PABX IP Documentation

Versão 0.1

Fabio Hiroki

20/03/2012

Sumário

CAPÍTULO 1

Introdução

Esta documentação tem o objetivo de servir de base para auxiliar o entendimento do código-fonte do projeto PABX-IP, facilitando as futuras extensões de funcionalidades a serem desenvolvidas.

CAPÍTULO 2

Recomendações

Antes de mais nada, o desenvolvedor deve estar a par da documentação das funcionalidades do PABX que é outro documento separado desse. Alguns diagramas de fluxo e esboço de telas também foram feitos também, mas é bem provável que estes estejam desatualizados agora.

Para entendimento do código e dessa documentação é importante que o leitor tenha um conhecimento geral de programação orientada a objetos, banco de dados e desenvolvimento web. É indispensável um conhecimento prévio no framework Django.

Conteúdo

3.1 Para começar

Nessa seção será mostrado as ferramentas que são pré-requisitos para desenvolvimento do projeto. Também mostraremos como instalá-las e configurá-las corretamente, para que o desenvolvedor já consiga ao menos rodar o PABX-IP em sua máquina local.

3.1.1 Ambiente de Desenvolvimento

- Sistema Operacional (recomendável): Ubuntu
- Python 2.7 (a versão 3 é incompatível com outras bibliotecas)
- Framework Django + Pinax
- Banco de dados SQLite
- Conexão com a internet

3.1.2 Instalação do ambiente de desenvolvimento

Esse tutorial foi feito no Ubuntu 10.04, usando o repositório git onde o código estava sendo versionado na época em que essa documentação foi feita. Caso o desenvolvedor já possua o código fonte, pule a parte 1.

Instalação do git e clone do repositório:

```
$ sudo apt-get install git-core
```

```
$ git clone https://github.com/fabiothiroki/pabx_ip.git
```

Instalação e ativação do virtualenv:

O virtualenv é uma ferramenta de criação de ambientes python isolados. Isso visa facilitar o deploy da aplicação.

Toda vez que o desenvolvedor quiser rodar o servidor local de desenvolvimento do django é necessário ativar esse ambiente, pois é nele que estarão instalados o Pinax e as bibliotecas python auxiliares.

Antes de mais nada é interessante que todas as bibliotecas do OS sejam atualizadas

```
$ sudo apt-get update
$ sudo apt-get dist-update
```

Agora então vamos a instalação do virtualenv:

```
$ sudo apt-get install python-pip

$ sudo pip install virtualenv

$ virtualenv pabx-env

$ source pabx-env/bin/activate
```

Instalação do Pinax e outros apps:

Pinax é uma plataforma baseada no Django que contém diversos apps pré-instalados.

Para mais informações consulte: http://pinaxproject.com/

```
(pabx-env)$ pip install Pinax
(pabx-env)$ pip install django_compressor
(pabx-env)$ pip install django_debug_toolbar
(pabx-env)$ pip install django_staticfiles
(pabx-env)$ pip install pinax_theme_bootstrap
```

Instalação do Django:

Django é o framework web utilizado no projeto.

Para mais informações consulte: http://djangoproject.com/

```
(pabx-env)$ pip install Django
```

Instalação do Django South:

O South é utilizado para implementar o controle da estrutura e migração do banco de dados. Seus arquivos estão na pasta 'south', no diretório raiz do projeto.

Para mais informações consulte: http://south.aeracode.org/

```
(pabx-env)$ pip install south
```

Conclusão:

Por fim utilize o seguinte comando no diretório raiz do projeto para ligar o servidor de desenvolvimento:

```
$ python manage.py runserver
```

