PABX IP Documentation

Versão 0.1

Fabio Hiroki

13/03/2012

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 começar, 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

3.1.1 1. Introdução

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.

3.1.2 2. 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

3.1.3 3. 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.

3.1. Instalação do git e clone do repositório:

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

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

3.2. 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

$ sudo apt-get install python-pip

$ sudo pip install virtualenv

$ virtualenv pabx-env

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

3.3. 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
```

3.4. Instalação do Django:

Django é o framework web utilizado no projeto.

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

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

3.5. 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
```

3.6. 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*.

3.2 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.

Para que o desenvolvedor não fique confuso com a quantidade de arquivos do projeto, será listado aqui os apps desenvolvidos de fato (cada app corresponde a uma pasta na raiz do projeto):

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

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

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*
- Modelos: relativo ao arquivo *models.py*
- Views: relativo ao arquivo *views.py*
- Decorators: relativo ao arquivo *decorators.py* (opcional)
- Templates: relativo aos arquivos dentro da pasta "templates" (opcional)

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.

O mesmo vale para os *decorators*, nem todos os apps possuem o arquivo *decorators.py* em seu diretório

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 ela se confirme, 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 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`**

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 groupos.

groups.views.**create**(*request*)
View que utiliza o *GroupForm* para criar novos grupos.

3.5 Bibliotecas externas

3.5.1 1. Introdução

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.

3.5.2 2. 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.

3.5.3 3. jQuery

http://jquery.com/

É o framework Javascript utilizado no projeto.

3.6 Futuras implementações

Índice de Módulos do Python

a

g

Índice