**Django Cheat Sheet**



**Django Cheat Sheet** tries to provide a basic reference for beginner and advanced developers, lower the entry barrier for newcomers, and help veterans refresh the old tricks. Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design.

[**Django: Create Project**](https://simplecheatsheet.com/django-create-project/)

To start a project we’ll create a new project, create a database, and start a development server.

**Create a new project**

django-admin.py startproject learning\_log .

**Create a database**

python manage.py migrate

**View the project.**

After issuing this command, you can view the project at http://localhost:8000/.

python manage.py runserver

**Create a new app**

A Django project is made up of one or more apps.

python manage.py startapp learning\_logs

[**Django: Template Inheritance Cheat Sheet**](https://simplecheatsheet.com/django-template-inheritance/)

Many elements of a web page are repeated on every page in the site, or every page in a section of the site. By writing one parent template for the site, and one for each section, you can easily modify the look and feel of your entire site

**The parent template**

The parent template defines the elements common to a set of pages, and defines blocks that will be filled by individual pages.

<p>

<a href="{% url 'learning\_logs:index' %}">

Learning Log

</a>

</p>

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

**The child template**

The child template uses the {% extends %} template tag to pull in the structure of the parent template. It then defines the content for any blocks defined in the parent template.

{% extends 'learning\_logs/base.html' %}

{% block content %}

<p>

Learning Log helps you keep track

of your learning, for any topic you're

learning about.

</p>

{% endblock content %}

[**Install Django Rest Framework**](https://simplecheatsheet.com/install-django-rest-framework/)

Install the package via pip:

$ pip install djangorestframework

Then, add 'rest\_framework' to INSTALLED\_APPS in your settings.py file.

INSTALLED\_APPS = [

# Rest of your installed apps ...

'rest\_framework',

]

[**Django: Shell Cheat Sheet**](https://simplecheatsheet.com/using-django-shell/)

Start a shell session

python manage.py shell

Access data from the project

from learning\_logs.models import Topic

Topic.objects.all()

topic = Topic.objects.get(id=1)

topic.text 'Chess'

[**Django: Rest Framework Serialization Cheat Sheet**](https://simplecheatsheet.com/django-rest-framework-serialization/)

Serializers allow complex data like querysets and model instances to be converted to native Python datatypes that can then be easily rendered into JSON, XML, and other formats.

**Using ModelSerializer class:**

Suppose we wanted to create a PostSerializer for our example Post model and CommentSerializer for our Comment model.

class PostSerializer(serializers.ModelSerializer):

class Meta:

model = Post

fields = ('id', 'title', 'text', 'created')

class CommentSerializer(serializers.ModelSerializer):

class Meta:

model = Comment

fields = ('post', 'user', 'text')

Or also you could use exclude to exclude certain fields from being seialized. ModelSerializer has default implementations for the create() and update() methods.

**Nested Serialization**

By default, instances are serialized with primary keys to represent relationships. To get nested serialization we could use, *General* or *Explicit* methods.

**General**

Using depth parameter.

class CommentSerializer(serializers.ModelSerializer):

class Meta:

model = Comment

fields = '\_\_all\_\_'

depth = 2

**Explicit**

Yuo can also define and nest serializers within eachother…

class CommentSerializer(serializers.ModelSerializer):

post = PostSerializer()

class Meta:

model = Comment

fields = '\_\_all\_\_'

So here, the comment’s post field (how we named it in models.py) will serialize however we defined it in PostSerializer.

**HyperlinkedModelSerializer**

This makes your web API a lot more easy to use (in browser) and would be a nice feature to add.

Let’s say we wanted to see the comments that every post has in each of the Post instances of our API.

With HyperlinkedModelSerializer, instead of having nested primary keys or nested fields, we get a link to each individual Comment (URL).

class PostSerializer(serializers.HyperlinkedModelSerializer):

class Meta:

model = Post

fields = ('id', 'title', 'text', 'created', 'comments')

read\_only\_fields = ('comments',)

**Note:** without the read\_only\_fields, the create form for Posts would always require a comments input, which doesn’t make sense (comments on a post are normally made AFTER the post is created).

Another way of hyperlinking is just adding a HyperlinkedRelatedField definition to a normal serializer.

class PostSerializer(serializers.ModelSerializer):

comments = serializers.HyperlinkedRelatedField(many=True, view\_name='comment-detail', read\_only=True)

class Meta:

model = Post

fields = ('id', 'title', 'text', 'created', 'comments')

**Dynamically modifying fields in the serializer**

This makes your web API a lot more easy for extract limited number of parameter in response. Let’s say you want to set which fields should be used by a serializer at the point of initialization.

