DJANGO FRAMEWORK –

What is Django?

Django is a high-level Python Web framework that encourages rapid development and clean pragmatic design. A Web framework is a set of components that provide a standard way to develop websites fast and easily. Django's primary goal is to ease the creation of complex database-driven websites. Some well known sites that use Django include PBS, Instagram, Disqus, Washington Times, Bitbucket and Mozilla.

The Model-View-Template (MVT) is slightly different from MVC. In fact the main difference between the two patterns is that Django itself takes care of the Controller part (Software Code that controls the interactions between the Model and View), leaving us with the template. The template is a HTML file mixed with Django Template Language (DTL).

A good Web framework addresses these common concerns:

It provides a method of mapping requested URLs to code that handles requests. In other words, it gives you a way of designating which code should execute for which URL. For instance, you could tell the framework, "For URLs that look like /users/joe/, execute code that displays the profile for the user with that username."

It makes it easy to display, validate and redisplay HTML forms. HTML forms are the primary way of getting input data from Web users, so a Web framework had better make it easy to display them and handle the tedious code of form display and redisplay (with errors highlighted).

It converts user-submitted input into data structures that can be manipulated conveniently. For example, the framework could convert HTML form submissions into native data types of the programming language you're using.

It helps separate content from presentation via a template system, so you can change your site's look-and-feel without affecting your content, and vice-versa.

It conveniently integrates with storage layers — such as databases — but doesn't strictly require the use of a database.

It lets you work more productively, at a higher level of abstraction, than if you were coding against, say, HTTP. But it doesn't restrict you from going "down" one level of abstraction when needed.

It gets out of your way, neglecting to leave dirty stains on your application such as URLs that contain ".aspx" or ".php".

Django does all of these things well — and introduces a number of features that raise the bar for what a Web framework should do.

Why Django for Web Development?

- Lets you divide code modules into logical groups to make it flexible to change

  MVC design pattern (MVT)
- Provides auto generated web admin to ease the website administration
- Provides pre-packaged API for common user tasks
- Provides you template system to define HTML template for your web pages to avoid code duplication

  DRY Principle
- Allows you to define what URL be for a given Function

  Loosely Coupled Principle
- Allows you to separate business logic from the HTML

  Separation of concerns
- Everything is in python (schema/settings)

History

- Named after famous Guitarist “Django Reinhardt” .
- Developed by Adrian Holovatyand Jacob Kaplan-moss at World Online News for efficient development .
- Open sourced in 2005
- First Version released September 3, 2008

Django with Python Overview

The framework is written in Python, a beautiful, concise, powerful, high-level programming language.

To develop a site using Django, we write Python code that uses the Django libraries.

…that encourages rapid development…

Regardless of how many powerful features it has, a Web framework is worthless if it doesn’t save our time.

Django’s philosophy is to do all it can to facilitate hyper-fast development.

With Django, we build Web sites in a matter of hours, not days; weeks, not years.

Python Features

- Python is an **interpreted language**,

which means there’s no need to compile code.

Just write your program and execute it.

In Web development, this means you can develop code and immediately see results by hitting “reload” in your Web browser.

- Python is **dynamically typed**,

which means you don’t have to worry about declaring data types for your variables.

- Python syntax is **concise yet expressive**,

which means it takes less code to accomplish the same task than in other,

more verbose, languages such as Java.

One line of python usually equals 10 lines of Java.

- Python offers **powerful introspection and meta-programming** features, which make it possible to inspect and add behavior to objects at runtime.

Beyond the productivity advantages inherent in Python, Django itself makes every effort to encourage rapid development,clean and pragmatic design

Finally, Django strictly maintains a clean design throughout its own code and makes it easy to follow best Web-development practices in the applications you create.

Data structure

Data strucutre is an structures which can hold some data together. In other words they are used to store a collection of related data.

There are four built-in data structures in Python:

- list
- tuple
- dictionary
- set

LIST

In python, A list is a data structure that holds an ordered collection of items i.e. you can store a sequence of items in a list.

In your python shell, define a variable called "django" and store some elements value into it.

empty

listdjango = []

list of integersdjango = [1, 2, 3]

list with mixed datatypesdjango = [1, "Hello", 3.4]

nested listdjango = ["mouse", [8, 4, 6]]

Accessing Values in Lists

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example:

>>> django = [1, 2, 3]>>> print django[0]

TUPLE

Tuples are sequences, just like lists.

The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Creating a tuple is as simple as putting different comma-separated values.

Let's define a tuple:

>>> best_os = ("ubuntu", "macos", "solaris" "windows")

Accessing tuple is same as list.

Since, we defined a tuple, let's access the best operating system in same python shell.

>>> print best_os[0]ubuntu>>>

DICTIONARY

Dictionary is another data type built into python.

Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by keys, which can be any immutable type; strings and numbers can always be keys.

Tuples can be used as keys if they contain only strings, numbers, or tuples;

if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key.

You can't use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like append() and extend().

SET

In simplest terms set can be defined as - set is an unordered collections of unique elements.

Or Sets are lists with no duplicate entries.

Let's say you want to collect a list of words used in a paragraph:

>>> print set("Hello Django is an open initiatives for python classes.".split())set(['initiatives', 'for', 'python', 'is', 'an', 'classes.', 'Django', 'open', 'Hello'])

Django with MVC

Django follows the "model-view-controller" (MVC) architecture.

Simply put, this is a way of developing software so that the code for defining and accessing data (the model) is separate from the business logic (the controller), which in turn is separate from the user interface (the view).

Django supports the MVC pattern.

The MVC pattern is a software architecture pattern that separates data presentation from the logic of handling user interactions(in other words, saves you stress:),

It has been around as a concept for a while, and has invariably seen an exponential growth in use since its inception.

It has also been described as one of the best ways to create client-server applications, all of the best frameworks for web are all built around the MVC concept

Overview of MVC

Model: This handles your data representation, it serves as an interface to the data stored in the database itself, and also allows you to interact with your data without having to get perturbed with all the complexities of the underlying database.

View: As the name implies, it represents what you see while on your browser for a web application or In the UI for a desktop application.

Controller: provides the logic to either handle presentation flow in the view or update the model's data i.e it uses programmed logic to figure out what is pulled from the database through the model and passed to the view, also gets information from the user through the view and implements the given logic by either changing the view or updating the data via the model , To make it more simpler, see it as the engine room.

Here is a rundown of steps involved in an MVC blog application.

1. Web browser or client sends the request to the web server, asking the server to display a blog post.
2. The request received by the server is passed to the controller of the application.
3. The controller asks the model to fetch the blog post.
4. The model sends the blog post to the controller.
5. The controller then passes the blog post data to the view.
6. The view uses blog post data to create an HTML page.
7. At last, the controller returns the HTML content to the client.

The MVC pattern not only helps us to create and maintain a complex application.

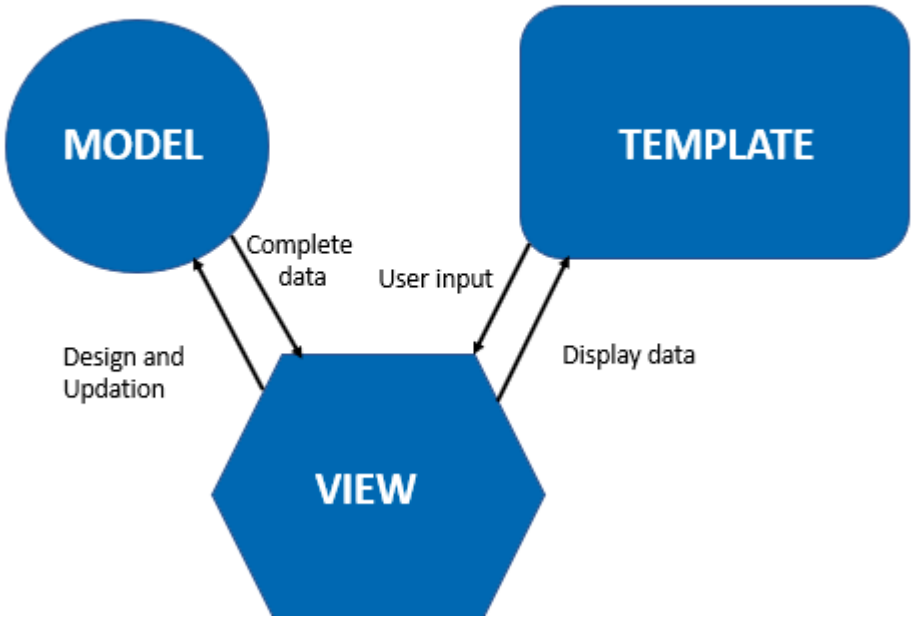
It really shines when it comes to separation of concerns.

For example, In a web development company, there are web designers and there are developers. The job of web designers is to create views. The developers take these views and incorporate them with models and controllers.

Django MTV

- Django follows MVC pattern very closely but it uses slightly different terminology.
- Django is essentially an MTV (Model-Template-View) framework.
- Django uses the term Templates for Views and Views for Controller.
- In other words, in Django views are called templates and controllers are called views.
- Hence our HTML code will be in templates and Python code will be in views and models.

MTV Architecture Components (Model, Template, and View)



In Django, the model does the linking to the database and each model gets mapped to a single table in the database.

These fields and methods are declared under the file models.py

With this linking to the database, we can actually each and every record or row from that particular table and can perform the DML operations on the table.

Django.db.models.

The model is the subclass that is used here.

We can use the import statement by defining as from django.db import models.

So after defining our database tables, columns and records; we are going to get the data linked to our application by defining the mapping in settings.py file under the INSTALLED_APPS.

This template helps us to create a dynamic website in an easy manner.

The dynamic website deals with dynamic data.

Dynamic data deals with a scenario where each user is displayed with their personalized data; as Facebook feeds, Instagram feeds, etc.

The configuration of the template is done in settings.py file under INSTALLED_APPS. So python code would search for the files under the template subdirectory.

We can create an HTML file or import any dynamic web page from the browser and place it under the template folder.

And after that our usual linking of this file in urls.py and views.py to get a response is mandatory.

In this way after linking all these together and running the server, we can get our web application ready.

View

This is the part where actually we would be mentioning our logic. T

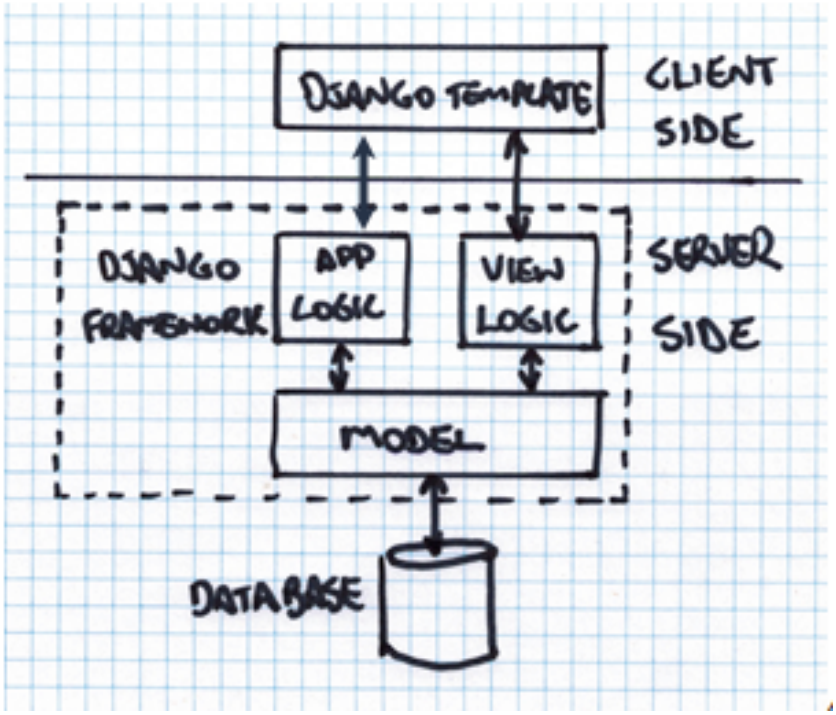
The coding is done through the python file views.py

This view also sends responses to the user when the application is used, to understand briefly, we can say that this view.py can deal with HttpResponse.

Now, after creating a view, how can we link it to our application?

How do you think that the system is going to understand to display a particular view?

This can be done by mapping the views.py in urls.py file. As already mentioned, urls.py keeps track of all those different pages that we created and hence map each of them.



Django Working

We have some components and two regions i.e., server side and client side.

Here you will notice that the View is on the server-side part while the template is on the client side.

Now, when we request for the website, the interface through which we use to make that request via our browser was the Template.

Then that request transmits to the server for the management of view file.

Django is literally a play between the requests and responses.

So whenever our Template is updating it's the input (request) we sent from here which on the server was seen by the View.

And, then it transports to the correct URL. It's one of the important components of Django MTV architecture.

There, the URL mapping in Django is actually done in regular expressions.

These expressions are much more understandable than IP addresses.

Now after the sending of a request to the correct URL, the app logic applies and the model initiates to correct response to the given request.

Then that particular response is sent back to the View where it again examines the response and transmits it as an HTTP response or desired user format.

Then, it again renders by the browser via Templates.

