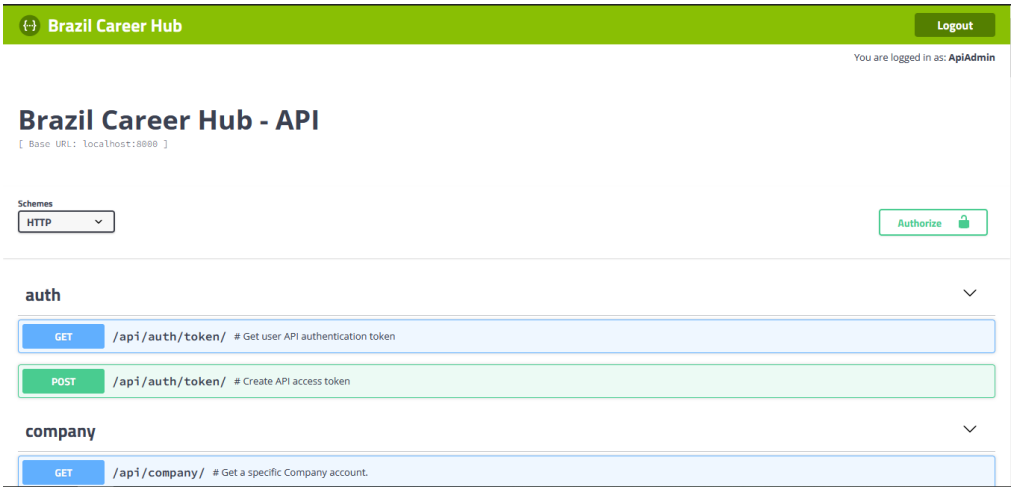
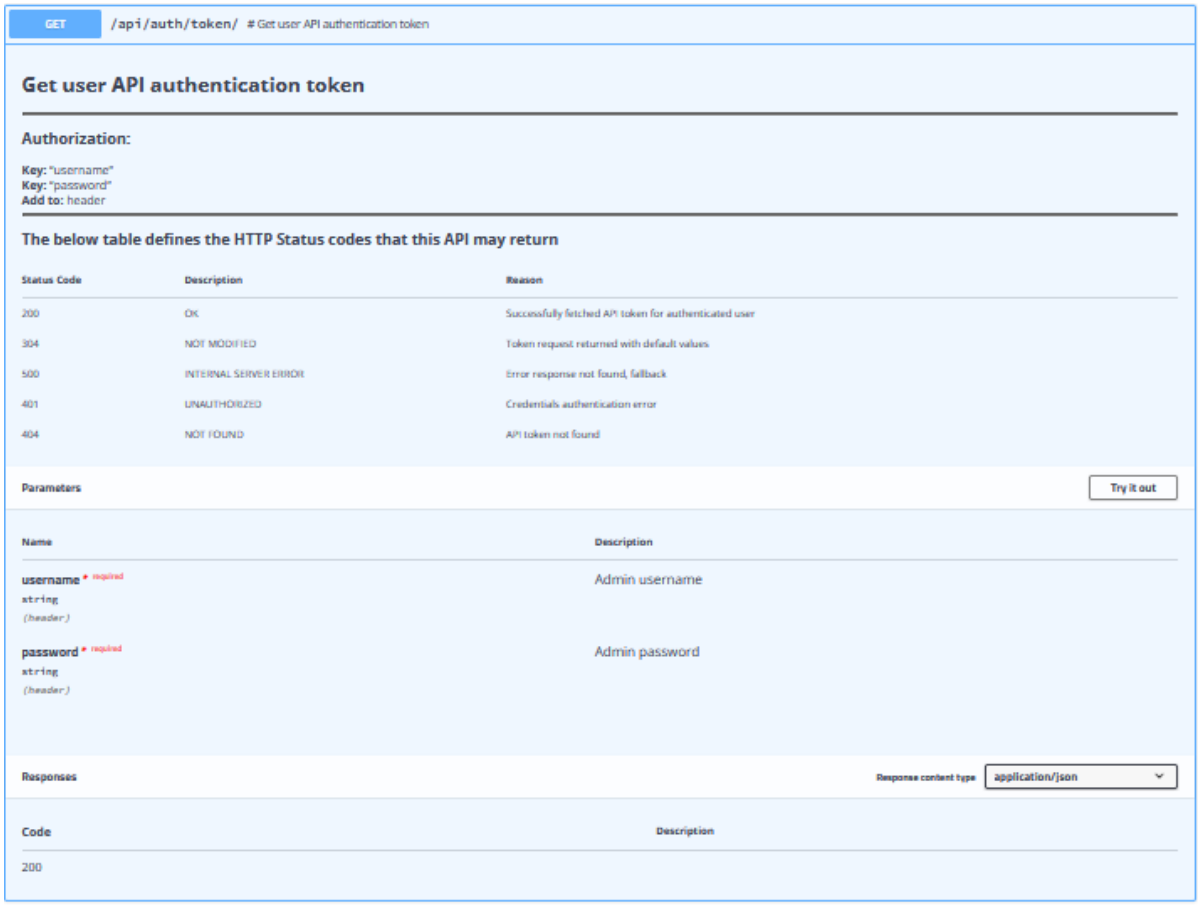
O ambiente backend é responsável por armazenar os dados de todas as empresas e usuários, vagas, currículos e por disponibilizar esses dados para a interface. A aplicação do backend é uma API utilizando o framework v4.2.3 codificado na linguagem de programação Python v3.11.4. A aplicação também implementa o framework RESTful (também conhecido como rest) encapsulado como uma biblioteca otimizada e customizada para ser utilizada juntamente com o framework do Django.

