
# Uma biblioteca Multi-Tenant para o framework Flask

por

**Bruno Resende Pinheiro**

**Trabalho de Graduação**


**UNIVERSIDADE FEDERAL DE PERNAMBUCO**

*CIN - CENTRO DE INFORMÁTICA*

*GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO*

www.cin.ufpe.br

RECIFE, JUNHO DE 2018



**UNIVERSIDADE FEDERAL DE PERNAMBUCO**

**CIN - CENTRO DE INFORMÁTICA**

**GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO**

**Bruno Resende Pinheiro**

# UMA BIBLIOTECA MULTI-TENANT PARA O FRAMEWORK FLASK

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, JUNHO DE 2018

# Resumo

Na última década o modelo de distribuição de software como serviço, mais conhecido como *Software as a Service* (*SaaS*), ficou em evidência pela facilidade do uso do mesmo e por terceirizar alguns custos que fugiam do negócio em si, como os custos de infraestrutura por exemplo. Tal modelo, faz parte de um grupo de modelos, que são configuráveis de forma que suas cobranças são calculadas de acordo com a demanda de seus respectivos usos. Uma abordagem organizacional deste modelo é a *Multi-Tenancy*, também chamada de multi-inquilino. Um sistema com arquitetura multi-tenant possui as características: configurabilidade elevada, banco de dados e outros recursos de hardware partilhados por seus inquilinos. Tais características resultam em melhor utilização dos recursos de hardware, simplifica e barateia a manutenção do sistema e custos globais. Dentre os *frameworks* habilitados para usufruto desta arquitetura existe o *micro-framework web* do *Python* chamado *Flask*. Baseado nestas características, esta pesquisa propõe uma biblioteca de código aberto, *open source*, para desenvolver projetos do *Flask* com arquitetura *multi-tenant*. Após avaliação, a biblioteca mostrou-se útil para melhor aproveitamento do tempo de desenvolvimento e portanto os mesmos não precisarão dividir o foco de implementação entre negócio e arquitetura.

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

# Abstract

In last decade, software as a service distribution model, known as SaaS, was evidenced by the easy of its use and by outsourcing some costs who deviated from the business itself, like infrastructure costs for instance. This model, is part of a group, that are configurable in a way that its billing are calculated by demand of its respective uses. An organizacional approach of this model is the *Multi-Tenancy* approach, also known as multi-tenant. A system with multi-tenant architecture brings these features: high configurability, shared database and some hardware resources shared by its tenants. Such features results in an improvement in hardware resources usage, simplifies and lower costs of system maintenance and global costs. Along the frameworks able to usufruct this architecture there is Python's web microframework called Flask. Based on this features, this research proposes an open source library, to develop Flask projects with multi-tenant architecture. After evaluation the library showed itself useful to improve development time so the developers would not need to divide the development focus between business and architecture.

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

# Sumário

# Lista de Figuras

# Capítulo 1

No presente capítulo será apresentada a estruturação do trabalho, os objetivos esperados e a motivação da realização deste, de forma a facilitar a compreensão do leitor.

## 1.1 Motivação

Computação em nuvem é uma tecnologia já presente em diversos ramos da economia que consiste de serviços gerenciados por terceiros, a estrutura fica a cargo destes também, providos pela Internet [1]. Ou seja, os usuários apenas utilizam a ferramenta que de fato necessitam, não se preocupando com manutenção, nem custos além da utilização da própria ferramenta. Como benefícios do uso da *cloud computing* vem o baixo custo do uso da ferramenta, cobrada por demanda. De acordo com pesquisa feita por Samorani [2], o cenário de infraestrutura de empresas das últimas décadas está sendo modificado. As empresas de grande e pequeno porte não estão mais mantendo infraestrutura própria para conduzirem seus negócios, portanto terceirizar recursos aparece como opção mais viável.

Com a popularização da nuvem convencionaram-se três tipos de serviço, são eles: IaaS (*Infrastructure as a Service)*, PaaS *(Platform as a Service)* e SaaS *(Software as a Service*). Infraestrutura como serviço, plataforma como serviço e software como serviço, respectivamente.

IaaS consiste do recurso de infraestrutura fornecido pela nuvem e apenas ele, como por exemplo espaço para dados. Já estão no mercado variadas opções para este

serviço, inclusive de gigantes do setor como o AWS EC2 [3]. A PaaS equivale a plataformas prontas utilizadas como auxílio para seu negócio sob demanda. Das mais conhecidas da PaaS existem a Google App Engine, OpenShift da RedHat, Salesforce, dentre outras [4]. O SaaS traz aplicações consagradas, comumente executadas nas máquinas dos usuários, como Microsoft Office, consumidas também por demanda, com nome de Office 365. Além de várias outras aplicações, dentre as quais: Google Drive e Dropbox [5] [6]. Adicionalmente existem outras divisões mais específicas, como também mais genéricas como a XaaS, Anything (ou Everything, dependendo da nomenclatura escolhida pelo autor) as a Service, que quer dizer “qualquer coisa como serviço” ou “ tudo como serviço” [7] [8].

No caso deste projeto focaremos no SaaS, na qual está inserida a abordagem organizacional *Multi-Tenancy*, como indicado por Custódio e Junior [9]. As principais características desta abordagem são: recursos de hardware compartilhados, dentre eles o banco de dados e aplicação. Não apenas a aplicação, mas também os inquilinos fazem parte do mesmo banco. Outra característica é o alto grau de configurabilidade provido pela abordagem [10].

Devido às características da *Multi-Tenancy* podemos destacar algumas vantagens geradas: melhor utilização dos recursos de hardware, melhor desempenho com relação à manutenção da aplicação e redução de custos quando comparado à abordagem tradicional [11].

Como facilitador, um *framework web* foi escolhido para desenvolver a solução proposta: *Flask*. *Flask* é um *microframework web* baseado em *Python* que está em ascensão e comumente é comparado a *Django*. Mais recente que este, *Flask* também vem em crescente uso, veja na Figura 1.1 [12]. Nesta figura é indicado um gráfico que simboliza um número crescente de perguntas relacionadas tanto à *Flask* como a *Django*. Fato é que *Flask* não tem o mesmo propósito que *Django*. *Django* é um framework que já traz todas as ferramentas prontas para o desenvolvedor utilizar, de forma que pode tornar uma aplicação simples um pouco mais carregada de código necessário para sua execução que o *Flask*. Por outro lado o desenvolvedor com *Flask* tem mais liberdade para desenvolver as próprias ferramentas, ou apenas utilizar aquilo que é necessário, pois a unidade mínima de código necessária para executar uma aplicação é bem menor. Dessa forma pode conseguir um desempenho melhor ao utilizar o sistema. Além disso essa liberdade também é útil para fins educativos, onde é necessário maior grau de conhecimento para

melhor utilização, porque torna-se necessário um estudo mais rigoroso quanto à aplicação de suas ferramentas e aquisição de conhecimento para o desenvolvimento *Web* em si [12].

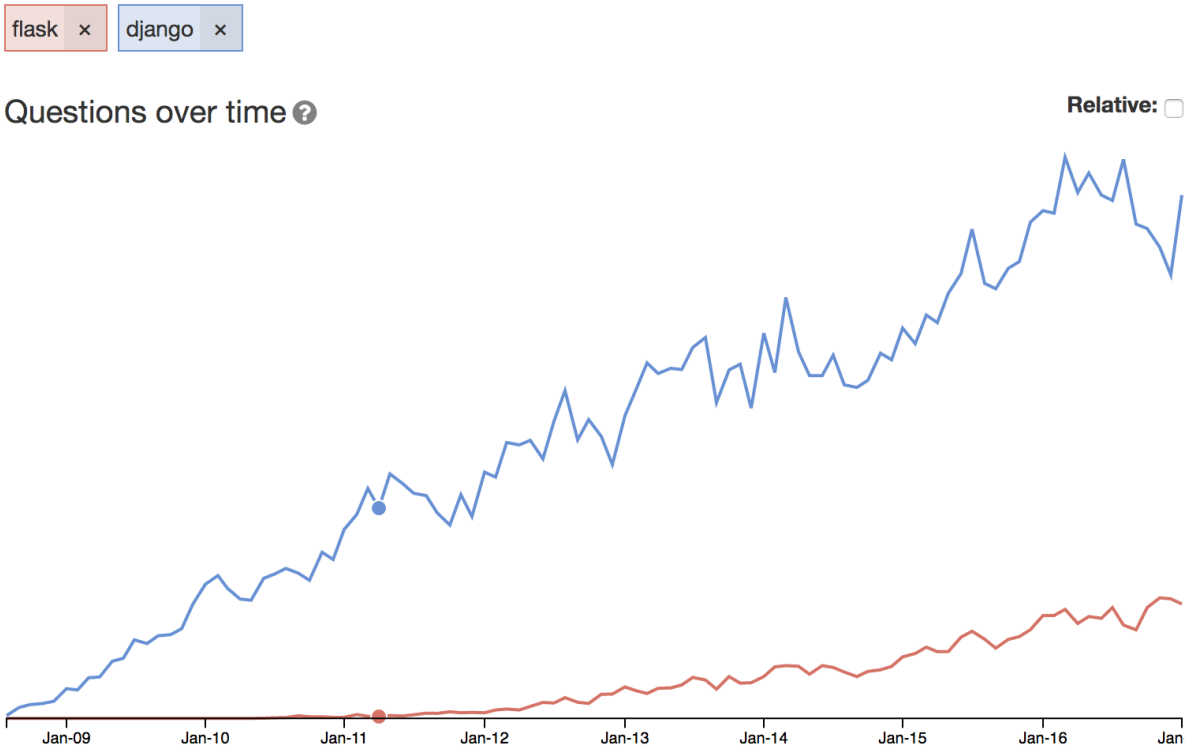


Figura 1.1: Crescimento Flask vs Django, figura extraída da comunidade Codementor de autoria de Gareth Dwyer

## 1.2 Objetivos

Nesta seção serão apresentados o objetivo geral e os objetivos específicos.

### 1.2.1 Objetivo Geral

O objetivo deste trabalho é desenvolver uma biblioteca *Multi-Tenancy open source* que possa ser utilizada por aplicações desenvolvidas em *Flask* em aplicações multi-inquilinas.

Desta forma o desenvolvedor não precisará mover recursos para desenvolver aplicações que não fazem parte da regra do negócio.

### 1.2.2 Objetivos Específicos

- Investigar e propor uma biblioteca *Multi-Tenant* para o *framework Flask*;
- Elaborar um estudo sobre a arquitetura *Multi-Tenancy* e avaliar técnicas para sua implementação;
- Mostrar através de um comparativo vantagens e desvantagens da arquitetura para demonstrar melhor adequação de utilização da mesma.

## 1.3 Organização do Trabalho

O trabalho está disposto em sete capítulos representados da seguinte maneira:

- No Capítulo 1, capítulo atual, foram apresentados conceitos base para melhor entendimento do trabalho e uma introdução com motivação e objetivos do mesmo;
- No Capítulo 2 são apresentados conceitos necessários de computação nas nuvens, SaaS e Multi-Tenancy, emparelhando as vantagens e desvantagens do mesmo, à medida que são apresentadas técnicas para implementação;
- No Capítulo 3 são apresentados o framework Flask, junto à sua arquitetura, as decisões tomadas à biblioteca implementada;
- No Capítulo 4 são mostrados trabalhos correlatos ao tema;
- No Capítulo 5 são exibidas todas as partes de implementação da biblioteca;
- No Capítulo 6 é exposto um exemplo de uso da biblioteca implementada neste trabalho;

- No Capítulo 7 são apresentadas conclusão, considerações finais e propostas para trabalhos futuros.

