

# Apostila de Desenvolvimento Web com Django

Colocando suas idéias na web de forma prática e rápida

Centro de Treinamento da Novatec  
Instrutor: Julio Cesar Eiras Melanda  
2016

## Índice

Capítulo 1.....	2
Preparação.....	2
O que é Django?.....	2
Pip e Virtualenv.....	3
PIP.....	3
Virtualenv.....	3
Instalando o Django.....	4
Django.....	4
Preparando o ambiente de desenvolvimento.....	4
Criando sua primeira aplicação.....	5
Entendendo o settings.py.....	7
Iniciando um servidor.....	11
Models.....	13
Criar o primeiro modelo.....	13
Como funcionam os fields.....	15
O que é classe Meta?.....	15
Gerando o banco de dados.....	16
Admin.....	17
O que é o Admin.....	17
Registrando modelos no Admin.....	17
Melhorando o Admin.....	19
Views.....	22
O que são views?.....	22
Usando as Views do Django.....	22
Criando suas próprias views.....	23
Urls.....	24
Mapeando URLs.....	24
Passando argumentos nas URLs.....	29
Templates.....	30

O que é o sistema de templates do Django.....	30
Como utilizar o Twitter Bootstrap?.....	34
Formulários.....	39
Os Forms do Django.....	39
Trabalhando com ModelForms.....	45
Banco de dados.....	48
Fazendo queries com o ORM do Django.....	48
Fazendo buscas mais avançadas.....	49
Inserindo relações nos modelos.....	49
Buscas avançadas.....	52
Migrações.....	53
Entendendo os arquivos de migração.....	53
Testes.....	54
TDD.....	55
Testes unitários.....	55
Criando uma página com TDD.....	57
Mock.....	59
Arquivos estáticos.....	60
O que são arquivos estáticos.....	60
Como o Django gerencia arquivos estáticos.....	60
Deploy.....	63
Controle de versão.....	63
O que é o Git.....	63
Comandos básicos.....	63
Criando uma gear no Openshift.....	64
Colocando seu projeto no Openshift.....	67

## Capítulo 1

### Preparação

Na primeira parte deste capítulo você aprenderá o que é o Django. Na segunda parte aprenderá a usar o pip e o virtualenv. Na última parte, vamos instalar o Django no nosso ambiente. Ao final deste capítulo você deverá saber o que é o Django, como criar um ambiente Python independente e instalar pacotes neste ambiente, inclusive o Django.

### O que é Django?

Segundo o site oficial do framework, o "Django torna mais fácil criar aplicações Web mais rapidamente e com menos código".

Django é um framework Web feito em Python que procura prover aos desenvolvedores funcionalidades que encapsulam grande parte do trabalho de criar aplicações web, permitindo que o desenvolvedor foque na criação da aplicação.

Com Django é possível fazer todo tipo de aplicação web desde gerenciadores de conteúdo a plataformas de computação científica.

Entre os principais exemplos de sites que usam Django temos:

- Instagram

- Disqus
- Open Stack
- Mozilla
- Pinterest

## Pip e Virtualenv

### PIP

*Pip* é um sistema de gerenciamento de pacotes Python.

Seu intuito é gerenciar essas operações de forma simples e direta, de forma similar ao *yum* do Fedora/RedHat ou o *apt-get* dos sistemas baseados em debian.

A forma mais fácil de instalar o *Pip* é através do seu gerenciador de pacotes, geralmente o nome do pacote é *python-pip*.

Para instalar pacotes, usamos o comando ***pip install pacote*** e para remover ***pip uninstall pacote***. É possível também buscar por pacotes com ***pip search pacote*** e verificar o que está instalado com ***pip freeze***.

Uma das funções mais importantes do pip é para centralizar o gerenciamento de dependências do seu projeto. Isto faz-se criando um arquivo de texto com o nome *requirements.txt*, por exemplo, e colocando dentro dele os nomes dos pacotes e também comandos de restrição de versão.

Vejamos um exemplo:

```
Django>=1.9.4
```

```
Pillow==3.1.1
```

O formato usado no arquivo *requirements.txt* é o mesmo formato de saída do comando ***pip freeze***. Logo vamos ver pra que podemos usar isto.

### Virtualenv

*Virtualenv* é um pacote python feito para isolar ambientes python, permitindo ao desenvolvedor ter várias versões das bibliotecas instaladas sem que haja conflito entre elas. Isto é muito útil para gerenciar as dependências de um projeto, fazer teste com versões novas de bibliotecas, tudo isto sem causar danos no ambiente Python do sistema.

O virtualenv pode ser instalado via pip ou com o gerenciador de pacotes.

Para criar um ambiente com *virtualenv* execute o seguinte comando:

```
virtualenv nome_do_virtualenv
```

Será criado um diretório com o nome *nome\_do\_virtualenv* no diretório onde você executar o comando. Dentro deste diretório é criada uma estrutura de

diretórios muito similar à estrutura padrão da raiz dos sistemas Linux. Ali dentro você encontrara scripts e o interpretador Python no diretório bin, as libs instaladas em libs, etc.

O ambiente criado pelo *virtualenv* já vem com a versão do python que você tiver usado para executar o comando e o pip instalados.

Para usar seu ambiente virtual, devemos alterar as variáveis de ambiente de nossa sessão de shell para reconhecer o ambiente virtual como sendo o ambiente python em uso. Ative seu ambiente virtual assim:

```
source nome_do_virtualenv/bin/activate
```

ou assim:

```
. nome_do_virtualenv/bin/activate
```

Ao executar este comando, você notará que seu prompt que normalmente mostrava algo do tipo

```
usuario@localhost ~:$
```

Agora mostrará:

```
(nome_do_virtualenv)usuario@localhost ~:$
```

Assim, você consegue saber se o ambiente virtual está ativado e qual é ele.

## Instalando o Django

Agora que já sabemos instalar o *pip* e o *virtualenv*, temos duas formas para instalar o *django*.

Podemos instalar usando o gerenciador de pacotes do sistema, o que instalará o *django* no sistema, ou podemos instalar com o *pip*. Se usarmos o *pip* do sistema, fará o mesmo efeito de instalar o *django* com o gerenciador de pacotes do sistema, o que não nos permite ter várias versões, então, vamos instalá-lo no nosso ambiente virtual.

```
pip install Django
```

## Django

### Preparando o ambiente de desenvolvimento

Bom, agora que você já sabe criar ambientes virtuais e instalar o Django vamos preparar o ambiente para nosso projeto.

Pra começar, vamos precisar do Python3.

Porque Python3?

Porque como diz nosso mestre Guido Van Rossum, Python3 é a versão atual e futura do Python e em geral, o Python2 deve ser usado somente para

aplicações legadas ou quando depender de bibliotecas que não tenham sido portadas ou não tenham similar no Python3 e você não tenha tempo de escrevê-las.

Então, vamos instalar o Python3. Se você usar fedora 23, ele já estará instalado. Em outras distribuições, você provavelmente conseguirá instalar através do repositório de pacotes. Em último caso, se não conseguir instalar com o gerenciador de pacotes da sua distribuição, você pode instalar compilando o interpretador.

Vamos agora criar nosso ambiente virtual.

Escolha um diretório para colocarmos o ambiente virtual. Qualquer diretório onde tenha permissão de escrita serve.

```
virtualenv minhas_tarefas1
```

Agora vamos ativar nosso ambiente e instalar as dependências da aplicação.

```
source minhas_tarefas/bin/activate
```

Com o ambiente ativado, vamos instalar o Django.

```
pip install django
```

Para editarmos os arquivos do projeto, podemos usar qualquer editor de textos. Aqui usaremos o editor chamado Atom, um editor cheio de boas *features* e que é livre e gratuito, sem contar que é desenvolvido pelos desenvolvedores do *GitHub*.

## Criando sua primeira aplicação

Agora que temos nosso ambiente preparado, vamos criar nosso projeto Django.

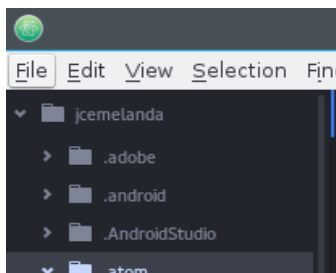
Entre no diretório que escolheu para colocar o projeto.

```
django-admin startproject minhas_tarefas
```

Vamos agora abrir o diretório do nosso projeto no Atom.

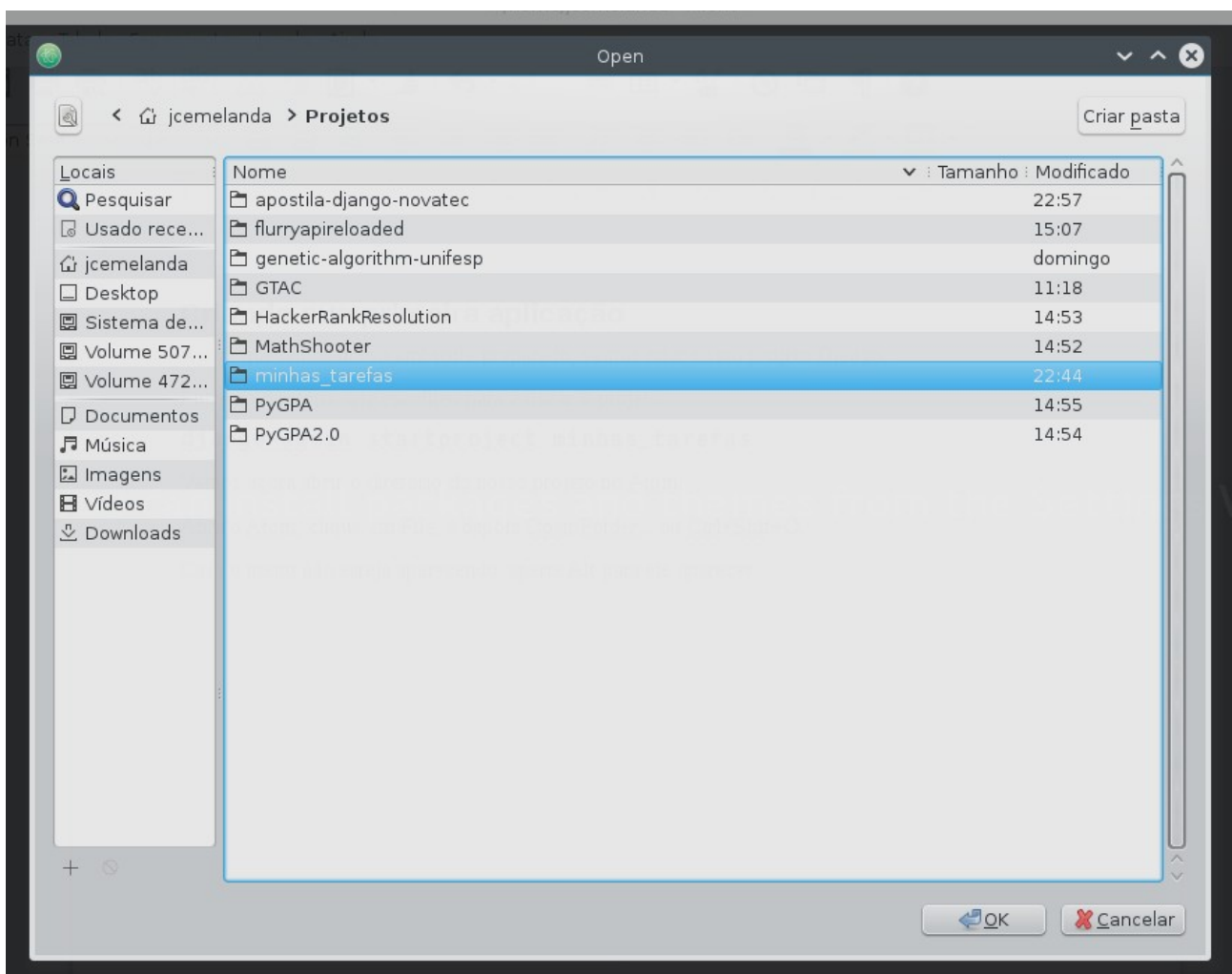
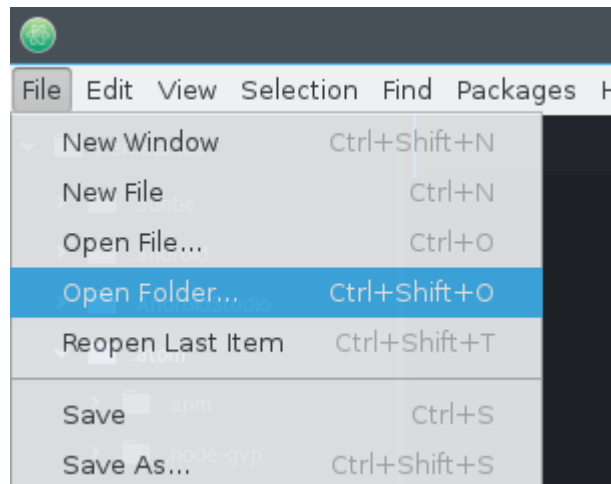
Abra o Atom, clique em File, e depois Open Folder... ou Ctrl+Shif+O.

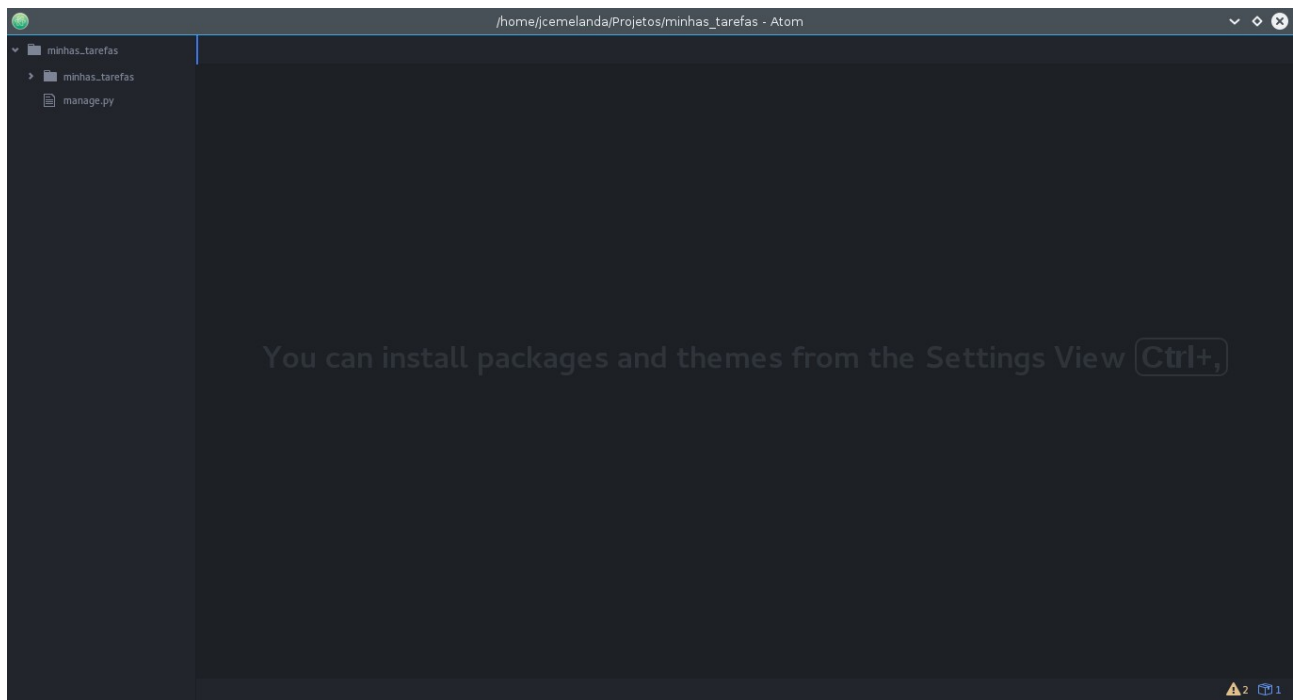
Caso o menu não esteja aparecendo, aperte Alt para ele aparecer.