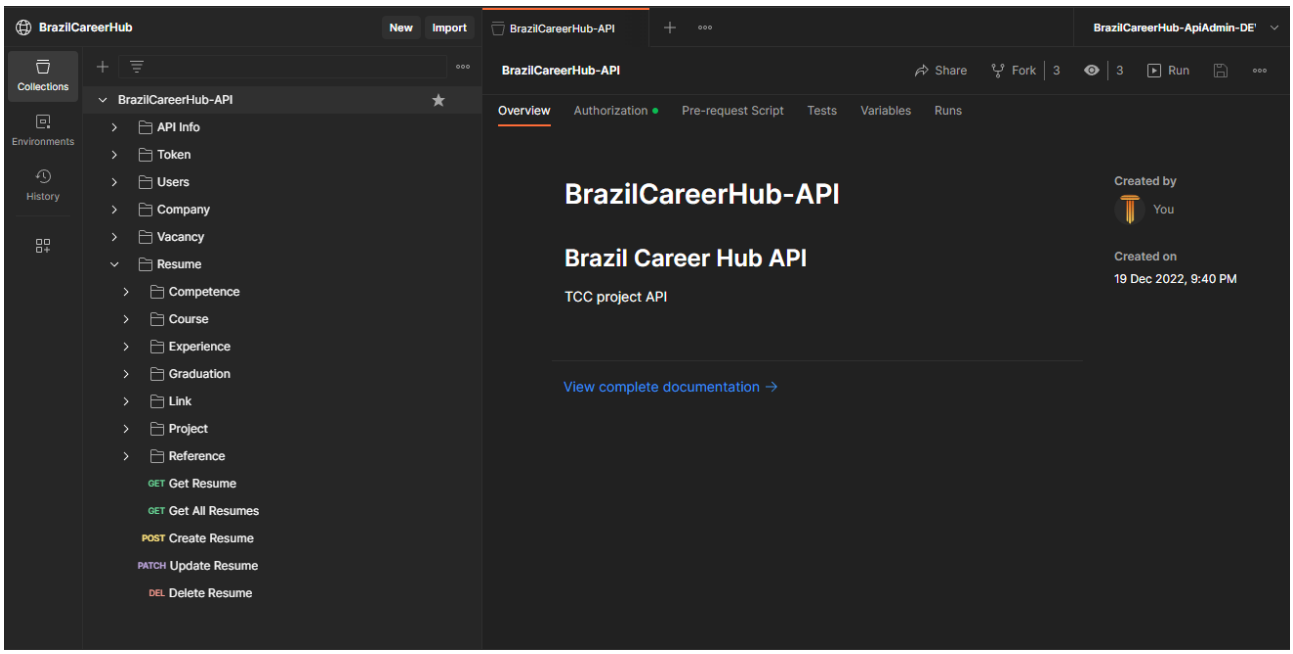
A aplicação faz uso de um banco de dados relacional MySQL, para facilitar o processo e reduzir os recursos necessários durante o desenvolvimento, estamos utilizando uma variação do MySQL que não necessita de um servidor e é independente, o sqlite, o qual permite persistir os dados em um banco de dados relacional não precisando hospedar ou se conectar com um servidor.