Django Uses

1. Django is time-tested

It's been 13 years Django started developing its framework and the first release of open source commit as it was under development quite a long time before release. During these years it had many releases some of them have new features other releases focuses on security enhancements etc. Django is the first framework to respond to new issues and vulnerabilities and alter other frameworks to make patches to frameworks. The Latest release of it is focusing on new features and boundary case problems.

2. Application Development

Django was developed by online news operation team with an aim to create web applications using the Python programming language. The framework has templates, libraries, and APIS which work together. In general, applications developed using Django can be upgraded with minimal cost, changes, and additions and it make a lot of web development easier.

3. Easy to Use

Django uses Python programming language which is a popular language in 2015 and now most choosing language by programmers who are learning to code and applications of Django framework is widely used as it is free and open-source, developed and maintained by a large community of developers. It means we can find answers to the problems easily using Google.

4. Operating System Dependent

Django framework runs on any platform like PC, Windows, Mac, Linux etc. It provides a layer between the developer and database called ORM (object-relational mapper) which makes it

possible to move or migrate our applications to other major databases with few lines of code change.

5. Excellent Documentation for real-world application

Applications of Django has one of the best documentation for its framework to develop different kinds of real-world applications whereas many other frameworks used an alphabetical list of modules, attributes, and methods. This is very useful for quick reference for developers when we had confused between two methods or modules but not for fresher's who are learning for the first time. It's a difficult task for Django developers to maintain the documentation quality as it is one of the best open source documentation for any framework.

6. Scalable and Reliable

As Django is a well-maintained web application framework and widely used across the industries so cloud providers taking all measures to provide support to run Django applications easily and quickly on cloud platforms. It means, once Django applications deployed then it can be managed by an authorized developer with a single command in a cloud environment. As Django developers are working in the same development environment for a long time so they will grow and expertise in these areas which means applications developed, websites created are getting better day by day, more functional, efficient and reliable.

7. Community Support

Django community is one of the best communities out there as it is governed by the Django software foundation which had some rules like for event there is a code of conduct. Django communities will have IRC and mailing list most welcome, even it may have bad appeals it will rectify immediately. Django offers stability, packages, documentation and a good community.

8. DRY – Don't repeat yourself

Django framework follows don't repeat yourself principle as it concentrates on getting most out of each and every line of code by which we can spend less time on debugging or code re-orientation etc. In general DRY code means all uses of data change simultaneously rather than a need to be replicated and its fundamental reason to use of variables and functions in all programming.

9. Batteries of Django

Django framework has everything to build a robust framework with main features as below:

Template layers,

Forms, development process,

Views layers, security,

Model layers, python compatibility,

Localization, performance, and optimization

Geographic framework, common tools for web application development

Other core functionalities required for websites.

As Django can be used to build any type of website with help of its frameworks like content management, Wikipedia pages, social networking applications, chat applications, and websites like Mozilla, Instagram, Pinterest, BitBucket etc. Django can work with any client-server applications and able to deliver content in any form (HTML, text, JSON, XML, RSS etc.)

10. Django Benefits

With the uses of Django framework, we can develop and deploy web applications within hours as it takes care of much of the hassle of web development. Django is very fast, fully loaded such as it takes care of user authentication, content administration, security as Django takes it very seriously and helps to avoid SQL injection, cross-site scripting etc. and scalable as applications can be scalable to meet high demands and used to build any type of applications that's why we call it as versatile framework. We can build different applications from content management to social networking websites using Django framework. It offers lots of resources and good documentation which helps new learners to learn and experienced people for reference.

Creating Django Views –

A view function, or "view" for short, is simply a Python function that takes a web request and returns a web response. This response can be the HTML contents of a Web page, or a redirect, or a 404 error, or an XML document, or an image, etc.

Make a Simple View

We will create a simple view in myapp to say "welcome to my app!"

See the following view –

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

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

In this view, we use HttpResponse to render the HTML (as you have probably noticed we have the HTML hard coded in the view).

We used HttpResponse to render the HTML in the view before. This is not the best way to render pages. Django supports the MVT pattern so to make the precedent view, Django - MVT like, we will need a template.

And now our view will look like –

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

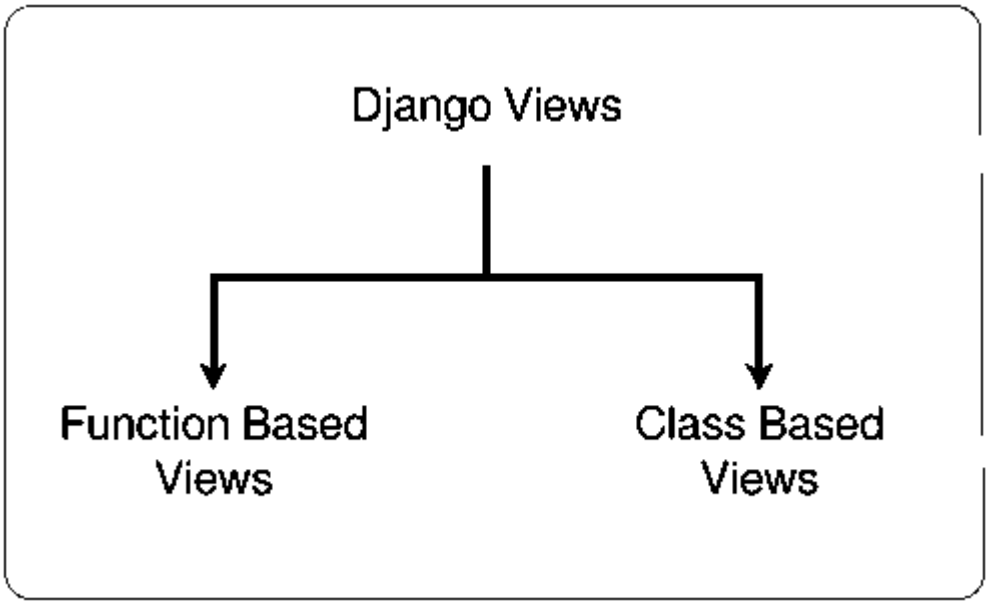
Views can also accept parameters –

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

Types of Views

Django views are divided into two major categories :-

- Function Based Views
- Class Based Views



Function Based Views: Function based views are writter using a function in python which recieves as an argument HttpRequest object and returns an HttpResponse Object. Function based views are generally divided into 4 basic strategies, i.e., CRUD (Create, Retrieve, Update, Delete). CRUD is the base of any framework one is using for development.

Function based view Example –

Let's Create a function based view list view to display instances of a model.
let's create a model of which we will be creating instances through our view.
In geeks/models.py,

```
# import the standard Django Model
# from built-in library
from django.db import models

# declare a new model with a name "GeeksModel"
class GeeksModel(models.Model):

    # fields of the model
    title = models.CharField(max_length = 200)
    description = models.TextField()

    # renames the instances of the model
    # with their title name
```

```
    def __str__(self):
        return self.title
```

After creating this model, we need to run two commands in order to create Database for the same.

Now, let us create a view and Template for the same –

```
from django.shortcuts import render

# relative import of forms
from .models import GeeksModel


def list_view(request):
    # dictionary for initial data with
    # field names as keys
    context ={}

    # add the dictionary during initialization
    context["dataset"] = GeeksModel.objects.all()

    return render(request, "list_view.html", context)
```

Django CRUD (Create, Retrieve, Update, Delete) Function Based Views :-

- Create View – Function based Views Django
- Detail View – Function based Views Django
- Update View – Function based Views Django
- Delete View – Function based Views Django

Class Based Views

Class-based views provide an alternative way to implement views as Python objects instead of functions. They do not replace function-based views, but have certain differences and advantages when compared to function-based views:

Organization of code related to specific HTTP methods (GET, POST, etc.) can be addressed by separate methods instead of conditional branching.

Object oriented techniques such as mixins (multiple inheritance) can be used to factor code into reusable components.

Django HttpResponse

HttpResponse is a response class with string data. While HttpRequest is created by Django, HttpResponse is created by programmer.

HttpResponse has subclasses including JsonResponse, StreamingHttpResponse, and FileResponse.

Django HttpResponse example

In the following example, we create a Django application that sends text data to the client. We send today's datetime.

$ mkdir httpresponse

$ cd httpresponse

$ mkdir src

$ cd src

We create the project and the and src directories. Then we locate to the src directory.

$ django-admin startproject httpresponse .

We create a new Django project in the src directory.

Note: If the optional destination is provided, Django will use that existing directory as the project directory. If it is omitted, Django creates a new directory based on the project name. We use the dot (.) to create a project inside the current working directory.

```
$ cd ..
$ pwd
/c/Users/Jano/Documents/pyprogs/django/httpresponse
```

We locate to the project directory.

```
$ tree /f
src
|   db.sqlite3
|   manage.py
|
└───httpresponse
        settings.py
        urls.py
        views.py
        wsgi.py
        __init__.py
```

The Django way is to put functionality into apps, which are created with django-admin startapp. In this tutorial, we do not use an app to make the example simpler. We focus on demonstrating how to send HttpResponse.

HttpRequest in Django

Request and Response Objects –

Django uses request and response objects to pass state through the system.

When a page is requested, Django creates an HttpRequest object that contains metadata about the request. Then Django loads the appropriate view, passing the HttpRequest as the first argument to the view function. Each view is responsible for returning an HttpResponse object.

HttpRequest and HttpResponse object Example

To explain these objects let's create a view home as below in views.py

```
# importing HttResponse from library
from django.http import HttpResponse
def home(request):
    # request is handled using HttpResponse object
    return HttpResponse("Any kind of HTML Here")
```

To handle the request let us map a URL to this view in urls.py

```
# importing view from views.py
from .views import home
urlpatterns = [
    path('', home),
]
```

HttpRequest Attributes – Django

You can use following attributes with HttpRequest for advanced manipulation

ATTRIBUTE	DESCRIPTION
HttpRequest.scheme	A string representing the scheme of the request (HTTP or HTTPs usually).
HttpRequest.body	It returns the raw HTTP request body as a byte string.
HttpRequest.path	It returns the full path to the requested page does not include the scheme or domain.
HttpRequest.path_info	It shows path info portion of the path.
HttpRequest.method	It shows the HTTP method used in the request.
HttpRequest.encoding	It shows the current encoding used to decode form submission data.
HttpRequest.content_type	It shows the MIME type of the request, parsed from the CONTENT_TYPE header.

ATTRIBUTE	DESCRIPTION
HttpRequest.content_params	It returns a dictionary of key/value parameters included in the CONTENT_TYPE header.
HttpRequest.GET	It returns a dictionary-like object containing all given HTTP GET parameters.
HttpRequest.POST	It is a dictionary-like object containing all given HTTP POST parameters.
HttpRequest.COOKIES	It returns all cookies available.
HttpRequest.FILES	It contains all uploaded files.
HttpRequest.META	It shows all available Http headers.
HttpRequest.resolver_match	It contains an instance of ResolverMatch representing the resolved URL.

HttpRequest Methods – Django

You can use following methods with HttpRequest for advanced manipulation

ATTRIBUTE	DESCRIPTION

ATTRIBUTE	DESCRIPTION
HttpRequest.get_host()	It returns the original host of the request.
HttpRequest.get_port()	It returns the originating port of the request.
HttpRequest.get_full_path()	It returns the path, plus an appended query string, if applicable.
HttpRequest.build_absolute_uri *(location)*	It returns the absolute URI form of location.
HttpRequest.get_signed_cookie *(key, default=RAISE_ERROR, salt=", max_age=None)*	It returns a cookie value for a signed cookie, or raises a django.core.signing.BadSignature exception if the signature is no longer valid.
HttpRequest.is_secure()	It returns True if the request is secure; that is, if it was made with HTTPS.
HttpRequest.is_ajax()	It returns True if the request was made via an XMLHttpRequest.

Configuring URL Confs:-

What is a URL?

URL stands for Uniform Resource Locator. It is the address used by your server to search for the right webpage.

All the servers take the URL which you searched in the browser and via that server, provides you the correct result and if they don't find anything matching the URL, it will show 404 FILE NOT FOUND ERROR.

URLs in Django

How does Django Server interpret URLs?

Django interprets URLs in a rather different way, the URLs in Django are in the format of regular expressions, which are easily readable by humans than the traditional URLs of PHP frameworks.

Regular Expressions

A Regular Expression also called RegEx is a format for search pattern in URLs which is much cleaner and easy to read for the humans and is very logical.That's important because it makes the process of SEO much easier then it would be with the traditional URL approach which contains much more special characters.

When we search for a URL in the URL Bar like this: https://www.bytehash.com/

The same URL is passed to the server after DNS connects us to the appropriate server. While your Django-server is running, it will receive the URL and search for the URL in the urls.py file or the value in the ROOT_URLCONF variable.

As from the name, this variable contains the address of the urls.py file. The server will now match the urls from that file.

Expression Example: Your server will now look for the URL patterns inside the urls.py file.Here is an example of expression used in URL.

```
    django contrib        admin
    django urls        path

urlpatterns   [
        ( admin/   admin site urls)
]
```

This is a list urlpatterns and name is the convention which should not be changed. In this list, we will be adding all the urls and patterns to be available for searching on our website.

In this file, the Django server matches the Regular Expression with the given URL, like 'admin/' in the path function parameter. Here you may wonder that it's not the site URL. Actually, it snips off the part of URL till website and only searches for the remaining part of URL.

Then it redirects the page to the provided webpage or python file 'admin.site.urls' in the second parameter, which will finally reach the views like functions and class objects or even another urls will return appropriate template.

