Universidade Federal de Pernambuco
Centro de Informática

Ciência da Computação

Django-SSTenants - Uma ferramenta para construir aplicações *Multi-Tenants*

Hugo Rafael Bessa de Andrade

Trabalho de Graduação

Recife
25 de Junho de 2018

Universidade Federal de Pernambuco
Centro de Informática

Hugo Rafael Bessa de Andrade

Django-SSTenants - Uma ferramenta para construir aplicações *Multi-Tenants*

Trabalho de Conclusão do Curso de Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco .

Orientador: *Nelson Souto Rosa*

Recife
25 de Junho de 2018

"Os primeiros 90% de código escritos levam os primeiros 90% do tempo de desenvolvimento... os 10% de código restantes levam outros 90% do tempo de desenvolvimento."
Tom Cargill

Resumo

A popularização dos softwares como serviço e a crescente flexibilidade de serviços de infraestrutura em nuvem tornaram mais comum a necessidade de se pensar em aplicações que suportem múltiplos clientes (*tenants*) sem que seja necessário aumentar rapidamente a infraestrutura física (hardware). Essas aplicações, além de trazerem diminuição de custos de infraestrutura e manutenção, facilitam a implementação de entrega contínua. Porém, essa mudança trouxe, além de grandes benefícios, algumas preocupações. Segurança da informação, distribuição de recursos, controle de acesso, desempenho, e a extração de métricas, são alguns dos desafios que aparecem nessa arquitetura. Neste trabalho será desenvolvida e avaliada uma biblioteca, chamada *Django-SSTenants*, para auxiliar desenvolvedores Web a construir e migrar aplicações baseadas no *framework Django* em arquiteturas *multi-tenants* lidando com esses desafios. A ferramenta buscará adicionar pouca complexidade à base de código e garantir qualidade em critérios como segurança, performance e extensibilidade.

Palavras-chave: *Multi-tenancy*, *Multi-tenant*, Django, Desenvolvimento Web, *Open source*.

Abstract

The popularisation of software as a service and the increased flexibility of cloud computing infrastructure services have made it more usual to think of applications that support multiple clients (tenants) without the need of rapidly growing hardware infrastructure. These applications, facilitate the implementation of continuous delivery in addition to reducing infrastructure and maintenance costs. But this change has brought, in addition to significant benefits, some concerns. Information security, resource distribution, access control, performance, metrics extraction are some of the challenges that come to light in this architecture. In this work, we present and assess a library called Django-SSTenants to help Web developers to build and migrate applications based on the Django framework to multi-tenant architecture dealing with these challenges. The tool adds the minimum of complexity to the code base and ensures the quality of criteria such as security, performance, and extensibility.

Keywords: Multi-tenancy, Multi-tenant, Django, Web development, Open source

Sumário

CAPÍTULO 1

Introdução

1.1 Motivação

A popularização dos softwares como serviço (*Software as a Service* - SaaS) e a crescente flexibilidade de serviços de infraestrutura em nuvem tornaram mais comum a necessidade de se pensar em aplicações que suportem múltiplos clientes (*tenants*) sem que seja necessário aumentar rapidamente a infraestrutura física (hardware). Essas aplicações, além de trazerem diminuição de custos de infraestrutura e manutenção, facilitam a implementação de entrega contínua.

Porém, essa mudança mesmo sendo geradora de grandes benefícios, também fez com que algumas preocupações se tornem mais frequentes: a segurança das informações, a distribuição de recursos, o controle de acesso, o desempenho, a extração de métricas, e assim por diante. Além disso, aparecem também questões que impactam diretamente o dia-a-dia dos desenvolvedores, como aumento da complexidade do código e dos testes.

Todos esses pontos tornam a construção de sistemas com suporte a *multi-tenant* um desafio complexo e muitas vezes árduo para engenheiros de software. Mas, muitos desses pontos podem ser minimizados através da utilização de abordagens e ferramentas adequadas.

O *framework* Django [Fou05], tem se tornado cada vez mais popular no desenvolvimento de aplicações web e já possui algumas ferramentas para auxiliar no desenvolvimento de aplicações *multi-tenant*, como é o caso do pacote de código aberto *Django-Tenants*[BP17]. Porém, as implementações de aplicações *multi-tenant* podem ser baseadas em diversas abordagens [HDX12], cada uma com suas vantagens e desvantagens.

Nesse trabalho foi construída uma biblioteca que abstrai preocupações da arquitetura *multi-tenant*, a *Django-SSTenants*. A principal motivação para construção da mesma foi a escassez de ferramentas de código aberto bem documentadas, e cujas APIs cobrem todos os aspectos necessários ao desenvolvimento de aplicações *multi-tenant* desenvolvidas utilizando *Django*.

1.2 Objetivos

O objetivo deste trabalho é construir uma biblioteca para o *framework* web *Django* com a finalidade de auxiliar seus usuários a começar novas aplicações com suporte a *multi-tenant* ou migrar suas aplicações existentes para a arquitetura de maneira suave, sem muitas modificações em seu código fonte.

O principal foco dessa biblioteca é minimizar a preocupação dos seus usuários com regras de *multi-tenant*, e garantir uma boa integração aos padrões de desenvolvimento *Django*. Dessa

maneira, desenvolvedores poderão aplicar os conceitos de *multi-tenant* em suas aplicações sem se preocupar com regras específicas da arquitetura, e focando principalmente nas regras do negócio.

O desenvolvimento da biblioteca se deu através das seguintes etapas:

1. Análise de referências sobre *multi-tenant* de maneira a entender suas especificidades e possibilidades de implementação, tendo em vista as vantagens e desvantagens de cada abordagem;

2. Definição das abordagens selecionadas para implementação de uma biblioteca focando principalmente na simplicidade de implementação e da infraestrutura necessária para construção da aplicação;

3. Análise da arquitetura e dos padrões de código utilizados no *framework Django* de maneira a construir uma biblioteca que faça uso desses padrões e garanta que o usuário final terá disponível uma Interface de Programação de Aplicações (*Application Programming Interface* - API) que lembre a API do próprio *framework*;

4. Desenvolvimento da biblioteca *Django-SSTenants*;

5. Análise comparativa da biblioteca em relação a outras bibliotecas similares existentes no mercado.

1.3 Organização do Trabalho

Este trabalho está organizado da seguinte forma:

Capítulo 2 - Conceitos Básicos: Este capítulo apresenta todos os conceitos essenciais para o entendimento deste trabalho;

Capítulo 3 - Implementação da *Django-SSTenants*: Nesse capítulo será mostrado como foi implementada a biblioteca e a forma com qual ela integra a arquitetura *multi-tenant* ao *Django*;

Capítulo 4 - Avaliação da *Django-SSTenants*: Nesse capítulo será feita uma análise das ferramentas de código aberto disponíveis para implementação da arquitetura *multi-tenant* e uma avaliação da *Django-SSTenants* em relação a quantidade de funcionalidades, complexidade de implantação e uma comparação de desempenho;

Capítulo 5 - Conclusões e Trabalhos Futuros: Nesse capítulo faremos as considerações finais sobre esse trabalho e os planos de desenvolvimento futuro da biblioteca.

CAPÍTULO 2

Conceitos Básicos

Nesse capítulo serão apresentados os conceitos necessários para implementação e avaliação da biblioteca *Django-SSTenants*. Primeiramente falaremos sobre o conceito de *Multi-tenancy*, e posteriormente sobre o *framework* web *Django*.

2.1 Multi-tenancy

De acordo com [RKK12], um *tenant* é um grupo de usuários que compartilham uma mesma visão de uma aplicação. Essa visão inclui os dados que eles acessam, a configuração, a gestão de usuários, funcionalidades específicas e características não-funcionais relacionadas.

Também de acordo com [RKK12], *Multi-tenancy* é uma abordagem para dividir uma instância de uma aplicação entre vários *tenants* provendo a cada um deles uma parte dedicada da instância, a qual é isolada das outras partes em relação o desempenho e privacidade dos dados.

Essa abordagem evita a necessidade de mudanças no código fonte assim como a criação de novas instâncias da aplicação para adição de um novo (*tenant*) no sistema, facilitando a automação do processo. A automação influencia diretamente na escalabilidade do software já que, se o próprio cliente pode ter a liberdade de iniciar e configurar seu próprio *tenant*, diminui-se o custo da operação por aquisição de clientes.

As tecnologias de computação em nuvem são uma realidade atualmente. Temos muitos serviços, cada vez mais customizáveis e fáceis de configurar, que nos permitem criar e remover servidores de forma rápida e com custos bastante acessíveis.

Essa maleabilidade da infraestrutura na qual o software é executado traz com ela muitas possibilidades, sendo uma delas a construção de serviços elásticos, que regulam o tamanho da sua infraestrutura disponível de acordo com a demanda. Isso faz com que tanto os clientes estejam satisfeitos com a disponibilidade/desempenho do software quanto os custos de infraestrutura sejam os menores possíveis através de uma boa gestão de recursos.

Essa facilidade de construir softwares em serviços elásticos acaba trazendo mais à tona a arquitetura *multi-tenant*. A facilidade de escalar a infraestrutura sempre que é necessário mais processamento para suprir as necessidades dos usuários, torna dispensável se preocupar sempre que se vai disponibilizar um software web para novos *tenants*.

A arquitetura *multi-tenant*, além de trazer grandes ganhos à escalabilidade de aplicações, também possui ganhos no processo de entrega contínua. Como temos apenas uma instância, ou, em alguns casos, um conjunto de instâncias exatamente iguais, o processo de atualização do software fica bem mais simples, podendo inclusive ser automatizado. Dessa forma, o desenvolvedor se preocupa menos com as atualizações e consegue focar mais nas novas funcionalidades

e melhorias do software.

O fato de ter apenas uma ou várias instância iguais também facilita o processo de manutenção, visto que é mais fácil reproduzir erros em ambiente de desenvolvimento, já que sabe-se exatamente como está o estado da aplicação em produção.

Além de adicionar grandes possibilidades para a escala de aplicações web e para o processo de desenvolvimento e manutenção do software, a arquitetura *multi-tenant* também apresenta vários desafios técnicos que precisam ser observados.

2.1.1 Desafios na Arquitetura *Multi-tenant*

Os principais desafios das arquiteturas *multi-tenant* são: garantir que a infraestrutura cresça proporcionalmente ao uso do software, e não a quantidade de *tenants*, a segurança dos dados, a facilidade de manutenção da aplicação e da infraestrutura, e que o software atenda as particularidades de cada *tenant*.

Crescimento da Infraestrutura

A infraestrutura necessária para uma aplicação *multi-tenant* varia de acordo com a abordagem que for utilizada, podendo exigir desde um servidor por *tenant*, até apenas um servidor para todos os *tenants*.

Sistemas que precisam garantir uma maior segurança na separação de dados e na distribuição de recursos para seus *tenants* tendem a requerer maior infraestrutura, visto que a melhor forma de garantir que não há vazamento é não compartilhar nada. Porém, isso aumenta também o custo, pois aumenta a probabilidade de haver recursos ociosos e aumenta o tamanho da operação necessária para garantir que a infraestrutura suporta a demanda de cada *tenant*.

Em contraponto, sistemas que possuem requisitos mais flexíveis em relação à separação de dados e divisão de recursos, podem utilizar abordagens que compartilham mais recursos garantindo melhor aproveitamento da infraestrutura disponível e garantindo um crescimento mais sustentável, e.g., atualizações e manutenção da infraestrutura mais baratos.

Segurança dos Dados

A segurança dos dados está muito atrelada à garantia de que um determinado usuário do sistema consiga ter acesso apenas à leitura e à escrita dos dados dos *tenants* cujas respectivas permissões lhe foram dadas.

A segurança também está relacionada à garantia de que os dados de um *tenant* nunca se misturem com os dados de outro *tenant* sem que seja essa a intenção. Esse critério é muito importante, pois muitas organizações armazenam informações sensíveis nos softwares que utilizam. Se houver algum vazamento dessas informações, essas organizações podem ser bastante prejudicadas, inclusive financeiramente.