---

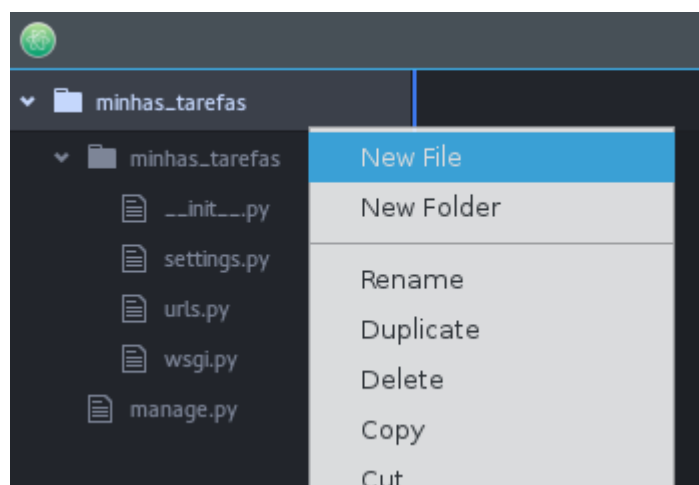
<sup>1</sup> O comando pode ser `virtualenv-3.4` no lugar de simplesmente `virtualenv` para instalar o ambiente correto.





Agora vamos criar nosso arquivo de *requirements*.

Clique com o botão direito na pasta principal do projeto e selecione **New File**.

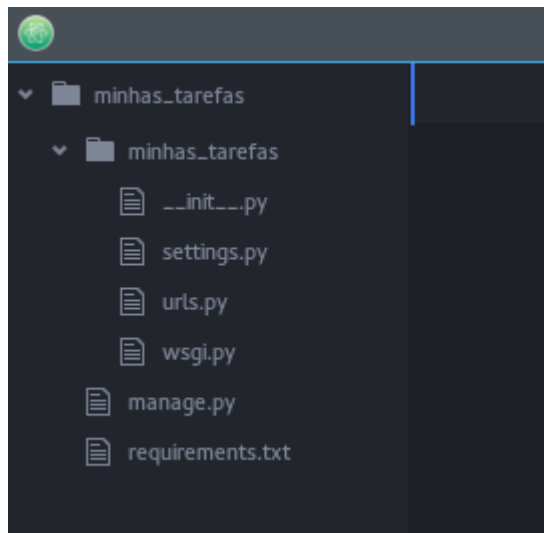


Chame o arquivo de *requirements.txt*. Neste arquivo colocaremos a saída do comando ***pip freeze***, o que aqui foi:

**Dj**ango==1.9.4

## Entendendo o settings.py

Vocês devem ter reparado que o Django criou alguns arquivos pra gente numa pasta chamada ***minhas\_tarefas***.



Vamos dar uma olhada no arquivo ***settings.py***. Ele é responsável por guardar as configurações do Django, e vamos mexer bastante nele durante o desenvolvimento de nossa aplicação.

```
"""
```

```
Django settings for minhas_tarefas project.
```

```
Generated by 'django-admin startproject' using Django 1.9.4.
```

```
For more information on this file, see  
https://docs.djangoproject.com/en/1.9/topics/settings/
```

```
For the full list of settings and their values, see  
https://docs.djangoproject.com/en/1.9/ref/settings/  
"""
```

```
import os
```

```
# Build paths inside the project like this: os.path.join(BASE_DIR,  
...)
```

```
BASE_DIR =  
os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
```

```
# Quick-start development settings - unsuitable for production
```

```
# See
```

```
https://docs.djangoproject.com/en/1.9/howto/deployment/checklist/
```

```
# SECURITY WARNING: keep the secret key used in production secret!
```

```
SECRET_KEY = '!c+%)fojxufsmhmao#w5a+-bu1q1_zv+%fa7$1tr*(4k-y)f7f'
```

```
# SECURITY WARNING: don't run with debug turned on in production!
```

```
DEBUG = True
```

```
ALLOWED_HOSTS = []
```



```

# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]

MIDDLEWARE_CLASSES = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',

    'django.contrib.auth.middleware.SessionAuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]

ROOT_URLCONF = 'minhas_tarefas.urls'

TEMPLATES = [
    {
        'BACKEND':
'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',

'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]

WSGI_APPLICATION = 'minhas_tarefas.wsgi.application'

# Database
# https://docs.djangoproject.com/en/1.9/ref/settings/#databases

DATABASES = {
    'default': {

```

```

        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}

# Password validation
# https://docs.djangoproject.com/en/1.9/ref/settings/#auth-
password-validators

AUTH_PASSWORD_VALIDATORS = [
    {
        'NAME':
'django.contrib.auth.password_validation.UserAttributeSimilarityVa
lidator',
    },
    {
        'NAME':
'django.contrib.auth.password_validation.MinimumLengthValidator',
    },
    {
        'NAME':
'django.contrib.auth.password_validation.CommonPasswordValidator',
    },
    {
        'NAME':
'django.contrib.auth.password_validation.NumericPasswordValidator'
,
    },
]

# Internationalization
# https://docs.djangoproject.com/en/1.9/topics/i18n/

LANGUAGE_CODE = 'en-us'

TIME_ZONE = 'UTC'

USE_I18N = True

USE_L10N = True

USE_TZ = True

# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/1.9/howto/static-files/

STATIC_URL = '/static/'

```

Logo no início do arquivo, temos informações de onde buscar mais informações sobre os *settings*.

Como não é recomendado usar caminhos relativos neste arquivo, logo de cara é declarada a variável **BASE\_DIR**, que será usada para referenciar pastas e arquivos.

**SECRET\_KEY** é uma string que deve ser longa e conter uma série imprevisível de caracteres. Ela é usada para assinatura digital em diversos momentos no fluxo das aplicações, como nas sessões, nas mensagens, etc.

**DEBUG** é uma flag para indicar se os erros que ocorrerem devem mostrar os detalhes e *stacktrace* na tela. Esta *flag* não deve ser verdadeira em produção. O mesmo ocorre para a *flag* **TEMPLATE\_DEBUG**. Muitas vezes alteramos esta *flag* para ficar assim:

```
TEMPLATE_DEBUG = DEBUG
```

**ALLOWED\_HOSTS** é usado para identificar quais os endereços que podem acessar a aplicação. Por enquanto, manteremos assim.

**INSTALLED\_APPS** contém uma tupla com todas as aplicações que o projeto contém. Por padrão, vem habilitadas as aplicações de *admin*, *auth*, *contenttypes*, *sessions*, *messages* e *staticfiles*. É aqui que vamos acrescentar novos *apps* ao nosso projeto.

**MIDDLEWARE\_CLASSES** contém uma tupla de classes que tratam os requests e acrescentam comportamentos específicos. Por exemplo, **CsrfViewMiddleware** impede ataques do tipo *Cross Site Request Forgery*, e **AuthenticationMiddleware** permite tratar a autenticação no sistema.

**ROOT\_URLCONF** indica qual é o arquivo principal de configuração de urls.

**WSGI\_APPLICATION** contém a aplicação WSGI que serve o site

**DATABASES** indica a configuração do banco de dados. Aqui é possível indicar qual o SGBD, senha, porta, host, nome do banco, etc.

**LANGUAGE\_CODE** indica o idioma da aplicação. Vamos mudar para nosso site ficar em português.

```
LANGUAGE_CODE = 'pt-br'
```

**TIME\_ZONE** indica o timezone da aplicação. Como vamos usar somente nosso horário local, vamos mudar também para ficar de acordo.

```
TIME_ZONE = 'America/Sao_Paulo'
```

**USE\_I18N**, **USE\_L10N** e **USE\_TZ** são respectivamente para ativar internacionalização, localização e timezones.

**STATIC\_URL** é uma variável para indicar qual url deve ser usada para acessar os arquivos estáticos.

## Iniciando um servidor

Agora que já sabemos o que é cada entrada do `settings.py`, vamos ver nosso sistema no ar como está agora.

Com o ambiente virtual ativado, e na pasta do projeto, execute o comando

**`python manage.py migrate`**

Será criado um banco de dados em *sqlite3* com os modelos básicos do sistema.

Agora vamos criar o superusuário do sistema.

**`python manage.py createsuperuser`**

Geralmente por simplificação, coloco nome *admin* e senha *123*, mas fique à vontade para usar como quiser, só não faça isto em produção. O e-mail não é obrigatório.

Agora, com banco criado e super usuário, vamos executar o servidor de desenvolvimento.

Vale ressaltar que não deve-se usar o servidor de desenvolvimento em produção. Existem diversos problemas de desempenho e segurança ao fazer isto. Este servidor foi criado para testarmos a aplicação enquanto desenvolvemos, e só.

**`python manage.py runserver`**

Vamos agora acessar então o sistema. Acesse no browser

<http://localhost:8000/>

**It worked!**

Congratulations on your first Django-powered page.

Of course, you haven't actually done any work yet. Next, start your first app by running `python manage.py startapp [app_label]`.

You're seeing this message because you have `DEBUG = True` in your Django settings file and you haven't configured any URLs. Get to work!

Se aparecer uma tela assim, deu tudo certo

Agora vamos acessar a área de administração do sistema.

<http://localhost:8000/admin/>



Você deve ver uma tela assim.

Digite o usuário e senha do sistema e acesse o admin.



Esta é a tela de *Admin* padrão do Django. Aqui conseguimos criar e editar grupos e usuários do sistema.

Falaremos mais de *admin* no futuro.

## Models

O Django tem um sistema de models muito direto e intuitivo. É possível representar as entidades do sistema com todas suas relações abstraindo grande parte da criação do banco de dados da aplicação.

Assim, os models do Django devem ser criados com uma visão de objetos, não como tabelas no banco.

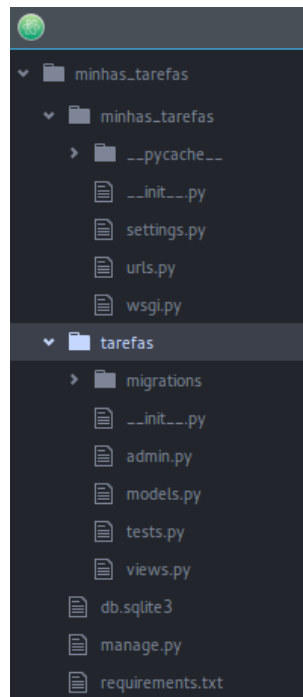
## Criar o primeiro modelo

Vamos então começar a criar nosso sistema de gerenciamento de tarefas.

Para tal criamos primeiro uma nova aplicação dentro de nosso projeto.

```
python manage.py startapp tarefas
```

Este comando vai criar uma aplicação dentro de nosso projeto chamada tarefas.



Veja que na nova pasta criada temos os arquivos *admin.py*, *models.py*, *tests.py* e *views.py*. São estes arquivos, em geral que vamos editar para criar realmente nossa aplicação.

Abra o arquivo *models.py*.

Ele deve estar assim:

```
from django.db import models
# Create your models here.
```

Os modelos Django são subclasses da classe Model do Django. Nela vamos especificar os modelos da nossa aplicação, definindo seus campos e métodos que sejam necessários.

Vamos alterar a primeira linha para ficar assim:

```
from django.db.models import Model
```

Agora criamos nossa classe Tarefa

```
class Tarefa(Model):
```

Num objeto tarefa vamos precisar de data de criação, nome, descrição, status e data de execução. Data de criação é um *DateTimeField*, nome é um *CharField*, descrição é um *TextField*, status é um *BooleanField* e finalmente, a data de execução é outro *DateTimeField*.

Vamos então importá-los em nosso *models.py*.

Nossos *imports* agora ficam assim:

```
from django.db.models import Model  
from django.db.models import BooleanField  
from django.db.models import CharField  
from django.db.models import DateTimeField  
from django.db.models import TextField
```

DICA: Vocês vão achar diversos materiais que importam somente o *models* e usam estas classes como *models.Model*, *models.CharField*, etc. Não usaremos esta forma aqui para evitarmos criar o vício de usar este tipo de acesso a classes e métodos. Como o Python é uma linguagem dinâmica, cada vez que fazemos *models.CharField*, por exemplo, o Python vai procurar no módulo *models* a classe *CharField*. Se importamos diretamente a classe, esta busca é feita somente uma vez, quando o módulo é carregado. Isto pode parecer uma otimização desnecessária, mas conforme nosso sistema cresce e especialmente se usarmos este tipo de chamadas dentro de laços, conseguimos ter algum ganho de desempenho.

Chega a hora então de criarmos efetivamente os *fields* no nosso modelo.

Dentro da classe Tarefa vamos acrescentar as seguintes linhas:

```
data_de_criacao = DateTimeField(auto_now_add=True,  
                                verbose_name='data de criação')  
data_de_execucao = DateTimeField(verbose_name='data de execução')  
nome = CharField(max_length=200, verbose_name='nome')  
descricao = TextField(verbose_name='descrição')  
status = BooleanField(default=False, verbose_name='status')
```

Agora já temos um modelo de tarefa com todos os campos que precisamos a princípio.

## Como funcionam os fields

Os *Fields* de um *Model* são classes que vão prover para o Django várias coisas, como o tipo do dado que deverá ser criado no banco de dados, o nome da tabela, relações entre as tabelas, os tipos de *widgets* que devem aparecer em formulários e no *admin*, validações de dados, etc.

Quando instanciarmos objetos destas classes, poderemos acessar estes atributos como se fosse atributos de classe comuns, retornando objetos dos tipos *boolean*, *string*, *int*, e não objetos *fields*.

## O que é classe Meta?

Nos models do Django podemos declarar uma classe interna especial que serve para colocarmos parâmetros de configuração do modelo, como o nome que aparecerá no admin, parâmetros de ordenação dos modelos quando é feita uma busca, entre várias outras configurações possíveis.

Esta classe é chamada sempre de *Meta*, e é criada da seguinte forma:

```
class Tarefa(Model):
    data_de_criacao = DateTimeField(auto_now_add=True,
                                    verbose_name='data de criação')
    data_de_execucao = DateTimeField(verbose_name='data de execução')
    nome = CharField(max_length=200, verbose_name='nome')
    descricao = TextField(verbose_name='descrição')
    status = BooleanField(default=False, verbose_name='status')

    class Meta:
        ordering = ['-data_de_criacao']
```

Neste caso, usamos o atributo *ordering*, para dizer que as tarefas devem estar ordenadas por ordem decrescente de data de criação (o “-” indica que será decrescente).

## Gerando o banco de dados

Finalmente, com nosso modelo criado, vamos criar a migração que persistirá nossos dados no banco de dados.

Para podermos criar as migrações, precisamos primeiro registrar a aplicação como uma aplicação instalada no nosso *settings.py*.

Lembram-se do `INSTALLED_APPS`?

Acrescente o nome de nossa aplicação lá. Ficará assim:



```
INSTALLED_APPS = (  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'tarefas',  
)
```

Por convenção, existe uma ordem para colocarmos estas aplicações, e esta ordem é: Primeiro, aplicações Django, em seguida, aplicações de terceiros, e somente no final, as suas aplicações.

Execute no terminal:

```
python manage.py makemigrations
```

Este comando criará sempre novos arquivos de migração para os modelos que tiverem sido alterados.

Agora vamos aplicar as migrações:

```
python manage.py migrate
```

Agora nosso banco está de acordo com os modelos da nossa aplicação.

## Admin

Já criamos nossa aplicação, com modelo e banco de dados, chega então o momento de registrarmos este modelo no admin para podermos criar nossas tarefas.

## O que é o Admin

O Admin é uma interface de CRUD do Django que é gerada automaticamente a partir dos modelos definidos.

Cada um dos fields dos modelos podem se tornar campos nos formulários de criação/edição dos estes modelos.

Apesar de gerado automaticamente, o Admin do Django é altamente customizável, permitindo não só escolher quais os campos que vão aparecer, os campos que aparecerão na lista de objetos, agrupamentos de objetos, e muito mais.

## Registrando modelos no Admin

Para registrar o modelo no admin, precisamos abrir o arquivo *admin.py* de dentro de nossa aplicação. Vamos primeiro corrigir os imports para ficarem de acordo com nossa estratégia de evitar lookups desnecessários.

```
from django.contrib.admin import site
```

Neste caso, não vamos conseguir importar somente o método *register*, então importamos *site* para usar seu método *register*.

Importe a classe *Tarefa* no módulo de Admin, e em seguida registre a classe *Tarefa* no admin com o seguinte comando.

