Uma biblioteca Multi-Tenant para o framework Django

por

José de Arimatea Rocha Neto

Trabalho de Graduação

UNIVERSIDADE FEDERAL DE PERNAMBUCO

CIN - CENTRO DE INFORMÁTICA

GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

www.cin.ufpe.br

RECIFE, JULHO DE 2016

UNIVERSIDADE FEDERAL DE PERNAMBUCO

CIN - CENTRO DE INFORMÁTICA

GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

José de Arimatea Rocha Neto

UMA BIBLIOTECA MULTI-TENANT PARA O FRAMEWORK DJANGO

Projeto de Graduação apresentado no Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Vinícius Cardoso Garcia

RECIFE, JULHO DE 2016

“A menos que modifiquemos a nossa maneira de pensar, não seremos capazes de resolver os problemas causados pela forma como nos acostumamos a ver o mundo”.

Albert Einstein

Agradecimentos

Agradeço à minha família pelo amor, dedicação e paciência.

Agradeço à minha noiva Priscila por sempre estar ao meu lado me apoiando em todos os momentos.

Agradeço aos meus amigos pelo incentivo e apoio durante toda a graduação.

Agradeço ao meu orientador Vinícius Garcia pela sua disponibilidade e por todo o apoio fornecido.

Agradeço a todos os professores do CIn por todo o conhecimento e dedicação fornecidos durante o curso.

Resumo

Nos últimos anos o modelo de entrega de software como serviço, ou Software as a Service (SaaS), surgiu trazendo softwares mais flexíveis e reutilizáveis. Este modelo provê suporte a diversos usuários sobre uma mesma infra-estrutura configurável, oferecendo funcionalidades sob demanda. *Multi-Tenancy*, ou multi-inquilino, é uma abordagem organizacional do modelo SaaS. Características de uma arquitetura *multi-tenant* são: compartilhamento de recursos de hardware, alto grau de configurabilidade e bancos de dados compartilhados. Alguns benefícios da utilização desta arquitetura são: maior utilização dos recursos de *hardware*, manutenção da aplicação facilitada e mais barata e redução nos custos globais do sistema. Tomando como base as características desta arquitetura, esta pesquisa propõe uma biblioteca *open source* para transformar projetos *Django* em um projeto *multi-tenant*, com o intuito de facilitar seu desenvolvimento. Assim, os desenvolvedores podem focar seus esforços na regra de negócio e não com a arquitetura, já implementada pela biblioteca.

Palavras-chave: *Multi-Tenancy*, *Multi-Tenant*, *Software as a Service*, *open source*, Arquitetura de *software*, *Django*.

Abstract

In recent years the Software as a Service (SaaS) model came bringing more flexible and reusable software. This model provides support for multiple users on the same configurable infrastructure by offering features on demand. Multi-Tenancy is an organizational approach of Software as a Service (SaaS). Some characteristics of a multi-tenant architecture are: the capability to share hardware and database resources and a high degree of configurability. Additionally, the use of this architecture brings benefits. Some of them are: better use of hardware resources, simpler and cheaper application maintenance and reduction of the system global costs. Based on the architectural characteristics, this paper proposes an open source library that facilitates the development of multi-tenant projects on Django framework. Thus, the developers can focus on the business rules and less on the architecture that is already implemented by the library.

Keywords: Multi-Tenancy, Multi-Tenant, Software as a Service, Open Source, Software Architecture, Django.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Neste capítulo será apresentada a motivação para a realização deste trabalho, os objetivos esperados e como este trabalho está estruturado, a fim de facilitar o entendimento do leitor.

1.1 Motivação

A Computação em Nuvem, ou *Cloud Computing*, é uma tecnologia global que permite acessar recursos como aplicações e servidores de forma facilitada. Esta tecnologia tem viabilizado o desenvolvimento e manutenção de sistemas de informação em diversos tipos de negócio [34]. Por isso, esta tecnologia vem sendo amplamente utilizada por diversas organizações.

De acordo com o estudo de Samorani [34], com a Computação em Nuvem, empresas em diversos setores, tanto de grande porte como de pequeno porte, vêm diminuindo seus custos, principalmente na infraestrutura, ao investir nos serviços de TI. Esta tecnologia permite que empresários e empreendedores estabeleçam a infraestrutura de seus negócios com um custo muito reduzido quando comparado aos investimentos feitos antes do século XXI.

Segundo o *National Institute of Standards and Technology* (NIST), a Computação em Nuvem pode ser dividida em três modelos de serviço: Software como Serviço (SaaS),

Plataforma como Serviço (PaaS) e Infraestrutura como Serviço (IaaS) [9].

O modelo SaaS, foco deste trabalho, pode ser definido como software implantado como serviço hospedado e acessado pela Internet [4]. Aplicativos de SaaS oferecem benefícios como uma arquitetura de instância única, para vários inquilinos e uma experiência rica em recursos. Geralmente, os aplicativos SaaS são vendidos com um modelo de assinatura onde os clientes pagam uma taxa contínua para uso do aplicativo. O SaaS oferece oportunidades a empresas de todos os portes de diminuir os riscos na aquisição de software.

Como indicado no trabalho de Custódio e Junior [9], *Multi-Tenancy*, ou multi-inquilino, é uma abordagem organizacional do modelo SaaS. Algumas das características de aplicações que utilizam arquitetura *Multi-Tenant* são [2]:

- Compartilhamento de recursos de hardware;
- Alto grau de configurabilidade;
- Aplicação e instância de banco de dados compartilhados.

Para Kabbedijk, Jaap, et al. [18], *Multi-Tenancy* é uma propriedade de um sistema onde vários clientes, também chamados de inquilinos, compartilham serviços, aplicações, bancos de dados, entre outros. Assim, é possível tanto reduzir custos, como ser capaz de configurar os sistemas voltados para as necessidades dos inquilinos.

Um ambiente *Multi-Tenancy* provê benefícios como [2]:

- Maior utilização dos recursos de hardware;
- Manutenção da aplicação torna-se mais fácil de ser realizada e mais barata;
- Custos globais reduzidos, permitindo oferecer um serviço com menor custo quando comparado aos concorrentes.

Para facilitar o desenvolvimento da biblioteca proposta, faz-se necessário o uso de um *framework web*. Dentre os disponíveis, o escolhido foi o Django, que é um dos mais utilizados hoje em dia [1]. Além disso, Python vem crescendo bastante no mercado mundial e hoje é a quarta linguagem de programação mais utilizada no mundo [35]. Para

exemplificar esse crescimento, vemos que projetos como Google, YouTube, Instagram, DropBox, Reddit, Pinterest e Spotify vem usando Python e Django na sua *stack* de desenvolvimento [12] [16].

1.2 Objetivos

1.2.1 Objetivos Gerais

O objetivo desta pesquisa é criar uma biblioteca *open source* que possa transformar projetos desenvolvidos em Django em um projeto *Multi-Tenant*. Com isso, desenvolvedores que utilizem o *framework* podem adotar a arquitetura *Multi-Tenant* como recurso para seus projetos, focando os seus esforços na regra de negócio do sistema e deixando a lógica da arquitetura *Multi-Tenant* por conta da biblioteca.

1.2.2 Objetivos Específicos

- Fazer um estudo acerca da arquitetura *Multi-Tenant* e analisar algumas técnicas de implementação das suas restrições;
- Fazer um comparativo das vantagens e das desvantagens de usar essa arquitetura;
- Investigar e propor uma biblioteca *Multi-Tenant* para o *framework* Django;

1.3 Organização do Trabalho

O presente trabalho está organizado em sete capítulos, dos quais o primeiro é a introdução e os próximos seis capítulos estão descritos abaixo:

- No Capítulo 2 é apresentado um conjunto de definições relevantes para o entendimento do trabalho;

- No Capítulo 3 é apresentado o conceito de *Multi-Tenancy*, juntamente com técnicas para a implementação da arquitetura e um comparativo das vantagens e desvantagens da mesma;
- No Capítulo 4 são apresentados trabalhos relacionados ao tema desta pesquisa;
- No Capítulo 5 é apresentada toda a implementação da biblioteca;
- No Capítulo 6 é apresentado um exemplo de utilização da biblioteca proposta neste trabalho;
- No Capítulo 7 são apresentados considerações finais do trabalho assim como as propostas para trabalhos futuros;

Capítulo 2

Referencial Teórico

Este capítulo é responsável por apresentar conceitos chave para o melhor entendimento deste trabalho, trazendo a definição dos conceitos e uma visão geral sobre os seguintes temas: Computação em Nuvem, Software como um Serviço (SaaS) e o *framework Django*. A computação em nuvem provê diversos serviços baseados em web com foco em aumentar a capacidade de processamento e armazenamento. Um dos modelos de negócio da computação em nuvem é o SaaS. Com foco neste modelo e na arquitetura multi-tenant, este trabalho propõe uma biblioteca Django para facilitar a utilização da arquitetura em projetos que utilizam este framework. Ao fim deste capítulo, o leitor será capaz de entender melhor o conteúdo do trabalho e a linha de raciocínio adotada no mesmo.

2.1 Computação em Nuvem

As primeiras citações referentes à Computação em Nuvem remetem à origem da *Utility Computing*, proposta por John McCarthy em sua apresentação no *Massachusetts Institute of Technology* (MIT) no ano de 1961. Segue uma passagem desta, emitida no jornal do MIT [11] em 1999:

> ”If computers of the kind I have advocated become the computers of the future, then computing may someday be organized as a public utility just as the telephone system is a public utility. ... The computer utility could become the basis of a new and important industry.”

Posteriormente, essa mesma definição foi citada por Leonard Kleinrock, cientista chefe da ARPANET, em 1969 [10].

Com o passar dos tempos, a definição de *Cloud Computing* foi sendo atualizada, e por volta da década de 1990 o termo *Cloud* era comumente utilizado na indústria de comunicações. *Network Cloud* era o nome dado a uma camada de abstração da transferência de dados entre provedor e cliente [10], como pode ser visto na Figura 2.1.

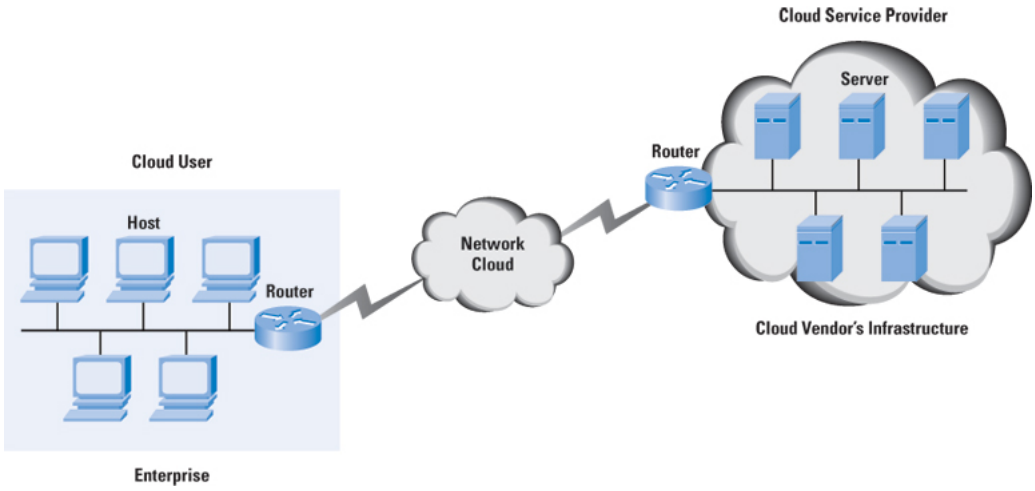

Figura 2.1: Network Cloud [33]

Foi somente no ano de 2006 que a definição mais atual de Computação em Nuvem foi introduzida por Eric Schmidt, quando em sua palestra no *Search Engine Strategies Conference*, ele anunciou o novo produto da Amazon: o *Elastic Compute Cloud* (EC2) [30].

Por remeter a várias tecnologias diferentes ao longo da história, atualmente não existe uma definição oficial de *Cloud Computing*. Dessa forma, é necessário fazer um estudo de diversas definições diferentes para se chegar na mais adequada. Em seu trabalho, Rodrigues [29] fez um agrupamento de várias definições propostas por especialistas da área [37], como pode ser visto na Tabela 2.1.