Em relação às permissões, uma arquitetura muito popular é a de Controle de Acesso de Baseado em Papéis (*Role Based Access Control* - RBAC) [FK92]. Essa arquitetura baseia as permissões de acesso ao sistema nos papéis que o usuário possui. Numa aplicação *multi-tenant* é preciso considerar também se o usuário tem os papéis/permissões necessárias naquele *tenant*.

Manutenibilidade

A manutenibilidade está muito relacionada à infraestrutura necessária à execução do software, pois ela influencia diretamente na complexidade da atualização de versão (*deploy*) e da configuração da infraestrutura para o bom funcionamento de todas as camadas da aplicação.

Tendo uma infraestrutura mais enxuta e um único código fonte para todos os *tenants*, o tempo dispendido para manutenção da infraestrutura e a resolução de *bugs* torna-se menor.

Ao mesmo tempo, quando se começa a desenvolver funcionalidades específicas para cada *tenant* em um único código fonte sem que essas funcionalidade afete *tenants* que a tem desabilitada, o código pode se tornar mais complexo.

A maior parte do desafio de garantir a facilidade na manutenção de aplicações *multi-tenant* está em garantir que todas as funcionalidades estão bem isoladas e que a aplicação funciona corretamente quando elas estão ativadas ou não para cada *tenant*.

Requisitos Específicos de Alguns *Tenants*

Quando fazemos uma aplicação *multi-tenant*, temos uma parte do software que suporta as funcionalidades comuns à maioria dos clientes. Porém, alguns clientes necessitam de campos e tabelas específicas para sua realidade. A habilidade de dar esse nível de customização aos clientes é um dos maiores desafios do projeto de aplicações *multi-tenant*, pois é difícil sincronizar bancos de dados que variam entre si com um mesmo código fonte.

2.1.2 Estratégias de implementação

Aqui apresentaremos as principais abordagens para implementação de aplicações *multi-tenant* e suas vantagens e desvantagens.

Multi-bancos

Nessa abordagem, cada *tenant* tem o seu próprio banco de dados, o que por si só já dificulta muito o vazamento de informações entre *tenants*. Isso permite a criação de tabelas do banco de dados especificamente para um *tenant*, assim como modificar as tabelas comuns a todos os *tenants* apenas para alguns, já que os bancos de dados são independentes uns dos outros. A arquitetura dessa abordagem está ilustrada na Figura 2.1.

Uma desvantagem dessa estratégia, além do aumento rápido no custo da infraestrutura, é a dificuldade de compartilhar dados entre *tenants*. Nesse caso, por exemplo, seria complicado uma aplicação permitir que um mesmo usuário, com apenas uma conta, tivesse acesso a vários *tenants*.

Multi-schemas

Nesta estratégia, cada *tenant* possui seu próprio *schema* de dados, uma funcionalidade que a maioria dos SGBDs (Sistemas Gerenciadores de Banco de Dados) atuais oferece. Um *schema*

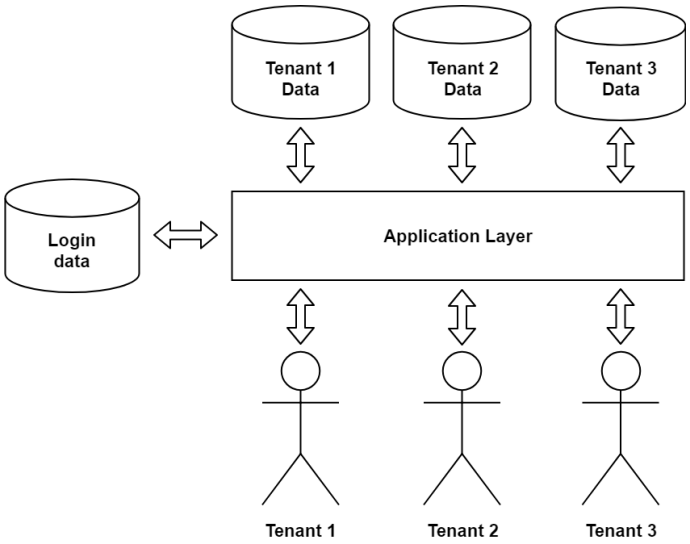


Figura 2.1 Arquitetura *multi-tenant* utilizando abordagem multi-bancos. Fonte: Próprio Autor

é uma espécie de *namespace* para agrupamento dos dados. Cada *schema* tem suas tabelas e seus dados, o que já separa automaticamente os dados por *tenant* da mesma maneira que a abordagem multi-bancos. Esta separação também permite a criação de tabelas e colunas específicas por *tenant*. A organização das tabelas no banco de dados pode ser observada na Figura 2.2.

Apesar de diminuir a velocidade do crescimento da infraestrutura por *tenant*, essa abordagem também sofre da mesma dificuldade de compartilhamento de dados que a abordagem multi-bancos.

Outra desvantagem desta estratégia é que a maioria dos SGBDs não foi feita para suportar uma quantidade muito grande de *schemas* por banco e, comumente, apresentam problemas quando isso ocorre. Por exemplo, no PostgreSQL [Gro96] existem alguns problemas conhecidos na interface de gerenciamento e no sistema de *backups* e recuperação de dados quando se tem uma quantidade de *schemas* na casa dos milhares [Ste13].

Múltiplas tabelas

Há também a possibilidade de criar um conjunto de tabelas para cada *tenant* de maneira que cada um tenha um conjunto de tabelas exclusivas para ele. Essa abordagem é semelhante a de múltiplos *schemas*, mas a divisão de tabelas é feita através do nome das tabelas. A organização das tabelas no banco de dados pode ser observada na Figura 2.3.

Da mesma forma que a abordagem *multi-schemas*, essa sofre do problema de desempenho quando o número de *tenants* cresce muito. O motivo também é o mesmo: a maioria dos *SGBDs* não foram feitos considerando uma quantidade muito grande de tabelas.

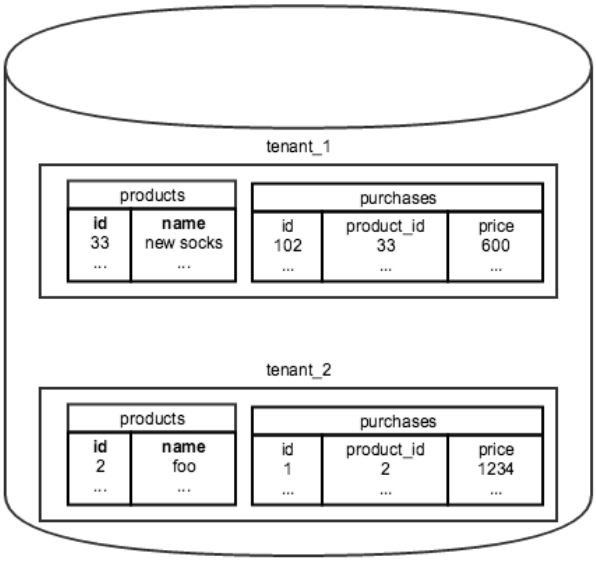


Figura 2.2 Arquitetura *multi-tenant* utilizando abordagem *Multi-schemas*. Fonte: Próprio Autor

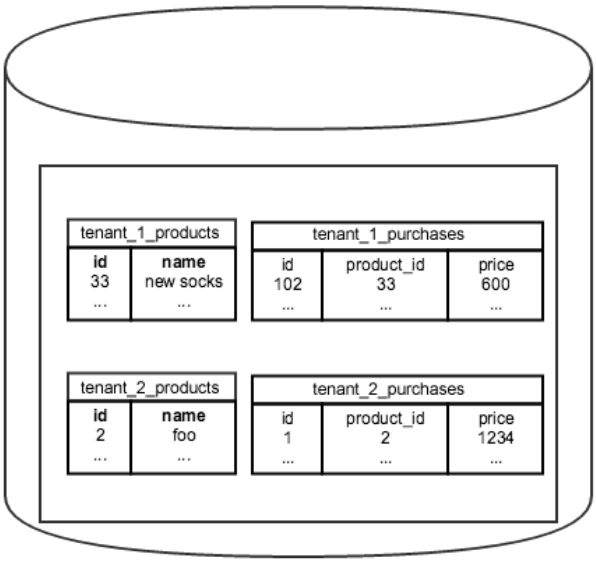


Figura 2.3 Diagrama representando arquitetura *multi-tenant* utilizando abordagem *Multi-tabelas*. Fonte: Próprio Autor

Tabelas compartilhadas ou *Schema* compartilhado

Nessa abordagem os *tenants* compartilham todas as tabelas do banco de dados e a separação desses dados por *tenant* é feita através de filtros em consultas. Essa abordagem requer muito cuidado, pois um erro em qualquer parte do código pode ocasionar vazamento de dados entre *tenants*. A organização das tabelas no banco de dados pode ser observada na Figura 2.4.

Apesar de dificultar pouco a segurança, a separação a nível de aplicação permite que parte dos dados seja compartilhada entre todos os *tenants*, e que, por exemplo, um usuário possa ter acesso a mais de um *tenant*, caso lhe seja dado permissão para tal. Além disso é muito mais fácil extrair relatórios que agreguem informações de mais de um *tenant*.

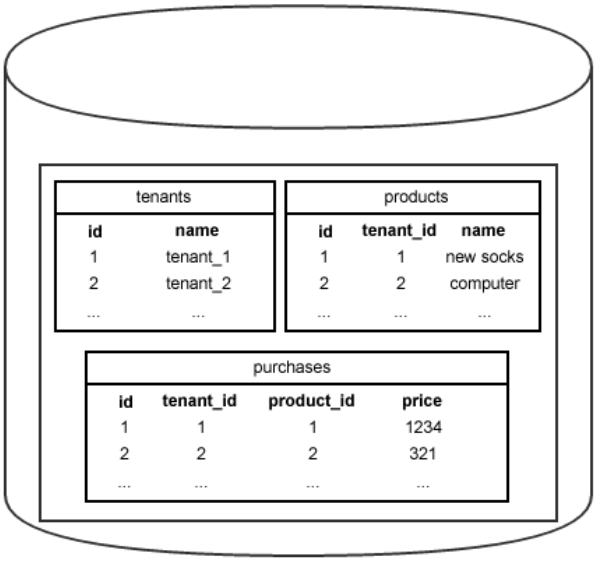


Figura 2.4 Arquitetura *multi-tenant* utilizando abordagem de Tabelas compartilhadas. Fonte: Próprio Autor

A criação de tabelas e colunas específicas para cada *tenant* é um problema que também é mais complexo nessa abordagem, pois todos os *tenants* compartilham as mesmas tabelas no banco de dados e, portanto, é difícil fazer alterações sem que todos os *tenants* sejam afetados. Algumas alternativas para tratar esse problema são:

Tabelas Privadas Essa é a forma mais básica de implementar extensibilidade: criar tabelas privadas para os *tenants* que precisam de extensão do seu esquema de tabelas. O problema dessa abordagem é que, apesar das tabelas serem utilizadas apenas por um *tenant*, elas estão disponíveis para todos os *tenants*.

Tabelas Estendidas Essa abordagem, assim como a de tabelas privadas, altera o esquema de dados do banco para todos os *tenants*. Trata-se da adição de colunas genéricas (*VARCHAR*) sobressalentes à tabelas já existentes no esquema. Essas colunas podem ser usadas de forma diferente por cada *tenant* e as que não forem utilizadas são preenchidas com *NULL*. Um exemplo pode ser observado na Figura 2.5.

Tabela Universal De acordo com [HDX12] e [GM14], esta é uma tabela genérica com uma coluna *Tenant*, uma coluna Tabela e uma quantidade grande de colunas genéricas. Essas colunas genéricas são do tipo *VARCHAR* que facilmente podem ser convertidas de e para outros tipos de dados.