Modifying urls.py file

The urls.py file in Django is like the address book of your Django website. It stores all the web addresses for your website. It connects that to some view component or any other urls-conf file for a certain application.

Importing the view function

We will be learning the steps to import the view functions and connecting different url-conf files with the main urls.py file.

A URLconf is similar to a table of contents for our Django-powered web site. It's a mapping between URL patterns and the view functions that need to be called for those URLs.
First, we will have to import the view function that we want our server to run when the URL is matched. For that, we have to import the function from the views.py file.

django.conf.urls functions for use in URLconfs

static()

Syntax: static.static(prefix, view=django.views.static.serve, **kwargs)

Helper function to return a URL pattern for serving files in debug mode:

```
from django.conf import settings
```

```
from django.conf.urls.static import static

urlpatterns = [

    # ... the rest of your URLconf goes here ...

] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Multiple URLConfs:

You need to expand your regex to match the additional characters that you want to allow in your URL.Use alphanumeric characters along with +, to make multiple confs work:

```
urlpatterns = patterns('',

    url(r'^product/(?P<name>[\w\+]+)/$', 'crunch.views.product_by_name',
name='preview_by_name'),

)
```

Passing Arguments in URL –

For passing argument in URL from a django application, you need to define a variable on the url. For example:

url(r'^pay/summary/(?P<value>\d+)/$', views.pay_summary, name='pay_summary')),

It could be a string true/false by replacing \d+ to \s+, but you would need to interpret the string. You can then use:
my link

Routing in Django REST

Routers are used with **ViewSets** in django rest framework to auto config the urls. Routers provides a simple, quick and consistent way of wiring ViewSet logic to a set of URLs. Router automatically maps the incoming request to proper viewset action based on the request method type(i.e GET, POST, etc). Let's start using routers.

We have two types of routers in Django REST

1. **SimpleRouter**
   - It includes actions for all actions (i.e `list`, `create`, `retrieve`, `update`, `partial_update` and `destroy`).
   - Let's see how it configures the urls

URL Style	HTTP Method	Action	URL Name
{prefix}/	GET	list	{basename}-list
	POST	create	
{prefix}/{url_path}/	GET, or as specified by `methods` argument	`@action(detail=False)` decorated method	{basename}-{url_name}
{prefix}/{lookup}/	GET	retrieve	{basename}-detail
	PUT	update	
	PATCH	partial_update	
	DELETE	destroy	
{prefix}/{lookup}/{url_path}/	GET, or as specified by `methods` argument	`@action(detail=True)` decorated method	{basename}-{url_name}

2. **DefaultRouter**
   - It also includes all actions just like `SimpleRouter` and it additionally provides a default root view which returns a response containing hyperlinks to all the list views.
   - Let's see how it configures the urls

URL Style	HTTP Method	Action	URL Name

[.format]	GET	automatically generated root view	api-root
{prefix}/[.format]	GET	list	{basename}-list
	POST	create	
{prefix}/{url_path}/[.format]	GET, or as specified by `methods` argument	`@action(detail=False)` decorated method	{basename}-{url_name}
{prefix}/{lookup}/[.format]	GET	retrieve	{basename}-detail
	PUT	update	
	PATCH	partial_update	
	DELETE	destroy	
{prefix}/{lookup}/{url_path}/[.format]	GET, or as specified by `methods` argument	`@action(detail=True)` decorated method	{basename}-{url_name}

Django Templates

Template Fundamentals –

Django provides a convenient way to generate dynamic HTML pages by using its template system. A template consists of static parts of the desired HTML output as well as some special syntax describing how dynamic content will be inserted.

Why Django Template?

In HTML file, we can't write python code because the code is only interpreted by python interpreter not the browser. We know that HTML is a static markup language, while Python is a dynamic programming language. Django template engine is used to separate the design from the python code and allows us to build dynamic web pages.

Template Tags:

In a template, Tags provide arbitrary logic in the rendering process. For example, a tag can output content, serve as a control structure e.g. an "if" statement or a "for" loop, grab content from a database etc.

Tags are surrounded by {% %} braces. For example.

```
{% csrf_token %}

{% if user.is_authenticated %}

  Hello, {{ user.username }}.

{% endif %}
```

Template Filters:

Filters the contents of the block through one or more filters. Multiple filters can be specified with pipes and filters can have arguments, just as in variable syntax. It is to be noted that the block includes all the text between the filter and endfilter tags.

Sample usage:

```
{% filter force_escape|lower %}

  This text will be HTML-escaped, and will appear in all lowercase.

{% endfilter %}
```

Note: The escape and safe filters are not acceptable arguments. Instead, use the autoescape tag to manage autoescaping for blocks of template code.

Template Inheritance in Django: -

The include tag allows us to include the contents of a template inside another template making the inheritance of template possible. Here is the syntax of the include tag:

{% include template_name %}

The template_name could be a string or a variable.

Rendering of Templates –

Render a HTML Template as Response –

A view function, or view for short, is simply a Python function that takes a Web request and returns a Web response. This article revolves around how to render an HTML page from Django using views. Django has always been known for its app structure and ability to manage applications easily. Let's dive in to see how to render a template file through a Django view.

After you have a project and an app, open views.py and let's start creating a view called geeks_view which is used print "Hello world" through a template file. A django view is a python function which accept an argument called request and returns an response.

Enter following code into views.py of app

```
from django.shortcuts import render

# Create your views here.

def geeks_view(request):

    # render function takes argument  - request

    # and return HTML as response

    return render(request, "home.html")
```

But this code won't work until into define a proper mapping of URL. Mapping means you need to tell Django what a user enters in the browser to render your particular view.

```
from django.contrib import admin

from django.urls import path, include

urlpatterns = [

    path('admin/', admin.site.urls),

    path('', include("geeks.urls")),

]
```

RequestContext to render_to_response

The Django documentation recommends passing RequestContext as an argument to render_to_response like this:

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


def my_view(request):
    # View code here...
    return render_to_response('my_template.html',
                              my_data_dictionary,
                              context_instance=RequestContext(request))
```

Writing Global Content Processor –

You can put this function anywhere in your project. My preference is to place this code inside a context_processors.py file under the related app.

```
# The context processor function
def categories(request):
    all_categories = Category.objects.all()
    return {
        'categories': all_categories,
    }
```

DJANGO FORMS –

Unless you're planning to build websites and applications that do nothing but publish content, and don't accept input from your visitors, you're going to need to understand and use forms. Django provides a range of tools and libraries to help you build forms to accept input from site visitors, and then process and respond to the input.

HTML forms

In HTML, a form is a collection of elements inside <form>...</form> that allow a visitor to do things like enter text, select options, manipulate objects or controls, and so on, and then send that information back to the server.

Some of these form interface elements - text input or checkboxes - are built into HTML itself. Others are much more complex; an interface that pops up a date picker or allows you to move a slider or manipulate controls will typically use JavaScript and CSS as well as HTML form <input> elements to achieve these effects.

As well as its <input> elements, a form must specify two things:

where: the URL to which the data corresponding to the user's input should be returned

how: the HTTP method the data should be returned by

As an example, the login form for the Django admin contains several <input> elements: one of type="text" for the username, one of type="password" for the password, and one of type="submit" for the "Log in" button. It also contains some hidden text fields that the user doesn't see, which Django uses to determine what to do next.

It also tells the browser that the form data should be sent to the URL specified in the <form>'s action attribute - /admin/ - and that it should be sent using the HTTP mechanism specified by the method attribute - post.

When the <input type="submit" value="Log in"> element is triggered, the data is returned to /admin/.

Here is an example:

Forms.py

```
from django import forms

from .models import Post

class PostForm(forms.ModelForm):

    class Meta:

        model = Post
```

```
fields = ('title', 'text',)
```

Django Forms

Django provides a Form class which is used to create HTML forms. It describes a form and how it works and appears. It is similar to the ModelForm class that creates a form by using the Model, but it does not require the Model. Each field of the form class map to the HTML form <input> element and each one is a class itself, it manages form data and performs validation while submitting the form.

Example:

```
from django import forms

class StudentForm(forms.Form):
    firstname = forms.CharField(label="Enter first name",max_length=50)
    lastname  = forms.CharField(label="Enter last name", max_length = 100)
```

Commonly used fields and their details are given in the below table.

Name	Class	HTML Input	Empty value
BooleanField	class BooleanField(**kwargs)	CheckboxInput	False
CharField	class CharField(**kwargs)	TextInput	Whatever you've given as empty_value.
ChoiceField	class ChoiceField(**kwargs)	Select	'' (an empty string)
DateField	class DateField(**kwargs)	DateInput	None
DateTimeField	class DateTimeField(**kwargs)	DateTimeInput	None
DecimalField	class DecimalField(**kwargs)	NumberInput	None
EmailField	class EmailField(**kwargs)	EmailInput	'' (an empty string)
FileField	class FileField(**kwargs)	ClearableFileInput	None

ImageField	class ImageField(**kwargs)	ClearableFileInput	None

Form Authentication:

Django authentication provides both authentication and authorization together and is generally referred to as the authentication system, as these features are somewhat coupled.

User objects: User objects are the core of the authentication system. They typically represent the people interacting with your site and are used to enable things like restricting access, registering user profiles, associating content with creators etc. Only one class of user exists in Django's authentication framework, i.e., 'superusers' or admin 'staff' users are just user objects with special attributes set, not different classes of user objects.

The primary attributes of the default user are:

- username
- password
- email
- first_name
- last_name

Creating users

The most direct way to create users is to use the included create_user() helper function:

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.create_user('john', 'lennon@thebeatles.com', 'johnpassword')
# At this point, user is a User object that has already been saved
# to the database. You can continue to change its attributes
# if you want to change other fields.
>>> user.last_name = 'Lennon'
>>> user.save()
```

If you have the Django admin installed, you can also create users interactively.

Creating superusers: Create superusers using the createsuperuser command:

$ python manage.py createsuperuser --username=joe --email=joe@example.com

You will be prompted for a password. After you enter one, the user will be created immediately. If you leave off the --username or --email options, it will prompt you for those values.

Changing passwords: -

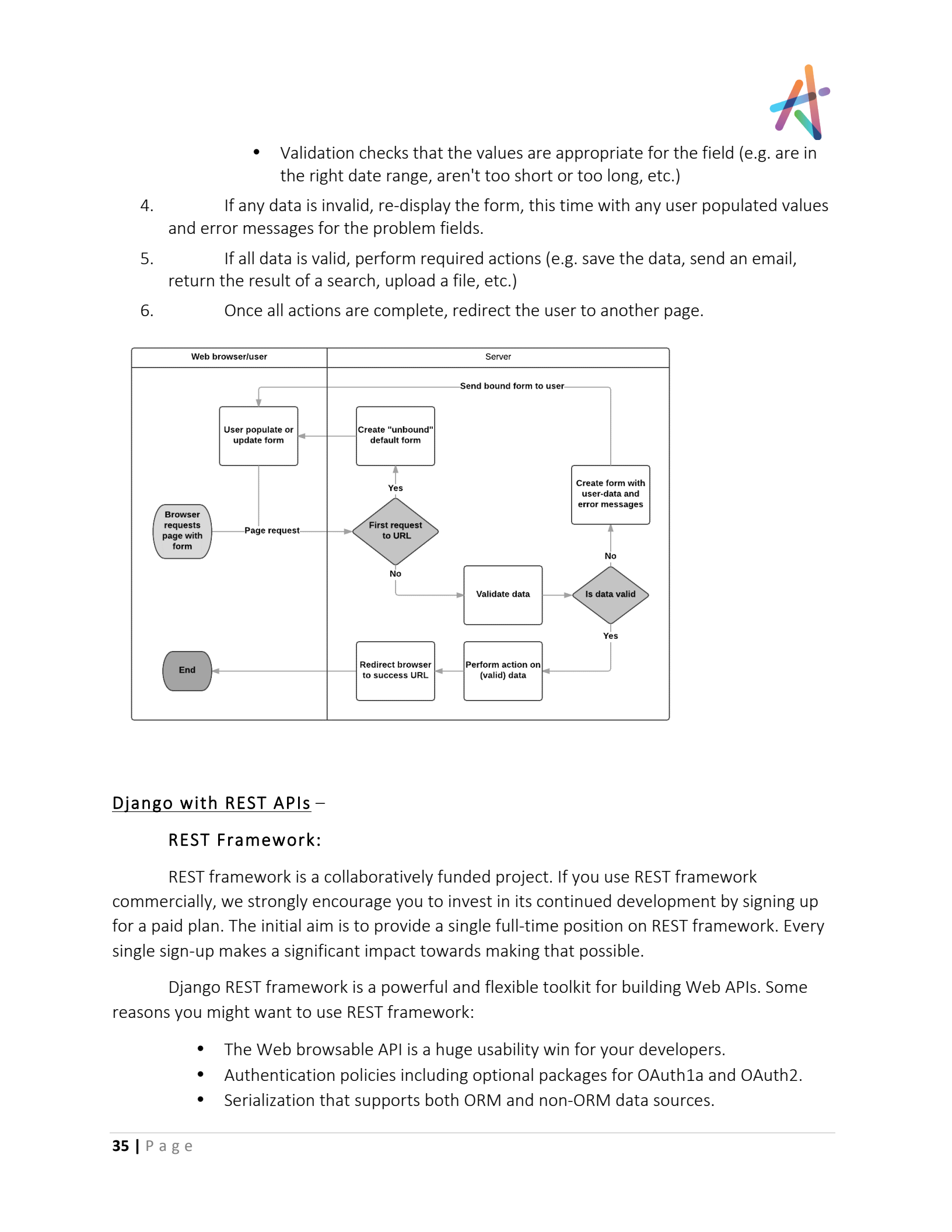
Django does not store raw (clear text) passwords on the user model, but only a hash. You can also change a password programmatically, using set_password():