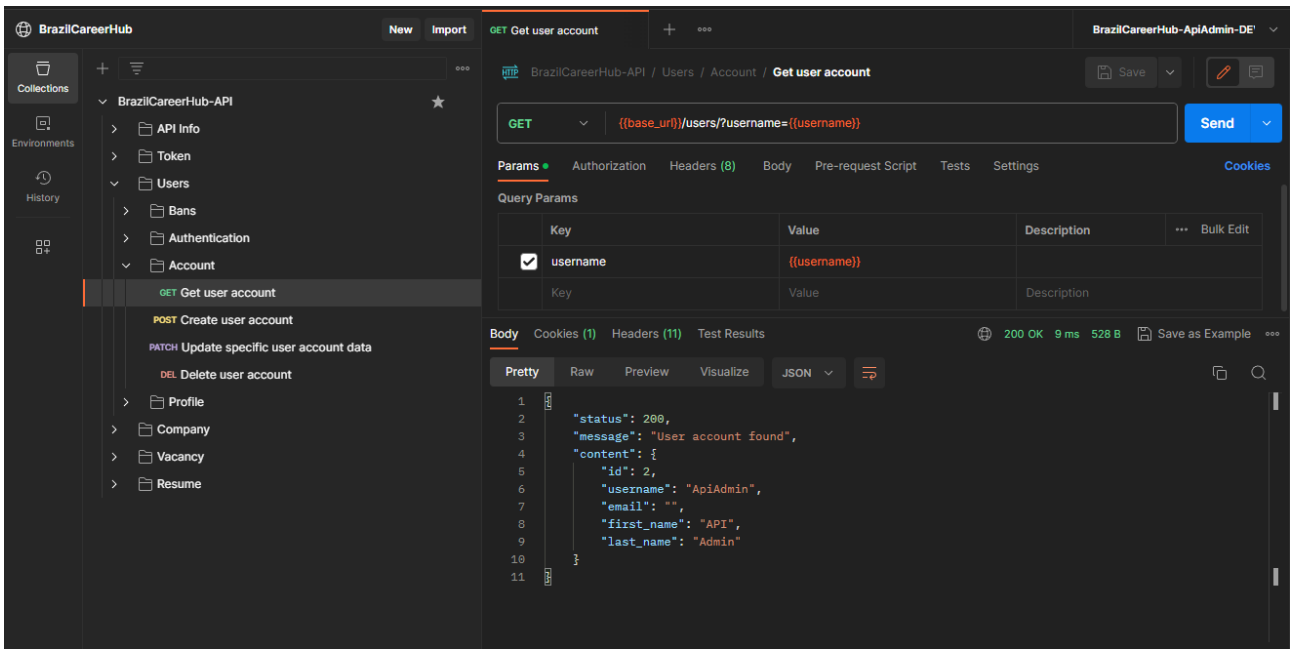
A API disponibilizada possui 19 endpoints com 15 deles possuindo operações CRUD completas, para facilitar o desenvolvimento o backend possui uma documentação completa para todas as APIs disponibilizando uma interface gráfica utilizando o Swagger e Swagger UI, os quais são conjuntos de ferramentas de API na qual a especificação OpenAPI é baseada.





