dentro do diretório "*template*", que fica dentro da aplicação. Dentro dessa pasta, devem ser criadas as pastas de cada entidade para alocar os arquivos de visualização dentro. A **Figura 3** mostra como fica a organização desses arquivos.

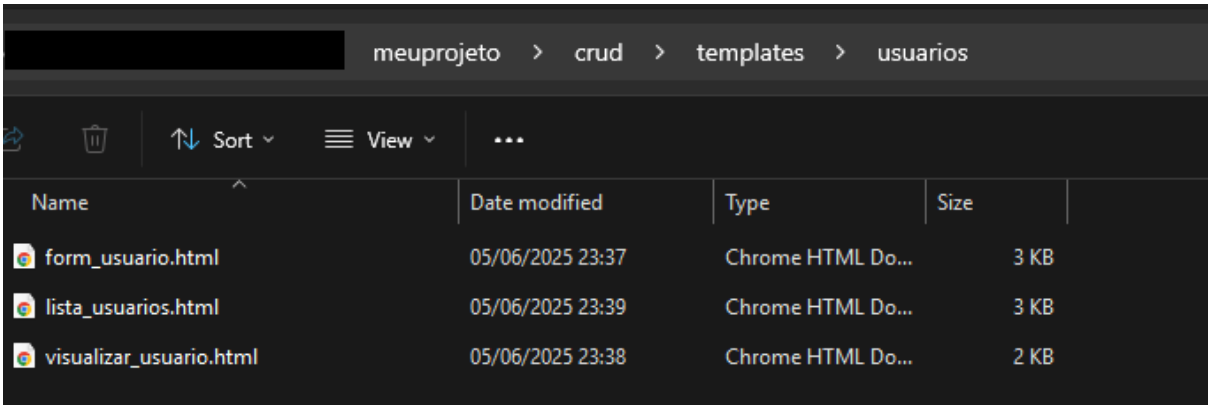

Figura 3: Organização das *templates* dentro do aplicativo Django
Fonte: Elaborada pelo autor (2025)

O Django dispõe de formas de manipular os dados recebidos da *View* na template. Para exibir o valor de uma variável usa-se "*{{variável}}*". Para utilizar de alguns recursos de laços de repetição e decisão dentro do arquivo HTML, usa-se "*{% estrutura %}*". Na **Listagem 12** há o trecho de um *template*, onde se percorre uma lista de entidades do tipo Usuário, que foi passada como terceiro argumento da função de renderização de *templates (render)* no controlador de listagem de usuários.

```
# /projeto/aplicativo/templates/entidade/template.html
<tbody>
  {% for usuario in usuarios %}
  <tr>
```

```
    <td>{{ usuario.id }}</td>
    <td>{{ usuario.nome }}</td>
    <td>{{ usuario.email }}</td>
    <td>{{ usuario.dataNascimento|date:"d/m/Y" }}</td>
  </tr>
  {% endfor %}
<\tbody>
```

Listagem 12: Manipulação de dados fornecidos pela *View* em um *Template* Django.
Fonte: Elaborada pelo autor (2025)

Para referenciar rotas em botões ou formulários dentro dos arquivos de *templates*, deve-se usar a sintaxe seguindo o exemplo da **Listagem 13**, utilizando, entre chaves e o símbolo de porcentagem, a palavra “url” seguida pelo nome da rota cercada de aspas simples.

```
<a href="{% url 'criar_usuario' %}">Criar usuário</a>
```

Listagem 13: Sintaxe de referenciação de uma rota dentro de um *template*
Fonte: Elaborada pelo autor (2025)

Em algumas *Views*, como a de edição e criação de um recurso, é preciso enviar um formulário para a *Template,* que serve para manipular de forma segura os dados a serem preenchidos. Na **Listagem 14** está a definição de um formulário.

```
# /projeto/aplicativo/forms.py
from django import forms
from .models import Usuario
class UsuarioForm(forms.ModelForm): class
  Meta:
    model = Usuario
    fields = ['nome', 'email', 'dataNascimento']
    widgets = { 'dataNascimento': forms.DateTimeInput(
attrs={'type': 'date'}, format='%Y-%m-%d' ), }
```