A seguinte mensagem deverá ser retornada no terminal em caso de sucesso:

```
Validating models...

0 errors found
```

```
Django version 1.3.1, using settings 'pabx_ip.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Entre com o endereço http://127.0.0.1:8000/ no seu navegador para acessar a interface web do projeto.

O banco de dados padrão do projeto vem com o usuário super-admin com login *root* e senha *1234*.

3.1.3 Máquina Virtual

Para facilitar a instalação do ambiente de desenvolvimento foi configurada uma máquina virtual já com as bibliotecas instaladas. O nome de usuário e a sua senha de root respectiva são:

- *username:* pabx
- *password:* pabx

O diretório do projeto está contido em "~/pabx_ip/".

3.2 Visão geral do código

A linguagem escolhida (Python) para o desenvolvimento do projeto prioriza a legibilidade do código sobre a velocidade, bem como o framework Django. Assim, um desenvolvedor com uma certa experiência em Python não terá maiores problemas para entender o código, mesmo porque o mesmo se encontra num estágio inicial.

Ainda sim esse documento visa explicar o código mais detalhadamente e ao mesmo tempo dando alguma noção de Django.

Além dos arquivos *urls.py* e *settings.py* nenhum dos outros arquivos na pasta raiz do projeto foi codificado de fato anteriormente, mas foi gerado automaticamente pelo framework e suas ferramentas.

Os outros arquivos codificados se encontram dentro das pastas dos respectivos Apps, que serão explicados abaixo.

3.2.1 Apps do Django

Os apps desenvolvidos anteriormente foram (cada app corresponde a uma pasta na raiz do projeto):

- **Accounts**
- **Groups**
- **Skypelist**
- **Smtp**

Cada App do Django corresponde a um objeto assim como na programação orientada a objetos, então cada um deles possui métodos e atributos próprios seguindo uma arquitetura de software MVC adaptada.

O código desenvolvido no projeto está seguindo a estrutura de apps do Django, assim como a documentação relativa a essa parte.

Cada App documentado contém seu arquivos numa pasta de mesmo nome na raiz principal do projeto. Assim, a estrutura da documentação de cada app passa a ser a seguinte:

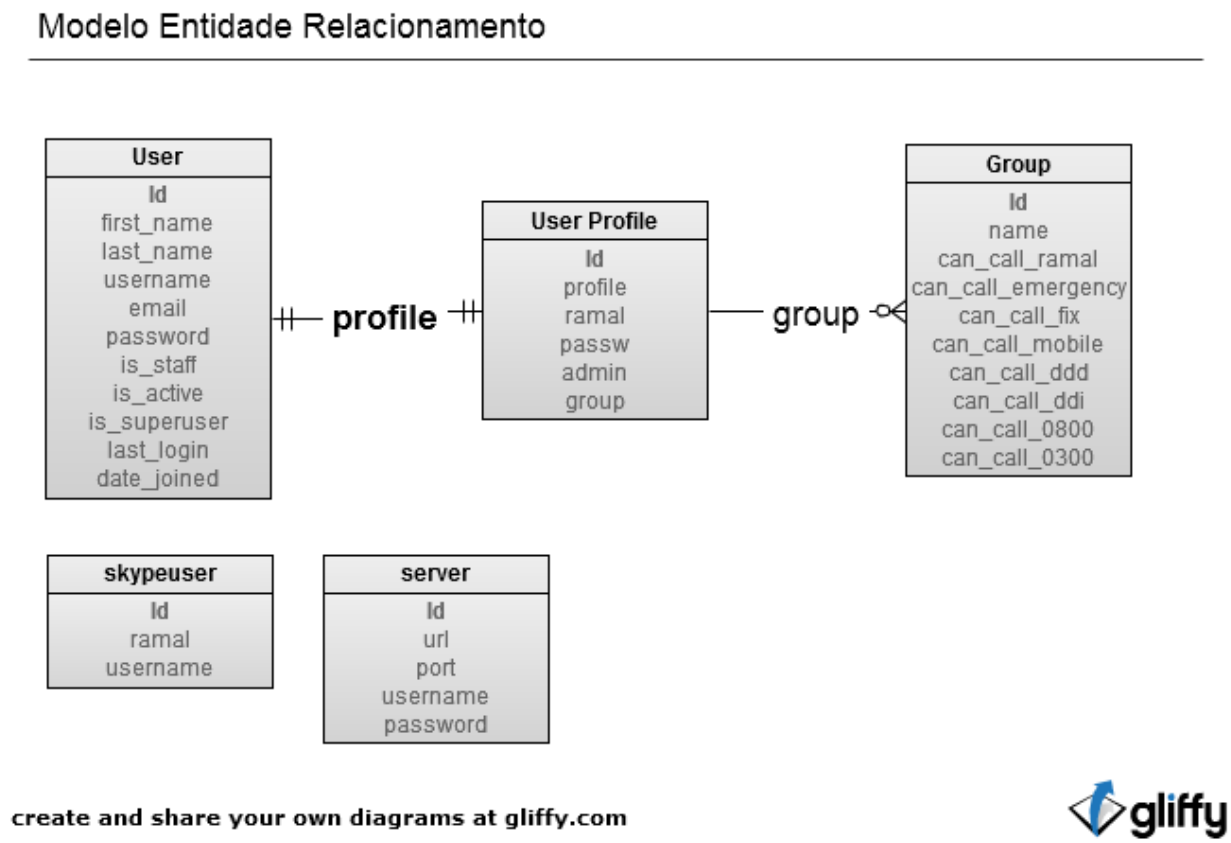
Formulários

Relativo ao arquivo *forms.py*, é responsável por criar as classes de formulários, indicando quais campos devem ser incluídos e sua validação. É um arquivo opcional, ou seja, ele existirá somente se o App precisar.

Modelos

Relativo ao arquivo *models.py*, é justamente o *Model* da arquitetura MVC.

Segue MER atual da aplicação web:



Views

Relativo ao arquivo *views.py*. Embora o nome seja confuso esse arquivo é mais parecido com o *Controller* da arquitetura MVC.

É reponsável por:

- Toda lógica de negócios (direta ou indireta)
- Atribuir variáveis a serem exibidas num *template*
- Fazer chamadas ao banco de dados

Decorators

Relativo ao arquivo *decorators.py*. Aqui são declaradas algumas funções de uso geral que são chamadas diversas vezes nas views. É um arquivo opcional, ou seja, ele existirá somente se o App precisar.

Templates

Relativo aos arquivos dentro da pasta "templates". Embora o nome seja confuso esse arquivo é mais pareciso com a *View* da arquitetura MVC, pois é aqui onde a interface do usuário é renderizada a partir de elementos construídos no arquivo *views.py*

Eventualmente algum App possa precisar de arquivos templates adicionais que estarão contidos na pasta "templates" dentro da pasta do App. Além desses arquivos templates, os outros templates padrão estarão contidos na pasta "templates" dentro da pasta raiz.

3.2.2 Integração com o Asterisk

Por enquanto a aplicação web do PABX-IP não está integrado com o Asterisk (o software responsável pelo PBX). Para isso seria necessário escrever os dados necessários nos arquivos de configurações utilizados pelo Asterisk, usando sua própria sintaxe.