# Capítulo 2

# Referencial Teórico

No capítulo presente serão apresentadas definições de conceitos chave e uma visão geral sobre os seguintes conteúdos: Computação em Nuvem, Software como Serviço (SaaS) e a arquitetura *Multi-Tenant*. De acordo com Vandresen a computação em nuvem permite que uma gama de dispositivos tenham acesso a serviços que antes não poderiam ser utilizados nestes dispositivos devido à limitações dos dispositivos, como por exemplo capacidade de armazenamento e processamento [13]. Como escrito no Capítulo 1, na seção 1.1 existem três tipos de serviço convencionados. Discorreremos com mais detalhes do SaaS, um dos tipos. Focando neste último e na arquitetura multi-inquilina, este trabalho propõe uma biblioteca *Flask* para auxiliar a utilização desta arquitetura em projetos com este *framework*. Ao concluir este capítulo, o leitor compreenderá mais adequadamente o conteúdo deste trabalho e a lógica seguida pelo mesmo.

## 2.1 Computação em Nuvem

O primeiro idealizador da ideia de Computação em Nuvem foi John McCarthy, que em 1961, no MIT (*Massachusetts Institute of Technology*), foram sugeridas idéias de compartilhamento de computador e atrelar o consumo de Internet por demanda, como um serviço público, só pagar o que usar. Este conceito foi chamado de *“Utility Computing”.* Em 1962, Joseph Carl Robnett Licklider arquitetou o que seria a ARPANET anos depois [14]. O termo “nuvem” foi utilizado pela primeira vez pelo professor de sistemas de informação Ramnath Chellappa durante uma palestra acadêmica em 1997. O termo foi

inspirado na Internet, por representar algo que está no “ar”. Em 2002 a Amazon lançou um conjunto de serviços de armazenamento, computação e inteligência humana baseados em nuvem. Anos depois lançou o EC2/S3 consolidando o termo *“cloud computing”* [14].

Por ser comumente adotada pela literatura, será utilizada a definição proposta pelo *NIST (National Institute of Standards and Technology).* Eles definem *Cloud Computing* como um modelo que permite uma série de recursos computacionais compartilhados que podem ser liberados e provisionados rapidamente com o mínimo de esforço de interação ou gerenciamento do provedor de serviço [15]. Além disso o *NIST* também definiu três modelos de serviço sobre computação nas nuvens [15].

### 2.1.1 Modelos de Serviços

Ainda de acordo com o *NIST,* a computação nas nuvens possui três modelos de serviços diferentes [15]. São eles:

- **Software como Serviço (*SaaS*)**: Fornece ao consumidor a aptidão de utilizar a aplicação do provedor localizada em uma infraestrutura de nuvem. As aplicações ficam disponíveis de diversas maneiras, sendo a principal delas através de um navegador;
- **Plataforma como Serviço (*PaaS*)**: Fornece ao consumidor a aptidão de fazer *deploy* de suas aplicações na infraestrutura da *Cloud* do provedor. O consumidor pode controlar sua própria aplicação e configurar o sistema, mas não tem um controle direto sobre os recursos de infraestrutura;
- **Infraestrutura como Serviço (*IaaS*)**: Fornece ao consumidor a aptidão de prover processamento, armazenamento, conexões com rede e outros recursos de infraestrutura.

## 2.2 Software como Serviço (Saas)

Como descrito previamente a computação nas nuvens pode ser dividida em três modelos de negócio: SaaS, PaaS e IaaS. Porém de acordo com Srinivasan podemos dizer que *Cloud Computing* é um sinônimo de SaaS [2]. Isso ocorreu pelo rápido crescimento deste modelo e sua importância no mercado [16]. Para reforçar a importância do SaaS, a previsão feita pela Cisco é de uma receita de mais de dois bilhões de dólares por ano até 2020. O dobro dos outros modelos [17], PaaS e IaaS.

Para uma análise mais profunda utilizaremos a definição de Chong e Carraro. A definição que Chong e Carraro [18] utilizou foi: “Software implantado como um serviço hospedado e acessado pela Internet”. Percebe-se que não foi apontada nenhuma tecnologia, arquitetura ou modelo de negócio específico. As únicas restrições feitas são relativas à onde foi hospedado e de onde deve ser acessado. Desta forma várias aplicações podem ser envolvidas nesta definição.

Segundo Chong e Carraro [18] existem três fatores indicadores da qualidade de projeção de uma aplicação SaaS. São: escalabilidade, configurabilidade e eficiência para múltiplos inquilinos. A partir desses quatro indicadores foi possível vincular a um dos níveis de maturidade, diferenciando-se entre pela presença ou ausência dos indicadores.

Os níveis de maturidade indicados por Chong e Carraro [19], são:

- **Nível 1: Personalizado**: Semelhante ao modelo tradicional, no qual cada consumidor tem sua própria instância personalizada da aplicação e cada uma independe da outra;
- **Nível 2: Configurável**: É adicionado o fator configurável com relação ao anterior. Agora não é necessário configurar o código em cada modificação de personalização, ou seja, todas as instâncias possuem o mesmo código;
- **Nível 3: Configurável e eficiente para múltiplos inquilinos**: É adicionado o fator eficiência para múltiplos inquilinos com relação ao anterior. Neste nível há apenas uma instância, mas cada inquilino pode configurar sua prórpria experiência. Segurança e isolamento são garantidos pela aplicação.

- **Nível 4: Escalonável, configurável e eficiente para múltiplos inquilinos**: É adicionado o fator escalabilidade com relação ao anterior. Os benefícios do nível anterior são mantidos. Uma *farm* de instâncias idênticas com balanceamento de cargas entre si é acrescida. De acordo com a necessidade esta quantidade será aumentada ou diminuída, realizando assim o papel da escalabilidade.

Dependendo da necessidade será utilizado um dos níveis supracitados. No caso deste trabalho será utilizado o nível três, no qual só existe uma instância configurável servindo todos os inquilinos. Conforme Souza [20], existe uma abordagem arquitetural que serve esta configuração, chamada de *Multi-Tenancy*.

## 2.3 Multi-Tenancy

A adoção da computação nas nuvens tem auxiliado na expansão mercadológica dos negócios envolvendo tecnologia da informação [21]. A categoria de software como serviço (SaaS) possui projeções para mais crescimento em diversos setores, inclusive no mercado interno brasileiro [22]. Um dos benefícios advindos do SaaS para a regra de negócio de seus empresários é o compartilhamento de recursos que traz uma melhor utilização dos mesmos [23].

Uma abordagem para compartilhamento de recursos de banco de dados para instância e aplicações definida como *Multi-Tenancy*, ou multi-inquilina, segundo Souza [20], refere-se ao aproveitamento de uma instância de uma aplicação no banco de dados de forma simultânea sendo utilizado por vários inquilinos (*tenants*). Diferentemente de uma aplicação *Web* convencional onde há apenas um inquilino por aplicação.

Ainda de acordo com Souza [20], cada inquilino de uma arquitetura intitulada de multi-inquilino consciente pode atuar sobre a aplicação de forma que presuma ser o único usuário da mesma. Em outras palavras o inquilino não pode acessar ou visualizar outras instâncias ou outros inquilinos. Dedicado de tal forma que o inquilino pode desejar customizar sua instância.

Conforme Bezemer e Zaidman [24], um inquilino alugado da aplicação pode ser operado por um *stakeholder* de uma empresa, isto é, pode ser operado por mais de um usuário daquele negócio. Os aspectos chave considerados por eles são:

- A capacidade da aplicação de compartilhar os recursos de hardware, focando na redução dos custos;
- A oferta de um alto grau de customização, permitindo que os usuários possam configurar da forma mais adequada para si;
- Uma abordagem arquitetural na qual os inquilinos façam uso de apenas uma instância da aplicação e do banco de dados.

### 2.3.1 Aspectos Chave

Os aspectos chave são os apresentados na introdução deste capítulo, citados por Bezemer e Zaidman [24]: compartilhamento de recursos, customização e compartilhamento de banco de dados. Nesta seção estes serão mais detalhados para definir melhor a importância dos aspectos em si.

#### 2.3.1.1 Compartilhamento de Recursos

Com *Multi-Tenancy* há um redução de custos significativa pela melhor utilização dos recursos se comparado à utilização tradicional com um inquilino apenas [25]. Esta redução pode chegar a ser dezesseis vezes do custo tradicional [26]. Isto deve-se principalmente ao tempo que o recurso de hardware fica ocioso. Não só apenas pelo fato dos vários inquilinos utilizarem a abordagem que haverá maximização de aproveitamento do recurso, como também há algoritmos que trazem mais rendimento [24].

Existem três abordagens quanto às divisões do banco do banco de dados e a arquitetura multi-inquilina [19]:

- Banco de dados separados;

- Banco de dados compartilhados, esquemas separados;
- Banco de dados compartilhados, esquemas compartilhados.

Esta última abordagem foi a escolhida por este projeto devido ao seu melhor desempenho quanto à escalabilidade [19].

#### 2.3.1.2 Customização

Aplicações de apenas um inquilino são mais simples de personalização, basta os desenvolvedores criarem uma nova versão (*branch*) do software [24]. Porém para aplicações com *Multi-Tenants* é necessário um esforço consideravelmente maior, que também é necessário dado que um *tenant* manuseia o sistema como se existisse apenas ele mesmo como inquilino [27].

O alto nível de configurabilidade será o diferencial no mercado [27]. As aplicações que utilizam a arquitetura *Multi-Tenancy* são indicadas para que preparem os metadados desenvolvendo no sistema a possibilidade do que é denominado como *Self Serve Configuration.* Desta forma cada inquilino poderá carregar a configuração que melhor lhe caracteriza.

#### 2.3.1.3 Compartilhamento do Banco de Dados e Aplicação

Aplicações *Single-Tenant* utilizam de várias instâncias para atingir seus objetivos de customização. Já para *Multi-Tenant* as mesmas modificações podem ser realizadas em tempo de execução na própria instância sem necessidade de modificação na aplicação para cada inquilino desejado [24]. Consequentemente a quantidade de instâncias para aplicações multi-inquilinas pode ser de apenas uma. Já para *Single-Tenant* pode existir uma instância para cada perfil desejado. Desta forma a quantidade de *deploys* atualizações a nível de sistema cresce conforme a quantidade de instâncias. Concluindo, se a aplicação *Multi-Tenant* possuir apenas uma instância será um *deploy,* para as *Single-Tenant's* serão 'n' *deploys* para 'n' instâncias.

Existem três categorias de compartilhamento de banco de dados: bancos de dados isolados, ou seja, nenhum compartilhamento; bancos de dados com esquemas separados, isto é, parcialmente compartilhados; e bancos de dados com esquema compartilhado, de outra maneira, completamente compartilhados.

## Bancos de Dados Isolados

Representa a maneira mais simples de isolar cada inquilino [19]. Cada inquilino possui o próprio grupo de dados isolados logicamente entre si. A própria arquitetura de isolamento de cada banco garante a segurança do mesmo.

Recuperação dos dados em caso de falhas é simples. Dado que cada *tenant* possui seu próprio banco torna-se fácil a tarefa de estender o modelo da aplicação. Com relação aos custos manter um inquilino por banco se torna mais dispendioso.

Dependendo da necessidade do negócio, caso seja necessário mais segurança e personalização é um bom negócio caso esteja disposto a investir mais [19].

## Banco de dados com esquema separado

A abordagem com esquema separado possui um esquema por inquilino, em outras palavras cada *tenant* possui seu conjunto de tabelas [19].