```
>>> from django.contrib.auth.models import User
>>> u = User.objects.get(username='john')
>>> u.set_password('new password')
>>> u.save()
```

Django Form Processing Technique –

Based on the diagram above, the main things that Django's form handling does are:

1. Display the default form the first time it is requested by the user.
   - The form may contain blank fields (e.g. if you're creating a new record), or it may be pre-populated with initial values (e.g. if you are changing a record, or have useful default initial values).
   - The form is referred to as *unbound* at this point, because it isn't associated with any user-entered data (though it may have initial values).
2. Receive data from a submit request and bind it to the form.
   - Binding data to the form means that the user-entered data and any errors are available when we need to redisplay the form.
3. Clean and validate the data.
   - Cleaning the data performs sanitization of the input (e.g. removing invalid characters that might be used to send malicious content to the server) and converts them into consistent Python types.



- Validation checks that the values are appropriate for the field (e.g. are in the right date range, aren't too short or too long, etc.)

4. If any data is invalid, re-display the form, this time with any user populated values and error messages for the problem fields.
5. If all data is valid, perform required actions (e.g. save the data, send an email, return the result of a search, upload a file, etc.)
6. Once all actions are complete, redirect the user to another page.

Django with REST APIs –

REST Framework:

REST framework is a collaboratively funded project. If you use REST framework commercially, we strongly encourage you to invest in its continued development by signing up for a paid plan. The initial aim is to provide a single full-time position on REST framework. Every single sign-up makes a significant impact towards making that possible.

Django REST framework is a powerful and flexible toolkit for building Web APIs. Some reasons you might want to use REST framework:

- The Web browsable API is a huge usability win for your developers.
- Authentication policies including optional packages for OAuth1a and OAuth2.
- Serialization that supports both ORM and non-ORM data sources.

- Customizable all the way down - just use regular function-based views if you don't need the more powerful features.
- Extensive documentation, and great community support.

Requirements

REST framework requires the following:

- Python (3.5, 3.6, 3.7, 3.8)
- Django (1.11, 2.0, 2.1, 2.2, 3.0)

We **highly recommend** and only officially support the latest patch release of each Python and Django series.

The following packages are optional:

- coreapi (1.32.0+) - Schema generation support.
- Markdown (3.0.0+) - Markdown support for the browsable API.
- Pygments (2.4.0+) - Add syntax highlighting to Markdown processing.
- django-filter (1.0.1+) - Filtering support.
- django-guardian (1.1.1+) - Object level permissions support.

Testing in Django

Automated testing is an extremely useful bug-killing tool for the modern Web developer. You can use a collection of tests – a **test suite** – to solve, or avoid, a number of problems:

- When you're writing new code, you can use tests to validate your code works as expected.
- When you're refactoring or modifying old code, you can use tests to ensure your changes haven't affected your application's behavior unexpectedly.

Testing a Web application is a complex task, because a Web application is made of several layers of logic – from HTTP-level request handling, to form validation and processing, to

template rendering. With Django's test-execution framework and assorted utilities, you can simulate requests, insert test data, inspect your application's output and generally verify your code is doing what it should be doing.

The preferred way to write tests in Django is using the `unittest` module built-in to the Python standard library. This is covered in detail in the Writing and running tests document.

You can also use any *other* Python test framework; Django provides an API and tools for that kind of integration.

Unit Testing with Django –

Unit Testing in Python – Unittest

What is Unit Testing?

Unit Testing is the first level of software testing where the smallest testable parts of a software are tested. This is used to validate that each unit of the software performs as designed. The unittest test framework is python's xUnit style framework.

Method:

White Box Testing method is used for Unit testing. OOP concepts supported by unittest framework are:

test fixture:

A test fixture is used as a baseline for running tests to ensure that there is a fixed environment in which tests are run so that results are repeatable.

Examples :

creating temporary databases.

starting a server process.

test case:

A test case is a set of conditions which is used to determine whether a system under test works correctly.

test suite:

Test suite is a collection of testcases that are used to test a software program to show that it has some specified set of behaviors by executing the aggregated tests together.

test runner:

A test runner is a component which set up the execution of tests and provides the outcome to the user.

Basic Test Structure:

Unittest defines tests by the following two ways :

- Manage test “fixtures” using code.
- test itself.

```
import unittest

class SimpleTest(unittest.TestCase):

    # Returns True or False.

    def test(self):

        self.assertTrue(True)


if __name__ == '__main__':

    unittest.main()
```

This is the basic test code using unittest framework, which is having a single test. This test() method will fail if TRUE is ever FALSE.

Running Tests

```
if __name__ == '__main__':

    unittest.main()
```

The above written block helps to run the test by running the file through the command line.

Testing tools

Django provides a small set of tools that come in handy when writing tests.

The test client

The test client is a Python class that acts as a dummy Web browser, allowing you to test your views and interact with your Django-powered application programmatically.

Some of the things you can do with the test client are:

- Simulate GET and POST requests on a URL and observe the response – everything from low-level HTTP (result headers and status codes) to page content.
- See the chain of redirects (if any) and check the URL and status code at each step.
- Test that a given request is rendered by a given Django template, with a template context that contains certain values.

Note that the test client is not intended to be a replacement for Selenium or other "in-browser" frameworks. Django's test client has a different focus. In short:

- Use Django's test client to establish that the correct template is being rendered and that the template is passed the correct context data.
- Use in-browser frameworks like Selenium to test *rendered* HTML and the *behavior* of Web pages, namely JavaScript functionality. Django also provides special support for those frameworks; see the section on `LiveServerTestCase` for more details.

A comprehensive test suite should use a combination of both test types.

Advanced Testing

The request factory

class `RequestFactory`

The `RequestFactory` shares the same API as the test client. However, instead of behaving like a browser, the RequestFactory provides a way to generate a request instance that can be used as the first argument to any view. This means you can test a view function the same way as you would test any other function – as a black box, with exactly known inputs, testing for specific outputs.

The API for the `RequestFactory` is a slightly restricted subset of the test client API:

- It only has access to the HTTP methods `get()`, `post()`, `put()`, `delete()`, `head()`, `options()`, and `trace()`.
- These methods accept all the same arguments *except* for `follow`. Since this is just a factory for producing requests, it's up to you to handle the response.
- It does not support middleware. Session and authentication attributes must be supplied by the test itself if required for the view to function properly.

Debugging:

pdb (The Python Debugger) and ipdb (IPython pdb): **pdb** is like the JavaScript debugger. You can drop an import pdb; pdb.set_trace() in your code and it will provide a REPL for you to step through your code. **ipdb** is like an enhanced version pdb. Useful features include syntax highlighting and tab completion. It is enormously helpful to be able to inspect objects.

You'll have to add ipdb to your requirements.txt and do the usual pip install -r requirements.txt. And then you can add import ipdb; ipdb.set_trace() anywhere in your code to use the interactive debugger.

Debugging Django in Docker: If you are using docker, you might notice that dropping a pdb or ipdb into your code does nothing. To fix this, you'll have to run docker with service ports enabled. For example:

```
docker-compose -f docker-compose.yml run --rm --service-ports name_of_container
```

Debugging Django and Gunicorn setup in Vagrant

While pdb and ipdb work when you are using the default runserver command, once you move over to Gunicorn for development, you no longer get the code-stopping functionality. I still have yet to find a good solution to this problem. However, there is a better workaround for this. The answer lies in test-driven development. You can drop pdb or ipdb into your test cases as well as your application code. Then, when you run your test, you'll get the code stopping functionality.

shell_plus: Another useful tool that I have found for debugging is shell_plus. You get this as part of the django-extensions package. This is useful when I need to experiment with various queries using the Django ORM.

DATABASE MODELS: -

Models:

A model is the single, definitive source of information about your data. It contains the essential fields and behaviors of the data you're storing. Generally, each model maps to a single database table.

The basics:

- Each model is a Python class that subclasses django.db.models.Model.
- Each attribute of the model represents a database field.
- With all of this, Django gives you an automatically-generated database-access API; see Making queries.

Quick example: This example model defines a Person, which has a first_name and last_name:

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

```
class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

first_name and last_name are fields of the model. Each field is specified as a class attribute, and each attribute maps to a database column.

The above Person model would create a database table like this:

```
CREATE TABLE myapp_person (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(30) NOT NULL
);
```

How To Create a Django App and Connect it to a Database:

A free and open-source web framework written in Python, Django allows for scalability, re-usability, and rapid development.

The following stepsshows how to set up the initial foundation for a blog website with connections to a MySQL database. This will involve creating the skeleton structure of the blog web application using django-admin, creating the MySQL database and then connecting the web application to the database.

1. Create the Initial Django Project Skeleton: The first thing that we need to do is navigate to the home directory, which we can do with the following command:

   cd ~

   Next, we can list the contents of our current directory:

   ls

   If you've started from scratch with the beginning of this series, you will notice that there is one directory:

   Output:

django-apps

This contains the skeleton project that we generated to verify that everything was installed correctly.

As that was only a test, we won't need this directory. Instead, we'll make a new directory for our blog app. Call the directory something meaningful for the app you are building. As an example, we'll call ours "my_blog_app".

mkdir my_blog_app

Now, navigate to the newly created directory:

cd my_blog_app

Then, create and activate your Python virtual environment.

python3 -m venv env

. env/bin/activate

Now install Django:

pip install django

While in the my_blog_app directory, we will generate a project by running the following command:

django-admin startproject blog

Verify that it worked by navigating to the blog/ directory:

cd blog

The blog/ directory should have been created in the current directory, ~/my_blog_app/, after running the previous django-admin command.

Run ls to verify that the necessary items were created. There should be a blog directory and a manage.py file:

Output

blog manage.py

Now that you've created a project directory containing the initial start of your blog application, we can continue on to the next step.

2. Edit Settings: Since we've generated the skeleton project, we now have a settings.py file. In order for our blog to have the correct time associated with our area, we will edit the settings.py file so that it will be using your current time zone. You can use this list of time zones as a reference. For our example, we will be using India/New Delhi time.

   Now navigate to the directory where the settings.py file is located:

   ```
   cd ~/my_blog_app/blog/blog/
   ```

   Then, using nano or a text editor of your choice, open and edit the settings.py file:

   ```
   nano settings.py
   ```

   We are editing the TIME_ZONE field, so navigate to the bottom section of the file that looks like this:

   ```
   settings.py

   ...

   # Internationalization

   # https://docs.djangoproject.com/en/2.0/topics/i18n/


   LANGUAGE_CODE = 'en-in'


   TIME_ZONE = 'UTC'

   USE_I18N = True

   USE_L10N = True

   USE_TZ = True
   ```

3. Install MySQL Database Connector: In order to use MySQL with our project, we will need a Python 3 database connector library compatible with Django. So, we will install the database connector, mysqlclient, which is a forked version of MySQLdb.

   According to the mysqlclient documentation, "MySQLdb is a thread-compatible interface to the popular MySQL database server that provides the Python database API." The main difference being that mysqlclient has the added benefit of including Python 3 support.

4. Create the Database: Now that the skeleton of your Django application has been set up and mysqlclient and mysql-server have been installed, we will to need to configure your Django backend for MySQL compatibility.

   Verify that the MySQL service is running:

   ```
   systemctl status mysql.service
   ```

Django Model Fields –

Fields

The most important part of a model – and the only required part of a model – is the list of database fields it defines. Fields are specified by class attributes. Be careful not to choose field names that conflict with the models API like clean, save, or delete.

Example:

```
from django.db import models


class Musician(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    instrument = models.CharField(max_length=100)


class Album(models.Model):
    artist = models.ForeignKey(Musician, on_delete=models.CASCADE)
    name = models.CharField(max_length=100)
    release_date = models.DateField()
    num_stars = models.IntegerField()
```

Field types:

Each field in your model should be an instance of the appropriate Field class. Django uses the field class types to determine a few things:

- The column type, which tells the database what kind of data to store (e.g. INTEGER, VARCHAR, TEXT).
- The default HTML widget to use when rendering a form field (e.g. <input type="text">, <select>).

Model Field options

Each field takes a certain set of field-specific arguments (documented in the model field reference). For example, **CharField** (and its subclasses) require a **max_length** argument which specifies the size of the **VARCHAR** database field used to store the data.

There's also a set of common arguments available to all field types. All are optional. They're fully explained in the reference, but here's a quick summary of the most often-used ones:

null

If **True**, Django will store empty values as **NULL** in the database. Default is **False**.

blank

If **True**, the field is allowed to be blank. Default is **False**.

Note that this is different than **null**. **null** is purely database-related, whereas **blank** is validation-related. If a field has **blank=True**, form validation will allow entry of an empty value. If a field has **blank=False**, the field will be required.

choices

A sequence of 2-tuples to use as choices for this field. If this is given, the default form widget will be a select box instead of the standard text field and will limit choices to the choices given.