Just copy below code and past it in your serliazer file

class DynamicFieldsModelSerializer(serializers.ModelSerializer):

def \_\_init\_\_(self, \*args, \*\*kwargs):

# Don't pass the 'fields' arg up to the superclass

fields = kwargs.pop('fields', None)

# Instantiate the superclass normally

super(DynamicFieldsModelSerializer, self).\_\_init\_\_(\*args, \*\*kwargs)

if fields is not None:

# Drop any fields that are not specified in the `fields` argument.

allowed = set(fields)

existing = set(self.fields.keys())

for field\_name in existing - allowed:

self.fields.pop(field\_name)

Extend DynamicFieldsModelSerializer from your serializer class

class UserSerializer(DynamicFieldsModelSerializer):

class Meta:

model = User

fields = ('id', 'username', 'email')

Mention the fields name inside fields

UserSerializer(user, fields=('id', 'email'))

Here, you will get only id and email from serializer instead of all.

[**Django Passing Data into Viet Cheat Sheet**](https://simplecheatsheet.com/django-passing-data-into-viet/)

Most pages in a project need to present data that are specific to the current user.

**URL parameters**

A URL often needs to accept a parameter telling it which data to access from the database. The second URL pattern shown here looks for the ID of a specific topic and stores it in the parameter topic\_id.

urlpatterns = [

url(r'^$', views.index, name='index'),

url(r'^topics/(?P<topic\_id>\d+)/$',

views.topic, name='topic'),

]

**Using data in a view**

The view uses a parameter from the URL to pull the correct data from the database. In this example the view is sending a context dictionary to the template, containing data that should be displayed on the page.

def topic(request, topic\_id):

"""Show a topic and all its entries."""

topic = Topics.objects.get(id=topic\_id)

entries = topic.entry\_set.order\_by('-date\_added')

context = {

'topic': topic,

'entries': entries,

}

return render(request, 'learning\_logs/topic.html', context)

**Using data in a template**

The data in the view function’s context dictionary is available within the template. This data is accessed using template variables, which are indicated by doubled curly braces. The vertical line after a template variable indicates a filter. In this case a filter called date formats date objects, and the filter linebreaks renders paragraphs properly on a web page.

{% extends 'learning\_logs/base.html' %}

{% block content %}

<p>Topic: {{ topic }}</p>

<p>Entries:</p>

<ul>

{% for entry in entries %}

<li>

<p>{{ entry.date\_added|date:'M d, Y H:i' }}</p>

<p>{{ entry.text|linebreaks }}</p>

</li>

{% empty %}

<li>There are no entries yet.</li>

{% endfor %}

</ul>

{% endblock content %}

[**Django: Rest Framework Views Cheat Sheet**](https://simplecheatsheet.com/django-rest-framework-views/)

There are many options for creating views for your web API, it really depends on what you want and personal preference.

**Using Function-based views:**

from rest\_framework.decorators import api\_view

from rest\_framework.response import Response

from rest\_framework.parsers import JSONParser

from rest\_framework import status

from posts.models import Post

from posts.serializers import PostSerializer

@api\_view(['GET', 'POST'])

def post\_list(request, format=None):

if request.method == 'GET':

posts = Post.objects.all()

serializer = PostSerializer(posts, many=True)

return Response(serializer.data)

elif request.method == 'POST':

data = JSONParser().parse(request)

serializer = PostSerializer(data=data)

if serializer.is\_valid():

serializer.save()

return Response(serializer.data, status=status.HTTP\_201\_CREATED)

return Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)

**Using Class-based views:**

from rest\_framework.response import Response

from rest\_framework import status

from rest\_framework.views import APIView

from posts.models import Post

from posts.serializers import PostSerializer

class PostList(APIView):

def get(self, request, format=None):

snippets = Post.objects.all()

serializer = PostSerializer(snippets, many=True)

return Response(serializer.data)

def post(self, request, format=None):

serializer = PostSerializer(data=request.data)

if serializer.is\_valid():

serializer.save()

return Response(serializer.data, status=status.HTTP\_201\_CREATED)

return Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)

**Using Generic Class-based views:**

from rest\_framework import generics

from posts.models import Post

from posts.serializers import PostSerializer

class PostList(generics.ListCreateAPIView):

queryset = Post.objects.all()

serializer\_class = PostSerializer

**Using Mixins:**

from rest\_framework import generics, mixins

from posts.models import Post

from posts.serializers import PostSerializer

class PostList(generics.GenericAPIView,

mixins.ListModelMixin,

mixins.CreateModelMixin

):

queryset = Post.objects.all()

serializer\_class = PostSerializer

def get(self, request, \*args, \*\*kwargs):

return self.list(request, \*args, \*\*kwargs)

def post(self, request, \*args, \*\*kwargs):

return self.create(request, \*args, \*\*kwargs)

**Using ViewSets:**

With ModelViewSet (in this case), you don’t have to create separate views for getting list of objects and detail of one object. ViewSet will handle it for you in a consistent way for both methods.

from rest\_framework import viewsets

from posts.models import Post

from posts.serializers import PostSerializer

class PostViewSet(viewsets.ModelViewSet):

"""

A viewset for viewing and editing post instances.