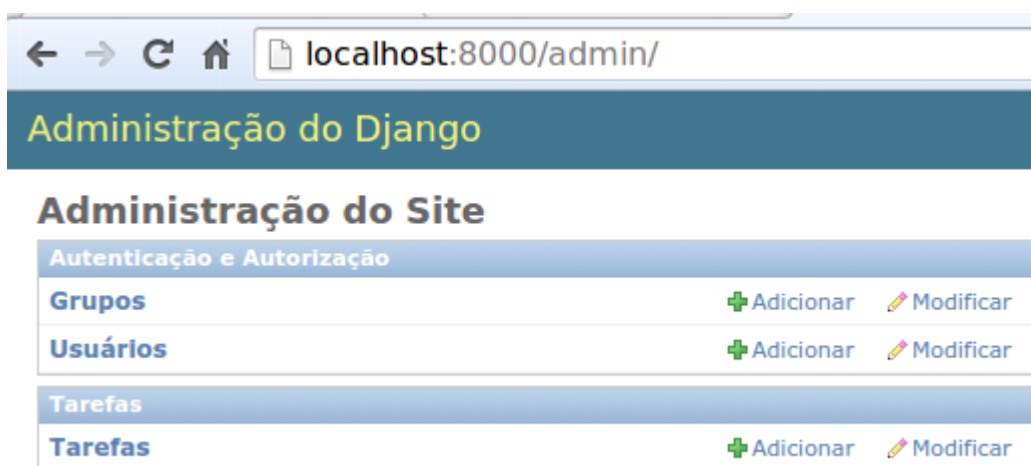
```
from tarefas.models import Tarefa
site.register(Tarefa)
```

Agora estamos prontos! Vamos subir o servidor e ver como está a nossa interface de administração.

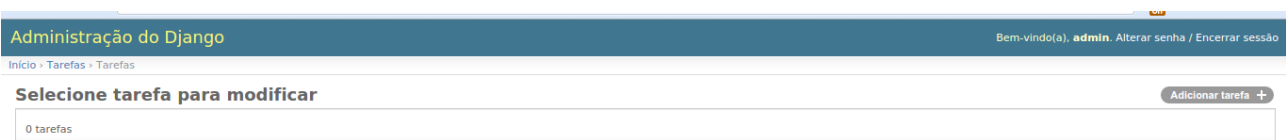
No terminal execute:

```
python manage.py runserver
```

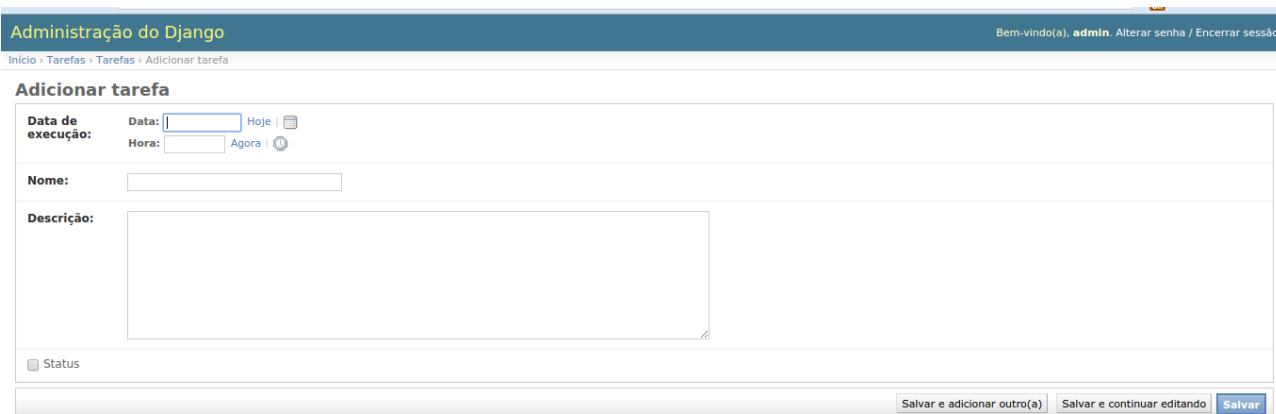
E acesse <http://localhost:8000/admin>



Veja que agora nossa aplicação *Tarefas* aparece no *admin*, e o modelo também. Clique em *Tarefas* para irmos para sua página específica.



Não temos ainda nenhuma tarefa cadastrada, vamos fazê-lo agora. Clique em **adicionar tarefa**.



The screenshot shows the 'Adicionar tarefa' (Add task) form in the Django Admin interface. The form is titled 'Adicionar tarefa' and is located under the breadcrumb 'Início > Tarefas > Tarefas > Adicionar tarefa'. The form fields include: 'Data de execução:' with 'Data:' and 'Hora:' dropdowns, 'Nome:' text input, 'Descrição:' text area, and a 'Status' checkbox. At the bottom right, there are three buttons: 'Salvar e adicionar outro(a)', 'Salvar e continuar editando', and 'Salvar'.

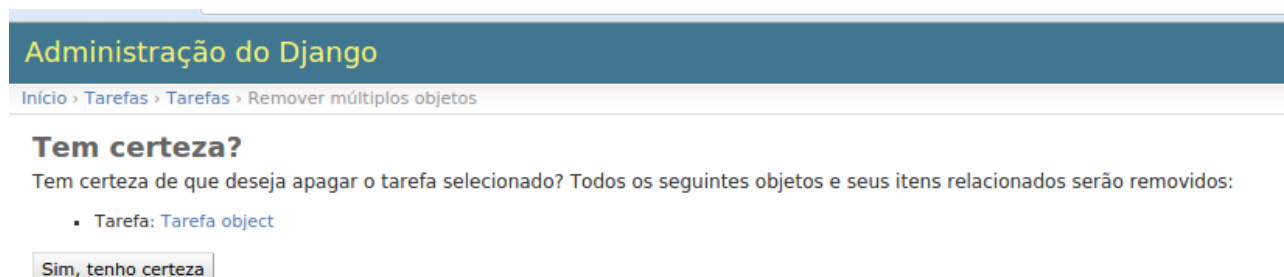
Preencha os dados e salve uma tarefa.



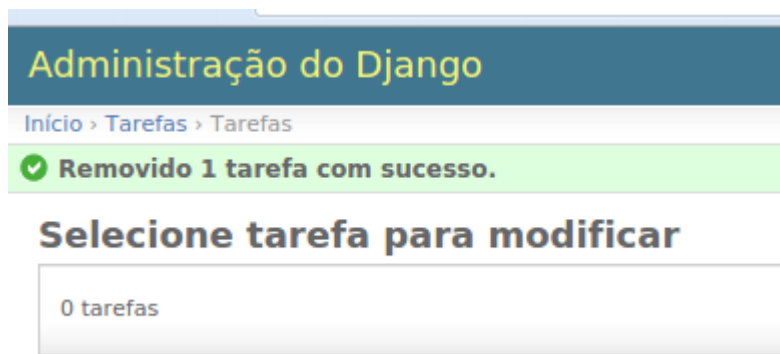
The screenshot shows the Django Admin interface after saving a task. A green success message at the top reads: 'Tarefa "Tarefa object": adicionado com sucesso.' Below this, the heading 'Selecione tarefa para modificar' is displayed. A table lists the tasks, with 'Tarefa object' selected. The table has columns for 'Ação:' (Action) and 'Fazer' (Do). The 'Ação:' column shows a dropdown menu with '-----' and a 'Fazer' button. The 'Fazer' column shows '0 de 1 selecionados'. Below the table, it says '1 tarefa'.

Está aí nossa tarefa salva, mas não está muito bom ainda né, logo mais vamos melhorar esta visualização.

Primeiro, vamos alterar deletar a tarefa. Clique no *checkbox* ao lado da tarefa, e em ação, selecione **Remove** tarefas selecionadas.



The screenshot shows the Django Admin confirmation dialog for deleting a task. The heading is 'Tem certeza?' (Are you sure?). The text below reads: 'Tem certeza de que deseja apagar o tarefa selecionado? Todos os seguintes objetos e seus itens relacionados serão removidos:'. Below this, there is a list of the selected task: 'Tarefa: Tarefa object'. At the bottom, there is a button labeled 'Sim, tenho certeza' (Yes, I am sure).



Agora brinque um pouco com o admin do Django, insira tarefas, edite, delete, entenda bem como ele funciona.

## Melhorando o Admin

Primeira coisa que vamos alterar, é que na nossa tarefa, a descrição é obrigatória. Se ainda não havia notado, verifique tentando salvar uma tarefa sem descrição.

Para que a descrição seja um campo opcional, no nosso modelo, vamos alterar a linha que cria o campo descrição acrescentando ao final da chamada para o *TextField* *null=True* e *blank=True*. Ficará assim:

```
descricao = TextField(verbose_name='descrição', null=True, blank=True)
```

Assim, estamos dizendo ao Django que o banco de dados deve aceitar null neste campo, e dizendo que no admin e em forms que usem este modelo, este campo pode ficar em branco,

Vamos gerar uma nova migração para aplicar esta alteração no banco

```
python manage.py makemigrations
```

```
python manage.py migrate
```

Agora, faça o teste novamente salvando uma tarefa sem descrição, e ela será salva corretamente.

Outra alteração que vamos fazer no modelo é com relação ao campo status. Ele aparece no *admin* como um *checkbox*, o que não faz muito sentido, vamos então alterar o *verbose\_name* do campo status para finalizado.

```
status = BooleanField(default=False, verbose_name='finalizado')
```

Isto já é suficiente para alterar o nome que aparece no admin.

Agora vamos alterar efetivamente o admin.

Abra seu arquivo `admin.py`. Agora ele deve estar assim:

```
from django.contrib.admin import site
from tarefas.models import Tarefa
# Register your models here.
site.register(Tarefa)
```

Antes do *register*, vamos criar uma classe *TarefasAdmin* que estende *ModelAdmin*. Assim, vamos primeiro importar *ModelAdmin*.

```
from django.contrib.admin import ModelAdmin
```

Por convenção, este *import* deve ser feito antes do *import* da classe *Tarefa*, pois *ModelAdmin* é uma classe do Django e *Tarefa* é do seu projeto.

Agora que temos nossa classe importada crie a classe *TarefaAdmin*.

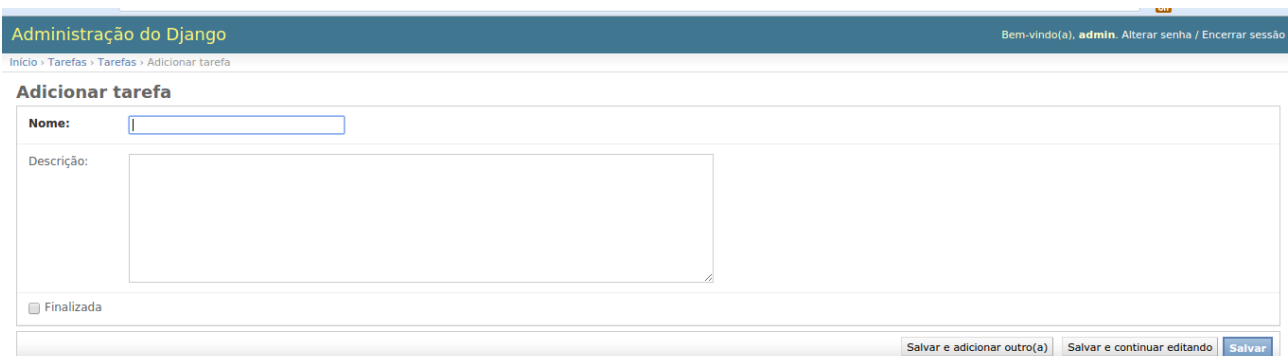
Dentro desta classe crie o atributo *fields*, que receberá uma lista com os nomes dos campos que queremos que apareçam.

```
class TarefaAdmin(ModelAdmin):
    fields = ['nome', 'descricao', 'status']
```

Agora com a classe criada, vamos alterar a forma como registramos nossa classe, passando também a classe de *admin*, informando assim ao Django que as configurações do *admin* desta classe estão nesta classe de *admin*.

```
site.register(Tarefa, TarefaAdmin)
```

Salve o arquivo e já podemos ver como ficou nosso *admin*.



The screenshot shows the Django Admin interface. At the top, there's a blue header with 'Administração do Django' on the left and 'Bem-vindo(a), admin. Alterar senha / Encerrar sessão' on the right. Below the header, a breadcrumb trail reads 'Início > Tarefas > Tarefas > Adicionar tarefa'. The main content area is titled 'Adicionar tarefa'. It contains a form with two fields: 'Nome:' with a text input box, and 'Descrição:' with a larger text area. Below these fields is a checkbox labeled 'Finalizada'. At the bottom of the form, there are three buttons: 'Salvar e adicionar outro(a)', 'Salvar e continuar editando', and a blue 'Salvar' button.

Muito melhor né? Você pode estar se perguntando sobre a data de execução. Ela deve ser informada automaticamente quando finalizarmos a tarefa, então, vamos colocá-la como *blank=True* e *null=True* também para não precisarmos

informá-la.

Feito isto, geramos uma nova migração e aplicamos, para podermos editar e criar tarefas sem informar a data de execução.

```
python manage.py makemigrations
```

```
python manage.py migrate
```

Agora devemos ter 3 migrações.

Vamos então mudar a lista de nosso admin, pois *Tarefa* object não é um bom nome para aparecer na listagem de tarefas.

Para isto, acrescente na sua classe *TarefaAdmin* um atributo chamado *list\_display*. Este atributo recebe uma lista com os nomes dos campos que devem aparecer na lista de objetos. Vamos colocar assim.

```
list_display = ['nome', 'descricao', 'data_de_criacao', 'status']
```

O resultado será este:

Selecione tarefa para modificar Adicionar tarefa +

Ação:  Fazer 0 de 1 selecionados

<input type="checkbox"/> Nome	Descrição	Data de criação	Finalizada
<input type="checkbox"/> teste	tarefa teste CTNovatec	5 de Abril de 2015 às 17:36	<input checked="" type="checkbox"/>

1 tarefa

Agora sim, nosso *admin* está ficando bom!

Mas falta uma alteração importante. Ninguém quer ter que acessar a tela de edição de uma tarefa para marcar que já executou ela. Queremos que ela seja editável na lista, o que é muito mais fácil.

Para isto, adicione na sua classe *TarefaAdmin* um atributo *list\_editable*. Este atributo também recebe uma lista de strings com os nomes dos campos que devem ser editáveis.

```
list_editable = ['status']
```

Agora, veja como ficou a lista de tarefas da nossa aplicação.

Selecione tarefa para modificar Adicionar tarefa +

Ação:  Fazer 0 de 1 selecionados

<input type="checkbox"/> Nome	Descrição	Data de criação	Finalizada
<input type="checkbox"/> teste	tarefa teste CTNovatec	5 de Abril de 2015 às 17:36	<input type="checkbox"/>

1 tarefa Salvar

Altere o status de algumas tarefas e salve para ver como funciona.

# Views

Agora que sabemos criar um crud bem completo com o Admin do Django chega o momento de vermos como criar nossas próprias telas.

## O que são views?

No Django, *Views* são funções que recebem as requisições web, tratam os dados recebidos e devolvem algum dado, que pode ser um *json*, uma *string* ou uma página renderizada.

Existem dois tipos de *views*. *Views* baseadas em funções e *views* baseadas em classes. As *views* baseadas em funções existem no Django desde sempre, já as baseadas em classes são uma *feature* relativamente nova, mas que trouxe muita flexibilidade tanto para estender as *views* genéricas que o Django já possui, quanto para criar novas *views* de forma clara e objetiva.

Vamos a seguir criar duas *views* que estendem *views* genéricas do Django.

## Usando as Views do Django

No diretório da nossa aplicação tem um arquivo chamado *views.py*. É neste arquivo que vamos colocar as *views* que criarmos.

Antes de mais nada, delete qualquer linha que já estiver neste arquivo. Não vamos precisar de nenhum import que esteja aí por padrão, e caso precisemos, é melhor que você mesmo faça para guardar melhor.

Vamos importar uma *view* genérica chamada *ArchiveIndexView* que nos dá uma lista de objetos ordenada por data, de acordo com o campo que informarmos.

Importe então a classe

```
from django.views.generic.dates import ArchiveIndexView
```

e também a classe *Tarefa*

```
from tarefas.models import Tarefa
```

Agora, vamos criar uma *view* chamada *TarefasView*, que estende *ArchiveIndexView*

```
class TarefasView(ArchiveIndexView):  
    model = Tarefa  
    date_field = 'data_de_criacao'
```

Os atributos *model* e *date\_field* servem respectivamente para indicar qual o modelo que será listado e qual o atributo que será usado para ordenar a lista.

Agora vamos criar uma *view* para os detalhes da aplicação.

```
class TarefaDetail(DetailView):  
    model = Tarefa
```

Aqui, o atributo *model* indica qual modelo deve fornecer os dados para a página.