Listagem 14: Formulário para manipulação dos dados de Usuários
Fonte: Elaborada pelo autor (2025)

É preciso associar rotas às *Views*, ou seja, qual função do controlador será executada quando uma rota for acessada. Essa configuração está descrita na **Listagem 15**, onde são definidas as principais rotas de manipulação básica de um recurso.

```
# /projeto/aplicativo/urls.py
from django.urls import path
from . import views
urlpatterns = [
  path('usuarios/', views.lista_usuarios,
name='lista_usuarios'), path('usuarios/<int:pk>/',
views.visualizar_usuario, name='visualizar_usuario'),
  path('usuarios/novo/', views.criar_usuario,
name='criar_usuario'),
```

```
  path('usuarios/excluir/<int:pk>/',
views.excluir_usuario, name='excluir_usuario'),
  path('usuarios/editar/<int:pk>/', views.editar_usuario,
name='editar_usuario')
]
```

Listagem 15: Associação das rotas com os controladores (Views)
Fonte: Elaborada pelo autor (2025)

As configurações de rotas do aplicativo devem ser incluídas no arquivo de configuração das rotas do projeto, o código a ser inserido está na **Listagem 16**.

```
# /projeto/urls.py
urlpatterns = [
  [...]
  path('', include('crud.urls'))
]
```

Listagem 16: Inclusão das rotas da aplicação no carregador de rotas do projeto base
Fonte: Elaborada pelo autor (2025)

Com essas informações já é possível executar o projeto com o aplicativo localmente utilizando o comando da **Listagem 17**. As **Figuras 4** e **5** demonstram a aplicação sendo executada em ambiente local.

```
python manage.py runserver
```

Listagem 17: Comando para executar o projeto em ambiente local
Fonte: Elaborada pelo autor (2025)

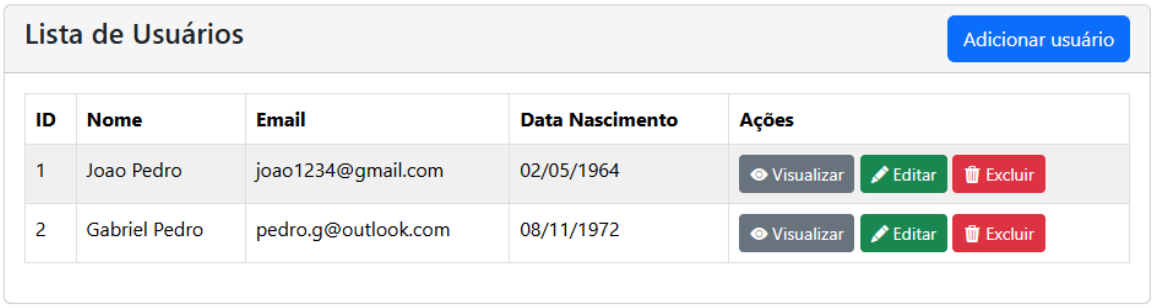

Figura 4: Listagem dos usuários na interface *web*
Fonte: Elaborada pelo autor (2025)

Figura 5: Visualização de um usuário na interface *web*
Fonte: Elaborada pelo autor (2025)

6 Heroku

6.1 Escolha

Para a publicação do projeto desenvolvido, será utilizada a plataforma Heroku, que propõe simplicidade no processo de deploy a um preço relativamente baixo.

Heroku é uma plataforma de nuvem que permite às empresas criar, entregar, monitorar e dimensionar aplicativos (SALESFORCE, 2024, [s. p.]).

