Django Rest API- From Udemy Beginner Class

### By Mark Winterbottom

Table of Contents

[By Mark Winterbottom 1](#_Toc14951382)

[Technologies included In This Course: 6](#_Toc14951383)

[Background in REST API 6](#_Toc14951384)

[What is a REST API? 6](#_Toc14951385)

[Understanding REST API Design 7](#_Toc14951386)

[**Client-Server** 7](#_Toc14951387)

[**Stateless** 7](#_Toc14951388)

[**Cache** 7](#_Toc14951389)

[**Uniform Interface** 8](#_Toc14951390)

[**Layered System** 8](#_Toc14951391)

[**Code on Demand** 9](#_Toc14951392)

[DRF: 9](#_Toc14951393)

[Vagrant: 10](#_Toc14951394)

[»Why Vagrant? 10](#_Toc14951395)

[Virtual Box: 10](#_Toc14951396)

[ModHeader 11](#_Toc14951397)

[Teacher’s Cheat Sheet for This Course 12](#_Toc14951398)

[Build a Backend REST API with Python & Django - Beginner 12](#_Toc14951399)

[Git 12](#_Toc14951400)

[SSH Key Management 13](#_Toc14951401)

[Virtual Environments 13](#_Toc14951402)

[Django Management Commands 13](#_Toc14951403)

[Vagrant 14](#_Toc14951404)

[Terminal / GitBash Commands 14](#_Toc14951405)

[Finding Your Home DIR on MAC OS 15](#_Toc14951406)

[Create Git Project 15](#_Toc14951407)

[Create Readme file 16](#_Toc14951408)

[Create .gitignore file 16](#_Toc14951409)

[Create a License File 19](#_Toc14951410)

[Git add and git commit 20](#_Toc14951411)

[Push Could to github 21](#_Toc14951412)

[Create a Public/Private key pair 21](#_Toc14951413)

[Store SSH Public Key (Never give away your private key) 22](#_Toc14951414)

[Create a new repo for this course on Github 23](#_Toc14951415)

[Do a git add origin to add the new repo to your local git repo through the CMD 23](#_Toc14951416)

[Then do a git push orgin master 23](#_Toc14951417)

[Create Local Dev Server 23](#_Toc14951418)

[Configure a Vagrantfile 23](#_Toc14951419)

[Configure Vagrant Box 24](#_Toc14951420)

[Running and connecting to our dev server 26](#_Toc14951421)

[Connect to dev box 26](#_Toc14951422)

[Exit vargrant dev server: 27](#_Toc14951423)

[Section 5: Creating our Django Application 27](#_Toc14951424)

[Create python virtual env on vagrant server 27](#_Toc14951425)

[Activate the virtual server 27](#_Toc14951426)

[Close virtual env 27](#_Toc14951427)

[Install required Python Packages for our project 27](#_Toc14951428)

[Verify what got installed: 28](#_Toc14951429)

[Create New Django Project 29](#_Toc14951430)

[Create a Django application 29](#_Toc14951431)

[Enable our app in Django settings.py file 30](#_Toc14951432)

[Start The server 30](#_Toc14951433)

[Models: 31](#_Toc14951434)

[Substituting a custom User model¶ 32](#_Toc14951435)

[Using a custom user model when starting a project¶ 33](#_Toc14951436)

[Create customer User Manager 34](#_Toc14951437)

[Set Custom User Model in settings.py 34](#_Toc14951438)

[Then push Model Changes to git and Github 35](#_Toc14951439)

[Create Super User 36](#_Toc14951440)

[Register New Model in admin.py file in your application 37](#_Toc14951441)

[Looking at Admin Console 37](#_Toc14951442)

[Section 8 – API Views 38](#_Toc14951443)

[APIView 38](#_Toc14951444)

[When to use APIViews? 38](#_Toc14951445)

[Create APIView 38](#_Toc14951446)

[Configure a View URL 39](#_Toc14951447)

[Projects url first 39](#_Toc14951448)

[Render our new view 40](#_Toc14951449)

[Now push again to git 40](#_Toc14951450)

[Add a serializer 41](#_Toc14951451)

[Serializer Doc: 41](#_Toc14951452)

[Serializers 41](#_Toc14951453)

[Declaring Serializers 42](#_Toc14951454)

[Serializing objects 43](#_Toc14951455)

[Deserializing objects 43](#_Toc14951456)

[Saving instances 44](#_Toc14951457)

[Validation 47](#_Toc14951458)

[Accessing the initial data and instance 51](#_Toc14951459)

[Partial updates 51](#_Toc14951460)

[Dealing with nested objects 51](#_Toc14951461)

[Writable nested representations 52](#_Toc14951462)

[Dealing with multiple objects 56](#_Toc14951463)

[Including extra context 57](#_Toc14951464)

[ModelSerializer 58](#_Toc14951465)

[Specifying which fields to include 59](#_Toc14951466)

[Specifying nested serialization 60](#_Toc14951467)

[Specifying fields explicitly 61](#_Toc14951468)

[Specifying read only fields 61](#_Toc14951469)

[Additional keyword arguments 62](#_Toc14951470)

[Relational fields 63](#_Toc14951471)

[Customizing field mappings 64](#_Toc14951472)

[**.serializer\_field\_mapping** 64](#_Toc14951473)

[**.serializer\_related\_field** 64](#_Toc14951474)

[**serializer\_url\_field** 64](#_Toc14951475)

[**serializer\_choice\_field** 64](#_Toc14951476)

[**The field\_class and field\_kwargs API** 64](#_Toc14951477)

[**.build\_standard\_field(self, field\_name, model\_field)** 65](#_Toc14951478)

[**.build\_relational\_field(self, field\_name, relation\_info)** 65](#_Toc14951479)

[**.build\_nested\_field(self, field\_name, relation\_info, nested\_depth)** 65](#_Toc14951480)

[**.build\_property\_field(self, field\_name, model\_class)** 65](#_Toc14951481)

[**.build\_url\_field(self, field\_name, model\_class)** 65](#_Toc14951482)

[**.build\_unknown\_field(self, field\_name, model\_class)** 66](#_Toc14951483)

[HyperlinkedModelSerializer 66](#_Toc14951484)

[Absolute and relative URLs 66](#_Toc14951485)

[How hyperlinked views are determined 67](#_Toc14951486)

[Changing the URL field name 68](#_Toc14951487)

[ListSerializer 69](#_Toc14951488)

[**allow\_empty** 69](#_Toc14951489)

[**Customizing ListSerializer behavior** 69](#_Toc14951490)

[BaseSerializer 73](#_Toc14951491)

[Advanced serializer usage 79](#_Toc14951492)

[Overriding serialization and deserialization behavior 79](#_Toc14951493)

[Serializer Inheritance 80](#_Toc14951494)

[Dynamically modifying fields 81](#_Toc14951495)

[**Example** 82](#_Toc14951496)

[Customizing the default fields 83](#_Toc14951497)

[Third party packages 84](#_Toc14951498)

[Django REST marshmallow 84](#_Toc14951499)

[Serpy 84](#_Toc14951500)

[MongoengineModelSerializer 84](#_Toc14951501)

[GeoFeatureModelSerializer 84](#_Toc14951502)

[HStoreSerializer 84](#_Toc14951503)

[Dynamic REST 84](#_Toc14951504)

[Dynamic Fields Mixin 85](#_Toc14951505)