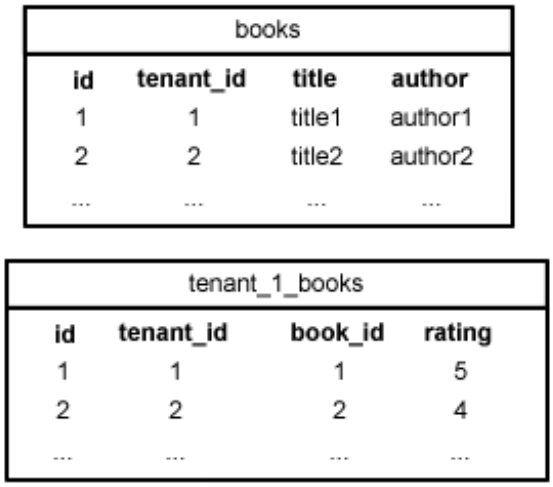


Figura 2.5 Exemplo de implementação no banco de dados usando a técnica de tabelas estendidas. Fonte: Próprio Autor

O principal problema dessa abordagem é a grande quantidade de colunas genéricas necessárias. A quantidade de colunas deve ser pelo menos igual a quantidade de colunas da maior tabela lógica. Se uma tabela lógica tem menos colunas que a tabela universal, todas as linhas que a referenciam terão as colunas sobressalentes preenchidas com *NULL*.

Um exemplo dessa abordagem pode ser observado na Figura 2.6.

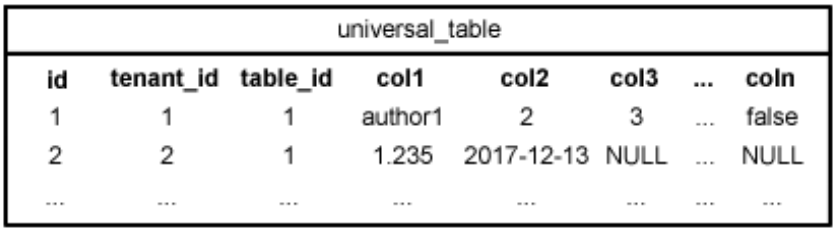
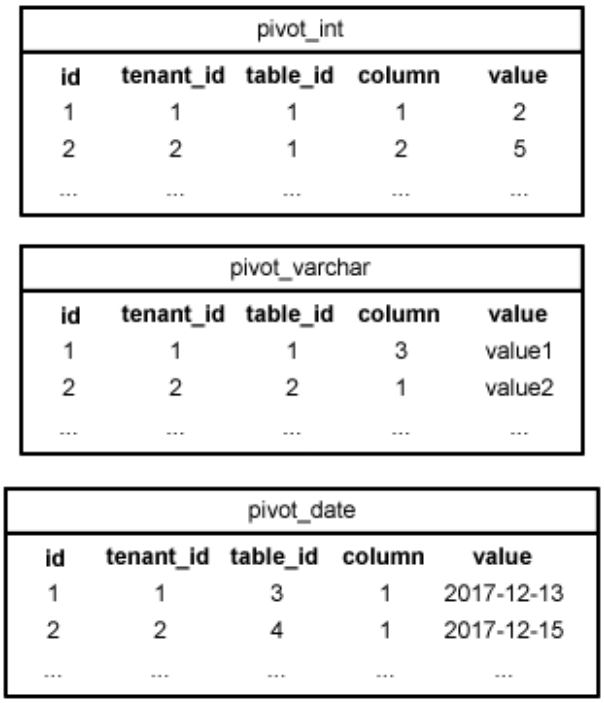


Figura 2.6 Exemplo de implementação no banco de dados usando a técnica da tabela universal. Fonte: Próprio Autor

Tabelas Pivot Numa tabela *pivot*, cada linha armazena um valor de uma coluna de uma tabela lógica. As tabelas *pivot* são associadas a um tipo e todas as linhas da mesma armazenam um dado desse tipo. Cada linha da tabela referencia o *tenant*, a tabela lógica, a linha da tabela lógica da qual faz parte e a coluna que representa na tabela lógica, além do dado que armazena. Essa tabela está ilustrada na Figura 2.7.

Um exemplo dessa abordagem pode ser observado na Figura 2.7.

Em relação às abordagens anteriores, essa armazena um número bem menor de *NULLs*, porém a tabela *pivot* também exige a utilização de agregações (*JOINs*) para a recuperação de linhas completas prejudicando um pouco o desempenho das consultas.



pivot_int

id	tenant_id	table_id	column	value
1	1	1	1	2
2	2	1	2	5
...	...	...	...	...

pivot_varchar

id	tenant_id	table_id	column	value
1	1	1	3	value1
2	2	2	1	value2
...	...	...	...	...

pivot_date

id	tenant_id	table_id	column	value
1	1	3	1	2017-12-13
2	2	4	1	2017-12-15
...	...	...	...	...

Figura 2.7 Exemplo de implementação no banco de dados usando a técnica de tabelas *pivots*. Fonte: Próprio Autor

Tabelas Chunk Uma tabela *chunk* se parece com uma tabela *pivot*, porém possui uma coluna genéricas de cada tipo de dado, permitindo armazenar valores de vários tipos numa só linha do chunk. Essa abordagem, em relação a tabelas *pivot*, diminui a quantidade de agregações necessárias para recuperar a informação, porém aumenta um pouco a quantidade de valores *NULL* armazenados.

Um exemplo dessa abordagem pode ser observado na Figura 2.8.

Existe também uma variação dessa abordagem chamada *Chunks Folding*, na qual os *chunks* estendem tabelas físicas presentes no esquema do banco de dados ao invés de tabelas lógicas.

chunk_data

id	tenant_id	col	row	int	date	char	bool
1	1	0	1	20	2017-12-12	test 1	false
2	2	3	1	53	NULL	NULL	true
...	...	...	...	...	...	...	...

Figura 2.8 Exemplo de implementação no banco de dados usando a técnica tabelas *chunk*. Fonte: Próprio Autor

2.2 *Django*

Django é um *framework* para *Python*, escrito em *Python*, para o desenvolvimento de aplicações Web [Geo17]. Ele tem como base em sua arquitetura o padrão MTV (*Models, Templates and Views*) [Geo17], que é uma variação do, comumente usado, padrão MVC (*Model, View and Controllers*)[EGH00].

Uma das maiores preocupações do *Django* é estar sempre acompanhando as necessidades da maioria dos desenvolvedores da maneira mais prática, porém dando sempre a possibilidade de se estender comportamentos e fazer funcionalidades complexas quando necessário.

2.2.1 Arquitetura

No padrão MVC, os *models* são responsáveis por gerenciar os dados da aplicação, as *Views* são responsáveis por decidir que informações apresentar para o usuário e como coletar informações do usuário, e os *Controllers* são responsáveis pelas regras de negócio e intermediam o relacionamento entre as *Views* e os *Models*.

No padrão MTV, apesar dos *Models* terem as mesmas funções que no MVC, no *Django* eles são gerenciados utilizando-se um ORM (*Object Relational Mapping*)[Fus97] que facilita as operações que envolvem manipulação de dados numa interface mais amigável que a linguagem do SGBD. As *Views*, neste caso, carregam mais responsabilidade, lidando com todo o processamento das requisições recebidas pela rede. Os *templates* são responsáveis por definir a lógica de exibição das informações para os usuários.

Um projeto em *Django* também incentiva a criação da sua aplicação utilizando *apps*, que são blocos de funcionalidades que funcionam independentemente de outros. Isso torna muito fácil estender as funcionalidades do próprio *framework*, que hoje já possui com uma grande quantidade de *apps* de código aberto para adicionar as mais diversas funcionalidades aos projetos feitos com *Django*[Fou05].

O próprio *Django* também conta com diversas ferramentas para resolver problemas comuns à maioria dos projetos como Autenticação, Papéis e Permissões (RBAC), Segurança, Migrações de dados, Formulários e uma interface gráfica administrativa genérica baseada nos *Models* dos

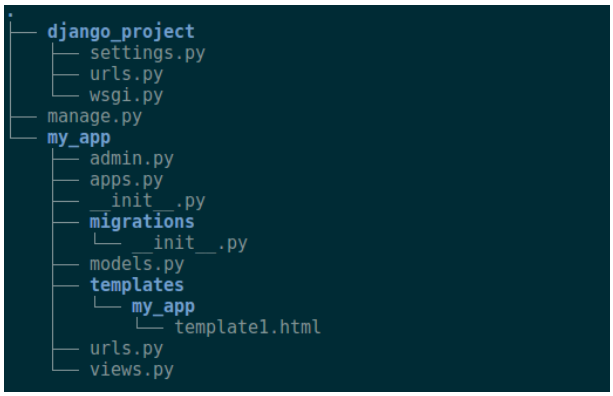


Figura 2.9 Exemplo de estrutura de arquivos num projeto *Django*. Fonte: Próprio Autor

apps para gerenciar os dados da aplicação.

2.2.2 *Django ORM*

O *Django ORM* é uma API para manipulação de dados integrada com diversos SGBDs. Além de unificar as diversas tecnologias de banco de dados, o ORM (*Object Relational Mapping*) fornece uma API amigável para fazer operações das mais simples às mais complexas. Essa API se baseia em três estruturas principais para garantir seu funcionamento: *Models*, *Managers* e *Querysets*.

Models

Models são classes definidas pelo desenvolvedor responsáveis por definir tabelas no banco de dados. Nessas classes podem ser definidas as colunas das tabelas e algumas outras informações como a ordem padrão das consultas, se devem ser criados índices para essas tabelas e quais as colunas indexadas.

Nos *models* também é definido como os dados devem ser salvos através do método `save`. Nesse método definimos, por exemplo, o preenchimento automático de dados que não necessitam de entradas externas e chamadas que devem ser feitas antes ou depois de uma nova coluna ser criada ou atualizada.

A seguir é mostrado um exemplo de *model* no qual é definida uma tabela `article` com as colunas `title`, `text`, `tags`, `author` e `is_admin_article`. Note que também é definido o método `save`, que preenche o campo `is_admin_article` dependendo do valor do atributo `author`.

```
1 class Article(models.Model):
2     title = models.CharField(max_length=100)
3     text = models.TextField()
4     tags = models.ManyToManyField('Tag')
5     author = models.ForeignKey(settings.AUTH_USER_MODEL)
```

```
6     is_admin_article = models.BooleanField()
7
8     def __str__(self):
9         return '%s - %s' % (self.title, str(self.author))
10
11    def save(self, *args, **kwargs):
12        self.is_admin_article = False
13        if self.author.is_staff:
14            self.is_admin_article = True
15
16        return super().save(*args, **kwargs)
```

Querysets

Querysets são classes responsáveis por definir como operações de manipulação de dados serão executadas. Aqui são definidos métodos como o `filter` para obter as linhas do banco que tenham os parâmetros nomeados passados (cujos nomes devem ser os nomes das colunas da tabela em questão). São definidos também os métodos `exclude`, que seleciona apenas as linhas que não possuam os parâmetros passados, e `update`, que atualiza colunas de uma porção de dados selecionada previamente (utilizando os métodos de consulta do próprio *Queryset*) com os parâmetros passados.

Existem diversos outros métodos que abstraem operações no banco de dados permitindo ao usuário fazer consultas diretas na linguagem do SGBD, ou fazer agregações, sub-consultas, escolher as regras de ordenação dos dados a serem recuperados e muitas outras operações para construir consultas mais complexas. Esses métodos são amplamente discutidos em [Fou05],

Outra característica dos *Querysets* é que muitos dos seus métodos podem ser aninhados, permitindo assim a elaboração legível de operações complexas através da combinação de operações mais simples. No final, essa combinação de operações será traduzida numa única operação convertida para a linguagem do bando de dados.

