UNIVERSIDADE PAULISTA

**DOCUMENTAÇÃO DO SERVIDOR DE RELATÓRIOS**

Gabriel Menezes de Antonio – RA F13GJI6

Mayara Marques Pereira de Souza – RA N542DD1

Carlos Eduardo dos Santos Ferreira – RA N6401C7

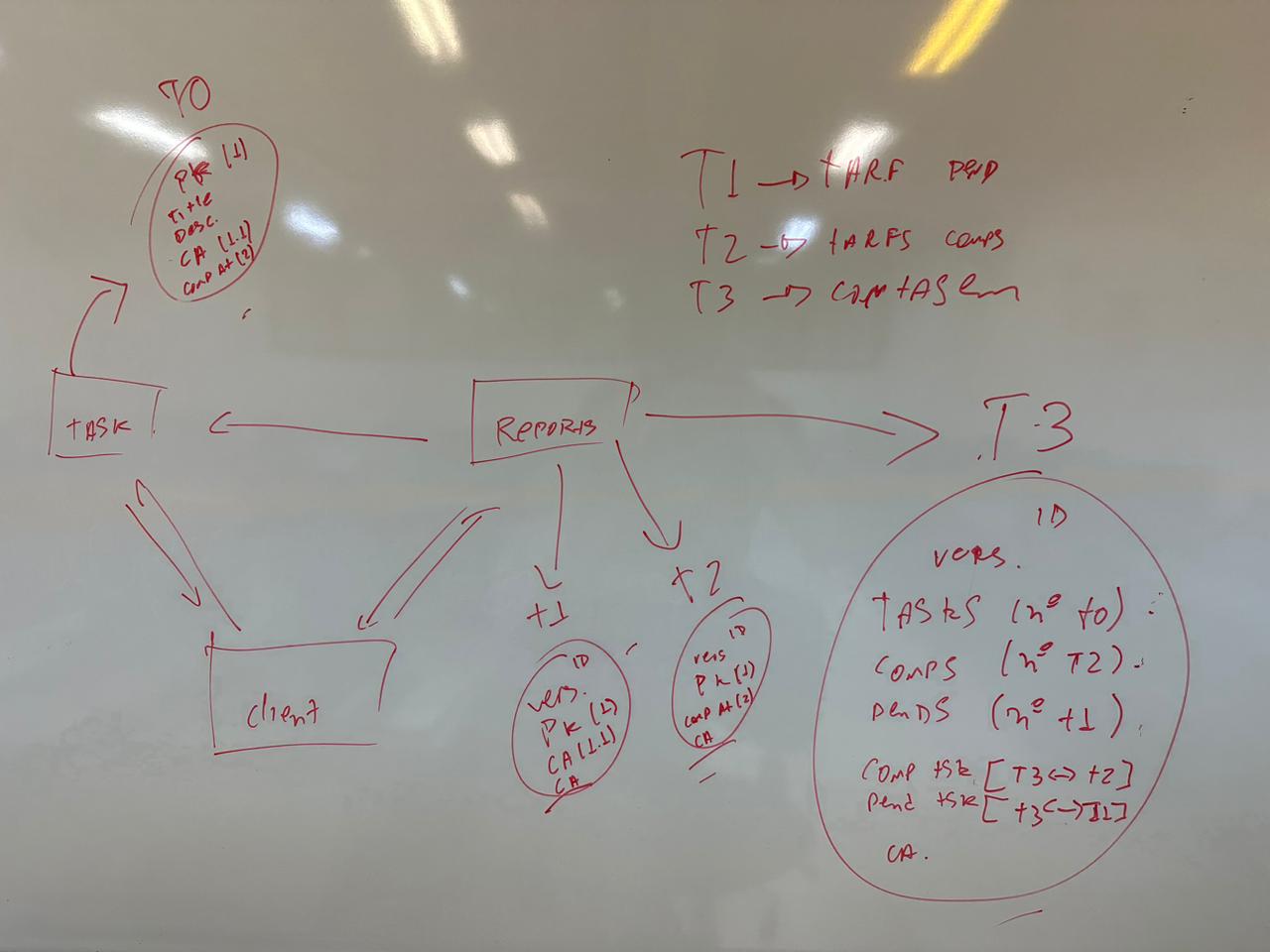
Gustavo Henrique dos Santos Faria – RA F22IFG2

**Sistemas Distribuídos - Ciência da computação**

**Campinas, 2023**

**Introdução**

Nossa proposta é desenvolver um sistema distribuído para o gerenciamento e armazenamento de uma lista de tarefas. O projeto consiste em dois servidores e um cliente, os quais serão explicados com mais detalhes futuramente. A topologia dos servidores e suas arquiteturas são as seguintes:



* O servidor de tarefas será responsável por realizar o CRUD completo de todas as tarefas do cliente, além de armazenar todos os dados em um banco de dados localizado no próprio servidor.
* O servidor de relatórios será o responsável por gerar relatórios, consolidações e outras informações relevantes com base no conteúdo armazenado no servidor de tarefas, tendo contato direto e exclusivo com o servidor de tarefas, realizando apenas comunicação de leitura para armazenar os dados em um banco de dados localizado no próprio servidor.
* O cliente será o ambiente responsável pela utilização e ligação entre os servidores, o cliente terá acesso exclusivo à ambos os servidores, podendo realizar qualquer requisição CRUD no servidor de tarefas e requisições GET no servidor de relatórios.

Cada servidor foi desenvolvido em uma linguagem diferente, utilizando tecnologias diferentes para gerar uma complexidade técnica para a comunicação entre plataformas. Com essa estrutura, teremos um sistema distribuído que poderão comunicar entre si, respeitando limites, protocolos entre outros.