[DRF FlexFields 85](#_Toc14951506)

[Serializer Extensions 85](#_Toc14951507)

[HTML JSON Forms 85](#_Toc14951508)

[DRF-Base64 85](#_Toc14951509)

[QueryFields 85](#_Toc14951510)

[DRF Writable Nested 85](#_Toc14951511)

[Create the serializer 86](#_Toc14951512)

## Technologies included In This Course:

* **Django Rest Framework-** Django REST framework is a powerful and flexible toolkit for building Web APIs
* **Vagrant** - Vagrant is a tool for building and managing virtual machine environments in a single workflow
* **Virtual Box** - VirtualBox is a powerful x86 and AMD64/Intel64 [virtualization](https://www.virtualbox.org/wiki/Virtualization) product for enterprise as well as home use
* **Mod Header**- It is a Chrome Extension that modifies request and response headers
* Atom- IDE (need to use separate Command Window)
* Git – Source Code
* GitHub

# Background in REST API

# **What is a REST API?**

REST or [RESTful API](https://www.mulesoft.com/resources/api/restful-api) design (Representational State Transfer) is designed to take advantage of existing protocols. While REST can be used over nearly any protocol, it usually takes advantage of HTTP when used for Web APIs. This means that developers do not need to install libraries or additional software in order to take advantage of a REST API design. REST API Design was defined by Dr. Roy Fielding in his 2000 doctorate dissertation. It is notable for its incredible layer of flexibility. Since data is not tied to methods and resources, REST has the ability to handle multiple types of calls, return different data formats and even change structurally with the correct implementation of hypermedia.

This freedom and flexibility inherent in REST API design allow you to build an API that meets your needs while also meeting the needs of very diverse customers. Unlike [SOAP](https://www.mulesoft.com/webinars/api/soap-connect), REST is not constrained to XML, but instead can return XML, JSON, YAML or any other format depending on what the client requests. And unlike RPC, users aren’t required to know procedure names or specific parameters in a specific order.

However, there are drawbacks to REST API design. You can lose the ability to maintain state in REST, such as within sessions, and it can be more difficult for newer developers to use. It’s also important to understand what makes a REST API RESTful, and why these constraints exist before [building your API](https://www.mulesoft.com/lp/ebook/api/building-api-blueprint). After all, if you do not understand why something is designed in the manner it is, you can hinder your efforts without even realizing it.

## Understanding REST API Design

While most APIs claim to be RESTful, they fall short of the requirements and constraints asserted by Dr. Fielding. There are six key constraints to REST API design to be aware of when deciding whether this is the right [API type](https://www.mulesoft.com/resources/api/types-of-apis) for your project.

### **Client-Server**

The client-server constraint works on the concept that the client and the server should be separate from each other and allowed to evolve individually and independently. In other words, I should be able to make changes to my mobile application without impacting either the data structure or the database design on the server. At the same time, I should be able to modify the database or make changes to my server application without impacting the mobile client. This creates a separation of concerns, letting each application grow and scale independently of the other and allowing your organization to grow quickly and efficiently.

### **Stateless**

REST APIs are stateless, meaning that calls can be made independently of one another, and each call contains all of the data necessary to complete itself successfully. A REST API should not rely on data being stored on the server or sessions to determine what to do with a call, but rather solely rely on the data that is provided in that call itself. Identifying information is not being stored on the server when making calls. Instead, each call has the necessary data in itself, such as the API key, access token, user ID, etc. This also helps increase the API’s reliability by having all of the data necessary to make the call, instead of relying on a series of calls with server state to create an object, which may result in partial fails. Instead, in order to reduce memory requirements and keep your application as scalable as possible, a RESTful API requires that any state is stored on the client—not on the server.

### **Cache**

Because a stateless API can increase request overhead by handling large loads of incoming and outbound calls, a REST API should be designed to encourage the storage of cacheable data. This means that when data is cacheable, the response should indicate that the data can be stored up to a certain time (expires-at), or in cases where data needs to be real-time, that the response should not be cached by the client. By enabling this critical constraint, you will not only greatly reduce the number of interactions with your API, reducing internal server usage, but also provide your API users with the tools necessary to provide the fastest and most efficient apps possible. Keep in mind that caching is done on the client side. While you may be able to cache some data within your architecture to perform overall performance, the intent is to instruct the client on how it should proceed and whether or not the client can store the data temporarily.

### **Uniform Interface**

The key to the decoupling client from server is having a uniform interface that allows independent evolution of the application without having the application’s services, models, or actions tightly coupled to the [API layer](https://www.mulesoft.com/resources/api/api-layer) itself. The uniform interface lets the client talk to the server in a single language, independent of the architectural backend of either. This interface should provide an unchanging, standardized means of communicating between the client and the server, such as using HTTP with URI resources, CRUD (Create, Read, Update, Delete), and JSON.

### **Layered System**

As the name implies, a layered system is a system comprised of layers, with each layer having a specific functionality and responsibility. If we think of a Model View Controller framework, each layer has its own responsibilities, with the models comprising how the data should be formed, the controller focusing on the incoming actions and the view focusing on the output. Each layer is separate but also interacts with the other. In REST API design, the same principle holds true, with different layers of the architecture working together to build a hierarchy that helps create a more scalable and modular application.

A layered system also lets you encapsulate legacy systems and move less commonly accessed functionality to a shared intermediary while also shielding more modern and commonly used components from them. Additionally, the layered system gives you the freedom to move systems in and out of your architecture as technologies and services evolve, increasing flexibility and longevity as long as you keep the different modules as loosely coupled as possible. There are substantial [security benefits](https://www.mulesoft.com/webinars/api/security-best-practices) of having a layered system since it allows you to stop attacks at the proxy layer, or within other layers, preventing them from getting to your actual server architecture. By utilizing a layered system with a proxy, or creating a single point of access, you are able to keep critical and more vulnerable aspects of your architecture behind a firewall, preventing direct interaction with them by the client. Keep in mind that security is not based on single “stop all” solution, but rather on having multiple layers with the understanding that certain security checks may fail or be bypassed. As such, the more security you are able to implement into your system, the more likely you are to prevent damaging Attacks.

### **Code on Demand**

Perhaps the least known of the six constraints, and the only optional constraint, Code on Demand allows for code or applets to be transmitted via the API for use within the application. In essence, it creates a smart application that is no longer solely dependent on its own code structure. However, perhaps because it’s ahead of its time, Code on Demand has struggled for adoption as Web APIs are consumed across multiple languages and the transmission of code raises security questions and concerns. (For example, the directory would have to be writeable, and the firewall would have to let what may normally be restricted content through.)

Together, these constraints make up the theory of [Representational State Transfer](https://www.mulesoft.com/resources/esb/restful-web-services-esb-rest-services-integration), or REST. As you look back through these you can see how each successive constraint builds on top of the previous, eventually creating a rather complex—but powerful and flexible—application program interface. But most importantly, these constraints make up a design that operates similarly to how we access pages in our browsers on the World Wide Web. It creates an API that is not dictated by its architecture, but by the representations that it returns, and an API that—while architecturally stateless—relies on the representation to dictate the application’s state.

For more information about REST [API Design](https://www.mulesoft.com/platform/api/anypoint-designer), check out the eBook Undisturbed REST: [A Guide to Designing the Perfect API](https://www.mulesoft.com/lp/ebook/api/restbook).

