

Django - Guia rápido

Django - Básico

Django é uma estrutura web Python de alto nível que incentiva o desenvolvimento rápido e um design limpo e pragmático. O Django facilita a construção de aplicativos web melhores, rapidamente e com menos código.

Nota - Django é uma marca registrada da Django Software Foundation e está licenciada sob licença BSD.

História do Django

- **2003** - Iniciado por Adrian Holovaty e Simon Willison como um projeto interno no jornal Lawrence Journal-World.
- **2005** - Lançado em julho de 2005 e batizado de Django, em homenagem ao guitarrista de jazz Django Reinhardt.
- **2005** - Maduro o suficiente para lidar com vários sites de alto tráfego.
- **Current** - Django é agora um projeto de código aberto com colaboradores em todo o mundo.

Django – Filosofias de Design

Django vem com as seguintes filosofias de design -

- **Loosely Coupled** - O Django visa tornar cada elemento de sua pilha independente dos demais.
- **Less Coding** - Menos código, portanto, um desenvolvimento rápido.
- **Don't Repeat Yourself (DRY)** - Tudo deve ser desenvolvido apenas em exatamente um lugar, em vez de repeti-lo indefinidamente.
- **Desenvolvimento Rápido** - A filosofia do Django é fazer tudo o que puder para facilitar o desenvolvimento hiper-rápido.
- **Clean Design** - O Django mantém estritamente um design limpo em todo o seu próprio código e torna mais fácil seguir as melhores práticas de desenvolvimento web.

Vantagens do Django

Aqui estão algumas vantagens de usar Django que podem ser listadas aqui -



- **Suporte a mapeamento relacional de objeto (ORM)** - Django fornece uma ponte entre o modelo de dados e o mecanismo de banco de dados e oferece suporte a um grande conjunto de sistemas de banco de dados, incluindo MySQL, Oracle, Postgres, etc. Django também oferece suporte a banco de dados NoSQL por meio do fork Django-nonrel. Por enquanto, os únicos bancos de dados NoSQL suportados são MongoDB e Google App Engine.
- **Suporte multilíngue** - Django oferece suporte a sites multilíngues por meio de seu sistema de internacionalização integrado. Assim você pode desenvolver seu site, que suportará vários idiomas.
- **Framework Support** - Django possui suporte integrado para Ajax, RSS, Caching e vários outros frameworks.
- **GUI de administração** - Django fornece uma interface de usuário agradável e pronta para uso para atividades administrativas.
- **Development Environment** - Django vem com um servidor web leve para facilitar o desenvolvimento e teste de aplicativos ponta a ponta.

Django - Visão geral

Como você já sabe, Django é um framework web Python. E como a maioria dos frameworks modernos, o Django suporta o padrão MVC. Primeiro vamos ver o que é o padrão Model-View-Controller (MVC), e depois veremos a especificidade do Django para o padrão Model-View-Template (MVT).

Padrão MVC

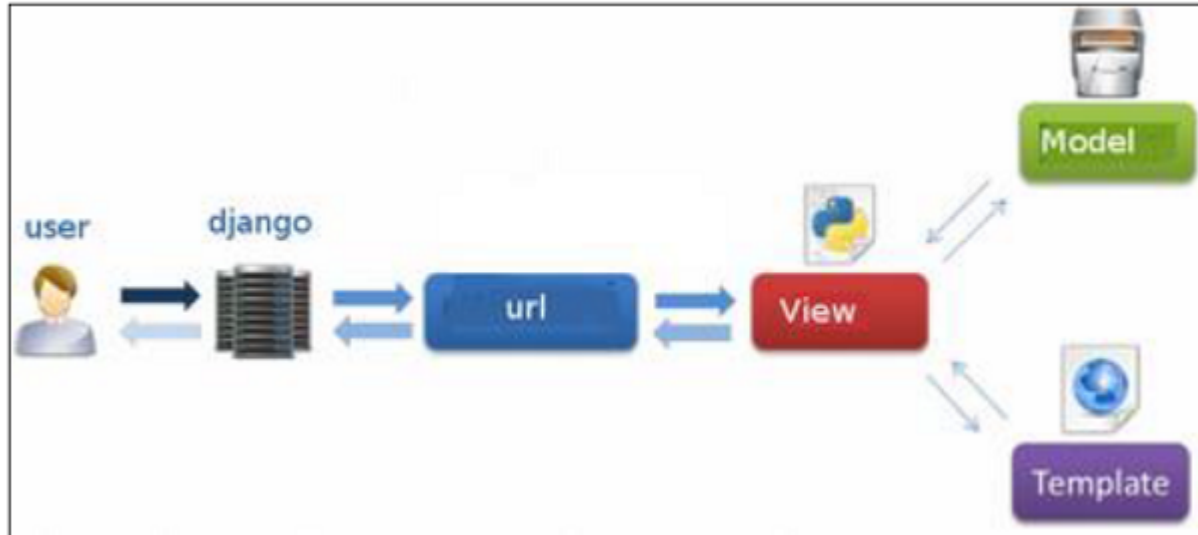
Quando falamos de aplicações que fornecem UI (web ou desktop), costumamos falar de arquitetura MVC. E como o nome sugere, o padrão MVC é baseado em três componentes: Modelo, Visualização e Controlador. [Confira nosso tutorial MVC aqui](#) para saber mais.



DJANGO MVC - Padrão MVT

O Model-View-Template (MVT) é ligeiramente diferente do MVC. Na verdade a principal diferença entre os dois padrões é que o próprio Django cuida da parte do Controller (Código do Software que controla as interações entre o Model e o View), deixando-nos com o template. O modelo é um arquivo HTML misturado com Django Template Language (DTL).

O diagrama a seguir ilustra como cada um dos componentes do padrão MVT interage entre si para atender a uma solicitação do usuário -



O desenvolvedor fornece o modelo, a visualização e o modelo e então apenas mapeia-os para uma URL e o Django faz a mágica para servi-los ao usuário.

Django - Meio Ambiente

O ambiente de desenvolvimento Django consiste na instalação e configuração de Python, Django e um sistema de banco de dados. Como o Django lida com aplicações web, vale a pena mencionar que você também precisaria de uma configuração de servidor web.

DE ANÚNCIOS

Passo 1 – Instalando Python

Django é escrito em código Python 100% puro, então você precisará instalar o Python em seu sistema. A versão mais recente do Django requer Python 2.6.5 ou superior para o branch 2.6.x ou superior a 2.7.3 para o branch 2.7.x.

Se você usa uma das distribuições Linux ou Mac OS X mais recentes, provavelmente já tem o Python instalado. Você pode verificar isso digitando o comando `python` em um prompt de comando. Se você vir algo assim, o Python está instalado.

```
$ python
Python 2.7.5 (default, Jun 17 2014, 18:11:42)
[GCC 4.8.2 20140120 (Red Hat 4.8.2-16)] on linux2
```

Caso contrário, você pode baixar e instalar a versão mais recente do Python no link <http://www.python.org/download> .

DE ANÚNCIOS

Passo 2 - Instalando o Django

Instalar o Django é muito fácil, mas os passos necessários para sua instalação dependem do seu sistema operacional. Como Python é uma linguagem independente de plataforma, o Django possui um pacote que funciona em qualquer lugar, independentemente do seu sistema operacional.

Você pode baixar a versão mais recente do Django no link

<http://www.djangoproject.com/download> .

Instalação UNIX/Linux e Mac OS X

Você tem duas maneiras de instalar o Django se estiver executando o sistema Linux ou Mac OS -

- Você pode usar o gerenciador de pacotes do seu sistema operacional ou usar `easy_install` ou `pip` se instalado.
- Instale-o manualmente usando o arquivo oficial que você baixou anteriormente.

Abordaremos a segunda opção, pois a primeira depende da distribuição do seu sistema operacional. Se você decidiu seguir a primeira opção, tome cuidado com a versão do Django que você está instalando.

Digamos que você obteve seu arquivo no link acima, deve ser algo como `Django-x.xx.tar.gz`:

Extraia e instale.

```
$ tar xzvf Django-x.xx.tar.gz
$ cd Django-x.xx
$ sudo python setup.py install
```

Você pode testar sua instalação executando este comando -

```
$ django-admin.py --version
```

Se você vir a versão atual do Django impressa na tela, então está tudo configurado.

Note - Para algumas versões do Django será `django-admin` o `".py"` foi removido.

Instalação do Windows

Presumimos que você tenha seu arquivo Django e python instalados em seu computador.

Primeiro, verificação PATH.

Em algumas versões do Windows (Windows 7), pode ser necessário garantir que a variável de sistema Path contenha o caminho a seguir C:\Python27\;C:\Python27\Lib\site-packages\django\bin\, é claro, dependendo da sua versão do Python.

Em seguida, extraia e instale o Django.

```
c:\>cd c:\Django-x.xx
```

Em seguida, instale o Django executando o seguinte comando, para o qual você precisará de privilégios administrativos no shell do Windows "cmd" -

```
c:\Django-x.xx>python setup.py install
```

Para testar sua instalação, abra um prompt de comando e digite o seguinte comando -

```
c:\>django-admin.py --version
```

Se você vir a versão atual do Django impressa na tela, então está tudo configurado.

OU

Inicie um prompt "cmd" e digite python então -

```
c:\> python
>>> import django
>>> print django.get_version()
```

Passo 3 – Configuração do Banco de Dados

Django suporta vários mecanismos de banco de dados importantes e você pode configurar qualquer um deles com base no seu conforto.

- [MySQL \(http://www.mysql.com/\)](http://www.mysql.com/)
- [PostgreSQL \(http://www.postgresql.org/\)](http://www.postgresql.org/)
- [SQLite 3 \(http://www.sqlite.org/\)](http://www.sqlite.org/)
- [Oráculo \(http://www.oracle.com/\)](http://www.oracle.com/)
- [MongoDb \(https://django-mongodb-engine.readthedocs.org\)](https://django-mongodb-engine.readthedocs.org)
- [Armazenamento de dados do GoogleAppEngine \(https://cloud.google.com/appengine/articles/django-nonrel\)](https://cloud.google.com/appengine/articles/django-nonrel)

Você pode consultar a respectiva documentação para instalar e configurar um banco de dados de sua escolha.

Note - Os números 5 e 6 são bancos de dados NoSQL.

Passo 4 – Servidor Web

Django vem com um servidor web leve para desenvolver e testar aplicações. Este servidor é pré-configurado para funcionar com Django e, mais importante, ele reinicia sempre que você modifica o código.

No entanto, o Django suporta Apache e outros servidores web populares, como Lighttpd. Discutiremos ambas as abordagens nos próximos capítulos, enquanto trabalhamos com diferentes exemplos.

Django - Criando um Projeto

Agora que instalamos o Django, vamos começar a usá-lo. No Django, todo aplicativo web que você deseja criar é chamado de projeto; e um projeto é uma soma de aplicações. Um aplicativo é um conjunto de arquivos de código que dependem do padrão MVT. Por exemplo, digamos que queremos construir um site, o site é o nosso projeto e o fórum, as notícias, o mecanismo de contato são aplicativos. Essa estrutura facilita a movimentação de um aplicativo entre projetos, pois cada aplicativo é independente.

Crie um projeto

Esteja você no Windows ou Linux, basta obter um terminal ou prompt **cmd** e navegar até o local onde deseja que seu projeto seja criado e usar este código -

```
$ django-admin startproject myproject
```

Isso criará uma pasta “myproject” com a seguinte estrutura -

```
myproject/  
  manage.py  
  myproject/  
    __init__.py  
    settings.py  
    urls.py  
    wsgi.py
```

A Estrutura do Projeto

A pasta “myproject” é apenas o contêiner do seu projeto, na verdade contém dois elementos -

- **manager.py** - Este arquivo é uma espécie de django-admin local do seu projeto para interagir com seu projeto via linha de comando (iniciar o servidor de desenvolvimento,

sincronizar banco de dados ...). Para obter uma lista completa de comandos acessíveis via `manage.py` você pode usar o código -

```
$ python manage.py help
```

- **A subpasta "myproject"** - Esta pasta é o pacote python real do seu projeto. Ele contém quatro arquivos -
 - **__init__.py** - Apenas para python, trate esta pasta como pacote.
 - **settings.py** - Como o nome indica, as configurações do seu projeto.
 - **urls.py** - Todos os links do seu projeto e a função a ser chamada. Uma espécie de ToC do seu projeto.
 - **wsgi.py** - Se você precisar implantar seu projeto no WSGI.

Configurando Seu Projeto

Seu projeto está configurado na subpasta `myproject/settings.py`. A seguir estão algumas opções importantes que você pode precisar definir -

```
DEBUG = True
```

Esta opção permite definir se o seu projeto está em modo de depuração ou não. O modo de depuração permite obter mais informações sobre o erro do seu projeto. Nunca defina como 'True' para um projeto ao vivo. No entanto, isso deve ser definido como 'True' se você quiser que o servidor Django light sirva arquivos estáticos. Faça isso apenas no modo de desenvolvimento.

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.sqlite3',  
        'NAME': 'database.sql',  
        'USER': '',  
        'PASSWORD': '',  
        'HOST': '',  
        'PORT': '',  
    }  
}
```

O banco de dados é definido no dicionário 'Banco de dados'. O exemplo acima é para o mecanismo SQLite. Como afirmado anteriormente, Django também suporta -

- MySQL (django.db.backends.mysql)
- PostgreSQL (django.db.backends.postgresql_psycopg2)
- Oracle (django.db.backends.oracle) e banco de dados NoSQL
- MongoDB (django_mongodb_engine)

Antes de configurar qualquer novo mecanismo, certifique-se de ter o driver db correto instalado.

Você também pode definir outras opções como: TIME_ZONE, LANGUAGE_CODE, TEMPLATE...

Agora que seu projeto foi criado e configurado, certifique-se de que está funcionando -

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

Você obterá algo como o seguinte ao executar o código acima -

```
Validating models...
```

```
0 errors found
```

```
September 03, 2015 - 11:41:50
```

```
Django version 1.6.11, using settings 'myproject.settings'
```

```
Starting development server at http://127.0.0.1:8000/
```

```
Quit the server with CONTROL-C.
```

Django - Ciclo de vida de aplicativos

Um projeto é uma soma de muitas aplicações. Cada aplicação tem um objetivo e pode ser reaproveitada em outro projeto, assim como o formulário de contato de um site pode ser uma aplicação, podendo ser reaproveitado para outros. Veja-o como um módulo do seu projeto.

Crie um aplicativo

Presumimos que você esteja na pasta do seu projeto. Em nossa pasta principal “myproject”, a mesma pasta de manager.py -

```
$ python manage.py startapp myapp
```

Você acabou de criar o aplicativo myapp e como projeto, o Django cria uma pasta “myapp” com a estrutura do aplicativo -

```
myapp/  
__init__.py
```



admin.py
models.py
tests.py
views.py

- **__init__.py** - Só para ter certeza de que o python trata esta pasta como um pacote.
- **admin.py** - Este arquivo ajuda a tornar o aplicativo modificável na interface administrativa.
- **models.py** - É aqui que todos os modelos de aplicativos são armazenados.
- **testes.py** - É aqui que estão seus testes de unidade.
- **views.py** - É aqui que estão as visualizações do seu aplicativo.

Faça com que o projeto conheça sua aplicação

Nesta fase temos a nossa aplicação "myapp", agora precisamos registrá-la no nosso projeto Django "myproject". Para fazer isso, atualize a tupla `INSTALLED_APPS` no arquivo `settings.py` do seu projeto (adicione o nome do seu aplicativo) -

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

Django - Interface Administrativa

Django fornece uma interface de usuário pronta para uso para atividades administrativas. Todos nós sabemos como uma interface administrativa é importante para um projeto web. Django gera automaticamente UI de administração com base em seus modelos de projeto.

Iniciando a interface administrativa

A interface Admin depende do módulo `django.countrib`. Para que funcione, você precisa garantir que alguns módulos sejam importados nas tuplas `INSTALLED_APPS` e `MIDDLEWARE_CLASSES` do arquivo `myproject/settings.py`.

Para `INSTALLED_APPS` certifique-se de ter -

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

Para MIDDLEWARE_CLASSES -

```
MIDDLEWARE_CLASSES = (  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    'django.middleware.common.CommonMiddleware',  
    'django.middleware.csrf.CsrfViewMiddleware',  
    'django.contrib.auth.middleware.AuthenticationMiddleware',  
    'django.contrib.messages.middleware.MessageMiddleware',  
    'django.middleware.clickjacking.XFrameOptionsMiddleware',  
)
```

Antes de iniciar seu servidor, para acessar sua interface administrativa, você precisa iniciar o banco de dados -

```
$ python manage.py migrate
```

O syncdb criará as tabelas ou coleções necessárias dependendo do tipo de banco de dados, necessárias para a execução da interface administrativa. Mesmo se você não tiver um superusuário, será solicitado que você crie um.

Se você já possui um superusuário ou o esqueceu, você pode criar um usando o seguinte código -