A integração entre a aplicação web e o Asterisk foi pensada da seguinte forma:

- Todas as configurações seriam salvas no banco de dados da aplicação web, evitando assim de ter que "carregar" essas configurações a partir do Asterisk toda vez que a aplicação web fosse acessada.
- Qualquer alteraçao dessas configurações seria primeiramente salva no banco de dados e em seguida encaminhada para o Asterisk.

3.3 App Accounts

Este app é responsável por:

- Definir privilégios de administrador a usuários e limitar acesso a usuários comuns
- Cadastro, edição, listagem e remoção de usuários
- Autenticação e recuperação de senha

3.3.1 Modelos

No arquivo models.py estão as classes definidas para armazenar informações extras do usuário. Podemos observar que a classe UserProfile tem uma chave estrangeira na classe User, que é a classe padrão para usuários do Django. Desta maneira, podemos extender os atributos da classe User sem alterar a estrutura de usuário do Django, bastando apenas fazer essa pequena extensão.

Os atributos definidos na classe UserProfile estão detalhados a seguir:

class `accounts.models.`**`UserProfile`**

- **profile**: Chave estrangeira que associa um UserProfile a um Usuário. Pode haver apenas um UserProfile por User.
- **ramal**: Ramal associado a um usuário. Esse número é utilizado para fazer ligações.
- **passw**: É a cópia da senha do usuário salva encriptadamente. É utilizada para recuperação de senha.

•**admin**: Indica se o usuário possui o privilégio de administrador.

•**group**: Indica o grupo o qual o usuário pertence.

3.3.2 Formulários

class accounts.forms.**UserForm**
Formulário utilizado pelo administrador para editar ou criar usuários. Além dos campos padrões relativos a classe User e a classe UserProfile existe um *hidden input* que contém o id do usuário em caso de edição, para que possamos validar as informações na hora de salvar no banco de dados.

O método clean é sobreescrito para validarmos a confirmação de senha e no caso de edição, temos que permitir o salvamento de um ramal ou email já existente.

class accounts.forms.**OnlyUserForm**
Formulário utilizado para que um usuário sem privilégios de administrador possa mudar seu email ou senha.

class accounts.forms.**PassResetForm**
Formulário utilizado para que um usuário possa receber sua senha esquecida no seu email. Apenas emails cadastrados são aceitos.

3.3.3 Views

accounts.views.**login**(*request*)
View que inicialmente mostra a tela de login para o usuário, caso o usuário entre na página inicial do projeto ou tente acessar alguma outra página através da url sem estar logado. O usuário ao submeter o formulário de login através de um método POST, fará com que a view tente autenticar esse usuário, e em caso de sucesso, guardará na sessão se o usuário é admininistrador, seu username e seu id e o redirecionará para a página principal. Em caso de falha, a view retorna uma mensagem de erro para o template de login.

accounts.views.**logout**(*request*)
Faz o logout do usuário logado e o redireciona para a página de login.

accounts.views.**settings**(*request*)
É a tela que lista todos os usuários do pabx-ip e permite que o administrador escolha qual usuário editar ou remover através de uma interface. Também possui um botão para a tela de cadastro de usuários. É importante lembrar que somente o usuário 'root' pode editar ou remover outros administradores.

accounts.views.**create**(*request*)
View que usa o UserForm para cadastrar um novo usuário no sistema. Somente acessível para administradores.

accounts.views.**edit**(*request*, *offset*)