## Criando suas próprias views

Para criarmos nossas próprias Views, criamos classes que estendem a classe View. Vamos criar uma view para a home de nosso projeto.

Primeiro importamos a classe View.

```
from django.views.generic import View
```

Vamos também importar alguns helpers

```
from django.shortcuts import render_to_response  
from django.template.context import RequestContext
```

E agora a classe

```
class Home(View):  
    template_name = 'tarefas/home.html'  
    context = {}  
  
    def get(self, request, *args, **kwargs):  
        return render_to_response(self.template_name, self.context,  
                                RequestContext(request))
```

Dissecando esta classe, temos o seguinte:

Criamos uma classe que estende a classe *View*. Dentro dela, criamos dois atributos, chamados respectivamente *template\_name* e *context*.

*template\_name* terá o nome do arquivo *html* do *template* de nossa página, e *context* é um dicionário que conterá informações que podem ser acessadas do *template*. Veremos isto em mais detalhes quando falarmos de *templates*.

Na nossa *view* declaramos o método *get*. Ele vai tratar as requisições do tipo



*get*. Se fôssemos por exemplo receber dados em um formulário, deveríamos ter um método *post* e assim por diante.

Como parâmetros do método *get* temos além do *self*, *request*, que é um objeto que contém todos os dados que recebemos das requisições e *\*args* e *\*\*kwargs* que recebem parâmetros que podemos passar pela URL. *\*args* e *\*\*kwargs* podem ser omitidos.

Dentro do método *get*, podemos trabalhar dados e chamar funções como quisermos, e no final, retornamos a página renderizada para aparecer no *browser*. Isto é feito na última linha. Ali, chamamos a função *render\_to\_response*. Esta função recebe como parâmetros o nome do *template*, a variável de contexto, e o contexto do *request*, dado pela classe *RequestContext*.

O contexto do request é indispensável pois é ele que permite usarmos sessões, cookies, etc.

## Urls

### Mapeando URLs

Agora que criamos nossas *views*, precisamos criar *urls* para podermos acessá-las.

Temos um arquivo *urls.py* no diretório do projeto. Vamos criar também um arquivo *urls.py* no diretório da aplicação tarefas. É importante que tudo que funciona junto esteja junto no projeto, então, as urls da aplicação devem ficar dentro da aplicação.

Mas, para o Django saber que temos um arquivo de urls dentro da aplicação vamos explicitar isto no urls do projeto.

Abra o arquivo *urls.py* do projeto. Ele deve estar assim:

```
from django.conf.urls import patterns, include, url
from django.contrib import admin

urlpatterns = patterns('',
    # Examples:
    # url(r'^$', 'minhas_tarefas.views.home', name='home'),
    # url(r'^blog/', include('blog.urls')),

    url(r'^admin/', include(admin.site.urls)),
)
```

Vamos importar nele o *urls* da aplicação, então se ainda não criou este arquivo, pode criar agora.

```
import tarefas.urls
```

Acrescente logo após a linha

```
url(r'^admin/', include(admin.site.urls)),
```

outra linha como o seguinte conteúdo

```
url(r'', include(tarefas.urls)),
```

Neste caso, estamos importando o módulo de *urls* na forma *módulo.submódulo*, porque no arquivo de *urls*, em projetos grandes, teremos vários arquivos *urls* incluídos aqui, e importar somente o *urls* pode criar conflitos.

Pra entender o arquivo, é tudo bem direto. Aqui, declaramos um objeto *patterns* que recebe como primeiro parâmetro o *namespace*, que muitas vezes será uma string vazia. Os parâmetros seguintes são *urls* que recebem como parâmetros uma expressão regular, e uma *view*, ou no caso aqui, a expressão regular e o *include* para outras *urls* contidas em outro módulo.

Na expressão regular definimos a *url* que será acessada para chegar àquela *view* ou que será tratada pelas próximas *urls* dos módulos ali incluídos.

Por exemplo, em `'^admin/'` dizemos que as *urls* que começam com **admin/** terão tudo que vier depois da barra / tratado pelo módulo que especificamos no *include*.

Precisamos agora então criar nosso arquivo *urls.py* da aplicação para tratar as *views* criadas anteriormente.

Para isto, vamos copiar o conteúdo do arquivo do projeto para o da aplicação, porém vamos remover o *import* do *admin* e remover as *urls* do *patterns*.

O módulo então ficará assim:

```
from django.conf.urls import patterns, include, url
```

```
urlpatterns = patterns('',  
)
```

Agora falta acrescentarmos as *urls* para nossas *views*.

Primeiro, vamos acrescentar uma *url* para nossa *home*. Precisaremos então

também criar nosso primeiro *template*.

No diretório da aplicação crie um diretório chamado *templates*, dentro dele um diretório chamado *tarefas* e dentro dele um arquivo *home.html*.

Crie no arquivo *home.html* uma estrutura básica de html. Logo mudaremos isto.

```
<html>
<head>
</head>
<body>
</body>
</html>
```

Agora, no módulo *urls*, importe a view *Home* e acrescente ao *patterns* a seguinte linha como temos no arquivo do projeto.

```
url(r'^$', Home.as_view()),
```

Nesta *url*, dizemos que quando a url que vier para este módulo estiver vazia, será a *Home* quem vai tratar a requisição. A chamada de *as\_view()* é necessária pois o *url* espera uma função, e este método das *views* contorna este problema.

Vamos agora rodar novamente nosso servidor e acessar <http://localhost:8000>

Você vai reparar que agora não existe mais aquela tela do Django com “It worked!” mas uma tela em branco.

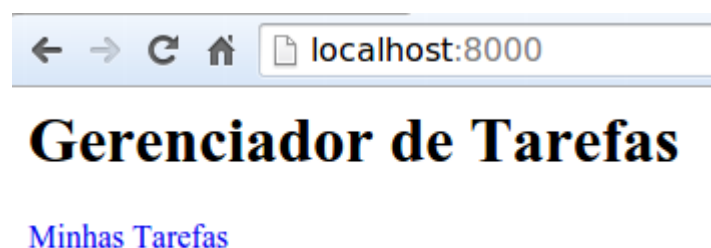
Não se assuste, pois nada deu errado. Simplesmente agora o django está renderizando nosso *html* vazio do *home.html*!

Vamos colocar algum conteúdo nele para comprovar.

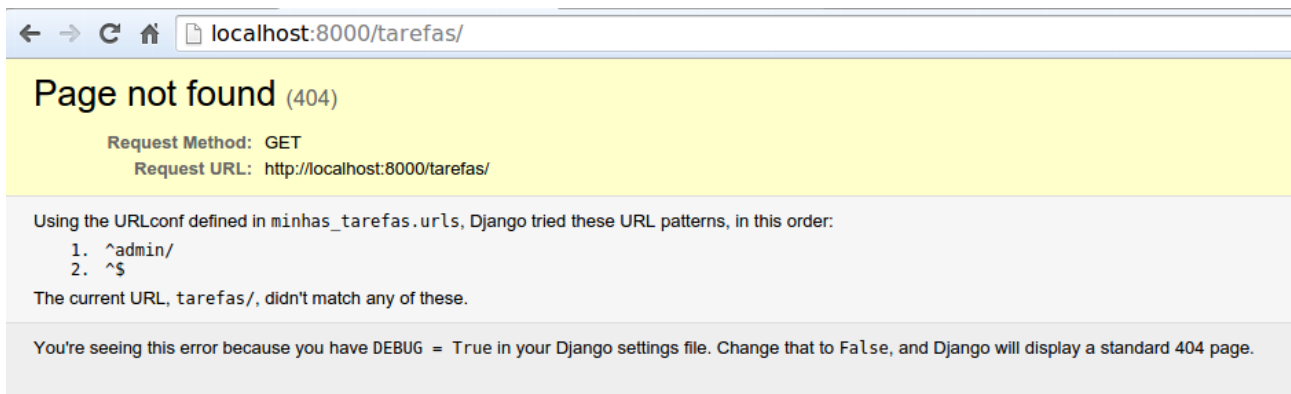
Dentro da *tag body*, coloque o seguinte conteúdo

```
<h1>Gerenciador de Tarefas</h1>
<a href="/tarefas/">Minhas Tarefas</a>
```

Agora salve o arquivo e recarregue a página



Clique no link que colocamos na página.



O Django não encontrou a url que tentamos acessar. É hora colocá-la no lugar.

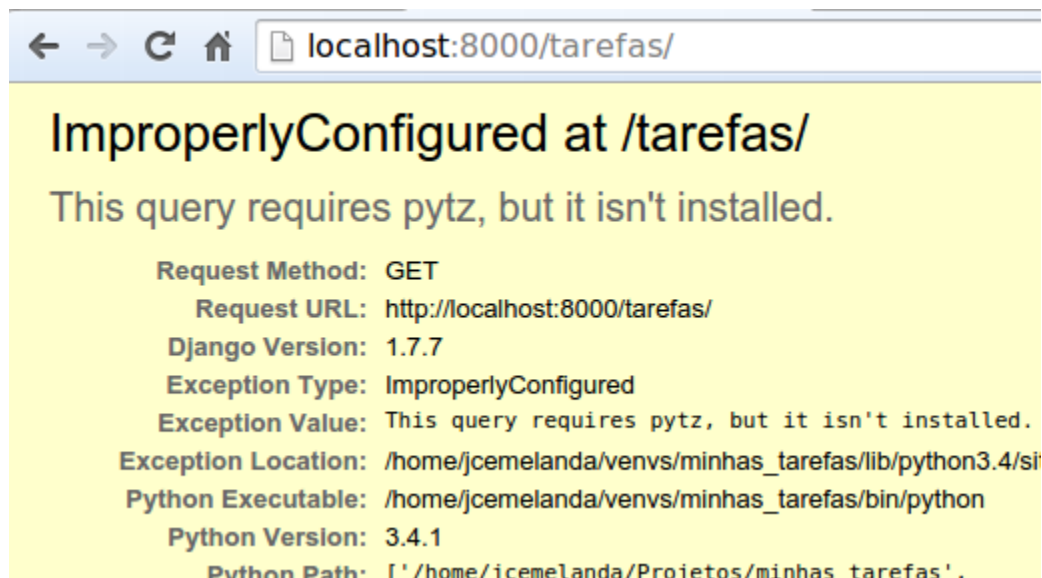
Primeiro, importe o TarefasView

```
from tarefas.views import TarefasView
```

e agora a configuração da url em si.

```
url(r'^tarefas/$', TarefasView.as_view()),
```

Tente acessar a url novamente.



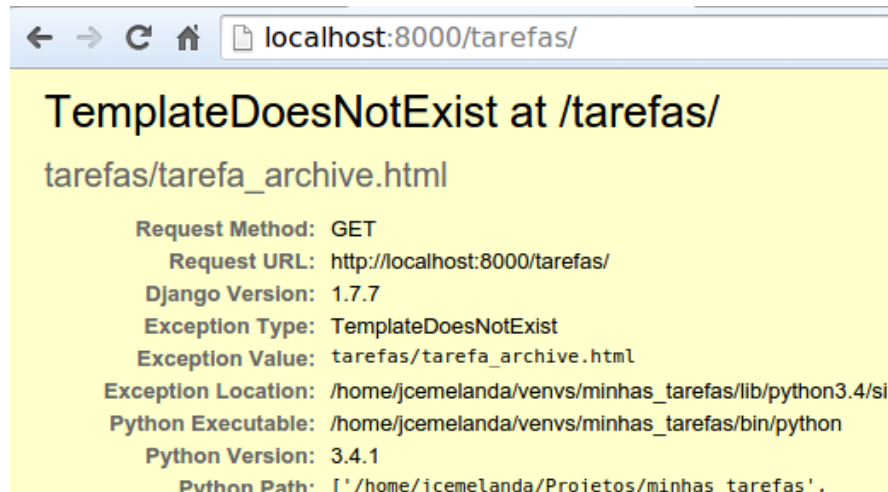
O que ocorre aqui é que no nosso settings.py temos USE\_TZ=True mas não instalamos o pytz, que é usado pelo Django para tratar datas usando timezones. Vamos instalá-lo.

```
pip install pytz
```

É bom colocar o `pytz` no `requirements.txt` também.

`pytz==2015.2`

Acesse novamente.



Como usamos uma *view* genérica para tratar a lista de tarefas, precisamos de um *template* para esta lista. A *view* genérica espera que exista um arquivo *html* com o nome `modelo_archive`, no nosso caso, `tarefa_archive`. Assim, crie um arquivo *html* no diretório *templates* com este nome. Crie uma estrutura *html* vazia dentro dele.

Quando acessar novamente o endereço, agora você verá uma página totalmente vazia!

Vamos acrescentar um pequeno trecho de *template* para mostrarmos as tarefas cadastradas. Falaremos mais de *templates* na próxima seção.

```
<ul>
    {% for item in latest %}
        <li>
            <a href="/tarefas/{{ item.pk }}">{{ item.nome }} {{ item.status }}</a>
        </li>
    {% endfor %}
</ul>
```

Dando uma pincelada no que ocorre neste *template*. As *views* do tipo *ArchiveIndexView* colocam no contexto uma lista contendo os objetos que queremos listar, no caso, tarefas. Numa *view* do tipo *ArchiveIndexView*, a lista sempre se chama `latest`.

Assim, iterando sobre esta lista conseguimos mostrar os valores dos atributos

na página.

## Passando argumentos nas URLs

Falta somente a *view* de detalhes das tarefas agora. Clique no link de uma das tarefas da nossa lista de tarefas.



Como ainda não criamos a url para esta página, não conseguimos acessá-la.

Repare que nesta url passamos a chave primária da tarefa que queremos visualizar. Assim, precisamos receber este parâmetro na url.

Importe no `urls.py` da aplicação a classe `TarefaDetail`

```
from tarefas.views import TarefaDetail
```

E insira entre as `urls`

```
url(r'^tarefas/(?P<pk>\d+)/$', TarefaDetail.as_view()),
```

Agora vamos acessar novamente a página de detalhes. Você verá uma tela dizendo que falta o template de detalhes, então vamos mais uma vez criar um `html` limpo no diretório `tarefas` do diretório `templates` da aplicação assim como fizemos com os anteriores.

Recarregue a página e verá novamente uma página em branco.

Vejamos como fizemos para indicar ao `django` que passaríamos a `primary key` na url.

```
r'^tarefas/(?P<pk>\d+)/$'
```

Nesta *regex* (expressão regular) indicamos que a *url* deve começar com tarefas, e após a barra definimos um grupo nomeado. O parêntese define que é um grupo, o ***?P<pk>*** indica que o nome do grupo será *pk* e o ***\d+*** indica que este grupo deverá conter um ou mais dígitos numéricos.

Assim, dentro da *view*, que estendemos, o método *get* tem um parâmetro ***\*\*kwargs***, e neste dicionário *kwargs* haverá um item cuja chave é ***'pk'*** e o valor é o número passado na url.

Estes são parâmetros nomeados. Caso quiséssemos passar um parâmetro não nomeado, bastaria no lugar de ***(?P<pk>\d+)*** usarmos somente ***(\d+)*** e o valor seria recebido na *view* em ***\*args***, que é uma lista, não em ***\*\*kwargs*** como acontece neste caso.

## Templates

Quando criamos *websites* precisamos de uma forma para injetar dados na página, e assim, poderemos mostrar informações que estão armazenadas no banco de dados ou foram calculadas no *backend*. É para isto que usamos os *templates*.

### O que é o sistema de templates do Django

O sistema de templates do Django foi desenvolvido para permitir a passagem de dados do backend para o html de forma simples e objetiva.

As tags que identificam os comandos de templates do Django são ***{% %}*** que são usadas para funções e comandos e ***{{ }}*** que são usadas para variáveis.

Por exemplo, em nossa página de lista de tarefas usamos

```
{% for item in latest %}
    <li>
        <a href="/tarefas/{{ item.pk }}">{{ item.nome }} {{ item.status }}</a>
    </li>
{% endfor %}
```

Quase todas as tags ***{% %}*** tem uma tag de abertura e uma de fechamento pois elas em geral determinam blocos de códigos como fazemos aqui com o *for*.

Em geral, *if* e *for* tem uma sintaxe bem parecida com a do python nos *templates*.

Operadores de comparação também são basicamente os mesmos que usamos no código.