```
$ python manage.py createsuperuser
```

Agora, para iniciar a interface administrativa, precisamos ter certeza de que configuramos uma URL para nossa interface administrativa. Abra myproject/url.py e você deverá ter algo como -

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



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

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

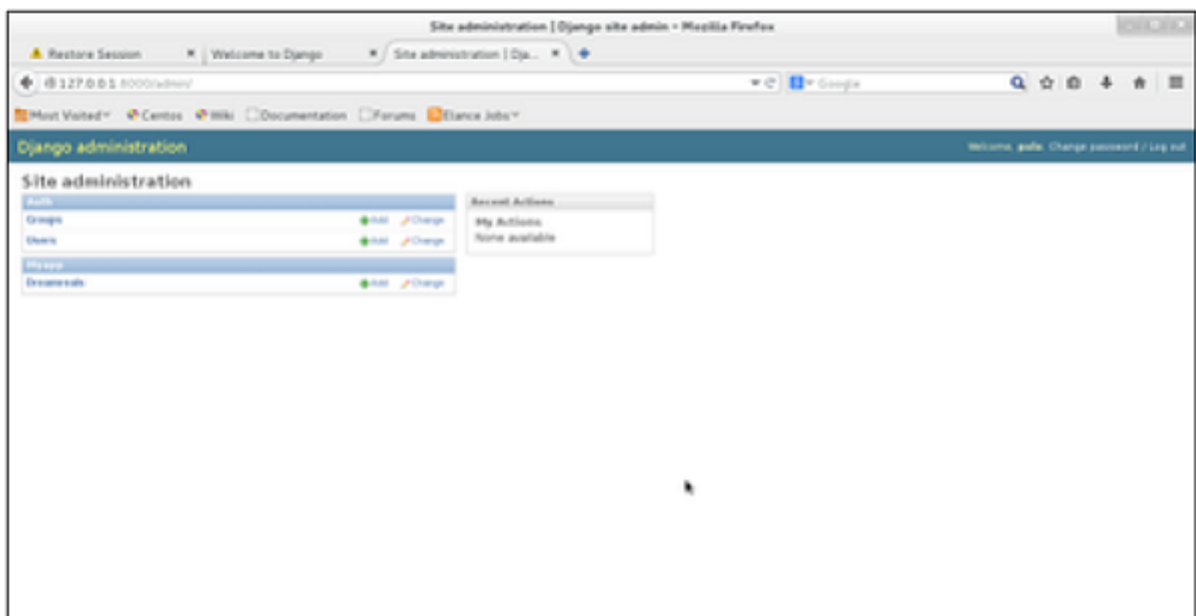
Agora é só executar o servidor.

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

E sua interface administrativa está acessível em: <http://127.0.0.1:8000/admin/>



Uma vez conectado à sua conta de superusuário, você verá a seguinte tela -



Essa interface permitirá que você administre grupos e usuários do Django e todos os modelos registrados em seu aplicativo. A interface oferece a capacidade de realizar pelo menos as

operações "CRUD" (Criar, Ler, Atualizar, Excluir) em seus modelos.

Django - Criando Visualizações

Uma função de visualização, ou “visualização”, é simplesmente uma função Python que recebe uma solicitação da web e retorna uma resposta da web. Esta resposta pode ser o conteúdo HTML de uma página da Web, ou um redirecionamento, ou um erro 404, ou um documento XML, ou uma imagem, etc. Exemplo: você usa a visualização para criar páginas da Web, observe que você precisa associar uma visualização a um URL para vê-lo como uma página da web.

No Django, as visualizações devem ser criadas no arquivo `views.py` do aplicativo.

Visualização Simples

Criaremos uma visualização simples em `myapp` para dizer "bem-vindo ao meu aplicativo!"

Veja a seguinte visualização -

```
from django.http import HttpResponse

def hello(request):
    text = """<h1>welcome to my app !</h1>"""
    return HttpResponse(text)
```

Nesta visualização, usamos `HttpResponse` para renderizar o HTML (como você provavelmente notou, temos o HTML codificado na visualização). Para ver esta visualização como uma página, precisamos apenas mapeá-la para uma URL (isso será discutido em um próximo capítulo).

Usamos `HttpResponse` para renderizar o HTML na visualização antes. Esta não é a melhor maneira de renderizar páginas. Django suporta o padrão MVT, então para fazer a visão precedente, como Django - MVT, precisaremos -

Um modelo: `myapp/templates/hello.html`

E agora nossa visão será semelhante a -

```
from django.shortcuts import render

def hello(request):
    return render(request, "myapp/template/hello.html", {})
```

As visualizações também podem aceitar parâmetros -

```
from django.http import HttpResponse
```

```
def hello(request, number):  
    text = "<h1>welcome to my app number %s!</h1>" % number  
    return HttpResponse(text)
```

Quando vinculada a uma URL, a página exibirá o número passado como parâmetro. Observe que os parâmetros serão passados através da URL (discutido no próximo capítulo).

Django - Mapeamento de URL

Agora que temos uma visão de trabalho conforme explicado nos capítulos anteriores. Queremos acessar essa visualização por meio de um URL. Django tem seu próprio caminho para mapeamento de URL e isso é feito editando o arquivo `url.py` do seu projeto (**myproject/url.py**) . O arquivo `url.py` se parece com -

```
from django.conf.urls import patterns, include, url  
from django.contrib import admin  
admin.autodiscover()  
  
urlpatterns = patterns('',  
    #Examples  
    #url(r'^$', 'myproject.view.home', name = 'home'),  
    #url(r'^blog/', include('blog.urls')),  
  
    url(r'^admin', include(admin.site.urls)),  
)
```

Quando um usuário faz uma solicitação para uma página em seu aplicativo web, o controlador Django assume a tarefa de procurar a visualização correspondente por meio do arquivo `url.py` e, em seguida, retorna a resposta HTML ou um erro 404 não encontrado, se não for encontrado. Em `url.py`, o mais importante é a tupla **"urlpatterns"** . É onde você define o mapeamento entre URLs e visualizações. Um mapeamento é uma tupla em padrões de URL como -

```
from django.conf.urls import patterns, include, url  
from django.contrib import admin  
admin.autodiscover()  
  
urlpatterns = patterns('',  
    #Examples  
    #url(r'^$', 'myproject.view.home', name = 'home'),  
    #url(r'^blog/', include('blog.urls')),
```

```
url(r'^admin/', include(admin.site.urls)),
url(r'^hello/', 'myapp.views.hello', name = 'hello'),
)
```

A linha marcada mapeia a URL `"/home"` para a visualização `hello` criada no arquivo `myapp/view.py`. Como você pode ver acima, um mapeamento é composto por três elementos -

- **The pattern** - Um regexp correspondente ao URL que você deseja resolver e mapear. Tudo o que pode funcionar com o módulo python `'re'` é elegível para o padrão (útil quando você deseja passar parâmetros via url).
- **The python path to the view** - O mesmo que quando você está importando um módulo.
- **The name** - Para realizar a reversão de URL, você precisará usar padrões de URL nomeados, conforme feito nos exemplos acima. Feito isso, basta iniciar o servidor para acessar sua visualização via: `http://127.0.0.1/hello`

Organizando seus URLs

Até agora, criamos as URLs no arquivo `"myprojects/url.py"`, porém como dito anteriormente sobre Django e criação de um aplicativo, o melhor ponto era poder reutilizar aplicativos em diferentes projetos. Você pode ver facilmente qual é o problema se estiver salvando todos os seus URLs no arquivo `"projecturl.py"`. Portanto, a prática recomendada é criar um `"url.py"` por aplicativo e incluí-lo em nosso arquivo `url.py` de projetos principais (incluímos URLs de administrador para a interface de administração antes).



Como isso é feito?

Precisamos criar um arquivo `url.py` em `myapp` usando o seguinte código -

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

urlpatterns = patterns('', url(r'^hello/', 'myapp.views.hello', name = 'hello'),)
```

Então myproject/url.py mudará para o seguinte -

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

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

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

Incluímos todos os URLs do aplicativo myapp. O home.html que era acessado através de “/hello” agora é “/myapp/hello”, que é uma estrutura melhor e mais compreensível para o aplicativo web.



Agora vamos imaginar que temos outra visualização em myapp “morning” e queremos mapeá-la em myapp/url.py, então mudaremos nosso myapp/url.py para -

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

urlpatterns = patterns('',
    url(r'^hello/', 'myapp.views.hello', name = 'hello'),
    url(r'^morning/', 'myapp.views.morning', name = 'morning'),
)
```

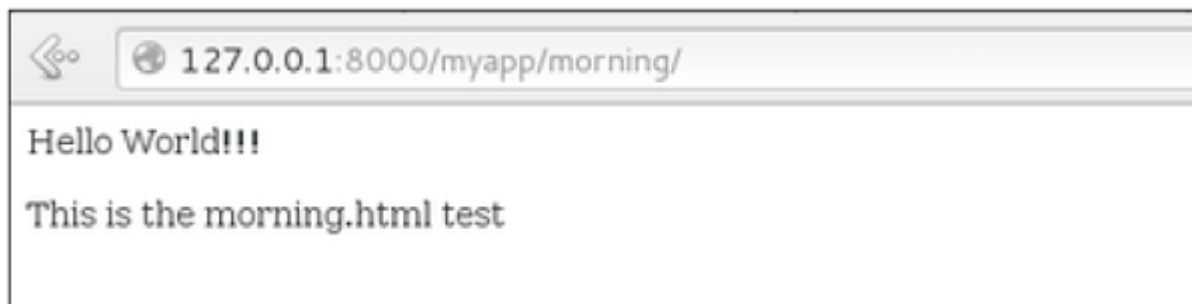
Isso pode ser refatorado para -

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

urlpatterns = patterns('myapp.views',
```

```
url(r'^hello/', 'hello', name = 'hello'),  
url(r'^morning/', 'morning', name = 'morning'))
```

Como você pode ver, agora usamos o primeiro elemento da nossa tupla **urlpatterns** . Isso pode ser útil quando você deseja alterar o nome do seu aplicativo.



Enviando parâmetros para visualizações

Agora sabemos como mapear URLs, como organizá-las, agora vamos ver como enviar parâmetros para visualizações. Um exemplo clássico é o exemplo do artigo (você deseja acessar um artigo via `"/articles/article_id"`).

A passagem de parâmetros é feita capturando-os com o **regex** no padrão URL. Se tivermos uma visualização como a seguinte em `"myapp/view.py"`

```
from django.shortcuts import render  
from django.http import HttpResponse  
  
def hello(request):  
    return render(request, "hello.html", {})  
  
def viewArticle(request, articleId):  
    text = "Displaying article Number : %s"%articleId  
    return HttpResponse(text)
```

Queremos mapeá-lo em `myapp/url.py` para que possamos acessá-lo via `"/myapp/article/articleId"`, precisamos do seguinte em `"myapp/url.py"` -

```
from django.conf.urls import patterns, include, url  
  
urlpatterns = patterns('myapp.views',  
    url(r'^hello/', 'hello', name = 'hello'),  
    url(r'^morning/', 'morning', name = 'morning'),  
    url(r'^article/(\d+)/', 'viewArticle', name = 'article'))
```

Quando o Django vir a url: `"/myapp/article/42"` ele passará os parâmetros '42' para a view `viewArticle`, e no seu navegador você deverá obter o seguinte resultado -



Observe que a ordem dos parâmetros é importante aqui. Suponha que queiramos a lista de artigos de um mês de um ano, vamos adicionar uma visualização `viewArticles`. Nosso `view.py` se torna -

```
from django.shortcuts import render
from django.http import HttpResponse

def hello(request):
    return render(request, "hello.html", {})

def viewArticle(request, articleId):
    text = "Displaying article Number : %s"%articleId
    return HttpResponse(text)

def viewArticles(request, month, year):
    text = "Displaying articles of : %s/%s"%(year, month)
    return HttpResponse(text)
```

O arquivo `url.py` correspondente será semelhante a -

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

urlpatterns = patterns('myapp.views',
    url(r'^hello/', 'hello', name = 'hello'),
    url(r'^morning/', 'morning', name = 'morning'),
    url(r'^article/(\d+)/', 'viewArticle', name = 'article'),
    url(r'^articles/(\d{2})/(\d{4})', 'viewArticles', name = 'articles'),)
```

Agora, quando você for para `"/myapp/articles/12/2006/"` você obterá 'Exibindo artigos de: 2006/12', mas se você inverter os parâmetros não obterá o mesmo resultado.



Para evitar isso, é possível vincular um parâmetro de URL ao parâmetro de visualização. Para isso, nosso **url.py** se tornará -

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

urlpatterns = patterns('myapp.views',
    url(r'^hello/', 'hello', name = 'hello'),
    url(r'^morning/', 'morning', name = 'morning'),
    url(r'^article/(\d+)/', 'viewArticle', name = 'article'),
    url(r'^articles/(?P\d{2})/(?P\d{4})', 'viewArticles', name = 'articles'),)
```

Django - Sistema de Modelos

O Django possibilita separar python e HTML, o python vai nas views e o HTML vai nos templates. Para vincular os dois, o Django depende da função render e da linguagem Django Template.

A função de renderização

Esta função leva três parâmetros -

- **Request** - A solicitação inicial.
- **The path to the template** - Este é o caminho relativo à opção TEMPLATE_DIRS nas variáveis settings.py do projeto.
- **Dicionário de parâmetros** - Um dicionário que contém todas as variáveis necessárias no modelo. Esta variável pode ser criada ou você pode usar locals() para passar todas as variáveis locais declaradas na view.

Linguagem de modelo Django (DTL)

O mecanismo de template do Django oferece uma minilinguagem para definir a camada da aplicação voltada para o usuário.

Exibindo variáveis

Uma variável se parece com isto: `{{variável}}`. O template substitui a variável pela variável enviada pela view no terceiro parâmetro da função `render`. Vamos mudar nosso `hello.html` para exibir a data de hoje -

olá.html

```
<html>

  <body>
    Hello World!!!<p>Today is {{today}}</p>
  </body>

</html>
```

Então nossa visão mudará para -

```
def hello(request):
    today = datetime.datetime.now().date()
    return render(request, "hello.html", {"today" : today})
```

Agora obteremos a seguinte saída após acessar o URL `/myapp/hello` -

```
Hello World!!!
Today is Sept. 11, 2015
```

Como você provavelmente notou, se a variável não for uma string, o Django usará o método `__str__` para exibi-la; e com o mesmo princípio você pode acessar um atributo de objeto da mesma forma que faz em Python. Por exemplo: se quiséssemos exibir a data ano, minha variável seria: `{{hoje.ano}}`.

Filtros

Eles ajudam você a modificar variáveis no momento da exibição. A estrutura dos filtros é semelhante a esta: `{{var|filters}}`.

Alguns exemplos -

- **{{string|truncatewords:80}}** - Este filtro truncará a string, então você verá apenas as primeiras 80 palavras.
- **{{string|lower}}** - Converte a string em minúsculas.
- **{{string|escape|linebreaks}}** - Escapa o conteúdo da string e converte quebras de linha em tags.

Você também pode definir o padrão para uma variável.

Tag

Tags permitem realizar as seguintes operações: condição if, loop for, herança de modelo e muito mais.

Marcar se

Assim como em Python, você pode usar if, else e elif em seu modelo -

```
<html>
  <body>

    Hello World!!!<p>Today is {{today}}</p>
    We are
    {% if today.day == 1 %}

      the first day of month.
    {% elif today.day == 30 %}

      the last day of month.
    {% else %}

      I don't know.
    {%endif%}

  </body>
</html>
```

Neste novo template, dependendo da data do dia, o template irá renderizar um determinado valor.

Etiqueta para

Assim como 'if', temos a tag 'for', que funciona exatamente como em Python. Vamos mudar nossa visualização hello para transmitir uma lista ao nosso modelo -

```
def hello(request):
    today = datetime.datetime.now().date()

    daysOfWeek = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
    return render(request, "hello.html", {"today" : today, "days_of_week" : daysOfWeek
```

O modelo para exibir essa lista usando {{ for }} -

```
<html>
  <body>

    Hello World!!!<p>Today is {{today}}</p>
    We are
    {% if today.day == 1 %}

    the first day of month.
    {% elif today.day == 30 %}

    the last day of month.
    {% else %}

    I don't know.
    {%endif%}

    <p>
      {% for day in days_of_week %}
      {{day}}
    </p>

    {% endfor %}

  </body>
</html>
```

E deveríamos conseguir algo como -

```
Hello World!!!
Today is Sept. 11, 2015
We are I don't know.
Mon
Tue
Wed
Thu
Fri
```



Sat

Sun

Bloquear e estender tags

Um sistema de templates não pode ser completo sem herança de templates. Ou seja, quando você estiver projetando seus modelos, você deve ter um modelo principal com buracos que o modelo da criança irá preencher de acordo com sua necessidade, como uma página pode precisar de um CSS especial para a guia selecionada.

Vamos alterar o modelo `hello.html` para herdar de `main_template.html`.

`main_template.html`

```
<html>
  <head>

    <title>
      {% block title %}Page Title{% endblock %}
    </title>

  </head>

  <body>

    {% block content %}
      Body content
    {% endblock %}

  </body>
</html>
```

`olá.html`

```
{% extends "main_template.html" %}
{% block title %}My Hello Page{% endblock %}
{% block content %}

Hello World!!!<p>Today is {{today}}</p>
We are
{% if today.day == 1 %}

the first day of month.
{% elif today.day == 30 %}
```



```

the last day of month.
{% else %}

I don't know.
{%endif%}

<p>
    {% for day in days_of_week %}
        {{day}}
    </p>

{% endfor %}
{% endblock %}

```

No exemplo acima, ao chamar /myapp/hello ainda obteremos o mesmo resultado de antes, mas agora contamos com extensões e blocos para refatorar nosso código -

No main_template.html definimos blocos usando a tag block. O bloco de título conterá o título da página e o bloco de conteúdo conterá o conteúdo principal da página. Em home.html usamos extends para herdar de main_template.html e então preenchemos o bloco definido acima (conteúdo e título).

Etiqueta de comentário

A tag de comentário ajuda a definir comentários em modelos, não em comentários HTML, eles não aparecerão na página HTML. Pode ser útil para documentação ou apenas para comentar uma linha de código.

Django - Modelos

Um modelo é uma classe que representa uma tabela ou coleção em nosso banco de dados, e onde cada atributo da classe é um campo da tabela ou coleção. Os modelos são definidos em app/models.py (em nosso exemplo: myapp/models.py)

Criando um modelo

A seguir está um modelo Dreamreal criado como exemplo -

```

from django.db import models

class Dreamreal(models.Model):

    website = models.CharField(max_length = 50)
    mail = models.CharField(max_length = 50)
    name = models.CharField(max_length = 50)
    phonenumber = models.IntegerField()

```

```
class Meta:
    db_table = "dreamreal"
```

Todo modelo herda de `django.db.models.Model`.

Nossa classe possui 4 atributos (3 CharField e 1 Integer), esses serão os campos da tabela.

