

You have **2** free member-only stories left this month. [Upgrade for unlimited access](#) to stories about programming and more.

[Overview] Building a Full Stack Quiz App With Django and React



Izen Oku

[Follow](#)

Sep 12, 2020 · 13 min read ★

Izen Trivia

3. Which of these is Izen allergic to?

(Choose 2.)

☐ A. Cats☐ B. Dogs☒ C. Peanuts☐ D. Crustaceans☐ E. Aspirin

<<

1

2

3

4

5

6

>>



Submit

Screenshot of quiz web application

My main goal with this application was to develop a remotely hosted quiz application, which users can choose a quiz, answer the questions, and know how he/she did at the end. **This article focuses on:** 1. building a Django backend (database & API) 2. hosting the backend and 3. hosting the frontend from a high-level understanding. If you wish for more detailed follow-alongs or tutorials, I would recommend checking out other sources.

Links to the live servers and my GitHub repository can be found at the end of this article.

Table of Content:

- [1. Building the Django backend](#)
- [2. Building the React frontend](#)
- [3. Hosting the Backend \(with Heroku\)](#)
- [4. Hosting the Frontend \(with Netlify\)](#)
- 5. Lessons Learned & Links

Building the Django Backend

I originally tried following [William Vincent's "Django Rest Framework with React Tutorial"](#) (which is a great tutorial), but knowing how I learn, I knew I needed an understanding from the ground up, as opposed to a quick follow-along.

To understand Django at a deeper level, I subscribed to [Treehouse's "Exploring Django"](#) course, which gave me a decent understanding of how Python's Object-relational Mappers work, and how Django's REST framework could be used to configure APIs. If you wish to learn Django from the ground up, I would recommend you to find an online course as well.

The Big Picture & End Goal

This section of the article tackles 2 main questions: 1. how to define the database schema, and 2. how to configure request & response (essentially the API) using Django's REST framework.

For this project, I knew I wanted to make users, and under each user there could be multiple quizzes, and under each quiz could be multiple questions, and under each question would be multiple answers.



Backend models diagram. “Foreign Key” refers to keys in a database object that points to a foreign table (e.g. each question object has a quiz key that points to a specific quiz object).

Furthermore, coming from a frontend background, I knew I needed to configure urls to fetch: 1. available quizzes for the user to choose from 2. specific questions under chosen quiz and its answer options, and potentially 3. the results on how the user did.

Defining the Database (Models)

Once I created the Django project (I named it ‘quiz_api’), and then the Django app within the project (‘quizzes’), I configured *quizzes/models.py* to reflect the model diagram. Below is an example of the ‘Quiz’ model object:

```
from django.db import models
from django.contrib.auth.models import User
```

```
class Quiz(models.Model):
    author = models.ForeignKey(User, on_delete=models.DO_NOTHING)
    title = models.CharField(max_length=255, default='')
    created_at = models.DateTimeField(auto_now_add=True)
    times_taken = models.IntegerField(default=0, editable=False)

    @property
    def question_count(self):
        ''' Method to get num of Qs for this quiz, used in Serializer'''
        return self.questions.count()

    class Meta:
        verbose_name_plural = "Quizzes"
        ordering = ['id']

    def __str__(self):
        return self.title
```

Once the models were defined, the next step for me was to figure out how to take a client request, process it, and return a response back to the client.

Using Django REST Framework to Configure Responses (the API)

What is Django REST Framework (DRF)? DRF can be thought of as a library that you can install into your Django project. We can import useful classes from DRF such as *viewsets*, *serializers*, and *routers*. Using these classes can save you time and reduce the code size.

To understand how DRF fits into the picture, first I'll explain the general request & response pattern between client & server. Below is a brief outline:

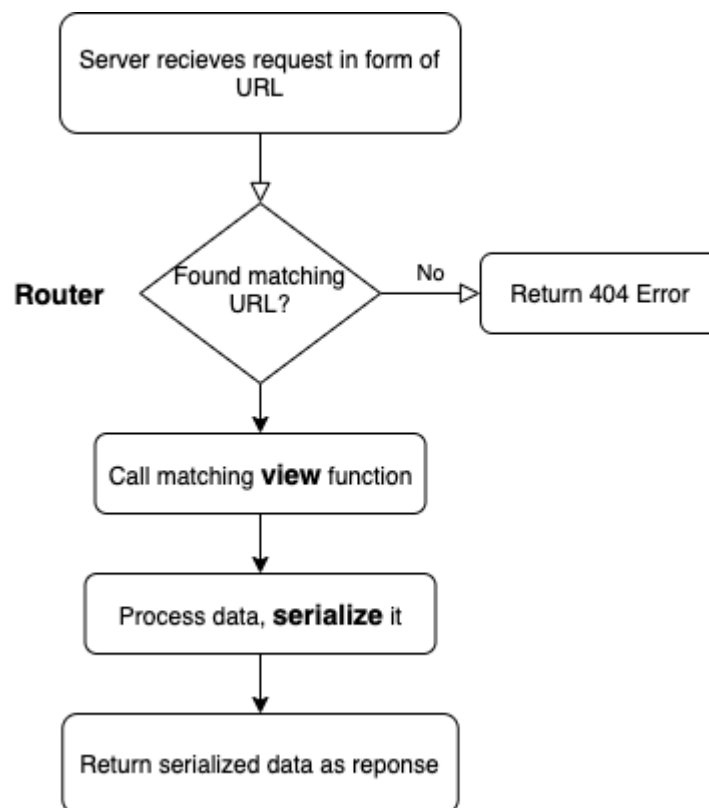


Brief outline of how client & server communication works.

As we can see from the diagram, the logic of the server can be broken down into 3 parts:

1. receive requests and look for a matching url pattern **2.** map the request to a corresponding function (to process & configure the response), and **3.** return the response. This is where DRF's routers (url patterns), view classes, and serializers come into play.

So where does DRF come into play? Specifically, how does DRF's views, serializers, and routers help the logic of the server? **Routers** map URL patterns to specific view functions, **view functions** extract the correct data and calls the **serializer** to turn the data into readable JSON format. Below I've made a brief flow chart:



Fluxograma de como o servidor lida com as solicitações.

Agora que expliquei como os conjuntos de visualizações, serializadores e roteadores do DRF estão conectados, vamos mergulhar nos detalhes e vê-los em ação!

Em primeiro lugar, uma isenção de responsabilidade: DRF's Generic Views vs ViewSets

Devo mencionar que, em meu projeto, tenho 2 conjuntos de APIs. Isso porque eu estava acompanhando o tutorial da Treehouse e exercitei 2 métodos diferentes de definição de classes de visualização.

```
class ListCreateQuiz(generics.ListCreateAPIView): ...

class RetrieveUpdateDestroyQuiz(generics.RetrieveUpdateDestroyAPIView): ...

class ListCreateQuestion(generics.ListCreateAPIView): ...

class RetrieveUpdateDestroyQuestion(generics.RetrieveUpdateDestroyAPIView): ...

class ListCreateAnswer(generics.ListCreateAPIView): ...

class RetrieveUpdateDestroyAnswer(generics.RetrieveUpdateDestroyAPIView): ...

class QuizViewSet(viewsets.ModelViewSet): ...

class QuestionViewSet(viewsets.ModelViewSet): ...

class AnswerViewSet(viewsets.ModelViewSet): ...
```

As 6 funções principais são definidas por meio do primeiro método, as 3 funções inferiores por meio do segundo. Ambos os métodos têm as mesmas funcionalidades de lista e recuperação, mas o método 2 tem menos linhas de código.

No primeiro método, criei as funções *ListCreate* <ModelName> & *RetrieveUpdateDestroy* <ModelName> para cada modelo (questionário, pergunta, resposta). Este conjunto de funções de visualização estende as subclasses *ListCreateAPIView* e *RetrieveUpdateDestroyAPIView* dos genéricos DRF, respectivamente. Conforme sugerido pelos nomes das classes, a função *ListCreate* lida com a listagem e criação do modelo especificado, whereas *RetrieveUpdateDestroy* lida com a recuperação, atualização e destruição dos modelos. Os padrões de URL para este conjunto de funções de visualização são definidos no *quizzes / urls.py*.

No segundo método, criei funções <ModelName> *ViewSet* para cada modelo. Este conjunto de funções estende da DRF *viewsets* 'subclasse *ModelViewSet*. A melhor

coisa sobre o uso de conjuntos de *visualizações* é que 1. A classe *ModelViewSet* já cobre as operações básicas de criação, leitura, atualização, exclusão (CRUD) (explicadas em detalhes abaixo) e 2. não há necessidade de configurar um arquivo *urls.py* no pasta do aplicativo. Em vez disso, importamos a classe de *roteadores* DRF no *urls.py* da pasta raiz e os URLs já estão configurados para você.