## DRF:

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](https://restframework.herokuapp.com/) is a huge usability win for your developers.
* [Authentication policies](https://www.django-rest-framework.org/api-guide/authentication/) including packages for [OAuth1a](https://www.django-rest-framework.org/api-guide/authentication/#django-rest-framework-oauth) and [OAuth2](https://www.django-rest-framework.org/api-guide/authentication/#django-oauth-toolkit).
* [Serialization](https://www.django-rest-framework.org/api-guide/serializers/) that supports both [ORM](https://www.django-rest-framework.org/api-guide/serializers#modelserializer) and [non-ORM](https://www.django-rest-framework.org/api-guide/serializers#serializers) data sources.
* Customizable all the way down - just use [regular function-based views](https://www.django-rest-framework.org/api-guide/views#function-based-views) if you don't need the [more](https://www.django-rest-framework.org/api-guide/generic-views/) [powerful](https://www.django-rest-framework.org/api-guide/viewsets/) [features](https://www.django-rest-framework.org/api-guide/routers/).
* [Extensive documentation](https://www.django-rest-framework.org/), and [great community support](https://groups.google.com/forum/?fromgroups#!forum/django-rest-framework).
* Used and trusted by internationally recognised companies including [Mozilla](https://www.mozilla.org/en-US/about/), [Red Hat](https://www.redhat.com/), [Heroku](https://www.heroku.com/), and [Eventbrite](https://www.eventbrite.co.uk/about/).

## Vagrant:

Vagrant is a tool for building and managing virtual machine environments in a single workflow. With an easy-to-use workflow and focus on automation, Vagrant lowers development environment setup time, increases production parity, and makes the "works on my machine" excuse a relic of the past.

If you are already familiar with the basics of Vagrant, the [documentation](https://www.vagrantup.com/docs/index.html) provides a better reference build for all available features and internals.

## **[»](https://www.vagrantup.com/intro/index.html" \l "why-vagrant-)Why Vagrant?**

Vagrant provides easy to configure, reproducible, and portable work environments built on top of industry-standard technology and controlled by a single consistent workflow to help maximize the productivity and flexibility of you and your team.

To achieve its magic, Vagrant stands on the shoulders of giants. Machines are provisioned on top of VirtualBox, VMware, AWS, or [any other provider](https://www.vagrantup.com/docs/providers/). Then, industry-standard [provisioning tools](https://www.vagrantup.com/docs/provisioning/) such as shell scripts, Chef, or Puppet, can automatically install and configure software on the virtual machine.

## Virtual Box:

When we describe VirtualBox as a "virtualization" product, we refer to "full virtualization", that is, the particular kind of virtualization that allows an *unmodified* operating system with all of its installed software to run in a special environment, on top of your existing operating system. This environment, called a "virtual machine", is created by the virtualization software by intercepting access to certain hardware components and certain features. The physical computer is then usually called the "host", while the virtual machine is often called a "guest". Most of the guest code runs unmodified, directly on the host computer, and the guest operating system "thinks" it's running on real machine.

This approach, often called "native virtualization", is different from mere emulation. With that approach, as performed by programs such as [BOCHS](http://bochs.sourceforge.net/), guest code is not allowed to run directly on the host. Instead, every single machine instruction is translated ("emulated"). While emulators theoretically allow running code written for one type of hardware on completely different hardware (say, running 64-bit code on 32-bit hardware), they are typically quite slow. Virtualizers such as VirtualBox, on the other hand, can achieve near-native performance for the guest code, but can only run guest code that was written for the same target hardware (such as 32-bit Linux on a 32-bit Windows host).

VirtualBox is also different from so-called "paravirtualization" solutions such as [Xen](http://www.xen.org/), which require that the guest operating system be modified.

There are several scenarios that make virtualization attractive:

* **Operating system support.** With a virtualizer such as VirtualBox, one can run software written for one operating system on another (say, Windows software on Linux) without having to reboot.
* **Infrastructure consolidation.** Since the full performance of today's computers is rarely needed full-time, instead of running many such physical computers, one can "pack" many virtual machines onto a few powerful hosts and balance the loads between them. This can save a lot of hardware costs: e.g. by consolidating many servers into a few. VirtualBox is unique on the virtualization market in that it also allows for consolidating clients onto just a few RDP servers, with full client USB support, while "thin clients" only need to display RDP data.
* **Testing and disaster recovery.** Especially with the use of snapshots?, one can mess with a computing environment by running it as a virtual machine. If something goes wrong, one can easily switch back to a previous snapshot and avoid the need of frequent backups and restores.

## ModHeader

Modify request and response headers

Add and modify HTTP request headers and response headers.

\*\* Features \*\*

- Add/modify/remove request headers and response headers

- Enable header modification based on URL/resource type

- Add comments to header

- Multiple different profiles

- Sorting headers by name, value, or comments

- Append value to existing request or response header

- Export and import header

- Clone profile

- Cloud backup

- Tab locking!

# Teacher’s Cheat Sheet for This Course

# Build a Backend REST API with Python & Django - Beginner

This is the supplementary cheat sheet document for our course: [Build a Backend REST API with Python & Django - Beginner](https://www.udemy.com/django-python/?couponCode=2019UPDATECHEATSHEET)

* [Git](https://github.com/LondonAppDev/build-a-backend-api-python-drf-beginner-cheat-sheet/blob/master/README.md#git)
* [SSH Key Management](https://github.com/LondonAppDev/build-a-backend-api-python-drf-beginner-cheat-sheet/blob/master/README.md#ssh-key-management)
* [Virtual Environments](https://github.com/LondonAppDev/build-a-backend-api-python-drf-beginner-cheat-sheet/blob/master/README.md#virtual-environments)
* [Django Management Commands](https://github.com/LondonAppDev/build-a-backend-api-python-drf-beginner-cheat-sheet/blob/master/README.md#django-management-commands)
* [Vagrant](https://github.com/LondonAppDev/build-a-backend-api-python-drf-beginner-cheat-sheet/blob/master/README.md#vagrant)
* [Terminal / GitBash Commands](https://github.com/LondonAppDev/build-a-backend-api-python-drf-beginner-cheat-sheet/blob/master/README.md#terminal-gitbash-commands)

## Git

Use the below Git commands in the Windows Command Prompt or macOS Terminal.

**Configure default email and name**

Note: This only needs to be done the first time you use Git on your machine

git config --global user.email "your@email.com"

git config --global user.name "Your Name"

**Initialise a new Git repository**

git init

**Commit changes to Git**

git add .

git commit -am "Commit message"

**Set Git remote**

Note: This only needs to be done once, the details are provided by GitHub after creating a new project

git remote add origin <URL TO PROJECT>

git push -u origin master

**Push changes to GitHub**

git push origin

## SSH Key Management

The below commands are used to manage SSH keys on your local development machine.

**Checking for existing SSH key**

ls ~/.ssh/

**Print contents of public key**

cat ~/.ssh/id\_rsa.pub

**Generate new SSH key on your local machine**

ssh-keygen -t rsa -b 4096 -C "EMAIL ADDRESS"

## Virtual Environments

The below commands are used for managing Virtual Environments using Python3-env. Use these commands when connected to your Vagrant server.

**Create new environment**

python -m venv ~/env

**Activate virtual environment**

source ~/env/bin/activate

**De-activate virtual environment**

deactivate

**Install requirements from requirements.txt**

Note: Virtual environment must be activated

pip install -r requirements.txt

## Django Management Commands

**Create new Django project**

django-admin.py startproject profiles\_project .