View que usa o UserForm para editar um usuário pré-cadastrado no sistema. Através do offset passado pela url a view sabe o id do usuário que se deseja modificar. Somente acessível para administradores.

accounts.views.**delete**(*request*)
View usada para remover um usuário do sistema. Através do offset passado pela url a view sabe o id do usuário que se deseja deletar. Primeiramente exibe uma tela de confirmação, e em seguida caso haja confirmação por parte do administrador, a classe User e sua respectiva classe UserProfile são removidas do banco de dados.

accounts.views.**edit_self**(*request*)
View que o usa o OnlyUserForm para que um usuário comum logado para editar seu email ou senha.

accounts.views.**save_or_update**(*form*, *user=None*, *profile=None*)
Método que faz a associação entre um form e os objetos User e UserProfile. Caso os parâmetros user e profile não sejam vazios, a função interpreta como edição de usuário, e terá que fazer a busca dele no banco.

accounts.views.**password_reset**(*request*)
Retorna inicialmente o template do formulário PassResetForm onde o usuário deverá digitar um email cadastrado válido. Após a submissão do formulário, o sistema checa se existe um servidor smtp pré-cadastrado pelo administrador para envio de emails. Em caso positivo, o email é enviado, e retorna o template indicando uma mensagem de sucesso.

accounts.views.**encrypt**(*plaintext*)
Função que retorna a variável *plaintext* encriptada. Usada para salvar a senha encriptada dos usuários no banco.

accounts.views.**unencrypt**(*encrypted_password*)
Função que retorna a variável *encrypted_password* desencriptada. Usada para recuperar a senha dos usuários em caso de esquecimento.

3.3.4 Templates

Aqui serão listados os templates específicos utilizados por esse App, contidos na pasta "accounts/templates/"

- password_reset_form.html: utilizado para renderizar o formulário para recuperação de senha.
- password_reset_success.html: utilizado para mostrar a mensagem de sucesso na recuperação de senha.

3.3.5 Decorators

accounts.decorators.**is_admin**(*function*)
Checa se o User logado possui o atributo admin na respectiva classe UserProfile. Usado para limitar o acesso a certas Views que apenas administradores podem acessar.

3.4 App Groups

Este app é responsável por:

- Criar, editar e excluir grupos
- Associar permissões de usuários a um ou mais grupos

Todas as funcionalidades de grupos é apenas acessível para administradores.

Aviso: Esse App não foi finalizado ainda, portanto não atende ainda a todos os requisitos atuais do projeto. Porém no estágio em que ele está desenvolvido é possível rodar o App com as funcionalidades atuais sem problemas.

3.4.1 Modelos

Existe uma única classe *Group* definida aqui, que corresponde obviamente a um grupo. Um uso prático para essa classe é poder separar grupos de usuários por departamento, por exemplo: "grupo recepção" e "grupo gerência".

A associação entre grupos e usuários está definida no model *UserProfile* no qual o atributo *group* define o grupo do usuário. Portanto, um grupo possui vários usuários, mas um usuário só pode se associar a um grupo.

Os atributos da classe grupo são:

class groups.models.**Group**

•**name**: nome do grupo, usado apenas para identificá-lo.

- **can_call_ramal**: permissão dada como padrão para todos os grupos, mantemos aqui apenas para fins de visualização.
- **can_call_emergency**: permissão dada como padrão para todos os grupos, mantemos aqui apenas para fins de visualização.
- **can_call_fix**: indica se o grupo pode fazer ligações para fixo local.
- **can_call_mobile**: indica se o grupo pode fazer ligações para celular local.
- **can_call_ddd**: indica se grupo pode fazer ligações DDD.
- **can_call_ddi**: indica se grupo pode fazer ligações DDI.
- **can_call_0800**: indica se grupo pode fazer ligações 0800.