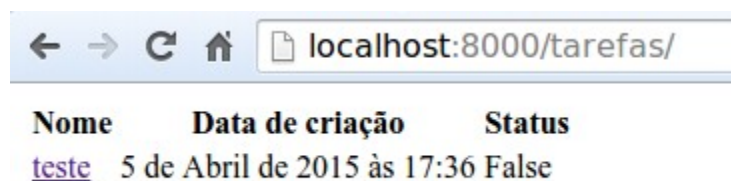
Vamos fazer algumas melhorias nos nossos *templates*.

No *template tarefa\_archive.html* vamos trocar a lista por uma tabela para

organizarmos os dados.

```
<table>
  <tr>
    <th>Nome</th>
    <th>Data de criação</th>
    <th>Status</th>
  </tr>
  {% for item in latest %}
  <tr>
    <td><a href="/tarefas/{{ item.pk }}">{{ item.nome }}</a></td>
    <td>{{ item.data_de_criacao }}</td>
    <td>{{ item.status }}</td>
  </tr>
  {% endfor %}
</table>
```

Quando colocarmos esta estrutura no lugar da lista no html, estamos criando uma tabela com os dados de cada tarefa.



The screenshot shows a web browser window with the address bar displaying 'localhost:8000/tarefas/'. Below the address bar, there is a table with three columns: 'Nome', 'Data de criação', and 'Status'. The first row of the table contains the text 'teste', '5 de Abril de 2015 às 17:36', and 'False'.

Nome	Data de criação	Status
<a href="#">teste</a>	5 de Abril de 2015 às 17:36	False

Temos dois problemas nesta tabela. O primeiro é que a data ficou muito grande. O segundo é que no nosso contexto, false como status não quer dizer nada. Semanticamente falando, False não encaixa ali. Vamos corrigir isto.

Para a data, precisamos de um *tag filter*. Tag filters são funções que são chamadas, passando como um dos parâmetros o texto que está sendo filtrado, e retorna um novo texto a partir do processamento do original. Vejamos no exemplo.

```
<td>{{ item.data_de_criacao|date:"d/m/Y H:i" }}</td>
```

Veja que adicionamos um pipe “|” e o filtro date. Para o date, passamos o parâmetro “d/m/Y H:i” que é uma *string* que representa a formatação do *datetime* que usamos aqui.

- d => dia com 2 dígitos
- m => mês com 2 dígitos



- Y => ano com 4 dígitos
- H => hora no formato 24h
- i => minutos

Para o status vamos usar um if da seguinte forma:

```
<td>{% if item.status %}Finalizado{% else %}Aberto{% endif %}</td>
```

Assim como no código python, aqui podemos no *if* simplesmente avaliar se uma variável é verdadeira sem precisar comparar com *True* ou *False*. O mesmo vale para verificar se listas estão vazias ou se objetos são nulos.

Repare que como é uma tag que define bloco, precisamos da tag de fechamento `{% endif %}`

Por hora nossa página de lista de tarefas está boa. Vamos melhorar um pouco nossa *home*.

Na view Home, acrescente o seguinte dentro do método get:

```
self.context['counter'] = Tarefa.objects.filter(status=True).count()
```

Agora no HTML da home vamos acrescentar logo após o h1:

```
<h2>Olá {{ user.first_name }}</h2>
```

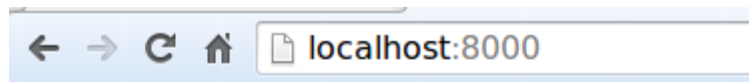
```
Você tem {{ counter }} {% if count > 1 %}tarefas abertas{% else %}tarefa  
aberta{% endif %}</br>
```

Assim, teremos na tela uma mensagem de olá e a informação de quantas tarefas temos.



Poxa, não tem nenhum nome cadastrado para o usuário, vamos resolver isto primeiro.

Acesse o *admin*, então *Usuários* e selecione o seu usuário. Coloque seu nome no campo *Primeiro nome* e salve.



# Gerenciador de Tarefas

**Olá Julio Cesar**

Você tem 1 tarefa aberta

[Minhas Tarefas](#)

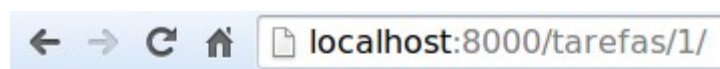
Agora temos uma *home* bem mais amigável.

Fica faltando então a tela de detalhes das tarefas.

Veja o código a seguir

```
Nome: {{ object.nome }}</br>
Data de Criação: {{ object.data_de_criacao|date:"d/m/Y H:i"}}</br>
Descirção: {{ object.descricao }}</br>
Status: {% if object.status %}Finalizado{% else %}Aberto{% endif %}</br>
{% if object.data_de_execucao %}
Data de Execução: {{ object.data_de_execucao }}</br>
{% endif %}
```

Acesse novamente a página de detalhe de uma tarefa.



Nome: teste  
Data de Criação: 05/04/2015 17:36  
Descirção: tarefa teste CTNovatec  
Status: Aberto

Muito bom, você já sabe agora criar *templates* para mostrar nas páginas os dados das *views*!

## Como utilizar o Twitter Bootstrap?

Já sabemos criar *templates*, mas eles ainda estão muito feios. Precisamos dar um tapa na interface. Para isto, vamos criar um *template* base para o site e vamos usar *bootstrap*.

Dentro do diretório *templates* crie um arquivo *base.html* e crie uma estrutura básica de *html* nele.

No header, vamos colocar os imports do bootstrap.

```
<!-- Latest compiled and minified CSS -->
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.4/css/bootstrap.min.css"
">

<!-- Optional theme -->
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.4/css/bootstrap-
theme.min.css">

<!-- Latest compiled and minified JavaScript -->
<script
src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.4/js/bootstrap.min.js">
</script>
```

Além disto, vamos acrescentar a *meta tag* de *viewport* para as funções responsivas funcionarem corretamente.

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

Isto já é suficiente para termos o bootstrap funcionando minimamente em nosso sistema.

Vamos então incrementar um pouco.

Acrescente no *body* o seguinte:

```
<div class="container">
  <nav class="navbar navbar-default">
    <div class="container-fluid">
      <div class="navbar-header">
        <h1>Gerenciador de Tarefas</h1>
      </div><!-- navbar-header -->
    </div><!-- container-fluid -->
  </nav> <!-- Navbar -->
  <div class="jumbotron">
    {% block content %}
    {% endblock %}
  </div><!-- jumbotron-->
</div> <!-- Container -->
```

Isto já é suficiente para nossa interface ficar bem agradável. Precisamos agora alterar nossos *templates* para usarem este base.

O html da home ficará assim:

```
{% extends "base.html" %}
{% block content %}
    <div class="page-header">
        <h2>Olá {{ user.first_name }}</h2>
    </div>
    Você tem {{ counter }} {% if count > 1 %}tarefas abertas{% else %}tarefa
aberta{% endif %}</br>
    <a href="/tarefas/">Minhas Tarefas</a>
{% endblock %}
```

Note as duas *tags* que usamos aqui. **`{% extends "base.html" %}`** indica uma relação de herança entre os *templates*. **`{% block content %}`** é uma *tag* que define um bloco genérico. Veja que em *base.html* nós definimos este bloco, e em *home.html* nós sobrescrevemos o bloco, assim como faríamos com métodos ou atributos de uma classe.

O efeito final é um *html* com tudo que tem no *base.html* mais tudo que tem no *home.html*, inserido no local definido no bloco *content*.

Acesse a página e veja para entender melhor.

---

## Gerenciador de Tarefas

Olá Julio Cesar

Você tem 1 tarefa aberta  
[Minhas Tarefas](#)

Vamos repetir o processo para as outras duas páginas.

Vejam como ficam os *htmls* *tarefa\_archive.html* e *tarefa\_detail.html* respectivamente.

```
{% extends "base.html" %}
{% block content %}
    <div class="page_header">
        <h2>Minhas Tarefas</h2>
    </div>
    <table>
        <tr>
            <th>Nome</th>
```

```

        <th>Data de criação</th>
        <th>Status</th>
    </tr>
    {% for item in latest %}
    <tr>
        <td><a href="/tarefas/{{ item.pk }}">{{ item.nome }}</a></td>
        <td>{{ item.data_de_criacao|date:"d/m/Y H:i" }}</td>
        <td>{% if item.status %}Finalizado{% else %}Aberto{% endif %}</td>
    </tr>
    {% endfor %}
</table>
{% endblock %}

```

---

```

{% extends "base.html" %}
{% block content %}
    <div class="page_header">
        <h2>Detalhes da Tarefa</h2>
    </div>
    Nome: {{ object.nome }}<br/>
    Data de Criação: {{ object.data_de_criacao|date:"d/m/Y H:i" }}<br/>
    Descrição: {{ object.descricao }}<br/>
    Status: {% if object.status %}Finalizado{% else %}Aberto{% endif %}<br/>
    {% if object.data_de_execucao %}
    Data de Execução: {{ object.data_de_execucao }}<br/>
    {% endif %}
{% endblock %}

```

Nossos *templates* ficam muito mais enxutos e claros quando fazemos desta forma.

Porém, visualmente, o efeito não foi tão bom quanto o esperado, ainda não temos aquele visual bonito do *bootstrap*. Precisamos fazer mais algumas alterações.

## Gerenciador de Tarefas

### Minhas Tarefas

Nome	Data de criação	Status
teste	05/04/2015 17:36	Aberto

## Gerenciador de Tarefas

### Detalhes da Tarefa

Nome: teste  
Data de Criação: 05/04/2015 17:36  
Descrição: tarefa teste CTNovatec  
Status: Aberto

Vamos primeiro melhorar a tabela que tem nossa lista de tarefas.

Para isto, basta colocar a classe table no elemento table.

```
<table class="table">
  <thead>
    <tr>
      <th>Nome</th>
      <th>Data de criação</th>
      <th>Status</th>
    </tr>
  </thead>
  <tbody>
    {% for item in latest %}
      <tr>
        <td><a href="/tarefas/{{ item.pk }}">{{ item.nome }}</a></td>
        <td>{{ item.data_de_criacao|date:"d/m/Y H:i" }}</td>
        <td>{% if item.status %}Finalizado{% else %}Aberto{% endif %}</td>
      </tr>
    {% endfor %}
  </tbody>
</table>
```

Vejamos o resultado:

## Gerenciador de Tarefas

### Minhas Tarefas

Nome	Data de criação	Status
<a href="#">teste</a>	05/04/2015 17:36	Aberto

Muito mais apresentável.

Hora de acertar a tela de detalhes. Vamos usar um componente *dl*, que é usado justamente para termos e descrições, o que cabe bem numa listagem de detalhes do objeto.

```
<dl class="dl-horizontal">
  <dt>Nome:</dt>
  <dd>{{ object.nome }}</dd>
  <dt>Data de Criação:</dt>
  <dd>{{ object.data_de_criacao|date:"d/m/Y H:i"}}</dd>
  <dt>Descrição:</dt>
  <dd>{{ object.descricao }}</dd>
  <dt>Status:</dt>
  <dd>{% if object.status %}Finalizado{% else %}Aberto{% endif %} </dd>
  {% if object.data_de_execucao %}
    <dt>Data de Execução:</dt>
    <dd>{{ object.data_de_execucao }}</dd>
  {% endif %}
</dl>
```

Heis o resultado visual:

## Gerenciador de Tarefas

### Detalhes da Tarefa

Nome: teste  
Data de Criação: 05/04/2015 17:36  
Descrição: tarefa teste CTNovatec  
Status: Aberto

Muitas vezes precisamos simplesmente exibir dados e esta é uma forma simples e elegante de mostrá-los.

## Formulários

Além de exibir dados, um sistema web precisa também receber dados inseridos pelos usuários. Para isto vamos usar os forms. Os forms estão entre as grandes facilidades que o Django nos provê.

## Os Forms do Django

No Django, existe uma classe *Form* que é uma abstração de um formulário web. Ali podemos especificar *widgets*, validações, etc.

Para exemplificar, vamos criar um formulário de criação de usuários em nossa aplicação.

Crie no diretório da aplicação um arquivo chamado *forms.py*. O primeiro passo é importar a classe Form.

```
from django.forms import Form
```

Para este formulário, vamos precisar de um campo de texto para o nome de usuário (*username*), outro campo de texto para o nome do usuário, um campo opcional de e-mail, um campo de senha e um de confirmação de senha. Vamos importá-los.

```
from django.forms import CharField
from django.forms import EmailField
from django.forms import PasswordInput
```

E agora criamos a classe de formulário.

```
class FormNovoUsuario(Form):
    nome_de_usuario = CharField()
    nome = CharField(required=False)
    email = EmailField(required=False)
    senha = CharField(widget=PasswordInput)
    repeticao_senha = CharField(widget=PasswordInput)
```

Mas, esta classe por si só não faz nada. Precisamos primeiro sobrescrever seu método *save* para que possamos criar um usuário ao salvar o formulário. E precisamos também criar a view para acessarmos o formulário.

Primeiro, o método *save*. Vamos precisar da classe *User* dentro dele, então vamos importá-la também.

```
from django.contrib.auth.models import User
```

Então o método *save* em si:

```
def save(self):
    params = {
        'username': self.cleaned_data['nome_de_usuario'],
        'email': self.cleaned_data['email'],
        'password': self.cleaned_data['senha'],
    }
```



```
if self.cleaned_data['nome']:
    params['first_name'] = self.cleaned_data['nome']
    User.objects.create_user(**params)
```

Muito bom! Mas ainda falta fazermos a verificação de senha. Para isto, vamos criar um método chamado *clean\_senha*. Os métodos do tipo *clean\_<atributo>* são métodos especiais com o intuito de fazer validações customizadas e pré processamento dos dados recebidos.

```
def clean_repeticao_senha(self):
    senha1 = self.cleaned_data['senha']
    senha2 = self.cleaned_data['repeticao_senha']

    if senha1 != senha2:
        raise ValidationError("As senhas devem ser iguais.")
    return senha
```

Aqui temos algumas coisas importantes. Primeiro o atributo *cleaned\_data*, que recebe um dicionário contendo os objetos correspondentes aos atributos do *form* quando o método *clean* é executado, o que já ocorreu quando o processamento chega ali.

Outro ponto importante é o *ValidationError*. Uma classe de erro específica para erros de validação. São estes erros que devemos usar nas validações de formulários.

Bom, já temos nosso formulário, mas ainda precisamos de uma *view* para apresentá-lo.

```
class CriaUsuario(View):
    template_name = 'tarefas/cria_usuario.html'
    context = {}

    def get(self, request, *args, **kwargs):
        self.context['form'] = FormNovoUsuario()
        return render_to_response(self.template_name, self.context,
                                  RequestContext(request))

    def post(self, request, *args, **kwargs):
        form = FormNovoUsuario(request.POST)
        if form.is_valid():
            form.save()
```

```

        return redirect('/')
    else:
        self.context['form'] = form
        return render_to_response(self.template_name, self.context,
                                   RequestContext(request))

```

Precisamos olhar com um pouco mais de calma para esta *view*. No método *get*, nós instanciamos um *form* vazio, e passamos este *form* para o *template* através do atributo *context*. No método *post*, nós primeiro instanciamos o formulário, passando *request.POST*. *request.POST* é um dicionário que contém todos os dados enviados via *post* para o sistema. Assim, a instância criada terá todos os dados passados no formulário da tela.

Em seguida, verificamos a validade do formulário com *form.is\_valid()*. Este método vai chamar todos os métodos de validação do formulário e inclusive o *clean\_\_repeticao\_senha* que implementamos, ou seja, caso as senhas informadas não coincidam, o método *is\_valid()* retornará *false*.

Caso o retorno de *is\_valid()* seja *True*, chamamos o método *save* do *form*, que também implementamos e criará um novo usuário.

Caso seja *false*, reatribuímos o *form* de volta para o contexto e renderizamos a tela novamente com o *form* preenchido.

Além dos dados do *form*, quando devolvemos a instancia para a tela, vão também os erros relacionados a cada *field*, então poderemos mostrar para os usuários os erros que ocorreram.

Crie então um arquivo *cria\_usuario.html* no diretório de *templates* que temos usado para todos os outros *templates*. Vamos estender o *base.html* e colocar o seguinte conteúdo:

```

{% extends "base.html" %}
{% block content %}
    <div class="page-header">
        <h2>Criação de Usuário</h2>
    </div>
    <form method="POST" action=".">
        {% csrf_token %}
        {{ form.as_p }}
        <input type="submit" value="Criar">
    </form>
{% endblock %}

```

Aqui temos um formulário *html* normal. Quando fazemos a chamada

**`{{ form.as_p }}`** os inputs do formulário são colocados na tela automaticamente como elementos *p*.

Note a tag **`{% csrf_token %}`**. Esta tag é usada para proteção contra ataques do tipo ***cross site request forgery***.