**Create new Django app**

python manage.py startapp profiles\_api

**Start Django development server**

python manage.py runserver 0.0.0.0:8000

**Create database migrations file**

python manage.py makemigrations

**Run migrations**

python manage.py migrate

**Create new superuser**

python manage.py createsuperuser

## Vagrant

These commands are used for managing Vagrant using the GitBash or Terminal windows.

**Initialise Vagrant on project**

vagrant init ubuntu/bionic64

**Start Vagrant box**

vagrant up

**Connect to Vagrant box**

vagrant ssh

**Disconnect from Vagrant box**

Note: This command is a standard linux command for ending an SSH session

exit

**Stop Vagrant box**

vagrant halt

**Remove Vagrant box**

vagrant destroy

**Update Vagrant box image**

Note: you must rebuild the image after updating

vagrant box update

## Terminal / GitBash Commands

Change directory

cd /directory\_name

Change to parent directory

cd ..

### Finding Your Home DIR on MAC OS

From File structure:

Go-🡪 “Go To Folder” and in the search field enter Tilda “~”

This will take you to your home dir for me it is my username

ATOM

Setup project

File 🡪 “ Add Project Folder” navigate to our newly created folder that we created “profiles-rest-api”

## Create Git Project

Claudias-iMac:profiles-rest-api claudiaacerra$ **git init**

\*\*\*MUST RESTART ATOM TO DETECT THAT IT IS NOW A GIT REPO\*\*\*

Now this icon will spear next to the project name



## Create Readme file

Create file called readme.md (md stands for markdown)

# is the syntax for a heading



## Create .gitignore file

These are files we will exclude from our git repo

HERE it is for a python and vagrant project. There is a link below on how to do this right in the .gitignore

# Created by https://www.gitignore.io/api/python,vagrant

# Edit at https://www.gitignore.io/?templates=python,vagrant

### Python ###

# Byte-compiled / optimized / DLL files

\_\_pycache\_\_/

\*.py[cod]

\*$py.class

# C extensions

\*.so

# Distribution / packaging

.Python

build/

develop-eggs/

dist/

downloads/

eggs/

.eggs/

lib/

lib64/

parts/

sdist/

var/

wheels/

pip-wheel-metadata/

share/python-wheels/

\*.egg-info/

.installed.cfg

\*.egg

MANIFEST

# PyInstaller

# Usually these files are written by a python script from a template

# before PyInstaller builds the exe, so as to inject date/other infos into it.

\*.manifest

\*.spec

# Installer logs

pip-log.txt

pip-delete-this-directory.txt

# Unit test / coverage reports

htmlcov/

.tox/

.nox/

.coverage

.coverage.\*

.cache

nosetests.xml

coverage.xml

\*.cover

.hypothesis/

.pytest\_cache/

# Translations

\*.mo

\*.pot

# Django stuff:

\*.log

local\_settings.py

db.sqlite3

# Flask stuff:

instance/

.webassets-cache

# Scrapy stuff:

.scrapy

# Sphinx documentation

docs/\_build/

# PyBuilder

target/

# Jupyter Notebook

.ipynb\_checkpoints

# IPython

profile\_default/

ipython\_config.py

# pyenv

.python-version

# celery beat schedule file

celerybeat-schedule

# SageMath parsed files

\*.sage.py

# Environments

.env

.venv

env/

venv/

ENV/

env.bak/

venv.bak/

# Spyder project settings

.spyderproject

.spyproject

# Rope project settings

.ropeproject

# mkdocs documentation

/site

# mypy

.mypy\_cache/

.dmypy.json

dmypy.json

# Pyre type checker

.pyre/

### Python Patch ###

.venv/

### Vagrant ###

# General

.vagrant/

# Log files (if you are creating logs in debug mode, uncomment this)

# \*.logs

### Vagrant Patch ###

\*.box

# End of https://www.gitignore.io/api/python,vagrant

eb-package.zip

static/

@theyogicoderRI

## Create a License File

Protect yourself so no one can sue you.

MIT License

Copyright (c) 2019 YogiCoder

Permission is hereby granted, free of charge, to any person obtaining a copy

of this software and associated documentation files (the "Software"), to deal

in the Software without restriction, including without limitation the rights

to use, copy, modify, merge, publish, distribute, sublicense, and/or sell

copies of the Software, and to permit persons to whom the Software is

furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all

copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR

IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,

FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE

AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER

LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,

OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE

SOFTWARE.

## Git add and git commit

Claudias-iMac:profiles-rest-api claudiaacerra$ **git add .**

Claudias-iMac:profiles-rest-api claudiaacerra$ **git commit -am "First** commit license, readme and .gitignore"

[master (root-commit) 847efd2] First commit license, readme and .gitignore

3 files changed, 162 insertions(+)

create mode 100644 .gitignore

create mode 100644 LICENSE

create mode 100644 readme.md

## Push Could to github

Do not commit any secret keys to github

### Create a Public/Private key pair

Claudias-iMac:profiles-rest-api claudiaacerra$ ls ~/.ssh

ls: /Users/claudiaacerra/.ssh: No such file or directory

Claudias-iMac:profiles-rest-api claudiaacerra$ **ssh-keygen -t rsa -b** 4096 -C "claudia.acerra@gmail.com"

Generating public/private rsa key pair.

Enter file in which to save the key (/Users/claudiaacerra/.ssh/id\_rsa):

Created directory '/Users/claudiaacerra/.ssh'.

Enter passphrase (empty for no passphrase):

Enter same passphrase again:

**Your identification has been saved in** /Users/claudiaacerra/.ssh/id\_rsa.

**Your public key has been saved in** /Users/claudiaacerra/.ssh/id\_rsa.pub.

**The key fingerprint is:**

SHA256:8MIOnrYL1HbymicbxTLz8VRssdfMXr5WtJq84kWaCGE claudia.acerra@gmail.com

**The key's randomart image is:**

+---[RSA 4096]----+

| . |

| . o + |

| .E = . + o|

| . o.o.o . . +.|

| . O B.S ...o|

| . o & =. . = o o|

| . = + .. o = o |

| o.=. .. o |

| \*= .... |

+----[SHA256]-----+

Claudias-iMac:profiles-rest-api claudiaacerra$ **ls ~/.ssh**

id\_rsa id\_rsa.pub

CAT THE PUBLIC KEY TO SAVE IT ON GITHUB FOR THIS COMPUTER

Claudias-iMac:profiles-rest-api claudiaacerra$ **cat ~/.ssh/id\_rsa.pub**

**ssh-rsa ** [**claudia.acerra@gmail.com**](mailto:claudia.acerra@gmail.com)

## Store SSH Public Key (Never give away your private key)

* 1. Login to GitHub
  2. Got to settings
  3. Go to SSH and GPG Keys
  4. Add new SSH key- which gets locked to a given host
* Paste the entire key into the “key text field” below



**After you add it, you will see this in the SSH link on Github**

****

## Create a new repo for this course on Github

### Do a git add origin to add the new repo to your local git repo through the CMD

Claudias-iMac:profiles-rest-api claudiaacerra$ pwd

/Users/claudiaacerra/courses/profiles-rest-api

Claudias-iMac:profiles-rest-api claudiaacerra$ **git remote add origin** [**https://github.com/cacerra1/profiles-rest-api.git**](https://github.com/cacerra1/profiles-rest-api.git)

### Then do a git push orgin master

Claudias-iMac:profiles-rest-api claudiaacerra$ **git push -u origin master**

Username for 'https://github.com': claudia.acerra@gmail.com

Password for 'https://claudia.acerra@gmail.com@github.com':

Counting objects: 5, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (5/5), done.