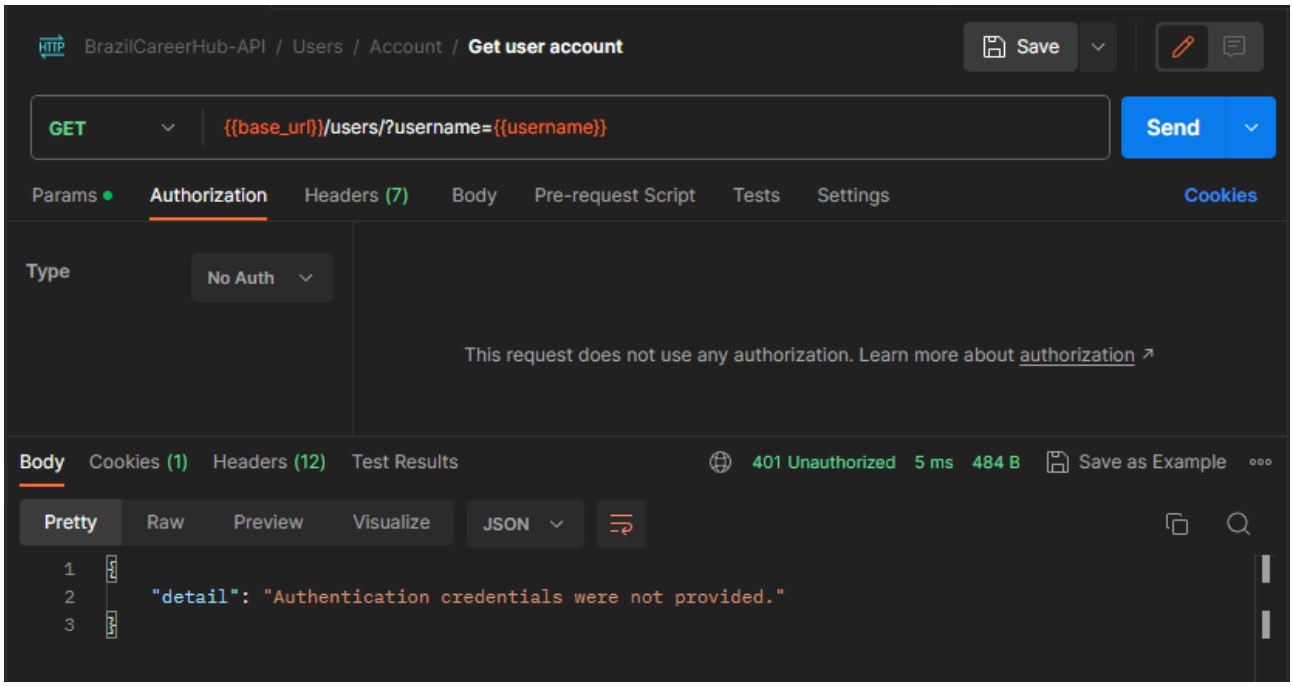
Além da interface gráfica, para facilitar ainda mais os testes durante o desenvolvimento, também implementamos uma coleção completa de endpoints hospedada no Postman, uma plataforma de API para desenvolvedores. Nessa coleção, o desenvolvedor pode realizar chamadas para os endpoints do backend para avaliar as respostas, os parâmetros e outros pontos necessários para o desenvolvimento de uma API.





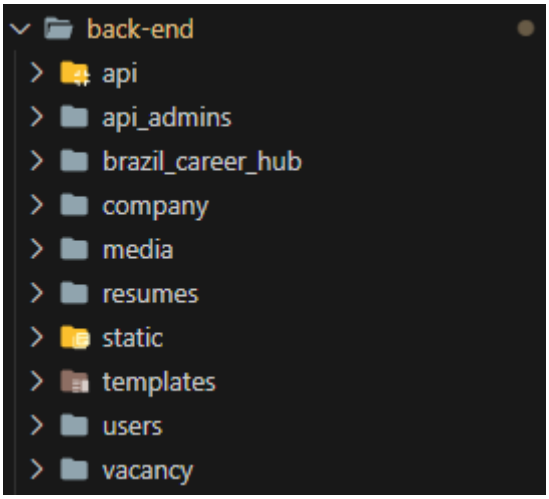
A aplicação possui um método de autenticação do tipo Bearer Token, o qual é um tipo de token de acesso em uma API utilizado para autenticação e autorização para manter as credenciais do usuário e indicar autorização para requisições de acesso. Os tokens Bearer são gerados com base em protocolos e especificações como OAuth e JWT (JSON Web Token).