Autor	**Definição**
M. Klems	... é possível escalar sua infraestrutura sob demanda em minutos ou em segundos, ao invés de dias ou semanas, evitando, desse modo, sub-utilização e sobrecarga dos seus recursos *in-house*.
P. Gaw	Uso da internet para permitir pessoas acessarem serviços tecnologicamente disponíveis. Esses serviços precisam ser massivamente escaláveis.
R. Buyya	Uma *Cloud* é um tipo de sistema pararelo distribuído que consistem de uma coleção de computadores virtualizados e interconectados que são dinâmicamente provisionados e apresentados como um ou mais recursos computacionais unificados, baseados em SLA estabelecida através de negociação entre o provedor de serviço e o consumidor.
R. Cohen	*Cloud Computing* é uma das várias *buzz words* existentes que tentam abranger uma variedade de aspectos com *deployment*, balanceamento de carga, provisionamento, modelo de negócios e arquitetura. Esse é o próximo passo em software. A explicação mais simples de *Cloud Computing* é descrevê-la como ”Software centrado na Internet”.
J. Kaplan	Uma ampla gama de serviços baseados na web que objetivam permitir ao usuário obter uma variedade de funcionalidades que são pagas por uso e que anteriormente requeria uma grande quantidade de investimentos em hardware/software e contratação de bons profissionais. *Cloud Computing* é a realizaçnao de idéias antigas de computação utilitária sem a complexidade técnica ou preocupação de *deployments* complicados.
D. Gourlay	... o próximo termo para campanhas publicitárias ... criado fora do modelo de software que a virtualização habilitou.
D. Edwards	... é o que é possível quando você alcança a infra-estrutura (de aplicação e física) para o tamanho da web de uma forma sob-demanda.
B. de Haff	Existem somente três tipos de serviço que são baseados em *cloud*: SaaS, PaaS e Plataformas de *Cloud Computing*. O autor não garante que escalar massivamente seja um requisito para caber em qualquer categoria.
B. Kepes	Simplificando, *Cloud Computing* é uma mudança de paradigma de infraestrutura que habilita a ascenção de software como serviço ... É uma gama de serviços baseados em web que tem como objetivo permitir a usuários obter uma ampla variedade de funcionalidades baseadas no modelo de pagamento por uso e que anteriormente requeria uma grande quantidade de investimentos e contratação de profissionais especialistas.

K. Sheynkman	*Cloud* foca em fazer a capacidade de processamento e armazenamento da camada de hardware consumível sob demanda. Esse é um primeiro passo importante, mas para as companhias aproveitarem o poder da *cloud*, infra-estruturas completas de aplicações precisam ser facilmente configuradas, distribuídas, dinâmicamente escaladas e gerenciadas nesses ambientes de *hardware* virtualizado.
K. Harting	Na realidade, é o acesso a recursos e serviços necessários para executar funcionalidades com necessidade de mudança dinâmica. É a virtualização de recursos auto-mantidos e auto-gerenciáveis.
J. Pritzker	*Clouds* são um vasto *pool* de recursos com alocação sob demanda... virtualizados ... e cobrados por utilização.
T. Doerkzen	*Cloud Computing* é... uma versão de *grid computing* amigável para o usuário.
T. Von Eicken	terceirizado, sob-demanda, pago por uso, em qualquer lugar da internet, ets.
M. Sheedan	a pirâmide de *cloud computing* ajuda a diferenciar as várias ofertas de *cloud*... no topo: SaaS, no meio: PaaS, na base: IaaS.
A. Ricadela	Projetos de *cloud computing* são mais poderosos e tolerantes a falhas que sistemas de *grid* desenvolvidos nos últimos anos.
I. Wladawsky Berger	A principal coisa que queremos virtualizar e ocultar do usuário é a complexidade ... todos aqueles *softwares* que serão virtualizados ou ocultod de nós e o tratamento de sistemas e/ou profissionais que estão em qualquer outro lugar, fora da *Cloud*.
B. Martin	*Cloud computing* inclui qualquer serviço baseado em assinatura ou pago por uso que, em tempo real, e rodando na internet, extende as capacidades de TI existentes.
R. Bragg	O conceito principal de *cloud* é a aplicação da *web* ... a *cloud* mais desenvolvida e confiável. Muitos acham barato migrar agora para uma *cloud* na *web* ao invés de investir em seus próprios servidores ... Isso é um *desktop* por pessoa sem computador.
G. Gruman and E. Knorr	*Cloud* é toda sobre: SaaS ... computação utilitária, PaaS ... integração na internet ... plataformas comerciais.
P. McFedries	*Cloud computing*, onde residem não somente nossos dados mas também alguns de nossos *softwares*. Nós acessamos qualquer coisa não somente através de nossos PCs, mas também de outros dispositivos, como *smatphones*, PDAs, ... O megacomputador é habilitado pela virtualização e software como serviço ... Essa é a utilização do poder de computação pela massiva utilização de *data centers*.

Tabela 2.1: Definições de *Cloud Computing*. Adaptada de [29]

Por ser bastante completa e adotada na academia, a definição proposta pelo *National Institute of Standards and Technology* (NIST) foi a escolhida para ser utilizada neste trabalho. O NIST define *Cloud Computing* como sendo um modelo que provém acesso conveniente e sobre demanda a um conjunto de serviços computacionais configuráveis, que podem ser rapidamente provisionados e lançados com o mínimo de esforço e interação com o provedor [25].

Ainda segundo o NIST, o modelo de Computação em Nuvem é composto por cinco características essenciais, três modelos de serviço e quatro modelos de implantação [25].

2.1.1 Características Essenciais

Como já foi falado anteriormente, de acordo com o NIST, a definição de Computação em Nuvem é composta por cinco características essenciais [25]. São elas:

- **Alocação de recursos *on-demand*:** O consumidor pode configurar e provisionar recursos de acordo com sua necessidade sem ter que depender do provedor do serviço;
- **Amplo acesso a rede:** Os recursos estão disponíveis na rede e podem ser acessados através diversas plataformas;
- ***Pooling* de recursos:** Os recursos são agrupados para servir vários consumidores através do modelo *multi-tenant*. Os recursos físicos e virtuais são fornecidos dinamicamente de acordo com a demanda do cliente;
- **Elasticidade rápida:** Recursos podem ser alterados, configurados e liberados rapidamente, e em alguns casos isso pode ser feito de forma automática. Dessa forma o cliente pode escalar o sistema rapidamente de acordo com sua demanda;
- **Medição do serviço:** A utilização dos recursos é controlada e otimizada automaticamente. Dessa forma, tanto o cliente como o provedor podem monitorar e controlar os recursos utilizados, permitindo uma maior transparência no serviço.

2.1.2 Modelos de Serviço

Ainda de acordo com a definição do NIST, a Computação em Nuvem é composta por três modelos de serviço distintos [25]. São eles:

- **Software como um Serviço (SaaS):** Provém ao consumidor a capacidade de usar a aplicação do provedor localizada em uma infraestrutura de *Cloud*. As aplicações estão acessíveis de várias maneiras, como por exemplo um navegador ou a interface de um sistema;
- **Plataforma como um Serviço (PaaS):** Provém ao consumidor a capacidade de fazer *deploy* de suas aplicações na infraestrutura da Nuvem do provedor. O consumidor não controla recursos como rede, servidores, sistema operacionais e armazenamento, mas tem total controle sobre a aplicação e pode ter controle sobre as configurações do sistema;
- **Infraestrutura como um Serviço (IaaS):** Provém ao consumidor a capacidade de provisionar processamento, armazenamento, conexões com a rede e outros recursos computacionais fundamentais.

2.1.3 Modelos de Implantação

Por último, o NIST define *Cloud Computing* como tendo três modelos de implantação do serviço [25]. São eles:

- **Nuvem privada:** A infraestrutura é provisionada para ser usada por uma única organização que contém vários clientes;
- **Nuvem comunitária:** A infraestrutura é provisionada para ser usada por um grupo específico de clientes de organizações que compartilham recursos;
- **Nuvem pública:** A infraestrutura é provisionada para ser aberta para todo o público;
- **Nuvem híbrida:** A infraestrutura é uma composição de duas ou mais nuvens com infraestruturas distintas (privada, comunitária ou pública).

Este trabalho irá abordar, em sua maioria, tópicos relacionados a SaaS. Entretanto, conceitos de PaaS e IaaS também serão mencionados, mas não largamente estudados devido a *multi-tenancy* fazer parte do contexto de SaaS.

2.2 Software como um Serviço (SaaS)

A definição atual de Computação em Nuvem tem três modelos de negócio, como citado anteriormente, são eles: SaaS, PaaS e IaaS. Entretanto, em uma definição mais simples, pode-se dizer que Cloud Computing é um sinônimo de SaaS [34]. Isso se dá pelo rápido crescimento desse modelo de serviço e sua importância no mercado de TI nos dias de hoje.

Para exemplificar o quão grande tem sido o crescimento do uso de SaaS, pode-se analisar a Figura 2.2, que trás informações sobre o número de arquivos hospedados na Amazon S3 [15], um dos provedores de plataformas de *Software as a Service*. Além da plataforma da Amazon, existem outros provedores de SaaS como Google, Salesforce, Heroku e Digital Ocean.

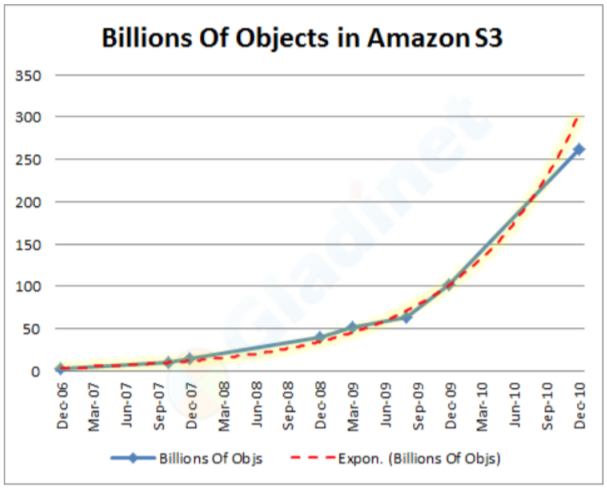

Figura 2.2: Crescimento do Amazon S3 [15]

Para melhor compreender o que é realmente Software como um Serviço, é necessário fazer uma análise mais profunda da sua definição. Para Chong e Carraro [3], SaaS pode ser definido da seguinte forma: “Software implantado como um serviço hospedado e acessado pela Internet.”.

Analisando a colocação de Chong e Carraro [3], essa definição não descreve nenhuma arquitetura, tecnologia, protocolo ou modelo de negócio específico. O que fica claro é onde o software deve estar implantado e a forma como ele deve ser hospedado e acessado, fazendo com que essa definição englobe vários tipos de aplicativos diferentes.

Em seu estudo, de Chong e Carraro [3] citam que existem três fatores que separam um aplicativo SaaS bem projetado de um mal projetado. São eles: escalabilidade, eficiência para vários inquilinos e configurabilidade. A partir desses três pilares, foi possível chegar a um modelo de maturidade com quatro níveis distintos, cada nível diferenciando-se do anterior pela adição de uma das carecterísticas citadas acima.

Os níveis de maturidade sugeridos por Chong e Carraro [5], como ilustrado na Figura 2.3, são os que seguem:

- **Nível I: Personalizado:** O primeiro nível se assemelha bastante ao modelo de serviço ASP, bastante utilizado na década de 1990, onde cada cliente tem sua própria instância personalizada do aplicativo e cada uma delas é completamente independente das outras instâncias do sistema.
- **Nível II: Configurável:** O segundo nível acrescenta ao sistema o fator de configurabilidade citado por Chong e Carraro [5]. Em termos de arquitetura, esse nível se assemelha bastante ao nível anterior e ao ASP. A diferença é que nesse nível, apesar de serem completamente independentes, todas as instâncias tem o mesmo código e toda a caracterização do sistema é feita a partir de configurações fornecidas pelo fornecedor, sem haver a necessidade de se alterar o código.
- **Nível III: Configurável e eficiente para vários inquilinos:** Já no terceiro nível o fator acrescentado é a eficiência para vários inquilinos. Nesse nível, existe uma única instância do sistema, mas que pode ser configurada por cada cliente, fornecendo uma experiência de usuário distinta para cada um deles. Além disso, políticas de autorização e segurança garantem a separação dos dados e não há qualquer indicação de que os clientes estejam compartilhando a mesma aplicação.
- **Nível IV: Escalonável, configurável e eficiente para vários inquilinos:** No quarto nível é acrescentado o fator da escalabilidade. Isso se dá devido a existência de uma *farm* de instâncias idênticas com balanceamento de cargas entre elas. Assim como no nível anterior os dados são mantidos separados e há a possibilidade

de configuração do sistema, fornecendo uma experiência do usuário única para cada cliente. Nesse nível de arquitetura, a quantidade de instâncias na *farm* pode ser aumentada e diminuída de acordo com a necessidade de fornecedor, facilitando assim a escalabilidade do sistema.