"""

queryset = Post.objects.all()

serializer\_class = PostSerializer

**Routers**

Routers in ViewSets allow the URL configuration for your API to be automatically generated using naming standards.

from rest\_framework.routers import DefaultRouter

from posts.views import PostViewSet

router = DefaultRouter()

router.register(r'users', UserViewSet)

urlpatterns = router.urls

**Custom Actions in ViewSets**

DRF provides helpers to add custom actions for *ad-hoc* behaviours with the @action decorator. The router will configure its url accordingly. For example, we can add a comments action in the our PostViewSet to retrieve all the comments of a specific post as follows:

from rest\_framework import viewsets

from rest\_framework.decorators import action

from posts.models import Post

from posts.serializers import PostSerializer, CommentSerializer

class PostViewSet(viewsets.ModelViewSet):

...

@action(methods=['get'], detail=True)

def comments(self, request, pk=None):

try:

post = Post.objects.get(id=pk)

except Post.DoesNotExist:

return Response({"error": "Post not found."},

status=status.HTTP\_400\_BAD\_REQUEST)

comments = post.comments.all()

return Response(CommentSerializer(comments, many=True))

Upon registering the view as router.register(r'posts', PostViewSet), this action will then be available at the url posts/{pk}/comments/.

[**Django: Building a Page Cheat Sheet**](https://simplecheatsheet.com/django-building-a-page/)

Users interact with a project through web pages, and a project’s home page can start out as a simple page with no data. A page usually needs a URL, a view, and a template.

**Mapping a project’s URLs**

The project’s main urls.py file tells Django where to find the urls.py  
files associated with each app in the project.

from django.conf.urls import include, url

from django.contrib import admin

urlpatterns = [

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

url(r'', include('learning\_logs.urls', namespace='learning\_logs')),

]

**Mapping an app’s URLs**

An app’s urls.py file tells Django which view to use for each URL in  
the app. You’ll need to make this file yourself, and save it in the  
app’s folder.

from django.conf.urls import url

from . import views

urlpatterns = [

url(r'^$', views.index, name='index'),

]

**Writing a simple view**

A view takes information from a request and sends data to the browser, often through a template. View functions are stored in an app’s views.py file. This simple view function doesn’t pull in any data, but it uses the template index.html to render the home page. from django.shortcuts import render

def index(request):

"""The home page for Learning Log."""

return render(request, 'learning\_logs/index.html')

**Writing a simple template**

A template sets up the structure for a page. It’s a mix of html and template code, which is like Python but not as powerful. Make a folder called templates inside the project folder. Inside the templates folder make another folder with the same name as the app. This is where the template files should be saved.

<p>Learning Log</p>

<p>Learning Log helps you keep track of your

learning, for any topic you're learning

about.</p>

[**Working with Django Models**](https://simplecheatsheet.com/working-with-django-models/)

**Defining a model**

To define the models for your app, modify the file models.py that was created in your app’s folder. The **str**() method tells Django how to represent data objects based on this model.

from django.db import models

"""A topic the user is learning about."""

class Topic(models.Model):

text = models.CharField(max\_length=200)

date\_added = models.DateTimeField(

auto\_now\_add=True)

def **str**(self):

return self.text

**Defining a model with a foreign key**

class Entry(models.Model):

"""Learning log entries for a topic."""

topic = models.ForeignKey(Topic)

text = models.TextField()

date\_added = models.DateTimeField(

auto\_now\_add=True)

def **str**(self):

return self.text[:50] + "…"

**Activating a model**

To use a model the app must be added to the tuple INSTALLED\_APPS, which is stored in the project’s settings.py file

INSTALLED\_APPS = (

--snip--

'django.contrib.staticfiles',

# My apps

'learning\_logs',

)

**Migrating the database**

The database needs to be modified to store the kind of data that the model represents.

python manage.py makemigrations learning\_logs

python manage.py migrate

**Creating a superuser**

A superuser is a user account that has access to all aspects of the project.

python manage.py createsuperuser

**Registering a model**

You can register your models with Django’s admin site, which  
makes it easier to work with the data in your project. To do this,  
modify the app’s admin.py file.

from django.contrib import admin  
from learning\_logs.models import Topic  
admin.site.register(Topic)

[**Install Django**](https://simplecheatsheet.com/install-django/)

It’s usually best to install Django to a virtual environment, where your project can be isolated from your other Python projects. Most commands assume you’re working in an active virtual environment.

Create a virtual environment

$ python –m venv v

Activate the environment (Linux and OS X)

$ source v/bin/activate

Activate the environment (Windows)

venv\Scripts\activate

Install Django to the active environment

(venv)$ pip install Django