É indicado utilizar do plano Dyno Eco, que desativa os serviços da aplicação quando esta não recebe requisições após um certo período de tempo para diminuir os custos. Por ser um serviço pago, deve se considerar outros serviços de PaaS (Platform as a Service), como Railway, Vercel ou Render, comparando seus preços em relação aos recursos ativos e passivos oferecidos.

Aplicações que não utilizam de operações de back-end, ou seja, são compostos apenas por HTML, CSS e Javascript e não necessitam de persistência de dados, podem ser melhores publicadas com o recurso GitHub Pages, do GitHub, que é gratuito, cabendo ao usuário validar essa possiblidade.

6.2 Configuração em Ambiente de Produção

Em alguns casos será possível notar que alguns passos mostrados no processo no ambiente local não precisarão ser executados novamente no ambiente de produção, no entanto, outros processos inéditos serão executados.

6.2.1 Criação da aplicação no Heroku

Após criar uma conta na plataforma, deve ser criada a aplicação. Este passo é demonstrado nas **Figuras 6** e **7**, onde se acessa a tela de criação de aplicação e se insere um nome.

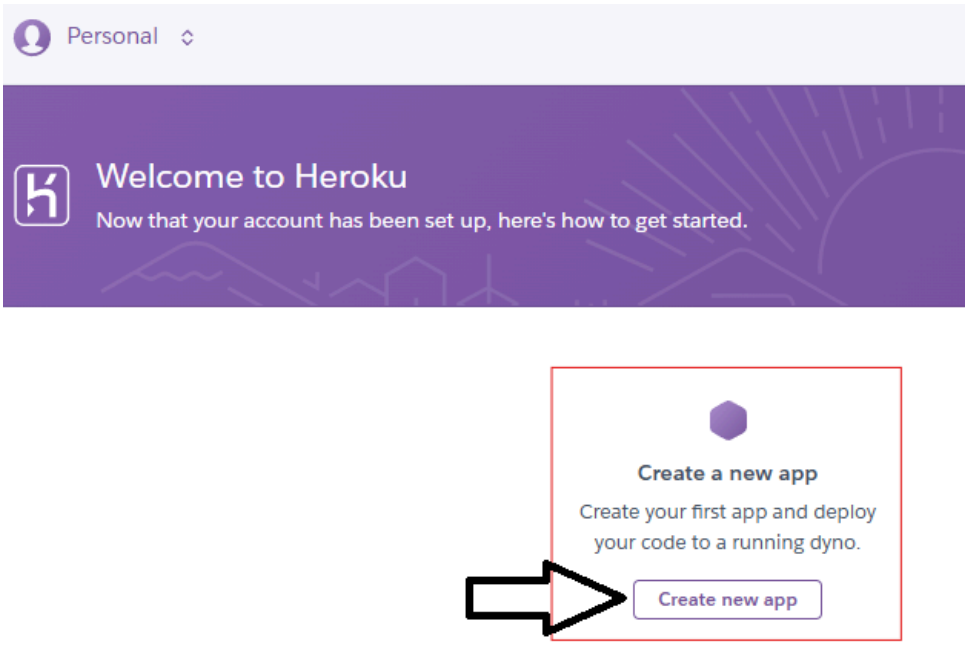

Figura 6: Tela inicial para acesso à criação de aplicações no Heroku
Fonte: Elaborada pelo autor (2025)