Toda requisição para o ambiente de backend e seus endpoints devem possuir um token localizado no cabeçalho (header) da requisição, caso a requisição não informe um token, o acesso não será autorizado e o usuário receberá um erro. Para gerar um token, o usuário deve possuir uma conta de administrador no ambiente de backend e solicitar a geração de um token de acesso a partir de um endpoint único, o qual deve se autenticar utilizando usuário e senha.



Toda a camada de endpoint, autenticação e resposta é tratada pelo próprio framework do Django com a ajuda do framework do rest. Todos os endpoints possuem o mesmo esquema de declaração e funcionamento da API, o que difere entre um endpoint e outro são dos dados que são gerenciados pelo endpoint, a complexidade e a lógica para aquisição e tratamento dos dados.

A arquitetura do ambiente de backend foi modularizada e seccionada cuidadosamente para criar uma arquitetura onde possibilita um fácil escalonamento, atualização e manutenção.



A pasta **api** possui os arquivos responsáveis pela informação geral da API. A pasta **api_admins** possui os arquivos responsáveis pelo armazenamento e gerenciamento de usuários administrativos, os quais poderão gerar seus tokens para realizar acessos remotos via endpoints. A pasta **company** é responsável pela lógica, armazenamento e gerenciamento de todos os dados (conta e perfil) das empresas cadastradas no site. Assim como a **company**, a pasta **users** é responsável pela lógica, armazenamento e gerenciamento de todos os dados (conta e perfil) dos usuários cadastrados no site. A pasta **vacancy**, assim como o nome define, é responsável pelo tratamento de todas as vagas criadas. Por fim, a pasta **resumes** é responsável pela lógica, armazenamento e gerenciamento de todos os currículos gerados pelos usuários, os quais serão utilizados para se candidatar nas vagas, as quais por sua vez, serão publicadas pelas empresas.

A estrutura de urls e endereços da API são definidas em arquivos específicos dentro do framework do Django, os quais são comumente nomeados de **urls.py**. Nesses arquivos, são definidos os endereços os quais serão acessados pelos usuário através dos endpoints, essas definições irão lincar a parte lógica da API (também conhecidas como views) e a área externa da aplicação (os endpoints).

```python
app_name = 'api'

# localhost:8000/api/
urlpatterns = [
    # ========== API Info ========== #
    path('info/status/', views.ApiStatus.as_view()),  # type: ignore
    path('info/version/', views.ApiVersion.as_view()),  # type: ignore

    # ========== Authentication ========== #
    path('auth/', include('api_admins.urls')),  # type: ignore

    # ========== Users ========== #
    path('users/', include('users.urls')),

    # ========== Company ========== #
    path('company/', include('company.urls')),

    # ========== Vacancy ========== #
    path('vacancy/', include('vacancy.urls')),

    # ========== Resumes ========== #
    path('resumes/', include('resumes.urls')),
]
```

As views, por sua vez, possuem uma abordagem orientada a objetos, também conhecida como **class-based views**, sendo ela uma das abordagens disponibilizada pelo framework do Django. A criação de uma API consiste em duas classes, um schema, o qual será responsável por produzir as informações necessárias para gerarmos a documentação do Swagger, e uma classe onde será responsável pelo endpoint.