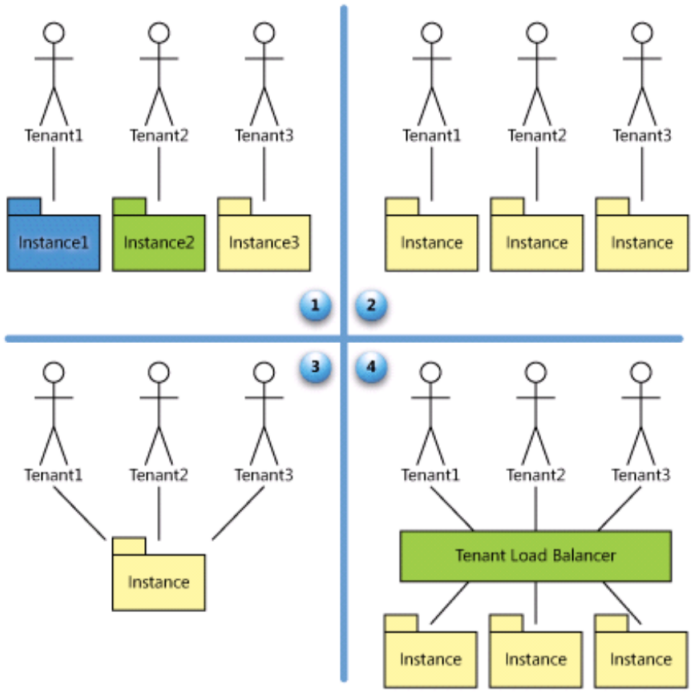

Figura 2.3: Níveis de maturidade SaaS. Chong e Carraro [3].

Apesar de existirem quatro níveis de maturidade de aplicações SaaS, não significa que o quarto, ou mais maduro, seja o foco para todos os sistemas. Deve-se analisar vários aspectos da aplicação, como o modelo de negócio, a arquitetura do sistema e o modelo operacional, para a partir daí definir qual o nível de maturidade mais adequado para o software. Como o foco desse trabalho é a arquitetura *Multi-Tenant*, o modelo abordado será o terceiro, onde existe só uma instância configurável do sistema servindo todos os clientes.

2.3 Django

Django é um *framework* de código aberto para desenvolvimento de aplicações Web, escrito em Python e que segue o modelo arquitetural MVC (*Model, View, Control*) [14].

O principal objetivo do *framework* é facilitar a criação de sistemas web complexos que tenham conexão com banco de dados. Além disso, o *framework* Django tem como ênfase o reuso de código, o desenvolvimento ágil e o princípio de não se repetir [14].

Django surgiu no ano de 2003, quando Adrian Holovaty e Simon Willison, até então desenvolvedores do jornal *Lawrence Journal-World*, começaram a utilizar Python para desenvolver aplicações Web. Em 2006, o *framework* foi nomeado em homenagem ao músico Django Reinhardt e disponibilizado publicamente através da licença BSD [14].

2.3.1 Arquitetura do *Framework*

A arquitetura do Django é baseada no modelo arquitetural MVC. Entretanto, algumas modificações foram feitas e esse novo modelo foi chamado de MVT (*Model, View, Template*), onde o *Model* faz a conexão com o banco de dados, a *View* controla toda a lógica do sistema e o *Template* cuida da parte de *layout* do sistema [14].

Além dos três componentes citados anteriormente, um sistema desenvolvido em Django conta com outros componentes de suma importância para o sistema, como pode ser visto na figura 2.4. São eles:

- ***URL Dispatcher:*** Esse componente é responsável por capturar a URL que está sendo requisitada e fazer um mapeamento dessa URL para a *View* adequada. Caso a página requisitada esteja na *cache* dos sistema, o próximos passos não serão executados [14];
- ***View:*** É uma função Python que executa a ação requisitada, que tipicamente envolve ler ou escrever no banco de dados, além de poder ter outras funcionalidades [14];
- ***Model:*** É responsável por definir os dados em Python e interagir com os mesmos. Apesar de geralmente trabalhar com bancos de dados relacionais, há a possibilidade de se usar outras abordagens como bancos não relacionais [14];
- ***Template***: Responsável pela geração das páginas HTML. Além disso, os *templates* do Django oferecem uma linguagem de marcação extra a fim de prover toda a lógica necessária para o desenvolvimento das telas do sistema [14].

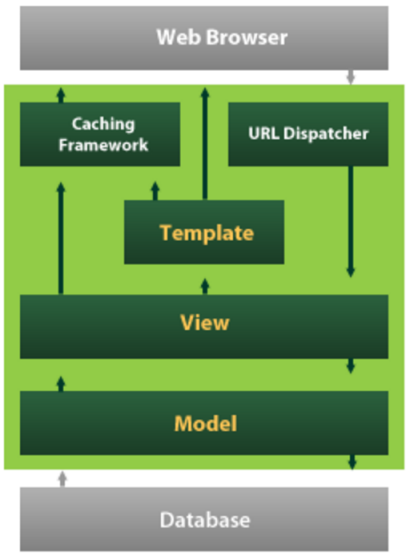

Figura 2.4: Modelo Arquitetural do Django [14].

O *framework* Django foi escolhido para ser utilizado nesse trabalho, pois além de prover várias funcionalidades que irão facilitar bastante o desenvolvimento da biblioteca proposta, é um dos *frameworks* mais utilizados na indústria hoje em dia [1]. De acordo com o *HotFrameworks* [1], Django é o quinto *framework* mais popular em ferramentas como Github e Stack Overflow. Além disso, Python vem crescendo bastante no mercado mundial. Como pode ser visto em [35], em 2001 *Pyhton* era a vigésima quinta linguagem de programação mais utilizada no mundo. Hoje, ela ocupa a quarta posição. Para exemplificar esse crescimento, vemos que projetos como Google, YouTube, Instagram, DropBox, Reddit, Pinterest e Spotify vem usando Python e Django na sua *stack* de desenvolvimento [12] [16].

2.4 Considerações Finais

Este capítulo apresentou os conceitos básicos que serão utilizados no trabalho. Inicialmente, foram apresentadas noções de *Cloud Computing*, SaaS, PaaS e IaaS. Com foco no modelo SaaS e na arquitetura multi-tenant, foco deste trabalho, é proposta a utilização do *framework* Django como ferramenta para facilitar o desenvolvimento da biblioteca Django Multi-Tenant.

Embora o que foi apresentado até aqui seja de grande importância, é necessário dar um

foco maior no conceito de *Multi-Tenancy*. O próximo capítulo conta com uma definição da arquitetura, apresenta técnicas de implementação e mostra vantagens e desvantagens da mesma, a fim de sintetizar os conhecimentos existentes na área.

Capítulo 3

Multi-Tenancy

Com o rápido crescimento da Computação em Nuvem e, mais especificamente, do modelo de serviço SaaS, novas tecnologias foram aparecendo e tomando lugar no mercado de software [6]. Um dos pontos chave de SaaS é a possibilidade de compartilhamento de recursos ao se oferecer software para diversos clientes, e para se beneficiar dessa característica, a arquitetura de aplicações SaaS deve ter como foco o compartilhamento da instância do software e do banco de dados [18].

Um dos métodos mais populares de se conseguir esse compartilhamento da aplicação e da instância do banco de dados é através da arquitetura *Multi-Tenant* [18]. Segundo Cordeiro e Lucena [7], *Multi-Tenancy* é uma arquitetura que possui uma única instância do aplicativo de *software* e que é utilizada por vários consumidores (*tenant*) . Além disso, é possível configurar a aplicação para atender a necessidade de cada *tenant*, de modo que cada um deles tenha a impressão de estar acessando um ambiente dedicado [2].

Segundo Bezemer e Zaidman [2], *tenants* são entidades organizacionais que alugam, por um tempo determinado, uma aplicação *Multi-Tenant*. Normalmente, um único *tenant*, ou inquilino, pode agrupar vários usuários, que são os *stakeholders* da organização.

Essas duas definições dadas por Bezemer e Zaidman focam no que eles consideram os aspectos chave de aplicações *Multi-Tenant* [2]. São eles:

- Possibilidade de compartilhamento do hardware, permitindo que haja uma redução dos custos [38];

- A oferta de um alto nível de configurabilidade, permitindo que cada *tenant* customize sua interface e o *workflow* na aplicação [24];
- Uma abordagem arquitetural onde os *tenants* façam uso de uma única instância da aplicação e do banco de dados [21].

As próximas seções apresentarão tópicos bastante importantes para o entendimento da arquitetura *Multi-Tenant*. São eles: Arquiteturas Correlatas, Características Chave, Vantagens X Desvantagens e Principais Desafios.

3.1 Arquiteturas Correlatas

Por ser uma definição bastante complexa e que abrange algumas definições que também fazem parte de outras arquiteturas, muitas vezes *Multi-Tenancy* é definida de forma equivocada. Esta seção tem como objetivo contrastar a arquitetura *Multi-Tenant* com outras duas definições do mundo SaaS: *Multi-User* e *Multi-Instance*.

3.1.1 Multi-Tenancy X Multi-User

Multi-Tenancy e *Multi-User* são dois conceitos que são comumente confundidos em suas definições [29]. Entretanto, existe uma diferença sutil, mas importante, entre as duas arquiteturas. Enquanto em uma aplicação *Multi-User* pode-se assumir que todos os usuários estão usando a mesma aplicação com opções de configuração limitadas, numa aplicação *Multi-Tenant* os usuários tem acesso a um vasto número de opções de configuração [2].

Essa diferença entre as duas arquiteturas gera um resultado que é uma das características mais importante de *Multi-Tenancy*. Embora os inquilinos estejam usando a mesma instância da aplicação, graças ao alto nível de configurabilidade, a interface e o fluxo do sistema podem ser diferentes para dois *tenants* [2]. Além disso, um outro argumento é que o *Service Level Agreement* (SLA) pode diferir para cada tenant. O que não é o caso em um sistema Multi-Usuário [23].

3.1.2 Multi-Tenancy X Multi-Instance

Outra abordagem que é bastante confundida com *Multi-Tenancy* é o conceito de *Multi-Instance*, onde cada *tenant* tem sua própria instância da aplicação e possivelmente do banco de dados [2]. Com os crescimento de *Cloud Computing* e a popularização de tecnologias de virtualização, *Multi-Instance* é o jeito mais simples de se oferecer aplicações parecidas com *Multi-Tenancy*. Basta apenas criar uma imagem do sistema pré-configurada e crias as instâncias a partir dessa imagem [29].

Bezemer e Zaidman definem a abordagem *Multi-Instance* como sendo a mais adequada para sistemas onde o número de inquilinos tende a ser baixo [2]. Isso se dá pois o custo de manutenção de um sistema *Multi-Instance* tende a crescer bastante com o aumento do número de inquilinos. Esse crescimento pode ser atribuído ao esforço necessário para se fazer *deploys* e corrigir *bugs* nas várias instâncias do sistema.

3.2 Características Chave

Como já foi falado anteriormente, em seu trabalho, Bezemer e Zaidman propõem as três características chave da arquitetura *Multi-Tenant* [2]: compartilhamento de hardware, configurabilidade e uma instância única da aplicação e do banco de dados. Essa seção tem como objetivo apresentar essas características de forma mais aprofundada, para que se possa entender melhor a importância das mesmas.