**Cliente**

O cliente será a interface utilizada pelo usuário final, realizando uma comunicação direta com ambos os servidores de tarefas e relatórios, ao criar, concluir e excluir uma tarefa, o cliente irá se comunicar diretamente com o servidor de tarefas.

Caso queira, o cliente poderá visualizar relatórios e sumarizações de suas tarefas criadas, esse relatório será gerado pelo servidor de relatórios, se comunicando apenas em modo leitura, não sendo permitido alterações ou manipulações dos relatórios.

**[Insert client docs here]...**

**Servidor de tarefas**

**[Fix and update tasks server docs]...**

A aplicação do servidor de tarefas é uma API feita utilizando .Net 6 no qual possui um controlador responsável pelo CRUD das tarefas, contendo algumas rotas.

**GET**

A computer screen shot of a program code

Description automatically generated

Esse é a rota utilizada para obter todas as tarefas que já foram criadas, ela obtém todas as tarefas no banco de dados um status 200 OK contendo as mesmas, caso algum erro aconteça é retornado um status 500.

**GET com ID**

A computer screen shot of a program code

Description automatically generated

Essa é a rota utilizada para obter uma tarefa com um id específico, ela retorna um 200 OK com a tarefa caso a mesma seja encontrada, se a tarefa não for identificada na base de dados, é retornado um status 404 e por fim caso um erro aconteça será retornado um status 500.

**POST**

A computer screen with white and blue text

Description automatically generated

Rota utilizada para criação de uma tarefa, a mesma é inserida no banco e logo em seguida é retornado um status 200 com a tarefa criada, e caso algum erro aconteça é retornado um status 500.

A screen shot of a computer program

Description automatically generated**POST de concluído**

Rota utilizada para marcar uma tarefa como concluída ou como incompleta, nela é obtido a tarefa com o id enviado, se a tarefa for identificada no banco e não estiver concluída, a mesma é finalizada e retornado um status 200 com a tarefa atualizada, e vice-versa caso a tarefa já esteja concluída. Caso um erro aconteça é retornado um status 500.

**PUT**

A screen shot of a computer program

Description automatically generated

Rota utilizada para atualizar a descrição e/ou título de uma tarefa, caso a mesma seja identificada no banco de dados, é retornado um status 200 contendo a tarefa atualizada ou status 500 caso um erro aconteça.

A computer screen with white and green text

Description automatically generated**DELETE**

Rota utilizada para deletar uma tarefa, é recebido o id da tarefa que deve ser deletada caso a mesma seja identificada, é retornado um status 200 caso a mesma seja deletada, um status 500 caso ocorra algum erro e status 404 caso a tarefa não seja identificada.

**Base de dados**

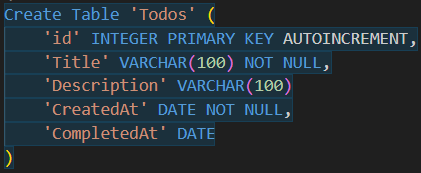
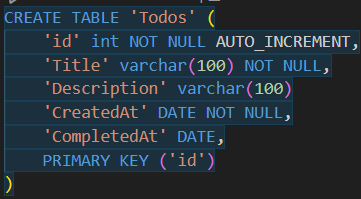
A base de dados utilizada foi o SQLite, o objetivo desse banco de dados ter sido escolhido foi pela facilidade de ter o mesmo em qualquer ambiente, por ser somente um arquivo torna se fácil a configuração e portabilidade dele. Esse banco contém uma única tabela para serem salvos o domínio tarefas, a configuração é feita automaticamente no momento que o servidor de tarefas é iniciado, ele roda o script abaixo:

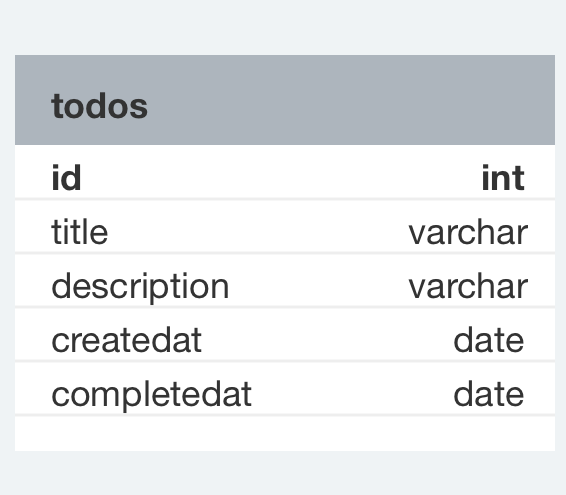
A computer screen shot of a black screen

Description automatically generated

**Script e modelo do banco de tarefas**

**SQL SQLite**

****

****

**Ferramenta ORM (Dapper)**

No servidor de tarefas foi utilizado a biblioteca Dapper, essa ferramenta é um micro ORM (Object Relational Mapping, ou Mapeamento Objeto Relacional em português) feita para mapear as colunas do banco de dados com as propriedades do nosso domínio tarefa, decidimos usar ele pela sua simplicidade de configuração e alta performance para execução de scripts no banco de dados.

**Domínio tarefa**

A screen shot of a computer program

Description automatically generated

Uma das principais partes do servidor de tarefas é o domínio tarefa em si, nele contém toda a lógica feita para registrar quando uma tarefa foi criada, concluída etc. Foi seguido as boas práticas de OO (Orientação a Objetos) deixando com que não seja uma estrutura fácil de ser manipulada externamente, o próprio domínio fornece todas as funcionalidades necessários para se modificar.

**Servidor de relatórios**

O servidor de relatórios é responsável pela geração de relatórios de sumarização das tarefas existentes no servidor de tarefas, realizando comunicação constante com esse servidor, que irá importar as tarefas existentes no servidor para o próprio banco de dados para otmização de desempenho e redução de fluxo de rede.