Tal abordagem não possui o mesmo nível de segurança e isolamento da anterior, porém ela reduz o custo para manutenção para cada instância. Todavia em caso de falhas a recuperação dos dados terá um grau de dificuldade mais elevado. Isto se deve porque no caso anterior em caso de recuperação de um *backup* de um inquilino, bastaria restaurar o banco para o *backup* mais recente feito deste. No caso atual o *backup* poderá sobrescrever alterações feitas por outros inquilinos que não tenha acontecido uma falha [19].

### Banco de dados com esquema compartilhado

A última abordagem põe todos os inquilinos no mesmo banco de dados, utilizando tabelas com o mesmo esquema, separados por uma identificação em suas próprias tabelas. Desta maneira a segurança deve ser garantida por uma proteção adicional, pois não pode-se permitir que inquilinos acessem dados de outros inquilinos [19].

O custo para restaurar *backups* neste caso é similar ao anterior, com a complicação adicional de restaurar linhas indesejadas. Em contrapartida o custo de manutenção do hardware envolvido é ainda inferior aos anteriores.

### Abordagem escolhida

Assim como Bezemer [24], a abordagem escolhida foi a de banco de dados com esquema compartilhado. Esta opção foi a escolhida pela relação de baixo custo com relação a manutenção do hardware mais utilizada no mercado.

## 2.3.2 Vantagens e Desvantagens

Na seção atual serão elicitadas as vantagens e desvantagens da arquitetura *Multi-Tenancy.*

### 2.3.2.1 Vantagens

Baseado na literatura foram levantadas algumas vantagens e desvantagens. O quesito de vantagens foi mais numeroso quando comparado às desvantagens. No meio das vantagens aparecem:

- Atualização em apenas um *deploy*, para todos os inquilinos [24];

- Custo reduzido para manutenção de infraestrutura de hardware [28];
- Redução do custo de venda do software, utilizando o conceito de “milhões de mercados de poucos” citado por Sousa [20];
- Baixa necessidade de adequação com relação às regras de negócio, em outras palavras, alto reaproveitamento do código da aplicação [24];
- Elasticidade rápida para escalar a aplicação para quantos inquilinos forem necessários e para o que cada inquilino usufruir [20].

#### 2.3.2.2 Desvantagens

Fazem parte das desvantagens encontradas na literatura:

- Reestruturar aplicações legadas para arquitetura multi-inquilina demanda um custo de desenvolvimento não trivial [24];
- Quanto maior for o grau de configurabilidade maior será o custo de manutenção da aplicação em termos de desenvolvimento [24];
- Existem limitações computacionais que podem tornar-se em gargalos, devido ao número de instâncias. Tais limitações devem ser combatidas/mitigadas para o correto funcionamento do sistema [29].

## 2.4 Considerações Finais

Os conceitos básicos necessários para este trabalho foram apresentados neste capítulo. Dos quais foram mostrados: *Cloud Computing*, *SaaS*, *PaaS* e *IaaS*. Posteriormente foi exibida a definição da arquitetura *Multi-Tenancy.* Foram discutidos os aspectos chave levantados pela literatura presente nas referências bibliográficas. Também foram descritas vantagens e desvantagens acerca da arquitetura de acordo com os artigos explorados. O modelo *SaaS* e a arquitetura *multi-tenant* são foco da proposta deste trabalho bem como

a utilização do framework *Flask*, que é a ferramenta sobre a qual a biblioteca MulT tem como base. Desta forma no próximo capítulo será apresentado como surgiu o framework Flask, além das razões da escolha de cada componente utilizado para o desenvolvimento da MulT.

# Capítulo 3

# Flask e Biblioteca Proposta

## 3.1 Flask

*Flask* é um *micro-framework* de código aberto para desenvolvimento de aplicações *Web*. Ela foi escrita em *Python* e por ser um *framework* minimalista originalmente não se encaixa nas nomenclaturas de modelos arquiteturais existentes como *MVC* (*Model*, *View*, *Control*). Mas com o suporte de algumas bibliotecas, como o *SQLAlchemy*, pode funcionar de maneira semelhante e dependendo do autor, considera-se que ele se encaixa neste padrão [30].

De acordo com Dwyer [12], *Flask* permite que o desenvolvedor escolha exatamente o que utilizar com relação a bibliotecas, banco, entre outras ferramentas. Isto pode gerar um melhor desempenho da aplicação dependendo de seu desenvolvimento, além do aprendizado necessário pra concluir a aplicação.

*Flask* surgiu de uma brincadeira de “1º de Abril”, em 2010, quando seu criador, Armin Ronacher, publicou uma página falando sobre um novo “*micro-web-framework*” e a comunidade gostou da idéia mais que suas expectativas [31]. O nome veio de uma brincadeira com o *micro-framework Bottle* [32] e a partir daí, Armin Ronacher, começou a pensar sobre como seria a arquitetura do Flask.

### 3.1.1 Arquitetura do Flask

A arquitetura do *framework* começa permitindo a escolha do desenvolvedor sobre que *ORM* (*Object-Relational Mapping*) utilizar para persistir. A mais utilizada de acordo com Garbade [33], é *SQLAlchemy*, quando não se trata do *framework Django* (este possui um *ORM* próprio). Ainda é possível utilizar outras bibliotecas conhecidas como *MongoDB* ou *SQLite*. Desta forma o critério para a *ORM* escolhida fica a cargo inteiramente do desenvolvedor e da aplicação.

Chegando aos formulários a indicada por Copperwaite e Leifer, foi a popular biblioteca *WTForms* [34]. Simples de utilizar e bem documentada, a *WTForms* abstrai desde campos de texto até campos de submissão, facilitando a vida do desenvolvedor. Sem esta biblioteca, de acordo com Picard [35], provavelmente os tratamentos dos campos nos formulários seriam realizados nas *views*.

Para *templates* o *Flask* necessita do *Jinja*, que é um *template engine* para *Python*. De acordo com a documentação oficial [36], para executar o *Flask* necessariamente, ele deve estar instalado, ou seja, ainda que o desenvolvedor escolha trabalhar com outro *template engine, o Jinja* é obrigatório para o bom funcionamento do *framework*. Porém por ser uma biblioteca independente do *Python* é possível utiliza-lo sem estar atrelado ao *Flask*. Similar à *engine* do *Django Templates*, o *Jinja* possui sintaxe parecida e tão poderosa quanto, sendo possível migrar o *template* com poucas alterações [37].

Abaixo na Figura 3.1 [38], segue um exemplo ilustrando o funcionamento de uma aplicação *Flask*, utilizando as mesmas ferramentas (*WTForms*, *Jinja*) à proposta deste trabalho.

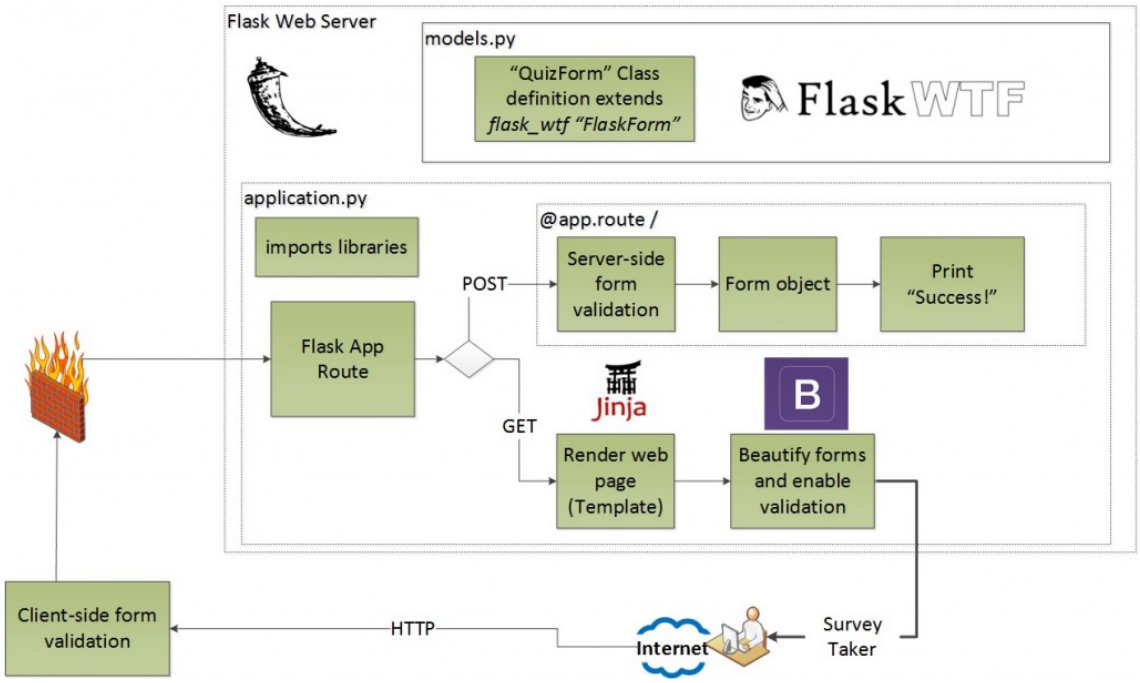


Figura 3.1: Exemplo de arquitetura do *Flask*

*WSGI* é um padrão *Python* [39] que faz a interface entre um *Web Server* e a aplicação *web Python*, conforme Figura 3.2 [40]:

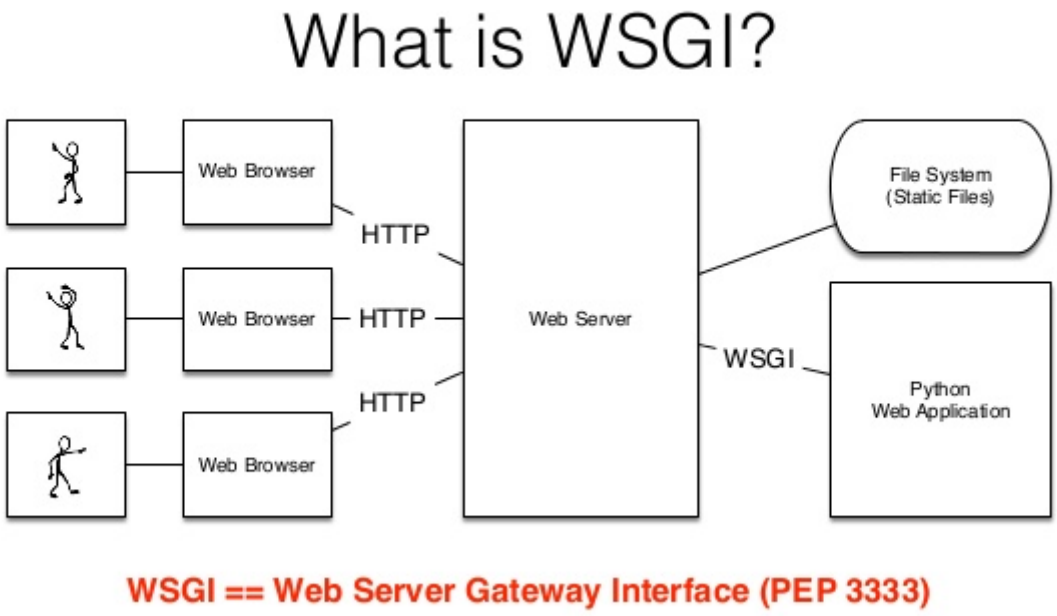


Figura 3.2: Função do *WSGI*

*Werkzeug* é um módulo avançado de utilidades baseado no *WSGI*, que segundo Ronacher [41] nasceu como um simples conjunto de utilidades. O *Werkzeug* junto ao *Jinja* é necessário para executar as aplicações *Flask*. Muitas dessas ferramentas são as mais populares entre os desenvolvedores, facilitando a busca por informações para o desenvolvimento.