3.2.1 Compartilhamento de Hardware

No modelo tradicional de desenvolvimento de software, ou *single-tenant*, cada cliente tem seu próprio servidor, o que é bastante similar ao modelo ASP [23]. Entretanto, nesse tipo de arquitetura há a possibilidade de desperdício da capacidade de processamento do servidor, e ao se colocar diversos *tenants* no mesmo servidor, a sua utilização é melhorada automaticamente [22].

Multi-Tenancy pode ser obtido de várias formas e, dependendo da forma utilizada, a taxa de utilização dos recursos de hardware pode ser maximizada [2]. As formas de se obter

um sistema *Multi-Tenant* são as seguintes [5, 21]:

- Aplicação compartilhada, mas banco de dados separados;
- Aplicação compartilhada, banco de dados compartilhado, mas tabelas separadas;
- Aplicação compartilhada, banco de dados compartilhado e tabelas compartilhadas (*Multi-Tenancy* puro).

Em termos de escalabilidade, a última opção, ou *Multi-Tenancy* puro, leva vantagens sobre as outras por dois motivos. Primeiramente, os dois primeiros métodos tem graves problemas de performance quando se trata de um número grande de *tenants* [5, 38]. E finalmente, no modelo de *Multi-Tanancy* puro as tabelas são carregadas apenas uma vez, resultando numa maior quantidade de *tenants* no sistema [2]. Esta última abordagem foi escolhida para ser utilizada neste trabalho, por se mostrar mais eficiente.

3.2.2 Alto Nível de Configurabilidade

No modelo *Single-Tenant*, que é o primeiro nível de maturidade sugerido por Chong e Carraro (2006), cada cliente tem sua própria versão personalizada do sistema. Já no modelo *Multi-Tenant*, quarto nível de maturidade, todos os clientes usam a mesma instância do sistema, apesar do mesmo parecer ter uma versão dedicada para cada cliente [3].

Devido a esta característica de arquiteturas *Multi-Tenant*, um dos aspectos chave dessas aplicações é a possibilidade de configuração e customização da mesma de acordo com as necessidades do cliente [27]. Em sistemas *Single-Tenant* essa customização geralmente é feita através da criação de novas versões do sistema que atenda às necessidades do cliente. Já em sistemas *Multi-Tenant* isso não é possível e, por esse motivo, opções de configuração devem estar integradas no design do produto [24].

O poder de alto nível de configurabilidade de uma aplicação *multi-tenant* auxilia na identificação de diferentes utilizadores que utilizam o mesmo serviço. Como retratado por Correia [8] em seu trabalho, isto significa que isto é implementado por meio da configuração de metadados originais, para que assim, o código do núcleo da aplicação

possa ser mantido sem depender das necessidades dos utilizadores finais. Adicionalmente, este aspecto da arquitetura auxilia o fornecedor da *cloud* na alocação dos recursos.

Como relatado por Rodrigues [29] em sua dissertação, três áreas de pesquisa estão em alta sobre customização: aplicações web personalizadas para o usuário, variação nas linhas de produto de *software* e arquiteturas *multi-tenancy*. Recomenda-se que pesquisas futuras unam esforços para desenvolver estes tópicos.

3.2.3 Compartilhamento da Aplicação e do Banco de Dados

Uma aplicação *Single-Tenant* pode ter várias instâncias rodando e elas podem ser diferentes uma das outras por causa da customização. Em *Multi-Tenancy*, essas diferenças deixam de existir, já que a aplicação é modificada em tempo de execução de acordo com as configurações de cada *tenant*[2].

Isso implica que o número de instâncias de uma aplicação com arquitetura *Multi-Tenant* vai ser menor - idealmente uma, que pode ser replicada afim de facilitar a escalabilidade - quando comparado com uma arquitetura *Single-Instance*. Como consequência dessa diminuição da quantidade de instâncias, os *deploys* ficam bem mais simples e menos custosos, principalmente aqueles de atualizações [2].

Dados compartilhados e dados isolados não são dois conceitos binários. Ou seja, existe uma variação entre eles. Levando em conta esta variação, todas as abordagens devem ser avaliadas. A Figura 3.1 ilustra essa situação.

Figura 3.1: Abordagens para o gerenciamento de dados com *Multi-Tenancy* [5].

Banco de dados separados

A forma mais simples de se obter um modelo de dados isolado é realizando o armazenamento dos dados do inquilino em bancos de dados separados [5], como ilustra a

Figura 3.2. Os recursos de *hardware* são normalmente compartilhados entre os inquilinos, porém cada um deles possui suas próprias informações de forma isolada logicamente dos dados de outros inquilinos. A segurança do banco de dados garante que um inquilino não tenha acesso aos dados dos outros.

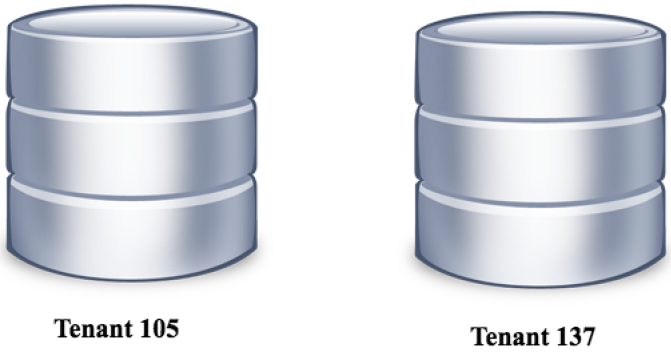

Figura 3.2: Abordagem utilizando bancos de dados separados. Adaptado de Chong [5].

A utilização de bancos de dados separados facilita a extensão do modelo de dados de uma aplicação. Inclusive, caso ocorra alguma falha, a recuperação dos dados de um inquilino se torna uma tarefa fácil de ser executada. No entanto, esta abordagem gera custos altos de manutenção [32].

Sendo assim, a separação dos bancos de dados dos inquilinos é uma boa alternativa. Mas deve-se observar os custos adicionais para a implantação. Por isso, para clientes que estejam dispostos a pagar mais por customização e segurança maiores, serão beneficiados com a escolha desta abordagem. Um exemplo de cliente que se beneficiaria muito com a utilização de bancos de dados separados seriam aqueles que trabalham com registros médicos, pois as informações precisam estar protegidas. Neste caso, esta abordagem torna-se essencial.

Banco de dados compartilhados, mas com Esquemas separados

A segunda abordagem é a que possui banco de dados compartilhados e esquemas separados, ou seja, cada inquilino possui seu conjunto de tabelas que são agrupadas em um esquema para cada inquilino. Esta abordagem está ilustrada na Figura 3.3.

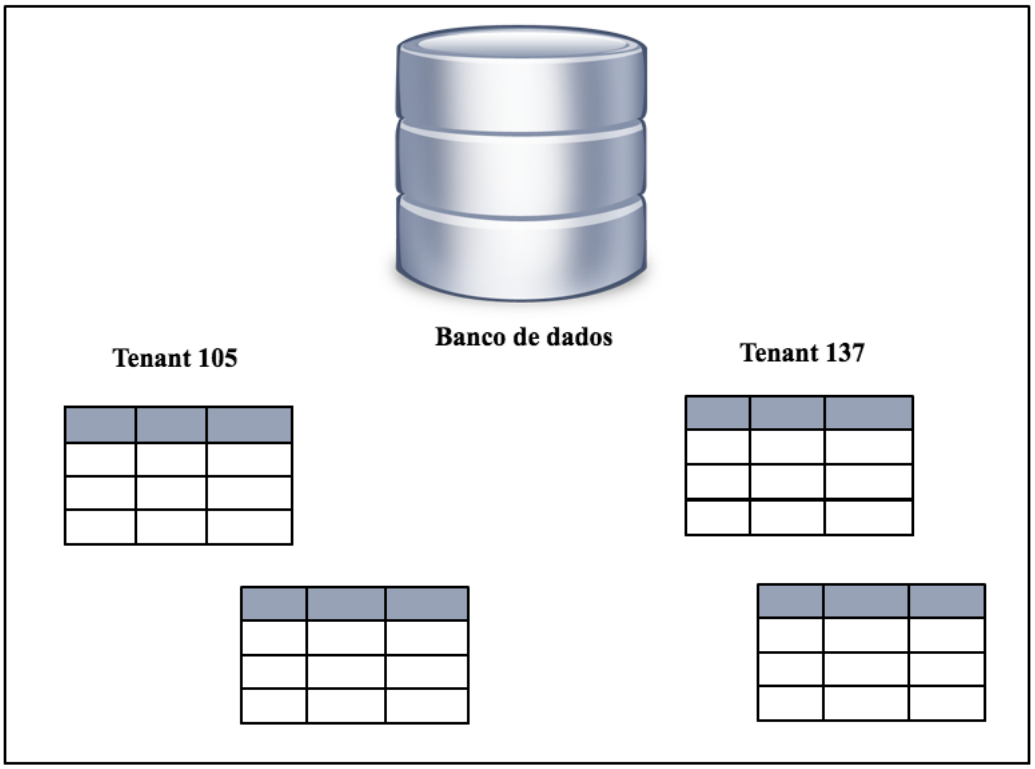

Figura 3.3: Abordagem utilizando um único banco de dados, com esquemas separados. Adaptado de Chong [5].

Esta abordagem possui um nível moderado de segurança e proteção dos dados dos inquilinos [5], não se equiparando a abordagem anterior neste quesito. Em contrapartida, uma desvantagem da utilização dessa abordagem é que por terem esquemas separados, caso ocorra alguma falha a correção será dificultada. Uma vantagem na utilização dessa abordagem é que ela reduz o custo de manutenção de gerenciamento de diferentes instâncias [32]. Quando os bancos de dados estão separados, a recuperação de dados é mais fácil devido a um backup recente resolver o problema. Já com um único banco de dados e esquemas separados, restaurar um banco de dados completo pode ocasionar sobreposição de dados dos inquilinos da base com o backup.

Banco de dados compartilhados e Esquema compartilhado

A terceira abordagem, ilustrada na Figura 3.4, possui um único banco de dados e um conjunto de tabelas para armazenar informações de vários inquilinos. Nesta, todos os inquilinos compartilham um mesmo conjunto de tabelas, possuindo um ID para cada

inquilino que associa cada registro com o inquilino apropriado. Porém, este compartilhamento pode ocasionar um esforço adicional no desenvolvimento de segurança, já que é preciso garantir que inquilinos nunca possam se associar a dados de outros inquilinos.

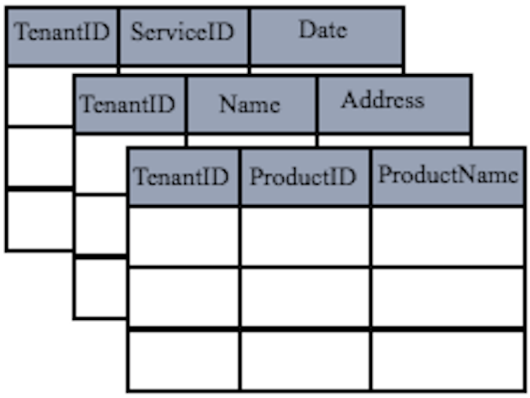

Figura 3.4: Abordagem utilizando um único conjunto de tabelas e um ID para associar cada inquilino a seus registros. Adaptado de Chong [5].

Ao comparar esta abordagem com as anteriormente citadas, pode-se constatar que esta é menos custosa em relação à hardware e backup [5], pois permite servir o maior número de inquilinos por servidor. A utilização dessa abordagem é recomendada quando a aplicação é capaz de servir a um alto número de inquilinos com um pequeno grupo de servidores, nos quais querem mudar de um modelo de isolamento para um compartilhado e diminuir seus custos.

Escolha da abordagem

De forma semelhante ao trabalho de Bezemer [2], a abordagem escolhida foi a de bancos de dados compartilhados e esquema compartilhado. Esta opção foi eleita a melhor porque as outras abordagens apresentam problemas. Além disso, existe uma biblioteca[1] *Python* que implementa a abordagem de bancos de dados compartilhados e esquemas separados (*tenant schemas*). A abordagem de bancos de dados separados não se mostra interessante em aplicações *multi-tenant* por quebrar seu princípio de aproveitamento de recursos.