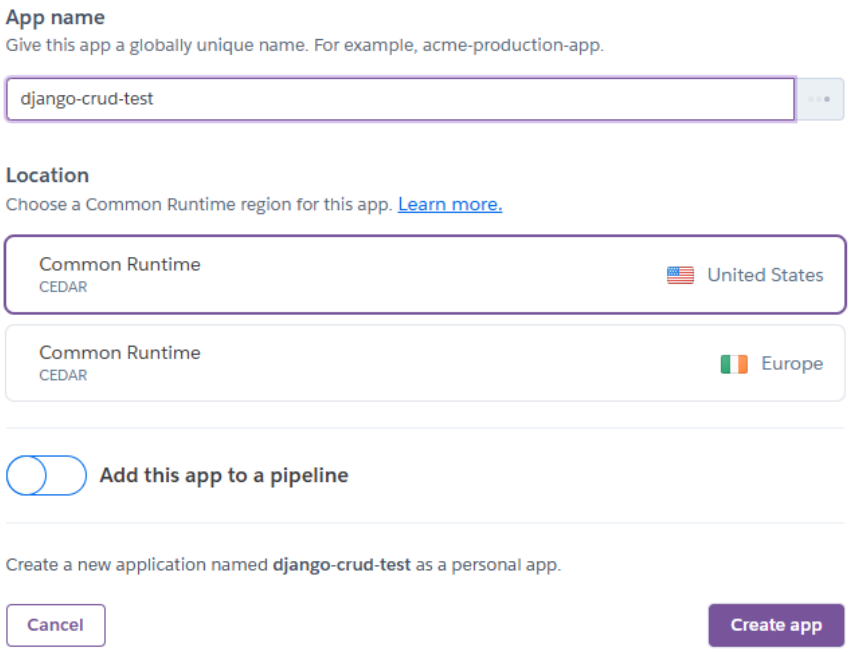

Figura 7: Dando um nome para a nova aplicação Heroku
Fonte: Elaborada pelo autor (2025)

6.2.2 Conectando repositório do GitHub

Para este processo, será utilizado o método de Deploy com GitHub. A **Figura 8** mostra como deve ser a configuração desta etapa, considerando que o código do projeto já se encontra no Github e as autorizações necessárias que o Heroku necessita para acessar o repositório no Github já foram realizadas.

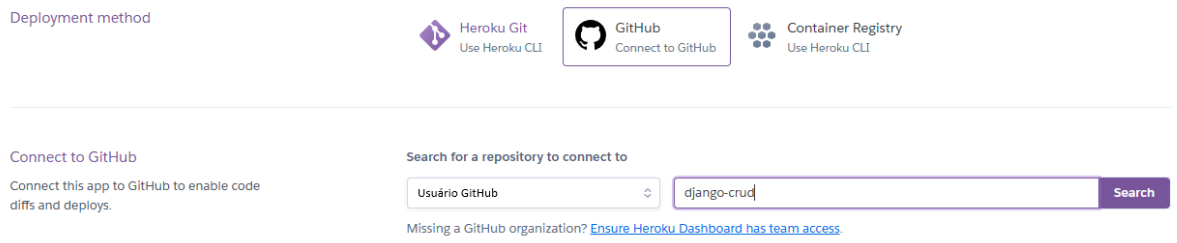

Figura 8: Conectando o repositório base hospedado no GitHub
Fonte: Elaborada pelo autor (2025)

6.2.3 Serviço de Banco de dados

Banco de dados no Heroku são um serviço à parte e devem ser configurados na aba *Add-ons*, em Recursos da aplicação (**Figura 9**). A configuração é de apenas um clique e a única demanda é verificar os preços oferecidos.