A classe responsável pelo schema renderizado na documentação do Swagger deve conter duas principais funções, uma para a descrição do conteúdo e outra para os parâmetros do endpoint. Essa classe será utilizada apenas para renderizar as informações corretas, porém, apesar de sua funcionalidade simples, mostra-se essencial para o conhecimento da API e de seus endpoints.

```
You, 4 months ago | 1 author (You)
class VersionSchema(AutoSchema):
    """Schema for version endpoint"""
    def get_description(self, path: str, method: str) -> str:
        authorization_info = """
## Authorization:

**Type:** Bearer
"""
        match method:
            case 'GET':
                responses = {
                    "200": {
                        'description': 'OK',
                        'reason': 'API Version and environment fetched successfully'
                    },
                }
                return description_generator(title="Fetches API version and environment",
                                             description=authorization_info,
                                             responses=responses)
            case _:
                return ''

    def get_path_fields(self, path: str, method: str) -> list[coreapi.Field]:
        match method:
            case _:
                return []
```

A classe responsável pelo endpoint deverá conter os métodos das requisições que deverão ser suportadas pelo endpoint, ou seja, se o endpoint permite uma requisição no método GET, uma função get deve existir dentro da classe pelo fato de que ela será executada ao receber uma requisição desse método.

Essa função deverá receber um parâmetro self, o qual será utilizado para referenciar a própria classe, e um parâmetro request, o qual será utilizado para referenciar a requisição que foi recebida, possuindo uma tipagem de um HttpRequest, uma estrutura baseada em dicionários disponibilizada no Python.

Essa função também deverá retornar uma resposta que possa ser serializada e enviada via HTTP, permitindo uma comunicação entre APIs sem a renderização de uma página. Essa resposta deverá receber um dicionário que irá conter os dados que devem ser retornados na resposta, assim como um código de status, o qual será utilizado para definir se a requisição retornou os dados com sucesso ou se ocorreu algum erro no processo.

```
You, 7 months ago | 1 author (You)
class ApiVersion(Base):
    """Returns API version and its environment"""

    schema = VersionSchema()

    def get(self, request):
        """Get request"""
        data = self.generate_basic_response_data(status.HTTP_200_OK, 'API version')
        data['version'] = tools.generate_version()
        return Response(data=data, status=data.get('status'))
```

Nesse caso em específico, o conteúdo retornado é apenas um texto (string) que irá informar a versão da API e qual o ambiente que está sendo executado (desenvolvimento ou produção). Outro modelo de resposta que também é muito utilizado é o retorno de objetos do banco de dados, porém, diferente de uma string, um objeto que seria uma classe na linguagem Python não pode ser serializado automaticamente pelo framework do rest.

Para permitir a serialização de um objeto, é necessário definir um serializador que realizará essa conversão de um objeto para algo que possa ser retornado via protocolos HTTP, o que no caso seriam dicionários.

```
You, 27 seconds ago | 1 author (You)
class VacancyModelSerializer(serializers.ModelSerializer):
    addresses = serializers.SerializerMethodField()
    company_name = serializers.SerializerMethodField()

    def get_addresses(self, obj):
        if obj.address:
            return obj.address.__getattribute__("serialize")

        return []

    def get_company_name(self, obj: models.VacancyModel):
        return obj.created_by.fantasy_name

    You, 27 seconds ago | 1 author (You)
    class Meta:
        model = models.VacancyModel
        fields = ["pk", "created_by", "company_name", "role", "description",
                  "modality", "created_at", "salary", "addresses", "resumes"]
```

O exemplo mostrado na imagem **<insert image number here>** é o serializador de objetos de vagas, ele é responsável por converter uma entrada do banco de dados em um dicionário que possa ser serializado e transmitido para o requisitante via protocolo HTTP. Em resumo, o serializador possui uma meta classe que irá definir duas principais variáveis, **model** e **fields.** A variável **model** será responsável por definir qual a tabela do banco de dados a qual os dados deverão ser buscados, já a variável **fields** irá informar

quais informações (colunas) devem ser selecionadas e serializadas para poderem ser transmitidas.