A classe Meta com o atributo `db_table` nos permite definir a tabela real ou o nome da coleção. O Django nomeia a tabela ou coleção automaticamente: `myapp_modelName`. Esta classe permitirá que você force o nome da tabela como desejar.

Depois de criar seu modelo, você precisará do Django para gerar o banco de dados real -

```
$python manage.py syncdb
```

Manipulando Dados (CRUD)

Vamos criar uma visão "crudops" para ver como podemos fazer operações CRUD em modelos. Nosso `myapp/views.py` ficará assim:

meuapp/views.py

```
from myapp.models import Dreamreal
from django.http import HttpResponse

def crudops(request):
    #Creating an entry

    dreamreal = Dreamreal(
        website = "www.polo.com", mail = "sorex@polo.com",
        name = "sorex", phonenumber = "002376970"
    )

    dreamreal.save()

    #Read ALL entries
    objects = Dreamreal.objects.all()
    res = 'Printing all Dreamreal entries in the DB : <br>'

    for elt in objects:
        res += elt.name+"<br>"

    #Read a specific entry:
    sorex = Dreamreal.objects.get(name = "sorex")
    res += 'Printing One entry <br>'
```



```

res += sorex.name

#Delete an entry
res += '<br> Deleting an entry <br>'
sorex.delete()

#Update
dreamreal = Dreamreal(
    website = "www.polo.com", mail = "sorex@polo.com",
    name = "sorex", phonenumber = "002376970"
)

dreamreal.save()
res += 'Updating entry<br>'

dreamreal = Dreamreal.objects.get(name = 'sorex')
dreamreal.name = 'thierry'
dreamreal.save()

return HttpResponse(res)

```

Outra manipulação de dados

Vamos explorar outras manipulações que podemos fazer nos Modelos. Observe que as operações CRUD foram feitas em instâncias do nosso modelo, agora estaremos trabalhando diretamente com a classe que representa o nosso modelo.

Vamos criar uma visualização de 'manipulação de dados' em **myapp/views.py**

```

from myapp.models import Dreamreal
from django.http import HttpResponse

def datamanipulation(request):
    res = ''

    #Filtering data:
    qs = Dreamreal.objects.filter(name = "paul")
    res += "Found : %s results<br>"%len(qs)

    #Ordering results
    qs = Dreamreal.objects.order_by("name")

    for elt in qs:
        res += elt.name + '<br>'

```



```
return HttpResponse(res)
```

Vinculando modelos

Django ORM oferece 3 maneiras de vincular modelos -

Um dos primeiros casos que veremos aqui são os relacionamentos um-para-muitos. Como você pode ver no exemplo acima, a empresa Dreamreal pode ter vários sites online. A definição dessa relação é feita usando `django.db.models.ForeignKey` -

meuapp/models.py

```
from django.db import models

class Dreamreal(models.Model):
    website = models.CharField(max_length = 50)
    mail = models.CharField(max_length = 50)
    name = models.CharField(max_length = 50)
    phonenumber = models.IntegerField()
    online = models.ForeignKey('Online', default = 1)

    class Meta:
        db_table = "dreamreal"

class Online(models.Model):
    domain = models.CharField(max_length = 30)

    class Meta:
        db_table = "online"
```

Como você pode ver em nosso `myapp/models.py` atualizado, adicionamos o modelo `online` e o vinculamos ao nosso modelo `Dreamreal`.

Vamos verificar como tudo isso está funcionando por meio do shell `manager.py` -

Primeiro vamos criar algumas empresas (entradas do `Dreamreal`) para testar em nosso shell Django -

```
$python manage.py shell

>>> from myapp.models import Dreamreal, Online
>>> dr1 = Dreamreal()
>>> dr1.website = 'company1.com'
>>> dr1.name = 'company1'
>>> dr1.mail = 'contact@company1'
```



```
>>> dr1.phonenumber = '12345'
>>> dr1.save()
>>> dr2 = Dreamreal()
>>> dr1.website = 'company2.com'
>>> dr2.website = 'company2.com'
>>> dr2.name = 'company2'
>>> dr2.mail = 'contact@company2'
>>> dr2.phonenumber = '56789'
>>> dr2.save()
```

Agora, alguns domínios hospedados -

```
>>> on1 = Online()
>>> on1.company = dr1
>>> on1.domain = "site1.com"
>>> on2 = Online()
>>> on2.company = dr1
>>> on2.domain = "site2.com"
>>> on3 = Online()
>>> on3.domain = "site3.com"
>>> dr2 = Dreamreal.objects.all()[2]
>>> on3.company = dr2
>>> on1.save()
>>> on2.save()
>>> on3.save()
```

Acessar o atributo da empresa de hospedagem (entrada Dreamreal) de um domínio online é simples -

```
>>> on1.company.name
```

E se quisermos conhecer todos os domínios online hospedados por uma empresa no Dreamreal usaremos o código -

```
>>> dr1.online_set.all()
```

Para obter um QuerySet, observe que todos os métodos de manipulação que vimos antes (filtro, todos, exclusão, pedido_por...)

Você também pode acessar os atributos do modelo vinculado para operações de filtragem, digamos que você deseja obter todos os domínios online onde o nome Dreamreal contém 'empresa' -

```
>>> Online.objects.filter(company__name__contains = 'company')
```

Note - Esse tipo de consulta é compatível apenas com banco de dados SQL. Não funcionará para bancos de dados não relacionais onde não existem junções e há dois '_'.

Mas essa não é a única forma de vincular modelos, você também tem o `OneToOneField`, um link que garante que a relação entre dois objetos é única. Se usássemos o `OneToOneField` em nosso exemplo acima, isso significaria que para cada entrada do Dreamreal apenas uma entrada online é possível e vice-versa.

E o último, a relação `ManyToManyField` for (nn) entre tabelas. Observe que eles são relevantes para banco de dados baseado em SQL.

Django - Redirecionamento de página

O redirecionamento de página é necessário por vários motivos em aplicativos da web. Você pode querer redirecionar um usuário para outra página quando ocorrer uma ação específica ou basicamente em caso de erro. Por exemplo, quando um usuário faz login no seu site, muitas vezes ele é redirecionado para a página inicial principal ou para o seu painel pessoal. No Django, o redirecionamento é realizado usando o método `'redirect'`.

O método `'redirect'` recebe como argumento: A URL para a qual você deseja ser redirecionado como string O nome de uma visualização.

O `myapp/views` se parece com o seguinte até agora -

```
def hello(request):
    today = datetime.datetime.now().date()
    daysOfWeek = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
    return render(request, "hello.html", {"today" : today, "days_of_week" : daysOfWeek})

def viewArticle(request, articleId):
    """ A view that display an article based on his ID"""
    text = "Displaying article Number : %s" %articleId
    return HttpResponse(text)

def viewArticles(request, year, month):
    text = "Displaying articles of : %s/%s"%(year, month)
    return HttpResponse(text)
```

Vamos mudar a visualização `hello` para redirecionar para `djangoproject.com` e nosso `viewArticle` para redirecionar para nosso `'/myapp/articles'` interno. Para fazer isso, `myapp/view.py` mudará para -

```
from django.shortcuts import render, redirect
from django.http import HttpResponse
import datetime
```

Create your views here.

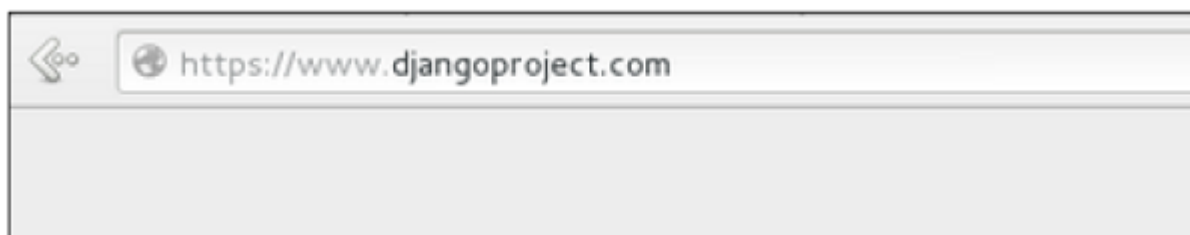
```
def hello(request):
    today = datetime.datetime.now().date()
    daysOfWeek = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
    return redirect("https://www.djangoproject.com")

def viewArticle(request, articleId):
    """ A view that display an article based on his ID"""
    text = "Displaying article Number : %s" %articleId
    return redirect(viewArticles, year = "2045", month = "02")

def viewArticles(request, year, month):
    text = "Displaying articles of : %s/%s"%(year, month)
    return HttpResponse(text)
```

No exemplo acima, primeiro importamos o redirecionamento de `django.shortcuts` e para o redirecionamento para o site oficial do Django apenas passamos a URL completa para o método 'redirect' como string, e para o segundo exemplo (a visualização `viewArticle`) o 'redirect' O método recebe o nome da visualização e seus parâmetros como argumentos.

Acessar `/myapp/hello`, você verá a seguinte tela -



E acessando `/myapp/article/42`, você verá a seguinte tela -



Também é possível especificar se o 'redirecionamento' é temporário ou permanente adicionando o parâmetro `permanente = True`. O usuário não verá diferença, mas esses são detalhes que os mecanismos de busca levam em consideração na hora de classificar o seu site.

Lembre-se também do parâmetro 'nome' que definimos em nosso `url.py` ao mapear os URLs -

```
url(r'^articles/(?P\d{2})/(?P\d{4})/', 'viewArticles', name = 'articles'),
```



Esse nome (aqui artigo) pode ser usado como argumento para o método 'redirect', então nosso redirecionamento `viewArticle` pode ser alterado de -

```
def viewArticle(request, articleId):  
    """ A view that display an article based on his ID"""  
    text = "Displaying article Number : %s" %articleId  
    return redirect(viewArticles, year = "2045", month = "02")
```

Para -

```
def viewArticle(request, articleId):  
    """ A view that display an article based on his ID"""  
    text = "Displaying article Number : %s" %articleId  
    return redirect(articles, year = "2045", month = "02")
```

Note - Existe também uma função para gerar URLs; é usado da mesma forma que redirecionamento; o método 'reverso' (`django.core.urlresolvers.reverse`). Esta função não retorna um objeto `HttpResponseRedirect`, mas simplesmente uma string contendo a URL para a visualização compilada com qualquer argumento passado.

Django - Enviando E-mails

Django vem com um mecanismo leve pronto e fácil de usar para enviar e-mail. Semelhante ao Python, você só precisa importar o `smtplib`. No Django você só precisa importar `django.core.mail`. Para começar a enviar e-mail, edite o arquivo `settings.py` do seu projeto e defina as seguintes opções -

- **EMAIL_HOST** - servidor smtp.
- **EMAIL_HOST_USER** - Credencial de login para o servidor smtp.
- **EMAIL_HOST_PASSWORD** - Credencial de senha para o servidor smtp.
- **EMAIL_PORT** - porta do servidor smtp.
- **EMAIL_USE_TLS ou _SSL** - Verdadeiro se a conexão for segura.

Enviando um e-mail simples

Vamos criar uma view "sendSimpleEmail" para enviar um e-mail simples.

```
from django.core.mail import send_mail  
from django.http import HttpResponseRedirect  
  
def sendSimpleEmail(request, emailto):
```



```
res = send_mail("hello paul", "comment tu vas?", "paul@polo.com", [emailto])
return HttpResponse('%s'%res)
```

Aqui estão os detalhes dos parâmetros de send_mail -

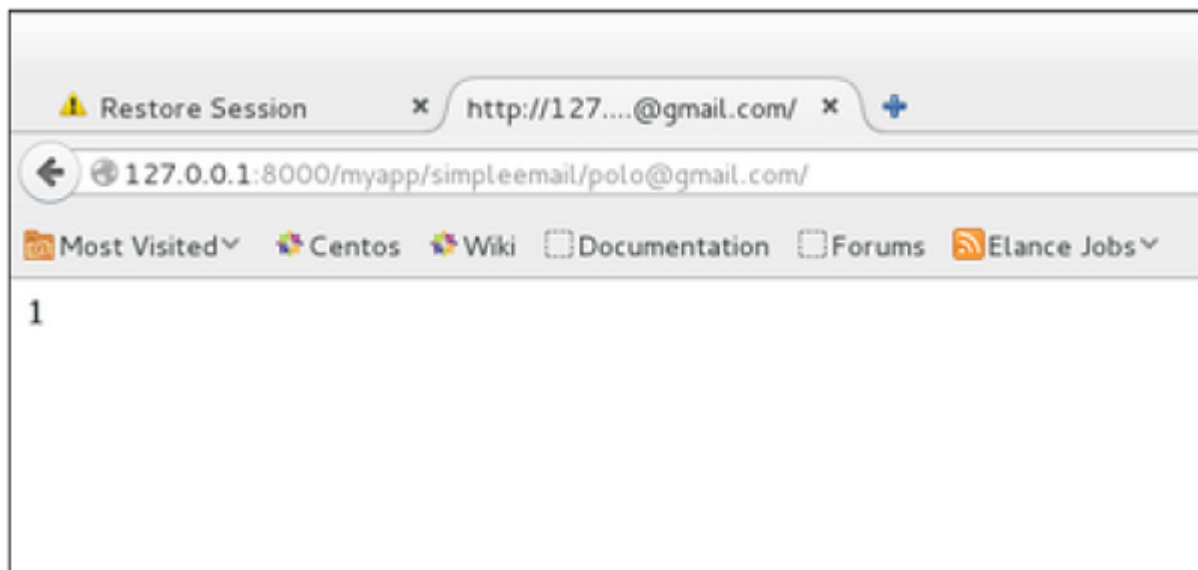
- **subject** - Assunto do e-mail.
- **message** - Corpo do e-mail.
- **from_email** - E-mail de.
- **destinatário_list** - Lista de endereços de e-mail dos destinatários.
- **fail_silently** - Bool, se for falso send_mail irá gerar uma exceção em caso de erro.
- **auth_user** - Login do usuário se não estiver definido em settings.py.
- **auth_password** - Senha do usuário se não estiver definida em settings.py.
- **connection** - back-end de e-mail.
- **html_message** - (novo no Django 1.7) se presente, o e-mail será multipart/alternativo.

Vamos criar uma URL para acessar nossa visualização -

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

urlpatterns = patterns('myapp.views', url(r'^simpleemail/(?P<emailto>
[\w.%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4})/',
'sendSimpleEmail' , name = 'sendSimpleEmail'),)
```

Portanto, ao acessar /myapp/simpleemail/polo@gmail.com, você obterá a seguinte página -



Enviando vários e-mails com send_mass_mail

O método retorna o número de mensagens entregues com sucesso. É o mesmo que `send_mail`, mas leva um parâmetro extra; `datatuple`, nossa visualização `sendMassEmail` será então -

```
from django.core.mail import send_mass_mail
from django.http import HttpResponse

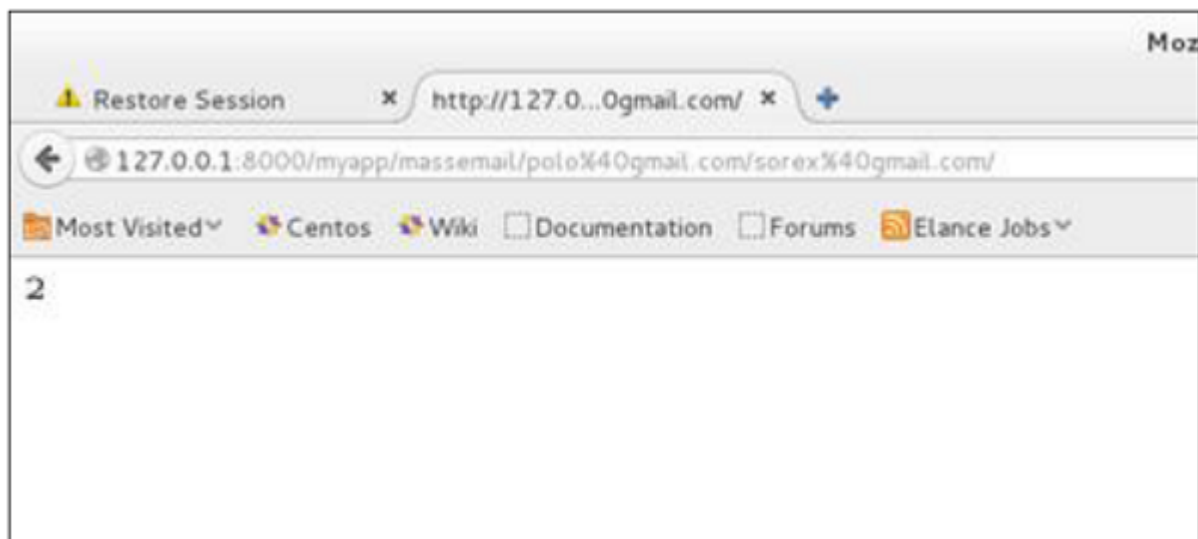
def sendMassEmail(request, emailto):
    msg1 = ('subject 1', 'message 1', 'polo@polo.com', [emailto1])
    msg2 = ('subject 2', 'message 2', 'polo@polo.com', [emailto2])
    res = send_mass_mail((msg1, msg2), fail_silently = False)
    return HttpResponse('%s'%res)
```

Vamos criar uma URL para acessar nossa visualização -

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

urlpatterns = patterns('myapp.views', url(r'^massEmail/(?P<emailto1>
[\w.%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4})/(?P<emailto2>
[\w.%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4})', 'sendMassEmail' , name = 'sendMassEmail
```

Ao acessar `/myapp/massemail/polo@gmail.com/sorex@gmail.com/`, obtemos -



Os detalhes dos parâmetros `send_mass_mail` são -

- **datatuples** - Uma tupla onde cada elemento é semelhante (assunto, mensagem, from_email, destinatário_list).
- **fail_silently** - Bool, se for falso send_mail irá gerar uma exceção em caso de erro.
- **auth_user** - Login do usuário se não estiver definido em settings.py.
- **auth_password** - Senha do usuário se não estiver definida em settings.py.
- **connection** - back-end de e-mail.