[1] A biblioteca Django Tenant Schemas foi desenvolvida por Bernardo Pires e pode ser acessada através do endereço: https://github.com/bernardopires/django-tenant-schemas .

3.3 Principais Desafios

Infelizmente, além de trazer vários benefícios, arquiteturas *Multi-Tenant*, também trazem consigo alguns obstáculos a nível de implementação. Apesar de alguns desses desafios também estarem presente em arquiteturas *Single-Tenant*, em *Multi-Tenancy* eles aparecem de uma forma diferente [2]. Esta seção faz um levantamento desses desafios, focando principalmente nos aspectos de *Multi-Tenancy*.

3.3.1 Desempenho

Devido a arquiteturas *Multi-Tenant* compartilharem os mesmos recursos e possuirem uma maior utilização de hardware, é preciso se certificar de que todos os *tenants* podem consumir esses recursos conforme necessário. Se um *tenant* utiliza todos os recursos, o desempenho dos outros *tenants* poderá ser comprometido [29] [13]. Por outro lado, em situações com *Single-Tenant*, o comportamento de um *tenant* só afeta a ele mesmo. Em situações com instâncias virtuais, este problema é desenvolvido através da atribuição de uma quantidade igual de recursos para cada caso. Esta solução pode levar à utilização muito ineficiente de recursos e, portanto, indesejável em um sistema *Multi-Tenant* puro.

3.3.2 Escalabilidade

Todos os *tenants* compartilham a mesma aplicação e armazenamento de dados. Por conta disso, a escalabilidade é um problema maior quando comparado às aplicações *Single-Tenant*. Assumimos que um *tenant* não requer mais do que uma aplicação e servidor de banco de dados. Já em situações *Multi-Tenant*, esta hipótese pode não nos ajudar, pois tal limitação não existe ao colocar vários *tenants* em um servidor [2].

Os *tenants* de uma ampla variedade de países podem usar um aplicativo, podendo ter impacto sobre os requisitos de escalabilidade. Cada país possui sua própria legislação em relação a alocação de dados ou roteamento. Por fim, podem haver mais restrições, como a exigência de posicionar todos os dados para um *tenant* no mesmo servidor para acelerar as consultas ao banco de dados realizadas normalmente. Restrições como estas influenciam fortemente a maneira pela qual uma aplicação e seu armazenamento

de dados pode ser escalado.

3.3.3 Segurança

Segundo Rodrigues [29], um dos assuntos mais importantes ligados à arquitetura multi-tenant é o fator segurança. Este é um dos maiores desafios na aceitação de aplicações SaaS.

Embora o nível de segurança de um ambiente *Single-Tenant* precise ser alto, o risco de roubo de dados é relativamente pequeno. Em um ambiente multi-tenant, uma quebra de segurança pode resultar na exposição de dados a outros *tenants* possivelmente competitivos. Consequentemente, as questões de segurança tais como a proteção dos dados tornam-se muito importantes [2].

3.3.4 Manutenção

Segundo Bezemer e Zaidman [2], no ciclo evolutivo típico de um software, a manutenção é um dos desafios enfrentados. Como por exemplo, adaptar o sistema de software aos requisitos em constante mudança e à sua posterior implantação.

Embora esteja claro que o paradigma *Multi-Tenant* pode trazer muitos benefícios na implantação de sistemas, minimizando o número de aplicações e instâncias do banco de dados que precisam ser atualizados, adicionar *Multi-Tenancy* em um *software* certamente aumentará sua complexidade. Afetando, assim, o processo de manutenção. Bezemer e Zaidman [2] citam em seu estudo que mais pesquisas na área precisam ser realizadas para certificar que os benefícios de hardware e de implementação compensam o aumento no custo de manutenção.

3.3.5 Disponibilidade 24/7

Apresentar novos *tenants* ou se adaptar às mudanças nos requisitos de negócio dos *tenants* existentes traz consigo a necessidade de constante crescimento e evolução de um sistema *Multi-Tenant*. No entanto, as adaptações não devem interferir nos serviços

prestados aos outros *tenants* existentes. Isso leva a uma forte exigência de que *softwares Multi-Tenant* estejam disponíveis 24 horas, 7 dias por semana, já que o custo do tempo de inatividade por hora pode chegar a 4500 dólares dependendo do tipo de negócio [2].

3.4 Vantagens X Desvantagens

Nesta seção serão detalhadas as vantagens e desvantagens de se utilizar a arquitetura *Multi-Tenancy*.

3.4.1 Vantagens

A partir da leitura e análise de artigos da literatura, foram observadas vantagens e desvantagens da utilização da arquitetura *Multi-Tenancy*. Alguns estudiosos apontam vantagens na utilização da arquitetura *Multi-Tenancy*. Na pesquisa, as vantagens foram mais numerosas quando comparadas às desvantagens. Dentre as vantagens, estão:

- Realizar a atualização do software em uma única vez para todos os *tenants* [28];
- Economia nos custos relacionados ao gerenciamento de infra-estrutura [31];
- Economia nos custos relacionados à infra-estrutura de hardware [31];
- O provedor de serviço pode utilizar a mesma versão da aplicação, com o mesmo código base, para dispor serviços para várias organizações [36];
- Consumidores ganham acesso à capacidade de processamento elástico. Esta pode ser aumentada ou diminuída conforme a demanda, se houver a necessidade de realizar manutenção em servidores [36];
- Possibilita reutilizar as regras de negócio com pouca necessidade de adaptação [2];
- Usuários organizados em vários níveis de acordo com suas necessidades e melhor gerenciamento de recursos [17];
- Redução dos custos de manutenção e venda de software por parte do provedor de software [39].

3.4.2 Desvantagens

Dentre as desvantagens na utilização da arquitetura *Multi-Tenancy* encontradas na literatura, estão:

- Devem ser identificados os fatores limitantes e os gargalos nos recursos computacionais que são exigidos pelas várias instâncias, não sendo esta uma atividade trivial [20];
- O custo inicial de reestruturar as aplicações legadas para *Multi-Tenancy* são uma preocupação das empresas [2];
- Preocupação a respeito de a arquitetura *Multi-Tenancy* introduzir problemas de manutenção adicionais devido ao alto grau de customização dos novos sistemas [2].
- Dificuldades na otimização das formas de distribuição dos *tenants* entre os servidores, devido ao grande número de variáveis envolvidas [20].

3.5 Considerações Finais

Este capítulo apresentou a definição de *Multi-Tenancy*. Foram comparadas à arquitetura *multi-tenancy* algumas arquiteturas correlatas, como a *multi-user* e a *multi-instance*. Além disso, foram detalhadas as características chave da arquitetura, que incluem: compartilhamento de hardware, alto nível de configurabilidade e compartilhamento da aplicação e do banco de dados. Os principais desafios, performance, escalabilidade, segurança, manutenção e disponibilidade, foram explanados. Por fim, foram mostradas vantagens e desvantagens da utilização da arquitetura, baseando-se na revisão bibliográfica feita em artigos científicos no tópico.

Tomando como base toda a teoria apresentada neste trabalho, o próximo capítulo busca propor uma uma biblioteca *multi-tenant* para o framework Django.

Capítulo 4

Trabalhos Correlatos

Este capítulo tem por objetivo apresentar trabalhos relacionados ao tema desta pesquisa.

4.1 Bancos de Dados

No trabalho desenvolvido por Bezemer [2], ele cita que utilizou a abordagem de *Multi-Tenancy* puro quanto ao aspecto de compartilhamento de hardware. Segundo o estudioso, os outros métodos como aplicação compartilhada com bancos de dados separados e aplicação compartilhada com bancos de dados compartilhados mas tabelas separadas possuem graves problemas de desempenho, principalmente quando o número de *tenants* é grande. No trabalho desenvolvido por Rodrigues [29], ele assegura a importância do isolamento dos dados, já que todos os tenants utilizam a mesma instância do banco de dados e é preciso garantir que eles podem acessar apenas seus dados.

Por isso, no presente trabalho, de forma semelhante a [2], foi utilizada a abordagem de *Multi-tenancy* puro, ou seja, todos os recursos são compartilhados. O compartilhamento se mostra mais interessante, pois é menos custosa e serve um número maior de inquilinos por servidor.

4.2 Customização

Bezemer [2] em sua pesquisa cita que a configurabilidade em *Multi-tenancy* é algo simples e eficiente. Ele a implementou na área de *deploy* e atualizações. Já no trabalho desenvolvido por Mietzner et al. [26], utilizam-se descritores de variabilidade em aplicações *Multi-tenant*, como por exemplo mecanismos para customizar *templates*. São utilizadas ferramentas de configuração para transformar os artefatos contidos no modelo que possuem pontos de variabilidade aberta em artefatos configurados. Na dissertação de Rodrigues [29], é citado também que em aplicações *multi-tenant* é necessário que exista algum mecanismo para que a aplicação se ajuste às necessidades dos usuários.

No presente trabalho, o aspecto customização foi implementado quanto ao layout, o que permite o usuário alterá-lo de forma eficiente. Os trabalhos de Bezemer [2] e Mietzner et al. [26] se assemelham ao presente trabalho neste aspecto, já que possuem o mecanismo de customizar alguma característica da aplicação.

4.3 Autenticação

Como mencionado por Rodrigues [29] em sua dissertação, devido a uma aplicação *multi-tenant* ter apenas uma instância da aplicação e do banco de dados, todos os *tenants* usam o mesmo ambiente físico. Com o intuito de oferecer a customização do ambiente e garantir que os *tenants* possam apenas acessar os seus próprios dados, eles devem ser autenticados. Geralmente, o controle de acesso em ambientes *multi-tenant* é uma funcionalidade fundamental para proteger os dados de cada inquilino dos demais.

Para H. Kim e D. Kim [19], o mecanismo de acesso mutualmente exclusivo permite que apenas que o *tenant* atual tenha acesso, bloqueando os demais. Este mecanismo faz com que aplicações *multi-tenant* compartilhem recursos de forma inteligente e controlada com políticas específicas. Em sua abordagem SOA, Mietzner et al. [26] explica uma maneira em que um *tenant* pode substituir ou estender pedaços de código, sem influenciar os outros *tenants*.

Dessa forma, no presente trabalho foi decidido utilizar mecanismos similares aos executados por de H. Kim e D. Kim [19] e Mietzner et al. [26] para realizar a autenticação em

aplicações *multi-tenant*.

4.4 Consideraçnoes Finais

Após analisar os trabalhos realizados por autores da literatura dos principais tópicos relacionados à biblioteca desenvolvida, foi visto que existem muitos projetos desenvolvidos que utilizam a mesma abordagem de bancos de dados, customização e autenticação utilizada pela biblioteca proposta neste trabalho. Porém, grande parte dos trabalhos levam em consideração apenas um fator, como por exemplo customização. O diferencial deste trabalho é que ele proporciona uma visão multitarefada na abordagem à arquitetura multi-tenant.

Capítulo 5

A Biblioteca: Django Multi-Tenant

A biblioteca[1] proposta neste trabalho tem como objetivo principal auxiliar desenvolvedores do *framework Django* a desenvolver sistemas *multi-tenant*. Para isso, foram implementadas diversas funcionalidades que atacam várias características da arquitetura. Dentre elas, estão:

- Arquitetura de dados compartilhada, incluindo facilidade de filtragem de dados de acordo com o *tenant*;
- Customização de temas para o sistema, de acordo com o *tenant*;
- Controle de acesso de usuário de acordo com o *tenant*;
- Funcionalidades auxiliares.

Nos tópicos seguintes cada funcionalidade será detalhada.

Por esta ser uma biblioteca que facilita a utilização da arquitetura multi-tenant por aplicações Django, os níveis de maturidades III e IV sugeridos por Chong e Carraro [4] podem ser utilizados, como detalhados no Capítulo 2. Fica a cargo do desenvolvedor da aplicação que venha a utilizar a biblioteca Django Multi-Tenant definir os níveis de maturidade que podem ser implementados. No exemplo da biblioteca, mostrado no Capítulo 6, foi utilizado o nível III de maturidade.