O *Django*, por padrão, já oferece uma implementação de *Queryset* bastante completa, com muitas ferramentas para fazer das consultas mais simples às mais complexas. Porém, se alguma operação não estiver implementada, recomenda-se fazê-la criando uma classe filha do *Queryset* padrão do *Django* e adicionando a nova operação. No exemplo a seguir, criamos um queryset específico para *articles* que adiciona uma operação para atualizar a coluna `is_admin_article` com o valor `False`.

```
1 class ArticleQueryset(Queryset):
2
3     def update_to_not_admin(self):
4         return super().update(is_admin_article=False)
```

Managers

São classes que ligam os *Models* aos *Querysets*. Nos *Managers*, definem-se que *queryset* usar, e também é o local ideal para definir métodos para realizar operações comuns dos *Models*,

como formas específicas de criar e atualizar seus dados, ou consultas nesse *model* que são executadas com frequência pela aplicação.

Todo *Model*, por padrão, possui ao menos um *Manager*. O *Django* já define o seu *Manager* padrão em qualquer *Model*. Além disto, o *framework* permite que o usuário defina outros *Managers* para aquele *Model* ou sobrescreva o *Manager* padrão, fazendo com que toda operação naquele *model* utilize o *Manager* customizado.

No exemplo a seguir, é implementado um `Manager` específico para o *model* `Article`. O `ArticleManager` define um método `admin_articles`, que retorna um *queryset* filtrado, contendo apenas `articles` cuja coluna `is_admin_article` contém o valor `True`.

```
class ArticleManager(Manager):

    def get_queryset():
        return ArticleQueryset(self.model, using=self._db)

    def admin_articles(self):
        return self.get_queryset().filter(is_admin_article=True)


class Article(models.Model):
    # ... fields
    objects = ArticleManager()
    # ...methods
```

Templates

Como qualquer *framework* Web, *Django* possui formas de gerar código HTML (*Hyper Text Markup Language*) dinamicamente. A forma mais comum são os *templates*. *Templates* possuem partes estáticas do HTML desejado, assim como as partes dinâmicas, geradas através de uma linguagem feita exclusivamente para isso, e de informações passadas através do *Context* e dos *ContextProcessors*.

No exemplo a seguir, tem-se um *template* que renderiza uma lista com os títulos dos `articles` do usuário que fez a requisição HTTP.

```
<h1> Seus artigos </h1>
<ul>
    {% for article in request.user.articles.all %}
        <li> {{article.title}} </li>
    {% endfor %}
</ul>
```

Context

Context um objeto que contém a informação que deseja-se disponibilizar aos *templates* no momento da renderização. É através do *context* que passamos, por exemplo, os dados extraídos

de consultas no banco para serem renderizados num *template*. No exemplo a seguir utilizamos a função `render` para renderizar um *template* de nome `'template_name.html'`. O terceiro parâmetro é *context* a ser utilizado pelo *template*.

No exemplo a seguir utilizamos a função `render()` do *Django* para renderizar o template de nome `'template_name.html'` passando o valor `'Nome da página'` nomeado como `page_name`. Nesse caso, como no *template* `template_name.html` há uma *tag* `{{page_name}}` ela será substituída por `Nome da página` na renderização.

```
render(request, 'template_name.html', {'page_name': 'Nome da página'})
```

```
<!-- template_name.html -->
<h1> {{page_name}} </h1>
<ul>
    {% for article in request.user.articles.all %}
        <li> {{article.title}} </li>
    {% endfor %}
</ul>
```

ContextProcessors

Django também oferece formas de adicionar informações globais ao contexto dos *templates*: os *ContextProcessors*. Eles são funções *Python* que recebem uma requisição HTTP e retornam um dicionário *python*. Esse dicionário é adicionado ao contexto de qualquer *template* renderizado pelo *Django*.

No exemplo a seguir, mostramos como definir um *context processor*. Além desse definição, o caminho para a função definida deve ser adiciono na configuração de templates do *Django*.

```
def articles_context_processor(request):
    return {
        'user_articles_count': request.user.articles.count()
    }
```

2.2.3 *Views*

Uma *view* é simplesmente uma função que recebe uma requisição como parâmetro e deve retornar uma resposta HTTP, podendo esta conter um *template* HTML, dados em XML ou JSON, ou até mesmo nada.

Um exemplo de uma *view* pode ser visto a seguir. No exemplo, a *view* retorna o conteúdo do *template* `template_name.html` renderizado.

```
def my_view(request):
    return render(request, 'template_name.html')
```

Class-Based Views

Existe também uma forma de definir uma *View* como uma classe, de maneira que possamos usar recursos como herança e tornar o processo de construção mais declarativo. O *Django* já dispõe de várias classes de *Views* genéricas que, quando usadas como superclasse, facilitam o desenvolvimento de novas *Views* com alguns comportamentos predefinidos.

Exemplos disso é a *TemplateView*, que só precisa da definição do atributo `template_name` para retornar um *template* para o usuário. Um exemplo de uma *class-based view* pode ser visto a seguir. Ela tem a mesma funcionalidade que a *view* do exemplo anterior (utilizando função).

```
class MyClassBasedView(TemplateView):
    template_name = 'template_name.html'
```

2.2.4 *URLs*

O sistema de *urls* do *Django* faz uso de expressões regulares para definir os padrões aceitos pela aplicação, e cada expressão regular é ligada à uma *View*. Sempre que a aplicação recebe uma requisição para uma URL, os padrões são percorridos um a um, até que alguma expressão regular aceite, encaminhando a requisição para a *View* relacionada à essa expressão.

Um exemplo de mapeamento de URLs em *views* pode ser visto a seguir.

```
urlpatterns = [
    url(
        regex=r'^my-view/$',
        view=my_view,
        name='my-view'
    ),
    url(
        regex=r'^my-class-based-view/$',
        view=MyClassBasedView.as_view(),
        name='my-class-based-view'
    ),
]
```

2.2.5 *Middleware*

Django possui um *middleware* que implementa a arquitetura *Chain of Responsability* [EGH00]. Ele é usado para processar as requisições HTTP que chegam para a aplicação.

Chain of Responsability é um padrão que, de acordo com [EGH00], é usado para "evitar o acoplamento do remetente de uma solicitação ao seu receptor, ao dar a mais de um objeto a oportunidade de tratar a solicitação e encadear os objetos receptores, passando a solicitação ao longo da cadeia até que um objeto a trate.".

No *Django*, cada *Middleware* (item da cadeia) é responsável por processar uma requisição que lhe é enviada e repassar o objeto processado para o próximo *Middleware*. O mesmo ocorre com a resposta da aplicação, que é processada em cada um dos *Middleware* até ser enviada para o cliente HTTP. O objetivo disso é dar a possibilidade de, para um propósito específico, facilmente adicionar informações à requisição antes dela atingir a respectiva *View* e/ou à resposta dessa *View* logo após ser retornada.

A seguir, damos um exemplo de *middleware* retirado de [Fou05].

```
1  class SimpleMiddleware(object):
2      def __init__(self, get_response):
3          self.get_response = get_response
4          # One-time configuration and initialization.
5
6      def __call__(self, request):
7          # Code to be executed for each request before
8          # the view (and later middleware) are called.
9
10         response = self.get_response(request)
11
12         # Code to be executed for each request/response after
13         # the view is called.
14
15         return response
```

2.2.6 Pacotes da Biblioteca Padrão

O *Django* possui diversos pacotes que não estão diretamente ligados com o seu núcleo, isto é, não são essenciais para o funcionamento do *framework*, porém implementam funcionalidades comuns em muitas aplicações. A seguir, apresentamos três deles que foram importantes para o desenvolvimento da *Django-SSTenants*: Autenticação, Sites e *ContentTypes*.

Autenticação

O *Django* possui, em sua biblioteca padrão, um sistema de autenticação de usuário completo, que inclui desde funcionalidades básicas como *login* e *logout*, como funcionalidades mais complexas como permissões de acesso e papéis de usuário implementando o padrão RBAC [FK92].

Apesar de já possuir uma implementação bastante agnóstica e extensível de autenticação, em muitas aplicações é necessário implementar fluxos específicos, que fogem do padrão sugerido pelo *framework*. O *Django* dá suporte a esse tipo de extensão através de *backends* de autenticação. Esses *backends* são classes que implementam as regras de autenticação, sendo a autenticação padrão integrada ao *Django* um desses *backends*. Isso dá a possibilidade de o desenvolvedor implementar sua própria lógica de forma integrada a API do *framework*.

Sites

O *Django* possui um *framework* para gestão de múltiplos domínios na mesma aplicação, permitindo que o usuário, a partir do domínio (*host*) da requisição HTTP, forneça diferentes tratamentos para requisição e diferentes respostas.

ContentTypes

Outro pacote da biblioteca padrão é o de *ContentTypes*, que adiciona um registro de todos os *models* da aplicação numa tabela do banco de dados, permitindo assim que sejam referenciados através chaves estrangeiras (colunas de tabelas do banco de dados que armazenam o id de uma linha em outra tabela), dessa forma, abrindo mais possibilidades de generalização de *models*.

2.2.7 Pacotes de Terceiros

Django é um *framework* maduro e com os vários anos de desenvolvimento acabou se tornando muito estável. Desde sua criação, em 2005, ele também tem se tornado muito popular [dja18] e, com isso, passa a ter uma grande quantidade de pacotes que estendem suas funcionalidades, sendo muitos deles de código aberto.

Alguns pacotes são especialmente populares, como o *Django REST Framework*, que facilita o desenvolvimento de APIs REST (*Representational State Transfer*). A utilização em massa de software distribuídos e aplicativos móveis tonou muito comum que novos pacotes sejam desenvolvidos com compatibilidade com o *Django REST Framework* tornou-se bastante comum na comunidade.

2.3 Considerações finais

Este capítulo apresentou uma visão geral sobre a arquitetura *multi-tenant* passando por suas vantagens, desvantagens, principais desafios e algumas abordagens de implementação, assim como introduziu o *framework Django* de maneira suficientemente abrangente para que fosse possível o entendimento do *Django-SSTenants*.

CAPÍTULO 3

Django-SSTenants

Neste capítulo serão apresentados os desafios encontrados no desenvolvimento da *Django-SSTenants*, bem como a forma como os solucionamos de maneira a deixar a sua utilização simples e com desempenho adequado.

3.1 Visão Geral

A *Django-SSTenants* visa facilitar a implantação da arquitetura *multi-tenant* utilizando a abordagem de tabelas compartilhadas. Essa escolha foi feita por conta da simplicidade da infraestrutura necessária e da facilidade de aumentar o número de *tenants* sem se preocupar com o crescimento da infraestrutura. Além disso, não havia alternativas para *Django* que auxiliassem na implementação dessa abordagem.

Os desafios que a *Django-SSTenants* aborda são principalmente a separação dos dados, a definição de papéis e permissões por *tenant* e a capacidade de criar campos e tabelas específicas para cada *tenant* utilizando tabelas *pivot*.

Nesse capítulo será descrita em detalhes a forma como a Django-SSTenants foi implementada de maneira a atingir esses requisitos, junto os ganhos e perdas de cada escolha técnica.

3.2 Armazenamento de informação dos *tenants*

Em aplicações *multi-tenant* é comum que cada *tenant* tenha uma tabela no banco de dados para salvar informações de configuração do mesmo e dados extra. De maneira similar, a *Django-SSTenants* modela essa tabela e dá a possibilidade do usuário da biblioteca definir que informações quer adicionar ao *tenant* através das colunas `settings` e `extra_data`. Estas colunas recebem conteúdo no formato JSON (*JavaScript Object Notation*), além de adicionar algumas ferramentas para adicionar regras de validação desses JSON. A coluna `settings` salva configurações do *tenant* em relação às funcionalidades da aplicação. Já a coluna `extra_data` serve para armazenar dados extra do *tenant* como logotipo, descrição, e-mail padrão, estilos customizados etc.