Writing objects: 100% (5/5), 1.85 KiB | 1.85 MiB/s, done.

Total 5 (delta 0), reused 0 (delta 0)

To https://github.com/cacerra1/profiles-rest-api.git

\* [new branch] master -> master

Branch 'master' set up to track remote branch 'master' from 'origin'.

# Create Local Dev Server

## Configure a Vagrantfile

Claudias-iMac:profiles-rest-api claudiaacerra$ **vagrant init ubuntu/bionic64**

* What this does it inits our project with a new vagrant file and bases it on the ubuntu/bionic64 image. This images are pibliclly available on the vagrant calatog box.
* The vagrant file now gets created automatically into our project folder as that is where we ran this command



## Configure Vagrant Box

Modifying the vagrant file with one we generated for us



I had to change my ports away from 8000 as they were already in use.

The host machine is the physical box we are running this project on and the guest machine is the development server itself. We need to make these ports accessible as we are doing here. **UPDATE: Changed by ports both back to 8000 after I killed some ports and rebooted**

Notice how we set python3 to the default python version. This lets us not need to type “python3” anymore but instead “python”… YAH!!!

## Running and connecting to our dev server

Claudias-iMac:profiles-rest-api claudiaacerra$ **vagrant up**

This downloads the image we have specified in our vagrant file and it will then use virtual box to create a new virtual machine and run our provisioning script when it starts the machine

## Connect to dev box

Claudias-iMac:profiles-rest-api claudiaacerra$ **vagrant ssh**

Now you will see we are in our virtual machine (by the green header on this command line)

Then CD into a folder called vagrant that gets automatically created

**vagrant@ubuntu-bionic**:**~**$ cd /vagrant

The create a new file could text.txt

**vagrant@ubuntu-bionic**:**/vagrant**$ touch test.txt

You will see that this gets created automatically in our project root



**vagrant@ubuntu-bionic**:**/vagrant**$ ls

LICENSE cheatSheet.txt readme.md ubuntu-bionic-18.04-cloudimg-console.log

Vagrantfile hello\_world.py test.txt

## Exit vargrant dev server:

**$exit**

Create a test hello.world file and run it from the cmd and it will print out its text:

**vagrant@ubuntu-bionic**:**/vagrant**$ python hello\_world.py

Hello World

This confirms all is working well

# Section 5: Creating our Django Application

## Create python virtual env on vagrant server

We create in env in the home dir of our vagrant server as opposed to the vagrant server which is synced to our local machine

## Activate the virtual server

**vagrant@ubuntu-bionic**:**/vagrant**$ source ~/env/bin/activate

(env) **vagrant@ubuntu-bionic**:**/vagrant**$

## Close virtual env

$ **deactivate**

## Install required Python Packages for our project

First create a requirements.txt file in the root of our project



Then run this command to install these two packages

(env) **vagrant@ubuntu-bionic**:**/vagrant**$ **pip install -r requirements.txt**

Collecting django==2.2 (from -r requirements.txt (line 1))

Downloading https://files.pythonhosted.org/packages/54/85/0bef63668fb170888c1a2970ec897d4528d6072f32dee27653381a332642/Django-2.2-py3-none-any.whl (7.4MB)

100% |████████████████████████████████| 7.5MB 258kB/s

Collecting djangorestframework==3.9.2 (from -r requirements.txt (line 2))

Downloading https://files.pythonhosted.org/packages/cc/6d/5f225f18d7978d8753c1861368efc62470947003c7f9f9a5cc425fc0689b/djangorestframework-3.9.2-py2.py3-none-any.whl (911kB)

100% |████████████████████████████████| 921kB 1.8MB/s

Collecting pytz (from django==2.2->-r requirements.txt (line 1))

Downloading https://files.pythonhosted.org/packages/3d/73/fe30c2daaaa0713420d0382b16fbb761409f532c56bdcc514bf7b6262bb6/pytz-2019.1-py2.py3-none-any.whl (510kB)

100% |████████████████████████████████| 512kB 2.5MB/s

Collecting sqlparse (from django==2.2->-r requirements.txt (line 1))

Downloading https://files.pythonhosted.org/packages/ef/53/900f7d2a54557c6a37886585a91336520e5539e3ae2423ff1102daf4f3a7/sqlparse-0.3.0-py2.py3-none-any.whl

Installing collected packages: pytz, sqlparse, django, djangorestframework

Successfully installed django-2.2 djangorestframework-3.9.2 pytz-2019.1 sqlparse-0.3.0

(env) **vagrant@ubuntu-bionic**:**/vagrant**$

## Verify what got installed:

(env) **vagrant@ubuntu-bionic**:**/vagrant**$ pip freeze

Django==2.2

djangorestframework==3.9.2

pkg-resources==0.0.0

pytz==2019.1

sqlparse==0.3.0

(env) **vagrant@ubuntu-bionic**:**/vagrant**$

## Create New Django Project

(env) **vagrant@ubuntu-bionic**:**/vagrant**$ **django-admin.py startproject profiles\_project .**

(env) **vagrant@ubuntu-bionic**:**/vagrant**$



## Create a Django application

(env) **vagrant@ubuntu-bionic**:**/vagrant**$ **python manage.py startapp profiles\_api**

(env) **vagrant@ubuntu-bionic**:**/vagrant**$



## Enable our app in Django settings.py file



We added the last three entries in the list above

## Start The server

(env) **vagrant@ubuntu-bionic**:**/vagrant**$ **python manage.py runserver 0.0.0.0:8000**

Watching for file changes with StatReloader

Performing system checks...

System check identified no issues (0 silenced).

You have 19 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, authtoken, contenttypes, sessions.

Run 'python manage.py migrate' to apply them.

July 23, 2019 - 20:41:01

Django version 2.2, using settings 'profiles\_project.settings'

Starting development server at http://0.0.0.0:8000/

Quit the server with CONTROL-C.

Exit

Then go to:

<http://127.0.0.1:8000/>

In your web browser to see the Django Rocket



# Models:

We are creating customer user authentication



## **Substituting a custom**User**model**[¶](https://docs.djangoproject.com/en/1.11/topics/auth/customizing/#substituting-a-custom-user-model)

Some kinds of projects may have authentication requirements for which Django’s built-in [**User**](https://docs.djangoproject.com/en/1.11/ref/contrib/auth/#django.contrib.auth.models.User) model is not always appropriate. For instance, on some sites it makes more sense to use an email address as your identification token instead of a username.

Django allows you to override the default user model by providing a value for the [**AUTH\_USER\_MODEL**](https://docs.djangoproject.com/en/1.11/ref/settings/#std:setting-AUTH_USER_MODEL) setting that references a custom model:

AUTH\_USER\_MODEL = 'myapp.MyUser'

This dotted pair describes the name of the Django app (which must be in your [**INSTALLED\_APPS**](https://docs.djangoproject.com/en/1.11/ref/settings/#std:setting-INSTALLED_APPS)), and the name of the Django model that you wish to use as your user model.