Como você pode ver na imagem acima, duas mensagens foram enviadas com sucesso.

Nota - Neste exemplo, estamos usando o servidor de depuração smtp Python, que você pode iniciar usando -

```
$python -m smtpd -n -c DebuggingServer localhost:1025
```

Isso significa que todos os seus e-mails enviados serão impressos em stdout e o servidor fictício será executado em localhost:1025.

Envio de e-mails para administradores e gerentes usando os métodos mail_admins e mail_managers

Esses métodos enviam e-mails para administradores de site conforme definido na opção ADMINS do arquivo settings.py e para gerentes de site conforme definido na opção MANAGERS do arquivo settings.py. Vamos supor que nossas opções de ADMINS e MANAGERS sejam semelhantes a -

ADMINISTRADORES = (('polo', 'polo@polo.com'),)

GERENTES = (('popoli', 'popoli@polo.com'),)

```
from django.core.mail import mail_admins
from django.http import HttpResponseRedirect

def sendAdminsEmail(request):
    res = mail_admins('my subject', 'site is going down.')
    return HttpResponseRedirect('%s'%res)
```

O código acima enviará um e-mail para todos os administradores definidos na seção ADMINS.

```
from django.core.mail import mail_managers
from django.http import HttpResponseRedirect

def sendManagersEmail(request):
```



```
res = mail_managers('my subject 2', 'Change date on the site.')
return HttpResponse('%s'%res)
```

O código acima enviará um e-mail para cada gerente definido na seção GERENCIADORES.

Detalhes dos parâmetros -

- **Assunto** - Assunto do e-mail.
- **message** - Corpo do e-mail.
- **fail_silently** - Bool, se for falso send_mail irá gerar uma exceção em caso de erro.
- **connection** - back-end de e-mail.
- **html_message** - (novo no Django 1.7) se presente, o e-mail será multipart/alternativo.

Enviando e-mail HTML

Enviar mensagem HTML no Django >= 1.7 é tão fácil quanto -

```
from django.core.mail import send_mail

from django.http import HttpResponse
res = send_mail("hello paul", "comment tu vas?", "paul@polo.com",
               ["polo@gmail.com"], html_message=)
```

Isso produzirá um e-mail multiparte/alternativo.

Mas para Django <1.7, o envio de mensagens HTML é feito através da classe django.core.mail.EmailMessage e depois chamando 'send' no objeto -

Vamos criar uma view "sendHTMLEmail" para enviar um e-mail em HTML.

```
from django.core.mail import EmailMessage
from django.http import HttpResponse

def sendHTMLEmail(request , emailto):
    html_content = "<strong>Comment tu vas?</strong>"
    email = EmailMessage("my subject", html_content, "paul@polo.com", [emailto])
    email.content_subtype = "html"
    res = email.send()
    return HttpResponse('%s'%res)
```

Detalhes dos parâmetros para a criação da classe EmailMessage -

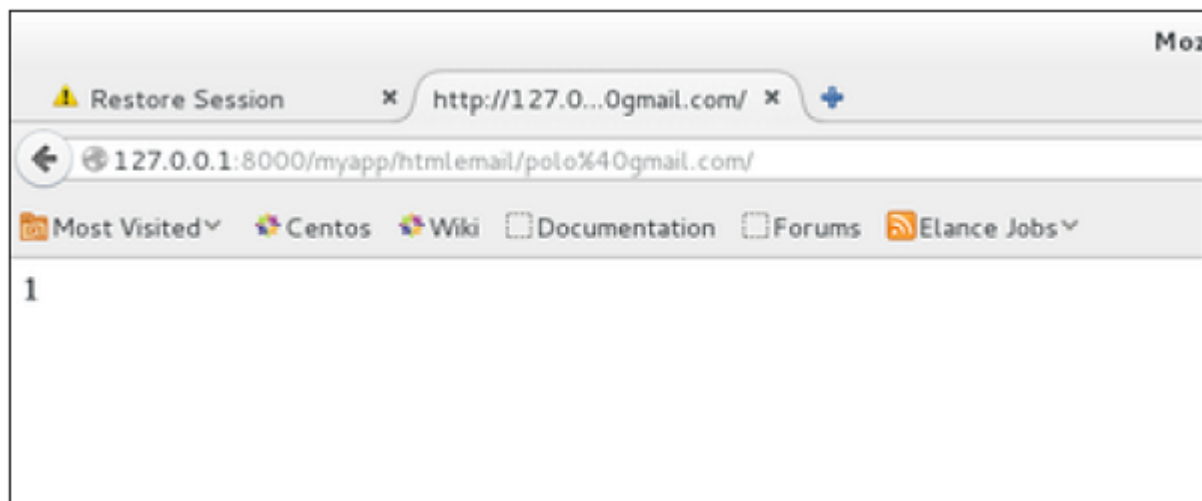
- **Assunto** - Assunto do e-mail.
- **message** - Corpo do e-mail em HTML.
- **from_email** - E-mail de.
- **to** - Lista de endereços de e-mail dos destinatários.
- **bcc** - Lista de endereços de e-mail dos destinatários "Cco".
- **connection** - back-end de e-mail.

Vamos criar uma URL para acessar nossa visualização -

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

urlpatterns = patterns('myapp.views', url(r'^htmlemail/(?P<emailto>
[\w.%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4})/',
'sendHTMLEmail' , name = 'sendHTMLEmail'),)
```

Ao acessar /myapp/htmlemail/polo@gmail.com, obtemos -



Envio de e-mail com anexo

Isso é feito usando o método 'attach' no objeto EmailMessage.

A forma de enviar um e-mail com anexo será -

```
from django.core.mail import EmailMessage
from django.http import HttpResponse

def sendEmailWithAttach(request, emailto):
    html_content = "Comment tu vas?"
    email = EmailMessage("my subject", html_content, "paul@polo.com", emailto])
    email.content_subtype = "html"

    fd = open('manage.py', 'r')
```

```
email.attach('manage.py', fd.read(), 'text/plain')

res = email.send()
return HttpResponse('%s'%res)
```

Detalhes sobre argumentos anexados -

- **filename** - O nome do arquivo a ser anexado.
- **content** - O conteúdo do arquivo a ser anexado.
- **mimetype** - O tipo MIME do conteúdo do anexo.

Django - Visualizações genéricas

Em alguns casos, escrever visualizações, como vimos anteriormente, é muito pesado. Imagine que você precisa de uma página estática ou de listagem. O Django oferece uma maneira fácil de definir essas visualizações simples chamadas de visualizações genéricas.

Ao contrário das visualizações clássicas, as visualizações genéricas são classes e não funções. Django oferece um conjunto de classes para visões genéricas em `django.views.generic`, e cada visão genérica é uma dessas classes ou uma classe que herda de uma delas.

Existem mais de 10 classes genéricas -

```
>>> import django.views.generic
>>> dir(django.views.generic)

['ArchiveIndexView', 'CreateView', 'DateDetailView', 'DayArchiveView',
 'DeleteView', 'DetailView', 'FormView', 'GenericViewError', 'ListView',
 'MonthArchiveView', 'RedirectView', 'TemplateView', 'TodayArchiveView',
 'UpdateView', 'View', 'WeekArchiveView', 'YearArchiveView', '__builtins__',
 '__doc__', '__file__', '__name__', '__package__', '__path__', 'base', 'dates',
 'detail', 'edit', 'list']
```

Isso você pode usar para sua visão genérica. Vejamos alguns exemplos para ver como funciona.

Páginas estáticas

Vamos publicar uma página estática do modelo "static.html".

Nosso static.html -

```
<html>
<body>
```

```
This is a static page!!!  
</body>  
</html>
```

Se fizéssemos isso da maneira que aprendemos antes, teríamos que mudar **myapp/views.py** para -

```
from django.shortcuts import render  
  
def static(request):  
    return render(request, 'static.html', {})
```

e **myapp/urls.py** para ser -

```
from django.conf.urls import patterns, url  
  
urlpatterns = patterns("myapp.views", url(r'^static/', 'static', name = 'static'),)
```

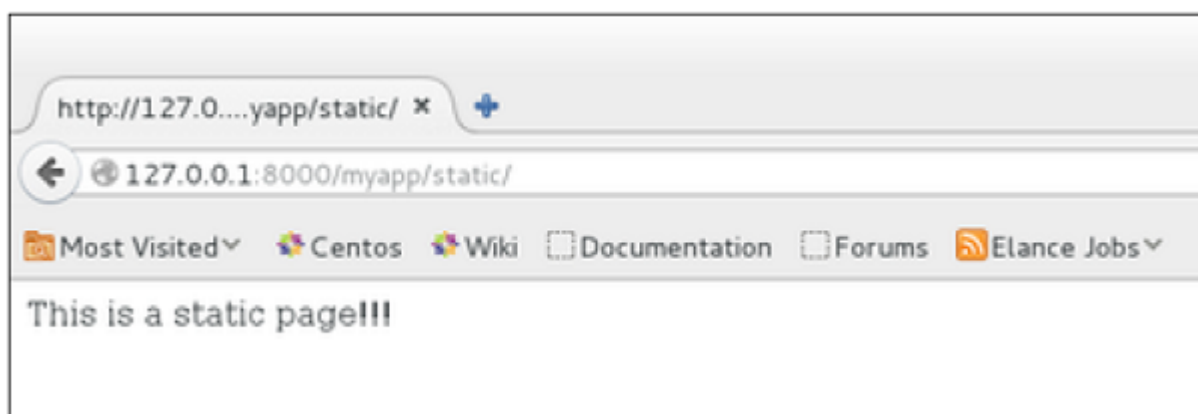
A melhor maneira é usar visualizações genéricas. Para isso, nosso myapp/views.py se tornará -

```
from django.views.generic import TemplateView  
  
class StaticView(TemplateView):  
    template_name = "static.html"
```

E nosso myapp/urls.py seremos -

```
from myapp.views import StaticView  
from django.conf.urls import patterns  
  
urlpatterns = patterns("myapp.views", (r'^static/$', StaticView.as_view()),)
```

Ao acessar /myapp/static você obtém -



Para o mesmo resultado, também podemos fazer o seguinte -

- Nenhuma mudança no views.py
- Altere o arquivo url.py para -

```
from django.views.generic import TemplateView
from django.conf.urls import patterns, url

urlpatterns = patterns("myapp.views",
    url(r'^static/', TemplateView.as_view(template_name = 'static.html')),)
```

Como você pode ver, você só precisa alterar o arquivo url.py no segundo método.

Listar e exibir dados do banco de dados

Listaremos todas as entradas em nosso modelo Dreamreal. Fazer isso é facilitado usando a classe de visualização genérica ListView. Edite o arquivo url.py e atualize-o como -

```
from django.views.generic import ListView
from django.conf.urls import patterns, url

urlpatterns = patterns(
    "myapp.views", url(r'^dreamreals/', ListView.as_view(model = Dreamreal,
        template_name = "dreamreal_list.html")),
)
```

É importante observar neste ponto que a variável passada pela visão genérica para o modelo é object_list. Se você mesmo quiser nomeá-lo, precisará adicionar um argumento context_object_name ao método as_view. Então o url.py se tornará -

```
from django.views.generic import ListView
from django.conf.urls import patterns, url

urlpatterns = patterns("myapp.views",
    url(r'^dreamreals/', ListView.as_view(
        template_name = "dreamreal_list.html")),
    model = Dreamreal, context_object_name = "dreamreals_objects" ,)
```

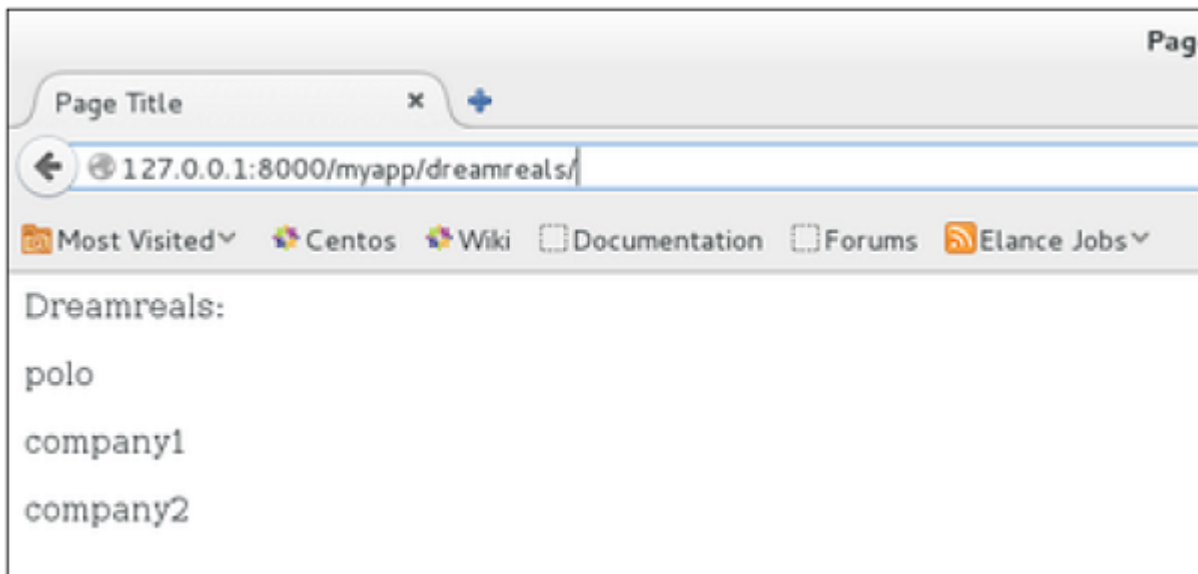
O modelo associado será então -

```
{% extends "main_template.html" %}
{% block content %}
Dreamreals:<p>
```



```
{% for dr in object_list %}
{{dr.name}}</p>
{% endfor %}
{% endblock %}
```

Acessar /myapp/dreamreals/ produzirá a seguinte página -



Django - Processamento de Formulários

Criar formulários no Django é muito semelhante a criar um modelo. Aqui novamente, só precisamos herdar da classe Django e os atributos da classe serão os campos do formulário. Vamos adicionar um arquivo **forms.py** na pasta myapp para conter os formulários do nosso aplicativo. Criaremos um formulário de login.

meuapp/forms.py

```
#-*- coding: utf-8 -*-
from django import forms

class LoginForm(forms.Form):
    user = forms.CharField(max_length = 100)
    password = forms.CharField(widget = forms.PasswordInput())
```

Como visto acima, o tipo de campo pode receber o argumento "widget" para renderização de HTML; no nosso caso, queremos que a senha fique oculta e não exibida. Muitos outros widgets estão presentes no Django: **DateInput** para datas, **CheckboxInput** para caixas de seleção, etc.

Usando formulário em uma visualização

Existem dois tipos de solicitações HTTP, GET e POST. No Django, o objeto request passado como parâmetro para sua view possui um atributo chamado "method" onde é definido o tipo

da requisição, e todos os dados passados via POST podem ser acessados através do dicionário `request.POST`.

Vamos criar uma visualização de login em nosso `myapp/views.py` -

```
#-*- coding: utf-8 -*-
from myapp.forms import LoginForm

def login(request):
    username = "not logged in"

    if request.method == "POST":
        #Get the posted form
        MyLoginForm = LoginForm(request.POST)

        if MyLoginForm.is_valid():
            username = MyLoginForm.cleaned_data['username']
        else:
            MyLoginForm = LoginForm()

    return render(request, 'loggedin.html', {"username" : username})
```

A visualização exibirá o resultado do formulário de login postado através do **login.html** . Para testá-lo, primeiro precisaremos do modelo de formulário de login. Vamos chamá-lo de `login.html`.

```
<html>
  <body>

    <form name = "form" action = "{% url "myapp.views.login" %}"
      method = "POST" >{% csrf_token %}

      <div style = "max-width:470px;">
        <center>
          <input type = "text" style = "margin-left:20%;"
            placeholder = "Identifiant" name = "username" />
        </center>
      </div>

      <br>

      <div style = "max-width:470px;">
        <center>
          <input type = "password" style = "margin-left:20%;"
            placeholder = "password" name = "password" />
        </center>
```



```

</div>

<br>

<div style = "max-width:470px;">
    <center>

        <button style = "border:0px; background-color:#4285F4; margin-top:8%;
            height:35px; width:80%;margin-left:19%;" type = "submit"
            value = "Login" >
                <strong>Login</strong>
        </button>

    </center>
</div>

</form>

</body>
</html>

```

O modelo exibirá um formulário de login e postará o resultado em nossa visualização de login acima. Você provavelmente notou a tag no modelo, que serve apenas para evitar ataques de falsificação de solicitação entre sites (CSRF) em seu site.

```
{% csrf_token %}
```

Assim que tivermos o modelo de login, precisamos do modelo login.html que será renderizado após o tratamento do formulário.