No restante do artigo, estarei me referindo aos URLs e às funções de visualização definidas no **método 2** (conjuntos de visualizações do DRF). Tendo esclarecido isso, primeiro vamos examinar mais de perto como escrever as funções de visualização.

Ver funções

Ao herdar as propriedades de *ModelViewSets*, escrever essas funções de visualização foi, na verdade, bastante simples. Tudo que eu tive que fazer foi **especificar o queryset que esta função de visão particular usará e a classe do serializador correspondente**. Vou chegar aos meus serializadores em um segundo. A função de visualização do meu questionário é mostrada abaixo:

```
class QuizViewSet ( viewsets.ModelViewSet ):  
    queryset = models.Quiz.objects.all ()  
    serializer_class = serializers.QuizSerializer  
  
    @action (detail = True, methods = ['get'])  
    def questions (self, request, pk = None):  
        questions = models.Question.objects.filter (quiz_id = pk)  
        serializer = serializers.QuestionSerializer (  
            questões,  
            muitos = True  
        )  
        resposta de retorno (serializer.data)  
  
    @action (detail = True, methods = ['get'])  
    def all_questions (self, request, pk = None):  
        questions = models.Question.objects.filter (quiz_id = pk)  
        serializer = serializers.QuestionSerializer (  
            questions,  
            many = True  
        )  
        resposta de retorno (serializer.data)
```

Posso definir **ações ad-hoc** com o decorador “@action”. Como o nome sugere, ao definir essas ações ad-hoc, posso definir padrões de URL mais específicos do que os métodos

padrão que o *ModelViewSet* fornece.

Neste cenário específico, adicionei as ações extras 'perguntas' e 'all_questions' à classe *QuizViewSet*. 'perguntas' retornará uma lista paginada de perguntas do questionário especificado. 'all_questions' retornará todas as questões do questionário especificado. A razão pela qual adicionei essas ações ad hoc é porque precisava de respostas mais específicas do que a lista padrão de perguntas. Posteriormente, explicarei como obter a lista de questionários e como usar as ações ad-hoc na seção de **roteadores** abaixo.

Serializadores

Vimos na imagem anterior que na classe *ModelViewSet*, temos que definir uma classe de serializador. Então, o que os serializadores fazem? A classe do serializador determina como o queryset recebido será moldado em um objeto JSON para a resposta. Por exemplo, defini meu serializador de questionário da seguinte forma:

```
class QuizSerializer(serializers.ModelSerializer):
    def get_fullname(self, obj): ...
    def get_qcount(self, obj):
        return obj.question_count # definimos este método em models.py

    question = serializers.HyperlinkedRelatedField(...)
    author_fullname = serializers.SerializerMethodField("get_fullname")
    question_count = serializers.SerializerMethodField("get_qcount")

class Meta:
    fields = [
        'id',
        'title',
        'author',
        'author_fullname',
        'question_count',
        'created_at',
        'questions'
    ]
    model = models.Quiz
```

A parte mais importante a se concentrar está na classe *Meta*. Aqui, definimos os **campos** e o **modelo** relacionado. *Fields* é uma matriz que especifica quais atributos do modelo especificado você deseja incluir na resposta.

No exemplo do serializador de questionário, a maioria dos atributos já está definida quando escrevemos os modelos em *quizzes / models.py* anteriormente. Os campos *question*, *author_fullname* e *question_count* são campos personalizados definidos acima da classe Meta e são atributos que não são definidos no modelo *quizzes.models.Quiz*.

Anteriormente, na definição da função de visualização *QuizViewSet*, definimos a classe *serializer* como *QuizSerializer*. Isso significa que sempre que os dados de resposta são retornados, *QuizSerializer* é chamado. Em breve, veremos o que isso significa.

Roteadores

Conforme mencionado anteriormente, usando *viewsets.ModelViewSet* do DRF em combinação com roteadores, os URLs já estão configurados para nós.

