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

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

```
cURL
curl --location 'http://localhost:8000/api/
    reports/count/'
```

Modelo de resposta do endpoint


```
{
    "status": 205,
    "message": "Tasks successfully counted [Mocked]",
    "content": {
        "task_count": 2,
        "completed_count": 1,
        "pending_count": 1,
        "completed_tasks_items": [
            {
                "task_pk": 2,
                "task_created_at": "2023-09-15T21:37:10.179432Z",
                "task_completed_at": "2023-09-15T21:37:06.246372Z"
            }
        ],
        "pending_tasks_items": [
            {
                "task_pk": 1,
                "task_created_at": "2023-09-15T21:37:06.246372Z"
            }
        ]
    }
}
```

(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

```
cURL
curl --location 'http://localhost:8000/api/
    reports/completed/'
```

Modelo de resposta do endpoint


```
{
    "status": 205,
    "message": "1 completed task(s) found [Mocked]",
    "content": [
        {
            "task_pk": 2,
            "task_created_at": "2023-09-15T21:37:10.179432Z",
            "task_completed_at": "2023-09-15T21:37:06.246372Z"
        }
    ]
}
```

(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

```
cURL
curl --location 'http://localhost:8000/api/
    reports/pending/'
```

Modelo de resposta


```
{
    "status": 205,
    "message": "1 pending task(s) found [Mocked]",
    "content": [
        {
            "task_pk": 1,
            "task_created_at": "2023-09-15T21:37:06.246372Z"
        }
    ]
}
```

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.

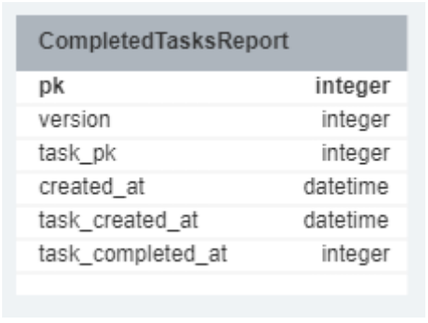
Tabela de tarefas completas

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



CompletedTasksReport	
pk	**integer**
version	integer
task_pk	integer
created_at	datetime
task_created_at	datetime
task_completed_at	integer

Código para criação da tabela no banco

```
CREATE TABLE `CompletedTasksReport` (
    `pk` INT NOT NULL AUTO_INCREMENT UNIQUE,
    `version` INT NOT NULL,
    `task_pk` INT NOT NULL,
    `created_at` DATETIME NOT NULL,
    `task_created_at` DATETIME NOT NULL,
    `task_completed_at` INT NOT NULL,
    PRIMARY KEY (`pk`)
);
```

Declaração da tabela em Python no framework Django

```python
You, last week | 1 author (You)
class CompletedTaskReport(models.Model):
    version = IntegerVersionField()
    task_pk = models.IntegerField("Task PK")
    created_at = models.DateTimeField("Created at", auto_now_add=True, editable=False)
    task_created_at = models.DateTimeField("Task created at", auto_now=False, auto_now_add=False)
    task_completed_at = models.DateTimeField("Task completed at", auto_now=False, auto_now_add=False)

    def __str__(self):
        return f"Completed report for task {self.task_pk}"

    def __getattribute__(self, __name: str) -> Any:
        if __name == "serialize":
            return {
                "task_pk": self.task_pk,
                "task_created_at": self.task_created_at,
                "task_completed_at": self.task_completed_at
            }
        return super().__getattribute__(__name)

    You, 2 weeks ago | 1 author (You)
    class Meta:
        verbose_name = 'Completed task'
        verbose_name_plural = 'Completed tasks'
```

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

PendingTaskReport	
pk	**integer**
version	integer
task_pk	integer
created_at	datetime
task_created_at	datetime

Código para criação da tabela no banco

```
CREATE TABLE `PendingTaskReport` (
    `pk` INT NOT NULL AUTO_INCREMENT UNIQUE,
    `version` INT NOT NULL,
    `task_pk` INT NOT NULL,
    `created_at` DATETIME NOT NULL,
    `task_created_at` DATETIME NOT NULL,
    PRIMARY KEY (`pk`)
);
```