Como nem todos os SGBDs suportam colunas JSON nativas, os campos `settings` e `extra_data` são definidos como `TEXT`(texto sem limite de tamanho) caso não haja suporte. Para tornar mais simples a manipulação desses campos, estão disponíveis algumas *properties* do *Python* para converter de *string* para dicionário (objeto que armazena dados numa estrutura chave/valor), quando os dados estiverem sendo recuperados do banco, e de dicionário para *string* quando ocorrer uma operação de escrita no banco. As *properties* são métodos (ou um

par de métodos, *set* e *get*) que se comportam como um atributo. Quando um valor é passado pra uma *property*, o método *set* da mesma é chamado. Quando o valor de uma *property* é solicitado pela aplicação, o método *get* é chamado e o valor retornado por ele é utilizado.

```
class Tenant(TimeStampedModel):
    name = models.CharField(max_length=255)
    slug = models.CharField(max_length=255, primary_key=True)

    if 'postgresql' in django_settings.DATABASES['default']['ENGINE']:
        from django.contrib.postgres.fields import JSONField
        extra_data = JSONField(
            blank=True, null=True,
            default=get_setting('DEFAULT_TENANT_EXTRA_DATA'))
        settings = JSONField(
            blank=True, null=True,
            default=get_setting('DEFAULT_TENANT_SETTINGS'))
    else:
        _extra_data = models.TextField(
            blank=True, null=True, validators=[validate_json],
            default=json.dumps(get_setting('DEFAULT_TENANT_EXTRA_DATA')))
        _settings = models.TextField(
            blank=True, null=True, validators=[validate_json],
            default=json.dumps(get_setting('DEFAULT_TENANT_SETTINGS')))

        @property
        def extra_data(self):
            import json
            return json.loads(self._extra_data)

        @extra_data.setter
        def extra_data(self, value):
            import json
            self._extra_data = json.dumps(value)

        @property
        def settings(self):
            import json
            return json.loads(self._settings)

        @settings.setter
        def settings(self, value):
            import json
            self._settings = json.dumps(value)

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

Essa escolha torna flexível a manipulação dos dados e torna uma operação pouco onerosa a recuperação das informações dos *tenants*, ao custo das buscas utilizando as informações

contidas dos campos `settings` e `extra_data` como parâmetros mais lentas do que se as informações fossem campos comuns do banco de dados (inteiros, *strings*, datas, etc).

Domínios (host names)

Uma funcionalidade muito comum em sistemas *multi-tenant* é que cada *tenant* tenha seu próprio domínio (ou um subdomínio do domínio principal da aplicação). Muitos software oferecem ainda a possibilidade de o *tenant* adicionar um domínio privado.

Pensando nessas duas possibilidades, a *Django-SSTenants* cria uma integração ao *framework* *django-sites* da biblioteca padrão do *Django*, para garantir que um *tenant* possa ter vários domínios associados, permitindo assim que consigamos recuperar o *tenant* da requisição a partir do domínio.

```
from django.contrib.sites.models import Site

class TenantSite(TimeStampedModel):
    tenant = models.ForeignKey('Tenant', related_name="tenant_sites")
    site = models.OneToOneField(Site, related_name="tenant_site")

    objects = SingleTenantModelManager

    def __str__(self):
        return '%s - %s' % (self.tenant.name, self.site.domain)


def post_delete_tenant_site(sender, instance, *args, **kwargs):
    if instance.site:
        instance.site.delete()
post_delete.connect(post_delete_tenant_site, sender=TenantSite)
```

3.3 Relacionamentos entre usuários e *tenants*

Para simbolizar relacionamentos entre usuários e *tenants*, a *Django-SSTenants* cria uma tabela de relacionamentos, que registra, não só o usuário e o *tenant* em questão, mas também que papéis e permissões o usuário possui naquele *tenant*. Esses relacionamentos utilizam os *models* do próprio sistemas de papéis e permissões do módulo de autenticação padrão do *Django* de forma a não duplicar informações e ter uma melhor integração com o *framework*.

```
class TenantRelationship(TimeStampedModel):
    tenant = models.ForeignKey('Tenant', related_name="relationships")
    user = models.ForeignKey(
        django_settings.AUTH_USER_MODEL, related_name="relationships")
    groups = models.ManyToManyField(
        'auth.Group', related_name="user_tenant_groups", blank=True)
    permissions = models.ManyToManyField(
```

```
        'auth.Permission', related_name="user_tenant_permissions",
        blank=True)

    def __str__(self):
        groups_str = ', '.join([g.name for g in self.groups.all()])
        return '%s - %s (%s)' % (
            str(self.user), str(self.tenant), groups_str)

    class Meta:
        unique_together = [('user', 'tenant')]
```

3.4 Recuperação do *tenant* a partir da requisição

Quando estamos implementando a arquitetura *multi-tenant*, a primeira informação que precisamos ter antes de consultar qualquer informação no Banco de Dados é qual o *tenant* da requisição.

A *Django-SSTenants* utiliza o sistema de *Middlewares* do *Django* para extrair a informação da requisição e processar qual o *tenant* desejado, colocando essa informação no próprio objeto da requisição que será repassado para as *Views* a aplicação, e também definindo métodos para obtenção direta do *tenant* atual de qualquer lugar da aplicação.

Para garantir também que o software não vai utilizar recursos desnecessários, a própria consulta no banco de dados que recupera o *tenant* atual só é executada caso o *tenant* seja utilizado pela aplicação, utilizando o ferramentas de recuperação *lazy* (não imediata) que o próprio *Django* provê. Dessa maneira, servir uma página estática que não depende do *tenant*, não vai fazer consultas desnecessárias.

O *Middleware* também salva o *tenant* extraído da requisição ou forçado pelo usuário (através do método `set_tenant` do próprio *middleware*) num dicionário. Como o *Django* trabalha com apenas uma *thread* por requisição, é seguro utilizar o identificador da *thread* como chave do dicionário de *tenants* atuais, garantido que o *tenant* atual estará disponível em qualquer lugar do código que for requisitado.

```
class TenantMiddleware(object):
    _threadmap = {}

    def __init__(self, get_response):
        self.get_response = get_response
        # One-time configuration and initialization.

    @classmethod
    def get_current_tenant(cls):
        try:
            return cls._threadmap[threading.get_ident()]
        except KeyError:
            return None

    @classmethod
```

```
16      def set_tenant(cls, tenant_slug):
17          cls._threadmap[threading.get_ident()] = SimpleLazyObject(
18              lambda: Tenant.objects.filter(slug=tenant_slug).first())
19
20      @classmethod
21      def clear_tenant(cls):
22          del cls._threadmap[threading.get_ident()]
23
24      def process_request(self, request):
25          request.tenant = SimpleLazyObject(lambda: get_tenant(request))
26          self._threadmap[threading.get_ident()] = request.tenant
27
28          return request
29
30      def process_exception(self, request, exception):
31          try:
32              del self._threadmap[threading.get_ident()]
33          except KeyError:
34              pass
35
36      def process_response(self, request, response):
37          try:
38              del self._threadmap[threading.get_ident()]
39          except KeyError:
40              pass
41          return response
42
43      def __call__(self, request):
44          # Code to be executed for each request before
45          # the view (and later middleware) are called.
46          request = self.process_request(request)
47          response = self.get_response(request)
48          return self.process_response(request, response)
```

Existem várias alternativas para extrair o *tenant* da requisição: ele pode ser inferido a partir do domínio (*host*) da aplicação em casos de *multi-tenancy* com domínios exclusivos para cada *tenant*, pode ser extraído de cabeçalhos HTTP, da sessão do navegador, ou de inúmeras outras maneiras. Essas alternativas dependem bastante da forma como se deseja implementar *multi-tenancy*.

Levando em consideração que a decisão de como extrair o *tenant* da requisição depende muito da aplicação, a *Django-SSTenants* implementa o conceito de *tenant retrievers*, que são funções definidas na configuração do projeto e que recebem um objeto de requisição HTTP e retornam uma instância do *model Tenant*. Caso eles não consigam extrair o *tenant*, eles podem também não retornar, ou ainda levantar uma exceção impedindo o processo de busca do *tenant* de continuar (no caso de um identificador do *tenant* enviado ser inexistente, por exemplo).

O *Django-SSTenants* também permite adicionar, automaticamente, o identificador do *tenant* à sessão ativa, para utilizar essa informação em requisições futuras, sem a necessidade do *tenant* ser explicitamente adicionado à requisição de alguma das maneiras citadas anteriormente.

```
def get_tenant(request):
    if not hasattr(request, '_cached_tenant'):
        tenant_retrievers = get_setting('TENANT_RETRIEVERS')

        for tenant_retriever in tenant_retrievers:
            tenant = import_item(tenant_retriever)(request)
            if tenant:
                request._cached_tenant = tenant
                break

        if not getattr(request, '_cached_tenant', False):
            lazy_tenant = TenantMiddleware.get_current_tenant()
            if not lazy_tenant:
                return None

            lazy_tenant._setup()
            request._cached_tenant = lazy_tenant._wrapped

        elif get_setting('ADD_TENANT_TO_SESSION'):
            try:
                request.session['tenant_slug'] = (
                    request._cached_tenant.slug
                )
            except AttributeError:
                pass

    return request._cached_tenant
```

Por padrão a biblioteca dispõe de três *tenant retrievers*: um para extrair o *tenant* do domínio da requisição, utilizando a integração com *framework django-sites*, um para extrair o *tenant* de um cabeçalho HTTP customizável, e um para extrair da sessão ativa.

```
def retrieve_by_domain(request):
    try:
        return get_current_site(request).tenant_site.tenant
    except (TenantSite.DoesNotExist, Site.DoesNotExist):
        return None
    except Tenant.DoesNotExist:
        raise TenantNotFoundError()


def retrieve_by_http_header(request):
    try:
        tenant_http_header = 'HTTP_' + get_setting(
            'TENANT_HTTP_HEADER').replace('-', '_').upper()
        return Tenant.objects.get(slug=request.META[tenant_http_header])
    except LookupError:
        return None
    except Tenant.DoesNotExist:
```