```

<html>

    <body>
        You are : <strong>{{username}}</strong>
    </body>

</html>

```

Agora, só precisamos do nosso par de URLs para começar: myapp/urls.py

```

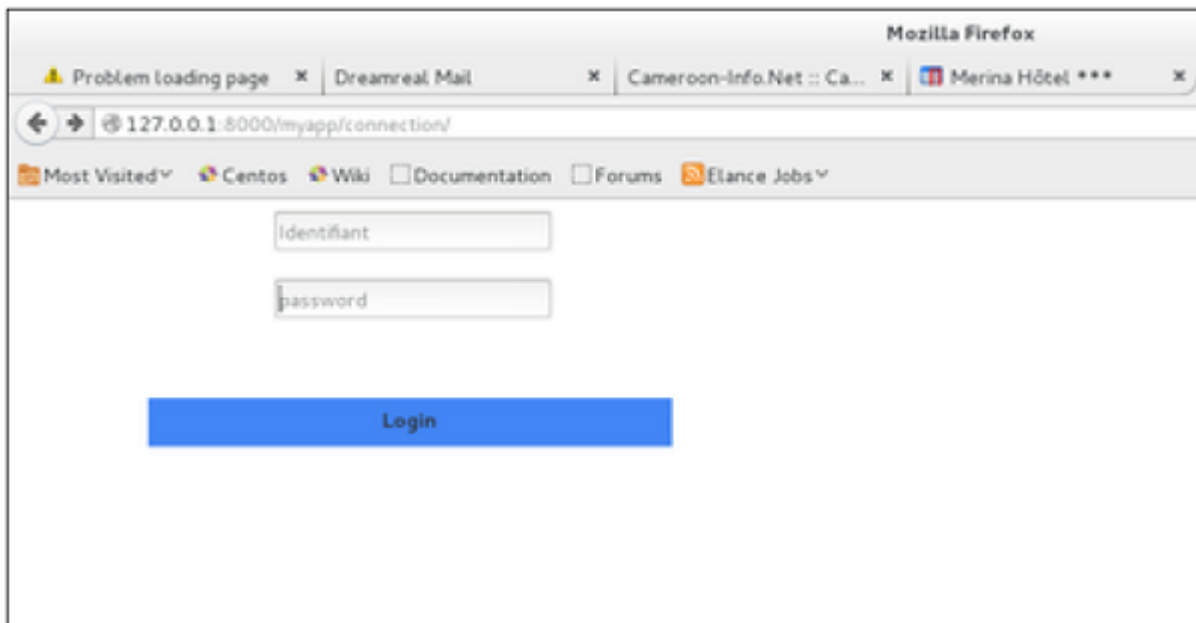
from django.conf.urls import patterns, url
from django.views.generic import TemplateView

urlpatterns = patterns('myapp.views',

```

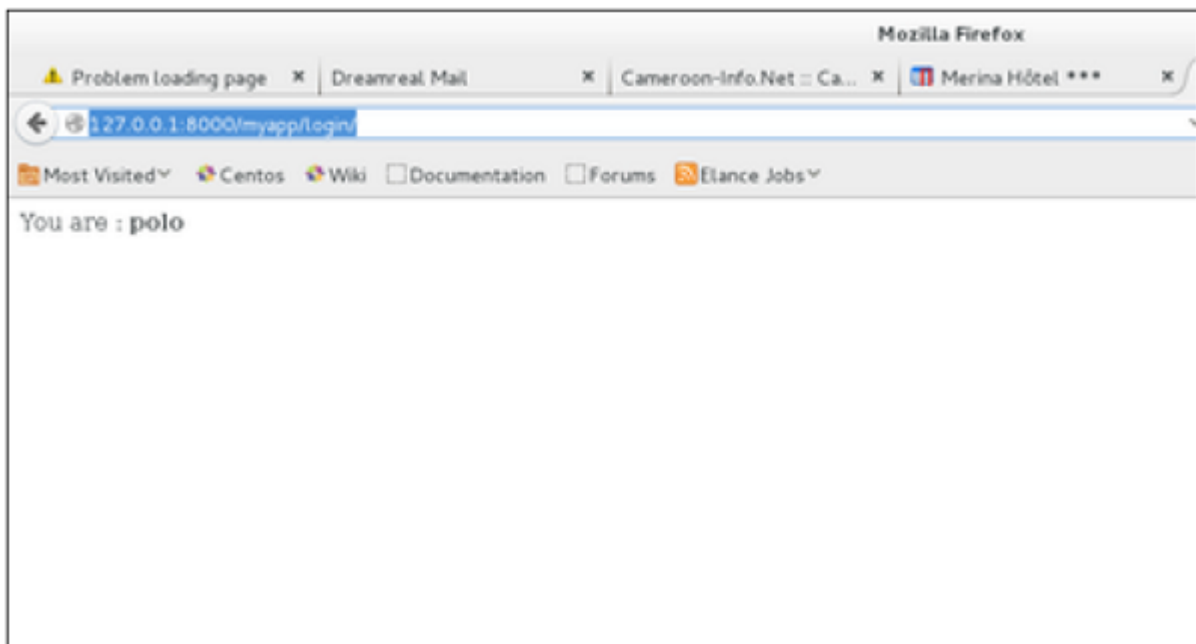
```
url(r'^connection/', TemplateView.as_view(template_name = 'login.html')),  
url(r'^login/', 'login', name = 'login'))
```

Ao acessar `"/myapp/connection"`, obteremos o seguinte modelo `login.html` renderizado -



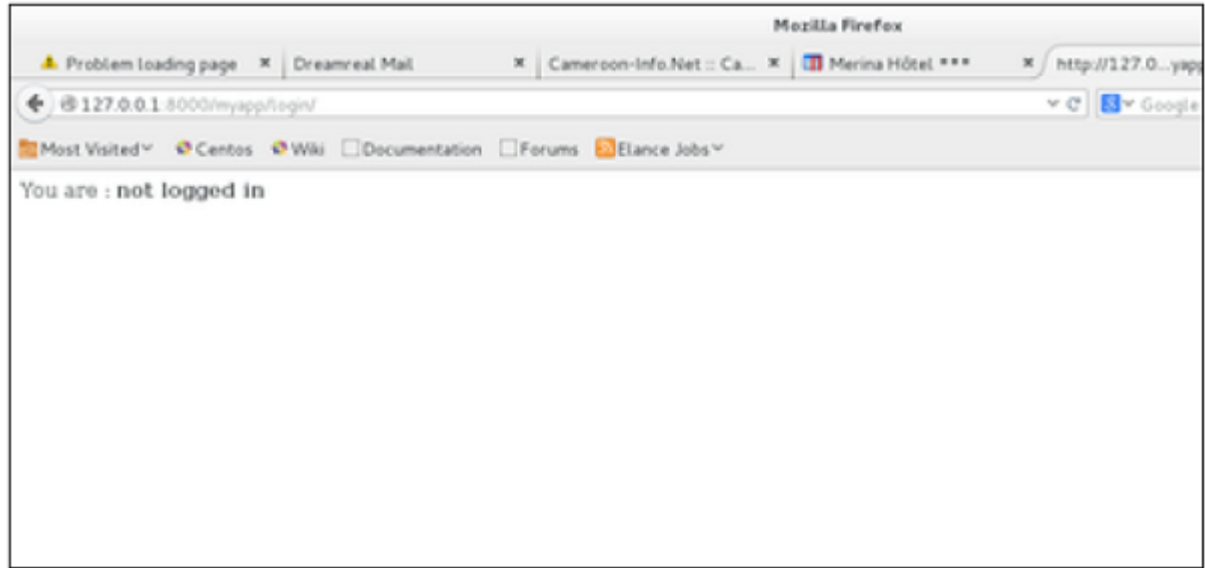
The screenshot shows a Mozilla Firefox browser window with the address bar displaying `127.0.0.1:8000/myapp/connection/`. The page contains a login form with two input fields: "Identifiant" and "password". Below these fields is a blue button labeled "Login". The browser's tab bar shows several open tabs, including "Problem loading page", "Dreamreal Mail", "Cameroon-Info.Net :: Ca...", and "Merina Hôtel ***". The bookmarks bar includes "Most Visited", "Centos", "Wiki", "Documentation", "Forums", and "Elance Jobs".

Na postagem do formulário, o formulário é válido. No nosso caso, certifique-se de preencher os dois campos e você obterá -



The screenshot shows the same Mozilla Firefox browser window, but the address bar now displays `127.0.0.1:8000/myapp/login`. The page content has changed to show a confirmation message: "You are : polo". The browser's tab bar and bookmarks bar remain the same as in the previous screenshot.

Caso seu nome de usuário seja polo e você tenha esquecido a senha. Você receberá a seguinte mensagem -



Usando nossa própria validação de formulário

No exemplo acima, ao validar o formulário -

```
MyLoginForm.is_valid()
```

Usamos apenas o mecanismo de validação de autoformulário Django, em nosso caso apenas certificando-nos de que os campos são obrigatórios. Agora vamos tentar garantir que o usuário que está tentando fazer login esteja presente em nosso banco de dados como entrada Dreamreal. Para isso, altere `myapp/forms.py` para -

```
#-*- coding: utf-8 -*-
from django import forms
from myapp.models import Dreamreal

class LoginForm(forms.Form):
    user = forms.CharField(max_length = 100)
    password = forms.CharField(widget = forms.PasswordInput())

    def clean_message(self):
        username = self.cleaned_data.get("username")
        dbuser = Dreamreal.objects.filter(name = username)

        if not dbuser:
            raise forms.ValidationError("User does not exist in our db!")
        return username
```

Agora, após chamar o método `"is_valid"`, obteremos a saída correta, somente se o usuário estiver em nosso banco de dados. Se você quiser verificar um campo do seu formulário, basta adicionar um método começando com `"clean_"` e depois o nome do seu campo à classe do formulário. Levantar um `forms.ValidationError` é importante.

Django - Upload de arquivos

Geralmente é útil para um aplicativo da web poder fazer upload de arquivos (foto de perfil, músicas, PDF, palavras.....). Vamos discutir como fazer upload de arquivos neste capítulo.

Carregando uma imagem

Antes de começar a brincar com uma imagem, certifique-se de ter a Python Image Library (PIL) instalada. Agora, para ilustrar o upload de uma imagem, vamos criar um formulário de perfil, em nosso myapp/forms.py -

```
#-*- coding: utf-8 -*-  
from django import forms  
  
class ProfileForm(forms.Form):  
    name = forms.CharField(max_length = 100)  
    picture = forms.ImageField()
```

Como você pode ver, a principal diferença aqui é apenas o **forms.ImageField** . ImageField garantirá que o arquivo enviado seja uma imagem. Caso contrário, a validação do formulário falhará.

Agora vamos criar um modelo de "Perfil" para salvar nosso perfil carregado. Isso é feito em myapp/models.py -

```
from django.db import models  
  
class Profile(models.Model):  
    name = models.CharField(max_length = 50)  
    picture = models.ImageField(upload_to = 'pictures')  
  
    class Meta:  
        db_table = "profile"
```

Como você pode ver no modelo, o ImageField recebe um argumento obrigatório: **upload_to** . Isto representa o local no disco rígido onde suas imagens serão salvas. Observe que o parâmetro será adicionado à opção MEDIA_ROOT definida em seu arquivo settings.py.

Agora que temos o Formulário e o Modelo, vamos criar a visualização, em myapp/views.py -

```
#-*- coding: utf-8 -*-  
from myapp.forms import ProfileForm  
from myapp.models import Profile
```



```
def SaveProfile(request):
    saved = False

    if request.method == "POST":
        #Get the posted form
        MyProfileForm = ProfileForm(request.POST, request.FILES)

        if MyProfileForm.is_valid():
            profile = Profile()
            profile.name = MyProfileForm.cleaned_data["name"]
            profile.picture = MyProfileForm.cleaned_data["picture"]
            profile.save()
            saved = True
        else:
            MyProfileForm = Profileform()

    return render(request, 'saved.html', locals())
```

A parte a não perder é que há uma mudança na hora de criar um ProfileForm, adicionamos um segundo parâmetro: **request.FILES** . Se não for aprovado, a validação do formulário falhará, exibindo uma mensagem informando que a imagem está vazia.

Agora, só precisamos do template **save.html** e do template **profile.html** , para o formulário e a página de redirecionamento -

meuapp/templates/saved.html -

```
<html>
  <body>

    {% if saved %}
      <strong>Your profile was saved.</strong>
    {% endif %}

    {% if not saved %}
      <strong>Your profile was not saved.</strong>
    {% endif %}

  </body>
</html>
```

meuapp/templates/profile.html -

```
<html>
  <body>
```



```

<form name = "form" enctype = "multipart/form-data"
    action = "{% url 'myapp.views.SaveProfile' %}" method = "POST" >{% csrf_token %}

    <div style = "max-width:470px;">
        <center>
            <input type = "text" style = "margin-left:20%;"
                placeholder = "Name" name = "name" />
        </center>
    </div>

    <br>

    <div style = "max-width:470px;">
        <center>
            <input type = "file" style = "margin-left:20%;"
                placeholder = "Picture" name = "picture" />
        </center>
    </div>

    <br>

    <div style = "max-width:470px;">
        <center>

            <button style = "border:0px;background-color:#4285F4; margin-top:8%;
                height:35px; width:80%; margin-left:19%;" type = "submit" value = "
                <strong>Login</strong>
            </button>

        </center>
    </div>

</form>

</body>
</html>

```

A seguir, precisamos de nosso par de URLs para começar: myapp/urls.py

```

from django.conf.urls import patterns, url
from django.views.generic import TemplateView

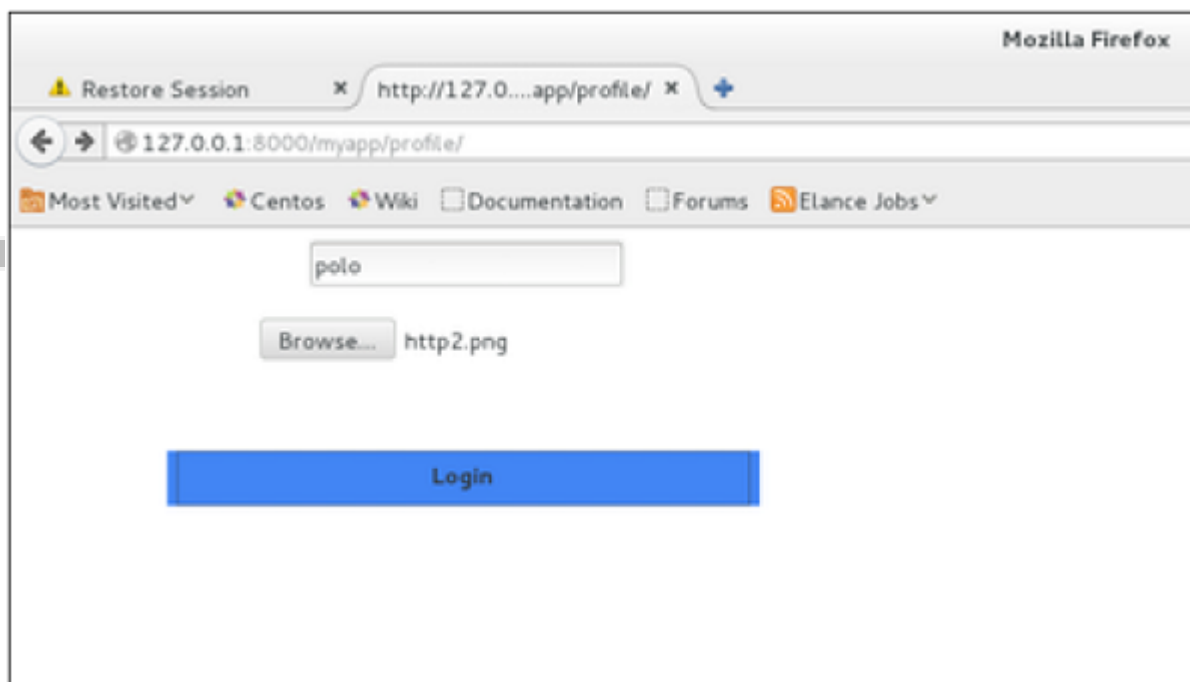
urlpatterns = patterns(
    'myapp.views', url(r'^profile/', TemplateView.as_view(

```

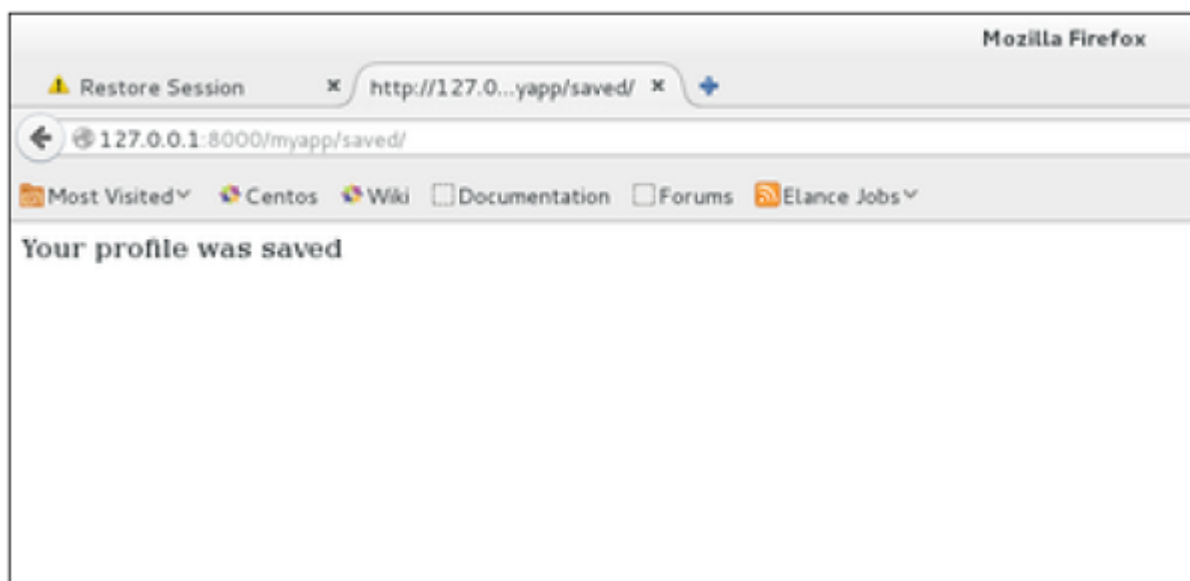


```
template_name = 'profile.html')), url(r'^saved/', 'SaveProfile', name = 'saved'
)
```

Ao acessar `"/myapp/profile"`, obteremos o seguinte modelo `profile.html` renderizado -



E na postagem do formulário, o modelo salvo será renderizado -



Temos um exemplo para imagem, mas se você quiser fazer upload de outro tipo de arquivo, não apenas imagem, basta substituir o **ImageField** tanto no Model quanto no Form por **FileField** .

Django - Configuração do Apache

Até agora, em nossos exemplos, usamos o servidor web de desenvolvimento Django. Mas este servidor é apenas para teste e não é adequado para ambiente de produção. Uma vez em produção, você precisa de um servidor real como Apache, Nginx, etc. Vamos discutir o Apache neste capítulo.

Servir aplicações Django via Apache é feito usando `mod_wsgi`. Portanto, a primeira coisa é garantir que você tenha o Apache e o `mod_wsgi` instalados. Lembre-se, quando criamos nosso projeto e olhamos para a estrutura do projeto, parecia:

```
myproject/  
manage.py  
myproject/  
__init__.py  
settings.py  
urls.py  
wsgi.py
```

O arquivo `wsgi.py` é quem cuida da ligação entre Django e Apache.

Digamos que queremos compartilhar nosso projeto (`myproject`) com o Apache. Só precisamos configurar o Apache para acessar nossa pasta. Suponha que colocamos nossa pasta `myproject` no padrão `/var/www/html`. Nesta fase o acesso ao projeto será feito via `127.0.0.1/myproject`. Isso fará com que o Apache apenas liste a pasta conforme mostrado no instantâneo a seguir.

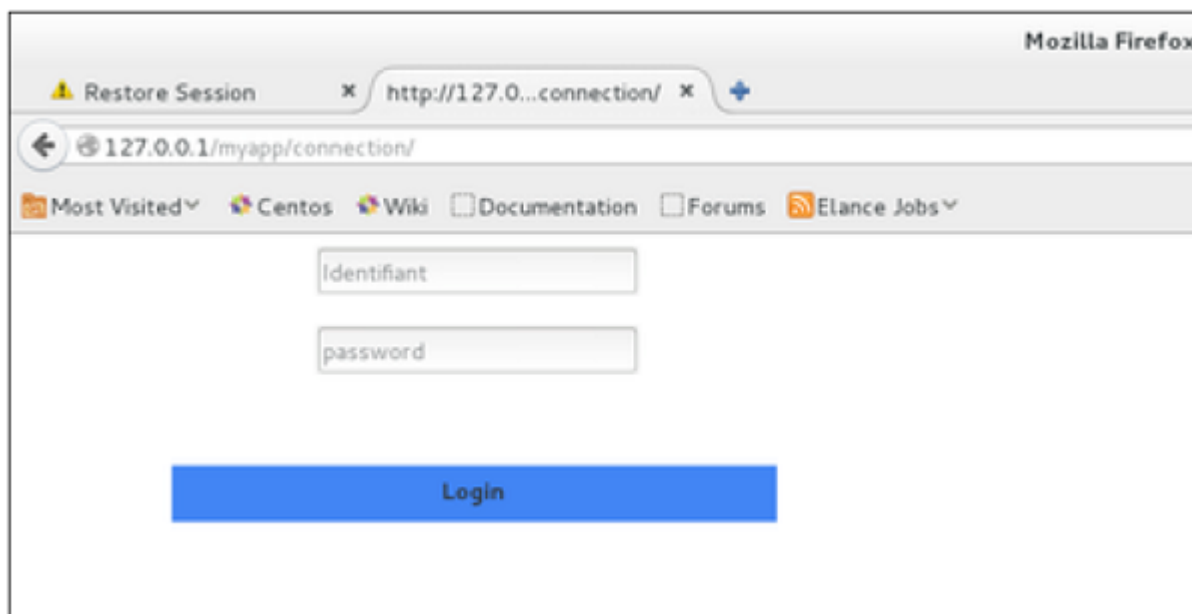


Como visto, o Apache não está lidando com coisas do Django. Para que isso seja resolvido, precisamos configurar o Apache em `httpd.conf`. Então abra o `httpd.conf` e adicione a seguinte linha -

```
WSGIScriptAlias / /var/www/html/myproject/myproject/wsgi.py  
WSGIProxyPath /var/www/html/myproject/
```

```
<Directory /var/www/html/myproject/>  
    <Files wsgi.py>  
        Order deny,allow  
        Allow from all  
    </Files>  
</Directory>
```


Se você puder acessar a página de login como 127.0.0.1/myapp/connection, verá a seguinte página -



Django - Manipulação de Cookies

Às vezes, você pode querer armazenar alguns dados por visitante do site, de acordo com os requisitos do seu aplicativo da web. Tenha sempre em mente que os cookies são salvos no lado do cliente e, dependendo do nível de segurança do navegador do cliente, a configuração de cookies às vezes pode funcionar e às vezes não.

Para ilustrar o tratamento de cookies no Django, vamos criar um sistema usando o sistema de login que criamos antes. O sistema manterá você conectado por X minutos e, além desse tempo, você estará fora do aplicativo.

Para isso, você precisará configurar dois cookies, `last_connection` e `username`.

Primeiramente, vamos mudar nossa visualização de login para armazenar nosso nome de usuário e cookies `last_connection` -

```
from django.template import RequestContext

def login(request):
    username = "not logged in"

    if request.method == "POST":
        #Get the posted form
        MyLoginForm = LoginForm(request.POST)

        if MyLoginForm.is_valid():
            username = MyLoginForm.cleaned_data['username']
        else:
            MyLoginForm = LoginForm()
```



```

response = render_to_response(request, 'loggedin.html', {"username": username},
    context_instance = RequestContext(request))

response.set_cookie('last_connection', datetime.datetime.now())
response.set_cookie('username', datetime.datetime.now())

return response

```

Conforme visto na visualização acima, a configuração do cookie é feita pelo método **set_cookie** chamado na resposta e não na solicitação, e observe também que todos os valores dos cookies são retornados como string.

Vamos agora criar um formView para o formulário de login, onde não exibiremos o formulário se o cookie estiver definido e não tiver mais de 10 segundos -

```

def formView(request):
    if 'username' in request.COOKIES and 'last_connection' in request.COOKIES:
        username = request.COOKIES['username']

        last_connection = request.COOKIES['last_connection']
        last_connection_time = datetime.datetime.strptime(last_connection[:-7],
            "%Y-%m-%d %H:%M:%S")

        if (datetime.datetime.now() - last_connection_time).seconds < 10:
            return render(request, 'loggedin.html', {"username": username})
        else:
            return render(request, 'login.html', {})

    else:
        return render(request, 'login.html', {})

```

Como você pode ver no formView acima, o acesso ao cookie que você definiu é feito através do atributo COOKIES (dict) da solicitação.

Agora vamos alterar o arquivo url.py para alterar o URL para que ele combine com nossa nova visualização -

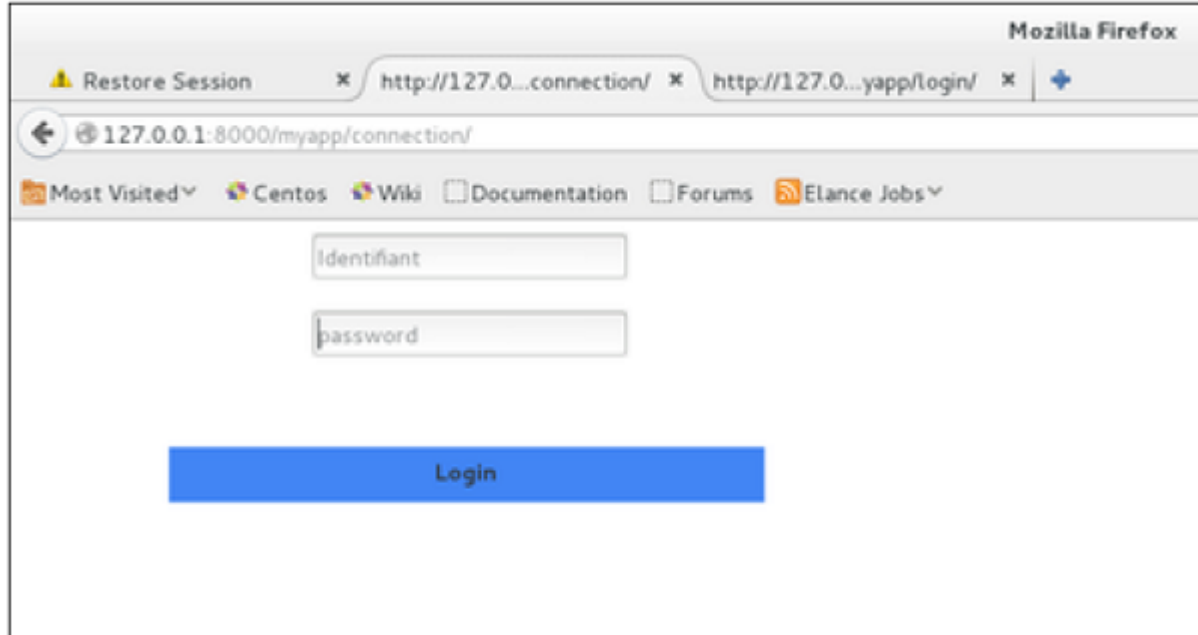
```

from django.conf.urls import patterns, url
from django.views.generic import TemplateView

urlpatterns = patterns('myapp.views',
    url(r'^connection/', 'formView', name = 'loginform'),
    url(r'^login/', 'login', name = 'login'))

```

Ao acessar /myapp/connection, você obterá a seguinte página -



E você será redirecionado para a seguinte tela ao enviar -



Agora, se você tentar acessar /myapp/connection novamente no intervalo de 10 segundos, você será redirecionado diretamente para a segunda tela. E se você acessar /myapp/connection novamente fora deste intervalo você receberá o formulário de login (tela 1).

Django - Sessões

Conforme discutido anteriormente, podemos usar cookies do lado do cliente para armazenar muitos dados úteis para o aplicativo web. Já vimos que podemos usar cookies do lado do cliente para armazenar vários dados úteis para nosso aplicativo web. Isso leva a muitas falhas de segurança, dependendo da importância dos dados que você deseja salvar.

Por razões de segurança, o Django possui um framework de sessão para manipulação de cookies. As sessões são utilizadas para abstrair o recebimento e envio de cookies, os dados são salvos no lado do servidor (como no banco de dados), e o cookie do lado do cliente possui apenas um ID de sessão para identificação. As sessões também são úteis para evitar casos em que o navegador do usuário esteja configurado para "não aceitar" cookies.

Configurando Sessões

No Django, a habilitação da sessão é feita no seu projeto **settings.py** , adicionando algumas linhas às opções **MIDDLEWARE_CLASSES** e **INSTALLED_APPS** . Isso deve ser feito durante a criação do projeto, mas é sempre bom saber, então **MIDDLEWARE_CLASSES** deveria ter -

```
'django.contrib.sessions.middleware.SessionMiddleware'
```

E **INSTALLED_APPS** deveria ter -

```
'django.contrib.sessions'
```

Por padrão, o Django salva as informações da sessão no banco de dados (tabela ou coleção `django_session`), mas você pode configurar o mecanismo para armazenar informações de outras formas como: em **arquivo** ou em **cache** .

Quando a sessão está habilitada, cada solicitação (primeiro argumento de qualquer visualização no Django) possui um atributo de sessão (dict).

Vamos criar um exemplo simples para ver como criar e salvar sessões. Nós construímos um sistema de login simples antes (veja o capítulo Processamento de formulários do Django e o capítulo Manipulação de cookies do Django). Vamos salvar o nome de usuário em um cookie para que, se não estiver desconectado, ao acessar nossa página de login você não verá o formulário de login. Basicamente, vamos tornar nosso sistema de login que usamos no Django Cookies mais seguro, salvando os cookies no servidor.

Para isso, primeiro vamos alterar nossa visualização de login para salvar nosso cookie de nome de usuário no servidor -

```
def login(request):
    username = 'not logged in'

    if request.method == 'POST':
        MyLoginForm = LoginForm(request.POST)

        if MyLoginForm.is_valid():
            username = MyLoginForm.cleaned_data['username']
            request.session['username'] = username
        else:
            MyLoginForm = LoginForm()

    return render(request, 'loggedin.html', {"username" : username})
```

Então, vamos criar uma visualização formView para o formulário de login, onde não exibiremos o formulário se o cookie estiver definido -

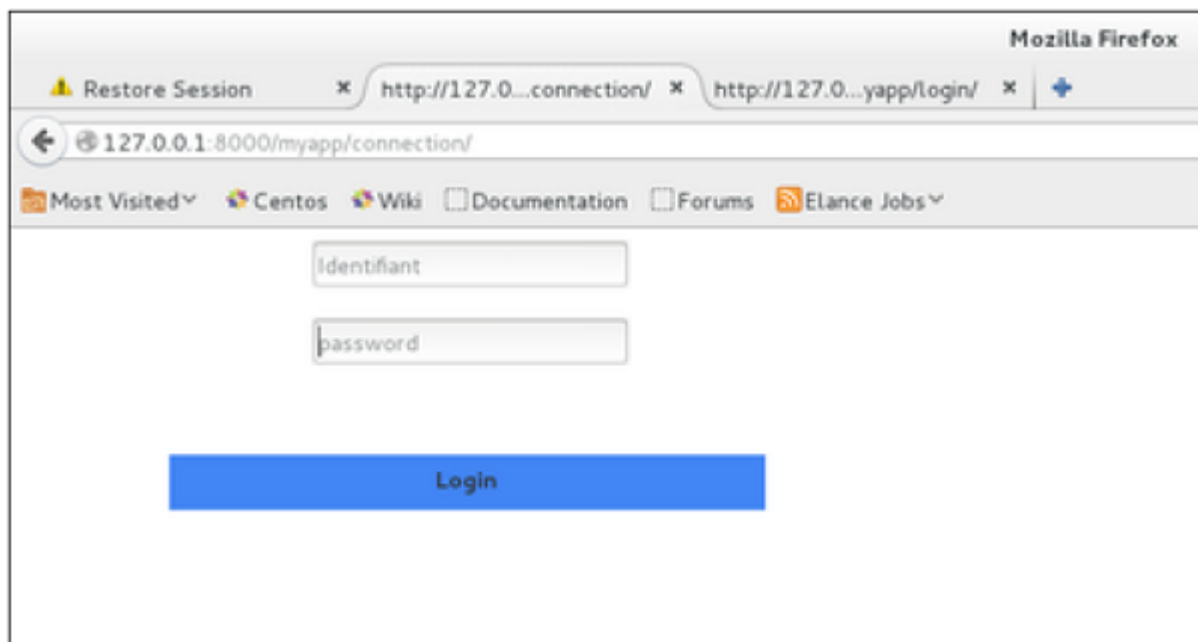
```
def formView(request):
    if request.session.has_key('username'):
        username = request.session['username']
        return render(request, 'loggedin.html', {"username" : username})
    else:
        return render(request, 'login.html', {})
```

Agora vamos alterar o arquivo url.py para alterar o URL para que ele combine com nossa nova visualização -

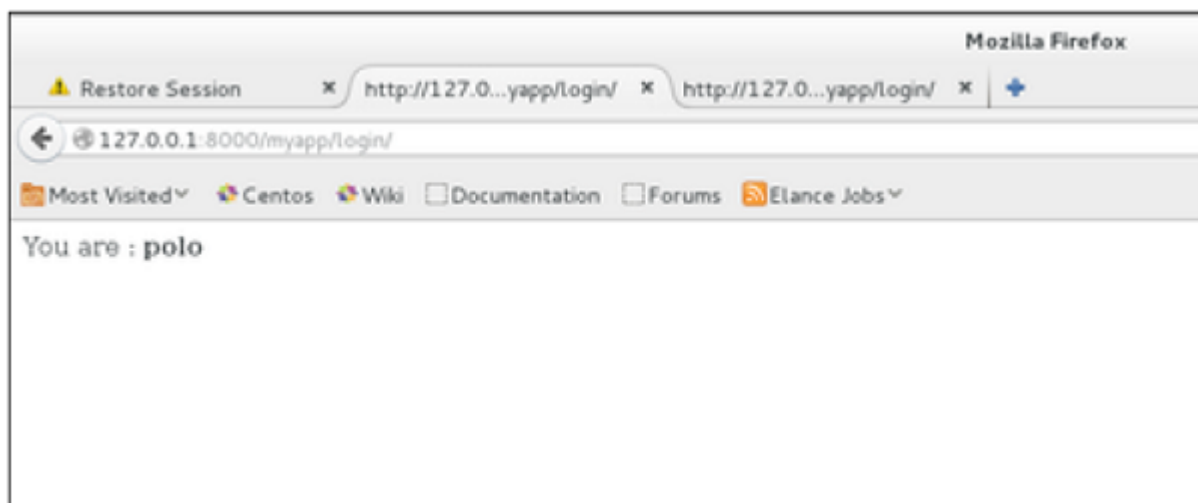
```
from django.conf.urls import patterns, url
from django.views.generic import TemplateView

urlpatterns = patterns('myapp.views',
    url(r'^connection/', 'formView', name = 'loginform'),
    url(r'^login/', 'login', name = 'login'))
```

Ao acessar /myapp/connection, você verá a seguinte página -



E você será redirecionado para a seguinte página -



Agora, se você tentar acessar /myapp/connection novamente, será redirecionado diretamente para a segunda tela.

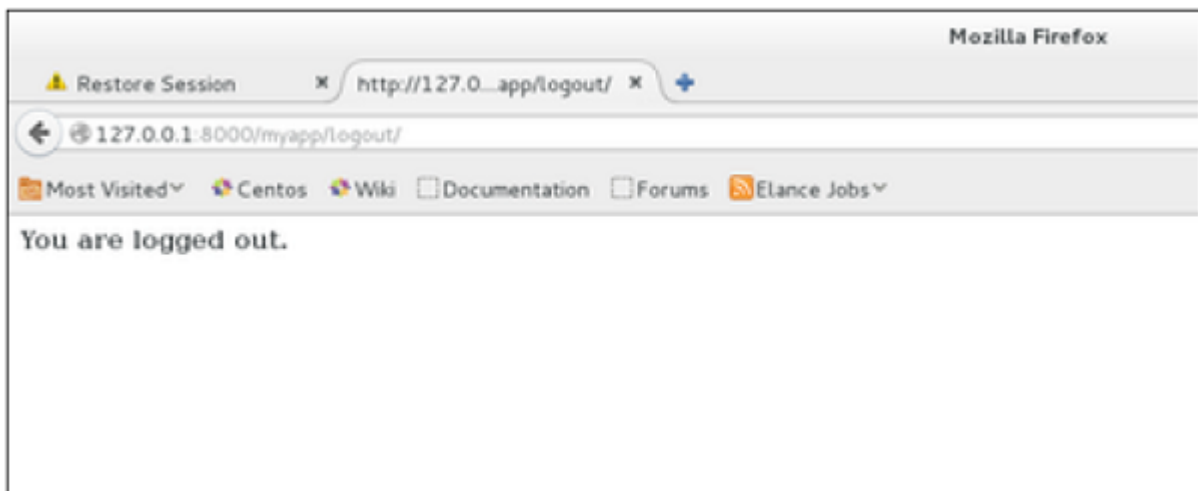
Vamos criar uma visualização de logout simples que apaga nosso cookie.