Declaração da tabela em Python no framework Django

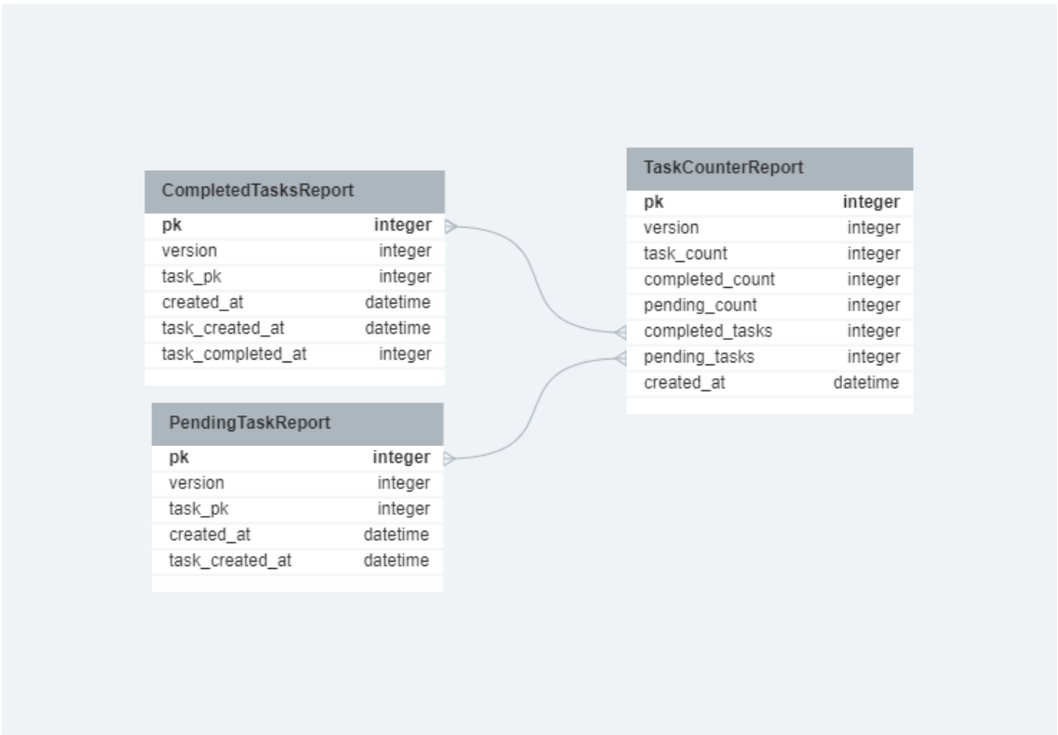
```python
You, last week | 1 author (You)
class PendingTaskReport(models.Model):
    version = IntegerVersionField()
    task_pk = models.IntegerField("Task PK")
    created_at = models.DateTimeField("Created at", auto_now_add=True, editable=False)
    task_created_at = models.DateTimeField("Task created at", auto_now=False, auto_now_add=False)

    def __str__(self):
        return f"Pending report for task {self.task_pk}"

    def __getattribute__(self, __name: str) -> Any:
        if __name == "serialize":
            return {
                "task_pk": self.task_pk,
                "task_created_at": self.task_created_at
            }
        return super().__getattribute__(__name)

    You, 2 weeks ago | 1 author (You)
    class Meta:
        verbose_name = 'Pending task'
        verbose_name_plural = 'Pending tasks'
```