## 3.2 Decisões de Projeto

Objetivando a utilização de maior parte de desenvolvedores e maior auxílio com relação à documentação, a escolha das ferramentas utilizadas dentro do projeto se basearam nas mais populares. Desta forma o trabalho pode atender a maior quantidade de desenvolvedores para o *framework Flask* possíveis, ainda que seja possível utilizar por cima da biblioteca MulT outras ferramentas que atendam às necessidades dos desenvolvedores. Isto é, as ferramentas utilizadas pela MulT não são necessariamente conflitantes com outras, apenas facilitam o trabalho exercido no desenvolvimento da maioria.

Para persistência a escolhida foi a *SQLAlchemy*, conforme informação passada por Garbade [33], é a biblioteca mais utilizada pelos desenvolvedores de *Flask*.

Segundo Picard [35], uma das bibliotecas mais populares para tratamento de formulários é a *WTForms*. Desta forma para facilitar a abstração da lógica e o *clean* dos campos que serão renderizados nos *templates*, esta foi selecionada.

Sobre *templates*, foi destacado que o *Jinja* é necessário para a execução da aplicação, além disto alia-se o fato de Flask-User ser uma poderosa ferramenta com ampla documentação e sintaxe moderna para utilização [36]. Portanto a versão mais recente do *Jinja*, *Jinja2*, foi adotada.

Uma biblioteca foi utilizada e partindo dela foi desenvolvida a biblioteca proposta no trabalho, foi a *Flask-User*. Tal biblioteca possui uma gama de ferramentas completas e prontas para utilização que fornecem autenticação com segurança para os desenvolvedores de *Flask*. Ainda sobre esta, notou-se que utiliza todas as ferramentas supracitadas mais populares.

As demais ferramentas utilizadas foram configuradas como padrão por não haver necessidade de customização para atender os objetivos especificados. Isso inclui o *Werkzeug*, necessário e suficiente para cumprir demanda essencial. Cada mecanismo escolhido para a biblioteca possui diversas carasterísticas que se adequam à esta e que viabilizam sua utilização.

### 3.2.1 SQLAlchemy

SQLAlchemy é um ORM, mapeamento objeto-relacional, que assim como outros auxilia no gerenciamento do banco de dados de um sistema. Como descrito por Garbade [33], para o Flask, a biblioteca mais utilizada para ORM é a SQLAlchemy. Abaixo está listado um conjunto de características, que segundo Leplatre [42] o SQLAlchemy possui:

- Leve e um framework independente;
- Dividido em um conjunto mínimo de características limpas e bem implementadas;
- Bastante flexível e suporta mapeamento dos objetos usando metadados ou declarações;
- A documentação de sua API é preenchida por longas explicações e casos de uso;
- Sua utilização requer domínio sobre mecanismos e arquitetura de Python avançados.

Como vantagens e desvantagens pode-se enumerar de acordo com Gantan [43] quatro características mais marcantes. São as vantagens:

- Uma API nível empreendimento, *enterprise-level*, que torna o código robusto e adaptável;
- Design flexível, para simplificar geração query's complexas.

Já como desvantagens:

- O conceito incomum de unidade de trabalho, *Unit-of-work*, que simplificadamente significa que a ORM reúne todas as modificações realizadas numa sessão e realiza o commit junto à esta [44];

- Possui uma grande curva de aprendizado.

### 3.2.2 Jinja 2

Jinja é uma engine de templates que auxilia no desenvolvimento front-end. O Jinja 2 é a segunda versão do Jinja e conta com a maior popularidade dentre as engines conhecidas e utilizadas no Flask [36]. As principais caracterísiticas do Jinja 2, para Dugar [45], são:

- Execução Sandbox, para prover maior segurança ao sistema;
- HTML Escaping, alguns caracteres possuem significados especiais nos templates, estes para utilização como texto regular precisam ser substituídos por entidades;
- Herança de template, que melhora o nível de reutilização do código para o front-end.

Para Rubio [46] podemos levantar como vantagens do Jinja 2:

- Compilação do código fonte do template para Python bytecode no primeiro carregamento, de modo que este só é traduzido uma vez resultando numa performance melhor em tempo de execução;
- Alta flexibilidade em termos do que pode conter, de forma que permite suporte a conceitos como macros e construtos “pythônicos”;
- Inspiração no Django templates, com baixa curva de aprendizado. Ainda possui características poderosas como herança de templates e blocos;
- Suporte à carregamento assíncrono de funções e elementos mais pesados. Este recurso está disponível apenas para projetos com Python 3.6 ou superior, pois utiliza geradores assíncronos.

Como desvantagens, Rubio [46] aponta:

- Em projetos Django, o suporte ao Jinja foi concedido apenas na versão 1.8, o que caracteriza uma versão relativamente recente, não sendo possível projetos inteiramente feitos com o Jinja, diferentemente do Flask;

- Novos conceitos, como macros e filtros do Jinja, requerem um certo grau de estudo para ambientação com tais ferramentas.

## 3.3 Considerações Finais

Fundamentando-se a partir do conhecimento levantado sobre as informações citadas neste capítulo, a escolha dessas ferramentas se mostraram adequadas com base nas suas vantagens que foram abordadas e o próximo capítulo propõe discernir sobre as soluções já existentes e verificar a aplicabilidade delas relativo à resolução do problema.

# Capítulo 4

# Trabalhos Correlatos

Neste capítulo serão exibidos os trabalhos pertinentes à pesquisa.

## 4.1 Aspectos chave em trabalhos correlatos

Há também os aspectos chave levantados na literatura que direcionam o comportamento ligado à banco de dados, customização e autenticação necessários para a biblioteca funcionar dentro dos padrões, descritos no capítulo anterior, que a arquitetura exige.

### 4.1.1 Banco de Dados

Segundo Bezemer [24], quanto ao compartilhamento do hardware, ele menciona a abordagem do *Multi-Tenancy* puro. Ainda de acordo com autor, os outros métodos, aplicação compartilhada com banco de dados separado e aplicação compartilhada com banco de dados compartilhado mas esquemas separados, possuem um desempenho aquém do aceitável, principalmente após crescimento da quantidade de inquilinos. Segundo Rodrigues [15], é importante assegurar o isolamento dos dados, pois todos os inquilinos utilizam o mesmo banco e é preciso garantir que só possam acessar seus próprios dados.

Por esta razão, o trabalho atual segue o pensamento de Bezemer [24], utilizando o banco compartilhado com esquema compartilhado, consequentemente o custo computacional será menor e servirá um número de *tenants* maior.

### 4.1.2 Customização

No trabalho de Bezemer [24], o autor apresenta a configurabilidade em *Multi-Tenancy* como simples e eficiente, a implementação ficou posicionada na área de atualizações e *deploys*. De acordo com Rodrigues [15], as aplicações multi-inquilinas devem possuir formas de se ajustar às necessidades de aplicação de seus usuários.

Neste trabalho, a customização foi implementada atrelada ao layout da página, assim o usuário será capaz alterá-lo com facilidade. De maneira semelhante à de Bezemer [24], pois está habilitado a personalizar alguma característica da aplicação.

### 4.1.3 Autenticação

Como apresentado por Rodrigues [15], por razão de uma aplicação utilizar uma instância de banco de dados e dela mesma, todos os inquilinos serão mantidos no mesmo ambiente físico. Para alcançar a customização garantindo que cada inquilino acesse apenas seus dados, será utilizado o mecanismo de autenticação.

De acordo com H. Kim e D. Kim [47], é possível que apenas o tenant atual tenha acesso bloqueando os outros, através do mecanismo de acesso mutualmente exclusivo. Assim as aplicações multi-inquilinas compartilham os recursos de maneira inteligente e de acordo com políticas determinadas.

O presente trabalho utiliza de mecanismos que baseiam-se também nos fundamentos apresentados por H. Kim e D. Kim [47] para autenticar os usuários na aplicação multi-inquilina.

## 4.2 Application Dispatching

É possível gerar múltiplas instâncias de aplicações *Flask* a nível do *WSGI* (*Web Server Gateway Interface*) do *Python*. Ou seja, apenas configurando no arquivo que gera a aplicação é possível gerar instâncias por subdomínios ou até *path's.* Cada instância de subdomínio pode ser vinculada a um usuário [48]. Uma das maneiras de fabricar cada instância é colocá-la dentro de funções com laços simples isto corresponde a uma aplicação do que a literatura denomina por *Application Factory*. Este padrão também útil para casos de testes unitários e utilizá-lo significa possuir o controle de inicialização de cada aplicação. Posteriormente cada aplicação deve ser vinculada à suas próprias dependências, entre elas o banco de dados. A inicialização dos bancos de dados não precisa ficar junto às aplicações. Desta forma podem ser levantados apenas quando necessários de acordo com cada aplicação [49].

Uma versão mais simplificada encontrada na página de fábrica das aplicações seria apenas configurar a mesma aplicação com mínimas características personalizadas, através de rotas para cada subdomínio e modificar o SERVER_NAME de cada um. Tal solução corresponde a mais singela das levantadas.

Porém há limitações na configurabilidade destas aplicações de forma que não atende a necessidade de customização de alguns usuários [50].

## 4.3 Blueprints

*Blueprints* são instâncias do *Flask* que se comportam como aplicações secundárias e se acoplam à principal. O conceito de Blueprints na literatura é: "elas registram operações para executar quando registradas numa aplicação" [51]. É uma ferramenta abstrata que possui diversas utilidades das quais numa destas seria simular cada inquilino numa Blueprint diferente. Blueprints compartilham do mesmo objeto de aplicação do *Flask*, ou seja, mesmos arquivos de configuração. Isto significa que para subir uma nova aplicação é necessário derrubar toda a aplicação anterior, em outras palavras todas as Blueprints.

Porém assim como no caso anterior, há limitações na customização destas “aplicações”, além disso não há o isolamento necessário entre as aplicações de forma que são interdependentes e portanto não atende à necessidade de alguns usuários [50].

## 4.4 SQLAlchemy

Esta solução é proposta para o caso em que múltiplos há bancos de dados. Solução utiliza biblioteca para banco de dados amplamente utilizada para o framework *Flask*, SQLAlchemy. A literatura desenvolveu uma função para selecionar que banco de dados vincular ao passar nome do banco via URL, usando como base a extensão a biblioteca SQLAlchemy [52]. Porém não faz parte do requisito levantado no projeto de compartilhar o mesmo banco de dados para todos os inquilinos e aplicação.

## 4.5 Django Multi-Tenant através de App Dispatching

O *framework Django* possui uma biblioteca, chamada de Django Multi-Tenant[1], que disponibiliza um *middleware* capaz de tornar suas aplicações aptas a utilizar a arquitetura *Multi-Tenancy.* De acordo com a documentação do *framework Flask* é possível executar aplicações *Flask* e *Django* lado a lado [48]. Pois ambos utilizam o *WSGI* (*Web Server Gateway Interface*). Para isso é necessário configurar o projeto *Flask* para receber a biblioteca *Django* como um *middleware*. Esta precisa receber as informações da requisição da página para adicionar o subdomínio e o *tenant*. Segundo Denis Kyorov algumas configurações adicionais ainda são necessárias [53].

Tal biblioteca possui algumas dependências como versão do *Python* que não abrange a diversos projetos. Por exemplo, o projeto atual foi feito com a versão 2.7, uma das versões mais utilizadas ainda, apesar da comunidade estar migrando para o *Python* 3. Portanto não será possível utilizá-lo. Na Figura 4.1 é possível visualizar o erro causado ao tentar instalar a biblioteca no ambiente proposto.

1 A biblioteca Django Multi-Tenant foi desenvolvida por Arimatea Neto e pode ser acessada através do endereço: https://github.com/arineto/django-multi-tenant .

