**API REST – CURSO TREINAWEB – DESENVOLVIMENTO DE APIs REST**

* **Documentação da API Rest:** <https://www.django-rest-framework.org/>
* **POSTMAN –** para teste de API – ferramenta que permite fazer requisições para a API que criamos e retorna as respostas.

Passo a passo:

* Criação do ambient, venv e ativação;
* Criação do git ignore e requirements;
* Install das bibliotecas **(django, mysqlclient, djangoframeworkrest);**
* Criar o projeto: **django-admin startproject django\_rest**
* Criar app: **python manage.py startapp**

Em INSTALLED APPS[] temos que declarar os apps:

'api',

'rest\_framework',

* Criar banco de dados no mysql

**CRIANDO O MODELS DA API:**

class Tecnologia(models.Model):

    nome = models.CharField(*max\_length*=20, *null*=False, *blank*=False)

    def \_\_str\_\_(*self*):

*return* *self*.nome

class Vaga(models.Model):

    CONTRATACAO\_CHOICES = [

        ('PJ', 'Pessoa Jurídica'),

        ('CLT', 'Registrado pela CLT'),

    ]

    titulo = models.CharField(*max\_length*=50, *null*=False, *blank*=False)

    descricao = models.TextField(*null*=False, *blank*=False)

    salario = models.FloatField(*null*=False, *blank*=False)

    local = models.CharField(*null*=False, *blank*=False)

    quantidade = models.IntegerField(*null*=False, *blank*=False)

    contato = models.EmailField(*null*=False, *blank*=False)

    tipo\_contratacao = models.CharField(*max\_length*=3, *null*=False, *blank*=False, *choices*=CONTRATACAO\_CHOICES)

    tecnologias = models.ManyToManyField(Tecnologia) *#relacionamento n pra n com varias tecnologias*

    def \_\_str\_\_(*self*):

*return* *self*.titulo

depois:

* Dar um makemigrations
* Dar um migrate

**PROTOCOLO HTTP:**

# O que é e como funciona o HTTP?

Antes de prosseguirmos, é interessante entendermos como funciona o protocolo HTTP, protocolo que iremos utilizar para trafegar as informações através da API Web que iremos criar.

HTTP é um acrônimo para HyperText Transfer Protocol, ou Protocolo de Transferência de HiperTexto. Trata-se de um protocolo que estabelece como deve ocorrer a comunicação entre uma máquina cliente que faz pedidos para uma máquina servidora. Ele é normatizado por uma especificação, a RFC 2616 (<http://tools.ietf.org/html/rfc2616> ).

O protocolo HTTP é baseado na comunicação entre uma máquina cliente que faz requisições para uma máquina servidora. Cada pedido que a máquina cliente faz para o servidor é chamado de requisição ou request, ao passo que a resposta do servidor para cada pedido é chamada de resposta ou response.

O protocolo HTTP é utilizado desde a década de 90 em páginas e aplicações Web, o que inclui APIs Web. No caso deste curso, quem faz o papel de cliente na história é quem for consumir a nossa API: pode ser uma outra aplicação Web, pode ser uma aplicação mobile, ou pode ser uma aplicação desktop, por exemplo.

Vamos a um exemplo: vamos acessar a página do Google. Se quisermos acessar a página inicial do Google, devemos digitar em nosso navegador:

[http://www.google.com.br](http://www.google.com.br/)

Acima, nós temos um HiperTexto, determinado por uma URL (Uniform Resource Locator – Localizador de Recursos Uniforme). Perceba que de fato vamos utilizar o HTTP para fazer a comunicação com o servidor do Google, pois a URL até se inicia com HTTP! Quando damos o enter para que o browser processe a solicitação, uma requisição HTTP é então encaminhada para os servidores do Google, onde ela será processada.

Toda requisição HTTP é composta basicamente por duas partes distintas: cabeçalho (header) e corpo (body). O cabeçalho contém algumas informações específicas da requisição, como o tipo de resposta esperada do servidor e até mesmo o tempo de timeout. Já o corpo pode conter informações adicionais que o cliente pode enviar para o servidor que estarão atreladas à requisição (request). O corpo não é obrigatório, mas o cabeçalho é.

Quando fazemos uma requisição para o Google, nós vamos ter o request similar ao exibido abaixo:

Text, letter

Description automatically generated

Perceba que o cabeçalho da requisição envia para o servidor uma série de informações, por exemplo:

* A URL que gerou a requisição (Request URL);
* O tipo de resposta esperada do servidor (Accept);
* O idioma nativo do browser que disparou a requisição (Accept-Language).