```
18          raise TenantNotFoundError()
19
20
21  def retrieve_by_session(request):
22      try:
23          return Tenant.objects.get(slug=request.session['tenant_slug'])
24      except (AttributeError, LookupError, Tenant.DoesNotExist):
25          return None
26      except Tenant.DoesNotExist:
27          raise TenantNotFoundError()
```

3.5 Autenticação

Como mencionado na Subseção 2.2.6, o *Django* possui um sistema de autenticação baseado em *backends*. Para implementar um sistema de autenticação baseado em permissões específicas para os *tenants*, a *Django-SSTenants* dispõe de um *backend* que considera os *TenantRelationships* para verificar o nível de acesso que um usuário tem em um *tenant*.

Isso torna o processo de verificação de permissões considerando *tenants* totalmente integrado a biblioteca do Django, sendo possível utilizar funções auxiliares, *decorators*, classes auxiliares, assim como se faz numa aplicação *Django* comum.

Para garantir uma melhor performance, são utilizadas técnicas de *caching* similares às usadas pelo próprio *backend* de autenticação padrão do *Django* para que não seja necessário consultar o banco de dados sempre que for necessário verificar permissões numa mesma requisição.

O código relativo a esse módulo é muito extenso e utiliza muitos conceitos internos do *Django*, por isso não será mostrado nessa secção. Detalhes adicionais podem ser encontrados no arquivo `shared_schema_tenants/auth_backends.py` no código fonte da biblioteca. O link para o mesmo pode ser encontrado em Seção 5.1.

O parte do esquema relacional relativa a autenticação e permissões pode ser observada na Figura **??**.

3.6 Como garantir o isolamento dos *tenants*

Para garantir o isolamento dos *tenants* de forma que essa preocupação não seja vazada para o usuário da *Django-SSTenants*, foi feita uma integração direta com a forma que o *Django* lida com dados através da implementação de *Managers*. Os *Managers* implementados já filtram em qualquer consulta apenas os dados do *tenant* atual, impedindo que sejam recuperados dados de outro *tenant* direto na fonte.

Além disso disso, a biblioteca dispõe de *mixins* que facilitam a definição de *models*: o `SingleTenantModel` e o `MultipleTenantsModel`.

Ao definir um *model* herdando do `SingleTenantModel`, garantimos que esse *model* tem uma chave estrangeira para um *tenant* e que seu *manager* padrão é o `SingleTenantModelManager`, que utiliza o *tenant* atual para filtrar apenas resultados de consultas que tenham em sua chave estrangeira *tenant* uma referência para o *tenant* atual,

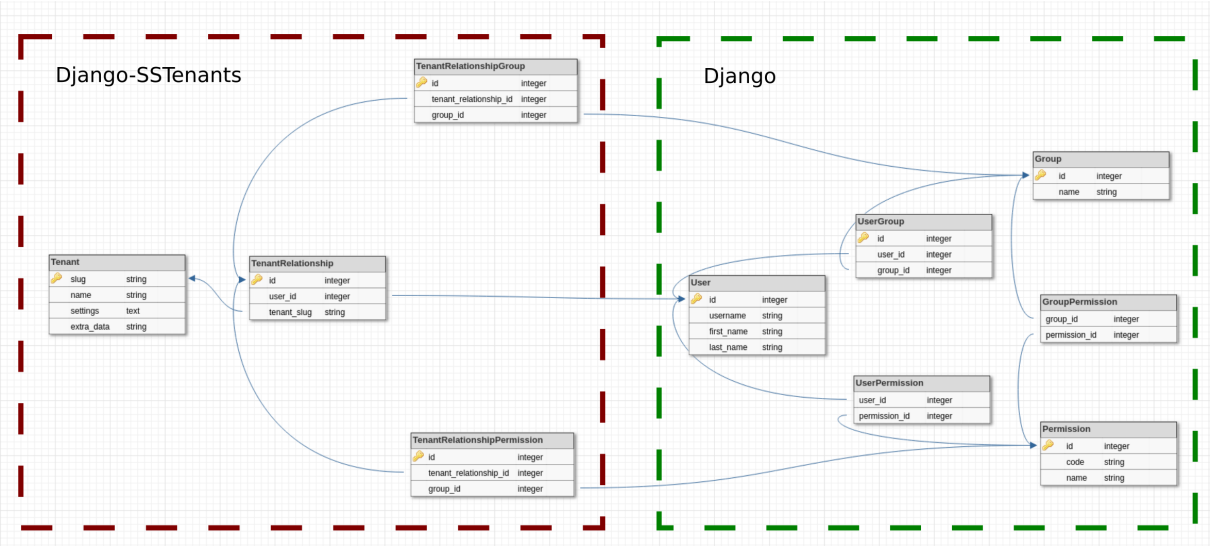


Figura 3.1 Esquema relacional das funcionalidades de autenticação e permissões da *Django-SSTenants*. Fonte: Próprio Autor

recuperado através do *TenantMiddleware*.

```
class SingleTenantModelManager(Manager):

    def get_original_queryset(self, *args, **kwargs):
        return super(SingleTenantModelManager, self).get_queryset(
            *args, **kwargs)

    def get_queryset(self, tenant=None, *args, **kwargs):
        if not tenant:
            tenant = get_current_tenant()
            if tenant:
                return super(SingleTenantModelManager, self).get_queryset(
                    *args, **kwargs).filter(tenant=tenant)
            else:
                return super(SingleTenantModelManager, self).get_queryset(
                    *args, **kwargs).none()
        else:
            return super(SingleTenantModelManager, self).get_queryset(
                *args, **kwargs).filter(tenant=tenant)


def get_default_tenant():
    from shared_schema_tenants.models import Tenant
    return Tenant.objects.filter(
        slug=get_setting('DEFAULT_TENANT_SLUG')).first()


```

```
27 class SingleTenantModelMixin(models.Model):
28     tenant = models.ForeignKey(
29         'shared_schema_tenants.Tenant', default=get_default_tenant)
30 
31     objects = SingleTenantModelManager()
32 
33     original_manager = models.Manager()
34     tenant_objects = SingleTenantModelManager()
35 
36     class Meta:
37         abstract = True
38         default_manager_name = 'original_manager'
39         base_manager_name = 'original_manager'
40 
41     def save(self, *args, **kwargs):
42         if not hasattr(self, 'tenant'):
43             self.tenant = get_current_tenant()
44 
45         if getattr(self, 'tenant', False):
46             return super(SingleTenantModelMixin, self).save(*args,
                ↪  **kwargs)
47         else:
48             raise TenantNotFoundError()
```

Similarmente, ao definir um *model* herdando do *MultipleTenantsModel*, garantimos que o *model* possui um relacionamento de muitos para muitos (*many to many*) com ao menos um *tenant*. Esse *model* também tem seu *manager* padrão definido como o `MultipleTenants-ModelManager`, que utiliza o *tenant* atual para filtrar apenas os resultados de consulta que tenha em um de seus *tenants* relacionados, o *tenant* atual.

```
1  class MultipleTenantModelManager(Manager):
2  
3      def get_original_queryset(self, *args, **kwargs):
4          return super(MultipleTenantModelManager, self).get_queryset(
5              *args, **kwargs)
6  
7      def get_queryset(self, tenant=None, *args, **kwargs):
8          if not tenant:
9              tenant = get_current_tenant()
10             if tenant:
11                 return super(
12                     MultipleTenantModelManager, self).get_queryset(
13                         *args, **kwargs).filter(tenants=tenant)
14             else:
15                 return super(
16                     MultipleTenantModelManager, self).get_queryset(
17                         *args, **kwargs).none()
18         else:
19             return super(
20                 MultipleTenantModelManager, self).get_queryset(
```

```
21                      *args, **kwargs).filter(tenants=tenant)
22
23
24 class MultipleTenantsModelMixin(models.Model):
25     tenants = models.ManyToManyField('shared_schema_tenants.Tenant')
26
27     objects = MultipleTenantModelManager()
28
29     tenant_objects = MultipleTenantModelManager()
30     original_manager = models.Manager()
31
32     class Meta:
33         abstract = True
34         default_manager_name = 'original_manager'
35         base_manager_name = 'original_manager'
36
37     def save(self, *args, **kwargs):
38         tenant = get_current_tenant()
39
40         if tenant:
41             instance = super(
42                 MultipleTenantsModelMixin, self).save(*args, **kwargs)
43             self.tenants.add(tenant)
44             return instance
45         else:
46             raise TenantNotFoundError()
```

Como a *Django-SSTenants* visa estender, e não limitar as funcionalidades do *Django*, ambos os *mixins* definidos na biblioteca possuem também um *manager* auxiliar `original_manager` que disponibiliza as funcionalidades do *manager* padrão do *Django*, isto é, consultas sem os filtros por *tenant*. Esse *manager* pode ser necessário em execução de *scripts* administrativos, migrações de dados e também extração de métricas.

No exemplo a seguir mostramos que acessando o *original_manager* podemos fazer consultas nos dados de todos os *tenants*, independete de qual *tenant* está definido no `TenantMiddleware`.

```
1 set_current_tenant('tenant1')
2 t1 = MyModel.objects.all()
3 # t1 recebe os objetos do 'tenant1'
4
5 set_current_tenant('tenant2')
6 t2 = MyModel.objects.all()
7 # t1 recebe os objetos do 'tenant2'
8
9 t_all = MyModel.original_manager.all()
10 # t_all recebe todas os objetos do banco, independente de tenants
```

Apesar de estar disponível em qualquer lugar da aplicação, o `original_manager` dos *models* deve ser utilizado com muito cuidado, pois ele abre precedentes para que dados de um *tenant* acabem ficando disponíveis para outros.

3.7 Criação de campos e tabelas específicos por tenant

A técnica que usamos para criação de tabelas e campos específicos para *tenants* foi a das tabelas *pivot*.

Essa técnica dá bastante flexibilidade na criação das tabelas, pois permite a criação e alteração de tabelas virtuais pela própria aplicação, sem causar qualquer modificação no esquema físico do banco de dados. Porém o uso de tabelas *pivot* torna muito mais caro fazer consultas nos dados dessas tabelas virtuais.

A modelagem do banco da *Django-SSTenants* segue quase totalmente as especificações da abordagem de tabelas *pivot*. A *Django-SSTenants* cria várias tabelas, uma para cada possibilidade de tipo de dado que possa ser armazenado (inteiros, textos, pontos flutuantes, datas etc). Porém, cada tabela, ao invés de conter apenas a referência à tabela lógica criada, contém também uma referência ao *content type* (*model* definido no *framework* da biblioteca padrão do *Django*, *Django ContentTypes*).

O *Django ContentTypes* define uma tabela no banco de dados que armazena todos os *models* da aplicação. Dessa maneira, pode-se referenciar *models*/tabelas do banco em chaves estrangeiras. A *Django-SSTenants* utiliza desse recurso, para implementar não só a criação de tabelas lógicas específicas para *tenants*, como também a extensão de tabelas físicas com colunas lógicas.

A seguir, apresentamos um exemplo de tabela *pivot* de dados do tipo inteiro. Observa-se que ela possui um campo `row_content_type` referenciando um *ContentType*, que na implementação do *Django-SSTenants* pode referenciar uma linha de tabela física ou uma linha de tabela virtual.