A choices list looks like this:

```
YEAR_IN_SCHOOL_CHOICES=[

('FR','Freshman'),

('SO','Sophomore'),

('JR','Junior'),
```

```
('SR','Senior'),
('GR','Graduate'),
]
```

default

The default value for the field. This can be a value or a callable object. If callable it will be called every time a new object is created.

help_text

Extra "help" text to be displayed with the form widget. It's useful for documentation even if your field isn't used on a form.

primary_key

If **True**, this field is the primary key for the model.

If you don't specify **primary_key=True** for any fields in your model, Django will automatically add an **IntegerField** to hold the primary key, so you don't need to set **primary_key=True** on any of your fields unless you want to override the default primary-key behavior. For more, see Automatic primary key fields.

The primary key field is read-only. If you change the value of the primary key on an existing object and then save it, a new object will be created alongside the old one. For example:

```
fromdjango.dbimport models


classFruit(models.Model):
   name = models.CharField(max_length=100, primary_key=True)
>>>fruit = Fruit.objects.create(name='Apple')
>>>fruit.name = 'Pear'
>>>fruit.save()
>>>Fruit.objects.values_list('name', flat=True)
<QuerySet ['Apple', 'Pear']>
```

unique

If **True**, this field must be unique throughout the table.

Table Naming Conventions –

To save you time, Django automatically derives the name of the database table from the name of your model class and the app that contains it. A model's database table name is constructed by joining the model's "app label" – the name you used in manage.py startapp – to the model's class name, with an underscore between them.

- ✓ Use lowercase table names for MariaDB and MySQL: It is strongly advised that you use lowercase table names when you override the table name via db_table, particularly if you are using the MySQL backend. See the MySQL notes for more details.

- ✓ Table name quoting for Oracle: In order to meet the 30-char limitation Oracle has on table names, and match the usual conventions for Oracle databases, Django may shorten table names and turn them all-uppercase. To prevent such transformations, use a quoted name as the value for db_table:

  db_table = '"name_left_in_lowercase"'

  Such quoted names can also be used with Django's other supported database backends; except for Oracle, however, the quotes have no effect. See the Oracle notes for more details.

- ✓ order_with_respect_to implicitly sets the ordering option: Internally, order_with_respect_to adds an additional field/database column named _order and sets the model's ordering option to this field. Consequently, order_with_respect_to and ordering cannot be used together, and the ordering added by order_with_respect_to will apply whenever you obtain a list of objects of this model.

- ✓ Changing order_with_respect_to: Because order_with_respect_to adds a new database column, be sure to make and apply the appropriate migrations if you add or change order_with_respect_to after your initial migrate.

Creating Django Models –

Step 1 — Create Django Application

To keep with the Django philosophy of modularity, we will create a Django app within our project that contains all of the files necessary for creating the blog website.

First activate your Python virtual environment:

- cd ~/my_blog_app
- . env/bin/activate
- cd blog

From there, let's run this command:

- python manage.py startapp blogsite

At this point, you'll have the following directory structure for your project:

```
my_blog_app/
□□□ blog
 ├□□ blog
 □  ├□□ __init__.py
 □  ├□□ __pycache__
 □  □  ├□□ __init__.cpython-35.pyc
 □  □  ├□□ settings.cpython-35.pyc
 □  □  ├□□ urls.cpython-35.pyc
 □  □  □□□ wsgi.cpython-35.pyc
 □  ├□□ settings.py
 □  ├□□ urls.py
 □  □□□ wsgi.py
 ├□□ blogsite
 □  ├□□ admin.py
 □  ├□□ apps.py
 □  ├□□ __init__.py
 □  ├□□ migrations
 □  □  □□□ __init__.py
```

□ ├□□ models.py
□ ├□□ tests.py
□ □□□ views.py
□□□ manage.py

The file we will focus on for this tutorial, will be the models.py file.

Step 2 — Add the Posts Model

First we need to open and edit the models.py file so that it contains the code for generating a Post model. A Post model contains the following database fields:

- title — The title of the blog post.
- slug — Where valid URLs are stored and generated for web pages.
- content — The textual content of the blog post.
- created_on — The date on which the post was created.
- author — The person who has written the post.

Now, change directories to where the models.py file is contained.

- cd ~/my_blog_app/blog/blogsite

Use the cat command to show the contents of the file in your terminal.

- cat models.py

The file should have the following code, which imports models, along with a comment describing what is to be placed into this models.py file.

models.py

```
from django.db import models

# Create your models here.
```

Using your favorite text editor or IDE, add the following code to the models.py file. We'll use nano as our text editor. But, you are welcome to use whatever you prefer.

- nano models.py

Within this file, the code to import the models API is already added, we can go ahead and delete the comment that follows. Then we'll import slugify for generating slugs from strings, and Django's User for authentication like so:

models.py

```
from django.db import models
from django.template.defaultfilters import slugify
from django.contrib.auth.models import User
```

Then, add the class method on the model class we will be calling Post, with the following database fields, title, slug, content, created_on and author.

models.py

```
...
class Post(models.Model):
    title = models.CharField(max_length=255)
    slug = models.SlugField(unique=True, max_length=255)
    content = models.TextField()
    created_on = models.DateTimeField(auto_now_add=True)
    author = models.TextField()
```

Next, we will add functionality for the generation of the URL and the function for saving the post. This is crucial, because this creates a unique link to match our unique post.

models.py

```
...
@models.permalink
def get_absolute_url(self):
return ('blog_post_detail', (),
        {
'slug': self.slug,
        })
def save(self, *args, **kwargs):
if not self.slug:
        self.slug = slugify(self.title)
        super(Post, self).save(*args, **kwargs)
```

Now, we need to tell the model how the posts should be ordered, and displayed on the web page. The logic for this will be added to a nested inner Meta class. The Meta class generally contains other important model logic that isn't related to database field definition.

models.py

```
...
class Meta:
```

```
    ordering = ['created_on']
def __unicode__(self):
return self.title
```

Finally, we will add the Comment model to this file. This involves adding another class named Comment with models.Models in its signature and the following database fields defined:

- name — The name of the person posting the comment.
- email — The email address of the person posting the comment.
- text — The text of the comment itself.
- post — The post with which the comment was made.
- created_on — The time the comment was created.

models.py

```
...
classComment(models.Model):
  name = models.CharField(max_length=42)
  email = models.EmailField(max_length=75)
  website = models.URLField(max_length=200, null=True, blank=True)
  content = models.TextField()
  post = models.ForeignKey(Post, on_delete=models.CASCADE)
  created_on = models.DateTimeField(auto_now_add=True)
```

When finished, your complete models.py file should look like this:

models.py

```
from django.db import models
from django.template.defaultfilters import slugify
from django.contrib.auth.models import User


classPost(models.Model):
  title = models.CharField(max_length=255)
  slug = models.SlugField(unique=True, max_length=255)
  content = models.TextField()
  created_on = models.DateTimeField(auto_now_add=True)
  author = models.TextField()

@models.permalink
defget_absolute_url(self):
```

```
return ('blog_post_detail', (),
            {
'slug': self.slug,
            })

defsave(self, *args, **kwargs):
ifnot self.slug:
          self.slug = slugify(self.title)
       super(Post, self).save(*args, **kwargs)

classMeta:
       ordering = ['created_on']

def__unicode__(self):
return self.title


classComment(models.Model):
   name = models.CharField(max_length=42)
   email = models.EmailField(max_length=75)
   website = models.URLField(max_length=200, null=True, blank=True)
   content = models.TextField()
   post = models.ForeignKey(Post, on_delete=models.CASCADE)
   created_on = models.DateTimeField(auto_now_add=True)
```

Be sure to save and close the file.

With the models.py file set up, we can go on to update our settings.py file.

Step 3 — Update Settings

Now that we've added models to our application, we must inform our project of the existence of the blogsite app that we've just added. We do this by adding it to the INSTALLED_APPS section in settings.py.

Navigate to the directory where your settings.py lives.

- cd ~/my_blog_app/blog/blog

From here, open up your settings.py file, with nano, for instance, using the command nano settings.py.

With the file open, add your blogsite app to the INSTALLED_APPS section of the file, as shown below.

settings.py

```
# Application definition
INSTALLED_APPS = [
'blogsite',
'django.contrib.admin',
'django.contrib.auth',
'django.contrib.contenttypes',
'django.contrib.sessions',
'django.contrib.messages',
'django.contrib.staticfiles',
]
```

With the blogsite app added, you can save and exit the file.

At this point we are ready to move on to apply these changes.

Step 4 — Make Migrations

With our models Post and Comment added, the next step is to apply these changes so that our MySQL database schema recognizes them and creates the necessary tables.

Let's take a look at what tables already exist in our blog_data database.

To do this, we need to log in to MySQL server.

Note: In this example, we'll be using the username root with no password, but you should use the username and password you have set up for MySQL.

- mysql blog_data -u root

You'll notice that if you type into the SHOW DATABASES; command, you'll see the following:

Output

```
+--------------------+
| Database           |
+--------------------+
| information_schema |
| blog_data          |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.00 sec)
```

We will be looking at the blog_data database and view the tables that already exist, if any.

- USE blog_data;

Then, list the tables that exist in the blog_data database:

- SHOW TABLES;

Output

```
Empty set (0.00 sec)
```

Right now it won't show any tables because we haven't made any migrations yet. But, when we do make migrations, it will display the tables that have been generated by Django.

Now we will proceed to make the migrations that apply the changes we've made in models.py.

Close out of MySQL with CTRL + D.

First, we must package up our model changes into individual migration files using the command makemigrations. These files are similar to that of commits in a version control system like git.

Now, if you navigate to ~/my_blog_app/blog/blogsite/migrations and run ls, you'll notice that there is only an __init__.py file. This will change once we add the migrations.

Change to the blog directory using cd, like so:

- cd ~/my_blog_app/blog
- python manage.py makemigrations

You should then see the following output in your terminal window:

Output
Migrations for 'blogsite':
  blogsite/migrations/0001_initial.py
    - Create model Comment
    - Create model Post
    - Add field post to comment

Adding the Django Application to Your Project:

To add a new Django application to an existing project, the steps are –

i. On the main menu, choose Tools | Run manage.py task
ii. In the Enter manage.py task name dialog, type startapp. Note suggestion list that appears under the dialog after entering the first letter, and shrinks as you type to show the exact match only.
iii. In the dialog that opens, type the name of the new Django application.

Validate Django app: -

Once you have added a newly created app for example: 'books' to the INSTALLED_APPS list in the settings.py file, you shouldn't use 'mysite.books' on your installed apps. For validating, simply use 'books' (according to this example).

Type on your command line:

python manage.py makemigrations

python manage.py migrate

It will create your model's tables on your db.

in your admin.py:

```
from django.contrib import admin

from .models import Publisher, Author, Book

admin.site.register(Publisher)
```

admin.site.register(Author)

admin.site.register(Book)

type on command line it will help you to create admin account:

python manage.py createsuperuser

then after filling the superuser info type on command line

python manage.py runserver

finally go on your browser and login: http://127.0.0.1:8000/admin

you will see your models on your admin page

Inserting data to a Django Table:

You need to create an instance of the model class, instantiate it with the appropriate values (name and address) and then call save(), which composes the appropriates SQL INSERT statement under the hood.

How to insert data into a database from an HTML form in Django:

If you use a ModelForm in Django, you wouldn't need to worry about inserting data into the database, because this would be done automatically. This is the case with Django ModelForms.

However, there are times where you may not want to use Django ModelForms and instead just want to code the form directly in HTML and then insert the data that the user has entered into the database. This is what this article addresses. We will show how to insert the data that a user enters into an HTML form into the database.

So let's create a form in which a user creates a post. This post form will simply take in 2 values, the title of the post and the content of the post.

We will then insert this into the database.

So the first thing we have to do is create our database (the model) in the models.py file.

models.py File

So the first thing we have to do is create our database.

We will call our database, Post.It will only have two fields: title and content.

#models.py File

```
from django.db import models
class Post(models.Model):
    title= models.CharField(max_length=300, unique=True)
    content= models.TextField()
```

Okay, this is a very basic database. We simply have a title and a content field.

After this, we save the file and then, within the command line, run the command, py manage.py makemigrations, and then run the command, py manage.py migrate.

createpost.html Template File

Now we'll create our template file. We'll call it, createpost.html

Within this template file, we are going to have the form where a user can submit a post.

It is a simple form that only contains 2 fields: title and content.

This is shown in the code below.

```
<html>
<head>
<title>Create a Post </title>
</head>
<body>
<h1>Create a Post </h1>
<form action="" method="POST">
{% csrf_token %}
Title: <input type="text" name="title"/><br/>
```

```
Content: <br/>
<textarea cols="35" rows="8" name="content">
</textarea><br/>
<input type="submit" value="Post"/>
</form>
</body>
</html>
```

So, this template contains a very basic form that has 2 fields: one which is title and the other which is content. We need a name attribute with each form field because this is how we will extract the data that the user enters into the field.

views.py File

Lastly, we have our views.py file.

In this file, we will take the data that the user has entered into the form fields and insert the data into a database.

The following code in the views.py file does this.

```
from django.shortcuts import render
from .models import Post
def createpost(request):
    if request.method == 'POST':
        if request.POST.get('title') and request.POST.get('content'):
            post=Post()
            post.title= request.POST.get('title')
            post.content= request.POST.get('content')
            post.save()
            return render(request, 'posts/create.html')
```

```
        else:
            return render(request,'posts/create.html')
```

So, this is the heart of our code in which we extract the data from the form fields that the user has entered and insert the data into a database.