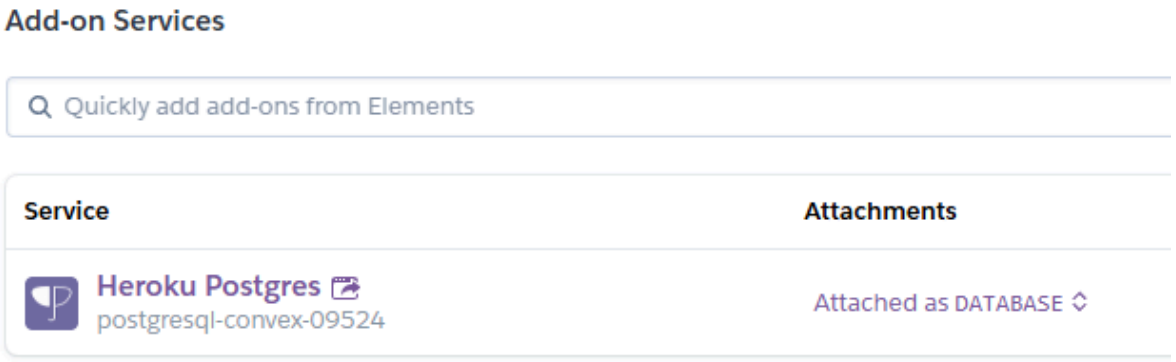

Figura 9: Heroku Postgres configurado como *Add-on*
Fonte: Elaborada pelo autor (2025)

Agora que o código já está disponível para o Heroku executar, e o serviço de banco de dados já foi adicionado à aplicação, é necessário fazer alguns ajustes no repositório de código para que o Heroku consiga publicar o projeto.

6.2.4 Arquivos estáticos

Para otimizar o carregamento de arquivos estáticos como estilização, imagens e outros recursos, em ambiente de produção será utilizada a biblioteca *WhiteNoise*, que pode ser instalada com o comando da **Listagem 17.**

```
pip install whitenoise
```

Listagem 17: Comando para instalação da biblioteca *WhiteNoise*
Fonte: Elaborada pelo autor (2025)

Deve-se incluir o *Middleware* do WhiteNoise na listagem de middlewares do projeto (**Listagem 18**).

```
# /projeto/settings.py
MIDDLEWARE = [
  [...]
  'whitenoise.middleware.WhiteNoiseMiddleware'
]
```

Listagem 18: Adição do *middleware* do pacote WhiteNoise para otimização do carregamento de arquivos estáticos na listagem de *middlewares* do projeto.
Fonte: Elaborada pelo autor (2025)

Em seguida, inserir os valores descritos na **Listagem 19** para que o pacote trabalhe corretamente.

```
# /projeto/settings.py
STATIC_URL = '/static/'
STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')
```

```
STATICFILES_STORAGE='whitenoise.storage.CompressedManifes
tStaticFilesStorage'
```

Listagem 19: Definição das configurações de arquivos estáticos utilizando WhiteNoise
Fonte: Elaborada pelo autor (2025)

6.2.5 Desativando o ambiente de testes

Uma outra chave a ser configurada no mesmo arquivo é “DEBUG”, que pode causar problemas de segurança quando em produção, devendo essa ser definida como “False”, ou seja, negativo, indicando que a aplicação não está em modo de desenvolvimento. O resultado é descrito na **Listagem 20**.

```
# /projeto/settings.py
DEBUG = False
```

Listagem 20: Definição das configurações de arquivos estáticos utilizando WhiteNoise
Fonte: Elaborada pelo autor (2025)

6.2.6 Credenciais do Banco de Dados

O banco de dados que foi adicionado como Add-on do Heroku à aplicação já exporta a variável necessária, “DATABASE_URL”, para a conexão da aplicação Django com o PostgreSQL, banco de dados utilizado aqui. No entanto para que essas variáveis de ambiente sejam lidas pelo Django, será utilizado o pacote *dj_database_url*, que decodifica as informações da variável exportada com um comando simples. Essa biblioteca pode ser instalada com o comando da **Listagem 21**. Na **Listagem 22** está definido o trecho que fornece à aplicação Django todas as informações necessárias para acessar o servidor de banco de dados, bem como, endereço, nome do banco de dados, usuário e senha.

```
$ pip install dj_database_url
```

Listagem 21: Comando para instalação da biblioteca dj-database-url, que facilita a leitura das credenciais do banco de dados
Fonte: Elaborada pelo autor (2025)

```
# /projeto/settings.py
DATABASES = {
  'default': dj_database_url.config(
    default = os.getenv('DATABASE_URL')
  )
}
```

Listagem 22: Utilização da biblioteca dj-database-url para leitura dinâmica das variáveis de conexão com o banco de dados
Fonte: Elaborada pelo autor (2025)