```
def logout(request):  
    try:  
        del request.session['username']  
    except:  
        pass  
    return HttpResponseRedirect("<strong>You are logged out.</strong>")
```

E emparelhe-o com um URL de logout em myapp/url.py

```
url(r'^logout/', 'logout', name = 'logout'),
```

Agora, se você acessar /myapp/logout, você obterá a seguinte página -



Se você acessar /myapp/connection novamente, receberá o formulário de login (tela 1).

Mais algumas ações possíveis usando sessões

Vimos como armazenar e acessar uma sessão, mas é bom saber que o atributo de sessão da solicitação possui algumas outras ações úteis como -

- **set_expiry (value)** - Define o tempo de expiração da sessão.
- **get_expiry_age()** - Retorna o número de segundos até que esta sessão expire.
- **get_expiry_date()** - Retorna a data em que esta sessão irá expirar.
- **clear_expired()** - Remove sessões expiradas do armazenamento de sessões.
- **get_expire_at_browser_close()** - Retorna True ou False, dependendo se os cookies de sessão do usuário expiraram quando o navegador do usuário foi fechado.

Django - Cache

Armazenar algo em cache é salvar o resultado de um cálculo caro, para que você não o execute na próxima vez que precisar. A seguir está um pseudocódigo que explica como funciona o cache -

```
given a URL, try finding that page in the cache

if the page is in the cache:
    return the cached page
else:
    generate the page
    save the generated page in the cache (for next time)
    return the generated page
```

Django vem com seu próprio sistema de cache que permite salvar suas páginas dinâmicas, para evitar calculá-las novamente quando necessário. O ponto positivo da estrutura Django Cache é que você pode armazenar em cache -

- A saída de uma visualização específica.
- Uma parte de um modelo.
- Todo o seu site.

Para usar o cache no Django, a primeira coisa a fazer é configurar onde o cache ficará. A estrutura de cache oferece diferentes possibilidades - o cache pode ser salvo no banco de dados, no sistema de arquivos ou diretamente na memória. A configuração é feita no arquivo **settings.py** do seu projeto.

Configurando cache no banco de dados

Basta adicionar o seguinte no arquivo settings.py do projeto -

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.db.DatabaseCache',
        'LOCATION': 'my_table_name',
    }
}
```

Para que isso funcione e complete a configuração, precisamos criar a tabela de cache 'my_table_name'. Para isso, você precisa fazer o seguinte -

```
python manage.py createcachetable
```

Configurando cache no sistema de arquivos

Basta adicionar o seguinte no arquivo settings.py do projeto -

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',
        'LOCATION': '/var/tmp/django_cache',
    }
}
```

Configurando cache na memória

Esta é a forma mais eficiente de armazenamento em cache, para usá-lo você pode usar uma das seguintes opções dependendo da biblioteca de ligação Python que você escolher para o cache de memória -

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': '127.0.0.1:11211',
    }
}
```

Ou

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': 'unix:/tmp/memcached.sock',
    }
}
```

Armazenando em cache todo o site

A maneira mais simples de usar cache no Django é armazenar em cache o site inteiro. Isso é feito editando a opção MIDDLEWARE_CLASSES no projeto settings.py. O seguinte precisa ser adicionado à opção -


```
MIDDLEWARE_CLASSES += (  
    'django.middleware.cache.UpdateCacheMiddleware',  
    'django.middleware.common.CommonMiddleware',  
    'django.middleware.cache.FetchFromCacheMiddleware',  
)
```

Observe que a ordem é importante aqui: a atualização deve vir antes do middleware Fetch.

Então, no mesmo arquivo, você precisa definir -

CACHE_MIDDLEWARE_ALIAS - The cache alias to use for storage.

CACHE_MIDDLEWARE_SECONDS - The number of seconds each page should be cached.

Armazenando uma visualização em cache

Se não quiser armazenar em cache o site inteiro, você pode armazenar em cache uma visualização específica. Isso é feito usando o decorador **cache_page** que vem com o Django. Digamos que queremos armazenar em cache o resultado da visualização **viewArticles** -

```
from django.views.decorators.cache import cache_page  
  
@cache_page(60 * 15)  
  
def viewArticles(request, year, month):  
    text = "Displaying articles of : %s/%s"%(year, month)  
    return HttpResponse(text)
```

Como você pode ver, **cache_page** leva o número de segundos que você deseja que o resultado da visualização seja armazenado em cache como parâmetro. No nosso exemplo acima, o resultado será armazenado em cache por 15 minutos.

Nota - Como vimos antes, a visualização acima foi mapeada para -

```
urlpatterns = patterns('myapp.views',  
    url(r'^articles/(?P<month>\d{2})/(?P<year>\d{4})/', 'viewArticles', name = 'articl
```

Como a URL recebe parâmetros, cada chamada diferente será armazenada em cache separadamente. Por exemplo, a solicitação para /myapp/articles/02/2007 será armazenada em cache separadamente para /myapp/articles/03/2008.

O armazenamento em cache de uma visualização também pode ser feito diretamente no arquivo url.py. Então o seguinte tem o mesmo resultado acima. Basta editar seu arquivo myapp/url.py e alterar o URL mapeado relacionado (acima) para -

```
urlpatterns = patterns('myapp.views',
    url(r'^articles/(?P<month>\d{2})/(?P<year>\d{4})/',
        cache_page(60 * 15)('viewArticles'), name = 'articles'),)
```

E, claro, não é mais necessário em myapp/views.py.

Armazenando em Cache um Fragmento de Modelo

Você também pode armazenar em cache partes de um modelo, isso é feito usando a tag **cache** . Vamos pegar nosso modelo **hello.html** -

```
{% extends "main_template.html" %}
{% block title %}My Hello Page{% endblock %}
{% block content %}

Hello World!!!<p>Today is {{today}}</p>
We are
{% if today.day == 1 %}

the first day of month.
{% elif today.day == 30 %}

the last day of month.
{% else %}

I don't know.
{%endif%}

<p>
    {% for day in days_of_week %}
        {{day}}
    </p>

{% endfor %}
{% endblock %}
```

E para armazenar em cache o bloco de conteúdo, nosso modelo se tornará -

```
{% load cache %}
{% extends "main_template.html" %}
{% block title %}My Hello Page{% endblock %}
{% cache 500 content %}
{% block content %}
```

```
Hello World!!!<p>Today is {{today}}</p>
```

```
We are
```

```
{% if today.day == 1 %}
```

```
the first day of month.
```

```
{% elif today.day == 30 %}
```

```
the last day of month.
```

```
{% else %}
```

```
I don't know.
```

```
{%endif%}
```

```
<p>
```

```
{% for day in days_of_week %}
```

```
{{day}}
```

```
</p>
```

```
{% endfor %}
```

```
{% endblock %}
```

```
{% endcache %}
```

Como você pode ver acima, a tag de cache terá 2 parâmetros - o tempo que você deseja que o bloco seja armazenado em cache (em segundos) e o nome a ser dado ao fragmento de cache.

Django - Comentários

Antes de começar, observe que o framework Django Comments está obsoleto, desde a versão 1.5. Agora você pode usar um recurso externo para fazer isso, mas se ainda quiser usá-lo, ele ainda está incluído nas versões 1.6 e 1.7. A partir da versão 1.8 ele está ausente, mas você ainda pode obter o código em uma conta GitHub diferente.

A estrutura de comentários facilita anexar comentários a qualquer modelo em seu aplicativo.

Para começar a usar a estrutura de comentários do Django -

Edite o arquivo settings.py do projeto e adicione **'django.contrib.sites'** e **'django.contrib.comments'** à opção INSTALLED_APPS -

```
INSTALLED_APPS += ('django.contrib.sites', 'django.contrib.comments',)
```

Obtenha o ID do site -

```
>>> from django.contrib.sites.models import Site
>>> Site().save()
```



```
>>> Site.objects.all()[0].id
u'56194498e13823167dd43c64'
```

Defina o ID obtido no arquivo settings.py -

```
SITE_ID = u'56194498e13823167dd43c64'
```

Sincronize o banco de dados, para criar toda a tabela ou coleção de comentários -

```
python manage.py syncdb
```

Adicione os URLs do aplicativo de comentários ao urls.py do seu projeto -

```
from django.conf.urls import include
url(r'^comments/', include('django.contrib.comments.urls')),
```

Agora que temos a estrutura instalada, vamos alterar nossos modelos de hello para comentários de rastreamento em nosso modelo Dreamreal. Iremos listar e salvar comentários para uma entrada específica do Dreamreal cujo nome será passado como parâmetro para a URL /myapp/hello.

Modelo Dreamreal

```
class Dreamreal(models.Model):

    website = models.CharField(max_length = 50)
    mail = models.CharField(max_length = 50)
    name = models.CharField(max_length = 50)
    phonenumber = models.IntegerField()

    class Meta:
        db_table = "dreamreal"
```

olá vista

```
def hello(request, Name):
    today = datetime.datetime.now().date()
    daysOfWeek = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
    dreamreal = Dreamreal.objects.get(name = Name)
    return render(request, 'hello.html', locals())
```



modelo olá.html

```
{% extends "main_template.html" %}
{% load comments %}
{% block title %}My Hello Page{% endblock %}
{% block content %}

<p>
    Our Dreamreal Entry:
    <p><strong>Name :</strong> {{dreamreal.name}}</p>
    <p><strong>Website :</strong> {{dreamreal.website}}</p>
    <p><strong>Phone :</strong> {{dreamreal.phonenumber}}</p>
    <p><strong>Number of comments :<strong>
    {% get_comment_count for dreamreal as comment_count %} {{ comment_count }}</p>
    <p><strong>List of comments :</p>
    {% render_comment_list for dreamreal %}
</p>

{% render_comment_form for dreamreal %}
{% endblock %}
```

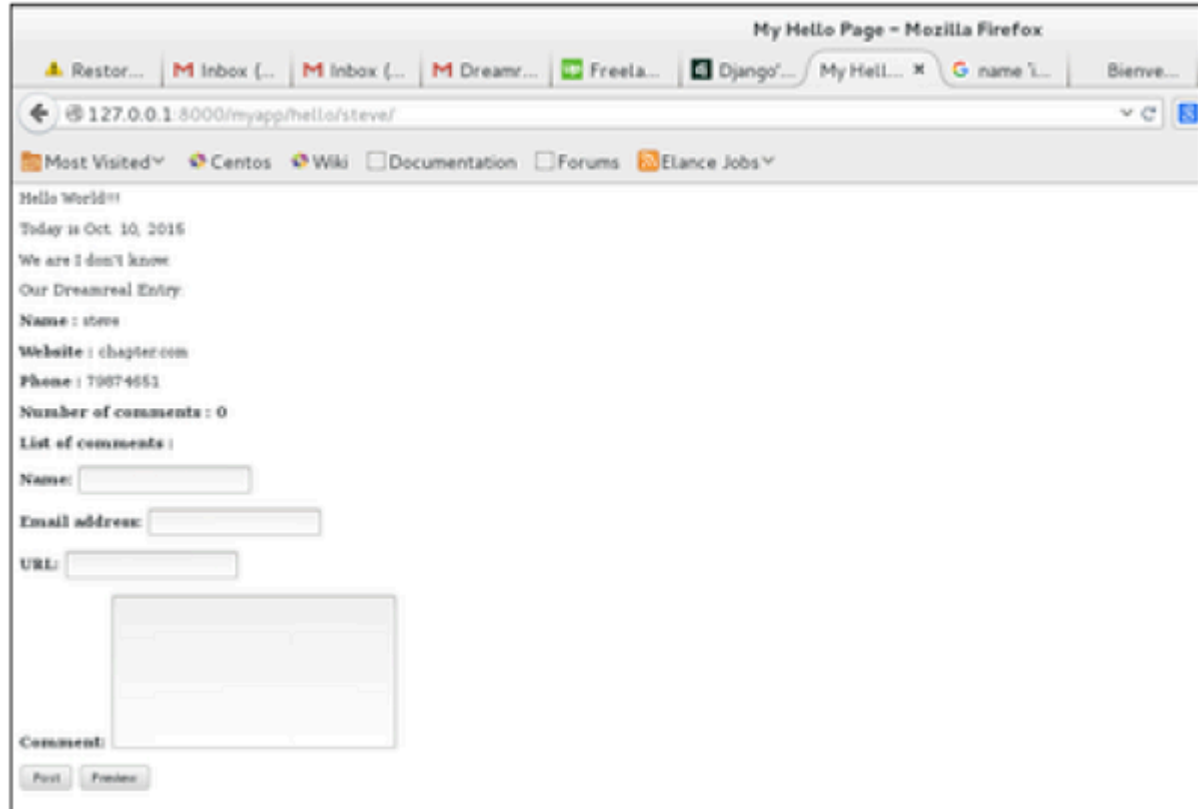
Finalmente, o URL de mapeamento para nossa visualização hello -

```
url(r'^hello/(?P<Name>\w+)/', 'hello', name = 'hello'),
```

Agora,

- Em nosso modelo (hello.html), carregue a estrutura de comentários com - {% load comments %}
- Obtemos o número de comentários para o objeto Dreamreal passado pela visualização - {% get_comment_count for dreamreal as comment_count %}
- Obtemos a lista de comentários para os objetos - {% render_comment_list for dreamreal %}
- Exibimos o formulário de comentários padrão - {% render_comment_form for dreamreal %}

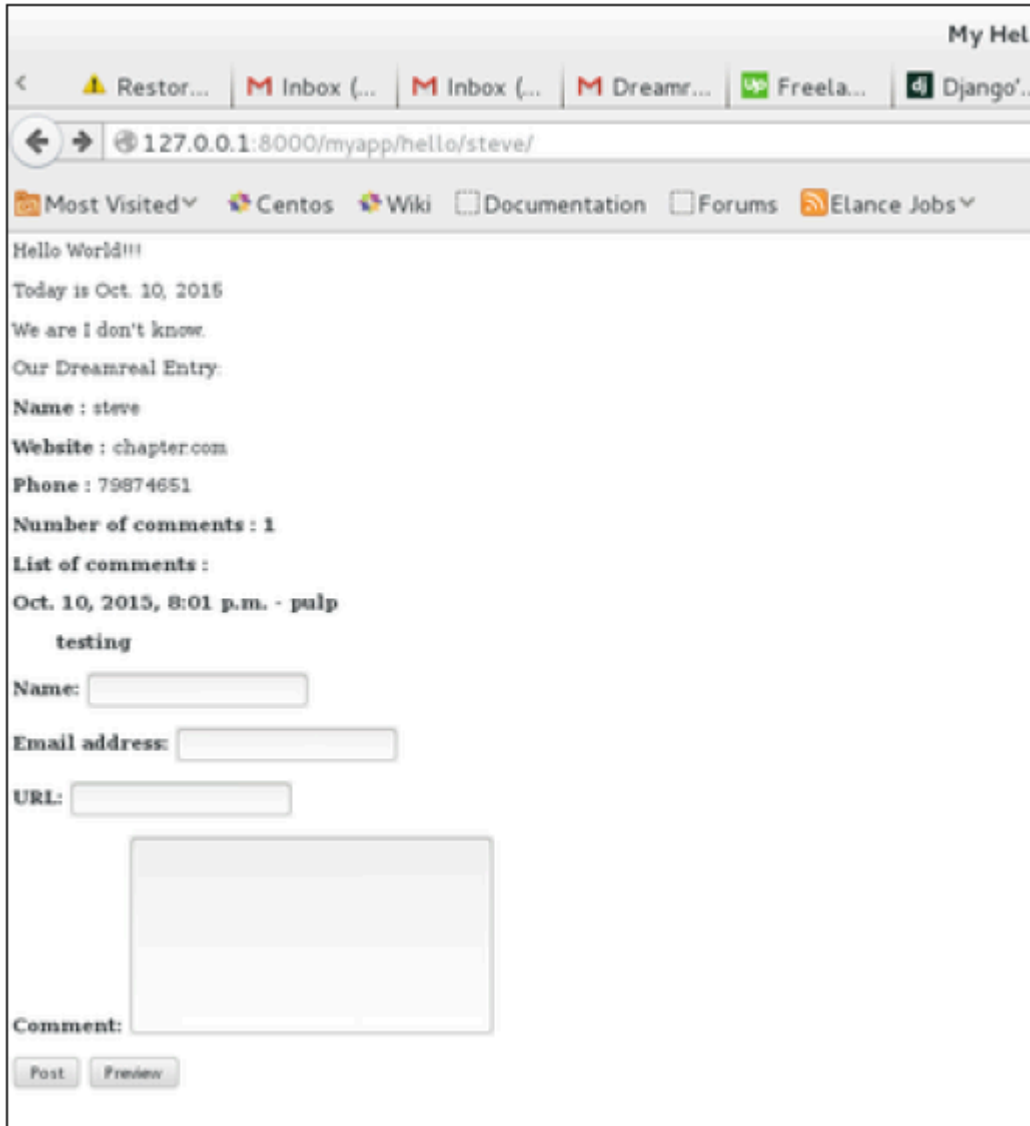
Ao acessar /myapp/hello/steve você obterá as informações dos comentários da entrada do Dreamreal cujo nome é Steve. Acessar esse URL levará você -



Ao postar um comentário, você será redirecionado para a seguinte página -



Se você acessar /myapp/hello/steve novamente, verá a seguinte página -



Como você pode ver, o número de comentários é 1 agora e você tem o comentário abaixo da linha da lista de comentários.

Django - RSS

Django vem com uma estrutura de geração de feed de distribuição. Com ele você pode criar feeds RSS ou Atom apenas subclassificando **`django.contrib.syndication.views.Feed class`**.

Vamos criar um feed para os últimos comentários feitos no aplicativo (veja também o capítulo Django - Comments Framework). Para isso, vamos criar um `myapp/feeds.py` e definir nosso feed (você pode colocar suas classes de feeds em qualquer lugar que desejar em sua estrutura de código).