- **can_call_0300**: indica se grupo pode fazer ligações 0300.

A função *unicode* serve para retornar o nome do grupo no caso de imprimirmos algum objeto Group.

3.4.2 Formulários

class groups.forms.**GroupForm**

ModelForm que utiliza como modelo a classe *Group*. A partir dele é possível criar ou editar grupos, e a única instância feita por essa classe é desabilitar o checkbox de permissão para ligar para ramal e para emergência, pois estas permissões são dadas como padrão para qualquer grupo.

3.4.3 Views

groups.views.**index**(*request*)

Função que retorna a lista com os grupos cadastrados, utilizando o template *crud*. Esse template possui links para edição, remoção e criação de grupos.

groups.views.**create**(*request*)

View que utiliza o *GroupForm* para criar novos grupos.

3.5 App Skypelist

Esse app é responsável por:

- Criar, editar e remover contatos skype

Um contato Skype é apenas uma associação entre um nome de usuário Skype pré-cadastrado e um ramal do pabx, sendo que este ramal deve ser único, independente de ser um ramal normal ou "skype". A idéia por trás disso está em possibilitar o usuário a fazer ligações skype de um telefone normal, conectado ao PABX-IP.

Todas essas funcionalidades desse App está apenas acessível para Administradores.

3.5.1 Modelos

Existe uma uńica classe *skypeuser* definida aqui, que corresponde a associação entre nome de usuário skype e ramal.

Os atributos da classe *skypeuser* são:

class skypelist.skypeuser.**skypeuser**

- **ramal**: número do ramal.

•**username**: nome de usuário Skype.

3.5.2 Formulários

class skypelist.forms.**skypeform**

ModelForm que utiliza como modelo a classe *skypeuser*. Acrescenta um campo *hidden input* para tratar o caso de edição do usuário, para evitar a validação de campos com mesmos valores.

3.5.3 Views

skypelist.views.**index**(*request*)

Função que retorna a lista com os usuários skype cadastrados, utilizando o template *crud*. Esse template possui links para edição, remoção e criação.

skypelist.views.**create**(*request*)

View que utiliza o *skypeform* para criar novos usuários skype.

skypelist.views.**edit**(*request*, *offset*)

View que utiliza o *skypeform* para editar usuários skype pré-cadastrados. Através do offset passado pela url a view sabe o id do usuário skype que se deseja modificar.

delete(request,offset):

View usada para remover um usuário skype do sistema. Através do offset passado pela url a view sabe o id do usuário skype que se deseja deletar.Primeiramente exibe uma tela de confirmação, e em seguida caso haja confirmação por parte do administrador, a classe *skypuser* é removida do banco de dados.

3.6 App Smtp

Esse App é responsável pelo cadastro de um servidor smtp que será usado pelo PABX-IP para enviar emails diversos. A partir de um servidor smtp configurado para o PABX-IP é possível, por exemplo, mandar emails a partir de uma conta cadastrada no gmail.

Apenas um único servidor smtp é permitido, sendo que somente o Administrador pode alterar suas configurações.

3.6.1 Modelos

Existe uma única classe *server* definida aqui, que corresponde aos dados do servidor smtp.

Os atributos da classe *server* são:

class smtp.server.**server**

•**url**: url para o servidor smtp.

•**port**: porta do servidor smtp.

•**username**: nome de usuário do servidor smtp.

•**password**: senha do servidor smtp.

3.6.2 Formulários

class smtp.forms.**smtpform**
Nesse *ModelForm* cujo modelo é a classe *server*, acrescentamos o campo de confirmação de senha e a validação da mesma. Esse formulário é utilizado tanto para cadastro como edição do servidor smtp.

3.6.3 Views

smtp.views.**index**(*request*)