6.2.7. Processamento das requisições

O Gunicorn (Green Unicorn) é um servidor WSGI (Web Server Gateway Interface) projetado para executar aplicações Python em ambientes de produção. Sua principal função é atuar como intermediário entre o servidor web (como Nginx ou os recursos de roteamento do Heroku) e o framework Django, garantindo o processamento eficiente de requisições HTTP. Diferente do servidor de desenvolvimento embutido no Django (runserver), que é inadequado para produção devido a limitações de concorrência e segurança, o Gunicorn oferece balanceamento de carga, gerenciamento de processos paralelos e tolerância a falhas.

Para integrar o Gunicorn ao Heroku é necessário tê-lo adicionado no arquivo de dependências do projeto e criar o arquivo de configuração que o Heroku utilizará para iniciar o serviço. O arquivo de configuração, criado na raiz do projeto, deve conter a diretriz da **Listagem 23**, atentando-se a alterar o nome do projeto.

```
# /projeto/Procfile
web: gunicorn projeto.wsgi --log-file -
```

Listagem 23: Comando para registrar as dependências do projeto
Fonte: Elaborada pelo autor (2025)

É importante destacar que o Gunicorn opera com *workers* (processos independentes) para lidar com múltiplas requisições simultâneas. Em cenários de alto tráfego, recomenda-se ajustar o número de workers no Procfile (ex: --workers 3), considerando a memória disponível no dyno do Heroku. Adicionalmente, o Gunicorn deve ser utilizado exclusivamente em produção, mantendo-se o servidor de desenvolvimento padrão do Django para ambientes locais. Essa abordagem assegura que a aplicação Django funcione de forma otimizada e segura na plataforma Heroku, transformando código em serviços web robustos e escaláveis.

6.2.8. Bibliotecas e Dependências

Esta plataforma, Heroku, faz o deploy inteligente instalando todas as dependências necessárias para o projeto. Essas dependências devem ser registradas na raiz do projeto, no arquivo requirements.txt. Os pacotes instalados no projeto podem ser obtidos e inseridos automaticamente nesse arquivo com o comando demonstrado na **Listagem 24**. É importante considerar aqui se o projeto foi desenvolvido com ou sem o ambiente virtual isolado, pois se o comando for executado no ambiente que não no qual foi desenvolvido o projeto, o resultado do comando será diferente, fazendo com que posteriormente não seja possível identificar quais pacotes foram utilizados no projeto durante o desenvolvimento.

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

Listagem 24: Comando para registrar as dependências do projeto

Fonte: Elaborada pelo autor (2025)

6.2.9 Hosts Reconhecidos

Por questão de segurança, o endereço da aplicação Heroku deve ser registrado na lista de hospedeiros autorizados no arquivo de configuração do projeto Django. Essa configuração é definida na **Listagem 25**. O endereço da aplicação pode ser encontrado na aba de configurações da aplicação Heroku (**Figura 10**).

```
# /projeto/settings.py
ALLOWED_HOSTS = [
   'projeto-django-1a2b3b4d.herokuapp.com'
]
```

Listagem 25: Incluindo a URL da aplicação Heroku à listagem de hospedeiros autorizados
Fonte: Elaborada pelo autor (2025)

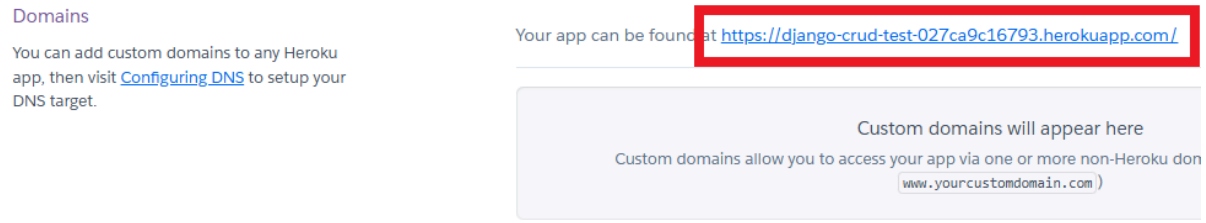

Figura 10: Disposição da URL na interface do Heroku
Fonte: Elaborada pelo autor (2025)

Adicionalmente, é preciso aplicar as migrações da aplicação Django inserindo o mesmo comando da **Listagem 10** que foi executado em ambiente de desenvolvimento, agora no console da aplicação Heroku (**Figura 10**).