Para adicionar URLs correspondentes às funções ViewSets que definimos em *quizzes / views.py*, simplesmente importamos roteadores de *rest_framework* e instanciamos uma classe *SimpleRouter*. Em seguida, registramos os conjuntos de visualizações junto com seu padrão regex desejado (por exemplo, a URL *'api / v2 / quizzes'* chamará *views.QuizViewSet*).

```
de django.contrib import admin
de django.conf.urls import url, include
de rest_framework import routers

de questionários importar visualizações

router = routers.SimpleRouter ()
router.register (r'quizzes ', views.QuizViewSet)
router.register (r'questions', views.QuestionViewSet)
router.register (r'answers ', views.AnswerViewSet)

urlpatterns = [...]

urlpatterns += url (r '^ api / v2 /', include (( router.urls ,
'quizzes'), namespace = 'apiv2'))
```

Incluimos o roteador nos padrões de URL do projeto adicionando um método url ao array *urlpatterns* (última linha do diagrama). Uma observação a ser adicionada aqui é que, usando conjuntos de visualização e roteadores do método 2, podemos evitar a

necessidade de definir URLs no arquivo `urls.py` específico do aplicativo (por exemplo, '`quizzes/urls.py`').

Depois de registrar as URLs dos conjuntos de visualizações no roteador e adicionar `router.urls` às URLs do projeto, vamos ver em ação. Vamos tentar buscar o URL `'api / v2 /'`:

Page not found (404)

Request Method: GET

Request URL: `http://127.0.0.1:8000/api/v2/`

Using the URLconf defined in `quiz_api.urls`, Django tried these URL patterns, in this order:

1. `admin/`
2. `^api-auth/`
3. `^api/quizzes/`
4. `^api/v2/ ^quizzes/$ [name='quiz-list']`
5. `^api/v2/ ^quizzes/(?P<pk>[^/.]+)/$ [name='quiz-detail']`
6. `^api/v2/ ^quizzes/(?P<pk>[^/.]+)/all_questions/$ [name='quiz-all-questions']`
7. `^api/v2/ ^quizzes/(?P<pk>[^/.]+)/questions/$ [name='quiz-questions']`
8. `^api/v2/ ^questions/$ [name='question-list']`
9. `^api/v2/ ^questions/(?P<pk>[^/.]+)/$ [name='question-detail']`
10. `^api/v2/ ^questions/(?P<pk>[^/.]+)/answers/$ [name='question-answers']`
11. `^api/v2/ ^answers/$ [name='answer-list']`
12. `^api/v2/ ^answers/(?P<pk>[^/.]+)/$ [name='answer-detail']`

The current path, `api/v2/`, didn't match any of these.

Resposta de 'localhost: 8000 / api / v2 /'

Recebemos um “erro 404 de página não encontrada” porque não há URL configurado como `'localhost: 8000 / api / v2 /'` seguido por nada. Além dos 3 principais padrões de URL, vemos vários URLs já definidos para nós pelo DRF.

Registramos o `QuizViewSet` com um padrão regex de 'questionários' para o `SimpleRouter`, e na linha 4 há um padrão de URL `questionários/`; como o nome 'lista de questionários' sugere, ele lista todos os questionários disponíveis. Na linha 5, temos um teste de padrão de URL `/(?P <pk>)/`, que pede uma chave primária; isso recupera o questionário associado a essa chave primária.