### Using a custom user model when starting a project[¶](https://docs.djangoproject.com/en/1.11/topics/auth/customizing/#using-a-custom-user-model-when-starting-a-project)

If you’re starting a new project, it’s highly recommended to set up a custom user model, even if the default [**User**](https://docs.djangoproject.com/en/1.11/ref/contrib/auth/#django.contrib.auth.models.User)model is sufficient for you. This model behaves identically to the default user model, but you’ll be able to customize it in the future if the need arises:

**from** **django.contrib.auth.models** **import** AbstractUser

**class** **User**(AbstractUser):

**pass**

Don’t forget to point [**AUTH\_USER\_MODEL**](https://docs.djangoproject.com/en/1.11/ref/settings/#std:setting-AUTH_USER_MODEL) to it. Do this before creating any migrations or running **manage.pymigrate** for the first time.

Also, register the model in the app’s **admin.py**:

**from** **django.contrib** **import** admin

**from** **django.contrib.auth.admin** **import** UserAdmin

**from** **.models** **import** User

admin.site.register(User, UserAdmin)

## Create customer User Manager



## Set Custom User Model in settings.py



Make Migrations and Migrate:



(env) **vagrant@ubuntu-bionic**:**/vagrant**$ python manage.py migrate

**Operations to perform:**

**Apply all migrations:** admin, auth, authtoken, contenttypes, profiles\_api, sessions

**Running migrations:**

Applying contenttypes.0001\_initial... **OK**

Applying contenttypes.0002\_remove\_content\_type\_name... **OK**

Applying auth.0001\_initial... **OK**

Applying auth.0002\_alter\_permission\_name\_max\_length... **OK**

Applying auth.0003\_alter\_user\_email\_max\_length... **OK**

Applying auth.0004\_alter\_user\_username\_opts... **OK**

Applying auth.0005\_alter\_user\_last\_login\_null... **OK**

Applying auth.0006\_require\_contenttypes\_0002... **OK**

Applying auth.0007\_alter\_validators\_add\_error\_messages... **OK**

Applying auth.0008\_alter\_user\_username\_max\_length... **OK**

Applying auth.0009\_alter\_user\_last\_name\_max\_length... **OK**

Applying auth.0010\_alter\_group\_name\_max\_length... **OK**

Applying auth.0011\_update\_proxy\_permissions... **OK**

Applying profiles\_api.0001\_initial... **OK**

Applying admin.0001\_initial... **OK**

Applying admin.0002\_logentry\_remove\_auto\_add... **OK**

Applying admin.0003\_logentry\_add\_action\_flag\_choices... **OK**

Applying authtoken.0001\_initial... **OK**

Applying authtoken.0002\_auto\_20160226\_1747... **OK**

Applying sessions.0001\_initial... **OK**

(env) **vagrant@ubuntu-bionic**:**/vagrant**$

### Then push Model Changes to git and Github

This from new command window- do not need to be in virtual env, just at root of our project folder

Last login: Thu Jul 25 09:05:38 on ttys000

Claudias-iMac:profiles-rest-api claudiaacerra$ pwd

/Users/claudiaacerra/courses/profiles-rest-api

Claudias-iMac:profiles-rest-api claudiaacerra$ git add .

Claudias-iMac:profiles-rest-api claudiaacerra$ git commit -am "Model and DB changes"

[master de75d57] Model and DB changes

4 files changed, 98 insertions(+), 1 deletion(-)

create mode 100644 profiles\_api/migrations/0001\_initial.py

Claudias-iMac:profiles-rest-api claudiaacerra$ git push origin

Counting objects: 9, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (9/9), done.

Writing objects: 100% (9/9), 863.73 KiB | 10.53 MiB/s, done.

Total 9 (delta 5), reused 0 (delta 0)

remote: Resolving deltas: 100% (5/5), completed with 5 local objects.

remote:

remote: GitHub found 2 vulnerabilities on cacerra1/profiles-rest-api's default branch (2 moderate). To find out more, visit:

remote: https://github.com/cacerra1/profiles-rest-api/network/alerts

remote:

To https://github.com/cacerra1/profiles-rest-api.git

b7dbfcd..de75d57 master -> master

Claudias-iMac:profiles-rest-api claudiaacerra$

# Create Super User

(env) **vagrant@ubuntu-bionic**:**/vagrant**$ python manage.py createsuperuser

Email: claudia.acerra@gmail.com

Name: admin

Password:

Password (again):

**This password is too common.**

Bypass password validation and create user anyway? [y/N]: y

Superuser created successfully.

(env) **vagrant@ubuntu-bionic**:**/vagrant**$

(PW is admin123)

## Register New Model in admin.py file in your application



# Looking at Admin Console

First thing is we loggin with our email address instead of user name

The Rest framework creates the other two things



# Section 8 – API Views

Django Rest Framework offers helper classes:

APIView and **ViewSet**

## APIView

Allows us define functions that match the standard HTTP methods

1. HTTP Get
2. HTTP Post
3. HTTP Put- Update
4. HTTP patch partially update an item
5. HTTP delete

Give us the most control of our HTTP logic

## When to use APIViews?

When you need full control over application logic (updating multiple data sources in one API call)?

1. Processing files and rendering a synchronous response
2. Calling other API services in one request
3. Access to local files or data

## Create APIView



## Configure a View URL

# Projects url first





## Render our new view



## Now push again to git

Claudias-iMac:profiles-rest-api claudiaacerra$ git add .

Claudias-iMac:profiles-rest-api claudiaacerra$ git commit -am "Add new view"

[master 96312d2] Add new view

4 files changed, 37 insertions(+), 24 deletions(-)

create mode 100644 profiles\_api/urls.py

rewrite profiles\_api/views.py (98%)

rewrite profiles\_project/urls.py (88%)

Claudias-iMac:profiles-rest-api claudiaacerra$ git push origin

Counting objects: 8, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (8/8), done.

Writing objects: 100% (8/8), 543.37 KiB | 6.97 MiB/s, done.

Total 8 (delta 4), reused 0 (delta 0)

remote: Resolving deltas: 100% (4/4), completed with 4 local objects.

remote:

remote: GitHub found 2 vulnerabilities on cacerra1/profiles-rest-api's default branch (2 moderate). To find out more, visit:

remote: https://github.com/cacerra1/profiles-rest-api/network/alerts

remote:

To https://github.com/cacerra1/profiles-rest-api.git

c7733f4..96312d2 master -> master

Claudias-iMac:profiles-rest-api claudiaacerra$

## Add a serializer

Allows us to convert data input to into python objects and vice versa

Kind of similar to a Django form.

The serializer will receive the content that we post to the API

## Serializer Doc:

# [**Serializers**](https://www.django-rest-framework.org/api-guide/serializers/#serializers)

*Expanding the usefulness of the serializers is something that we would like to address. However, it's not a trivial problem, and it will take some serious design work.*

*— Russell Keith-Magee,*[*Django users group*](https://groups.google.com/d/topic/django-users/sVFaOfQi4wY/discussion)

**This is what they do. Two things:**

1. Serializers allow complex data such as querysets and model instances to be converted to native Python datatypes that can then be easily rendered into JSON, XML or other content types.
2. Serializers also provide deserialization, allowing parsed data to be converted back into complex types, after first validating the incoming data.