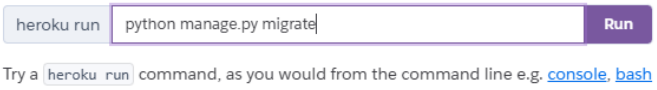

Figura 10: Execução das migrações de banco de dados dentro da interface de console da aplicação no Heroku.
Fonte: Elaborada pelo autor (2025)

6.2.10 Publicação

Agora, deve-se escolher qual ramo do GitHub será usado como base para o início do processo de Deploy. Por fim, pode-se clicar no botão de Deploy na tela inicial de detalhes da aplicação do Heroku. Dependendo do tamanho e tipo de aplicação, esse processo pode levar mais ou menos tempo. O processo de construção é detalhado em um pequeno painel na mesma tela (**Figura 12**)

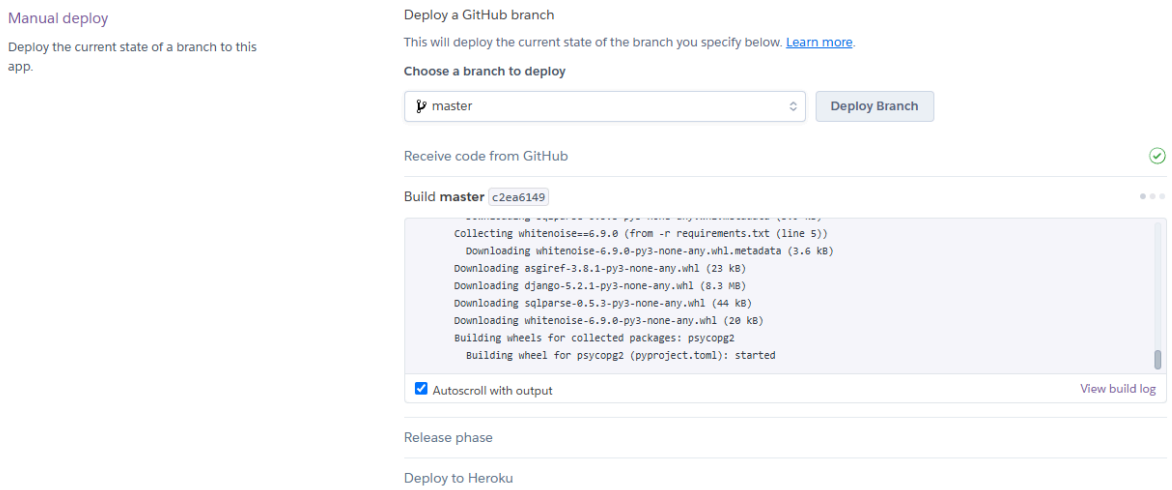

Figura 12: Processo de publicação de uma aplicação com Heroku
Fonte: Elaborada pelo autor (2025)

Ao finalizar o processo, pode-se visitar a URL da aplicação e verificar seu funcionamento na internet pública (**Figura 13**). Os registros de erros podem ser encontrados no painel inicial do Heroku.