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

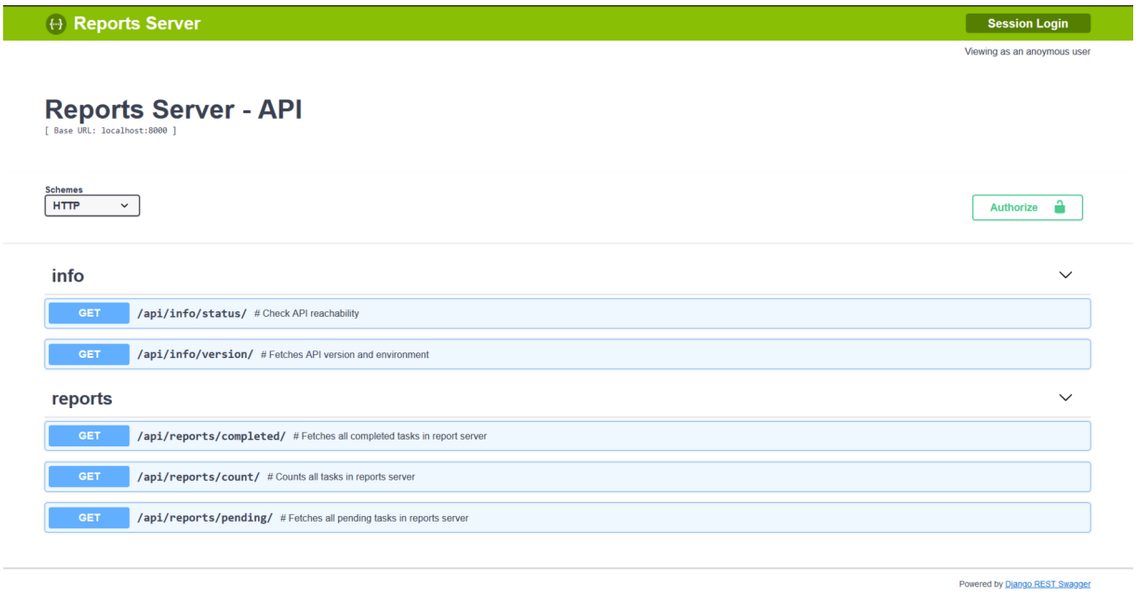
```sql
CREATE TABLE `TaskCounterReport` (
    `pk` INT NOT NULL AUTO_INCREMENT UNIQUE,
    `version` INT NOT NULL,
    `task_count` INT NOT NULL,
    `completed_count` INT NOT NULL,
    `pending_count` INT NOT NULL,
    `completed_tasks` INT NOT NULL,
    `pending_tasks` INT NOT NULL,
    `created_at` DATETIME NOT NULL,
    PRIMARY KEY (`pk`)
);

ALTER TABLE `TaskCounterReport` ADD CONSTRAINT `TaskCounterReport_fk0` FOREIGN KEY (`completed_tasks`) REFERENCES `CompletedTasksReport`(`pk`);

ALTER TABLE `TaskCounterReport` ADD CONSTRAINT `TaskCounterReport_fk1` FOREIGN KEY (`pending_tasks`) REFERENCES `PendingTaskReport`(`pk`);
```

Declaração da tabela em Python no framework Django

```python
You, 2 weeks ago | 1 author (You)
class TasksCounterReport(models.Model):
    version = IntegerVersionField()
    task_count = models.IntegerField("Created tasks")
    completed_count = models.IntegerField("Completed tasks")
    pending_count = models.IntegerField("Pending tasks")
    completed_tasks = models.ManyToManyField(CompletedTaskReport, verbose_name="Completed tasks", blank=True)
    pending_tasks = models.ManyToManyField(PendingTaskReport, verbose_name="Pending tasks", blank=True)
    created_at = models.DateTimeField("Created at", auto_now_add=True, editable=False)

    def __str__(self):
        return "Tasks counter"

    You, 2 weeks ago | 1 author (You)
    class Meta:
        verbose_name = 'Task counter'
        verbose_name_plural = 'Task counters'
```