[1] Link para a biblioteca Django Multi-Tenant no GitHub: https://github.com/arineto/django-multi-tenant

5.1 Arquitetura de Dados

Foram analisadas diversas abordagens de organização de dados para arquitetura *multi-tenant*, como visto na seção 3.2.3. Após essa análise, a que se mostrou mais interessante foi a de bancos de dados compartilhados com esquema compartilhado.

Foram criados dois modelos do *Django* com o intuito de facilitar esta abordagem. São eles: *tenant* e *tenant model*, como ilustrado nas Figuras 5.1 e 5.2.

O modelo *Tenant* descrito na Figura 5.1 representa os *tenants* do sistema. Este modelo é composto pelo nome do *tenant*, um *slug*, que identifica unicamente o *tenant*, a data máxima de atividade do *tenant* no sistema, uma chave estrangeira para o tema utilizado por aquele *tenant* e uma associação com os usuários que tem acesso ao sistema por esse *tenant*.

```
18  class Tenant(models.Model):
19      """
20      This model represents a tenant instance.
21      """
22
23      name = models.CharField(max_length=100, unique=True)
24      slug = models.SlugField(max_length=200, unique=True)
25      active_until = models.DateField(null=True, blank=True)
26      theme = models.ForeignKey(Theme, null=True, blank=True)
27      users = models.ManyToManyField(User)
```

Figura 5.1: Modelo *Tenant*.

O modelo *TenantModel* apresentado na Figura 5.2 é um modelo abstrato que deve ser herdado por modelos que irão usufruir do esquema de dados compartilhado. Esse modelo abstrato adiciona ao modelo que o herda uma chave estrangeira para o tenant.

```
44  class TenantModel(models.Model):
45      """
46      This is a abstract model to provide a shared database aproach to the
47      project tables.
48      """
49
50      tenant = models.ForeignKey(Tenant, related_name='tenant')
51      objects = TenantModelManager()
52
53      class Meta:
54          abstract = True
```

Figura 5.2: Modelo *TenantModel*.

Além disso, também são adicionadas funções de *managers* que facilitam a filtragem de dados por *tenant*, como ilustrado na Figura 5.3.

```
4  class TenantModelManager(models.Manager):
5      """
6      This manager makes it easy to filter by tenant
7      """
8
9      def by_tenant(self, tenant):
10         return self.filter(tenant=tenant)
11
12     def by_tenants(self, tenants):
13         return self.filter(tenant__in=tenants)
```

Figura 5.3: *TenantModelManager*.

A abordagem de dados sendo implementada deste modo, facilita tanto a utilização de uma chave estrangeira para o *tenant* como a filtragem por *tenant*.

5.2 Customização

Como discutido na seção 5.2, a customização é um aspecto chave na arquitetura *multi-tenant*. A biblioteca proposta neste trabalho possibilita ao *tenant* customizar o tema da interface do sistema através da escolha de um dos temas disponibilizados pelo *software*.

A biblioteca fornece um modelo de tema, como pode ser visto na Figura 5.4. Além disso, o modelo do *tenant* tem uma chave estrangeira para os temas, como visto na Figura 5.1, possibilitando uma experiência de usuário diferente para cada *tenant* através da escolha do tema.

```
6  class Theme(models.Model):
7      """
8      This model stores all the available themes.
9      """
10
11     name = models.CharField(max_length=100, unique=True)
12     stylesheets = models.FileField(upload_to='themes/')
```

Figura 5.4: *Theme*.

A disponibilização de temas é feita a partir de novas entradas na tabela *Theme*. Para isso, é necessário escolher um nome único para o tema e anexar um arquivo .css que realiza as alterações necessárias.

5.3 Controle de Acesso

Como já foi visto nas seções 3.3.3 e 4.3, um mecanismo de controle de acesso baseado na literatura que se mostrou interessante foi o controle de acesso mutualmente exclusivo. Este permite que apenas o *tenant* atual tenha acesso ao sistema, bloqueando qualquer outro que tente o acesso.

Como Django já provém um sistema de autenticação completo e já existem bibliotecas focadas em autenticação através de OAuth, o uso deles fica a cargo do desenvolvedor. Assim, foram implementadas duas funcionalidades para complementar estes sistemas de autenticação: uma função que verifica se um dado usuário tem acesso ao *tenant* desejado e um *Mixin* que assegura que o usuário tem acesso à página desejada.

Uma função chamada *belongs_to_tenant* foi implementada para verificar se um usuário tem acesso ao *tenant* desejado, como pode ser visualizada na Figura 5.5.

```
5  def belongs_to_tenant(username, tenant):
6      """
7      This function checks if the user belongs to the given Tenant.
8      """
9      user_exists = tenant.users.filter(username=username).exists()
10     if not user_exists:
11         raise IncorrectTenantException()
```

Figura 5.5: *Classe Belongs to tenant*.

Uma outra funcionalidade implementada na biblioteca é um *Mixin*, apresentado na Figura 5.6. Ele é utilizado pelas *views* a fim de permitir que apenas usuários logados e pertencentes ao *tenant* tenham acesso ao conteúdo protegido. São realizadas três verificações:

- Se uma página está sendo acessada através de um *tenant*;
- Se o usuário está autenticado;
- Se o usuário autenticado tem acesso ao *tenant*.

```python
class TenantRequiredMixin(AccessMixin):
    """
    View mixin which verifies that there is a tenant in the request, that the
    user is authenticated and that the user belongs to the tenant.

    NOTE:
        This should be the left-most mixin of a view, except when
        combined with Django-Braces CsrfExemptMixin - which in that case
        should be the left-most mixin.
    """
    def dispatch(self, request, *args, **kwargs):
        if not request.tenant:
            return self.handle_no_permission(request)

        if not request.user.is_authenticated():
            return self.handle_no_permission(request)

        try:
            belongs_to_tenant(request.user.username, request.tenant)
        except IncorrectTenantException:
            return self.handle_no_permission(request)

        return super(TenantRequiredMixin, self).dispatch(
            request, *args, **kwargs)
```

Figura 5.6: *TenantRequiredMixin*.

5.4 Funcionalidades Auxiliares

Também foram desenvolvidas funcionalidades auxiliares que colocam informações na requisição e na geração do contexto. São elas:

5.4.1 *Middlewares*

Django Middlewares são classes que tem como função adicionar parâmetros à requisição. Como pode ser visto na Figura 5.7, a biblioteca conta com dois *middlewares*.

O primeiro, *SubdomainMiddleware* é responsável por capturar o subdomínio na URL e adicioná-lo ao *request*. Já o segundo, *TenantMiddleware*, adiciona o *tenant*, que é reconhecido através do subdomínio, ao *request*.

```python
class SubdomainMiddleware(object):
    """
    Adds a ``subdomain`` attribute to the ``request`` parameter.
    """

    def process_request(self, request):
        domain = request.META.get('HTTP_HOST')
        pieces = domain.split('.')
        try:
            request.subdomain = pieces[-2]
        except IndexError:
            request.subdomain = None


class TenantMiddleware(object):

    def process_request(self, request):
        """
        Adds a ``tenant`` attribute to the ``request`` parameter.

        It will try to find a TENANT_MODEL on the settings, if it can't find
        the specified model, it will user the multi_tenant.Tenant model.
        """
        model_path = getattr(settings, 'TENANT_MODEL', 'multi_tenant.Tenant')
        tenant_model = apps.get_model(*tuple(model_path.split('.')))
        try:
            tenant = tenant_model.objects.get(slug=request.subdomain)
        except tenant_model.DoesNotExist:
            tenant = None
        request.tenant = tenant
```

Figura 5.7: *Middlewares*.

5.4.2 *Context Processors*

Context Processors são funções que tem como objetivo adicionar informação ao contexto a ser passado para o *template* html. Como pode ser visto na Figura 5.8, foram implementados três funções deste tipo.

A primeira, *subdomain*, adiciona o subdomínio no contexto. A segunda, *tenant*, adiciona o *tenant* no contexto. Por fim, a terceira *theme*, adiciona o tema escolhido pelo *tenant* no contexto.

```python
def subdomain(request):
    return {
        'subdomain': request.subdomain
    }


def tenant(request):
    return {
        'tenant': request.tenant
    }


def theme(request):
    return {
        'theme': request.tenant.theme if request.tenant else None
    }
```

Figura 5.8: *Context Processors*.

5.5 Considerações Finais

A implementação da biblioteca busca facilitar o desenvolvimento de aplicações *multi-tenant* em *Django*. Além disso, a biblioteca deverá ser disponibilizada de forma *open source* para que outros desenvolvedores possam utilizá-la. Um exemplo de seu uso será demonstrado no Capítulo 6.

Capítulo 6

Um exemplo de uso da biblioteca

Neste capítulo será demonstrado um exemplo de uso da biblioteca[1] proposta neste trabalho. Este exemplo abrange o funcionamento de cada uma das funcionalidade implementadas. São elas:

- Nome do subdomínio e Tenant injetados no objeto Request e no Contexto;
- Itens filtrados por Tenant sem restrição de acesso;
- Itens filtrados por Tenant com restrição de acesso;
- Função de login com checagem de Tenant.
- Customização de tema por Tenant;

Para instalar a biblioteca, é necessário ter o instalador de bibliotecas Python (PIP) e executar o comando:

```
pip install git+https://github.com/arineto/django-multi-tenant.git
```

Para maiores detalhes sobre a utilização da ferramenta, acessar a página no GitHub[2].

[1] Link para o exemplo da biblioteca no GitHub: https://github.com/arineto/django-multi-tenant-example
[2] Link para a biblioteca Django Multi-Tenant: https://github.com/arineto/django-multi-tenant

6.1 Nome do subdomínio e Tenant injetados no objeto Request e no Contexto

As funcionalidades envolvidas nos *middlewares* e nos *context processors* são responsáveis por capturar o *tenant* através do subdomínio e adicioná-lo ao contexto. Como pode ser visto na Figura 6.1, ao acessar o sistema sem o subdomínio, nenhuma informação de *tenant* é adicionada ao contexto. Entretanto, ao acessar com o subdomínio, como pode ser visto na Figura 6.2, o subdomínio é capturado e as informações do *tenant* são disponibilizadas automaticamente.

Figura 6.1: Acesso sem *tenant*.

Figura 6.2: Acesso com *tenant*.

6.2 Itens filtrados por Tenant sem restrição de acesso

Como foi visto na seção 5.1, modelos que terão o esquema de dados compartilhado devem herdar da classe *TenantModel*. No exemplo, a classe Item herda da classe *TenantModel*, como ilustrado na Figura 6.3. Dessa forma, todo item está ligado a um *tenant* e a filtragem é facilitada.

```
1  from django.db import models
2  from multi_tenant.models import TenantModel
3
4
5  class Item(TenantModel):
6
7      name = models.CharField(max_length=10)
8      code = models.IntegerField()
9      created_at = models.DateTimeField(auto_now_add=True)
```

Figura 6.3: *ItemModel*.

Na figura 6.4, acessamos a listagem de itens sem o subdomínio. Por conta disso, nenhum item é mostrado já que não há nenhum *tenant*.

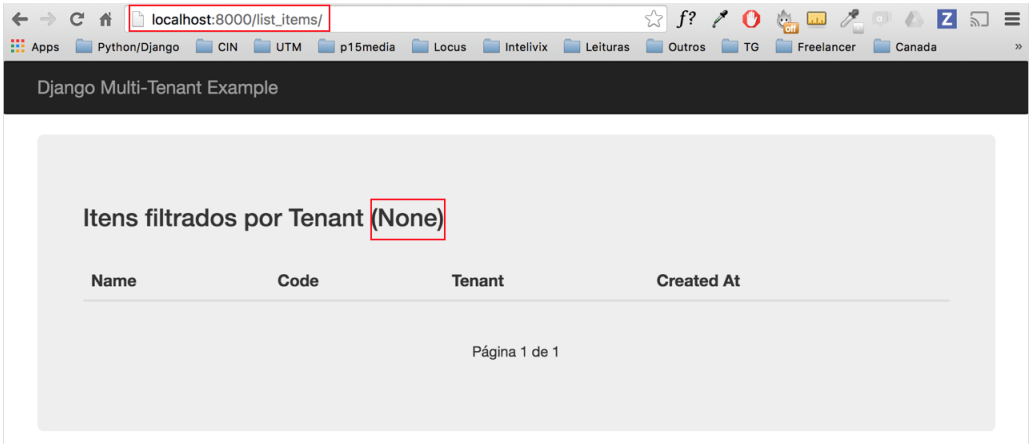

Figura 6.4: Listagem sem subdomínio.