Understanding QuerySets –

Digging Into Django QuerySets

Object-relational mappers (or ORMs for short), such as the one that comes built-in with Django, make it easy for even new developers to become productive without needing to have a large body of knowledge about how to make use of relational databases. They abstract away the details of database access, replacing tables with declarative model classes and queries with chains of method calls. Since this is all done in standard Python developers can build on top of it further, adding instance methods to a model to wrap reusable pieces of logic. However, the abstraction provided by ORMs is not perfect. There are pitfalls lurking for unwary developers, such as the N + 1 problem. On the bright side, it is not difficult to explore and gain a better understanding of Django's ORM. Taking the time and effort to do so will help you become a better Django developer.

In this section, you will be setting up a simple example app, consisting of nothing more than a few models, and then making use of the Django shell to perform various queries and examine the results. You don't have to follow along, but it is recommended that you do so.

First, create a clean virtualenv. Here I'll be using Python 3 all of the way, but there should be little difference with Python 2.

```
$ mkvirtualenv -p $(which python3) querysets
Already using interpreter /usr/bin/python3
    Using base prefix '/usr'
New python executable in /home/jrb/.virtualenvs/querysets/bin/python3
Also creating executable in /home/jrb/.virtualenvs/querysets/bin/python
Installing setuptools, pkg_resources, pip, wheel...done.
```

Next, install Django and IPython,

```
(querysets) $ pip install django ipython
```

Create the new project.

```
(querysets) $ django-admin.py startproject querysets
(querysets) $ cd querysets/
(querysets) $ ./manage.py startapp qs
```

Update querysets/settings.py to add 'qs', to the end of the INSTALLED_APPS list. Then, edit qs/models.py to add the simple models we will be dealing with

```
from django.db import models
class OnlyOne(models.Model):
    name = models.CharField(max_length=16)

class MainModel(models.Model):
    name = models.CharField(max_length=16)
    one = models.ForeignKey(OnlyOne)

class RelatedModel(models.Model):
    name = models.CharField(max_length=16)
    main = models.ForeignKey(MainModel, related_name='many')
```

Finally, set up the database.

```
(querysets) jrb@caktus025:~/caktus/querysets$ ./manage.py makemigrations qs
Migrations for 'qs':
  qs/migrations/0001_initial.py:
    - Create model MainModel
    - Create model OnlyOne
    - Create model RelatedModel
    - Add field one to mainmodel
(querysets) jrb@caktus025:~/caktus/querysets$ ./manage.py migrate
```

...

How to Filter QuerySets?

Filtering QuerySets dynamically is a fairly common use case. Sure thing there is a pluggable app to make your life easier. This tutorial is about how to use the django-filter app to add a hassle-free filtering to your views. To illustrate this, an example is implementedwith a view to search for users. As usual the code used in this tutorial is available on GitHub. You can find the link in the end of this post.

Installation: Easiest way is to install it with pip:

```
pip install django-filter
```

That's it. It's ready to be used. Make sure you update your requirements.txt. The default language of the app is English

filters.py

```
fromdjango.contrib.auth.modelsimportUser
importdjango_filters

classUserFilter(django_filters.FilterSet):
classMeta:
model=User
fields=['username','first_name','last_name',]
```

The view is as simple as:

views.py

```
fromdjango.contrib.auth.modelsimportUser
fromdjango.shortcutsimportrender
from.filtersimportUserFilter

defsearch(request):
user_list=User.objects.all()
user_filter=UserFilter(request.GET,queryset=user_list)
returnrender(request,'search/user_list.html',{'filter':user_filter})
```

Then a route:

urls.py

```
fromdjango.conf.urlsimporturl
frommysite.searchimportviews

urlpatterns=[
url(r'^search/$',views.search,name='search'),
]
```

And finally the template:

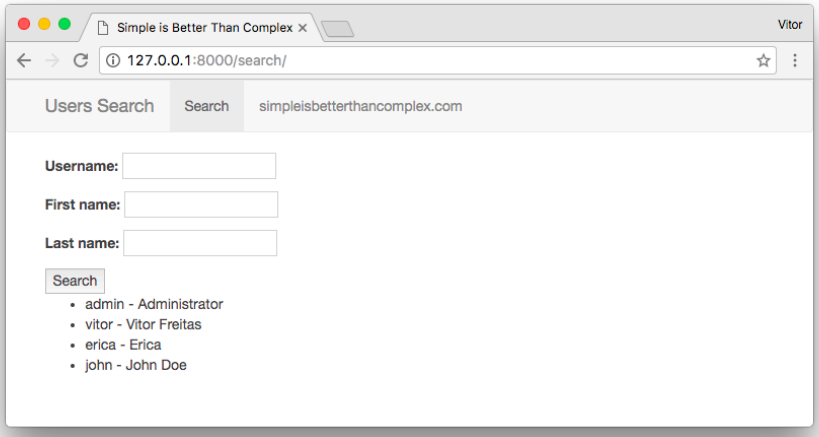
user_list.html

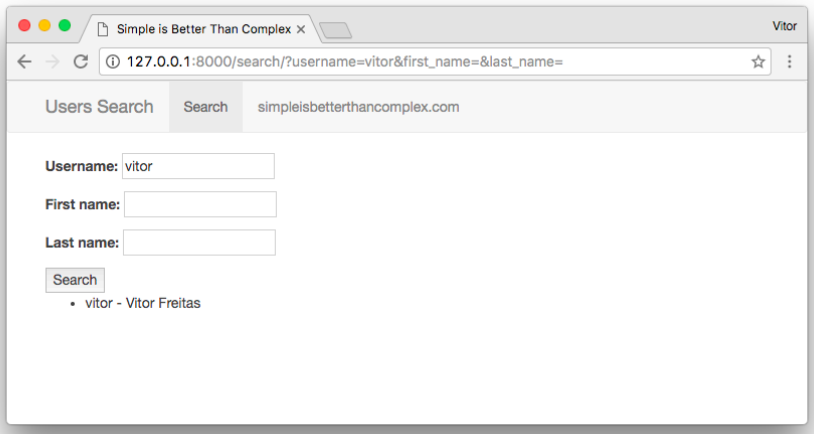
```
{%extends'base.html'%}

{%blockcontent%}
<formmethod="get">
{{filter.form.as_p}}
<buttontype="submit">Search</button>
</form>
<ul>
{%foruserinfilter.qs%}
<li>{{user.username}} - {{user.get_full_name}}</li>
{%endfor%}
</ul>
{%endblock%}
```

The magic happens inside the `UserFilter` class. We simply have to pass the `request.GET` data to the `UserFilter` class, along with the QuerySet we want to filter. It will generate a Django Form with the search fields as well as return the filtered QuerySet.

So basically we will be working inside the UserFilter definition and the HTML template, displaying properly the data.

Output:





Django Lookups: Django offers a wide variety of built-in lookups for filtering (for example, exact and icontains). Lookups define the corresponding ajax views used by the auto-completion fields and widgets. They take in the current request and return the JSON needed by the jQuery auto-complete plugin.

A lookup example: Let's start with a small custom lookup. We will write a custom lookup ne which works opposite to exact. Author.objects.filter(name__ne='Jack') will translate to the SQL:

"author"."name" <> 'Jack'

This SQL is backend independent, so we don't need to worry about different databases.

There are two steps to making this work. Firstly we need to implement the lookup, then we need to tell Django about it:

```
from django.db.models import Lookup

class NotEqual(Lookup):
    lookup_name = 'ne'

    def as_sql(self, compiler, connection):
        lhs, lhs_params = self.process_lhs(compiler, connection)
        rhs, rhs_params = self.process_rhs(compiler, connection)
        params = lhs_params + rhs_params
        return '%s <> %s' % (lhs, rhs), params
```

Types of Lookups –

Defining a Lookup

django-selectable uses a registration pattern similar to the Django admin. Lookups should be defined in a *lookups.py* in your application's module. Once defined you must register in with django-selectable. All lookups must extend from `selectable.base.LookupBase` which defines the API for every lookup.

```
fromselectable.baseimportLookupBase
fromselectable.registryimportregistry

classMyLookup(LookupBase):
defget_query(self,request,term):
data=['Foo','Bar']
return[xforxindataifx.startswith(term)]

registry.register(MyLookup)
```

Lookup API

LookupBase.get_query(*request, term*)

This is the main method which takes the current request from the user and returns the data which matches their search.

Parameters:
- **request** – The current request object.
- **term** – The search term from the widget input.

Returns: An iterable set of data of items matching the search term.

LookupBase.get_item_label(*item*)

This is first of three formatting methods. The label is shown in the drop down menu of search results. This defaults to `item.__unicode__`.

Parameters: **item** – An item from the search results.

Returns: A string representation of the item to be shown in the search results. The label can include HTML. For changing the label format on the client side see Advanced Label Formats.

LookupBase.get_item_id(*item*)

This is second of three formatting methods. The id is the value that will eventually be returned by the field/widget. This defaults to `item.__unicode__`.

Parameters: **item** – An item from the search results.

Returns: A string representation of the item to be returned by the field/widget.

LookupBase.split_term(*term*)

Split searching term into array of subterms that will be searched separately. You can override this function to achieve different splitting of the term.

Parameters: **term** – The search term.

Returns: Array with subterms

LookupBase.get_item_value(*item*)

This is last of three formatting methods. The value is shown in the input once the item has been selected. This defaults to `item.__unicode__`.

Parameters: **item** – An item from the search results.

Returns: A string representation of the item to be shown in the input.

LookupBase.get_item(*value*)

`get_item` is the reverse of `get_item_id`. This should take the value from the form initial values and return the current item. This defaults to simply return the value.

Parameters: **value** – Value from the form inital value.

Returns: The item corresponding to the initial value.

LookupBase.create_item(*value*)

If you plan to use a lookup with a field or widget which allows the user to input new values then you must define what it means to create a new item for your lookup. By default this raises a `NotImplemented` error.

Parameters: **value** – The user given value.

Returns: The new item created from the item.

LookupBase.format_item(*item*)

By default `format_item` creates a dictionary with the three keys used by the UI plugin: id, value, label. These are generated from the calls

to `get_item_id`, `get_item_value` and `get_item_label`. If you want to add additional keys you should add them here.

The results of `get_item_label` is conditionally escaped to prevent Cross Site Scripting (XSS) similar to the templating language. If you know that the content is safe and you want to use these methods to include HTML should mark the content as safe with `django.utils.safestring.mark_safe` inside the `get_item_label` method.

`get_item_id` and `get_item_value` are not escapted by default. These are not a XSS vector with the built-in JS. If you are doing additional formating using these values you should be conscience of this fake and be sure to escape these values.

Parameters: `item` – An item from the search results.

Returns: A dictionary of information for this item to be sent back to the client.

There are also some additional methods that you could want to use/override. These are for more advanced use cases such as using the lookups with JS libraries other than jQuery UI. Most users will not need to override these methods.

LookupBase.format_results(*self*, *raw_data*, *options*)

Returns a python structure that later gets serialized. This makes a call to paginate_results prior to calling format_item on each item in the current page.

Parameters:

- `raw_data` – The set of all matched results.
- `options` – Dictionary of `cleaned_data` from the lookup form class.

Returns: A dictionary with two keys `meta` and `data`. The value of `data` is an iterable extracted from page_data. The value of `meta` is a dictionary. This is a copy of options with one additional element `more` which is a translatable “Show more” string (useful for indicating more results on the javascript side).

LookupBase.paginate_results(*results*, *options*)

If SELECTABLE_MAX_LIMIT is defined or `limit` is passed in request.GET then `paginate_results` will return the current page using Django's built in pagination. See the Django docs on pagination for more info.

Parameters:
- **results** – The set of all matched results.
- **options** – Dictionary of `cleaned_data` from the lookup form class.

Returns: The current Page object of results.

Lookups Based on Models

Perhaps the most common use case is to define a lookup based on a given Django model. For this you can extend `selectable.base.ModelLookup`. To extend `ModelLookup` you should set two class attributes: `model` and `search_fields`.

```
from__future__importunicode_literals

fromselectable.baseimportModelLookup
fromselectable.registryimportregistry

from.modelsimportFruit


classFruitLookup(ModelLookup):
model=Fruit
search_fields=('name__icontains',)

registry.register(FruitLookup)
```

The syntax for `search_fields` is the same as the Django field lookup syntax. Each of these lookups are combined as OR so any one of them matching will return a result. You may optionally define a third class attribute `filters` which is a dictionary of filters to be applied to the model queryset. The keys should be a string defining a field lookup and the value should be the value for the field lookup. Filters on the other hand are combined with AND.

Slicing a Quertset:

Slicing: A QuerySet can be sliced, using Python's array-slicing syntax. Slicing an unevaluated QuerySet usually returns another unevaluated QuerySet, but Django will execute the database query if you use the "step" parameter of slice syntax, and will return a list. Slicing a QuerySet that has been evaluated also returns a list.

Also note that even though slicing an unevaluated QuerySet returns another unevaluated QuerySet, modifying it further (e.g., adding more filters, or modifying ordering) is not allowed, since that does not translate well into SQL and it would not have a clear meaning either.

Ordering Querysets –

order_by(*fields): By default, results returned by a QuerySet are ordered by the ordering tuple given by the ordering option in the model's Meta. You can override this on a per-QuerySet basis by using the order_by method.