A aplicação do servidor de relatórios é uma API utilizando o framework Django v4.2.5 codificado na linguagem de programação Python v3.11.5. O framework 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 API possui três endpoints utilizados por uma requisição no de método GET:

**(GET) api/reports/count/**

Esse endpoint é responsável por realizar a contagem e a sumarização de todas as tarefas existentes no servidor de tarefas, contabilizando as seguintes métricas:

* A quantidade de tarefas criadas
* A quantidade de tarefas completas
* A quantidade de tarefas pendentes

Esse endpoint também exibe uma lista das tarefas completas e outra lista das tarefas pendentes, informando os seguintes dados em cada objeto:

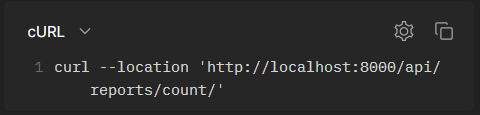
**Tarefas completas**

* ID da tarefa (chave primária no servidor de tarefas para referência do cliente)
* Quando a tarefa foi criada (permitindo filtros e ordenação conforme o cliente desejar)
* Quando a tarefa foi completa (permitindo filtros e ordenação conforme o cliente desejar)

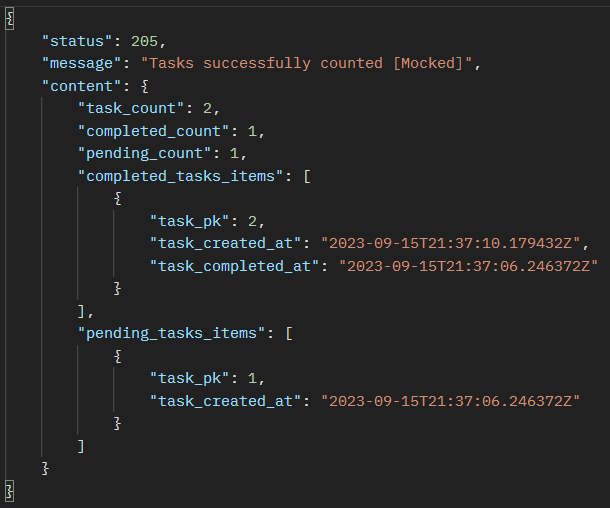
**Tarefas pendentes**

* ID da tarefa (chave primária no servidor de tarefas para referência do cliente)
* Quando a tarefa foi criada (permitindo filtros e ordenação conforme o cliente desejar)

**cURL de requisição do endpoint**

****

**Modelo de resposta do endpoint**



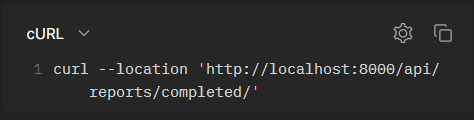
**(GET) api/reports/completed/**

Esse endpoint é responsável pela listagem de todas as tarefas completas existentes no servidor de tarefas, permitindo uma referência e uma liberdade maior para a requisição e gerenciamento de dados ao cliente.

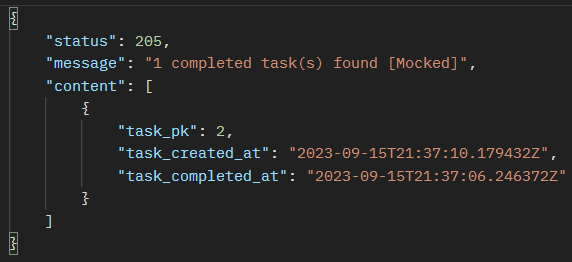
Cada objeto de resposta desse endpoint exibe as seguintes informações:

* ID da tarefa (chave primária no servidor de tarefas para referência do cliente)
* Quando a tarefa foi criada (permitindo filtros e ordenação conforme o cliente desejar)
* Quando a tarefa foi completa (permitindo filtros e ordenação conforme o cliente desejar)

**cURL de requisição do endpoint**

****

**Modelo de resposta do endpoint**

****

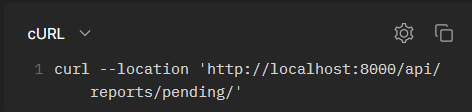
**(GET) api/reports/pending/**

Esse endpoint é responsável pela listagem de todas as tarefas pendentes existentes no servidor de tarefas, permitindo uma referência e uma liberdade maior para a requisição e gerenciamento de dados ao cliente.

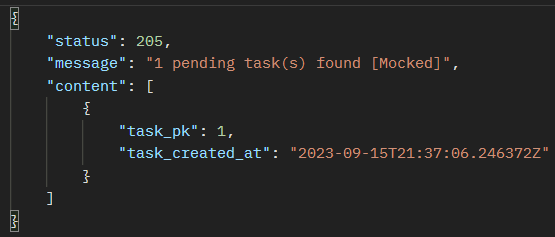
Cada objeto de resposta desse endpoint exibe as seguintes informações:

* ID da tarefa (chave primária no servidor de tarefas para referência do cliente)
* Quando a tarefa foi criada (permitindo filtros e ordenação conforme o cliente desejar)

**cURL de requisição do endpoint**

****

**Modelo de resposta**

****

**Banco de dados**

O banco de dados utilizado no servidor foi o sqlite, possuindo um arquivo .sqlite3 local com o intúito de facilitar o acesso e o armazenamento dos dados sem necessitar a requisição repetitiva ao servidor de tarefas.