Da parte do request, há uma informação importantíssima que é enviada no cabeçalho: o método de requisição (Request Method). Este dado do cabeçalho indica que tipo de ação a URL que foi disparada para o servidor deverá realizar, dando sentido semântico - ou seja, significado - à requisição. O protocolo HTTP tem uma série de métodos, como GET, POST, PUT, DELETE, HEAD, OPTIONS, TRACE e CONNECT. Nós, na maioria do tempo, utilizamos mais os métodos GET, POST, PUT e DELETE. O significado destes métodos está na tabela abaixo:

|  |  |
| --- | --- |
| Método | Significado semântico |
| GET | Significa que queremos “pegar” algo no servidor: uma página, por exemplo. Requisições GET fazem com que o servidor devolva algo para o cliente, algo que estava “dentro” do servidor. |
| POST | Significa que estamos querendo incluir alguma coisa no servidor. Por exemplo, se temos uma página de cadastro de usuários, a requisição que vai fazer com que o servidor faça o insert no banco de dados deve ser uma requisição POST, afinal, estamos criando um novo item que vai ficar no servidor. |
| PUT | Significa que estamos querendo atualizar alguma coisa no servidor. |
| DELETE | Significa que estamos querendo apagar alguma coisa do servidor. |

Voltando à requisição para o Google, verifique que ela é uma requisição com o método GET. Isso significa que os servidores do Google deverão retornar alguma coisa para o cliente que disparou a requisição: neste caso, deverá ser retornado o HTML da página inicial do Google, para que o navegador (o nosso cliente neste caso) possa então desenhar a página para nós.

As respostas também possuem cabeçalho. Vamos analisar o cabeçalho da resposta do servidor do Google.

Graphical user interface, text, application, email

Description automatically generated

Perceba que o servidor também retorna uma série de informações sobre ele. A resposta também contém indicadores sobre o controle de cache que o browser deverá executar (cache-control), o tipo de resposta retornado pelo servidor (content-type) e até mesmo a data em que a requisição foi processada no servidor. Agora, existe um item muito importante no cabeçalho de resposta: trata-se do status da resposta. É através deste status que o cliente sabe se a requisição retornou sucesso ou se algo deu errado.

Os status HTTP também são padronizados pela especificação. Os principais status HTTP que temos são:

|  |  |
| --- | --- |
| Status | Descrição |
| 200 | OK. Significa que o servidor entendeu a requisição e a processou sem problemas. |
| 302 | Found. Significa que o recurso solicitado de fato existe no servidor (status típico de requisições GET). |
| 401 | Unauthorized. Significa que você tentou acessar algum recurso do servidor que exige autenticação para acesso, e você ainda não realizou este processo. |
| 404 | Not Found. Significa que você solicitou algum recurso no servidor que não existe no lugar que você indicou. Por exemplo: se você tenta acessar alguma página de algum site que não existe. |
| 500 | Internal Server Error. Significa que o servidor encontrou um erro durante o processamento da requisição. |

É através destes status HTTP que o cliente sabe se a requisição que ele disparou deu certo ou não.

O protocolo HTTP ainda possui algumas características que precisamos conhecer:

* O protocolo HTTP é stateless. Isso significa que ele não guarda estado. Por exemplo, você não consegue, somente com o protocolo HTTP, guardar se você acessou determinada página ou não, ou mesmo se um usuário fez o login em sua aplicação web ou não. Isso ocorre porque as requisições HTTP são independentes entre si: quando você faz uma requisição, é aberto um canal de comunicação com o servidor. Por este canal, é trafegada a requisição e a resposta. Logo quando o cliente recebe a resposta, este canal é imediatamente fechado (salvo algumas indicações que podemos fornecer no cabeçalho da requisição). Devido a isso, ele não pode guardar estado, pois esse canal é constantemente aberto e fechado, fora que as requisições ocorrem de maneira isolada (uma requisição não sabe se existe alguma outra requisição sendo feita ou não). Existem estruturas que podemos utilizar para burlar esta característica do HTTP que veremos neste curso;
* Como foi dito, o protocolo HTTP é independente. Quando você faz uma requisição para o servidor, ela é tratada de maneira isolada das demais requisições, sendo impossível fazer com que requisições se comuniquem umas com as outras. Por exemplo: vamos imaginar que você faz uma requisição para uma página HTML que possui um texto e uma imagem. O navegador fará no mínimo duas requisições para carregar esta página: uma para recuperar o texto e outra para recuperar a imagem. Porém, apesar de ambas as requisições serem necessárias para montar uma única página, o servidor as entenderá de maneira completamente isolada, cabendo ao cliente juntar ambas as respostas para montar a página solicitada;
* O protocolo HTTP é assíncrono, ou seja: você pode fazer várias requisições ao mesmo tempo. Como estas requisições são independentes, elas também serão tratadas ao mesmo tempo. O servidor também irá devolver as respostas não necessariamente na mesma ordem em que as requisições foram realizadas, ele irá devolver à medida que o processamento for sendo finalizado.