Example:

```
ordered_authors = Author.objects.order_by('-score', 'last_name')[:30]
```

Methods That Return QuerySets

Method	Description
filter()	Filter by the given lookup parameters. Multiple parameters are joined by SQL AND statements
exclude()	Filter by objects that don't match the given lookup parameters
annotate()	Annotate each object in the QuerySet. Annotations can be simple values, a field reference or an aggregate expression
order_by()	Change the default ordering of the QuerySet

Method	Description
reverse()	Reverse the default ordering of the QuerySet
distinct()	Perform an SQL SELECT DISTINCT query to eliminate duplicate rows
values()	Returns dictionaries instead of model instances
values_list()	Returns tuples instead of model instances
dates()	Returns a QuerySet containing all available dates in the specified date range
datetimes()	Returns a QuerySet containing all available dates in the specified date and time range
none()	Create an empty QuerySet
all()	Return a copy of the current QuerySet
union()	Use the SQL UNION operator to combine two or more QuerySets
intersection()	Use the SQL INTERSECT operator to return the shared elements of two or more QuerySets
difference()	Use the SQL EXCEPT operator to return elements in the first QuerySet that are not in the others
select_related()	Select all related data when executing the query (except many-to-many relationships)
prefetch_related()	Select all related data when executing the query (including many-to-many relationships)

Method	Description
defer()	Do not retrieve the names fields from the database. Used to improve query performance on complex datasets
only()	Opposite of defer() : return only the name fields
using()	Select which database the QuerySet will be evaluated against (when using multiple databases)
select_for_update()	Return a QuerySet that will lock rows until the end of the transaction
raw()	Execute a raw SQL statement
AND (&)	Combine two QuerySets with the SQL AND operator. Using AND (&) is functionally equivalent to using filter() with multiple parameters
OR (|)	Combine two QuerySets with the SQL OR operator

Deleting in Django –

Deleting objects: To delete a single object, we need to write the following commands:

```
>>> a = Album.objects.get(pk = 2)
>>> a.delete()
>>> Album.objects.all()
<QuerySet [<Album: Divide>, <Album: Revolver>]>
```

To delete multiple objects, we can use filter() or exclude() functions as follows:

```
>>> Album.objects.filter(genre = "Pop").delete()
>>> Album.objects.all()
```

<QuerySet []>

Retrieving related records in Django –

Let us add 2 more Albums records for the sake of demonstration.

```
>>> a = Album(title = "Abbey Road", artist = "The Beatles", genre = "Rock")
>>> a.save()
>>> a = Album(title = "Revolver", artist = "The Beatles", genre = "Rock")
>>> a.save()
```

To retrieve all the objects of a model, we write the following command:

```
>>> Album.objects.all()
<QuerySet [<Album: Divide>, <Album: Abbey Road>, <Album: Revolver>]>
```

The output is a QuerySet, or a set of objects that match the query. Notice that the name printed is the output of the __str__() function.

We can also filter queries using the functions filter(), exclude() and get(). The filter() function returns a QuerySet having objects that match the given lookup parameters.

```
>>> Album.objects.filter(artist = "The Beatles")
<QuerySet [<Album: Abbey Road>, <Album: Revolver>]>
```

The exclude() function returns a QuerySet having objects other than those matching the given lookup parameters.

```
>>> Album.objects.exclude(genre = "Rock")
<QuerySet [<Album: Divide>]>
```

The get() function returns a single object which matches the given lookup parameter. It gives an error when the query returns multiple objects.

```
>>> Album.objects.get(pk = 3)
<QuerySet [<Album: Revolver>]>
```

Django Q objects:

Q object encapsulates a SQL expression in a Python object that can be used in database-related operations. Using Q objects we can make complex queries with less and simple code.

For example, this Q object filters whether the question starts wiht 'what':

```
from django.db.models import Q
Q(question__startswith='What')
```

Q objects are helpfull for complex queries because they can be combined using logical operators and(&), or(|), negation(~)

For example, this statement returns if the question starts with 'who' or with 'what'.

```
Q(question__startswith='Who')| Q(question__startswith='What')
```

Note:>If the operator is not included then by default 'AND' operator is used

The following code is source of Q class:

```
class Q(tree.Node):
  AND ='AND'
  OR ='OR'
  default= AND
```

```
def __init__(self,*args,**kwargs):

  super(Q,self).__init__(children=list(args)+ list(six.iteritems(kwargs)))



def _combine(self, other, conn):
  ifnot isinstance(other, Q):
    raiseTypeError(other)
  obj = type(self)()
  obj.connector = conn
  obj.add(self, conn)
  obj.add(other, conn)
  return obj

def __or__(self, other):
  returnself._combine(other,self.OR)

def __and__(self, other):
  returnself._combine(other,self.AND)

def __invert__(self):
  obj = type(self)()
  obj.add(self,self.AND)
  obj.negate()
  return obj
```

As you can interpret from above code, we have three operators 'or', 'and' and invert(negation)

and the default operator is AND.

Dynamic querying with Q objects:

This is an interesting feature as we can use the operator module to create dynamic queries.

```
importoperator

from django.db.models import Q

from your_app.models import your_model_object
```

```
q_list =[Q(question__startswith='Who'), Q(question__startswith='What')]

your_model_object.objects.filter(reduce(operator.or_, q_list))
```

Creating forms from models: -

ModelForm

class ModelForm

If you're building a database-driven app, chances are you'll have forms that map closely to Django models. For instance, you might have a BlogComment model, and you want to create a form that lets people submit comments. In this case, it would be redundant to define the field types in your form, because you've already defined the fields in your model.

For this reason, Django provides a helper class that lets you create a Form class from a Django model.

For example:

```
>>> from django.forms import ModelForm
>>> from myapp.models import Article


# Create the form class.
>>> class ArticleForm(ModelForm):
...     class Meta:
...         model = Article
...         fields = ['pub_date', 'headline', 'content', 'reporter']


# Creating a form to add an article.
>>> form = ArticleForm()
```

```
# Creating a form to change an existing article.

>>> article = Article.objects.get(pk=1)

>>> form = ArticleForm(instance=article)
```

Cookies in Django:

Sometimes you might want to store some data on a per-site-visitor basis as per the requirements of your web application. Always keep in mind, that cookies are saved on the client side and depending on your client browser security level, setting cookies can at times work and at times might not.

To illustrate cookies handling in Django, let's create a system using the login system we created before. The system will keep you logged in for X minute of time, and beyond that time, you will be out of the app.

For this, you will need to set up two cookies, last_connection and username.

At first, let's change our login view to store our username and last_connection cookies –

```
from django.templateimportRequestContext

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

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

ifMyLoginForm.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
```

As seen in the view above, setting cookie is done by the **set_cookie** method called on the response not the request, and also note that all cookies values are returned as string.

Let's now create a formView for the login form, where we won't display the form if cookie is set and is not older than 10 second –

```
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',{})
```

As you can see in the formView above accessing the cookie you set, is done via the COOKIES attribute (dict) of the request.

Now let's change the url.py file to change the URL so it pairs with our new view –

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

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

What are Sessions?

After observing these problems of cookies, the web-developers came with a new and more secure concept, Sessions. The session is a semi-permanent and two-way communication between the server and the browser.

Sessions framework can be used to provide persistent behaviour for anonymous users in the website.Sessions are the mechanism used by Django for you store and retrieve data on a per-site-visitor basis.Django uses a cookie containing a special session id.

To enable the session in the django, you will need to make sure of two things in settings.py:

MIDDLEWARE_CLASSES has ‘django.contrib.sessions.middleware.SessionMiddleware’ activated

INSTALLED_APPS has ‘django.contrib.sessions’ added.

```
# Application definition
INSTALLED APPS = [
'dhun',
'django.contrib.admin',
'django.contrib.auth',
'django.contrib.contenttypes',
'django.contrib.sessions',
'django.contrib.messages',
'django.contrib.staticfiles',
]
MIDDLEWARE = [
'django.middleware.securitY.SecuritYMiddleware',
'django.contrib.sessions.middleware.SessionMiddLeware',
'django.middleware.common.CommonMiddleware',
'django.middleware.csrf.CsrfViewMidd1eware',
'django.contrib.auth.middleware.AuthenticationMiddleware',
'django.contrib.messages.middleware.MessageMiddleware',
'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

After enabling the session, the session database table has to create and to do this run the following command:

```
python manage.py syncdb
```

After running previous command and if it didn't find any errors then later run the command given below to finally reflect the changes saved onto the migration file onto the database.

python manage.py migrate

Now once sessions are created, then testing of the cookies has to be done. In views.py, set the test cookie in the index view, and test the cookie in your about view.

Handling Sessions in Django –

Django provides full support for anonymous sessions. The session framework lets you store and retrieve arbitrary data on a per-site-visitor basis. It stores data on the server side and abstracts the sending and receiving of cookies. Cookies contain a session ID – not the data itself (unless you're using the cookie based backend).

Enabling sessions: Sessions are implemented via a piece of middleware.

To enable session functionality, do the following:

Edit the MIDDLEWARE setting and make sure it contains 'django.contrib.sessions.middleware.SessionMiddleware'. The default settings.py created by django-admin startproject has SessionMiddleware activated.

If you don't want to use sessions, you might as well remove the SessionMiddleware line from MIDDLEWARE and 'django.contrib.sessions' from your INSTALLED_APPS. It'll save you a small bit of overhead.

Configuring the session engine: By default, Django stores sessions in your database (using the model django.contrib.sessions.models.Session). Though this is convenient, in some setups it's faster to store session data elsewhere, so Django can be configured to store session data on your filesystem or in your cache.

Using database-backed sessions:

If you want to use a database-backed session, you need to add 'django.contrib.sessions' to your INSTALLED_APPS setting. Once you have configured your installation, run manage.py migrate to install the single database table that stores session data.

User authentication in Django:Django comes with a user authentication system. It handles user accounts, groups, permissions and cookie-based user sessions.

The Django authentication system handles both authentication and authorization. Briefly, authentication verifies a user is who they claim to be, and authorization determines what an authenticated user is allowed to do. Here the term authentication is used to refer to both tasks.

The auth system consists of:

Users: a valid user.

Permissions: Binary (yes/no) flags designating whether a user may perform a certain task.

Groups: A generic way of applying labels and permissions to more than one user.

A configurable password hashing system

Forms and view tools for logging in users, or restricting content

A pluggable backend system

The authentication system in Django aims to be very generic and doesn't provide some features commonly found in web authentication systems. Solutions for some of these common problems have been implemented in third-party packages:

Password strength checking

Throttling of login attempts

Authentication against third-parties (OAuth, for example)

Object-level permissions.

Installation

Authentication support is bundled as a Django contrib module in django.contrib.auth. By default, the required configuration is already included in the settings.py generated by django-admin startproject, these consist of two items listed in your INSTALLED_APPS setting:

- ✓ 'django.contrib.auth' contains the core of the authentication framework, and its default models.
- ✓ 'django.contrib.contenttypes' is the Django content type system, which allows permissions to be associated with models you create.

and these items in your MIDDLEWARE setting:

- ✓ SessionMiddleware manages sessions across requests.
- ✓ AuthenticationMiddleware associates users with requests using sessions.

With these settings in place, running the command manage.py migrate creates the necessary database tables for auth related models and permissions for any models defined in your installed apps.

Django Login and Logouts –

In case you are starting a new project just to follow this, and create a user using the command line just so we can test the login and logout pages.

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

Configure the URL routes

First import the `django.contrib.auth.views` module and add a URL route for the login and logout views:

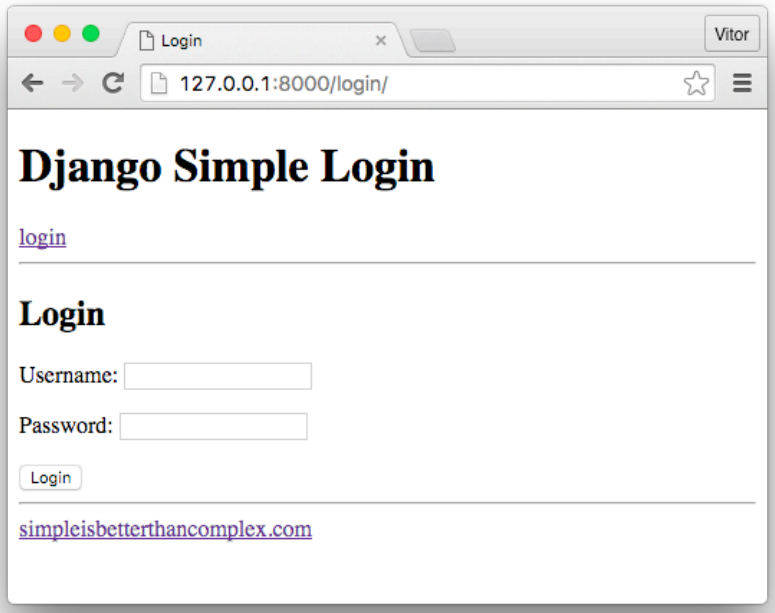
```
fromdjango.conf.urlsimporturl
fromdjango.contribimportadmin
fromdjango.contrib.authimportviewsasauth_views

urlpatterns=[
url(r'^login/$',auth_views.login,name='login'),
url(r'^logout/$',auth_views.logout,name='logout'),
url(r'^admin/',admin.site.urls),
]
```

Building your ownLogin template

By default, the `django.contrib.auth.views.login` view will try to render the `registration/login.html` template. So the basic configuration would be creating a folder named `registration` and place a `login.html` template inside.

Following a minimal login template:

```
{% extends 'base.html' %}

{% block title %}Login{% endblock %}

{% block content %}
<h2>Login</h2>
<formmethod="post">
  {% csrf_token %}
  {{ form.as_p }}
<buttontype="submit">Login</button>
</form>
{% endblock %}
```



Setting up your Logout view –

After acessing the `django.contrib.auth.views.logout` view, Django will render the `registration/logged_out.html` template. In a similar way as we did in the `login` view, you can pass a different template like so:

```
url(r'^logout/$',auth_views.logout,{'template_name':'logged_out.html'},name='logout'),
```

Usually I prefer to use the `next_page` parameter and redirect either to the homepage of my project or to the login page when it makes sense.