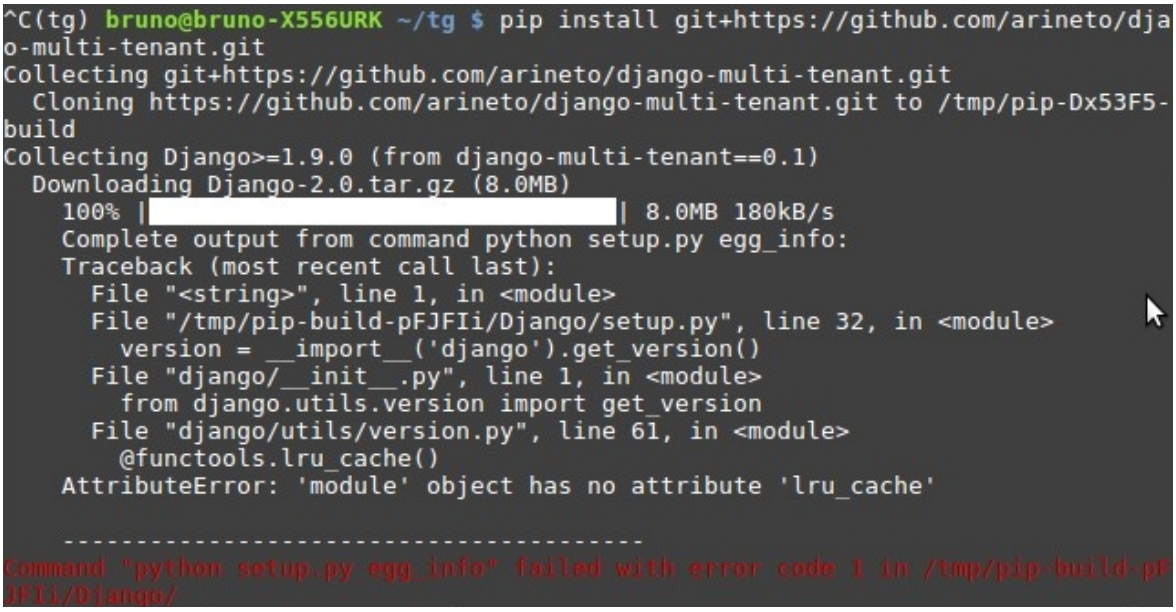


Figura 4.1: Erro devido à versão da linguagem utilizada

## 4.6 Django Shared Schema Tenants

Além da biblioteca Django Multi-Tenant, foi criada uma biblioteca nomeada Django Shared Schema Tenants[2], ou Django-SSTenants. Esta possui flexibilidade maior que a anterior no que concerne a configuração dos projetos. Isto é, não é tão simples para inserir um projeto como a Django Multi-Tenant, porém possui uma gama superior de possibilidades devido a quantidade de suposições menor feitas pelo desenvolvedor da última. Para vinculá-lo ao Flask configurações adicionais são necessárias, conforme mencionado no tópico Django Multi-Tenant [53]. Porém para diminuir a complexidade da configuração exigida não iremos utilizá-la.

2 A biblioteca Django Shared Schema Tenants foi desenvolvida por Hugo Bessa e pode ser acessada através do endereço: https://github.com/hugobessa/django-shared-schema-tenants .

## 4.7 Grails Multi-Tenant

Existem abordagens de outras linguagens que trazem uma solução multi-inquilina para seus próprios frameworks. Desta forma bastaríamos configurá-las para o *Flask* para que as utilizássemos sem necessidade de desenvolver uma própria. Para o caso de *Grails*, existe uma biblioteca desenvolvida por Josino Rodrigues Neto [15], chamada Grails Multi-Tenant.

Um motivo que torna o Grails Multi-Tenant inviável para ser aplicado num projeto *Flask* é a interface do *WebServer*, que no caso de *Flask,* por utilizar *Python*, é a *WSGI*. Para o *Grails* quem desempenha esta função é a *JVM*, da Oracle, que também cumpre tal papel para outras linguagens, como *Java* por exemplo. Levando em consideração esta inviabilidade destacada, não será possível utilizar a biblioteca desenvolvida por Josino para viabilizar a arquitetura Multi-Tenant no Flask.

## 4.8 Considerações Finais

Após ponderar sobre os trabalhos encontrados em pesquisa de correlatos do tema, foi verificado que utilizando as soluções existentes não foram atendidas as demandas necessárias nos casos referenciados. Algumas vezes por pouca customização, outra por discrepância de requisitos como utilização de múltiplos bancos de dados e não isolamento de aplicações. Em outros casos por discrepância de dependências ou frameworks.

Além disso foram definidos os aspectos necessários para desenvolver a ferramenta e atacar o problema.

# Capítulo 5

# MulT

MulT[3] é como foi denominada a biblioteca que visa auxiliar no desenvolvimento de sistemas multi-inquilinos para os desenvolvedores de *Flask*. Para a biblioteca foram desenvolvidas funcionalidades que pretendem solucionar vários problemas. Tais como:

- Modelos para classes de inquilinos e abstrações necessárias. Além de managers auxiliadores;
- Personalização para cada inquilino;
- Controle de acesso de acordo com a permissão do usuário;
- Funcionalidades extras, para auxiliar o desenvolvedor.

Nas seções sequentes serão explanados cada um dos itens. A Figura 5.1 ilustra de maneira simplificada o modelo conceitual da MulT.

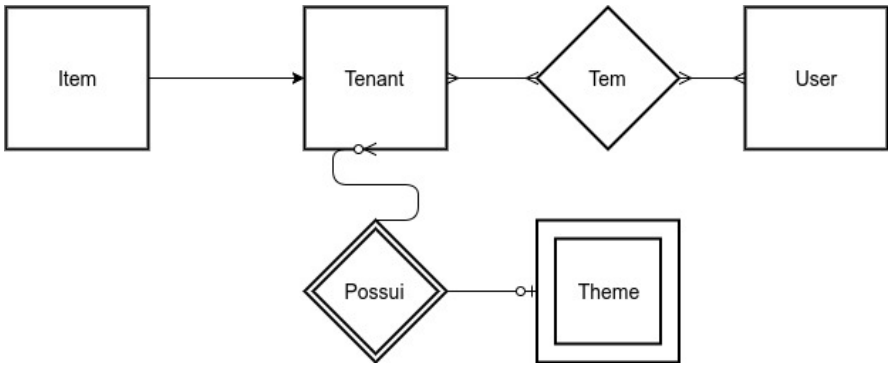


Figura 5.1: Modelo Conceitual do MulT

3 A biblioteca pode ser acessada através do link: https://github.com/brunoresende1991/mult

## 5.1 Arquitetura da biblioteca proposta

Foram investigadas algumas possibilidades de abordagens de dados para arquitetura *multi-tenant,* como analisado na seção 2.3.1.3. Ao final da seção foi escolhida a de banco de dados compartilhado com esquema compartilhado.

Foram criados dois modelos do Flask, como ilustrado nas Figuras 5.2 e 5.3.

O modelo *Tenant* apresentado na Figura 5.2 representa os inquilinos do sistema. Este modelo é composto por um id, um alias que identifica unicamente o inquilino no sistema, uma associação de usuários que tem acesso ao sistema por esse inquilino e uma associação de temas que estão vinculados a este inquilino.

```
class Tenant(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    alias = db.Column(db.String(200), nullable=False, unique=True)
    users = db.relationship('User', secondary=users, lazy='subquery',
                            backref=db.backref('tenants', lazy=True))
    themes = db.relationship('Theme', secondary=themes, lazy='subquery',
                             backref=db.backref('users', lazy=True))
```

Figura 5.2: Modelo *Tenant*

O modelo TenantModel descrito na Figura 5.3 representa a abstração que será herdada por modelos que irão aproveitar do esquema de dados compartilhados. É através deste modelo que será herdada uma chave estrangeira para o inquilino vinculado.

```
class TenantModel(db.Model):
    __abstract__ = True

    @declared_attr
    def theme_id(cls):
        return db.Column(db.Integer, db.ForeignKey('theme.id'), nullable=False)

    @declared_attr
    def tenant_id(cls):
        return db.Column(db.Integer, db.ForeignKey('tenant.id'),
                         nullable=False)

    id = db.Column(db.Integer, primary_key=True)
```

Figura 5.3: Modelo *TenantModel*

Além do mais foram implementados *managers* para facilitar na busca pelos tenants, como mostrado na Figura 5.4.

```
@classmethod
def by_tenant(cls, dbsession, tenant_id):
    return dbsession.query(cls).filter_by(id=tenant_id).all()

@classmethod
def by_tenants(cls, dbsession, tenants_id):
    return dbsession.query(cls).filter(cls.id.in_(tenants_id))
```

Figura 5.4: Métodos de Classe *by_tenant* e *by_tenants*

Desta forma o desenvolvedor terá trabalho facilitado para buscar tenants e também utilizá-los.

## 5.2 Personalização

Conforme descrito nos aspectos chaves a configurabilidade é uma das principais características da arquitetura *Multi-Tenant*. Personalização é um aspecto também presente em arquiteturas que não sejam multi-inquilinas, mas para esta ele é fundamental. Através de um dos temas disponíveis o inquilino pode personalizar seu perfil na aplicação.

Está disponível pela biblioteca um modelo *Theme*, como ilustrado na Figura 5.5 de um tema que é representado por uma coluna para armazenar o nome de um arquivo predefinido que está no diretório *static*.

```
class Theme(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    style = db.Column(db.String(255), nullable=False)
```

Figura 5.5: Modelo *Theme*

Para disponibilizar um novo tema é necessário fazer uma atualização na aplicação inserindo um novo arquivo css no diretório *static*.

## 5.3 Controle de acesso

Por Flask ser um micro-framework muitas vezes algumas bibliotecas não existem prontas para uso ou necessitam ser forkeadas de terceiros. Neste caso para evitar o esforço apenas com controle de acesso foi forkeada uma biblioteca chamada Flask-User[4] feita pelo desenvolvedor Ling Thio.

Tal biblioteca possui várias ferramentas prontas com relação à autenticação e apenas foi necessária configurá-la junto ao projeto. Desta forma toda a problemática relacionada ao controle de acesso foi terceirizada. Na Figura 5.6, é possível identificá-la no modelo User.

```
# Define the User data model. Make sure to add flask_user UserMixin !!!
class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    # User authentication information
    username = db.Column(db.String(50), nullable=False, unique=True)
    password = db.Column(db.String(255), nullable=False, server_default='')

    # User email information
    email = db.Column(db.String(255), nullable=False, unique=True)
    confirmed_at = db.Column(db.DateTime())

    # User information
    active = db.Column(
        'is_active', db.Boolean(), nullable=False, server_default='0')
    first_name = db.Column(
        db.String(100), nullable=False, server_default='')
    last_name = db.Column(
        db.String(100), nullable=False, server_default='')
```

Figura 5.6: Modelo *User*

## 5.4 Funcionalidades Auxiliares

Algumas funcionalidades auxiliares também foram desenvolvidas. Elas introduzem informações na geração de contexto e na requisição.