É importante conhecermos estas características do protocolo HTTP para justificarmos uma série de recursos e técnicas que vamos utilizar mais à frente, quando começaremos a criar APIs Web.

# O que é o REST

REST é um acrônimo para Representational State Transfer, ou Transferência de Estado Representacional, conceito criado por Roy Fielding em sua tese de doutorado. Roy Fielding é um dos autores da especificação do protocolo HTTP, além de ser co-fundador do projeto Apache HTTP Server (conhecido simplesmente como Apache).

Podemos entender o REST como sendo uma arquitetura que visa o aproveitamento ao máximo do protocolo HTTP para transferência de informações. O REST está ligado a APIs Web justamente porque o papel destas é trafegar informações entre um cliente e um servidor através do protocolo HTTP.

Um ponto é importante e precisa ser reforçado: REST não é uma tecnologia, não é um framework, não é uma linguagem e nem é uma biblioteca. REST é uma arquitetura que visa tirar proveito ao máximo do protocolo HTTP, implementando a distribuição de recursos principalmente através dos hiperlinks.

APIs que conseguem implementar corretamente o protocolo HTTP, bem como suas características, são chamadas de APIs RESTful.

Temos um artigo onde falamos bastante sobre REST e a importância deste com relação à APIs Web, você pode acessá-lo em: <https://www.treinaweb.com.br/blog/rest-nao-e-simplesmente-retornar-json-indo-alem-com-apis-rest/>

# Para que servem os serializers?

A principal função de um serializer no Django REST é permitir que os dados recebidos por uma requisição (normalmente em JSON ou XML) sejam convertidos para os tipos de dados em Python, linguagem que a aplicação foi construída, e que as respostas sejam convertidas de Python para XML ou JSON. Estes processos são conhecidos como serialização e desserialização e possuem papel fundamental para que uma API possa ser utilizada por aplicações escritas em diferentes linguagens.

No Django REST, os serializers funcionam de forma bem semelhante às classes Form e ModelForm do Django, permitindo que, além da serialização e desserialização, os dados possam ser validados com base nos models do projeto.

**PAPEL DO SERIALIZER:**

* Definir quais campos serão exibidos quando uma consulta for feita;
* Validação dos dados que serão exibidos e salvar os dados no banco de dados

CRIAR AS PASTAS DO SERIALIZER:

Text

Description automatically generated

Arquiv tecnologia\_serializer.py:

from rest\_framework *import* serializers

from ..models *import* Tecnologia

class TecnologiaSerializer(serializers.ModelSerializer):

    class Meta:

        model = Tecnologia *#model de base para verificação se os dados*

*#estão validos - no caso somente vamos validar o unico campo que é*

*#o nome da Tecnologia - validacao de dados que entrarão no banco de dados*

        fields = ('nome', )

arquivo vaga\_serializer.py:

from rest\_framework *import* serializers

from ..models *import* Vaga

class VagaSerializer(serializers.ModelSerializer):

    class Meta:

        model = Vaga *#model de base para verificação se os dados*

*#estão validos - no caso somente vamos validar o unico campo que é*

*#o nome da Tecnologia - validacao de dados que entrarão no banco de dados*

        fields = ('titulo','descricao', 'salario', 'local', 'quantidade',

        'contato', 'tipo\_contratacao', 'tecnologias', )

**FAZENDO CRUDE BÁSICO:**

Criar os getters and setters:



Cria os services:

from ..models *import* Tecnologia

def listar\_tecnologias(): *#metodo que vai obter todas as tecnologias do banco*

*#de dados e retornar a lista de tecnologias*

    tecnologias = Tecnologia.objects.all()

*return* tecnologias

**CRIANDO AS VIEWS DA API – usando class-based views**

* Metodos que nao precisam de parametros
* Metodos que precisam de parametros

Depois das views, coloca nas urls:

from django.contrib *import* admin

from django.urls *import* path, include

from .views *import* tecnologia\_views

urlpatterns = [

    path('tecnologias/', tecnologia\_views.TecnologiaList.as\_view(), *name*='tecnologia-list'),

]

Depois manda pro Postman a requisição:

Graphical user interface, text, application, email

Description automatically generated

GET – obtem um valor por parametro e retorna uma tecnologia por id

METODO DELETE: Passando o metodo DELETE no postman, usar a url e passar o id que queremos deletar, ele deleta de acordo com nosso service + views + serializer.