```
url(r'^logout/$',auth_views.logout,{'next_page':'/'},name='logout'),
```

Authentication decorator –

Here is the source program of how to implement authentication decorator:

```
from functools import wraps

from urllib.parse import urlparse
```

```
from django.conf import settings
from django.contrib.auth import REDIRECT_FIELD_NAME
from django.core.exceptions import PermissionDenied
from django.shortcuts import resolve_url
[docs]def user_passes_test(test_func, login_url=None,
redirect_field_name=REDIRECT_FIELD_NAME):
    """
    Decorator for views that checks that the user passes the given test,
    redirecting to the log-in page if necessary. The test should be a callable
    that takes the user object and returns True if the user passes.
    """
    def decorator(view_func):
        @wraps(view_func)
        def _wrapped_view(request, *args, **kwargs):
            if test_func(request.user):
                return view_func(request, *args, **kwargs)
            path = request.build_absolute_uri()
            resolved_login_url = resolve_url(login_url or settings.LOGIN_URL)
            # If the login url is the same scheme and net location then just
            # use the path as the "next" url.
            login_scheme, login_netloc = urlparse(resolved_login_url)[:2]
            current_scheme, current_netloc = urlparse(path)[:2]
            if ((not login_scheme or login_scheme == current_scheme) and
                    (not login_netloc or login_netloc == current_netloc)):
                path = request.get_full_path()
            from django.contrib.auth.views import redirect_to_login
```

```
            return redirect_to_login(
                path, resolved_login_url, redirect_field_name)
        return _wrapped_view
    return decorator


[docs]def login_required(function=None,
redirect_field_name=REDIRECT_FIELD_NAME, login_url=None):
    """
    Decorator for views that checks that the user is logged in, redirecting
    to the log-in page if necessary.
    """
    actual_decorator = user_passes_test(
        lambda u: u.is_authenticated,
        login_url=login_url,
        redirect_field_name=redirect_field_name
    )
    if function:
        return actual_decorator(function)
    return actual_decorator



[docs]def permission_required(perm, login_url=None, raise_exception=False):
    """
    Decorator for views that checks whether a user has a particular permission
    enabled, redirecting to the log-in page if necessary.
    If the raise_exception parameter is given the PermissionDenied exception
```

```
    is raised.
    """
    def check_perms(user):
        if isinstance(perm, str):
            perms = (perm,)
        else:
            perms = perm
        # First check if the user has the permission (even anon users)
        if user.has_perms(perms):
            return True
        # In case the 403 handler should be called raise the exception
        if raise_exception:
            raise PermissionDenied
        # As the last resort, show the login form
        return False
    return user_passes_test(check_perms, login_url=login_url)
```

Asynchronous Messaging –

To send asynchronous messages to users (eg from offline scripts). Useful for integration with Celery. To install this package, run the command:

```
install -c auto django-async-messages
```

Once you install the package, you can use the example to implement the code –

```
from django.contrib import messages
from async_messages import get_messages
class AsyncMiddleware(object):
```

```
    def process_response(self, request, response):
        """
        Check for messages for this user and, if it exists,
        call the messages API with it
        """
        if hasattr(request, "session") and hasattr(request, "user") and
request.user.is_authenticated():
            msgs = get_messages(request.user)
            if msgs:
                for msg, level in msgs:
                    messages.add_message(request, level, msg)
        return response
```

Managing Permissions:

There are many ways to handle permissions in a project. For instance we may have model level permissions, object level permissions, fine grained user permission or role based. Either way we don't need to be writing any of those from scratch, Django ecosystem has a vast amount of permission handling apps that will help us with the task.

1. Native Django permissions

As we would expect, Django has a native permission system that works very well in many situations and it's great if final users are making intense use of the Django admin interface. It is a model level permission system that can be applied to either single users or a group. If you have `django.contrib.auth` in your `INSTALLED_APPS` django will automatically create `add`, `change`and `delete` permission types to every model in your system (or any one you add later). Giving or revoking this permissions to users will limit the actions they can take when they log in to the admin. Instead of managing user by user, you can also create a group, add users to it and manage the permissions for the whole group.

But this is not limited to the admin. You can verify user permissions in your own views using the `has_perm` method provided in the user model.

```
if user.has_perm('foo.add_bar'):

return HttpResponse("You are authorized to add content!")

else:

return HttpResponse("Permission to add denied")
```

you can also check user permissions in your templates, using the perms variable that is automatically added to the template context:

```
{% if perms.app_label.can_do_something %}

    This content will be shown users with can_do_something permission.

{% endif %}

This content will be shown to all users.
```

You can also create your own permissions to the models:

```
class Content(models.Model):

owner=models.ForeignKey(User)

content=models.TextField()



class Meta:

permissions=(

('view_content','View content'),
```

```
)
```

Check the official documentation for more information on how to create new permissions, programmatically managing user permissions and working with groups.

2. *Django guardian*

Django Guardian is an third party app focused on handling object level permissions. While the native Django permissions manages the access to all objects in a model, with guardian we can control the access to a particular instance of a model. A good thing is that it has a similar interface to Django's:

```
>>>fromdjango.contrib.auth.modelsimportUser

>>>fromcontents.modelimportContent

>>>fromguardian.shortcutsimportassign_perm

>>>

>>>writer=User.objects.create(username='The Writer')

>>>joe=User.objects.create(username='joe')

>>>content=Content.objects.create(

owner=writer,content='This is the content!')

>>>joe.has_perm('view_content',content)

False

>>>assign_perm('view_content',joe,content)

>>>joe.has_perm('view_content',content)

True
```

You can do the same to groups:

```
>>>fromdjango.contrib.auth.modelsimportGroup

>>>group=Group.objects.create(name='reviewers')

>>>assign_perm('view_content',group,content)

>>>joe.groups.add(group)

>>>joe.has_perm('view_content',content)

True
```

It also has some very useful methods like get_perms and get_objects_for_user, decorators for your views and template tags.

3. *Django role permissions*

Django role permissions uses a more high level approach to manage access to views and objects. Instead of saying individually which user has access to which object, we will be defining roles with a default set of permissions defined.

```
fromrolepermissions.rolesimportAbstractUserRole



classWriter(AbstractUserRole):

available_permissions={

'create_content':True,

'view_content':True,

}
```

```
classReviewer(AbstractUserRole):

available_permissions={

'create_content':False,

'view_content':True,

}
```

Once we give a role to a user we will be able to make verifications about his permissions and also about his role.

```
>>>fromrolepermissions.verificationsimporthas_permission,has_role

>>>has_permission(joe,'view_content')

False

>>>Reviewer.assign_role_to_user(joe)

>>>has_permission(joe,'view_content')

True

>>>has_role(joe,'reviewer')

True
```

It also allows you to define object level permissions using programming logic:

```
fromrolepermissions.permissionsimportregister_object_checker
```

```
@register_object_checker()

defpublish_content(role,user,obj):

ifobj.owner==user:

returnTrue

returnFalse
```

and verify:

```
>>>fromrolepermissions.verificationsimporthas_object_permission

>>>has_object_permission('publish_content',writer,content)
```

GENERIC VIEWS –

Simple Generic Views –

In some cases, writing views, as we have seen earlier is really heavy. Imagine you need a static page or a listing page. Django offers an easy way to set those simple views that is called generic views.

Unlike classic views, generic views are classes not functions. Django offers a set of classes for generic views in django.views.generic, and every generic view is one of those classes or a class that inherits from one of them.

```
>>> 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']
```

This you can use for your generic view. Let's look at some example to see how it works.

Static Pages: Let's publish a static page from the “static.html” template.

Our static.html –

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

```
</html>
```

If we did that the way we learned before, we would have to change the myapp/views.py to be –

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

and myapp/urls.py to be –

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

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

The best way is to use generic views. For that, our myapp/views.py will become –

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

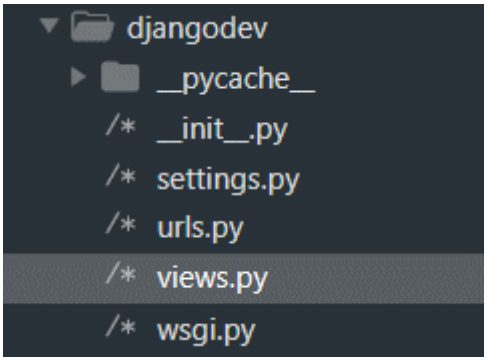
And our myapp/urls.py we will be –

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

Redirect Page in Django –

Django has a specially made redirects function, just for this sole purpose.As we have learned in previous tutorials, every request has a response. Requests are always Http Requests, but there can be different Responses like we returned HttpResponse in our first views.py.

We will now return, an HttpResponseRedirect instance or class object to redirect the User. You need to write redirect in place of render in the views file. Let's edit views.py file.



Here you can see, we created views.py file in our project's main app, where our settings.py and main urls-config are saved.

If you don't have that file you will have to create it. It will be easy for you to follow-up.

Now open views.py file. We will make a new function which will redirect the view to a different URL, which we have already added in the urls.py.

Inside the views.py, enter;

Code:

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

This is the previous function:

```
def index(request):
return HttpResponse("<h1>Data Flair Django</h1>Hello, you just configured your First URL")
```

This is the Redirect Function:

```
def data_flair(request):
return redirect('/dataflair')
```

Your file should look like this.

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

def index(request):
    return HttpResponse("<h1>Data Flair Django</h1>Hello, you just configured your First URL")


def data_flair(request):
    return redirect('/dataflair')
```

Now, open urls.py file

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

```
from django.urls import path, include
from .views import *

urlpatterns = [
path('admin/', admin.site.urls),
path('redirect/', data_flair),
path('dataflair/', index),
]
```

The redirect()

This function is capable of taking more than just URLs, you can redirect your users to models, redirect to a particular URL, while also passing the key values and other needed information. But, function-based redirects are most easy to use and are quite sufficient.

Generic Views Editing –

The following views are described on this page and provide a foundation for editing content:

a) django.views.generic.edit.CreateView:
Example myapp/views.py:

```
from django.views.generic.edit import CreateView

from myapp.models import Author


class AuthorCreate(CreateView):
    model = Author
    fields = ['name']
```

Example myapp/author_form.html:

```
<form method="post">{% csrf_token %}
    {{ form.as_p }}
```

<input type="submit" value="Save">

</form>

b) django.views.generic.edit.UpdateView:

Example myapp/views.py:

```
from django.views.generic.edit import UpdateView
from myapp.models import Author

class AuthorUpdate(UpdateView):
    model = Author
    fields = ['name']
    template_name_suffix = '_update_form'
```

Example myapp/author_update_form.html:

```
<form method="post">{% csrf_token %}
    {{ form.as_p }}
<input type="submit" value="Update">
</form>
```

c) django.views.generic.edit.DeleteView

Example myapp/views.py:

```
from django.urls import reverse_lazy
from django.views.generic.edit import DeleteView
from myapp.models import Author

class AuthorDelete(DeleteView):
```

```
    model = Author

    success_url = reverse_lazy('author-list')
```

Example myapp/author_confirm_delete.html:

```
<form method="post">{% csrf_token %}

<p>Are you sure you want to delete "{{ object }}"?</p>

<input type="submit" value="Confirm">

</form>
```

Data Caching for Performance:–

To cache something is to save the result of an expensive calculation, so that you don't perform it the next time you need it. Following is a pseudo code that explains how caching works –

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

if the page isin the cache:
return the cached page
else:
  generate the page
  save the generated page in the cache (fornext time)
return the generated page
```

Django comes with its own caching system that lets you save your dynamic pages, to avoid calculating them again when needed. The good point in Django Cache framework is that you can cache –

- The output of a specific view.
- A part of a template.
- Your entire site.

To use cache in Django, first thing to do is to set up where the cache will stay. The cache framework offers different possibilities - cache can be saved in database, on file system or directly in memory. Setting is done in the **settings.py** file of your project.

Setting Up Cache in Database

Just add the following in the project settings.py file –

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

Setting Up Cache in File System

Just add the following in the project settings.py file –

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

Setting Up Cache in Memory

This is the most efficient way of caching, to use it you can use one of the following options depending on the Python binding library you choose for the memory cache –

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

Or

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

Caching a View

If you don't want to cache the entire site you can cache a specific view. This is done by using the **cache_page** decorator that comes with Django. Let us say we want to cache the result of the **viewArticles** view –

```
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)
returnHttpResponse(text)
```

As you can see **cache_page** takes the number of seconds you want the view result to be cached as parameter. In our example above, the result will be cached for 15 minutes.

Note – As we have seen before the above view was map to –

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