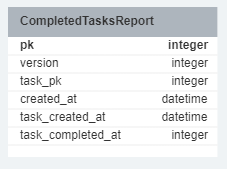
**Tabela de tarefas completas**

A tabela de tarefas completas consiste em cinco informações

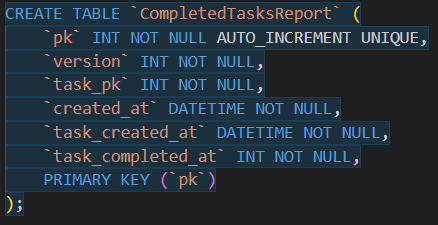
* version – Que representa o controle de versão para gerenciamento de concorrência.
* task\_pk – Que representa a chave primária da tarefa presente no servidor de tarefas
* created\_at – Que representa a data a qual a tarefa foi importada e criada no servidor de relatórios
* task\_created\_at – Que representa a data a qual a tarefa foi criada no servidor de tarefas
* task\_completed\_at – Que representa a data a qual a tarefa foi marcada como completa no servidor de tarefas

O método \_\_getattribute\_\_ é uma serialização customizada para retorno no endpoint de tarefas completas

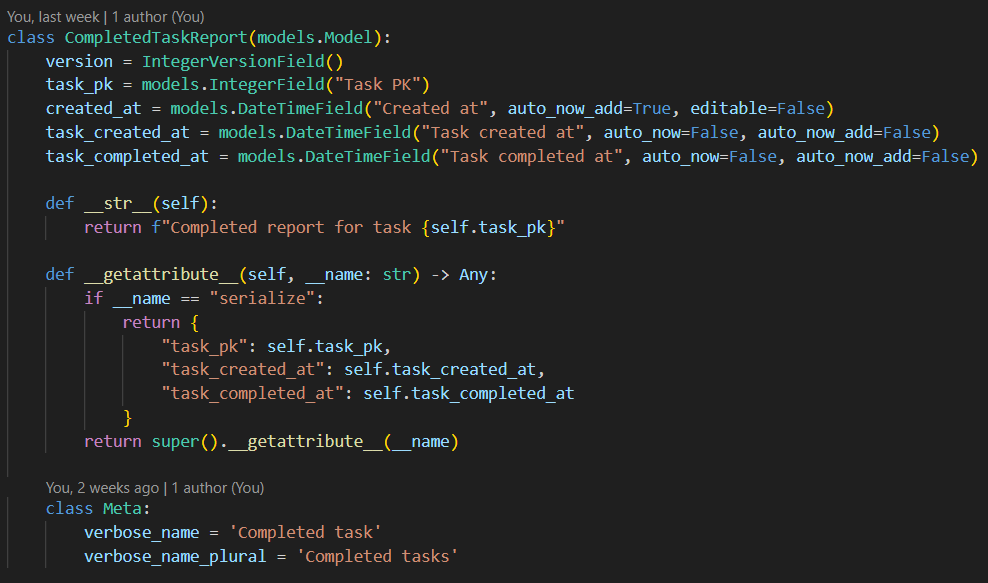
**Modelo da tabela no banco**



**Código para criação da tabela no banco**



**Declaração da tabela em Python no framework Django**

****

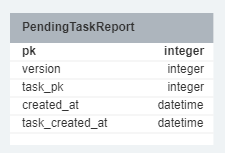
**Tabela de tarefas pendentes**

A tabela de tarefas pendentes consiste em quatro informações:

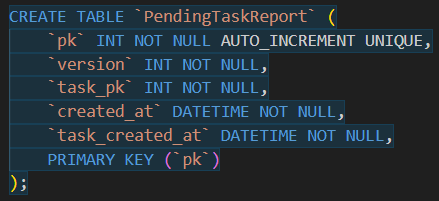
* version – Que representa o controle de versão para gerenciamento de concorrência.
* task\_pk – Que representa a chave primária da tarefa presente no servidor de tarefas
* created\_at – Que representa a data a qual a tarefa foi importada e criada no servidor de relatórios
* task\_created\_at – Que representa a data a qual a tarefa foi criada no servidor de tarefas

O método \_\_getattribute\_\_ é uma serialização customizada para retorno no endpoint de tarefas pendentes