The serializers in REST framework work very similarly to Django's Form and ModelForm classes. We provide a Serializerclass which gives you a powerful, generic way to control the output of your responses, as well as a ModelSerializer class which provides a useful shortcut for creating serializers that deal with model instances and querysets.

## [**Declaring Serializers**](https://www.django-rest-framework.org/api-guide/serializers/#declaring-serializers)

Let's start by creating a simple object we can use for example purposes:

1.create an object (instance of a class to serialize)

from datetime import datetime

class Comment(object):

def \_\_init\_\_(self, email, content, created=None):

self.email = email

self.content = content

self.created = created or datetime.now()

# instance of the class

comment = Comment(email='leila@example.com', content='foo bar')

We'll declare a serializer that we can use to serialize and deserialize data that corresponds to Comment objects.

Declaring a serializer looks very similar to declaring a form:

* 1. Create a serializer

from rest\_framework import serializers

class CommentSerializer(serializers.Serializer):

email = serializers.EmailField()

content = serializers.CharField(max\_length=200)

created = serializers.DateTimeField()

## [**Serializing objects**](https://www.django-rest-framework.org/api-guide/serializers/#serializing-objects)

We can now use CommentSerializer to serialize a comment, or list of comments. Again, using the Serializer class looks a lot like using a Form class.

* 1. Use the serializer to format the output of our object into a py dict

serializer = CommentSerializer(comment)

serializer.data

# {'email': 'leila@example.com', 'content': 'foo bar', 'created': '2016-01-27T15:17:10.375877'}

At this point we've translated the model instance into Python native datatypes. To finalise the serialization process we render the data into json.

* 1. Serialize the data into JSON

from rest\_framework.renderers import JSONRenderer

json = JSONRenderer().render(serializer.data)

json

# b'{"email":"leila@example.com","content":"foo bar","created":"2016-01-27T15:17:10.375877"}'

## [**Deserializing objects**](https://www.django-rest-framework.org/api-guide/serializers/#deserializing-objects)

Deserialization is similar. First we parse a stream into Python native datatypes...

import io

from rest\_framework.parsers import JSONParser

stream = io.BytesIO(json)

data = JSONParser().parse(stream)

...then we restore those native datatypes into a dictionary of validated data.

serializer = CommentSerializer(data=data)

serializer.is\_valid()

# True

serializer.validated\_data

# {'content': 'foo bar', 'email': 'leila@example.com', 'created': datetime.datetime(2012, 08, 22, 16, 20, 09, 822243)}

## [**Saving instances**](https://www.django-rest-framework.org/api-guide/serializers/#saving-instances)

If we want to be able to return complete object instances based on the validated data we need to implement one or both of the .create() and .update() methods. For example:

class CommentSerializer(serializers.Serializer):

email = serializers.EmailField()

content = serializers.CharField(max\_length=200)

created = serializers.DateTimeField()

def create(self, validated\_data):

return Comment(\*\*validated\_data)

def update(self, instance, validated\_data):

instance.email = validated\_data.get('email', instance.email)

instance.content = validated\_data.get('content', instance.content)

instance.created = validated\_data.get('created', instance.created)

return instance

If your object instances correspond to Django models you'll also want to ensure that these methods save the object to the database. For example, if Comment was a Django model, the methods might look like this:

def create(self, validated\_data):

return Comment.objects.create(\*\*validated\_data)

def update(self, instance, validated\_data):

instance.email = validated\_data.get('email', instance.email)

instance.content = validated\_data.get('content', instance.content)

instance.created = validated\_data.get('created', instance.created)

instance.save()

return instance

Now when deserializing data, we can call .save() to return an object instance, based on the validated data.

comment = serializer.save()

Calling .save() will either create a new instance, or update an existing instance, depending on if an existing instance was passed when instantiating the serializer class:

# .save() will create a new instance.

serializer = CommentSerializer(data=data)

# .save() will update the existing `comment` instance.

serializer = CommentSerializer(comment, data=data)

Both the .create() and .update() methods are optional. You can implement either neither, one, or both of them, depending on the use-case for your serializer class.

#### [**Passing additional attributes to .save()**](https://www.django-rest-framework.org/api-guide/serializers/#passing-additional-attributes-to-save)

Sometimes you'll want your view code to be able to inject additional data at the point of saving the instance. This additional data might include information like the current user, the current time, or anything else that is not part of the request data.

You can do so by including additional keyword arguments when calling .save(). For example:

serializer.save(owner=request.user)

Any additional keyword arguments will be included in the validated\_data argument when .create() or .update() are called.

#### [**Overriding .save() directly.**](https://www.django-rest-framework.org/api-guide/serializers/#overriding-save-directly)

In some cases the .create() and .update() method names may not be meaningful. For example, in a contact form we may not be creating new instances, but instead sending an email or other message.

In these cases you might instead choose to override .save() directly, as being more readable and meaningful.

For example:

class ContactForm(serializers.Serializer):

email = serializers.EmailField()

message = serializers.CharField()

def save(self):

email = self.validated\_data['email']

message = self.validated\_data['message']

send\_email(from=email, message=message)

Note that in the case above we're now having to access the serializer .validated\_data property directly.

## [**Validation**](https://www.django-rest-framework.org/api-guide/serializers/#validation)

When deserializing data, you always need to call is\_valid() before attempting to access the validated data, or save an object instance. If any validation errors occur, the .errors property will contain a dictionary representing the resulting error messages. For example:

serializer = CommentSerializer(data={'email': 'foobar', 'content': 'baz'})

serializer.is\_valid()

# False

serializer.errors

# {'email': ['Enter a valid e-mail address.'], 'created': ['This field is required.']}

Each key in the dictionary will be the field name, and the values will be lists of strings of any error messages corresponding to that field. The non\_field\_errors key may also be present, and will list any general validation errors. The name of the non\_field\_errors key may be customized using the NON\_FIELD\_ERRORS\_KEY REST framework setting.

When deserializing a list of items, errors will be returned as a list of dictionaries representing each of the deserialized items.

#### [**Raising an exception on invalid data**](https://www.django-rest-framework.org/api-guide/serializers/#raising-an-exception-on-invalid-data)

The .is\_valid() method takes an optional raise\_exception flag that will cause it to raise a serializers.ValidationErrorexception if there are validation errors.

These exceptions are automatically dealt with by the default exception handler that REST framework provides, and will return HTTP 400 Bad Request responses by default.

# Return a 400 response if the data was invalid.

serializer.is\_valid(raise\_exception=True)

#### [**Field-level validation**](https://www.django-rest-framework.org/api-guide/serializers/#field-level-validation)

You can specify custom field-level validation by adding .validate\_<field\_name> methods to your Serializer subclass. These are similar to the .clean\_<field\_name> methods on Django forms.

These methods take a single argument, which is the field value that requires validation.

Your validate\_<field\_name> methods should return the validated value or raise a serializers.ValidationError. For example:

from rest\_framework import serializers

class BlogPostSerializer(serializers.Serializer):

title = serializers.CharField(max\_length=100)

content = serializers.CharField()

def validate\_title(self, value):

"""

Check that the blog post is about Django.

"""

if 'django' not in value.lower():

raise serializers.ValidationError("Blog post is not about Django")

return value

**Note:** If your <field\_name> is declared on your serializer with the parameter required=False then this validation step will not take place if the field is not included.