```
1  class TenantSpecificPivotTable(models.Model):
2      definition = models.ForeignKey('TenantSpecificFieldDefinition')
3
4      row_content_type = models.ForeignKey(
5          ContentType, on_delete=models.CASCADE)
6      row_id = models.PositiveIntegerField()
7      row = GenericForeignKey(
8          ct_field='row_content_type', fk_field='row_id')
9
10     def __str__(self):
11         return '%s: %s' % (str(self.definition), self.value)
12
13     class Meta:
14         unique_together = [
15             ('definition', 'row_id', 'row_content_type')]
16         abstract = True
17
18 class TenantSpecificFieldIntegerPivot(
19         SingleTenantModelMixin, TenantSpecificPivotTable):
20     value = models.IntegerField()
21
22     def __str__(self):
23         return '%s: %d' % (str(self.definition), self.value)
```

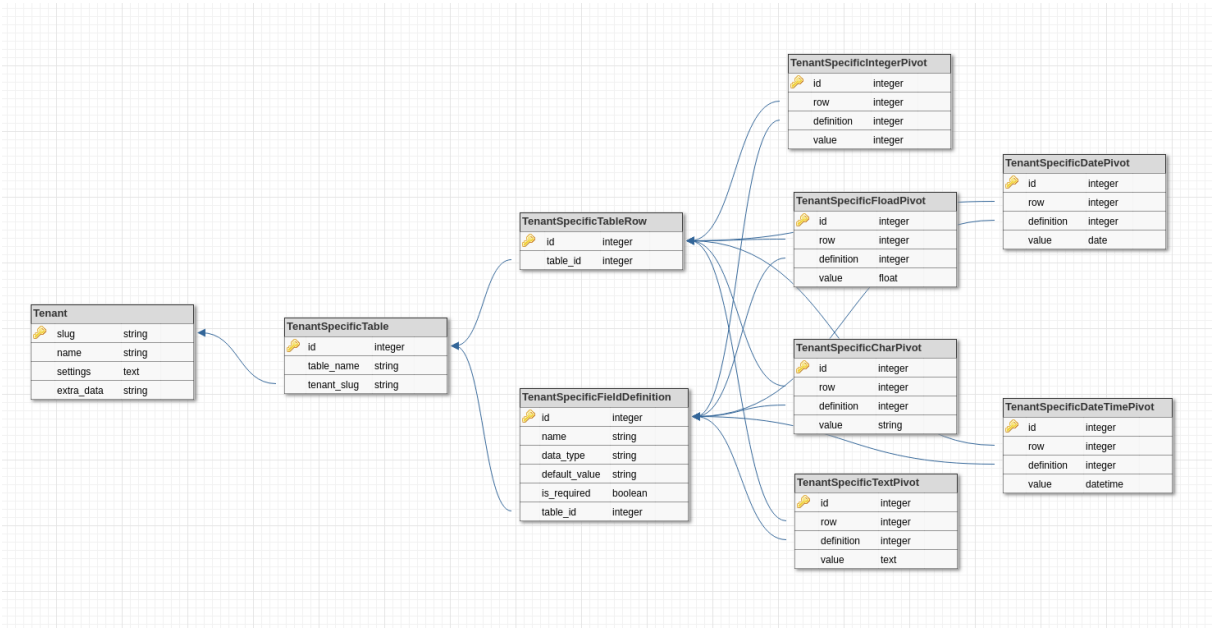


Figura 3.2 Esquema relacional da funcionalidade de tabelas específicas por *tenant* da *Django-SSTenants*. Fonte: Próprio Autor

---

Além da criação das tabelas de dados específicos, a *Django-SSTenants* também cria *Managers* e *Querysets* próprios que permitem que os usuários da mesma tratem colunas lógicas da mesma forma que tratam colunas físicas tanto em consultas, quanto na criação e atualização dos dados.

No caso das consultas, a adição das colunas lógicas se dá a partir de agregações e subconsultas, o que faz com que a quantidade de acesso ao banco permaneça a mesma que seria em uma aplicação comum em *Django*, porém tornando as consultas realizadas mais complexas e custosas.

Já a criação e atualização dos dados ocorrem da seguinte forma: primeiro são atualizadas as colunas físicas; após a linha física ser salva, são criado/atualizadas as colunas virtuais uma a uma.

O esquema relacional da implementação de tabelas pivot da *Django-SSTenants* pode ser observado na Figura 3.2.

3.8 Considerações Finais

Nesse capítulo, vimos que a *Django-SSTenants*, utilizando as abordagens de tabelas compartilhadas para separação dos dados e tabelas *pivot* para extensão das tabelas do banco de dados por *tenant*, facilita a implementação de arquiteturas *multi-tenant* com *Django*. Vimos também algumas as ferramentas disponíveis em sua API para tornar simples a identificação do *tenant*

pelas requisições HTTP recebidas e funcionalidades como Autenticação, sempre integrado ao *framework*.

CAPÍTULO 4

Avaliação da *Django-SSTenants*

Nesse capítulo, apresentaremos uma avaliação comparativa da *Django-SSTenants* com outra biblioteca de código aberto para implementação da arquitetura *multi-tenant* em relação às funcionalidades disponíveis e a complexidade de implantação.

A bibliotecas usada para comparação será a Django-Tenants, a biblioteca *open-source* mais popular para implementação de *multi-tenancy* em projetos *Django*.

4.1 Avaliação Qualitativa

Aqui faremos uma comparação das abordagens utilizadas pelas duas bibliotecas e das funcionalidades que ambas disponibilizam em suas APIs.

A *Django-Tenants*, assim como a *Django-SSTenants*, possui foco em tornar o mais transparente possível o uso da arquitetura *multi-tenant*, permitindo assim que o usuário se preocupe apenas com as regras de negócio.

4.1.1 Separação de dados

Essa biblioteca utiliza a abordagem de múltiplos *schemas* para implementação da arquitetura *multi-tenant*, isto é, cada *tenant* possui o seu próprio *schema* mas a aplicação utiliza apenas um banco de dados.

Ela funciona utilizando um *driver* próprio de integração com o *PostgreSQL* (único SGBD suportado pela biblioteca), um roteador de banco também customizado e um *middleware*, que permitem a seleção do *schema* correto para cada requisição a partir do domínio.

Essa abordagem garante um bom desempenho, principalmente em aplicações com poucos *tenants*. Se a quantidade de *tenants* cresce muito, o *PostgresSQL* apresenta alguns problemas de *backup* e recupeção de dados [Ste13].

Em relação a separação dos dados, a *Django-Tenants* leva vantagem sobre a *Django-SSTenants*, pois os dados estão separados a nível de banco, e não de aplicação. Porém, essa característica dá certa flexibilidade à *Django-SSTenants* quando é necessário fazer uma agregação de informações de mais de um *tenant*. Um exemplo disso é um relatório ou painel administrativo que mostra informações analíticas de diversos *tenants*.

4.1.2 Recuperação do *tenant* da requisição

A *Django-Tenants* recupera o *tenant* a partir do domínio da requisição. Ele compara o domínio com uma tabela de domínios, definida pela biblioteca em um *model* `Domain`, na qual cada

linha possui uma referência para um *tenant*.

Esse comportamento é um tanto limitado em relação ao da *Django-SSTenants*, que por padrão extrai a informação do domínio ou de um cabeçalho HTTP, fornecendo total liberdade para desenvolvedores customizarem essa lógica através dos *tenant retrievers*.

4.1.3 Autenticação

Por padrão, a *Django-Tenants* não provê uma forma de integração de autenticação de usuários entre *tenants*, isto é, permitir que um usuário tenha permissões em mais de um *tenant*. Para conseguir acessar mais de um *tenant*, os usuários dos sistemas desenvolvidos com a *Django-Tenants* devem ter uma conta para cada *tenant*. Outra alternativa na *Django-Tenants* é tornar a tabela de usuários compartilhada entre os *tenants*. Porém, o sistema de autenticação e permissões deve ser implementado fora da biblioteca.

Como na *Django-SSTenants* a divisão dos dados é feita apenas a nível de aplicação, o sistema de autenticação pode ser mais flexível, dando a possibilidade ao desenvolvedor de permitir que o mesmo usuário tenha acesso a mais de um *tenant* com a apenas uma conta sem ter que implementar essa lógica.

4.1.4 Dados específicos por tenants

A *Django-Tenants* dá suporte a criação de *apps* do *Django* específicos por *tenant*, isto é, possibilita ao desenvolvedor criar tabelas específicas por *tenant*, que só estarão disponíveis no *schema* dele.

Uma desvantagem dessa abordagem é que a customização do banco de dados depende de desenvolvedores para adição de dados específicos dos *tenants*. Dessa forma, se mantém o padrão do *Django* na criação de *models*, mantendo a consistência do código.

Em contraste a isso, a *Django-SSTenants* salva metadados como dados. Isso diminui a consistência do código, pois os campos e tabelas virtuais não estarão descritos no mesmo, porém possibilita que usuários do sistema criem tabelas lógicas e tabelas lógicas estendidas sem ter que alterar o esquema físico, isto é, sem alterar as tabelas físicas do banco de dados.

4.2 Avaliação Quantitativa

Nessa sessão mostraremos um plano de avaliação para comparação do desempenho das bibliotecas *Django-SSTenants* e a *Django-Tenants* e, posteriormente, os dados e conclusões obtidos com a execução do plano.

4.2.1 Plano de Avaliação

Para construção desse plano foi utilizado o método descrito em [Jai91]. Aqui descreveremos quais os critérios e métodos usados para a avaliação, assim como a estratégia utilizada para compilação dos resultados.

4.2.1.1 Definição do sistema

O objetivo do estudo é comparar o desempenho de aplicações *multi-tenant* construídas com o *framework Django* utilizando a biblioteca *Django-Tenants* e a *Django-SSTenants*. O principal componente desse estudo são as consultas dos dados de um *tenant* no banco de dados. Cada uma das bibliotecas utiliza uma maneira diferente de abstrair a divisão de dados por *tenant*. Consequentemente, é diferente a forma como é feita a consulta dos dados.

4.2.1.2 Serviços

Os serviços são *tenants* separados por *schema* do Banco de Dados (*Django-Tenants*) e *tenants* com tabelas compartilhadas (*Django-SSTenants*). Nesse estudo, escolhemos uma aplicação com uma consulta simples e uma consulta complexa para avaliação do desempenho em cada uma delas utilizado a primeira e a segunda biblioteca. A aplicação escolhida foi um software gerenciamento de entregas de comida.

4.2.1.3 Métricas

Nesse estudo, a métrica utilizada é o tempo para execução de uma consulta em um banco de dados com 10, 100 e 10000 *tenants*. Isto é, cada uma das bibliotecas terá avaliado:

(a) O tempo para execução de consulta simples com 10 *tenants* cadastrados;

(b) O tempo para execução de consulta complexa com 10 *tenants* cadastrados;

(c) O tempo para execução de consulta simples com 100 *tenants* cadastrados;

(d) O tempo para execução de consulta complexa com 100 *tenants* cadastrados;

(e) O tempo para execução de consulta simples com 10000 *tenants* cadastrados;

(f) O tempo para execução de consulta complexa com 10000 *tenants* cadastrados.

4.2.1.4 Parâmetros

Os parâmetros de carga que afetam o desempenho da aplicação são:

(a) A biblioteca utilizada para *multi-tenancy*

(b) A complexidade das consultas

(c) A quantidade de *tenants* cadastrados

Os fatores que foram selecionados para esse estudo são:

(a) A biblioteca utilizada: duas bibliotecas serão comparadas, a *Django-Tenants* e a *Django-SSTenants*;

(b) A complexidade das consulta: duas consultas serão utilizadas uma mais simples e uma mais complexa (que faz uso de sub-consultas);

(c) A quantidade de *tenants* cadastrados: serão utilizados 10, 100 e 10000 *tenants* para comparação.

Em relação aos parâmetros do sistema, temos a seguinte configuração no computador que rodou os experimentos:

- Processador: i7-6500U com 2 *cores*, 4 *threads*, frequência base de 2.50 GHz;
- Memória: 8GB DDR4-2132, *Dual-Channel*;
- Sistema Operacional: Ubuntu 16.04 LTS.

Técnica de avaliação: Duas aplicações com o mesmo escopo (software de gerenciamento de entregas de comida) foram desenvolvidas, uma utilizando *Django-Tenants*, a outra utilizando *Django-SSTenants*. Foram feitas medidas de tempo de execução de consultas nessas aplicações. As consultas foram executadas utilizando diferentes parâmetros relativos aos fatores definidos.