Graphical user interface, text, application, email

Description automatically generated

Usando o metodo GET para a retornar as tecnologias que estão cadastradas no BD, retornaremos apenas:

Graphical user interface, text, application, email

Description automatically generated

* PARA DE **EDITAR UMA VAGA**, TEREMOS QUE PASSAR O ID DA VAGA E FAZER ALTERAÇÃO NO CORPO;
* USANDO O METODO **PUT**;

Graphical user interface, text, application, email

Description automatically generated

# Para que serve a paginação?

A paginação é de grande importância para a listagem dos nossos registros. É com ela que evitamos que todos os dados sejam retornados de uma única vez para o cliente que as solicitou.

Imagine que nossa API possua mil registros a serem exibidos e estes mil registros sejam retornados de uma única vez. Além de ter problemas para este retorno (requisição muito pesada), o cliente teria sérios problemas em ler a requisição, já que a quantidade de dados seria muito grande.

Para que este problema seja evitado, nesta aula veremos como aplicar a paginação em nossa listagem para que uma quantidade X de vagas sejam retornadas por página.

**CONFIGURANDO PAGINAÇÃO:**

https://www.django-rest-framework.org/api-guide/pagination/

SETTINGS:

Text

Description automatically generated

VIEWS.PY: ATIVANDO A PAGINAÇÃO:

Graphical user interface, text, application, email

Description automatically generated

# Para que serve a autenticação?

A autenticação é a forma com que os usuários que utilizem uma aplicação possuam prévio cadastro e utilize estas informações para se identificarem. Basicamente, a autenticação é o ato de estabelecer ou confirmar algo como autêntico, reivindicando a veracidade das informações.

A autenticação é o modo mais básico de garantir que apenas usuários previamente cadastrados em nossa API possuam acesso às informações disponibilizadas. Sendo assim, veremos nesta aula como implementar uma autenticação básica utilizando Bearer Tokens.

Autenticando:

Primeiro faz o seguinte trajeto:

* Configura o settings.py:
* REST\_FRAMEWORK = {
* 'DEFAULT\_PAGINATION\_CLASS': 'rest\_framework.pagination.LimitOffsetPagination',
* 'PAGE\_SIZE': 2,
* 'DEFAULT\_AUTHENTICATION\_CLASSES': (
* 'oauth2\_provider.contrib.rest\_framework.OAuth2Authentication',
* )
* } *#por padrao nossa classe de autenticacao será essa - geraremos os tokens de acesso às credenciais*
* OAUTH2\_PROVIDER = {
* 'ACCESS\_TOKEN\_EXPIRE\_SECONDS': 60,
* *#tempo de expiração da Sessao*
* 'OAUTH2\_BACKEND\_CLASS': 'oauth2\_provider.oauth2\_backends.JSONOAuthLibCore'
* } *#usando como backend essa configuracao*
* Adiciona o app abaixo no array do Installedapps:

'oauth2\_provider',

* Adiciona o path abaixo no urls.py no mesmo nivel do settings.py:

path('oauth/', include('oauth2\_provider.urls', *namespace*='oauth2\_provider')),

* Faz as migrações com migrate
* Criar um super usuario para entrar na parte de django admin
* Depois faz um application no django admin:

Graphical user interface, text, application, email

Description automatically generated

\*importante que devemos pegar as informações de Client id e Client Secret para incluir no dict do POSTMAN. E incluir conforme abaixo como POST:

{

    "grant\_type": "password",

    "client\_id": "vWCNi53AooW3SmlBI2psIozW1vvHzu7K39llHhJi",

    "client\_secret": "qY5IK2jMCh1V1BferdvPUonvuilbwCfurKjVIDllaXYRhlBUitOzwZPEVZEl5frOMO23Tv2HLqmZvno7DVCsBXfohOI9x2JQMgc2YigQPBCsuUa1I0Ga24CyUkScrpKR",

    "username": "renata",

    "password": "iiee32"

}

As informações saem da forma conforme o PRINT:

Graphical user interface, text, application, email

Description automatically generated

**Utilizando esse tolken de acesso “access\_token” para fazer requisições dentro da nossa API.**

**AGORA FAREMOS FAREMOS A PROTEÇÃO DAS ROTAS E METODOS DE AUTENTICAÇÃO:**

Aqui, depois de feitas as configurações em usuarios\_views.py, podemos permitir o cadastro de usuários:

permission\_classes = [AllowAny]

E assim receber a chave de acesso para uso da API:

Graphical user interface, text, application, email

Description automatically generated

**COMO ENVIAR O TOKEN DE ACESSO:**

* Colocamos Authorization (como keys) depois Bearer+token (como value).