Função que retorna os dados do servidor smtp e imprime num template *crud*. Se o servidor não estiver configurado ainda, uma mensagem será exibida. Essa tela ainda possui links para cadastrar ou editar um servidor smtp.

smtp.views.**configure**(*request*)
Função que permite o cadastro ou a edição do servidor smtp, utilizando o *smtpform*.

3.7 Templates Padrão

Templates são usados pelo Django para exibir o conteúdo em questão para o usuário, ou seja, é o que o usuário vê de fato.

Os templates padrão são os arquivos html contidos na pasta *templates* logo na raiz do projeto. Apenas alguns arquivos dentre outros contidos nessa mesma pasta são utilizados no projeto, sendo que os não usados são arquivos criados automaticamente pelo Pinax.

Serão listados aqui os templates utilizados:

- base.html
- crud.html
- form_create.html
- form_delete.html
- form_success.html
- login.html

3.7.1 base.html

É o template base do projeto, que utiliza a mesma estrutura do *fluid layout* do Twitter bootstrap, com alguma modificação para fixar a barra superior. Aqui fica definido o *menu*, o *header* e o *footer*. O conteúdo que eventualmente será carregado na parte do meio, será parte de outro template "filho" deste.

A variável "highlight" deve ser definida em cada view para manter o link do menu destacado, na seção que o usuário estiver navegando. Assim como a variável "title" que define o valor da tag html *title*.

3.7.2 crud.html

Template que extende o template *base.html* e é responsável por criar as tabelas que darão acesso as operações CRUD (acrônimo de Create, Read, Update e Delete em inglês).

3.7.3 form_create.html

Template que extende o template *base.html* e é responsável por montar a estrutura dos formulários utilizados por criação e edição de dados.

Aqui também existe um script Javascript que facilita a criação do botão "cancelar", pois o link é passado pela *view* correspondente.

3.7.4 form_delete.html

Template que extende o template *base.html* e é responsável por montar a estrutura da tela de confirmação de remoção de algum dado.

Aqui também existe um script Javascript que facilita a criação do botão "cancelar", pois o link é passado pela *view* correspondente.

3.7.5 form_success.html

Template que extende o template *base.html* e é responsável por montar a estrutura da tela de sucesso no caso de alguma ação (criação,edição ou remoção) ter sido executada com sucesso.

Aqui também existe um script Javascript que facilita a criação do botão "voltar", pois o link é passado pela *view* correspondente.

3.7.6 login.html

Template que é responsável por montar o formulário de login do usuário.

3.8 Bibliotecas externas

Nessa seção listaremos as bibliotecas auxiliares ao projeto, que já vem previamente instaladas e configuradas. O desenvolvedor deve ter conhecimento delas caso precise utilizá-las futuramente.

Cada biblioteca utilizada possui também uma documentação própria, que pode elucidar dúvidas mais específicas.

3.8.1 Twitter Bootstrap

http://twitter.github.com/bootstrap/

Utilizado como base da identidade visual do projeto. Embora o Pinax já venha com um template padrão usando o Twitter Bootstrap, foi preferível começar um novo template para fins de customização.

3.8.2 jQuery

http://jquery.com/

É o framework Javascript utilizado no projeto.

3.8.3 Sphinx

http://sphinx.pocoo.org/

Biblioteca utilizada para gerar essa documentação, cujo código fonte se encontra na pasta "doc" dentro da pasta raiz do projeto.

3.9 Futuras implementações

A base do projeto PABX-IP se encontra na manipulação de arquivos, por isso, sugerimos o uso da biblioteca *fileinput* para tal: http://docs.python.org/library/fileinput.html

Conforme visto no StackOverflow: http://stackoverflow.com/questions/4746190/find-and-replace-within-a-text-file

Índice de Módulos do Python

a

g

s

Índice