Falta então colocarmos uma url para esta view. Acrescente esta linha às suas urls:

```
url(r'^criar_usuario/$', CriaUsuario.as_view()),
```

O visual da página não ficou muito bom.

Agora temos duas possibilidades. Podemos alterar os *widgets* no Django e colocar as classes do bootstrap na definição dos widgets do formulário. Ou criar os inputs manualmente no html com nomes que o django reconheça quando enviarmos os formulários.

Vamos escolher a segunda opção por uma razão muito simples “*Separation of Concerns*”.

Não devemos misturar as coisas. *Backend* cuida do *backend* e *frontend* cuida do *frontend*. Especificar detalhes do *frontend* no *backend* não é legal para reusabilidade nem para manutenibilidade do código.

Vamos mudar o HTML da seguinte forma:

```
{% extends "base.html" %}
{% block content %}
    <div class="page-header">
        <h2>Criação de Usuário</h2>
    </div>
    <form method="POST" action=".">
        {% csrf_token %}
        <div class="form-group">
            <label for="id_nome_de_usuario">Username</label>
            <input type="text" id="id_nome_de_usuario"
                name="nome_de_usuario" class="form-control">
        </div>
        <div class="form-group">
            <label for="id_nome">Nome</label>
            <input type="text" id="id_nome" name="nome" class="form-
                control">
        </div>
        <div class="form-group">
```

```

        <label for="id_email">Email</label>
        <input type="email" id="id_email" name="email" class="form-
            control">

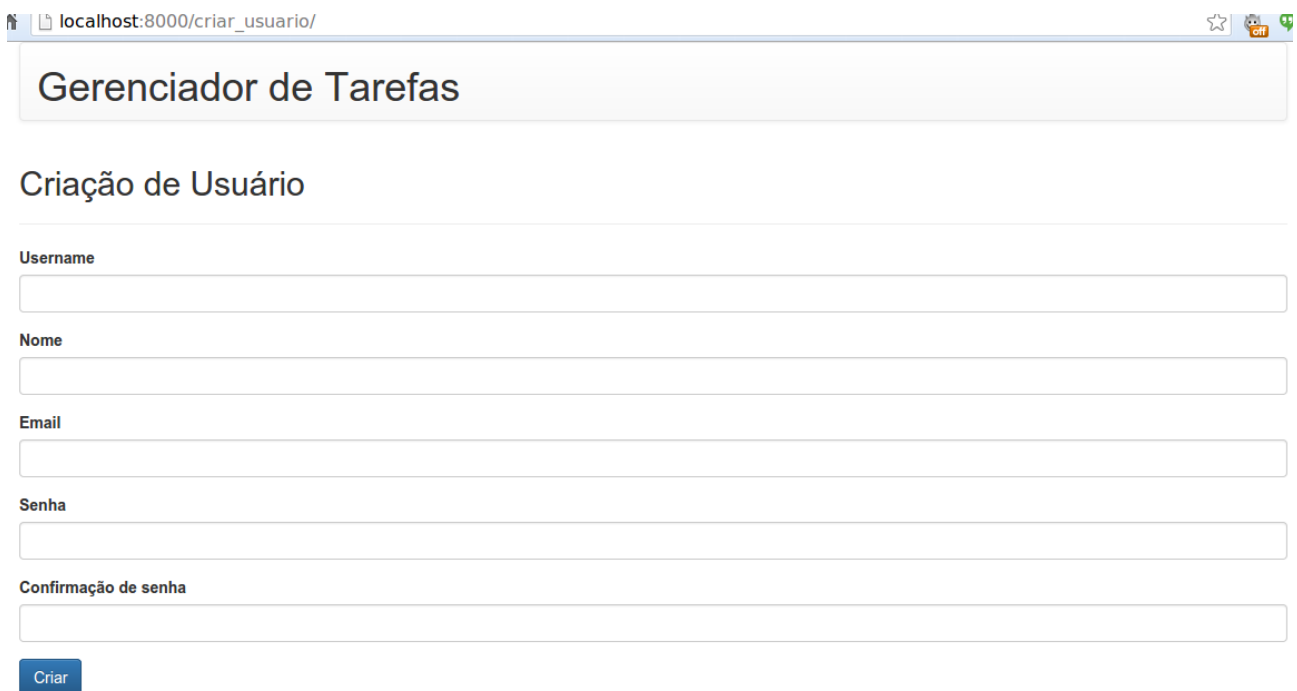
    </div>
    <div class="form-group">
        <label for="id_senha">Senha</label>
        <input type="password" id="id_senha" name="senha" class="form-
            control">

    </div>
    <div class="form-group">
        <label for="id_repeticao_senha">Confirmação de senha</label>
        <input type="password" id="id_repeticao_senha"
            name="repeticao_senha" class="form-control">

    </div>
    <input type="submit" class="btn btn-primary" value="Criar">
</form>
{% endblock %}

```

Vejamos como ficou agora nosso formulário.



The screenshot shows a web browser window with the address bar displaying 'localhost:8000/criar\_usuario/'. The page title is 'Gerenciador de Tarefas'. Below the title, the heading 'Criação de Usuário' is visible. The form contains five input fields, each with a label above it: 'Username', 'Nome', 'Email', 'Senha', and 'Confirmação de senha'. At the bottom left of the form is a blue button labeled 'Criar'.

Teste o formulário, crie usuários submeta com informações faltantes para ver os resultados.

Ainda não está bom, né, falta a renderização dos erros e é ruim termos que criar toda esta estrutura na mão para cada campo. Vamos ver se não conseguimos um meio termo entre o automático e o totalmente manual.

```

{% extends "base.html" %}
{% block content %}
    <div class="page-header">
        <h2>Criação de Usuário</h2>
    </div>
    <form method="POST" action=".">
        {% csrf_token %}
        {{ form.non_field_errors }}
        {% for field in form %}
            <div class="form-group">
                <label for="{{ field.id_for_label }}" class="label-control">
                    {{ field.label }}
                    {% if field.field.required %}*{% endif %}
                </label>
                {{ field }}
                {% if field.errors %}
                    <div class="alert alert-danger">{{ field.errors }}</div>
                {% endif %}
            </div>
        {% endfor %}

        <button type="submit" class="btn btn-primary">Criar</button>
    </form>
    <script>
        $("input").addClass("form-control");
    </script>
{% endblock %}

```

Aqui, iteramos sobre os campos do *form*, e acessamos os campos de forma mais automatizada. No final, acrescentamos a classe *form-control* aos *inputs* via *jquery*.

Veja que agora temos *feedback* dos erros de validação.

## Trabalhando com ModelForms

Vamos também permitir ao usuário criar tarefas pela interface padrão do sistema. Para tal usaremos uma classe especial de formulários chamada *ModelForm*.

No arquivo `forms.py` importe a classe *ModelForm*

```
from django.forms import ModelForm
```

Importe também o modelo Tarefa

```
from tarefas.models import Tarefa
```

Agora a classe de ModelForm

```
class FormTarefa(ModelForm):  
    class Meta:  
        model = Tarefa  
        fields = ['nome', 'descricao', 'status']
```

Esta é uma classe bastante simples. Criamos a classe interna *Meta*, e dentro dela, definimos qual o modelo para o qual queremos o formulário, e quais os campos que devem estar no form.

É hora de criar uma view para a criação das tarefas. Vamos chamá-la de AdicionaTarefa.

Importe o *form* em *views.py* e acrescente

```
class AdicionaTarefa(View):  
    template_name = 'tarefa/cria_tarefa.html'  
    context= {}  
  
    def get(self, request, *args, **kwargs):  
        self.context['form'] = FormTarefa()  
        return render_to_response(self.template_name, self.context,  
                                   RequestContext(request))  
  
    def post(self, request, *args, **kwargs):  
        form = FormTarefa(request.POST)  
        if form.is_valid():  
            form.save()  
            return redirect('/tarefas/')  
        else:  
            self.context['form'] = form  
            return render_to_response(self.template_name, self.context,  
                                       RequestContext(request))
```

Perceba que esta *view* é muito parecida com a outra *view* que criamos para

criação de usuário.

Falta agora somente criar a *urls* para podermos acessá-la e o seu *template*.

Em *urls.py*, importe nossa view e acrescente a seguinte url.

```
url(r'^tarefas/criar/', AdicionaTarefa.as_view()),
```

E no diretório de *templates*, crie o arquivo *cria\_tarefa.html*. Seu conteúdo será muito parecido com o nosso *template* de criação de usuário.

```
{% extends "base.html" %}
{% block content %}
    <div class="page-header">
        <h2>Nova Tarefa</h2>
    </div>
    <form method="POST" action=".">
        {% csrf_token %}
        {% if form.non_field_errors %}
            <div class="alert alert-danger">
                {{ form.non_field_errors }}
            </div>
        {% endif %}
        {% for field in form %}
            <div class="form-group">
                <label for="{{ field.id_for_label }}" class="label-control">
                    {{ field.label }}
                    {% if field.field.required %}*{% endif %}
                </label>
                {{ field }}
                {% if field.errors %}
                    <div class="alert alert-danger">{{ field.errors }}</div>
                {% endif %}
            </div>
        {% endfor %}

    <button type="submit" class="btn btn-primary">Criar</button>
</form>
<script>
$("input").addClass("form-control");
$("textarea").addClass("form-control");
```

```

        $("input[type=checkbox]").removeClass("form-control");
    </script>
{% endblock %}

```

A maior diferença neste template se encontra no javascript, onde além de adicionarmos a classe form-control aos inputs acrescentamos ao textarea, e posteriormente, removemos a classe do nosso checkbox para que ele fique alinhado corretamente.

Gerenciador de Tarefas

Nova Tarefa

Nome \*

Descrição

Finalizada ☐

Criar

Agora só falta colocarmos um botão na nossa lista de tarefas para acrescentarmos novas tarefas.

No arquivo *tarefas\_archive.html* vamos acrescentar antes da tabela o seguinte trecho:

```
<a class="btn btn-primary" href="/tarefas/criar/">Adicionar Tarefas</a>
```

Gerenciador de Tarefas

Minhas Tarefas

Adicionar Tarefas

Nome	Data de criação	Status
teste	07/04/2015 15:55	Aberto
teste	05/04/2015 17:36	Aberto

Agora finalizamos a interface de nossa aplicação!

## Banco de dados

Como já vimos superficialmente anteriormente, o Django tem formas de



automatizar a geração e manipulação do banco de dados. Isto se dá através do seu mapeador objeto-relacional (ORM).

Inicialmente, em settings.py fazemos as Configurações de acesso ao banco. Lá especificamos o backend, que por padrão vem configurado para usar o sqlite3. Também podemos configurar para usar PostgreSQL, MySQL e Oracle.

Além do backend, podemos configurar a porta, nome do banco, host, senha e usuário. Como estamos usando sqlite3 somente informamos o backend e o nome do arquivo do banco.

## Fazendo queries com o ORM do Django

Com o banco configurado e iniciado conforme explicamos na seção anterior, falta entendermos melhor como acessados os dados.

Vamos para isto acessar o shell do Django que nada mas é que um shell python configurado com as variáveis de ambiente do Django.

Execute o comando:

```
python manager.py shell
```

Agora temos o shell pronto para uso. Vamos importar um modelo.

```
from tarefas.modelos import Tarefa
```

Então vamos listar todas as tarefas existentes.

```
Tarefa.objects.all()
```

Você verá uma listam de objetos tarefa.

Vamos então buscar somente as tarefas ativas:

```
Tarefa.objects.filter(is_active=True)
```

Podemos também filtrar usando comparações

```
from datetime import datetime
```

```
dia=datetime(10, 04, 2015)
```

```
Tarefa.objects.filter(data_de_criação__gt=dia)
```

Poderia ser *gt = greater then*, *gte = greater then or equals* *lt = lower then* *lte= lower then or equals*

Também é possível verificar pertinência a um grupo:

```
Tarefa.objects.filter(id__in=[1, 2, 3, 4])
```

Além de filtrar grupos e de filtrar objetos podemos buscar todos exceto um grupo, trocando *filter* por *except*.

```
Tarefa.objects.except(is_active=True)
```

Finalmente, podemos buscar um individuo especifico do conjunto de dados usando *get*.

```
Tarefa.objects.get(pk=2)
```

## Fazendo buscas mais avançadas

### Inserindo relações nos modelos

Agora vamos fazer uma pequena alteração a nosso modelo para podermos ver comandos mais avançados de bancos de dados.

Acrescente ao modelo Tarefa:

```
usuario = ForeignKey(User, verbose_name="usuário", null=True, blank=True)
```

Para funcionar, faremos o import de *ForeignKey* e de *User*

```
from django.contrib.auth.models import User
from django.db.models import ForeignKey
```

Então geramos a migração com essa alteração e aplicamos

```
python manage.py makemigrations
python manage.py migrate
```

Agora, vamos alterar a nossa *view* para salvar o usuário ativo em cada tarefa que criarmos.

Nas *view*, após criarmos a tarefa, acrescente a seguinte linha

```
tarefa.usuario = request.user
tarefa.save()
```

Assim, quando o `form.is_valid()` for verdadeiro, teremos

```
tarefa = form.save()
tarefa.usuario = request.user
tarefa.save()
return redirect('/tarefas/')
```

Faca login via admn e salve uma nova tarefa na nossa view.

Vamos então acrescentar este dado nas nossas páginas de listas e detalhes para podermos ver o usuário de cada tarefa.

Ao template de lista, acrescente:

```
<th>Usuario</th>
```

ao final dos *headers* da tabela e

```
<td>{{ item.usuario }}</td>
```

ao final da linha.

Nossa tabela fica então assim no template:

```
<table class="table">
  <thead>
    <tr>
      <th>Nome</th>
      <th>Data de criação</th>
      <th>Status</th>
      <th>Usuario</th>
    </tr>
  </thead>
  <tbody>
    {% for item in latest %}
      <tr>
        <td><a href="/tarefas/{{ item.pk }}">{{ item.nome }}</a></td>
        <td>{{ item.data_de_criacao|date:"d/m/Y H:i" }}</td>
        <td>{% if item.status %}Finalizado{% else %}Aberto{% endif %}</td>
        <td>{{ item.usuario }}</td>
      </tr>
    {% endfor %}
  </tbody>
```

</table>

Vejamos o resultado.

## Gerenciador de Tarefas

### Minhas Tarefas

Adicionar Tarefas			
Nome	Data de criação	Status	Usuario
Tarefa Nova	14/04/2015 20:56	Aberto	admin
teste	07/04/2015 15:55	Aberto	None
teste	05/04/2015 17:36	Aberto	None

Então, acertamos também a tela de detalhes. Acrescente antes do fechamento </dl>.

<dt>Usuário</dt>

<dd>{{ object.usuario }}</dd>

Agora, o resultado.

## Gerenciador de Tarefas

### Detalhes da Tarefa

Nome: Tarefa Nova  
Data de Criação: 14/04/2015 20:56  
Descrição: Nova tarefa com usuário  
Status: Aberto  
Usuário: admin

Finalmente, nos dois trechos acima, acrescente *.first\_name* após *usuario* e teremos o nome do usuário assim como mostrado na tela inicial.

## Buscas avançadas

Agora que já temos uma relação entre o modelo de tarefas e o de usuários, vamos fazer algumas buscas mais avançadas.

De volta ao shell, digite

```
Tarefa.objects.filter(user__username="admin")2
```

O que fazemos neste filtro é buscar tarefas através de um campo de um atributo da tarefa.

---

<sup>2</sup> No lugar de *admin*, use o nome que tiver dado ao seu usuário.

Esta busca retorna todas as tarefas cujo modelo referido em `usuario` tenha o `username` com valor *“admin”*. A relação do atributo da tarefa com o atributo do atributo é dada pelos 2 *underscores*.

Podemos fazer este filtro com vários níveis caso necessário.

Se quiser por exemplo buscar todos os usuários que tem alguma tarefa, faça:

```
User.objects.exclude(tarefa_set=None)
```

Observe aqui que usamos um atributo *tarefas\_set* que não está declarado no `User`. Porém, isto faz parte da mágica do Django.

Quando criamos campos de relação (`ManyToMany`, `ForeignKey`, `OneToOne`), é criado automaticamente um campo no outro lado da relação. Ou seja, quando criamos `usuario` em `tarefas` apontando para `user`, o Django insere em `User` um atributo *tarefa\_set*, que será o conjunto de tarefas que apontam para ele.