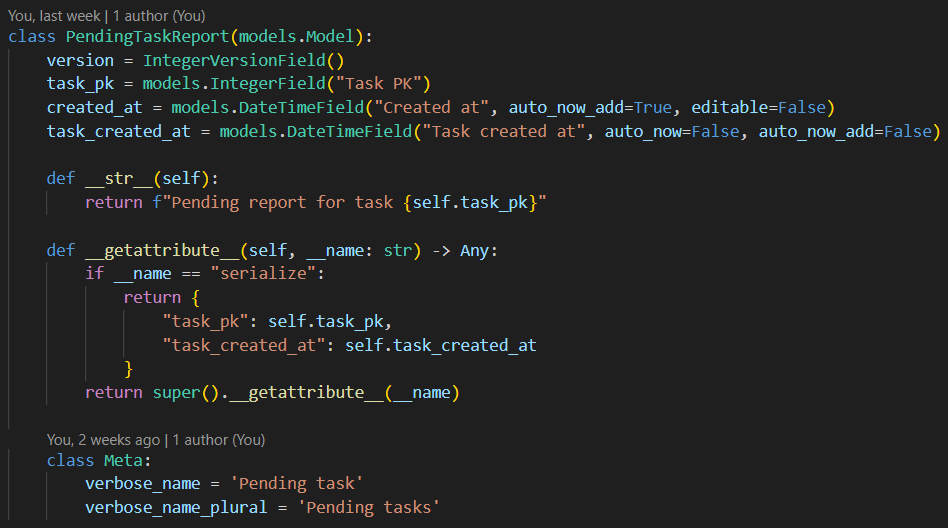
**Modelo da tabela no banco**



**Código para criação da tabela no banco**



**Declaração da tabela em Python no framework Django**

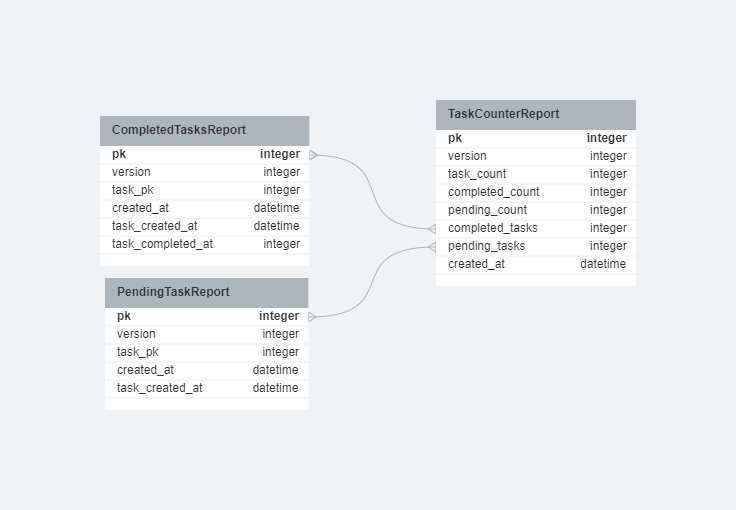


**Tabela de contagem de tarefas**

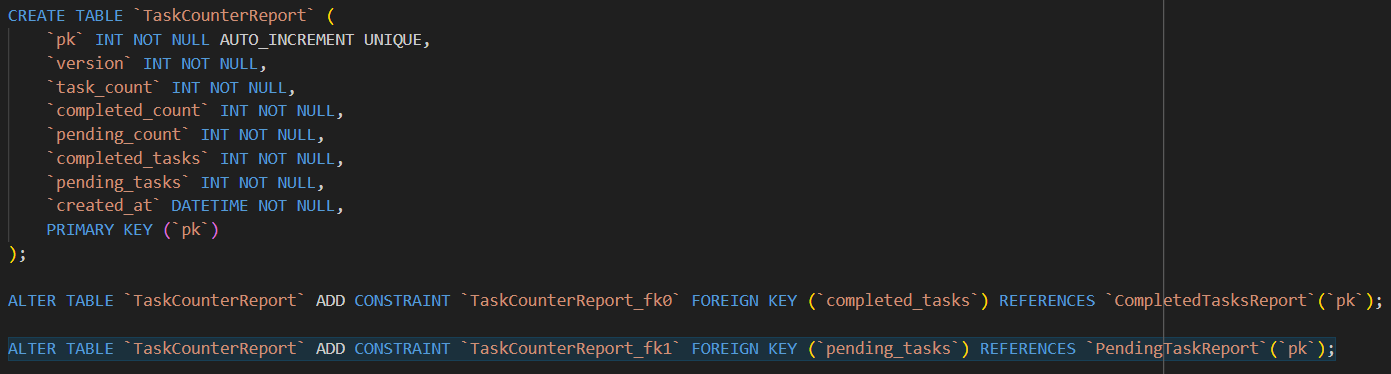
A tabela de contagem de tarefas consiste em sete informações

* version – Que representa o controle de versão para gerenciamento de concorrência.
* task\_count – Que representa a quantidade de tarefas existentes no banco de dados de tarefas, o task\_count possui o mesmo valor do resultado da soma do completed\_count + pending\_count
* completed\_count – Que representa a quantidade de tarefas completas existentes no banco de dados de tarefas
* pending\_count – Que representa a quantidade de tarefas pendentes existentes no banco de dados de tarefas
* completed\_tasks – Que representa a lista de todas as tarefas completas importadas para o banco de dados do servidor de relatórios
* pending\_tasks – Que representa a lista de todas as tarefas pendentes importadas para o banco de dados do servidor de relatórios
* created\_at – Que representa a data a qual as tarefas foram contabilizadas e atualizadas no servidor de relatórios