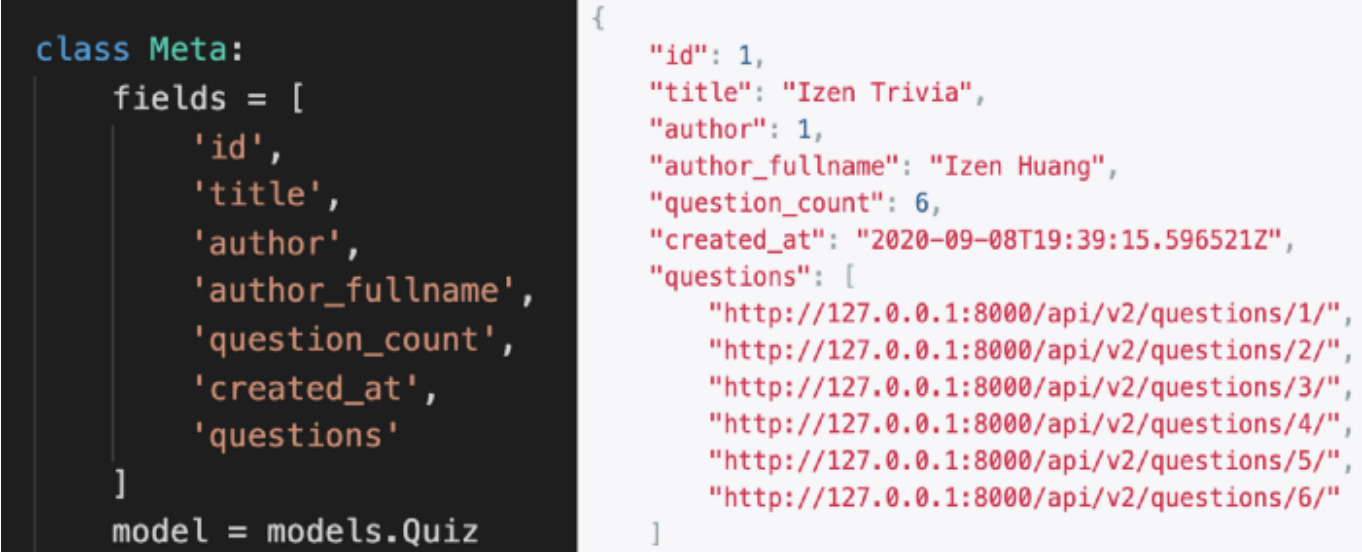
Se tivéssemos ido com o método 1 do uso da DRF *genéricos* aulas, teríamos de definir um conjunto separado de URLs para a “lista de todos os quizzes” e “recuperar um questionário específico” em `quizzes/urls.py`.

Você se lembra das ações ad-hoc 'perguntas' e 'all_questions' que definimos em QuizViewSet anteriormente ? O SimpleRouter do DRF registrou automaticamente essas ações como uma correspondência de URL válida nas linhas 6 e 7. Se especificarmos uma chave primária para um questionário com padrão de URL 6 (questionário-todas-perguntas), receberemos a lista de todas as perguntas relacionadas àquele específico questionário. Este não é um URL padrão que vem com *ModelViewSet*, mas sim uma ação definida pelo usuário que o DRF adicionou automaticamente à lista de URLs válidos para a solicitação a ser correspondida.

Embrulhando tudo junto

Os roteadores definem a quais URLs a solicitação pode ser correspondida. Isso cuida da primeira camada da lógica do servidor. Se um URL correspondente for encontrado, uma **função de visualização** correspondente será chamada. Em nosso caso, usamos *viewsets.ModelViewSet* herdados, portanto, tudo o que tivemos que fazer foi definir o queryset e a classe do serializador. O **serializer** define o objeto JSON que será incluído no corpo da resposta.

Prometi mostrar o serializador em ação mais cedo. Para ver isso em vigor, vamos chamar um URL válido (número 5 'detalhe do questionário') e ver o que obtemos:



```
class Meta:
    fields = [
        'id',
        'title',
        'author',
        'author_fullname',
        'question_count',
        'created_at',
        'questions'
    ]
    model = models.Quiz

{
  "id": 1,
  "title": "Izen Trivia",
  "author": 1,
  "author_fullname": "Izen Huang",
  "question_count": 6,
  "created_at": "2020-09-08T19:39:15.596521Z",
  "questions": [
    "http://127.0.0.1:8000/api/v2/questions/1/",
    "http://127.0.0.1:8000/api/v2/questions/2/",
    "http://127.0.0.1:8000/api/v2/questions/3/",
    "http://127.0.0.1:8000/api/v2/questions/4/",
    "http://127.0.0.1:8000/api/v2/questions/5/",
    "http://127.0.0.1:8000/api/v2/questions/6/"
  ]
}
```

Uma comparação lado a lado dos campos Meta do QuizSerializer e o primeiro questionário no corpo da resposta de 'api / v2 / quizzes /'

Como podemos ver, os campos que especificamos no QuizSerializer definem o objeto JSON retornado na resposta.

Então aí está - roteadores, funções de visualização e serializadores trabalham juntos para processar a solicitação do cliente, extrair os dados necessários do modelo de banco de dados e retornar os dados configurados como a resposta.