Isto significa, que no Django, as relações são bidirecionais, ou seja, posso acessar `user` de `tarefa`, ou `tarefa` de `user`.

Estes campos criados automaticamente tem nomes padrão.

Quando criamos uma `foreignkey`, o nome padrão é *atributo\_set*. Quando a relação é `manytomany`, o nome padrão é *atributos* e quando é `onetoone`, é *atributo*. Sendo neste caso, *atributo* o nome do modelo pai da relação (onde inserimos manualmente o campo).

Porém nem sempre este nome padrão é conveniente, ou então podemos querer mais de uma relação para o mesmo objeto. Neste caso, precisamos especificar este nome manualmente.

Isto é feito através de um parâmetro do *field* chamado *related\_name*. Vamos alterar o atributo usuário da tarefa para vermos como fazer isto.

```
usuario = ForeignKey(User, verbose_name='usuário', null=True, blank=True,
                     related_name='tarefas')
```

Agora, se quisermos fazer a busca, temos que mudar o comando, usando *tarefas* no lugar de *tarefa\_set*.

```
User.objects.exclude(tarefas=None)
```

## Migrações

As migrações são muito importantes nos projetos Django. Existem duas importantes razões para termos as migrações.

Uma delas é termos um versionamento do banco, nos permitindo voltar a estados anteriores do banco para fazermos testes, etc.

Mas provavelmente a principal motivadora das migrações foi uma limitação que as versões mais antigas do Django tinham.

Ao criar um modelo, rodávamos um comando chamado syncdb, que alterava o banco de acordo com nossas mudanças, mas mudanças de tipos de dados entre outras coisas eram bastante complicadas de fazer, muitas vezes precisando criar script que executavam queries no banco de dados diretamente.

Isto aumentava a dificuldade em gerenciar o banco, e às vezes impedia a troca do sistema de banco de dados sem a necessidade de alteração de código, pois existia sql puro no programa.

Com as migrações, passamos a ter comandos de alto nível para criação de tabelas, criação, edição e deleção de colunas, entre outras coisas.

## Entendendo os arquivos de migração

Veamos abaixo a estrutura básica de um arquivo de migração usando nossa terceira migração como exemplo.

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals
from django.db import models, migrations

class Migration(migrations.Migration):

    dependencies = [
        ('tarefas', '0002_auto_20150405_1725'),
    ]

    operations = [
        migrations.AlterField(
            model_name='tarefa',
            name='data_de_execucao',
            field=models.DateTimeField(verbose_name='data de execução',
                                       blank=True, null=True),
            preserve_default=True,
        ),
    ]
```

Nossas migrações até agora foram todas geradas automaticamente.

Elas contem uma classe de migrações, onde são declaradas as dependências, que serão as últimas migrações que precisam ter sido executadas antes delas. Em nosso caso temos somente a migração anterior, mas quando temos várias aplicações, pode ser possível, uma migração depender de uma ou mais migração de outras apps.

Também são declaradas as operações. Neste caso, `AlterField`, que é uma classe que sabe tratar quando a migração é aplicada ou quando voltamos para a migração anterior com base nos dados informados de nome do modelo, nome do campo, tipo do campo, etc.

## Testes

Python é uma linguagem de tipagem dinâmica, o que significa que as variáveis assumem os tipos dos dados atribuídos a elas.

Assim, quando aplicamos operações de inteiros ao retorno de uma função, estas operações somente vão funcionar corretamente se o retorno desta função for um inteiro. Mas como o Python não checa tipos, você não saberá que o retorno da função está errado até que seu cliente esteja usando o sistema e apareça um erro bem na cara dele.

Por este motivo em especial, softwares feitos em Python precisam de testes. Quanto mais testes, mais seguro você estará de que o sistema realmente funciona e não haverá imprevistos em tempo de execução.

## TDD

TDD, vem de Test Driven Development que significa desenvolvimento orientado a testes. Isto quer dizer que você não escreve uma linha de código sem ter feito testes para isto antes. Com TDD, criamos primeiro um teste para uma funcionalidade, rodamos o teste e nos certificamos de que ele falha, e então escrevemos o mínimo de código necessário para o teste passar.

Então você cria mais testes para a funcionalidade, ou incrementa os já existentes e implementa as correções necessárias no código para fazer os novos testes passarem, assim, sucessivamente incrementando os testes e em seguida o código.

Você certamente percebeu que não estamos fazendo nosso sistema com TDD. Mas como nunca é tarde para começar, vamos fazer alguns testes para nossa aplicação.

## Testes unitários

Testes unitários são testes que verificam unidades do código. Mais comumente, funções e métodos de classes. Mas também podem ser usados

para testar funcionalidades do sistema.

Veja que quando criamos nossa app Tarefas, o Django já criou para nós um arquivo chamado tests.py

Abra o arquivo.

```
from django.test import TestCase
```

```
# Create your tests here.
```

Bastante simples e autoexplicativo, este módulo já importa para nós a classe TestCase que usaremos para criar nossos testes.

TestCase é uma classe que agrupa testes, além de preparar automaticamente o banco de dados, carregar fixtures, criar transações, etc. É bom lembrar que o banco de dados usado pelos testes não é o banco de dados da aplicação.

Vamos começar fazendo um teste de exemplo bem simples.

Acrescente ao arquivo:

```
class TesteDummy(TestCase):  
    def test_true(self):  
        self.assertTrue(False)
```

Agora, no terminal, execute o comando

```
python manage.py test
```

Veja na saída do comando que seu teste falhou, o que era esperado pois mandamos ele garantir que era verdadeiro um valor falso. Troque False por True e execute novamente.

Agora nosso teste falhou. Mas chega de testes que não testam nada. Delete esta classe de teste. Vamos testar a criação de nosso modelo.

Acrescente no módulo o seguinte

```
from django.db.utils import IntegrityError  
from tarefas.models import Tarefa
```

```
class CriaTarefaTestCase(TestCase):
```

```
    def test_cria_tarefa_vazia(self):
```



```

        self.assertIsNotNone(Tarefa.objects.create())

    def test_cria_tarefa_sem_nome(self):
        with self.assertRaises(IntegrityError):
            Tarefa.objects.create(nome=None)

    def test_cria_tarefa_com_nome(self):
        self.assertIsNotNone(Tarefa.objects.create(nome="tarefa"))

```

Execute novamente os testes agora. Bom, todos estes testes devem passar.

Só para deixar mais claro os três testes que executamos aqui são:

- Quando criamos uma tarefa vazia programaticamente, funciona.
- Quando criamos uma tarefa como None no nome, retorna uma exceção.
- Quando criamos uma tarefa com nome, funciona.

Vamos agora fazer testes para garantir que nossa home está funcionando corretamente.

Acrescente o import:

```
from django.test import Client
```

Agora a classe de testes:

```

class HomeTestCase(TestCase):
    def test_home_access(self):
        response = self.client.get('/')
        self.assertEqual(response.status_code, 200)

        self.assertContains(response, 'Olá')

```

Nesta classe de testes, nosso método de teste faz duas verificações. A primeira certifica-se que o status da página quando acessada seja 200, ou seja, a página é encontrada, e não dá erro no servidor.

A segunda certifica-se de que exista a palavra “Olá” na página.

## Criando uma página com TDD

Você deve ter reparado que não temos acesso à página de login do site a não ser através do admin. Vamos mudar isto?

Vamos então primeiro criar nosso teste.

Acrescente ao HomeTestCase:

```
def test_home_tem_link_login(self):
    response = self.client.get('/')
    if response.context['user'].is_authenticated():
        self.assertNotContains(response, "Login")
    else:
        self.assertContains(response, "Login")
```

Aqui verificamos se o usuário está autenticado no site. Caso não esteja, vamos colocar na tela um link para o login, então o teste verifica se este link está lá.

Como ainda não temos o link, o teste vai quebrar.

Agora colocamos o link na tela.

Altere o HTML da home para que ele fique assim:

```
{% extends "base.html" %}
{% block content %}
    <div class="page-header">
        <h2>Olá {{ user.first_name }}</h2>
    </div>
    {% if user.is_authenticated %}
        Você tem {{ counter }} {% if count > 1 %}tarefas abertas{% else %}tarefa
aberta{% endif %}<br/>
        <a href="/tarefas/">Minhas Tarefas</a>
    {% else %}
        <a href="/admin/login">Login</a>
    ||
        <a href="/criar_usuario/">Criar usuário</a>
    {% endif %}
{% endblock %}
```

Perceba, que aqui mudamos o comportamento da home. Agora, quando o usuário está logado, ele mostra quantas tarefas há e o link para elas, do contrário, dá o link de login. Aproveitamos para acrescentar também um link de criação de usuário.

Execute o teste que agora passará.

Veja também como ficou na página.

Bom, clique no link.

Você será direcionado para a tela de login, e ao terminar, irá para a tela do admin.

Não é o que queremos. Queremos voltar para o home.

Precisamos acertar isto.

Vamos fazer mais um teste.

Para voltarmos para o home, o link de login deve conter `next=/'`. Assim, vamos testar se o link contém este parâmetro.

```
def test_login_volta_para_home(self):
    response = self.client.get('/')
    if not response.context['user'].is_authenticated():
        self.assertContains(response,
                             '<a href="/admin/login/?next=/'>Login</a>', html=True)
```

Neste teste, verificamos novamente se a resposta contém um texto, mas desta vez passamos o parâmetro `html=True` para indicar que queremos testar um pedaço do html e não do conteúdo da página. Assim, conseguimos verificar se o link está correto.

Execute o teste, e veja-o falhar.

Agora vamos corrigir o html.

Em `home.html`, o link que estava assim

```
<a href="/admin/login/">Login</a>
```

agora deve ficar assim

```
<a href="/admin/login/?next=/'>Login</a>
```

Execute novamente o teste, que agora vai passar!

Da mesma forma, um teste deve ser criado para a funcionalidade de criação do usuário.

## Mock

Mock é uma estratégia usada para testes em casos em que precisamos de objetos complexos dentro de uma rotina de testes.

Vamos sair um pouco de nossa aplicação para exemplificar um Mock.

Imagine a seguinte função.

```
def perimetro_do_circulo(circulo):  
    return 2*3.14*circulo.raio
```

Agora imagine que nossa classe círculo foi criada para um contexto muito específico, e que existem dezenas de parâmetros obrigatórios que precisamos informar para criar um objeto círculo.

Na classe de teste, podemos criar uma classe

```
class CirculoMock:  
    def __init__(self, raio)  
        self.raio = raio
```

Assim, no nosso teste, podemos criar um círculo “mockado”, ou seja, é uma classe que vai se comportar como esperaríamos que o círculo se comportasse e a função perimetro\_do\_circulo não vai perceber que aquele não é o mesmo círculo que seria passado dentro da aplicação.

O método do TestCase ficaria assim:

```
def test_perimetro_circulo(self):  
    self.assertEqual(12.56, perimetro_do_circulo(CirculoMock(2)))
```

Obviamente, esta ideia faz muito mais sentido quando as classes envolvidas são realmente muito complexas, com vários parâmetros obrigatórios, ou ainda parâmetros que devem ser trazidos do banco de dados e então deveriam já estar inseridos no banco anteriormente.

Voltemos então ao nosso gerenciador de tarefas.

## Arquivos estáticos

### O que são arquivos estáticos

Arquivos estáticos são arquivos que estão no servidor, e que não serão modificados a menos que você mesmo vá lá e altere. Como exemplos, podemos ter, logos, scripts, folhas de estilo.

O Django consegue gerenciá-los muito bem no servidor de testes, mas quando colocamos nosso sistema em produção, estes arquivos podem se tornar um problema para o Django. Assim, é comum que configuremos nossos servidores web como NGINX e Apache para servirem os arquivos estáticos enquanto o Django se preocupa somente com os conteúdos dinâmicos do site. O conceito

de arquivos estáticos pode-se estender também a arquivos enviados pelos usuarios, como imagens, pdfs, executáveis, tudo depende de sua aplicação.

Em suma, o ideal é deixar o Django se preocupar com os htmls enquanto um webserver cuida dos outros arquivos.

## Como o Django gerencia arquivos estáticos

Como dito na seção anterior, o Django consegue tratar bem os arquivos estáticos quando em desenvolvimento. Isto é feito por uma aplicação chamada *django.contrib.staticfiles*.

Através desta aplicação, o django serve os arquivos estáticos quando em desenvolvimento ou os agrupa quando em produção.

Veja que em nosso settings, existe a configuração

```
STATIC_URL = '/static/'
```

Aqui dizemos ao Django que os arquivos estáticos serão servidos neste endereço. Assim, arquivos javascript ou css, por exemplo, devem ser acessados por esta url.

Vamos extrair nosso scripts para um arquivo para vermos como isto funciona na prática.

Logo abaixo desta configuração, crie uma nova configuração

```
STATIC_ROOT = os.path.join(BASE_DIR, 'static')
```

Aqui estamos dizendo ao Django que este diretório de arquivos estáticos será o diretório principal.

Agora vamos então criá-lo. Lembre-se que como usamos o *BASE\_DIR*, este diretório deve estar no mesmo diretório que o *settings.py*.

Agora, recrie a mesma estrutura dentro de sua aplicação, ou seja, dentro da aplicação tarefas. Dentro deste diretório *static*, crie um diretório chamado *js*.

Agora, dentro do diretório *js* da aplicação tarefas, crie um arquivo *scripts.js*.

O conteúdo deste arquivo deve ser:

```
acertaClassesUsuario = function(){  
    $("input").addClass("form-control");  
};
```

```
acertaClassesTarefa = function(){  
    $("input").addClass("form-control");
```

```
    $(".textarea").addClass("form-control");  
    $(".input[type=checkbox]").removeClass("form-control");  
};
```

Veja que extraímos as sequências de comandos javascript de nossos htmls e trouxemos para estas funções javascript, assim, caso em algum momento precisamos alterá-los, basta vir a este arquivo e não mais procurar nos htmls.

Agora, vejamos como ficaram os htmls das páginas:

Em cria\_usuario.html

```
<script>  
    acertaClassesUsuario();  
</script>
```

Em cria\_tarefa.html

```
<script>  
    acertaClassesTarefa();  
</script>
```

Porém para usarmos estas chamadas nos htmls, devemos importar nosso script na página. Vamos fazer isto no base.html

Acrescente nele a seguinte tag:

```
{% load staticfiles %}
```

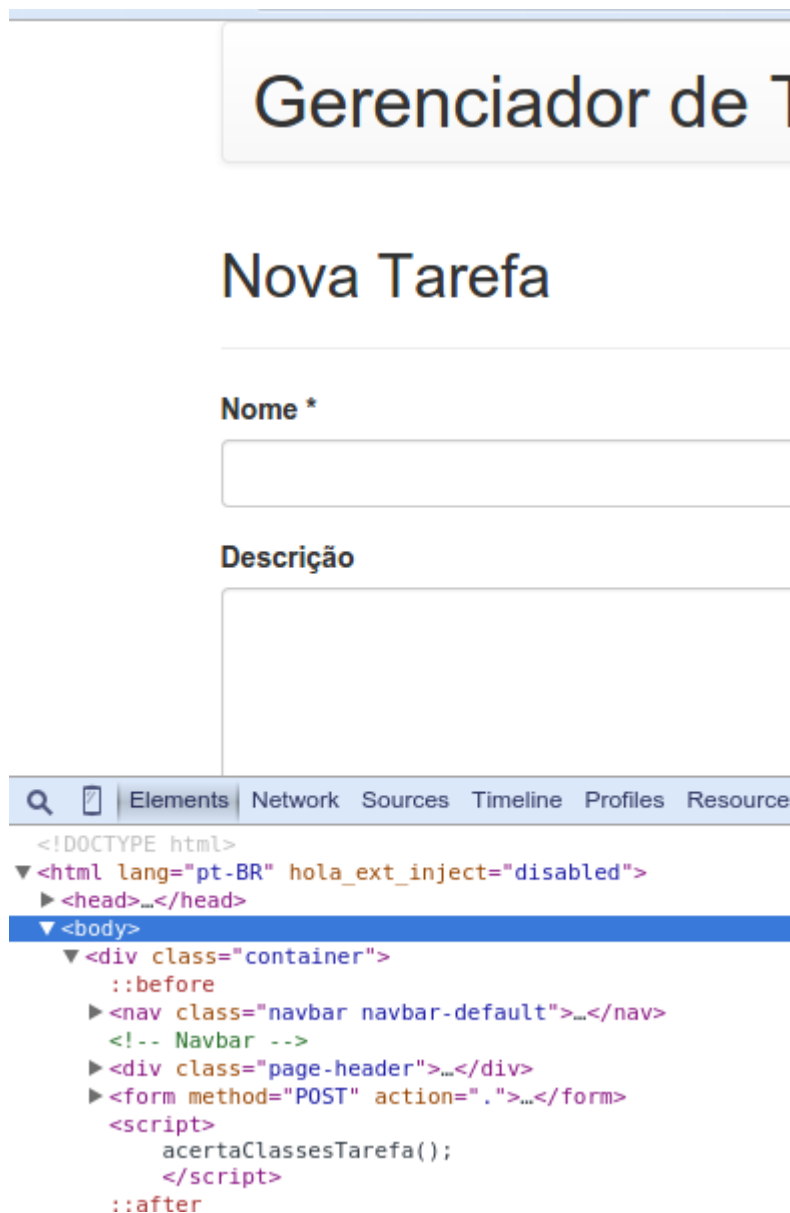
A tag load nos permite carregar conjuntos de funções (novas tags) para dentro de nossos templates.

Depois, antes de fechar o cabeçalho da página (</head>) vamos importar nosso script.