Carga: Foi criado um banco com 10, 100 e 10000 *tenants* para cada uma das bibliotecas. Nessa aplicação, usuários tem uma lista de refeições disponíveis. Cada refeição possui uma lista de ingredientes. Cada usuário possui uma lista de ingredientes aos quais ele possui alergia. Cada usuário tem uma lista de refeições as quais marcou que não gosta. Cada *tenant* conta com 10 usuários, 9 ingredientes, cada usuário possui alergia a 1 ingrediente, 100 refeições, cada refeição possui 1 ingrediente em sua composição, e 1/3 das refeições (33) foi avaliada negativamente por cada usuário. Cada aplicação conta com uma consulta simples e uma complexa. Como consulta simples temos a lista de refeições cadastradas por *tenant* no banco de dados. Como consulta complexa temos a lista de refeições que não possui nenhum ingrediente alergênico (ingredientes que não foram marcado como alergênico por nenhum usuário) e que também não foi negativada por nenhum usuário.

Design do experimento: Para avaliação, cada uma das consultas será executada 1000 (mil) vezes em cada banco. O tempo, em segundos, de todas essas execuções num banco será o valor utilizado para um experimento. Cada experimento será executado 100 (cem) vezes.

Análise e interpretação dos dados: A análise será feita por comparação da média dos valores dos experimentos em cada combinação de fatores.

Apresentação dos dados: As médias serão plotados em função dos fatores selecionados (biblioteca utilizada, complexidade das consultas e quantidade de *tenants*).

4.2.2 Resultados e Conclusões

Após realizar o experimento proposto, obtivemos os resultados mostrados nas Figuras 4.1 e 4.2. Nestas figurea, vemos que a *Django-SSTenants* foi mais lenta que a *Django-Tenants* em todos os experimentos. Porém, a diferença foi proporcional a uma constante nas duas consultas e em todas as configurações de quantidades de *tenants*.

No caso da consulta mais complexa, tivemos um resultado pior da *Django-SSTenants*, o que mostra que a complexidade da consulta influencia bastante na diferença de desempenho entre as duas bibliotecas. Da consulta simples para a consulta complexa, a *Django-Tenants* teve um aumento percentual das médias de aproximadamente 549,13%. Já na *Django-SSTenants* o aumento das médias foi de aproximadamente 1144,03%.

Como conclusão, vemos que o desempenho da *Django-SSTenants* foi inferior em todas as situações. Por isso, recomenda-se o uso da biblioteca somente se o desempenho de consultas não for um fator crucial para a aplicação em questão. Ainda assim, a *Django-SSTenants* possui diversas vantagens em relação a sua API e a possibilidade de customização, sendo ainda uma alternativa viável em certas aplicações.

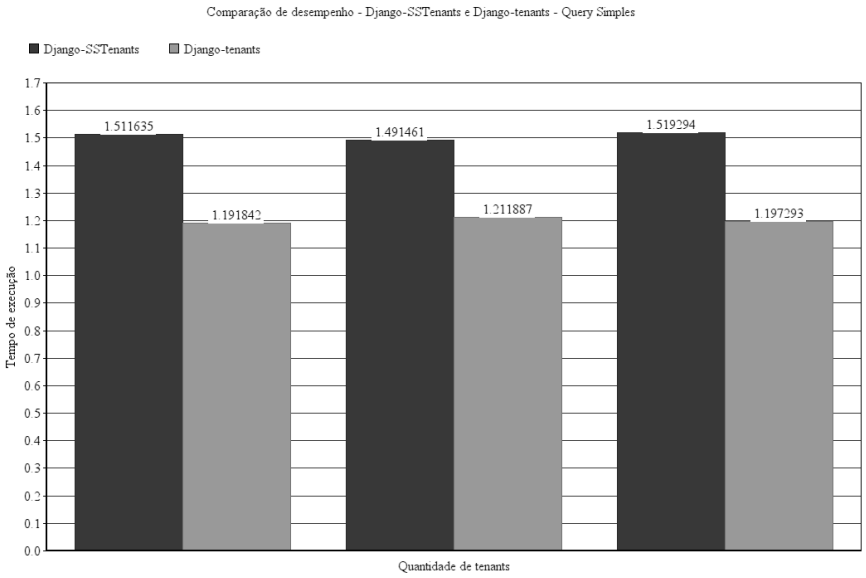


Figura 4.1 Comparação de desempenho - *Django-SSTenants* e *Django-Tenants* - *Query* Simples. Fonte: Próprio Autor

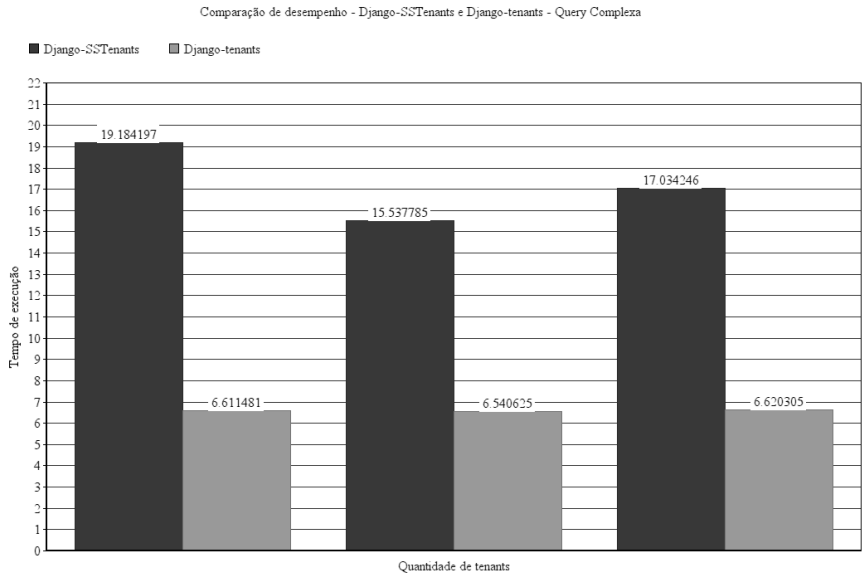


Figura 4.2 Comparação de desempenho - *Django-SSTenants* e *Django-Tenants* - *Query* Complexa. Fonte: Próprio Autor

CAPÍTULO 5

Conclusões e Trabalhos Futuros

Nesse capítulo, traremos algumas conclusões adquiridas nesse trabalho e detalharemos quais os próximos passos para melhoria da biblioteca *Django-SSTenants*.

5.1 Conclusões

Multi-tenancy é uma arquitetura que pode trazer muitos benefícios à diversas aplicações que precisam executar em muitos clientes, ao mesmo tempo que possui muitos desafios. Nesse trabalho foi possível conhecer alguns dos principais desafios e preocupações de quem implementa aplicações baseadas nesta arquitetura.

Motivados pelos inúmeros desafios que a arquitetura *multi-tenancy* possui, muitas ferramentas tem surgido para facilitar o desenvolvimento de aplicações tirando dos desenvolvedores algumas dessas preocupações.

O *Django* não suporta *multi-tenancy* por padrão, mas já existem ferramentas de terceiros que auxiliam o desenvolvimento de aplicações utilizando a arquitetura. A *Django-SSTenants*, biblioteca desenvolvida, apresenta uma nova estratégia para reduzir a complexidade da implementação de software *multi-tenant* escritos em *Django*.

Como diferencial, a *Django-SSTenants* utiliza a abordagem de *schema* compartilhado e com extensão de tabelas por *tenant* implementada através de tabelas *pivot*. Essa escolha de abordagens possui como vantagens a maior facilidade de agregação dos dados de *tenants* diferentes e a facilidade de migração para a arquitetura *multi-tenant* de aplicações que já não a utilizam.

A biblioteca desenvolvida no decorrer desse projeto é de código aberto. Para ter acesso ao código-fonte basta acessar https://github.com/hugobessa/django-shared-schema-tenants. A biblioteca tambem foi adicionada a PyPI, API que centraliza e indexa diversos pacotes Python, e pode ser acessada no link https://pypi.org/project/django-shared-schema-tenants/.

5.2 Melhorias na biblioteca

A *Django-SSTenants* encontra-se estável e publica, porém existem diversas maneiras de tornar a biblioteca mais robusta. Seguem alguns pontes que podem ser trabalhados de forma a melhorá-la:

Melhor integração com o *django-admin*: Uma das melhores funcionalidades do *Django* é a área administrativa que ele fornece baseada nos *models* definidos na aplicação. Atualmente, a *Django-SSTenants* possui uma integração mínima com o *django-admin* permi-

tindo a criação de novos *tenants*. Porém, essa integração deve ser expandida de forma a permitir que usuários dos *tenants* possam acessá-la sem notar que estão utilizando uma aplicação *multi-tenant*;

Adicionar funções para controlar tabelas e campos lógicos: A *Django-SSTenants* já possui total suporte ao gerenciamento de tabelas e campos lógicos específicos por *tenant*. Os planos futuros são para criar mais funções auxiliares de maneira que o usuário da biblioteca tenha que ter menos conhecimento sobre a modelagem física dos dados;

Customização do *model* de *tenants*: Hoje já é possível ter flexibilidade de adicionar as mais diversas informações ao *model* `Tenant` utilizando os campos JSON `extra_data` e `settings`, porém seria interessante que o usuário da biblioteca tenha possibilidade de usar um *model* customizado no lugar do padrão, de maneira a ter campos físicos na tabela;

Melhorar o desempenho das consultas: Hoje as consultas em aplicações utilizando a *Django-SSTenants* estão levando mais tempo que consultas similares em aplicações utilizando a *django-tenants*. Isso está muito relacionado à natureza da abordagem utilizada na *Django-SSTenants* para separação dos *tenants*, porém existem maneiras de otimizar a parte de consultas da biblioteca para reduzir o tempo de execução das mesmas;

Melhorar a cobertura do código: Hoje a cobertura da suíte de teste abrange cerca de 85% das linhas de código. O planos são de desenvolver mais testes até que a cobertura chegue a 100% das linhas;

Melhorar a documentação: A biblioteca já conta com um manual para utilização das funcionalidades principais que a biblioteca provê, mas ainda carece de documentação de algumas funcionalidades secundárias e mais recentes;

Criar guias para contribuição: Desenvolver alguns guias para que qualquer desenvolvedor consiga contribuir com o desenvolvimento e manutenção da biblioteca.

5.3 Trabalhos Futuros

A partir do que foi desenvolvido, seguem alguns tópicos que servem como base para trabalhos futuros:

Relação entre a arquitetura *multi-tenants* e dados poliglotas: Em várias ocasiões a *Django-SSTenants* faz uso do banco de dados padrão da aplicação *Django* quando poderia utilizar ferramentas que trouxessem maior desempenho e/ou flexibilidade à aplicação. Ferramentas como banco de dados NoSQL e chave-valor podem trazer grandes ganhos se utilizadas da maneira correta. Investigar que interações com essas ferramentas podem trazer ganhos a aplicações *multi-tenants* seria de grande valor.

Relação entre a arquitetura *multi-tenants* e micro-serviços: O uso de micro-serviços tem se tornado muito popular recentemente facilidade de escalar apenas partes da aplicação que demandam mais recursos. Porém existem muitos cuidados a se tomar quando utilizar micro-serviços numa aplicação *multi-tenants*, pois a distribuição da informação pode gerar problemas com manutenção da consistência e a separação dos dados. Um estudo sobre as implicações que a arquitetura *multi-tenants* no desenvolvimento de micro-serviços seria um bom trabalho futuro;

Referências Bibliográficas

[BP17] Thomas Turner Bernardo Pires. *Django-tenants Documentation*, 2017. [Online; acessado em 26-11-2017].

[dja18] Django, ruby on rails, express.js, zend framework - explore - google trends. `https://trends.google.com/trends/explore?cat=32&date=2004-01-01%202018-06-12&q=%2Fm%2F06y_qx,%2Fm%2F0505cl,%2Fm%2F0_v2szx,%2Fm%2F0cdvjh`, 2018. [Online; acessado em 12-06-2018].

[EGH00] Ralph Johnson Erich Gamma, John Vlissides and Richard Helm. *Padrões de Projeto - Soluções Reutilizaveis de Software Orientado a Objetos*. Bookman, 2000.

[FK92] David F. Ferraiolo and D. Richard Kuhn. Role based access control. 1992. [Online; acessado em 08-06-2018].

[Fou05] Django Software Foundation. *Django Documentation*, 2005. [Online; acessado em 08-06-2018].

[Fus97] Mark Fussell. Foundations of object-relational mapping. 1997. [Online; acessado em 08-06-2018].

[Geo17] Nigel George. *The Django Book*. 2017. [Online; acessado em 26-11-2017].

[GM14] Basant Kumar Gupta and Keisam Thoiba Meetei. Multi-tenant saas application platform: A survey. *International Journal of Science and Research (IJSR)*, 3(6), 2014.

[Gro96] The PostgreSQL Global Development Group. *PostgreSQL: Documentation*, 1996. [Online; acessado em 08-06-2018].

[HDX12] Li Heng, Yang Dan, and Zhang Xiaohong. Survey on multi-tenant data architecture for saas. *International Journal of Computer Science Issues (IJCSI)*, 9(6), 2012.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1991.

[RKK12] Christof Momm Rouven Krebs and Samuel Kounev. Architectural concerns in multi-tenant saas applications. 2012. [Online; acessado em 08-06-2018].

[Ros14] Roberto Rosario. Awesome django: A curated list of django apps, projects and resources. https://gitlab.com/rosarior/awesome-django, 2014. [Online; acessado em 08-06-2018].

[Ste13] Pavel Stehule. Stackoverflow answer for how many schemas can be created in postgres. https://stackoverflow.com/a/14895450/1772934, 2013. [Online; acessado em 08-06-2018].