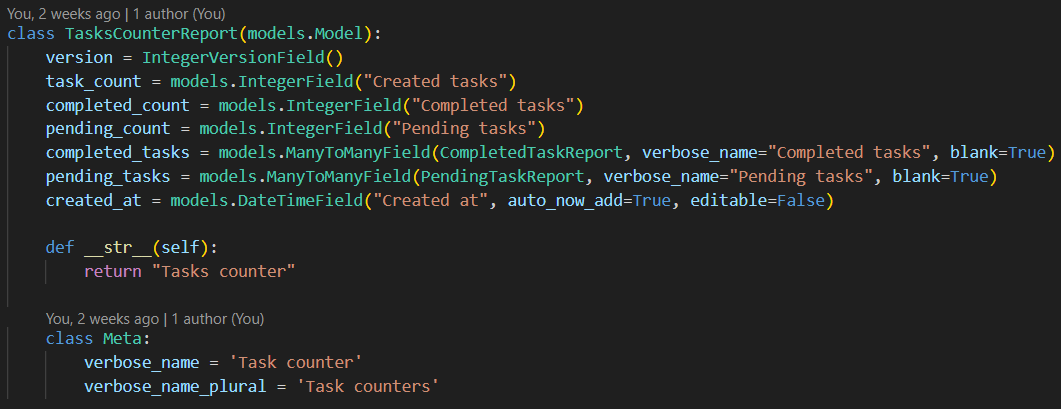
**Modelo da tabela no banco**



**Código para criação da tabela no banco**

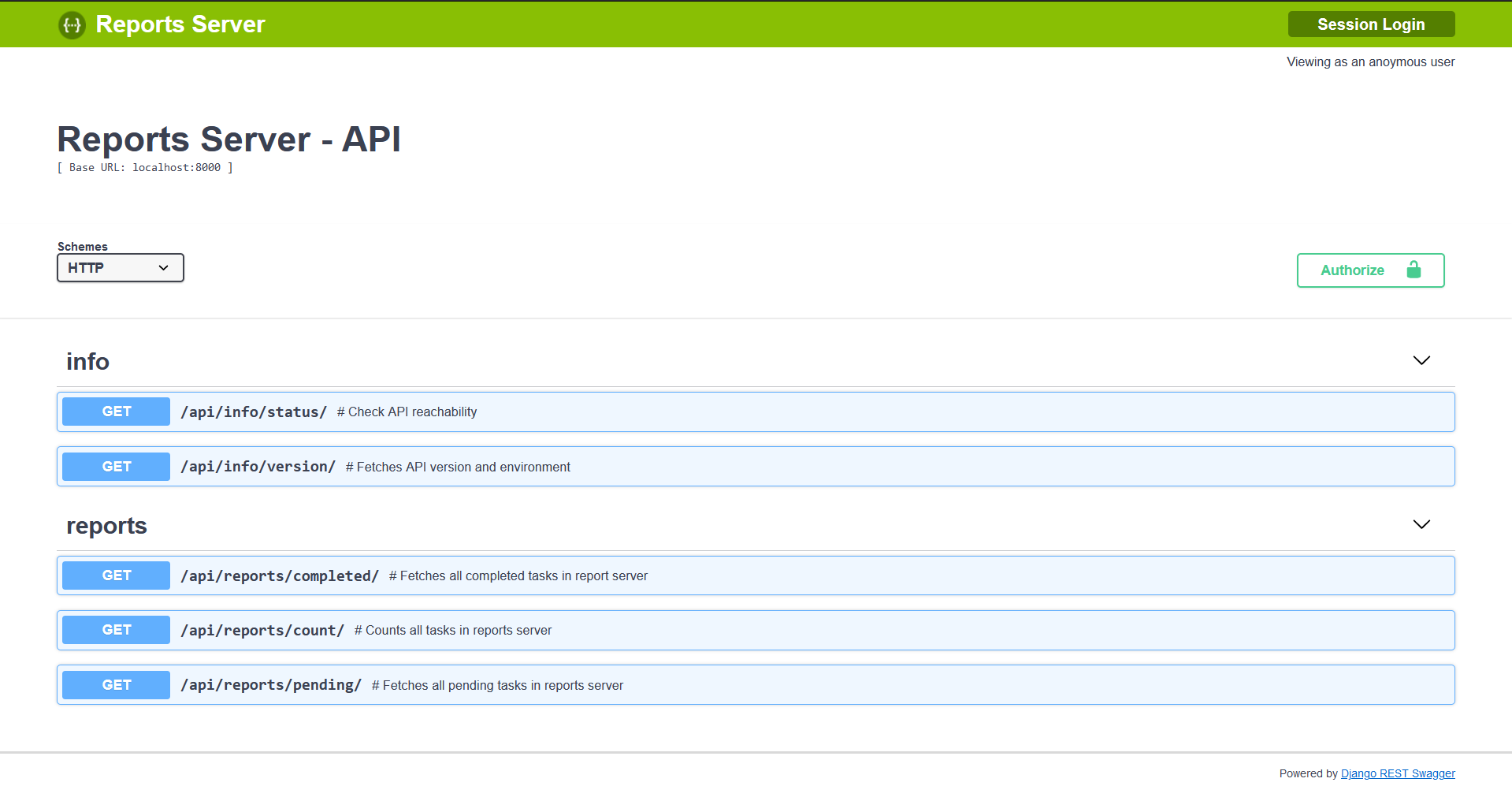


**Declaração da tabela em Python no framework Django**

****

**Documentação da API**

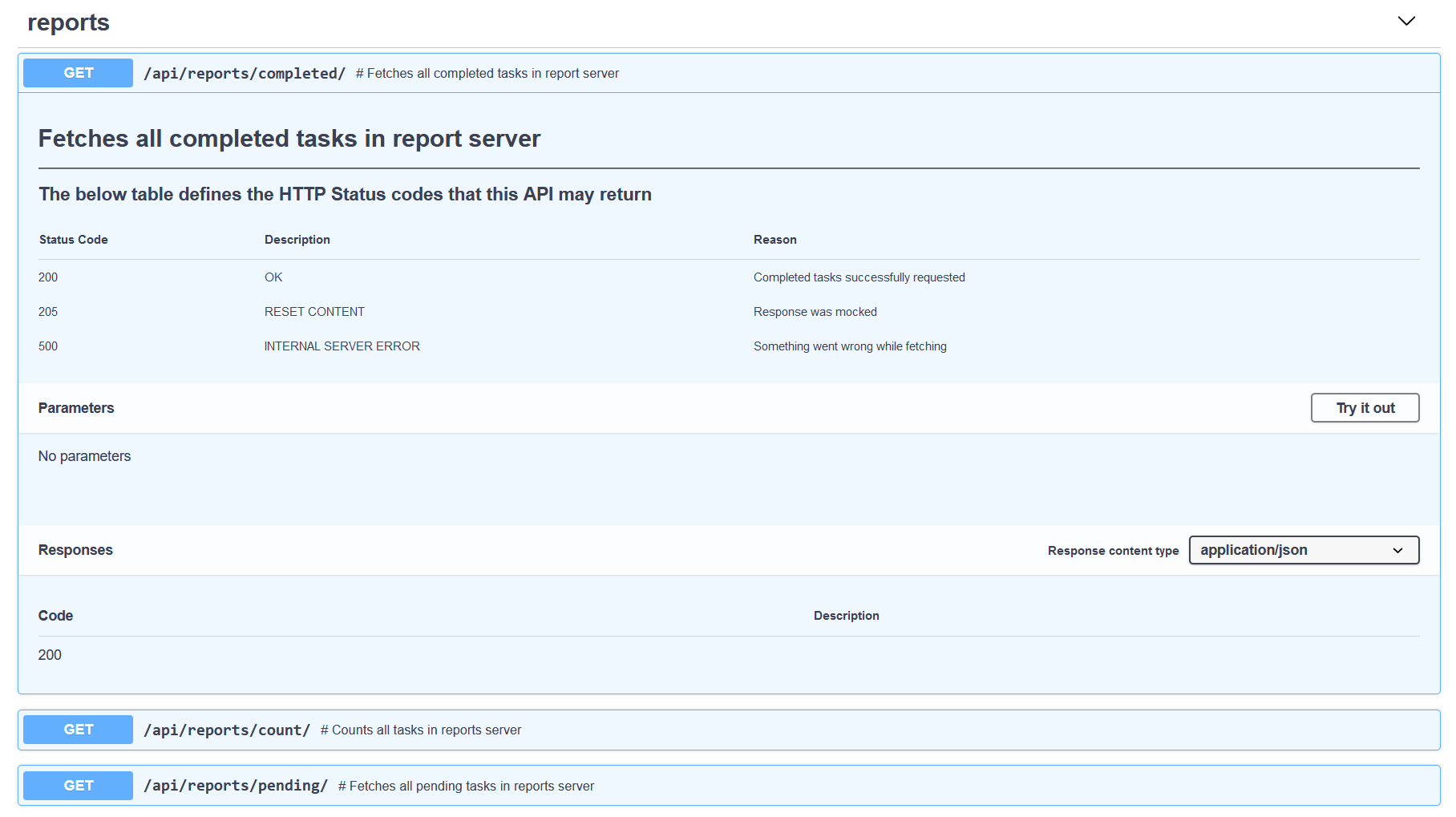
A documentação da API do servidor de relatórios está hospedada no próprio servidor no endereço [**http://localhost:8000/api/docs/**](http://localhost:8000/api/docs/), a documentação é exibida utilizando a interface gráfica do Swagger, sendo configurada e definida manualmente no código.



O framework do Django permite que a página seja fácilmente exibida utilizando a biblioteca django\_rest\_swagger sem necessitar de um trabalho muito complexo para poder renderizar. Para permitir documentar corretamente o endpoint devemos cumprir dois pré-requisitos:

1. Gerar a estrutuda do Swagger de forma que a biblioteca possa interpretar e gerar corretamente os dados.
2. Um endpoint real para que possa ser adquirido o endereço e configurada a funcionalidade de “Try out”