Já nas Figuas 6.5 e 6.6, a listagem de itens é acessada através de subdomínios. Filtrando, dessa forma, os itens associados a cada *tenant*.

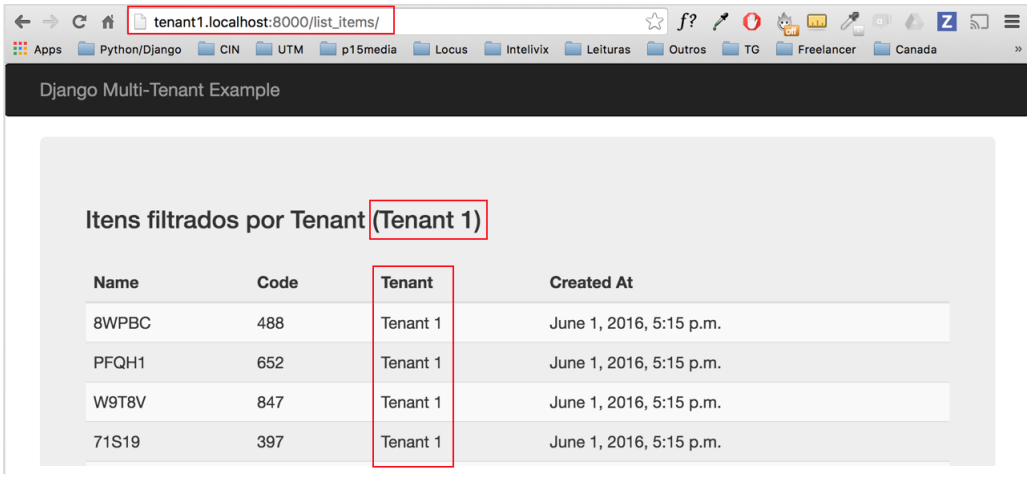

Figura 6.5: Listagem Tenant 1.

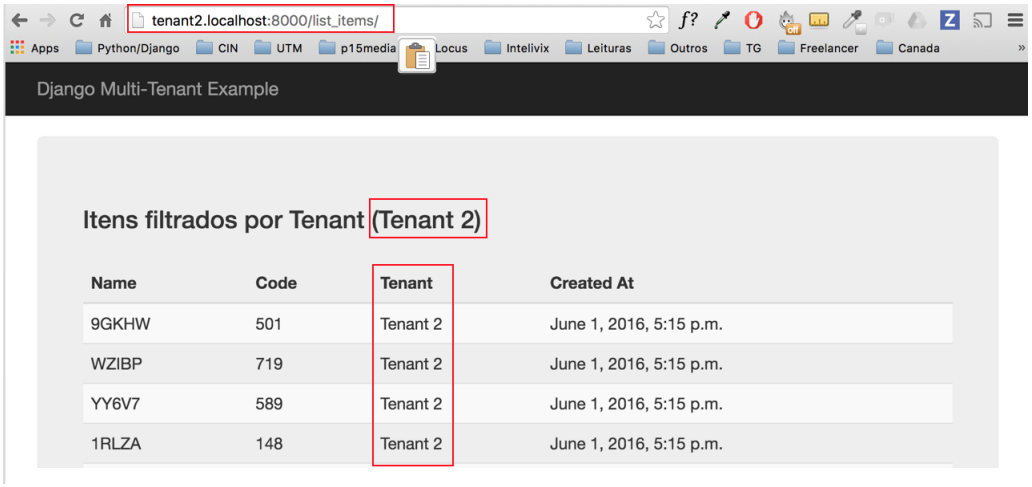

Figura 6.6: Listagem Tenant 2.

6.3 Itens filtrados por Tenant com restrição de acesso

Como já foi dito na seção 5.3, a classe *TenantRequiredMixin* adiciona algumas restrições de acesso às páginas que o utilizam. As permissões de acesso por *tenant* estão definidas segundo a Tabela 6.1.

Usuário	**Tenant**
user1	Tenant 1
user2	Tenant 1 e Tenant 2
user3	Tenant 2

Tabela 6.1: Tabela de controle de acesso.

Ao tentar acessar uma página que tenha o *TenantRequiredMixin* sem estar logado com o usuário correto, o acesso não será permitido e o usuário será redirecionado para uma página de erro, como pode ser visto na Figura 6.7. Entretanto, ao acessar a mesma página com um usuário que tenha permissão o acesso será permitido, como ilustrado na Figura 6.8.

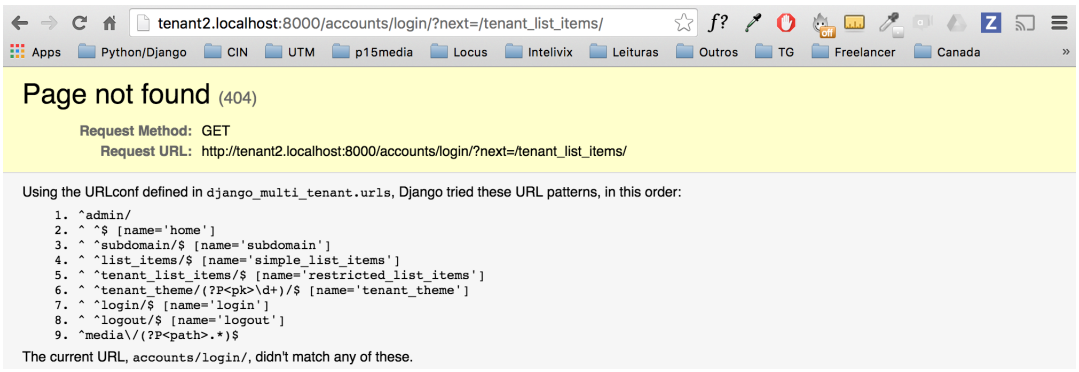

Figura 6.7: Acesso negado.

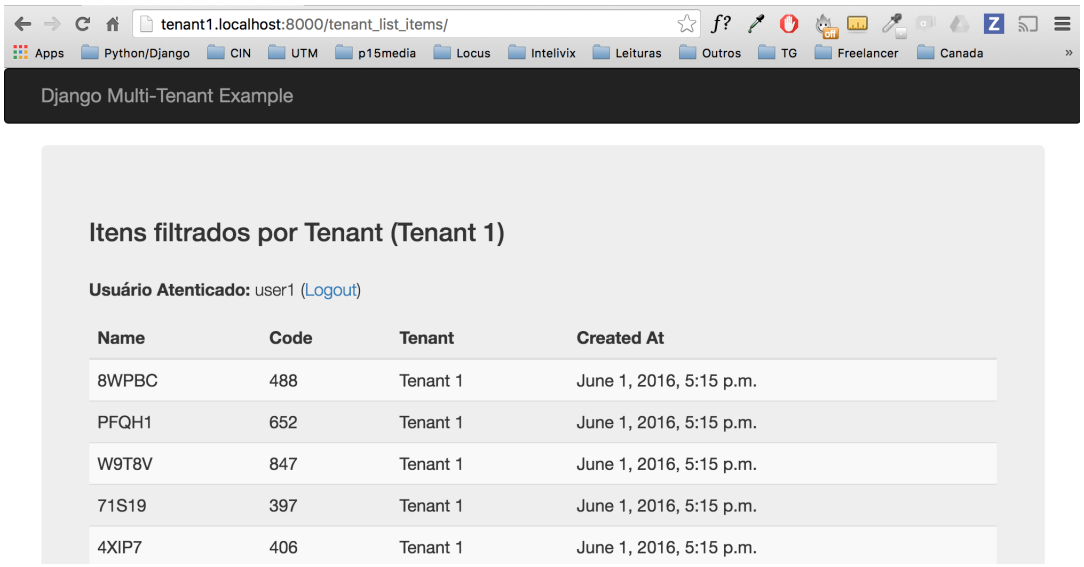

Figura 6.8: Acesso permitido.

6.4 Função de login com checagem de Tenant

Ainda falando sobre controle de acesso, a função *belongs_to_tenant* foi encorporada na autenticação de usuário. Dessa forma, os usuários só conseguem acessar *tenants* nos quais eles tem permissão.

No exemplo, podemos ver que ao autenticar o *user3* para o *Tenant 1* o acesso foi negado, como ilustrado na Figura 6.9. Já ao tentar autenticar o *user1* para *Tenant 1* o acesso foi permitido, como apresentado na Figura 6.10.

O controle de acesso mencionado nesta seção está de acordo com a Tabela 6.1 mostrada anteriormente.

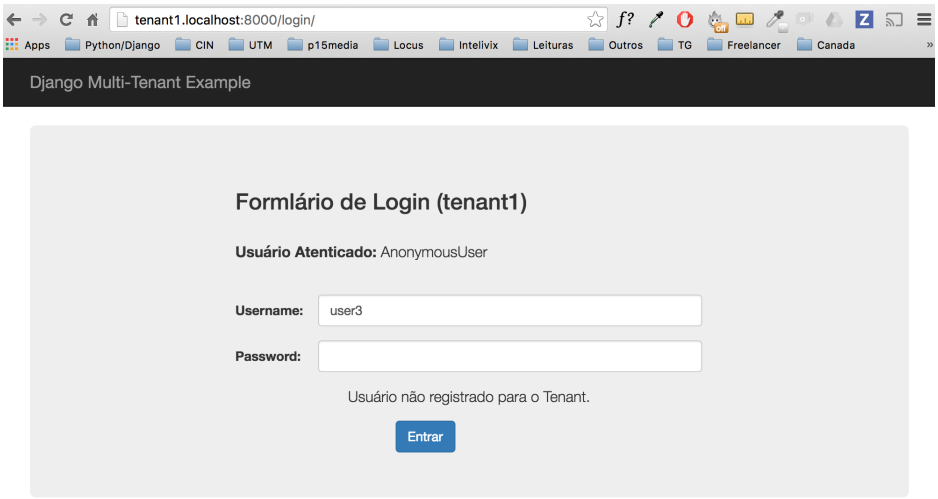

Figura 6.9: Acesso negado.

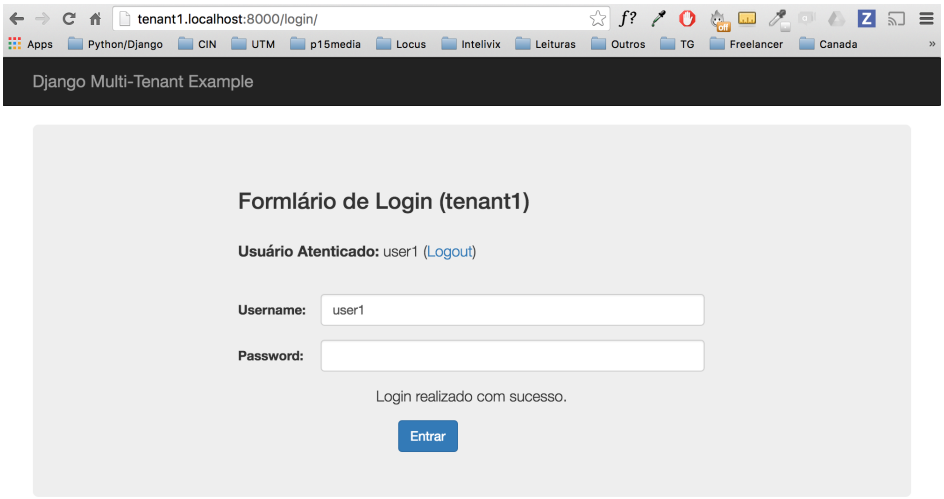

Figura 6.10: Acesso permitido.

6.5 Customização de tema por Tenant

Como mencionado na seção 5.2, a biblioteca disponibiliza customização de tema por *tenant*. Como podemos ver na Figura 6.11, o exemplo já possui um tema *default* para todos os *tenants*. Entretanto, é possível alterar este tema através de uma lista de temas disponíveis, como pode ser visto nas Figuras 6.12 e 6.13.