```
from django.contrib.syndication.views import Feed
from django.contrib.comments import Comment
from django.core.urlresolvers import reverse

class DreamrealCommentsFeed(Feed):
    title = "Dreamreal's comments"
    link = "/drcomments/"
    description = "Updates on new comments on Dreamreal entry."
```

```
def items(self):
    return Comment.objects.all().order_by("-submit_date")[:5]

def item_title(self, item):
    return item.user_name

def item_description(self, item):
    return item.comment

def item_link(self, item):
    return reverse('comment', kwargs = {'object_pk':item.pk})
```

- Em nossa classe de feed, os atributos **title** , **link** e **description** correspondem aos elementos RSS padrão **<title>** , **<link>** e **<description>** .
- O método **items** retorna os elementos que devem ir no feed como elemento item. No nosso caso, os últimos cinco comentários.
- O método **item_title** obterá o que será usado como título para nosso item de feed. No nosso caso, o título será o nome do usuário.
- O método **item_description** obterá o que será usado como descrição para nosso item de feed. No nosso caso, o próprio comentário.
- O método **item_link** criará o link para o item completo. No nosso caso, você chegará ao comentário.

Agora que temos nosso feed, vamos adicionar uma visualização de comentários em views.py para exibir nosso comentário -

```
from django.contrib.comments import Comment

def comment(request, object_pk):
    mycomment = Comment.objects.get(object_pk = object_pk)
    text = '<strong>User :</strong> %s <p>%mycomment.user_name</p>'
    text += '<strong>Comment :</strong> %s <p>%mycomment.comment</p>'
    return HttpResponse(text)
```

Também precisamos de alguns URLs em nosso myapp urls.py para mapeamento -

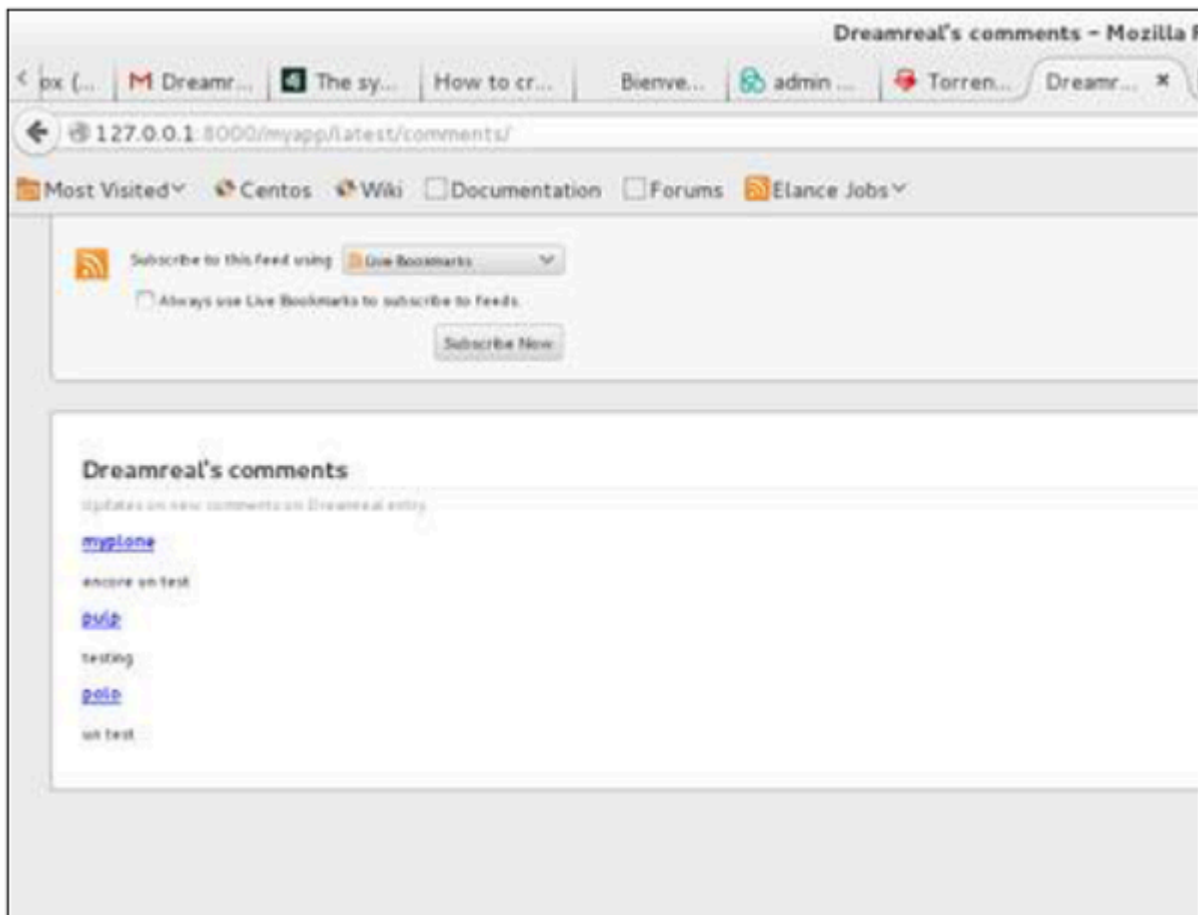
```
from myapp.feeds import DreamrealCommentsFeed
from django.conf.urls import patterns, url

urlpatterns += patterns('',
    url(r'^latest/comments/', DreamrealCommentsFeed()),
```

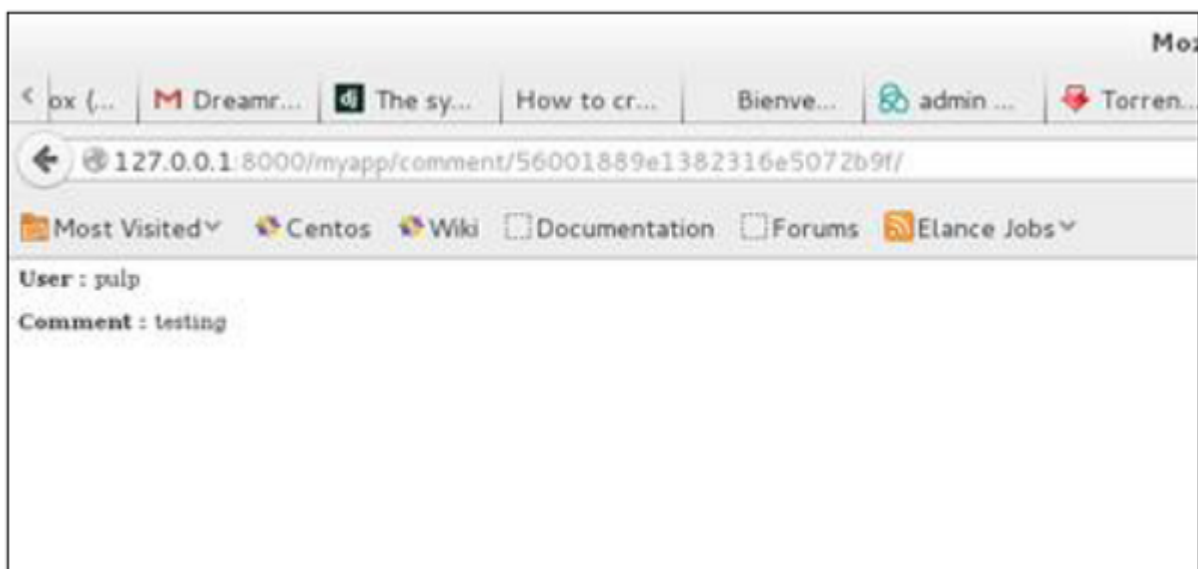


```
url(r'^comment/(?P<w+>)/', 'comment', name = 'comment'),
)
```

Ao acessar /myapp/latest/comments/ você receberá nosso feed -



Em seguida, clicar em um dos nomes de usuário levará você a:
/myapp/comment/comment_id conforme definido em nossa visualização de comentários anterior e você obterá -



Assim, definir um feed RSS é apenas uma questão de subclassificar a classe Feed e garantir que as URLs (uma para acessar o feed e outra para acessar os elementos do feed) estejam definidas. Apenas como comentário, isso pode ser anexado a qualquer modelo em seu aplicativo.

Django-Ajax

Ajax é essencialmente uma combinação de tecnologias integradas para reduzir o número de carregamentos de páginas. Geralmente usamos Ajax para facilitar a experiência do usuário final. O uso do Ajax no Django pode ser feito diretamente usando uma biblioteca Ajax como JQuery ou outras. Digamos que você queira usar JQuery, então você precisa baixar e servir a biblioteca em seu servidor através do Apache ou outros. Em seguida, use-o em seu modelo, assim como faria ao desenvolver qualquer aplicativo baseado em Ajax.

Outra maneira de usar Ajax no Django é usar o framework Django Ajax. O mais comumente usado é Django-dajax, que é uma ferramenta poderosa para desenvolver lógica de apresentação assíncrona em aplicações web de maneira fácil e super rápida, usando Python e quase nenhum código-fonte JavaScript. Suporta quatro dos frameworks Ajax mais populares: Prototype, jQuery, Dojo e MooTools.

Usando Django-dajax

A primeira coisa a fazer é instalar o Django-dajax. Isso pode ser feito usando easy_install ou pip -

```
$ pip install django_dajax
$ easy_install django_dajax
```

Isto instalará automaticamente o django-dajaxice, exigido pelo django-dajax. Precisamos então configurar o dajax e o dajaxice.

Adicione dajax e dajaxice em seu projeto settings.py na opção INSTALLED_APPS -

```
INSTALLED_APPS += (
    'dajaxice',
    'dajax'
)
```

Certifique-se de que no mesmo arquivo settings.py você tenha o seguinte -

```
TEMPLATE_LOADERS = (
    'django.template.loaders.filesystem.Loader',
    'django.template.loaders.app_directories.Loader',
    'django.template.loaders.eggs.Loader',
)
```

```
TEMPLATE_CONTEXT_PROCESSORS = (
    'django.contrib.auth.context_processors.auth',
    'django.core.context_processors.debug',
    'django.core.context_processors.i18n',
```



```

'django.core.context_processors.media',
'django.core.context_processors.static',
'django.core.context_processors.request',
'django.contrib.messages.context_processors.messages'
)

STATICFILES_FINDERS = (
    'django.contrib.staticfiles.finders.FileSystemFinder',
    'django.contrib.staticfiles.finders.AppDirectoriesFinder',
    'dajaxice.finders.DajaxiceFinder',
)

DAJAXICE_MEDIA_PREFIX = 'dajaxice'

```

Agora vá para o arquivo `myapp/url.py` e certifique-se de ter o seguinte para definir URLs `dajax` e carregar arquivos `js` estáticos `dajax` -

```

from dajaxice.core import dajaxice_autodiscover, dajaxice_config
from django.contrib.staticfiles.urls import staticfiles_urlpatterns
from django.conf import settings

Then dajax urls:

urlpatterns += patterns('',
    url(r'^%s/' % settings.DAJAXICE_MEDIA_PREFIX, include('dajaxice.urls')),)

urlpatterns += staticfiles_urlpatterns()

```

Vamos criar um formulário simples baseado em nosso modelo `Dreamreal` para armazená-lo, usando `Ajax` (significa sem atualização).

Primeiramente, precisamos de nosso formulário `Dreamreal` em `myapp/form.py`.

```

class DreamrealForm(forms.Form):
    website = forms.CharField(max_length = 100)
    name = forms.CharField(max_length = 100)
    phonenumber = forms.CharField(max_length = 50)
    email = forms.CharField(max_length = 100)

```

Então precisamos de um arquivo `ajax.py` em nosso aplicativo: `myapp/ajax.py`. É aí que está a nossa lógica, é onde colocamos a função que vai salvar nosso formulário e depois retornamos o `popup` -

```

from dajaxice.utils import deserialize_form
from myapp.form import DreamrealForm

```

```

from dajax.core import Dajax
from myapp.models import Dreamreal

@dajaxice_register
def send_form(request, form):
    dajax = Dajax()
    form = DreamrealForm(deserialize_form(form))

    if form.is_valid():
        dajax.remove_css_class('#my_form input', 'error')
        dr = Dreamreal()
        dr.website = form.cleaned_data.get('website')
        dr.name = form.cleaned_data.get('name')
        dr.phonenumber = form.cleaned_data.get('phonenumber')
        dr.save()

        dajax.alert("Dreamreal Entry %s was successfully saved." %
                    form.cleaned_data.get('name'))
    else:
        dajax.remove_css_class('#my_form input', 'error')
        for error in form.errors:
            dajax.add_css_class('#id_%s' % error, 'error')

    return dajax.json()

```

Agora vamos criar o template dreamreal.html, que tem nosso formato -

```

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

    <form action = "" method = "post" id = "my_form" accept-charset = "utf-8">
      {{ form.as_p }}
      <p><input type = "button" value = "Send" onclick = "send_form();"></p>
    </form>

  </body>
</html>

```

Adicione a visualização que acompanha o modelo em myapp/views.py -

```

def dreamreal(request):
    form = DreamrealForm()
    return render(request, 'dreamreal.html', locals())

```

Adicione o URL correspondente em myapp/urls.py -

```
url(r'^dreamreal/', 'dreamreal', name = 'dreamreal'),
```

Agora vamos adicionar o necessário em nosso template para fazer o Ajax funcionar -

No topo do arquivo, adicione -

```
{% load static %}
{% load dajaxice_templatetags %}
```

E na seção <head> do nosso modelo dreamreal.html adicione -

Estamos usando a biblioteca JQuery para este exemplo, então adicione -

```
<script src = "{% static '/static/jquery-1.11.3.min.js' %}"
    type = "text/javascript" charset = "utf-8"></script>
<script src = "{% static '/static/dajax/jquery.dajax.core.js' %}"></script>
```

A função Ajax que será chamada ao clicar -

```
<script>

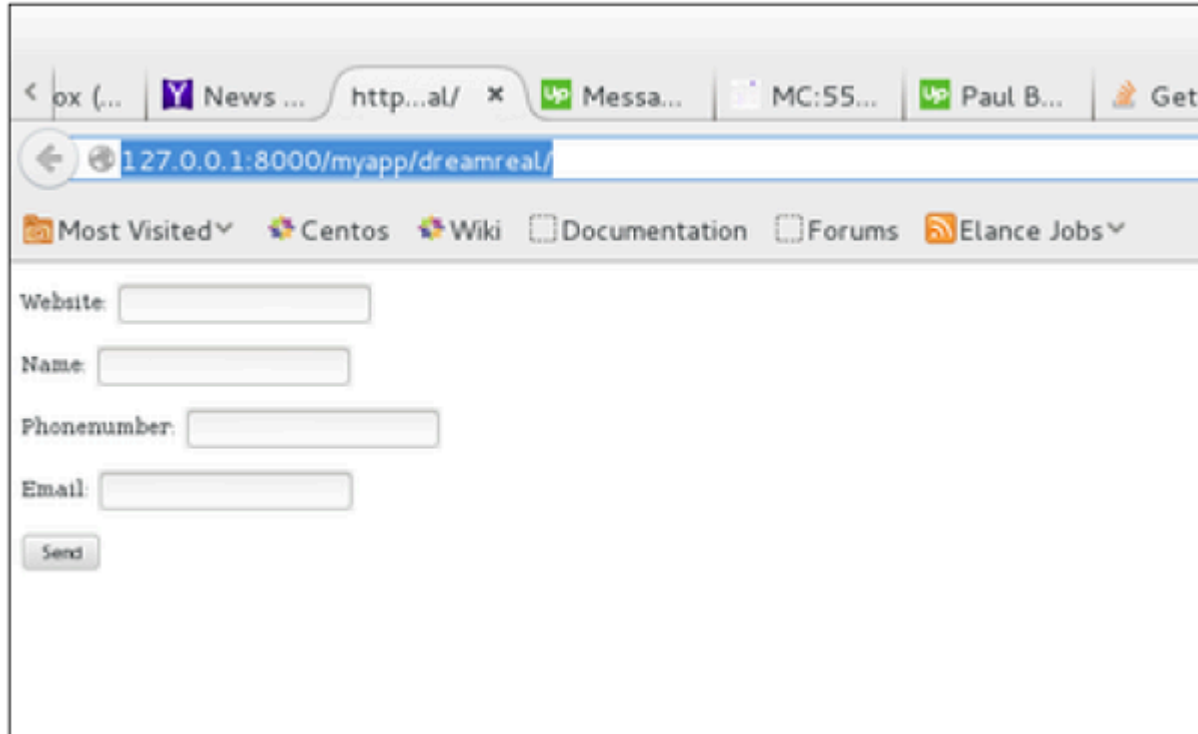
    function send_form(){
        Dajaxice.myapp.send_form(Dajax.process,{'form':$('#my_form').serialize(true)});
    }
</script>
```

Observe que você precisa de "jquery-1.11.3.min.js" em seu diretório de arquivos estáticos e também de jquery.dajax.core.js. Para garantir que todos os arquivos estáticos dajax sejam servidos em seu diretório estático, execute -

```
$python manage.py collectstatic
```

Note - Às vezes, o jquery.dajax.core.js pode estar faltando, se isso acontecer, basta baixar o código-fonte, pegar esse arquivo e colocá-lo em sua pasta estática.

Você verá a seguinte tela ao acessar /myapp/dreamreal/ -



Ao enviar, você receberá a seguinte tela -