```
<script src="{% static 'js/scripts.js' %}"></script>
```

Perceba que aqui usamos uma nova tag, a tag static. Ela nos permite acessar arquivos estáticos sem saber exatamente onde se encontram no servidor ou mesmo sem sabermos qual a url que deve ser acessada.

Faça o teste acessando a tela de criação de tarefas. Ela deve estar totalmente igual estava anteriormente, pois o javascript será carregado e a função será chamada sem problemas.



Veja que o estilo dos inputs está aplicado e não chamamos mais as funções *jQuery* diretamente, mas o nossa função do arquivo *javascript*.

## Deploy

Chegamos à tão esperada fase de deploy de nosso sistema! Ele está completo, ou praticamente completo, e agora já podemos colocá-lo no ar! Assim, podemos testá-lo e fazer testes com usuários reais!

Para o deploy de nossa aplicação, usaremos um serviço chamado Openshift.

Openshift é um serviço de Platform as a Service (PaaS) e oferece máquinas pré configuradas e bastante robustas e escaláveis. Uma das grandes vantagens do openshift são as 3 máquinas gratuitas que você tem direito! Então, vamos criar uma conta lá, e criar nosso servidor.

# Controle de versão

Uma característica importante do Openshift é que os deploys são feitos através do git, então vamos dar uma breve olhada nele e criar um repositório local para entendermos como funciona.

## O que é o Git

Git é uma ferramenta de controle de versão distribuída criada por Linus Torvalds inicialmente para gerenciar os fontes do Linux.

Hoje o Git figura entre os sistemas de controle de versão mais usados tanto em projetos open source como projetos corporativos e de código fechado.

## Comandos básicos

Dentro do diretório de nosso projeto, inicie um repositório git.

```
git init
```

Agora, vamos verificar estado do repositório

```
git status
```

Observe que aparecerão vários arquivos. Estes arquivos não estão adicionados nos índices do git, então ele não sabe o que fazer com eles. Vamos adicioná-los aos índices.

```
git add .
```

O ponto quer dizer adicione tudo que encontrar neste diretório.

Verifique novamente o status.

Agora os arquivos aparecerão como adicionados ao git.

Agora que os arquivos estão no índice do git, vamos fazer um commit.

Costumo dizer para efeitos didáticos que os commits do git são como pacotes de alterações. Vamos então criar nosso primeiro destes pacotes.

```
git commit -m "Primeiro commit"
```

Os *commits* sempre precisam de uma mensagem, então é bom já fornecê-la diretamente, assim.

Nunca caia na tentação de criar um *commit* com uma mensagem genérica. Se tem que escrever a mensagem já aproveite e escreva algo que identifique as alterações feitas.



Vamos agora verificar os *commits* que já foram feitos.

**git log**

Obviamente só será mostrado um *commit* que foi o que acabamos de fazer.

Estamos suficientemente prontos para o *openshift* agora!

## Criando uma gear no Openshift

Acesse o site do openshift:

<https://openshift.redhat.com>

Crie uma conta gratuita clicando em Sign Up.

Confirme seu e-mail através do link de confirmação enviado para seu e-mail.

Na página seguinte aceite os termos de uso.

Agora, clique em “*Create your first application now*”

[→ Create your first application now](#)

Entre as opções de aplicações, encontre a seção de aplicações python e selecione Python 3.3.



Em Public URL escolha um nome e um domínio para sua aplicação

Public URL   -  .rhcloud.com

Because this is your first application, you need to provide a domain under which your applications will be grouped

Todas as outras configurações podem ficar em seus valores padrão.

Clique então em *Create Application*

A aplicação pode demorar alguns minutos para ser criada.

Não criamos a aplicação usando a aplicação django padrão para evitar problemas caso seja necessário alterar a versão do Django instalada.

Após a aplicação criada você vai para uma página onde pode seguir um tutorial ou simplesmente ir para a página de sua aplicação. Selecione *Not now, continue*.

Você verá uma tela assim:

The screenshot shows the OpenShift Online interface. At the top, there's a navigation bar with 'Applications', 'Settings', and 'Help'. The main content area displays the application details for 'julio-djangoctnovatec.rhcloud.com', which was created 3 minutes ago in the 'djangoctnovatec' domain and 'aws-us-east-1' region. The status is 'Started' with 1 gear icon. Below this, there are sections for 'Cartridges' (showing 'Python 3.3' with status 'Started', 1 small gear, and 1 GB storage), 'Databases' (with options to add MongoDB 2.4, MySQL 5.5, or PostgreSQL 9.2), 'Continuous Integration' (with an 'Enable Jenkins' button), 'Source Code' (with instructions to add an SSH public key), and 'Remote Access' (with a 'Delete this application...' button). A footer link says 'Browse the Marketplace, or see the list of cartridges you can add'.

Veja que no lado direito há uma mensagem dizendo que você deve inserir uma chave pública de ssh antes de acessar ou enviar o sistema para sua aplicação. Vamos fazê-lo então.

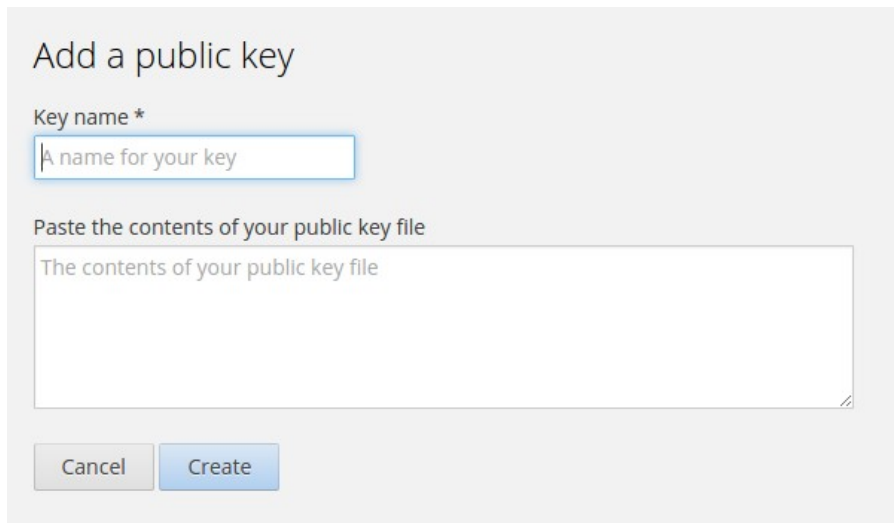
No terminal, digite:

**ssh-keygen**

Siga pressionando enter até o final da criação da chave. Os valores padrão são bons o suficiente.

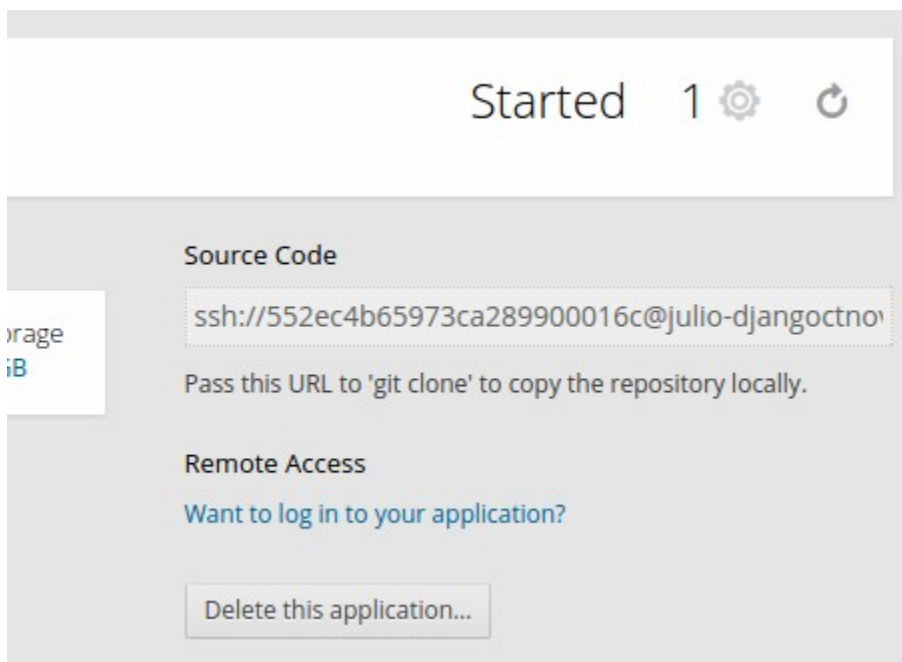
Agora, você deve ter uma chave pública em `.ssh/id_rsa.pub` na sua home. É este valor que temos que inserir no site. Abra este arquivo com um editor qualquer ou então com `cat`, no terminal e copie seu conteúdo na íntegra e sem alterar nada.

Clique no link para inserir uma nova chave e cole os dados do arquivo no campo correspondente. É preciso criar um nome para sua chave para que você possa reconhecer futuramente de onde é a chave.



The screenshot shows a dialog box titled "Add a public key". It contains a text input field labeled "Key name \*" with the placeholder text "A name for your key". Below this is a larger text area labeled "Paste the contents of your public key file" with the placeholder text "The contents of your public key file". At the bottom of the dialog are two buttons: "Cancel" and "Create".

Clique em applications no menu principal da página e selecione novamente sua aplicação.



The screenshot shows the configuration page for an application in OpenShift. At the top, it says "Started" followed by a count of "1" and icons for settings and refresh. Below this, there's a section titled "Source Code" which contains an SSH URL: `ssh://552ec4b65973ca289900016c@julio-djangoctno`. Below the URL, it says "Pass this URL to 'git clone' to copy the repository locally." There's also a section titled "Remote Access" with a link that says "Want to log in to your application?". At the bottom, there's a button that says "Delete this application...".

Veja que agora temos uma url ssh para podermos clonar o repositório git da aplicação do openshift para nossa máquina local, e assim podermos subir nossa aplicação.

## Colocando seu projeto no Openshift

Vamos clonar o projeto.

Antes de qualquer coisa, saia do diretório do nosso projeto. Precisamos dos dados intactos como estão agora.

Fora do diretório do projeto, crie um diretório chamado openshift e dentro dele, execute o comando

```
git clone url_de_ssh_do_openshift
```

Onde url\_de\_ssh\_do\_openshift é a url que aparece na página de sua aplicação.

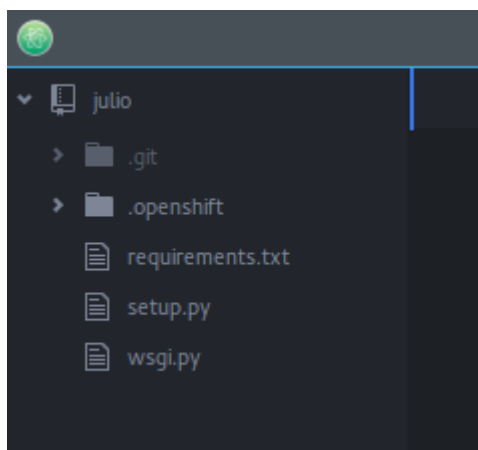
Este comando clona um repositório de exemplo git para a sua máquina, assim, você passa a ter uma cópia do repositório localmente também.

Verifique o status do repositório git

```
git status
```

A saída do comando dirá que você está na *branch master* e que não existem alterações para fazer *commit*.

Vamos abrir o diretório que foi clonado no atom, assim como fizemos com nosso projeto.



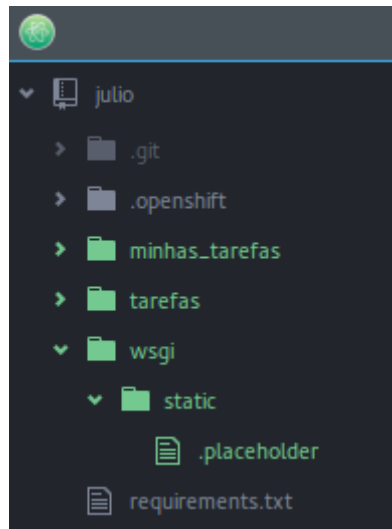
Esta deve ser a estrutura de pastas.

Vamos criar um diretório nesta pasta chamado *wsgi* e dentro dele um outro diretório chamado *static*.

Em static vamos colocar um arquivo vazio somente para o git não ignorar este diretório. Vamos chamar este arquivo de *.placeholder*, que será um arquivo oculto.

Vamos agora copiar nosso projeto *minhas\_tarefas* aqui para esta pasta. Mas não vamos copiar o diretório externo, mas somente os diretórios internos *minhas\_tarefas* e *tarefas*, assim como os arquivos *manage.py* e *requirements.txt*.

A estrutura de pastas agora fica assim então:



Precisamos agora editar o arquivo wsgi que o openshit nos fornece. Delete todo seu conteúdo e no lugar, insira o seguinte:

```
#!/usr/bin/python
import os
virtenv = os.environ['OPENSIFT_PYTHON_DIR'] + '/virtenv/'
virtualenv = os.path.join(virtenv, 'bin/activate_this.py')
try:
    with open(virtualenv) as f:
        code = compile(f.read(), virtualenv, 'exec')
        exec(virtualenv, dict(__file__=virtualenv))
except IOError:
    pass

from minhas_tarefas.wsgi import application
```

Dentro de .opeshift/action\_hooks crie 2 arquivos com os seguintes conteúdos

build

```
#!/bin/bash
#this is .openshift/action_hooks/build
#remember to make it +x so openshift can run it.
if [ ! -d ${OPENSIFT_DATA_DIR} ]; then
    mkdir -p ${OPENSIFT_DATA_DIR}
fi
ln -snf ${OPENSIFT_DATA_DIR} ${OPENSIFT_REPO_DIR}/wsgi/static/
```

deploy

```
#!/bin/bash
#this one is the deploy hook .openshift/action_hooks/deploy
source $OPENSIFT_HOMEDIR/python/virtenv/venv/bin/activate
cd $OPENSIFT_REPO_DIR
echo "Executing 'python manage.py migrate'"
python manage.py migrate
echo "Executing 'python manage.py collectstatic --noinput'"
python manage.py collectstatic --noinput
```

Agora precisamos torná-los executáveis.

No terminal, vá até o diretório onde se encontram estes arquivos e execute:

```
chmod a+x build
chmod a+x deploy
```

Falta agora somente alterarmos a configuração de STATIC\_ROOT para apontar para a nova localização.

```
STATIC_ROOT = os.path.join(BASE_DIR, 'wsgi', 'static')
```

Adora, adicionamos todas as alterações no índice do git

```
git add -all
```

e fazemos um commit

```
git commit -m "Meu site no openshift"
```

Agora, vamos enviar as alterações para o openshift

```
git push origin master
```

O *git push* envia as alterações locais para o servidor. Os parâmetros que usamos são *origin* que é o nome padrão para o servidor *git* e *master* que é a *branch* padrão.

Na página da aplicação no openshift agora, clique na url da aplicação e acesse seu gerenciador de tarefas.

Veja com seus próprios olhos seu gerenciador de tarefas na web para ser usado!



julio-djangoctnovatec.rhcloud.com



# Gerenciador de Tarefas

Olá

---

[Login](#) || [Criar usuário](#)