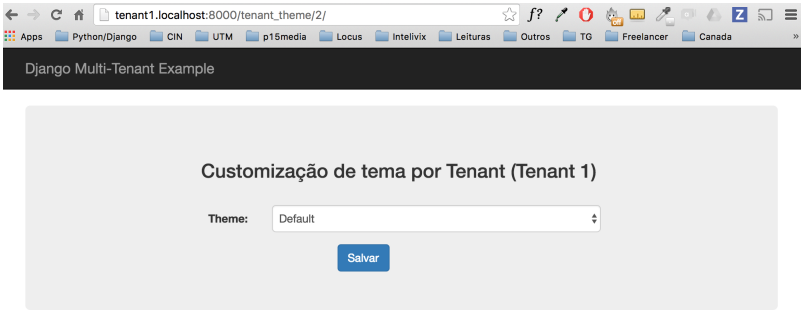

Figura 6.11: Tema Default do Tenant 1.

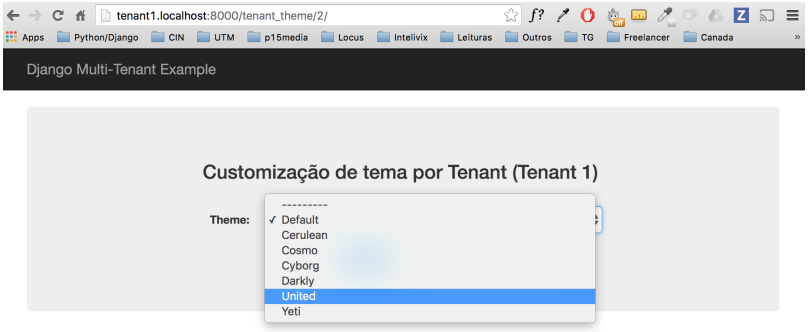

Figura 6.12: Listagem com as opções de tema para customização do *tenant*.

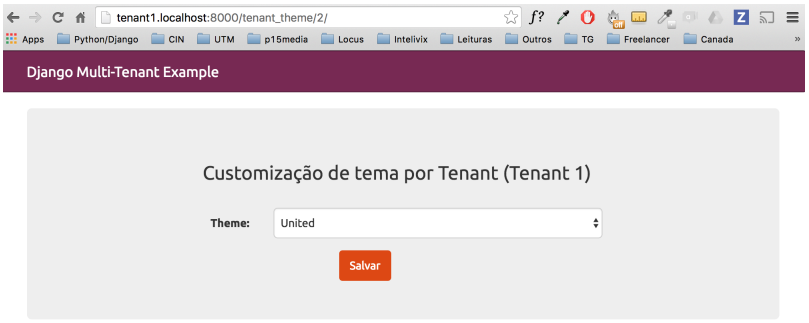

Figura 6.13: Tema customizado do Tenant 1.

Capítulo 7

Conclusão e trabalhos futuros

Devido o surgimento da Computação em Nuvem, o acesso a recursos computacionais tem se tornado cada vez mais facilitado. *Multi-tenancy*, por ser uma abordagem organizacional do modelo SaaS, possui diversas características benéficas para projetos no geral, quando utilizada. Este trabalho mostrou os benefícios provindos da utilização de uma biblioteca *multi-tenant*, como por exemplo uma maior utilização de recursos, facilidade na manutenção da aplicação e custos reduzidos.

O objetivo principal desta pesquisa foi propor uma biblioteca *open source* para transformar projetos *Django* em projetos *multi-tenant*, com o intuito de facilitar seu desenvolvimento. Assim, os desenvolvedores podem focar seus esforços na regra de negócio e não com a arquitetura, já implementada pela biblioteca.

A biblioteca desenvolvida neste trabalho inclui funcionalidades importantes como arquitetura de dados compartilhada, customização de temas e controle de acesso. Partindo destas funcionalidades, a biblioteca viabiliza o trabalho de desenvolvedores *Django* ao implementar *software* sem gastar tempo e esforços extras para implementar e garantir a funcionalidade da arquitetura *multi-tenant*.

Trabalhos futuros incluem:

- Implementar a biblioteca proposta em outros *frameworks web* como *flask* e *rails* com o intuito de difundir ainda mais a arquitetura *multi-tenant*;
- Avaliar empiricamente a importância desta biblioteca *multi-tenant* para projetos

Django;

- Adicionar mais ferramentas de customização na bilbioteca, como por exemplo customização de fluxo, relatório e formulários;
- Disponibilizar na biblioteca outras arquiteturas de dados como instâncias de bancos de dados compartilhados com schemas separados;
- Disponibilizar a biblioteca de forma open source.

Referências Bibliográficas

[1] Find your new favorite web framework. `http://hotframeworks.com/`. Accessed: 2016-04-19.

[2] Cor-Paul Bezemer and Andy Zaidman. Multi-tenant saas applications: maintenance dream or nightmare? In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, pages 88–92. ACM, 2010.

[3] Fred Chong and Gianpaolo Carraro. Architecture strategies for catching the long tail. *MSDN Library, Microsoft Corporation*, pages 9–10, 2006.

[4] Frederick Chong and Gianpaolo Carraro. Software como serviço (saas): uma perspectiva corporativa. *Microsoft MSDN*, 2007.

[5] Frederick Chong, Gianpaolo Carraro, and Roger Wolter. Multi-tenant data architecture. *MSDN Library, Microsoft Corporation*, pages 14–30, 2006.

[6] Louis Columbus. Cloud computing and enterprise software forecast update, 2012. *Forbes. com LLC*, 2012.

[7] Matheus de A Cordeiro and Emanuell FH de Lucena. Framework de persistência para implementação de aplicações multi-tenant em java. *Simpósio Brasileiro de Sistemas de Informação*, pages 436–437, 2014.

[8] Edite Correia. Cloud computing: o modelo saas–problemas e desafios de segurança e privacidade. *Universidade do Porto*, pages 4–13, 2015.

[9] Anderson Fernando Custódio and Edson A Oliveira Junior. Um exemplo de aplicação multi-tenancy com hibernate shards. *Trabalho de Conclusão de Curso*, 2012.

[10] Thomas Erl, Ricardo Puttini, and Zaigham Mahmood. *Cloud computing: concepts, technology, & architecture*. Pearson Education, 2013.

[11] John Feather. Architects of the information society. thirty-five years of the laboratory for computer science at mit. *Publishing Research Quarterly*, 16(2):95–95, 2000.

[12] Mikko Flagan. 25 of the most popular python and django websites. `https://www.shoop.io/en/blog/25-of-the-most-popular-python-and-django-websites/`. Accessed: 2016-04-19.

[13] Chang Jie Guo, Wei Sun, Ying Huang, Zhi Hu Wang, and Bo Gao. A framework for native multi-tenancy application development and management. In *The 9th IEEE International Conference on E-Commerce Technology and The 4th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services (CEC-EEE 2007)*, pages 551–558. IEEE, 2007.

[14] A HOLOVATY and J KAPLAN-MOSS. The django book: Version 2.0. *The Django Book*, 16, 2009.

[15] J Huang. Amazon s3's amazing growth. *Cloud Computing Journal*, 2011.

[16] Alex Ivanovs. 14 popular sites powered by django web framework. `http://codecondo.com/popular-websites-django/`. Accessed: 2016-04-19.

[17] Amarnath Jasti, Payal Shah, Rajeev Nagaraj, and Ravi Pendse. Security in multi-tenancy cloud. In *Security Technology (ICCST), 2010 IEEE International Carnahan Conference on*, pages 35–41. IEEE, 2010.

[18] Jaap Kabbedijk, Cor-Paul Bezemer, Slinger Jansen, and Andy Zaidman. Defining multi-tenancy: A systematic mapping study on the academic and the industrial perspective. *Journal of Systems and Software*, 100:139–148, 2015.

[19] Seong Hoon Kim and Daeyoung Kim. Enabling multi-tenancy via middleware-level virtualization with organization management in the cloud of things. *IEEE Transactions on Services Computing*, 8(6):971–984, 2015.

[20] Thomas Kwok and Ajay Mohindra. Resource calculations with constraints, and placement of tenants and instances for multi-tenant saas applications. In *Service-Oriented Computing–ICSOC 2008*, pages 633–648. Springer, 2008.

[21] Thomas Kwok, Thao Nguyen, and Linh Lam. A software as a service with multi-tenancy support for an electronic contract management application. In *Services Computing, 2008. SCC'08. IEEE International Conference on*, volume 2, pages 179–186. IEEE, 2008.

[22] Xin Hui Li, Tian Cheng Liu, Ying Li, and Ying Chen. Spin: Service performance isolation infrastructure in multi-tenancy environment. In *Service-Oriented Computing–ICSOC 2008*, pages 649–663. Springer, 2008.

[23] Hailue Lin, Kai Sun, Shuan Zhao, and Yanbo Han. Feedback-control-based performance regulation for multi-tenant applications. In *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, pages 134–141. IEEE, 2009.

[24] MIETE Mcsi. Configurability in saas (software as a service) applications. In *Proc. 2nd Annual Conference on India Software Engineering Conference*, pages 19–26, 2009.

[25] Peter Mell and Tim Grance. The nist definition of cloud computing. *Computer Security Division; Information Technology Laboratory; National Institute of Standards and Technology Gaithersburg*, 2011.

[26] Ralph Mietzner, Frank Leymann, and Mike P Papazoglou. Defining composite configurable saas application packages using sca, variability descriptors and multi-tenancy patterns. In *Internet and Web Applications and Services, 2008. ICIW'08. Third International Conference on*, pages 156–161. IEEE, 2008.

[27] Ralph Mietzner, Andreas Metzger, Frank Leymann, and Klaus Pohl. Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications. In *Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, pages 18–25. IEEE Computer Society, 2009.

[28] Ralph Mietzner, Tobias Unger, Robert Titze, and Frank Leymann. Combining different multi-tenancy patterns in service-oriented applications. In *Enterprise Distributed Object Computing Conference, 2009. EDOC'09. IEEE International*, pages 131–140. IEEE, 2009.

[29] Josino Rodrigues Neto. Software as a service: Desenvolvendo aplicações multi-tenancy: um estudo de mapeamento sistemático. Master's thesis, UFPE, Brasil, 2012.

[30] Eric Schmidt. Conversation with eric schmidt hosted by danny sullivan. In *search engine strategies conference (August 2006)*, 2006.

[31] Larisa Shwartz, Yixin Diao, and Genady Ya Grabarnik. Multi-tenant solution for it service management: A quantitative study of benefits. In *Integrated Network Management, 2009. IM'09. IFIP/IEEE International Symposium on*, pages 721–731. IEEE, 2009.

[32] Flávio RC Sousa, Leonardo O Moreira, JAF Macêdo, and Javam C Machado. Gerenciamento de dados em nuvem: Conceitos, sistemas e desafios. *Topicos em sistemas colaborativos, interativos, multimidia, web e bancos de dados, Sociedade Brasileira de Computacao*, pages 101–130, 2010.

[33] T Sridhar. Cloud computing—a primer part 1: Models and technologies. *The Internet Protocol Journal*, 12(3):2–19, 2009.

[34] S Srinivasan. *Cloud Computing Basics*. Springer, 2014.

[35] The Quality Software Company TIOBE. Tiobe index for april 2016. `http://www.tiobe.com/tiobe_index`, 2016. Accessed: 2016-04-19.

[36] Wei-Tek Tsai, Qihong Shao, and Jay Elston. Prioritizing service requests on cloud with multi-tenancy. In *e-Business Engineering (ICEBE), 2010 IEEE 7th International Conference on*, pages 117–124. IEEE, 2010.

[37] Luis M Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, 2008.

[38] Zhi Hu Wang, Chang Jie Guo, Bo Gao, Wei Sun, Zhen Zhang, and Wen Hao An. A study and performance evaluation of the multi-tenant data tier design patterns for service oriented computing. In *e-Business Engineering, 2008. ICEBE'08. IEEE International Conference on*, pages 94–101. IEEE, 2008.

[39] Jiecai Zheng, Xueqing Li, Xinxiao Zhao, Xiaomin Zhang, and Sheng Hu. The research and application of a saas-based teaching resources management platform. In *Computer Science and Education (ICCSE), 2010 5th International Conference on*, pages 765–770. IEEE, 2010.