Além dessas duas variáveis, o exemplo exibe também algumas variáveis computadas, os quais seriam **addresses** e **company_name**. Essas variáveis são variáveis únicas geradas durante a serialização, geralmente utilizadas para obter informações de outras tabelas que estão lincadas por chaves estrangeiras, nesse caso, a variável **addresses** é responsável por obter o endereço disponível para essa vaga em específico, a qual é armazenada em uma outra tabela de endereços da empresa. Já a variável **company_name** é responsável por obter o nome da empresa a qual criou a vaga, a qual também é armazenada em uma outra tabela de contas e perfis das empresas.

O retorno desse serializador é um dicionário que poderá ser transmitido via protocolo HTTP, caso algum erro ocorra durante a serialização, uma exceção será levantada, alertando o usuário sobre o erro.

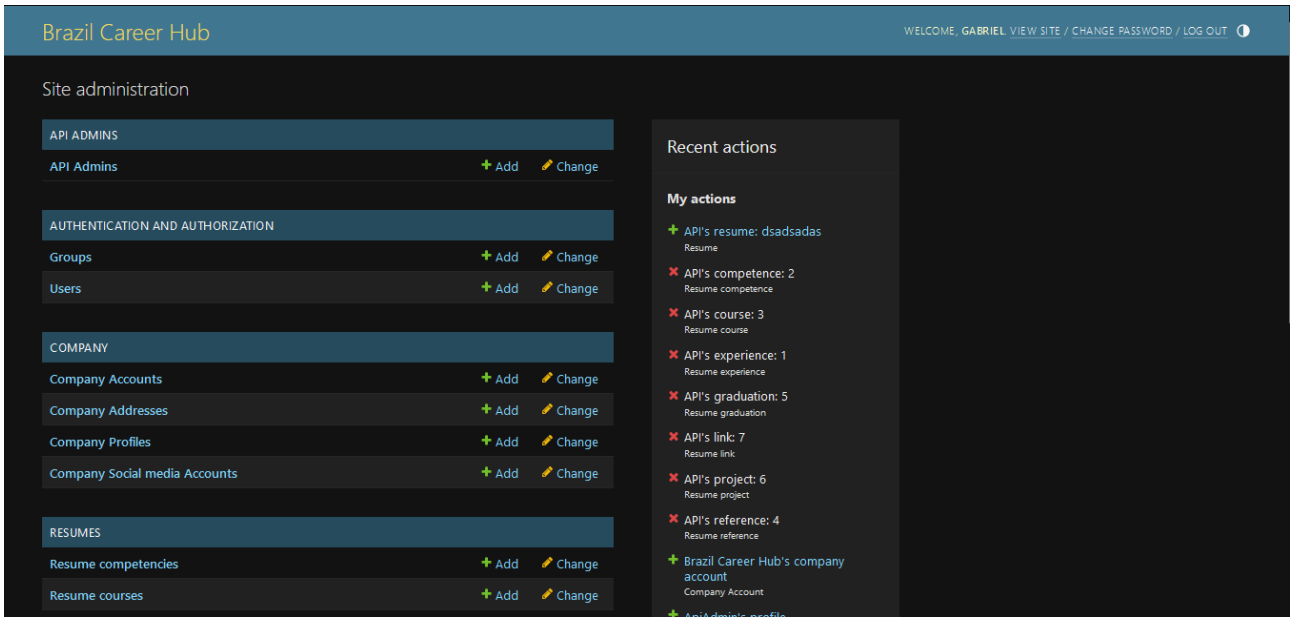
No framework Django, o banco de dados é tratado completamente pelo próprio framework, sendo necessária a criação do chamado modelo. O modelo é uma classe a qual possui uma estrutura específica para que possa ser interpretada pelo framework e manipulada como tabelas em um banco de dados relacional.

```
You, last month | 1 author (You)
class VacancyModel(models.Model):
    created_by = models.ForeignKey(CompanyAccountModel,
                                   verbose_name="Created by",
                                   on_delete=models.CASCADE)
    role = models.CharField(verbose_name="Role", max_length=255)
    description = models.TextField(verbose_name="Description")
    modality = models.CharField(verbose_name="Modality", max_length=255)
    created_at = models.DateTimeField(verbose_name="Created At",
                                      auto_now_add=True,
                                      editable=False)
    salary = models.PositiveIntegerField(verbose_name="Salary")
    address = models.ForeignKey(VacancyAddress,
                                verbose_name="Address",
                                on_delete=models.SET_NULL,
                                null=True,
                                blank=True)
    resumes = models.ManyToManyField(ResumeModel,
                                     verbose_name="Resumes",
                                     blank=True)

    def __str__(self) -> str:
        return self.role

    You, 2 months ago | 1 author (You)
    class Meta:
        verbose_name = 'Job vacancy'
        verbose_name_plural = 'Job vacancies'
```