4 A biblioteca pode ser acessada através do link: https://github.com/lingthio/Flask-User

### 5.4.1 *Middlewares*

Estes *middlewares* tem a função de acrescentar parâmetros à requisição. Como pode ser visto na Figura 5.7, esta biblioteca possui dois *middlewares*. O *SubdomainMiddleware* captura o subdomínio na URL e o acrescenta na *request*. Enquanto o *TenantMiddleware* acrescenta o *tenant* ao request, que pode ser reconhecido pelo subdomínio.

```
@app.before_request
def middleware_subdomain():
    domain = request.environ['HTTP_HOST']
    parts = domain.split('.')
    try:
        request.subdomain = parts[-2]
    except IndexError:
        request.subdomain = None

@app.before_request
def middleware_tenant():
    tenant_model = Tenant
    try:
        tenant = tenant_model.query.filter_by(
            alias=request.subdomain).one()
    except exc.NoResultFound:
        tenant = None
    request.tenant = tenant
```

Figura 5.7: *Middlewares*

### 5.4.2 *Context Processors*

Para adicionar informações ao contexto existem as funções dos *Context Processors*, por fim tais informações são passadas aos templates. Como mostrado na Figura 5.8, três destas foram implementadas.

A *subdomain* coloca o subdomínio no contexto. A *tenant* coloca o *tenant* no contexto. Já a *theme* coloca o tema no contexto.

```
@app.context_processor
def subdomain():
    return {
        'subdomain': request.subdomain
    }

@app.context_processor
def tenant():
    return {
        'tenant': request.tenant
    }

@app.context_processor
def theme():
    return {
        'theme': request.tenant.themes[0] if request.tenant and
        len(request.tenant.themes) > 0 else None
    }
```

Figura 5.8: *Context Processors*

## 5.5 Considerações finais

A biblioteca foi disponibilizada como open source, através de licença BSD 2, versão simplificada, para que outros desenvolvedores já possam utilizá-la e aprimorá-la. Tal licença foi herdada da ferramenta Flask-User e mantida com propósito de maior difusão. Este capítulo mostrou a estrutura da biblioteca através do código de seus principais modelos e módulos para ilustrar o papel de cada um.

# Capítulo 6

# Um exemplo de uso da biblioteca

No capítulo atual será demonstrado um exemplo de uso da MulT. Serão demonstrados durante o capítulo cada parte implementada. São elas:

- Personalização do tema por inquilino;
- Autenticação como controle de acesso para o inquilino;
- Filtros de inquilinos vinculados à usuários;
- Recuperação de informações de contexto e *request*, para *tenant*, subdomínio e tema. Este último para caso do contexto;
- Modelo com herança do TenantModel. Possui intúito de herdar característica do modelo abstrato que mantém relacionamento com *tenant*.

## 6.1 Personalização do tema por inquilino

Conforme descrito na seção 5.2, o MulT oferece personalização de tema por inquilino. Por padrão o inquilino quando criado possui o *body* na coloração preta, quando acessada a Página de Membros, assim como mostrado na Figura 6.1.

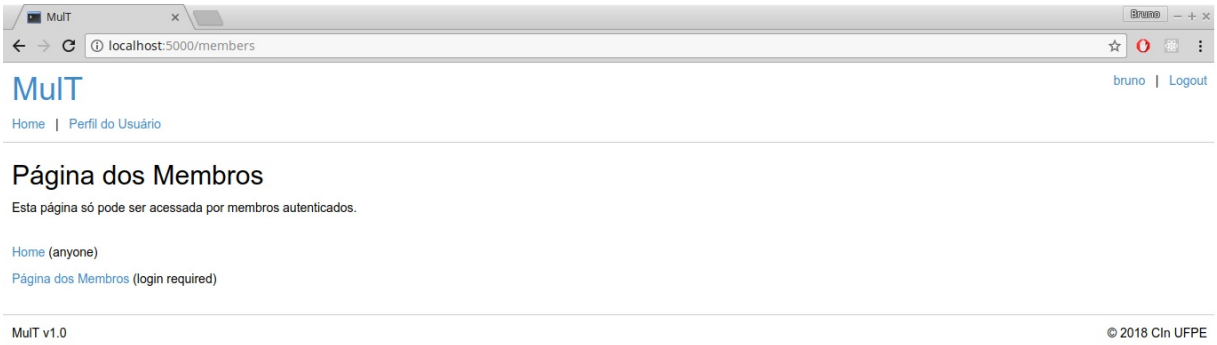


Figura 6.1: Tema padrão, *body* com coloração preta

Ao acessar a página Perfil de Usuário → Change Theme, escrever um dos temas válidos (*blue*, *red* ou *green*), indicar o *tenant* que deseja modificar e informar a senha do usuário para validar a coloração mudará para a indicada. A Figura 6.2 ilustra o passo descrito.

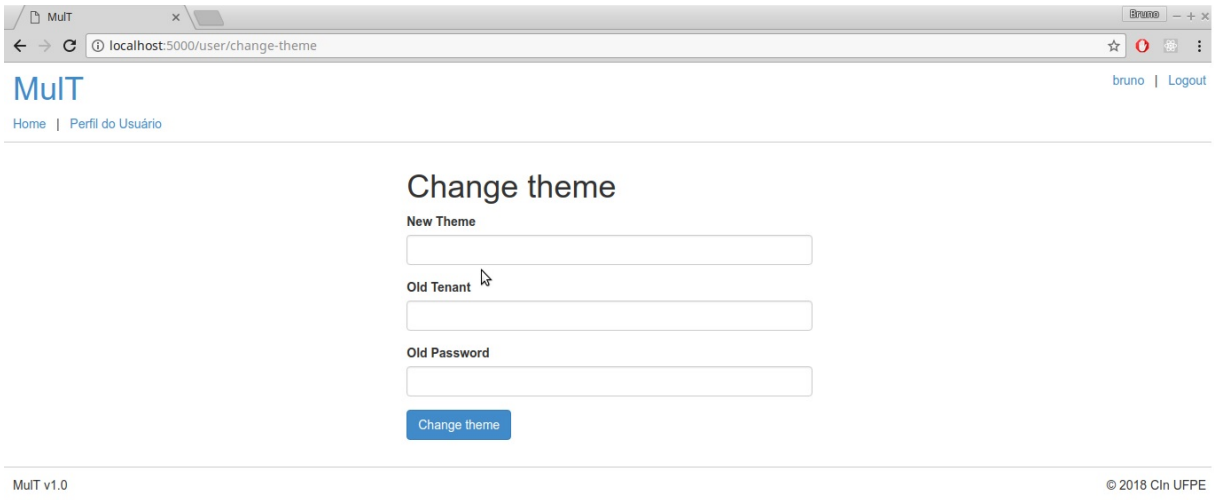


Figura 6.2: Página para modificação do tema

Na sequência a Figura 6.3 mostra o efeito esperado ao modificar o tema padrão para o *red*.

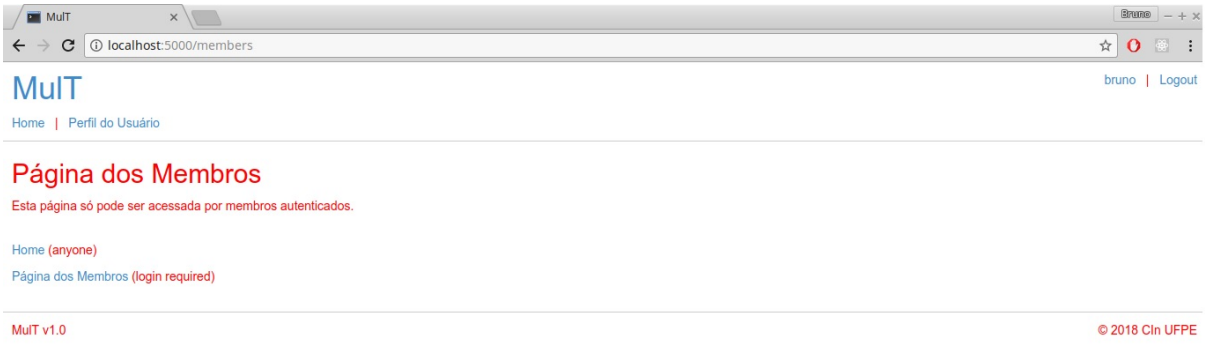


Figura 6.3: Tema red, *body* com coloração vermelha

## 6.2 Autenticação como controle de acesso para o inquilino

Nesta demonstração foi criado um inquilino para um novo usuário brp, chamado tenant2. De acordo com a Figura 6.4 foi mostrado que este usuário não tinha acesso ao tenant1, que foi utilizado para fazer a demonstração anterior.

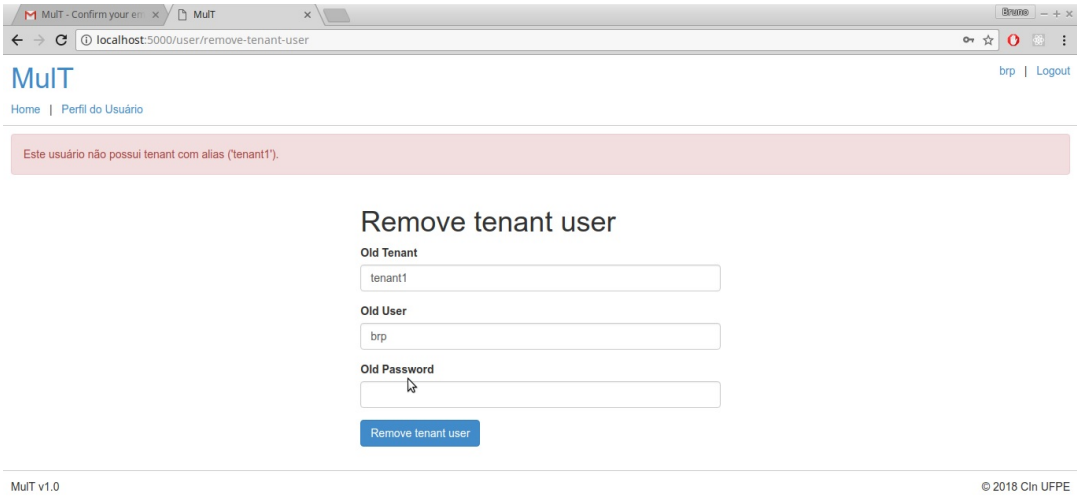


Figura 6.4: Acesso negado ao inquilino tenant1

Já na Figura 6.5 é possível visualizar que este usuário consegue acessar e completar a ação determinada ao tenant2.

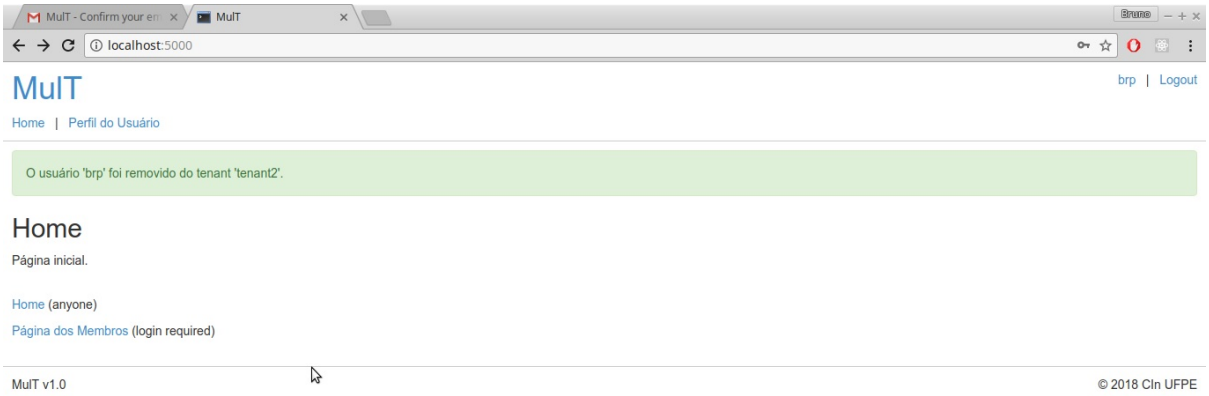

Figura 6.5: Acesso permitido ao inquilino tenant2

## 6.3 Filtros de inquilinos vinculados à usuários

Os *managers*, indicados na seção 5.1, são ferramentas para auxiliar desenvolvedores a implementar códigos futuros, que correspondem à métodos de classe. Estes atendem uma necessidade para buscar através de *querys* no banco de dados informações especificamente de um *tenant* ou um conjunto destes. Na Figura 6.6 é indicado um exemplo de uso com os *tenants* criados acima.