Graphical user interface, text, application, email

Description automatically generated

Como a nossa aplicação tem um tempo máximo para uso do Access Token, temos que sempre se atentar a gerar novos access tolkens.

Neste caso abaixo, vamos colocar as infos passadas + novo access token e a API retorna, os dados solicitados.

Graphical user interface, text, application, email

Description automatically generated

# O que é o HATEOAS?

HATEOAS é uma abreviação para **Hypermedia As The Engine Of Application State**, ou Hipermídia como Motor do Estado da Aplicação, dados que este deve armazenar. Trata-se de uma regra que deveria ser seguida por todas as APIs REST.

Segundo o conceito do HATEOAS, clientes interagem com APIs REST através de rotas, que também poderiam ser chamadas de hiperlinks - conceito que também pode ser válido para APIs baseadas em RPC. O grande ponto é que APIs baseadas em RPC realizam essa interação entre cliente e servidor através de um protocolo fixo: se um cliente quiser invocar um endpoint para recuperação de um cliente pelo ID, por exemplo, ele deverá reconhecer esse endpoint. A mesma coisa se o cliente quiser inserir ou alterar um cliente. O cliente passa a ter conhecimento do funcionamento da API baseada em RPC, se acoplando a ela. Se o endpoint de listagem de clientes é alterado, o cliente irá parar de funcionar, pois este está intimamente relacionado à API através desse “contrato” que acabou sendo firmado. Isso pode atrapalhar a evolução da API a um longo prazo.

O HATEOAS visa justamente quebrar esse acoplamento entre cliente e servidor, retirando a necessidade desse “contrato” que acaba sendo firmado. Segundo o HATEOAS, o cliente deve interagir com uma API através de hiperlinks genéricos, sem precisar conhecer necessariamente o significado desses hiperlinks.

O HATEOAS conceitualmente consegue esse desacoplamento através da adição de informações quando um recurso é solicitado. Além de o próprio recurso ser transferido, a API também deve devolver as operações que aquele recurso pode sofrer, bem como os endpoints necessários para invocação daquele serviço.

Por exemplo: se invocássemos o endpoint que recupera um cliente por seu ID, provavelmente poderíamos ter uma resposta similar à exibida abaixo:



{

nome: 'TreinaWeb',

idade: 10,

saldo: 555.55

}

JavaScript

Perceba que os dados solicitados realmente foram devolvidos pela API. Porém, o que pode ser feito com este recurso? Será que podemos alterá-lo? Será que podemos excluí-lo? Será que podemos realizar um depósito para esse cliente? Aí está o problema: o cliente passa a precisar entender o que pode ser feito com este recurso, criando o acoplamento com a API.

Se implementássemos o HATEOAS, os serviços que podem ser invocados com aquele recurso também seriam devolvidos pela API. Poderíamos ter uma resposta parecida com a ilustrada abaixo:



nome: 'TreinaWeb',

idade: 10,

saldo: 555.55,

links: [

{ rel: 'alterar', link: '/api/clientes/1' },

{ rel: 'excluir, link: '/api/clientes/1' },

{ rel: 'depositar, link: '/api/clientes/1/depositar' },

]

}

JavaScript

Perceba que a própria API informa os serviços disponíveis para aquele recurso que foi devolvido, quebrando este acoplamento: o cliente agora não precisa saber o que pode ser feito com aquele recurso, bastando a ele interagir com os hiperlinks que foram devolvidos pela API, hiperlinks estes que expõem as ações que podem ser realizadas.

Usando o HATEOAS podemos pegar os links atraves do JSON da API, onde abaixo teremos os LINKS:

Como geramos as rotas, teremos os links de delete, update e get:

Graphical user interface, text, application, email

Description automatically generated

Neste curso vimos como funciona e como implementar uma API REST utilizando o Django e o Django REST Framework.

Aprendemos como funcionam os serializers, paginação e autenticação no Django REST Framework. Para isso, construímos uma API com relacionamento NN entre as entidades Tecnologia e Vaga, para que vagas de emprego fossem cadastradas utilizando nosso serviço.