Construindo a interface do React

Tendo tido experiência anterior na construção de projetos React, já sabia como estruturar o frontend. Para obter um guia rápido sobre como iniciar seu primeiro aplicativo React, você pode seguir [este tutorial](#) .

No entanto, havia algumas partes sobre as quais eu não tinha certeza, especificamente relacionadas com a busca de dados do meu back-end e como lidar com a natureza assíncrona da solicitação / resposta HTTP. Para este projeto, eu queria aprender: 1. como buscar dados de APIs de terceiros 2. como configurar funções de busca assíncrona e 3. permitir compartilhamento de recursos de origem cruzada (CORS).

Buscando respostas

Desde que eu só estava construindo uma aplicação de pequena escala, I decidi centralizar meus buscar métodos em *app.js* . Especificamente, nos *componentDidMount* e *componentDidUpdate* funções. Em retrospecto, eu colocaria as funções de busca nas funções *componentDidMount ()* ou *componentWillMount ()* de qualquer componente em que os dados de resposta deveriam residir.

Devido à natureza assíncrona das solicitações / respostas HTTP, você deve agrupar qualquer função que está sendo buscada com uma declaração **assíncrona** e o operador **await** sempre que estiver lidando com um objeto de promessa . Para saber mais sobre como lidar com funções assíncronas, clique neste [link](#) .

Aqui está um exemplo de uma função *fetch*, *fetchQuizzes ()*, que busca a lista de questionários disponíveis no banco de dados:

```
async fetchQuizzes () {
  const {url_header} = this.state
  // ao executar back-end localmente, 'url_header' pode ser
  // localhost: 8000

  try {
    const res = Await buscar ( `${url_header} / questionários /`)
    const quizzes_body = Await res.json ();
```

```
if (quizzes_body! == undefined) {
  this.setState ({
    isFetchingQuizzes: false,
    quizzes: quizzes_body,
  })
}
} catch (error) {
  console.log (error)
}
}
```

Depois que o objeto de promessa é retornado, podemos analisar a resposta HTTP e obter as informações do corpo com a *resposta* .json (). Em seguida, defino os dados de resposta para um *questionário* de variável de estado com *this.setState*, que é então passado para os componentes inferiores.

Permitindo CORS

Você pode ter enfrentado um problema de tempo limite ao tentar buscar dados de seu back-end hospedado localmente. O principal culpado é porque você não permitiu o CORS. Existem algumas maneiras de contornar esse problema.

Uma maneira é instalar a extensão Allow CORS: Access-Control-Allow-Origin do Google Chrome. No entanto, isso deve ser executado apenas durante o desenvolvimento, não na produção (o eventual site hospedado remotamente).

Outra forma é configurar os cabeçalhos de suas solicitações no frontend para sempre incluir “Access-Control-Allow-Origin = '*'”. Eu brinquei com essa solução depois de configurar o servidor front-end, mas sem sucesso.

Eventualmente, o que funcionou para mim foi **permitir o CORS do back-end** . Para permitir o CORS do Django, você precisa 1. instalar o aplicativo *django-cors-headers* e 2. configurar a lista branca do CORS em *settings.py*.

As instruções para instalar o aplicativo *django-cors-headers* , e adicionar os aplicativos e middlewares necessários, podem ser encontradas na [documentação django-cors-headers](#) do pacote de índice do Python .

Para configurar a lista de permissões do CORS, navegue até *settings.py* e adicione a seguinte configuração:

```
# CORS
CORS_ORIGIN_ALLOW_ALL = False

CORS_ORIGIN_WHITELIST = [
    'http: // localhost: 8000',
    'http: // localhost: 3000', # frontend hospedado localmente
    'https://izens-quiz.netlify.app' # frontend hospedado remotamente
]
```

Você deve adicionar quaisquer sites com os quais deseja compartilhar seus dados de back-end em `CORS_ORIGIN_WHITELIST`. Adicionando 'localhost: 3000', posso executar o front-end localmente e ainda receber os dados do back-end. Eventualmente, vou querer colocar o site de front-end funcional na lista também ('izens-quiz.netlify.app')

Se você já estiver hospedando o back-end em um servidor remoto, lembre-se de confirmar as alterações e enviar para o site de back-end em funcionamento (as instruções para fazer isso estão detalhadas abaixo).