O exemplo mostrado na imagem **<insert image number here>**, é o modelo da tabela das vagas criadas por uma empresa, a classe do modelo herda de uma classe específica do framework que permitirá que o Django interprete suas variáveis, métodos e metaclasses criadas e gere comandos SQL a partir dessas interpretações.

As variáveis declaradas na classe são as variáveis que irão definir as colunas das tabelas, os seus valores são os tipos / atributos nos quais os valores dessa coluna serão tratados, assim como a herança da classe, os valores das variáveis são classes específicas do framework, os quais permitirão a interpretação e manipulação conforme necessário. Das classes, a classe **ForeignKey** é responsável por gerar uma relação de chave estrangeira com outra tabela, o que neste caso, seria outro modelo gerado a partir do framework. A classe **ManyToManyField** é responsável por gerar uma relação muitos-para-muitos com outro modelo.

Devido a responsabilidade de manipulação do banco de dados ser completamente do framework, o Django disponibiliza nativamente uma interface de administração, também conhecido como “Django administration”, para que os dados do banco possam ser facilmente acessados e modificados sem necessitar uma conexão manual com o banco, podendo causar problemas e incompatibilidades com o tratamento do framework.



O registro de um modelo para exibição na página de administração do framework é realizado por um arquivo específico de sua estrutura chamado de **admin.py**, nele são criadas classes que irão exibir, além das seções presentes na imagem **<insert image number here>**, as páginas de gerenciamento e exibição dos dados da tabela.

```python
You, 2 months ago | 1 author (You)
class VacancyAddressAdmin(admin.ModelAdmin):
    list_display = ('pk', 'title', 'address', 'number')
    list_display_links = ('pk', 'title')
    list_per_page = 35
    search_fields = ('title', 'address')


You, last month | 1 author (You)
class VacancyAdmin(admin.ModelAdmin):
    list_display = ('pk', 'created_by', 'role', 'description', 'modality', 'created_at', 'salary')
    list_display_links = ('pk', 'created_by')
    list_per_page = 35
    list_filter = ('created_at',)
    search_fields = ('role', 'description', 'modality')
    readonly_fields = ('created_at',)


admin.site.register(models.VacancyAddress, VacancyAddressAdmin)
admin.site.register(models.VacancyModel, VacancyAdmin)
```

O exemplo mostrado na imagem **<Insert image number here>** é o processo de registro da exibição das vagas criadas no banco de dados, a classe **VacancyAdmin** é criada herdando, assim como seu modelo, de uma classe específica do próprio framework, permitindo interpretar as informações e renderizar a página conforme solicitado.

As variáveis declaradas na classe são responsáveis pela configuração da exibição do modelo, a variável **list_display** será responsável pela configuração de quais colunas da tabela serão exibidas na listagem e o **list_display_links** serão os links para acessar a página de gerenciamento dessa entrada em específico. O **list_per_page** é responsável por configurar o número máximo de entradas permitidas sem paginação, o **list_filter** será um filtro localizado na extrema direita da página que irá permitir a filtragem dos dados listados, o **search_fields** será responsável por informar ao framework quais são as colunas as quais o texto buscado deverá ser pesquisado e o **readonly_fields** é responsável pela configuração de quais campos não devem ser modificados pelo usuário manualmente. Veja um exemplo da listagem na imagem **<Insert image number here>**