Documentação da API

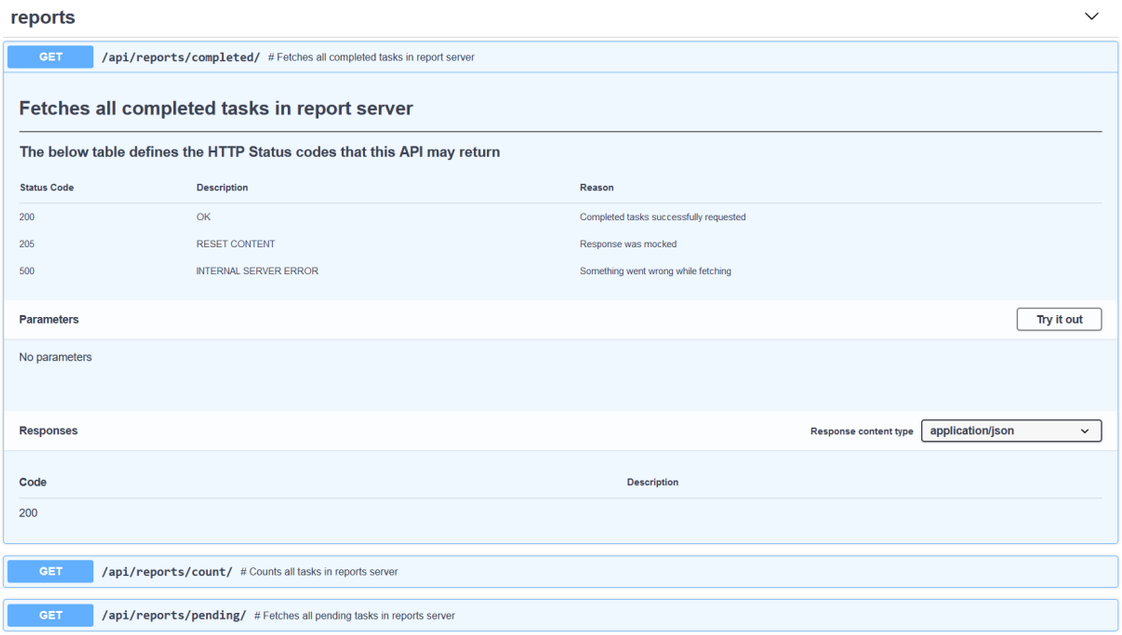
A documentação da API do servidor de relatórios está hospedada no próprio servidor no endereço **http://localhost:8000/api/docs/**, a documentação é exibida utilizando a interface gráfica do Swagger, sendo configurada e definida manualmente no código.



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

```python
# ============== Completed ============== #
You, 2 weeks ago | 1 author (You)
class CompletedTasksSchema(AutoSchema):
    def get_description(self, path: str, method: str) -> str:
        match method:
            case 'GET':
                responses = {
                    "200": {
                        'description': 'OK',
                        'reason': 'Completed tasks successfully requested'
                    },
                    "205": {
                        'description': 'RESET CONTENT',
                        'reason': 'Response was mocked'
                    },
                    "500": {
                        'description': 'INTERNAL SERVER ERROR',
                        'reason': 'Something went wrong while fetching'
                    }
                }
                return description_generator(title="Fetches all completed tasks in report server",
                                             description='',
                                             responses=responses)
            case _:
                return ''

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

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:

```python
You, 2 weeks ago | 1 author (You)
class CompletedTasks(Base):
    schema = CompletedTasksSchema()

    def get(self, _):
        status_code: int

        if SHOULD_MOCK:
            success = reports_manager.populate_database_with_mocked_data()
            completed_tasks = reports_manager.get_all_completed_tasks()

            if completed_tasks is None or not success:
                return self.generate_basic_response(status.HTTP_500_INTERNAL_SERVER_ERROR,
                                                    "Something went wrong while trying to get mocked completed tasks, please see server logs")

            tasks = completed_tasks
            status_code = status.HTTP_205_RESET_CONTENT

        else:
            success = reports_manager.populate_database()
            completed_tasks = reports_manager.get_all_completed_tasks()

            if completed_tasks is None or not success:
                return self.generate_basic_response(status.HTTP_500_INTERNAL_SERVER_ERROR,
                                                    "Something went wrong while trying to get completed tasks, please see server logs")

            tasks = completed_tasks
            status_code = status.HTTP_200_OK

        mocked_warning = ' [Mocked]' if SHOULD_MOCK else ''

        response_data = self.generate_basic_response_data(status_code,
                                                          f"{tasks.count()} completed task(s) found{mocked_warning}")
        serializer = serializers.CompletedTaskReportSerializer(tasks, many=True)
        response_data["content"] = serializer.data
        return Response(data=response_data, status=status_code)
```

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.

```
You, 2 weeks ago | 1 author (You)
class CompletedTaskReportSerializer(serializers.ModelSerializer):
    You, 2 weeks ago | 1 author (You)
    class Meta:
        model = models.CompletedTaskReport
        fields = ['task_pk', 'task_created_at', 'task_completed_at']
```

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.

```
app_name = 'reports'

urlpatterns = [
    # ======= Pending ======= #
    path('pending/', views.PendingTasks.as_view()),  # type: ignore

    # ======= Completed ======= #
    path('completed/', views.CompletedTasks.as_view()),  # type: ignore

    # ======= Count ======= #
    path('count/', views.TaskCount.as_view()),  # type: ignore
]
```

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/.**