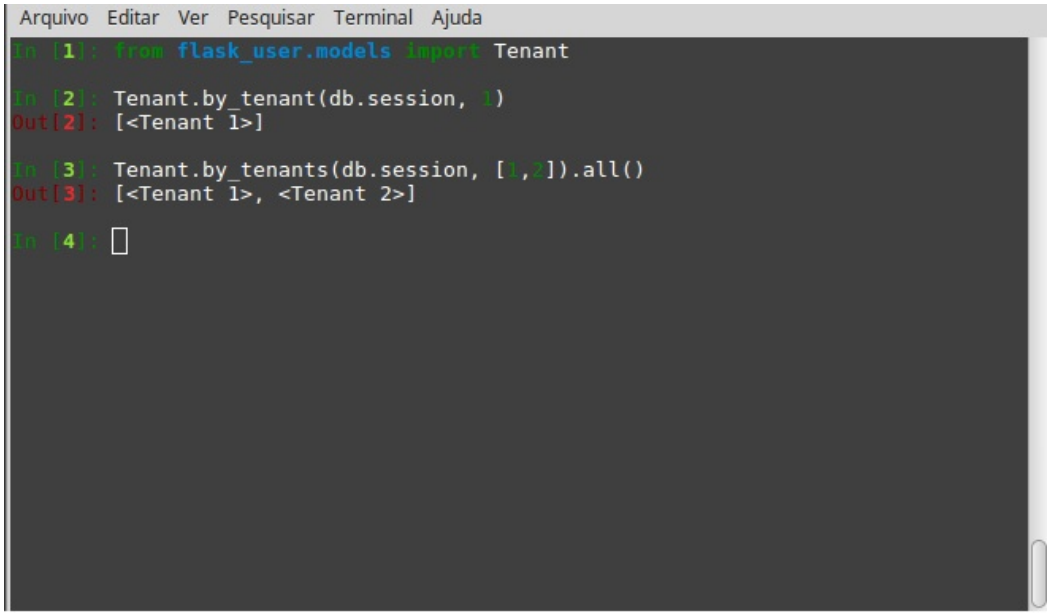


Figura 6.6: Managers *by_tenant* e *by_tenants* no shell

## 6.4 Recuperação de informações de contexto e *request*

Uma forma que o administrador tem de recuperar informações de estado do sistema é verificando as que estão presentes no contexto e na requisição enviada pelo usuário. Desta forma é possível verificar autenticação do usuário acessando o *tenant* através do subdomínio. Ou seja, o usuário pode passar informações direto no navegador sem

necessidade de passar via sistema ou *shell*, como um administrador. A Figura 6.7 ilustra o acesso dos *tenants* em paralelo através de sessões diferentes por conta de diferentes versões de janelas, uma convencional e uma anônima.

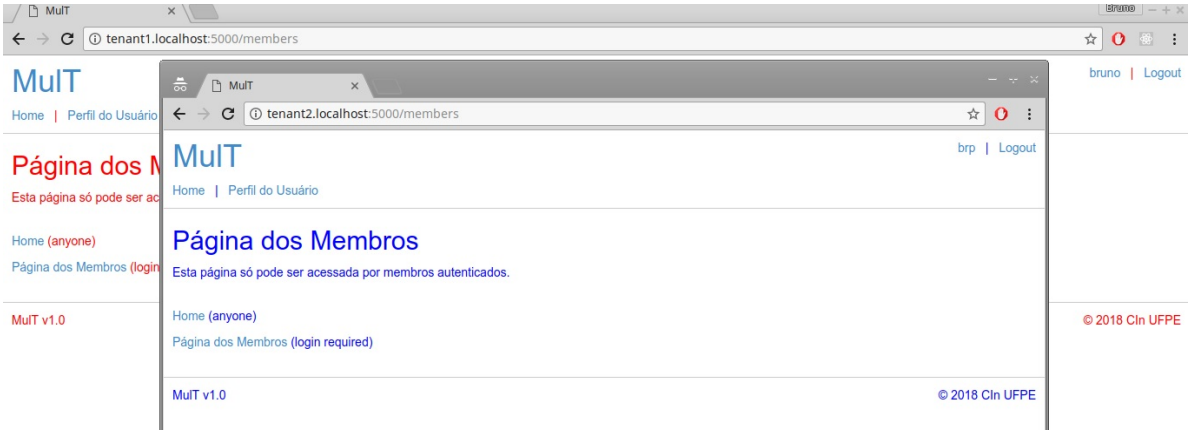


Figura 6.7: Acesso à *tenants* através de diferentes sessões

Na Figura 6.8 é possível visualizar a recuperação de informações de subdomínio e *tenant*, para o administrador, via *shell*.

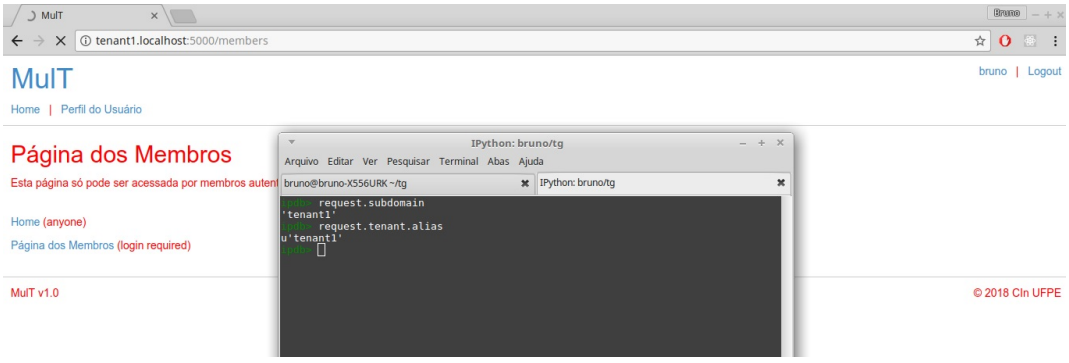

Figura 6.8: Informação de *tenant* e subdomínio consultadas no *request* através do *shell*

Já na Figura 6.9 é possível verificar na imagem o contexto de uma aplicação em execução parada por depuração, por motivos de demonstração, as informações de subdomínio, *tenant* e tema.

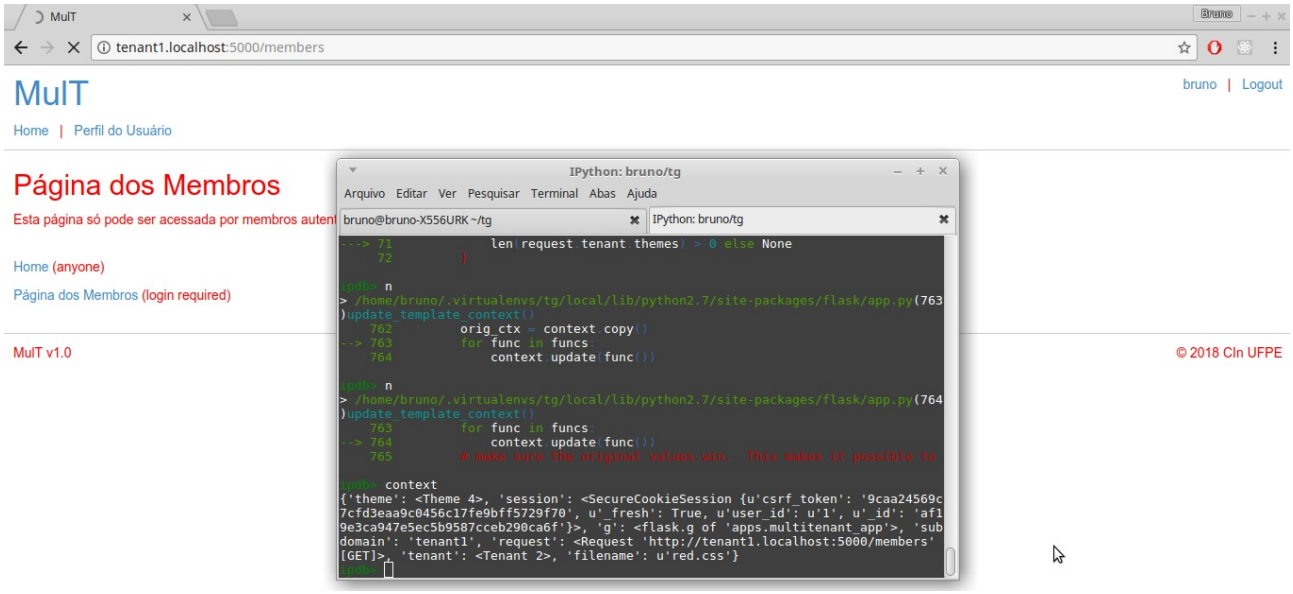


Figura 6.9: Informação de *tenant,* subdomínio e tema verificadas no contexto através de depuração

## 6.5 Modelo com herança do TenantModel

Para simbolizar a herança que pode ser utilizada na biblioteca foi construído um modelo, chamado Item, que herda de TenantModel. Este modelo possui todas as características e atributos do seu pai, TenantModel. Ele mantém um vínculo a um determinado tema e *tenant.* Além disso atribui novas colunas à tabela que pertencerão a uma possível regra de negócio. A Figura 6.10 ilustra o código do modelo.

```
class Item(TenantModel):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(200), nullable=False)
    code = db.Column(db.Integer, nullable=False)
    created_at = db.Column(db.DateTime, default=datetime.datetime.utcnow)
```

Figura 6.10: Modelo Item, herdando do modelo TenantModel

## 6.6 Considerações finais

O funcionamento da biblioteca foi demonstrado da maneira idealizada, conforme objetivo que propunha uma biblioteca para realizar trabalho de arquitetura multi-inquilina. Com isto a expectativa é que o foco esteja na regra do negócio ao invés dos meios para alcançá-lo.

# Capítulo 7

# Conclusão e trabalhos futuros

Com a crescente da tecnologia de Computação nas Nuvens tem ficado cada vez mais popular o SaaS. Junto a ele a arquitetura *Multi-Tenancy* traz diversos benefícios para as aplicações que fazem parte do SaaS. Este trabalho mostrou as benesses oriundas da utilização de uma biblioteca multi-inquilina como por exemplo uma melhoria no aproveitamento dos recursos e um avanço na manutenção da aplicação.

O objetivo desta atividade foi de auxiliar no desenvolvimento de aplicações com a questão arquitetural *Multi-Tenant* de modo à concentrar os esforços dos desenvolvedores na regra de negócio e não na arquitetura. Isto é, obter uma redução na complexidade da aplicação por utilizar uma ferramenta que atende ao requisito da arquitetura multi-inquilina.

O projeto traz funcionalidades como controle de acesso, customização de temas e arquitetura de dados compartilhada. Desta forma a aplicação *Flask* pode ser desenvolvida sem os percalços da antiga faltante arquitetura necessária.

Para atingir tais funcionalidades foram encontradas dificuldades atreladas ao fato do Flask ainda não ter atingido tantos desenvolvedores como outros frameworks Python, de forma que diminui sua documentação. A implementação desta também contou com exaustiva busca por bibliotecas com autenticação que instaladas não necessitam de alto grau de configurabilidade, pois não era o foco do trabalho desenvolver uma biblioteca de autenticação, nem o Flask traz uma biblioteca pronta com tal propósito. Foram necessárias além destas, horas de estudo para entender como se deve implementar com o padrão utilizado pelo SQLAlchemy, para se construir corretamente as consultas e relacionamentos.