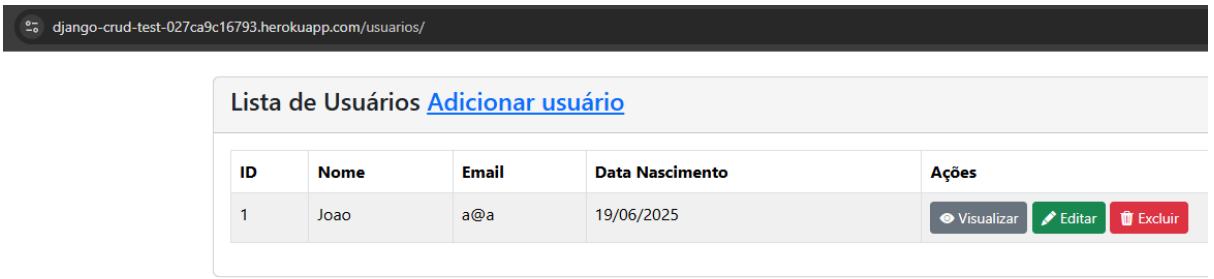

Figura 13: Aplicação Django publicada sob o domínio da plataforma Heroku
Fonte: Elaborada pelo autor (2025)

8 Conclusão

Colocar aplicativos Django em ambientes como o Heroku pode trazer algumas dificuldades técnicas, mesmo com a facilidade que a abordagem PaaS (Platform as a Service) oferece. Durante este estudo, foi possível perceber que fazer essa implantação de forma eficaz exige saber como configurar servidores, lidar com bancos de dados e garantir a segurança. Para exemplificar, foi criado um aplicativo CRUD com Django, mostrando passo a passo como publicar projetos, desde o começo até a implantação no Heroku, incluindo o gerenciamento de dependências, a configuração do banco de dados e a integração com o GitHub.

Essa análise revelou alguns obstáculos frequentes na implantação de aplicativos Django, como problemas com arquivos estáticos, configuração de hosts e atualizações de banco de dados no ambiente de produção. Resolver esses

desafios ajuda a aprimorar as habilidades técnicas cruciais para quem está começando na área. Além disso, foi observado que o plano gratuito do Heroku tem suas limitações, como a interrupção do funcionamento dos dynos e a necessidade de pagar por serviços extras para recursos mais avançados. Isso mostra como é importante considerar alternativas como Railway ou Render para projetos mais ambiciosos.

A segurança se mostrou crucial na hora de implantar, principalmente ao desativar o modo debug em produção, configurar conexões seguras e proteger informações confidenciais usando variáveis de ambiente. Essas medidas são indispensáveis para criar aplicativos web profissionais. Este estudo mostra que a implantação deve ser vista como parte fundamental do processo de criação de software, merecendo destaque em cursos de Sistemas de Informação, com a inclusão de temas sobre DevOps básico e práticas recomendadas para publicar projetos.

Saber como publicar aplicativos Django é uma habilidade técnica valiosa para quem está começando a programar. Este estudo apresentou um método organizado para publicar projetos, ajudando a formar profissionais mais preparados para o mercado. Para o futuro, sugiro explorar técnicas mais sofisticadas, como usar Docker para conteinerização, automatizar os processos de implantação e integração contínua e monitoramento de aplicativos em produção.

Referências

DJANGO SOFTWARE FOUNDATION. Documentação do Django: Versão 5.2. [S. l.], s. d. Disponível em: https://www.djangoproject.com/start/overview/. Acesso em: 18 maio 2025.

POSTGRESQL GLOBAL DEVELOPMENT GROUP. Documentação do PostgreSQL: Versão 17. [S. l.], s. d. Disponível em: https://www.postgresql.org/docs/17/. Acesso em: 22 junho 2025.

PINKHAM, Andrew. Django Unleashed. Tradução de DALCECO MARTINS BONILHA, José Diego. Ponta Porã: UFMS, 2025. p. 8. Tradução não publicada do original em inglês. ISBN-13: 978-0-321-98507-1.

SALESFORCE. Documentação do Heroku. [S. l.], s. d. Disponível em: https://www.heroku.com/what/. Acesso em: 29 maio 2025.