**Documentação renderizada na UI do swagger**



**Declaração da configuração em Python para exibição no Swagger**

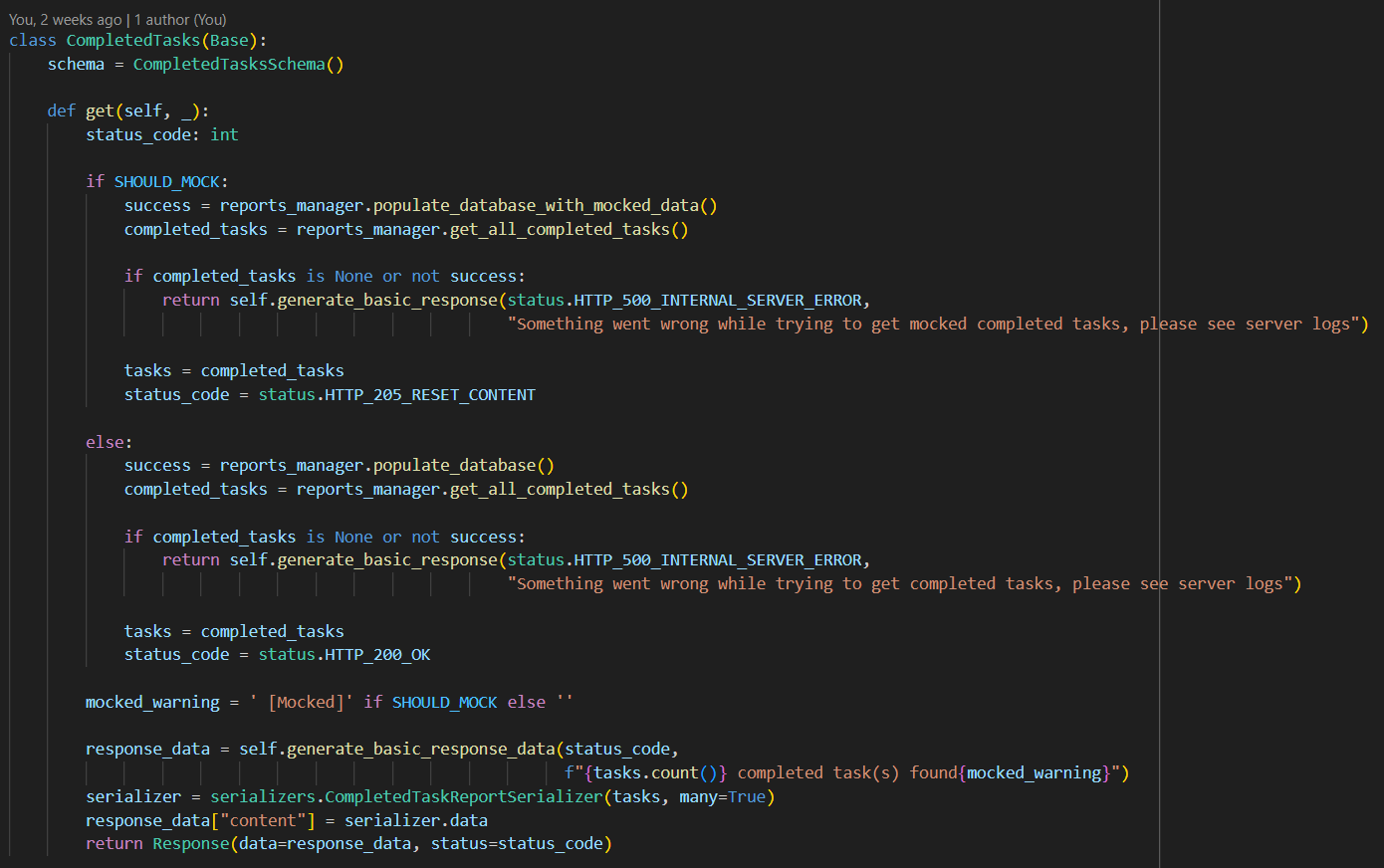


**Declaração e funcionamento da API**

Todos os endpoints possuem o mesmo esquema de declaração e funcionamento da API, o que difere entre um endpoint e outro é a complexidade da lógica para aquisição dos dados.

Para facilitar a implementação do servidor de relatórios, implementamos uma lógica de mock de resposta do servidor de tarefas para testes, otimizações e atualizações. O Mock irá retornar um dicionário no mesmo formato retornado pelo endpoint do servidor de tarefas, permitindo utilizar o mesmo fluxo sem precisar de muita lógica.

Como exemplo, utilizaremos a declaração do endpoint de tarefas completas:



A arquitetura utilizada é class based views, utilizando programação orientada a objetos, permitindo uma modularização, definição, melhor manutenção etc.

Na classe, existe a declaração do schema para ser exibido na UI de documentação do Swagger. A função get define o método de requisição do endpoint, ou seja, ao realizar uma requisição GET no endpoint **api/reports/completed**, a função invocada pelo framework do rest será a get.

Dentro da função, é definida uma variável para utilização futura **status\_code**, essa variável irá conter o código de resposta do endpoint, caso o conteúdo tenha sido mockado, o status será 205, caso seja uma requisição ao servidor o status será 200, caso ocorra qualquer erro durante a requisição e clonagem de tarefas será retornado o status 500.

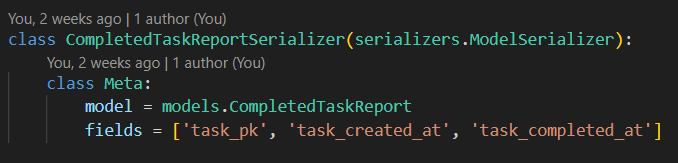
Ambos os fluxos, mockado ou não são os mesmos, o que irá diferenciar é a função que é chamada para requisitar os dados, no caso de uma requisição real para o servidor de tarefas é realizada a população do banco de dados, essa função será responsável por popular o banco de dados do servidor de relatórios com as tarefas existentes no servidor de tarefas.

Depois de populado, é realizada a chamada da função para buscar todas as tarefas completas, para otimização de desempenho e evitar um constante fluxo de rede toda a lógica de ordenação, contagem, filtragem entre outros é realizado utilizando os dados do banco de dado local importado anteriormente do servidor de tarefas.

Em seguida, é realizada a validação dos dados adquiridos, caso não sejam válidos, é retornado uma resposta de erro 500 com uma mensagem que descreve o motivo de ter retornado o erro. Em seguida, é armazenado no escopo local da função as tarefas buscadas e o status definido para 200.

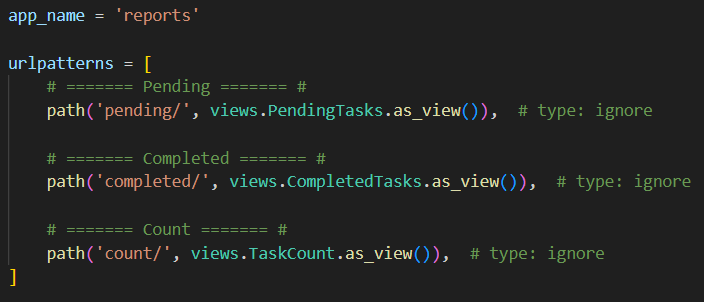
Em seguida, para evitar confusões, caso a busca tenha sido mockada, a mensagem de sucesso é enviada com um sufixo **[Mocked]**, para explicitar de que o conteúdo não condiz com o que realmente contém no servidor de tarefas.

Por fim, os objetos e seus conteúdos são serializados para um formato JSON que pode ser enviado como uma resposta HTTP, a função de serialização utilizada é disponibilizada pela biblioteca do rest, permitindo uma completa compatibilidade entre a resposta e seu conteúdo, abaixo, como exemplo, a declaração da função de serialização do endpoint das tarefas completas.



A classe consiste em uma meta classe que irá explicitar as informações necessárias para serem serializadas. A variável **model** irá definir qual a tabela do banco de dados que deve ter seus dados serializados (vide seção “Banco de dados” para mais informações). Já a variável **fields** irá explicitar para a biblioteca quais são os dados dessa tabela que devem ser serializados e enviados, permitindo retornar dados específicos e omitir informações internas.

Por fim, o esquema de URLs e caminhos do servidor é definido utilizando a própria estrutura do framework do Django, permitindo que a biblioteca do rest utilize o framework para redirecionar chamadas e requisições para o endpoint e função correto conforme declarado.



A variável **app\_name** é utilizada para definir o escopo das urls e permitir um redirecionamento mais específico. A variável **urlpatterns** é a lista de endpoints que serão utilizados para esse escopo, a string presente dentro da função path é a string que irá aparecer na url, como por exemplo, a primeira declaração com a string ‘pending/’ irá gerar a url **api/reports/pending/.**