Não foram realizadas análises de escalabilidade, porque a ferramenta é encaixada no nível três de maturidade, de forma que apenas no nível quatro se atende ao critério de escalabilidade.

Podem ser trabalhos futuros:

- Implementar a biblioteca em outros frameworks como *rails* com o intúito de expandir ainda mais a mesma;
- Disponibilizar nesta ou em outra biblioteca arquiteturas irmãs, como por exemplo com o mesmo banco de dados mas esquemas separados;

- Adicionar mais ferramentas de personalização para a biblioteca de acordo com a necessidade dos *stakeholders*;
- Medir empiricamente a relevância desta biblioteca em projetos *Flask,* de forma a demonstrar o desempenho de aplicações multi-inquilinas e convencionais.

# Referências Bibliográficas

[1] Mitchell, Bradley. What is Cloud Computing? 2017. Disponível em: <https://www.lifewire.com/what-is-cloud-computing-817770>. Acesso em: 28 maio 2018.

[2] S Srinivasan. Cloud Computing Basics. Springer, 2014.

[3] Sullivan, Dan. Getting Started with the Cloud: Amazon EC2 Cloud. 2013. Disponível em: <http://www.tomsitpro.com/articles/ec2-elastic_map_reduce-iaas-amazon_machine_images-amazon_s3,2-448.html>. Acesso em: 28 maio 2018.

[4] Burns, Christine. 10 most powerful PaaS companies. 2014. Disponível em: <https://www.networkworld.com/article/2288002/paas/cloud-computing-10-most-powerful-paas-companies.html#slide2>. Acesso em: 28 maio 2018.

[5] Jordão, Fabio. Comparação: 8 dos melhores serviços para você guardar arquivos na nuvem. 2016. Disponível em: <https://www.tecmundo.com.br/computacao-em-nuvem/57904-comparacao-8-melhores-servicos-voce-guardar-arquivos-nuvem.htm>. Acesso em: 28 maio 2018.

[6] Panettieri, Joe. Top 25 Most Popular SaaS, Cloud apps for business 2017. 2017. Disponível em: <https://www.channele2e.com/channel-partners/csps/top-25-saas-cloud-apps-for-business-2017/>. Acesso em: 28 maio 2018.

[7] Dixon, John. X as a service (XaaS): What the future of cloud computing will bring. 2014. Disponível em: <https://www.cloudcomputing-news.net/news/2014/aug/18/x-as-a-service-xaas-what-the-future-of-cloud-computing-will-bring/>. Acesso em: 28 maio 2018.

[8] Garcia, R. C.; Chung, J. M. XaaS for XaaS: An evolving abstraction of web services for the entrepreneur, developer, and consumer. Midwest Symposium on Circuits and Systems, p. 853–855, 2012.

[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] Wang H., Zheng Z. (2010) Software Architecture Driven Configurability of Multi-tenant SaaS Application. In: Wang F.L., Gong Z., Luo X., Lei J. (eds) Web Information Systems and Mining. WISM 2010. Lecture Notes in Computer Science, vol 6318. Springer, Berlin, Heidelberg.

[11] Shaikh, F.; Patil, D. SaaS based Multi-tenant E-commerce to reduce cost and improve resource utilization. Proceedings os 2015 IEEE 9th International Conference on Intelligent Systems and Control, ISCO 2015, p. 1-6, 2015.

[12] Dwyer, Gareth. Flask vs. Django: Why Flask Might Be Better. 2017. Disponível em: <https://www.codementor.io/garethdwyer/flask-vs-django-why-flask-might-be-better-4xs7mdf8v>. Acesso em: 23 maio 2018.

[13] Vandresen, R. S.; Magalhães, W. B. Conceitos E Aplicações Da Computação Em Nuvem. p. 2009–2013, 2009.

[14] IPM. História da computação em nuvem: como surgiu a cloud computing? 2017. Disponível em: <https://www.ipm.com.br/blog/historia-da-computacao-em-nuvem-como-surgiu-a-cloud-computing/>. Acesso em: 28 maio 2018.

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

[16] Kowalke, Peter. What Does the Future of the SaaS Industry Look Like? 2017. Disponível em: <https://www.agilecrm.com/blog/future-saas-industry-look-like/>. Acesso em: 28 maio 2018.

[17] CISCO. Cisco Global Cloud Index : Forecast and Methodology , 2014–2019. White Paper, p. 1–41, 2014.

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

[19] Chong, Frederick, Gianpaolo Carraro, and Roger Wolter. "Multi-tenant data architecture." MSDN Library, Microsoft Corporation (2006): 14-30.

[20] Sousa, Flávio RC, et al. "Gerenciamento de dados em nuvem: Conceitos, sistemas e desafios." Topicos em sistemas colaborativos, interativos, multimidia, web e bancos de dados, Sociedade Brasileira de Computacao (2010): 101-130.

[21] Kono, C.M., Junior, L.C.F.E.S., Rodrigues, L.C.,. Inovação Na Gestão Com a Adoção Da Computação Em Nuvem : Aplicação Na Modelagem De Negócio De Pequenas E Médias Empresas. III Simpósio Internacional de Gestão de Projetos, 2013.

[22] Pereira, J., Da Silva, E.O., Batista, T., Delicato, F.C., Pires, P.F., Khan, S.U., 2017. Cloud Adoption in Brazil. IT Professional 19, 50–56. doi:10.1109/MITP.2017.27

[23] W; Garg, S. K.; Buyya, R. SLA-Based Resource Allocation for Software as a Service Provider (SaaS) in Cloud Computing Environments. 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, p. 195–204, 2011.

[24] 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.

[25] Guo, C. J. et al. A Framework for Native Multi-Tenancy Application Development and Management A Native Multi-tenancy Enablement Framework Challenges of the Native Multi-tenancy Pattern. ECommerce Technology and the 4th IEEE International Conference on Enterprise Computing ECommerce and EServices 2007 CECEEE 2007 The 9th IEEE International Conference on, p. 551--558, 2007.

[26] Warfield, Bob. Multitenancy Can Have a 16:1 Cost Advantage Over Single-Tenant. 2007. Disponível em: <https://smoothspan.com/2007/10/28/multitenancy-can-have-a-161-cost-advantage-over-single-tenant/>. Acesso em: 28 maio 2018.

[27] Sun, W. et al. Software as a service: Configuration and customization perspectives. Proceedings - 2008 IEEE Congress on Services, SERVICES 2008, v. PART2, p. 18–24, 2008.

[28] Walraven, Stefan, Eddy Truyen, and Wouter Joosen. "A middleware layer for flexible and cost-efficient multi-tenant applications." Proceedings of the 12th International Middleware Conference. International Federation for Information Processing, 2011.

[29] 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.

[30] Sivji, Aly. Building a Flask Web Application (Flask Part 2). 2017. Disponível em: <https://alysivji.github.io/flask-part2-building-a-flask-web-application.html>. Acesso em: 28 maio 2018.

[31] Ronacher, Armin. April 1st Post Mortem. 2010. Disponível em: <http://lucumr.pocoo.org/2010/4/3/april-1st-post-mortem/>. Acesso em: 28 maio 2018.

[32] Lubanovic, Bill. Introducing Python: modern computing in simple packages. " O'Reilly Media, Inc.", 2014. pagina 235

[33] Garbade, Dr. Michael J.. Django, Flask ou Pyramid: Qual é o melhor framework Python para você? 2017. Disponível em: <http://blog.liveedu.tv/django-flask-pyramid-framework-python/>. Acesso em: 21 maio 2018.

[34] Copperwaite, Matt; Leifer, Charles. Forms and Validation. In: Copperwaite, Matt; Leifer, Charles. Learning Flask Framework. Livery Place: Packt Publishing, 2015. cap. 4, p. 76-76.

[35] Picard, Robert. Handling forms. Disponível em: <http://exploreflask.com/en/latest/forms.html>. Acesso em: 21 maio 2018.

[36] Ronacher, Armin. Templates. 2010. Disponível em: <http://flask.pocoo.org/docs/1.0/templating/>. Acesso em: 21 maio 2018.

[37] Ronacher, Armin. Switching from other Template Engines. 2008. Disponível em: <http://jinja.pocoo.org/docs/2.10/switching/>. Acesso em: 21 maio 2018.

[38] Sobansk, John. Easy ReCAPTCHA with Flask-WTF. 2018. Disponível em: <https://www.freshlex.com/add-recaptcha-to-your-flask-application.html>. Acesso em: 19 maio 2018.

[39] Eby, Phillip J.. What is WSGI? Disponível em: <http://wsgi.readthedocs.io/en/latest/what.html>. Acesso em: 21 maio 2018.

[40] Dumpleton, Graham. Secrets of a WSGI master. 2017. Disponível em: <https://www.slideshare.net/GrahamDumpleton/secrets-of-a-wsgi-master>. Acesso em: 21 maio 2018.

[41] Ronacher, Armin. Werkzeug. 2014. Disponível em: <http://werkzeug.pocoo.org/>. Acesso em: 21 maio 2018.

[42] Leplatre, Mathieu. SQLAlchemy, a brave new World. 2011. Disponível em: <http://blog.mathieu-leplatre.info/sqlalchemy-a-brave-new-world.html>. Acesso em: 25 maio 2018.

[43] Central, Python. Python’s SQLAlchemy vs Other ORMs. 2014. Disponível em: <https://www.pythoncentral.io/sqlalchemy-vs-orms/>. Acesso em: 15 abr. 2018.

[44] Kanzler, Zach. Convenient PyCharm Live Templates. 2017. Disponível em: <https://gist.github.com/theY4Kman>. Acesso em: 15 nov. 2017.

[45] Dugar, Diva. Jinja2 Explained in 5 Minutes! 2016. Disponível em: <https://codeburst.io/jinja-2-explained-in-5-minutes-88548486834e>. Acesso em: 19 jan. 2018.

[46] Rubio, Daniel. Jinja advantages and disadvantages. Disponível em: <https://www.webforefront.com/django/usejinjatemplatesindjango.html>. Acesso em: 19 nov. 2017.

[47] 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.

[48] Ronacher, Armin. Application Dispatching. 2017. Disponível em: <http://flask.pocoo.org/docs/0.12/patterns/appdispatch/>. Acesso em: 28 maio 2018.

[49] Ronacher, Armin. Application Factories. 2017. Disponível em: <http://flask.pocoo.org/docs/0.12/patterns/appfactories/#app-factories>. Acesso em: 28 maio 2018.

[50] Cuba, Roberto. Flask multi-tenant and app feature configuration. 2016. Disponível em: <https://stackoverflow.com/questions/37353533/flask-multi-tenant-and-app-feature-configuration>. Acesso em: 28 maio 2018.

[51] Ronacher, Armin. Modular Applications with Blueprints. 2017. Disponível em: <http://flask.pocoo.org/docs/0.12/blueprints/>. Acesso em: 28 maio 2018.

[52] Koskinen, Miikka. Multitenant Flask-SQLAlchemy. 2016. Disponível em: <https://quanttype.net/posts/2016-03-15-flask-sqlalchemy-and-multitenancy.html>. Acesso em: 28 maio 2018.

[53] Kyorov, Denis. Django, Flask, and Redis Tutorial: Web Application Session Management Between Python Frameworks. 2013. Disponível em:

<https://www.toptal.com/django/django-flask-and-redis-sharing-user-sessions-between-frameworks>. Acesso em: 28 maio 2018.