Hospedando o Backend

Configurando o servidor e banco de dados

Hospedar um back-end Django no **Heroku** é relativamente simples. Você precisa configurar um ambiente virtual, inicializar um repositório Heroku dentro desse ambiente virtual, adicionar o Procfile & requirements.txt e, finalmente, enviar para o branch master do Heroku.

Além disso, você também deve **mudar o banco de dados de SQLite3 para PostgreSQL**. Isso porque até agora salvamos nossos dados em um armazenamento baseado em disco na pasta local SQLite3 que foi criada automaticamente durante o desenvolvimento do Django. No entanto, devido à natureza efêmera dos motores de contêiner do Heroku, todo o armazenamento baseado em disco será limpo pelo menos uma vez a cada 24 horas. Para corrigir isso, moveremos o banco de dados para um servidor hospedado remotamente (por exemplo, uma instância AWS EC2).

Para saber como fazer as etapas mencionadas acima, siga o **tutorial passo-a-passo** sobre [este vídeo Youtube](#).

Adicionando arquivos CSS estáticos à implantação

Finalmente, se você deseja manter o estilo da página administrativa do Django ou estrutura REST quando executado no Heroku, você precisa **instalar o middleware WhiteNoise para que o Heroku possa hospedar arquivos css estáticos** . Para fazer isso, inicie seu ambiente virtual e siga estas etapas:

1. Instale WhiteNoise com pip (ou pip3)

```
pip3 instalar ruído branco
```

2. Adicione ruído branco ao MIDDLEWARE em *settings.py* da pasta do projeto

```
MIDDLEWARE = [  
    'whitenoise.middleware.WhiteNoiseMiddleware',  
    ...  
]
```

3. Configure as rotas estáticas em *settings.py*

```
STATIC_URL = '/ static /'  
STATIC_ROOT = os.path.join (BASE_DIR, 'staticfiles')  
STATIC_FILES_DIR = (  
    os.path.join (BASE_DIR, 'static'),  
)
```

4. Faça commit das mudanças e envie para o branch master do Heroku

```
git add.  
git commit -m "feat: whitenoise & static path"  
git push Heroku Master
```

Hospedando o Frontend

Implantando no Netlify

Eu escolhi hospedar o front-end com o Netlify, que geralmente é para *sites estáticos* (ou seja, sites que não buscam e postam dados de / para um servidor remoto). Parece funcionar bem com meu aplicativo de pequena escala, no entanto, pode não ser verdade quando o tamanho do projeto aumentar.

O processo foi relativamente simples. Como minhas pastas de back-end e front-end residiam em um repositório GitHub existente, eu simplesmente tive que conectar esse repositório a um projeto Netlify e definir as configurações de compilação.

Build settings

Repository:	github.com/izenennn/udemy-quiz-izen
Base directory:	frontend
Build command:	yarn build
Publish directory:	frontend/build

Crie configurações para minha implantação de front-end (usando Netlify)

Como meu repositório contém o diretório de backend e frontend, e estou hospedando o frontend no Netlify, especifico o **diretório base** como 'frontend'. O **comando build**, 'yarn build', é um atalho para chamar o comando 'react-scripts build'. Este comando faz um diretório 'construir' dentro do meu diretório de front-end e agrupa corretamente o React em produção para melhor desempenho. O diretório de construção criado é o que o Netlify precisa para o '**diretório de publicação**'.

Para obter instruções mais detalhadas, você pode visitar a documentação do Netlify "[Introdução ao CLI do Netlify](#)".

Lições aprendidas e links

Este projeto me ensinou muito. Eu sabia como o padrão de solicitação / resposta HTTP funciona, mas nunca o havia codificado. Há muitos detalhes que não discuti neste artigo,

como escrever um paginador personalizado, que eu não conhecia por um entendimento de alto nível. Este projeto me ensinou como lidar com esses detalhes.

Uma grande lição de aprendizado foi aprender a escrever as APIs antes de começar no front-end. Cometi o erro de escrever o front-end primeiro com dados fictícios, e isso acabou introduzindo um monte de análise JSON desnecessária e transformação na lógica do front-end.

Você pode encontrar o link para meu repositório GitHub e sites de trabalho para este projeto [aqui](#) . Obrigado por ler!

[Django Rest Framework](#)[Pilha completa](#)[Django](#)[Netlify](#)[Heroku](#)[SobreAjudaJurídico](#)

Get the Medium app