#### [**Object-level validation**](https://www.django-rest-framework.org/api-guide/serializers/#object-level-validation)

To do any other validation that requires access to multiple fields, add a method called .validate() to your Serializersubclass. This method takes a single argument, which is a dictionary of field values. It should raise a serializers.ValidationError if necessary, or just return the validated values. For example:

from rest\_framework import serializers

class EventSerializer(serializers.Serializer):

description = serializers.CharField(max\_length=100)

start = serializers.DateTimeField()

finish = serializers.DateTimeField()

def validate(self, data):

"""

Check that start is before finish.

"""

if data['start'] > data['finish']:

raise serializers.ValidationError("finish must occur after start")

return data

#### [**Validators**](https://www.django-rest-framework.org/api-guide/serializers/#validators)

Individual fields on a serializer can include validators, by declaring them on the field instance, for example:

def multiple\_of\_ten(value):

if value % 10 != 0:

raise serializers.ValidationError('Not a multiple of ten')

class GameRecord(serializers.Serializer):

score = IntegerField(validators=[multiple\_of\_ten])

...

Serializer classes can also include reusable validators that are applied to the complete set of field data. These validators are included by declaring them on an inner Meta class, like so:

class EventSerializer(serializers.Serializer):

name = serializers.CharField()

room\_number = serializers.IntegerField(choices=[101, 102, 103, 201])

date = serializers.DateField()

class Meta:

# Each room only has one event per day.

validators = UniqueTogetherValidator(

queryset=Event.objects.all(),

fields=['room\_number', 'date']

)

For more information see the [validators documentation](https://www.django-rest-framework.org/api-guide/validators/).

## [**Accessing the initial data and instance**](https://www.django-rest-framework.org/api-guide/serializers/#accessing-the-initial-data-and-instance)

When passing an initial object or queryset to a serializer instance, the object will be made available as .instance. If no initial object is passed then the .instance attribute will be None.

When passing data to a serializer instance, the unmodified data will be made available as .initial\_data. If the data keyword argument is not passed then the .initial\_data attribute will not exist.

## [**Partial updates**](https://www.django-rest-framework.org/api-guide/serializers/#partial-updates)

By default, serializers must be passed values for all required fields or they will raise validation errors. You can use the partialargument in order to allow partial updates.

# Update `comment` with partial data

serializer = CommentSerializer(comment, data={'content': 'foo bar'}, partial=True)

## [**Dealing with nested objects**](https://www.django-rest-framework.org/api-guide/serializers/#dealing-with-nested-objects)

The previous examples are fine for dealing with objects that only have simple datatypes, but sometimes we also need to be able to represent more complex objects, where some of the attributes of an object might not be simple datatypes such as strings, dates or integers.

The Serializer class is itself a type of Field, and can be used to represent relationships where one object type is nested inside another.

class UserSerializer(serializers.Serializer):

email = serializers.EmailField()

username = serializers.CharField(max\_length=100)

class CommentSerializer(serializers.Serializer):

user = UserSerializer()

content = serializers.CharField(max\_length=200)

created = serializers.DateTimeField()

If a nested representation may optionally accept the None value you should pass the required=False flag to the nested serializer.

class CommentSerializer(serializers.Serializer):

user = UserSerializer(required=False) # May be an anonymous user.

content = serializers.CharField(max\_length=200)

created = serializers.DateTimeField()

Similarly if a nested representation should be a list of items, you should pass the many=True flag to the nested serialized.

class CommentSerializer(serializers.Serializer):

user = UserSerializer(required=False)

edits = EditItemSerializer(many=True) # A nested list of 'edit' items.

content = serializers.CharField(max\_length=200)

created = serializers.DateTimeField()

## [**Writable nested representations**](https://www.django-rest-framework.org/api-guide/serializers/#writable-nested-representations)

When dealing with nested representations that support deserializing the data, any errors with nested objects will be nested under the field name of the nested object.

serializer = CommentSerializer(data={'user': {'email': 'foobar', 'username': 'doe'}, 'content': 'baz'})

serializer.is\_valid()

# False

serializer.errors

# {'user': {'email': ['Enter a valid e-mail address.']}, 'created': ['This field is required.']}

Similarly, the .validated\_data property will include nested data structures.

#### [**Writing .create() methods for nested representations**](https://www.django-rest-framework.org/api-guide/serializers/#writing-create-methods-for-nested-representations)

If you're supporting writable nested representations you'll need to write .create() or .update() methods that handle saving multiple objects.

The following example demonstrates how you might handle creating a user with a nested profile object.

class UserSerializer(serializers.ModelSerializer):

profile = ProfileSerializer()

class Meta:

model = User

fields = ['username', 'email', 'profile']

def create(self, validated\_data):

profile\_data = validated\_data.pop('profile')

user = User.objects.create(\*\*validated\_data)

Profile.objects.create(user=user, \*\*profile\_data)

return user

#### [**Writing .update() methods for nested representations**](https://www.django-rest-framework.org/api-guide/serializers/#writing-update-methods-for-nested-representations)

For updates you'll want to think carefully about how to handle updates to relationships. For example if the data for the relationship is None, or not provided, which of the following should occur?

* Set the relationship to NULL in the database.
* Delete the associated instance.
* Ignore the data and leave the instance as it is.
* Raise a validation error.

Here's an example for an .update() method on our previous UserSerializer class.

def update(self, instance, validated\_data):

profile\_data = validated\_data.pop('profile')

# Unless the application properly enforces that this field is

# always set, the follow could raise a `DoesNotExist`, which

# would need to be handled.

profile = instance.profile

instance.username = validated\_data.get('username', instance.username)

instance.email = validated\_data.get('email', instance.email)

instance.save()

profile.is\_premium\_member = profile\_data.get(

'is\_premium\_member',

profile.is\_premium\_member

)

profile.has\_support\_contract = profile\_data.get(

'has\_support\_contract',

profile.has\_support\_contract

)

profile.save()

return instance

Because the behavior of nested creates and updates can be ambiguous, and may require complex dependencies between related models, REST framework 3 requires you to always write these methods explicitly. The default ModelSerializer.create() and .update() methods do not include support for writable nested representations.

There are however, third-party packages available such as [DRF Writable Nested](https://www.django-rest-framework.org/api-guide/serializers/#drf-writable-nested) that support automatic writable nested representations.

#### [**Handling saving related instances in model manager classes**](https://www.django-rest-framework.org/api-guide/serializers/#handling-saving-related-instances-in-model-manager-classes)

An alternative to saving multiple related instances in the serializer is to write custom model manager classes that handle creating the correct instances.

For example, suppose we wanted to ensure that User instances and Profile instances are always created together as a pair. We might write a custom manager class that looks something like this:

class UserManager(models.Manager):

...

def create(self, username, email, is\_premium\_member=False, has\_support\_contract=False):

user = User(username=username, email=email)

user.save()

profile = Profile(

user=user,

is\_premium\_member=is\_premium\_member,

has\_support\_contract=has\_support\_contract

)

profile.save()

return user

This manager class now more nicely encapsulates that user instances and profile instances are always created at the same time. Our .create() method on the serializer class can now be re-written to use the new manager method.

def create(self, validated\_data):

return User.objects.create(

username=validated\_data['username'],

email=validated\_data['email']

is\_premium\_member=validated\_data['profile']['is\_premium\_member']

has\_support\_contract=validated\_data['profile']['has\_support\_contract']

)

For more details on this approach see the Django documentation on [model managers](https://docs.djangoproject.com/en/stable/topics/db/managers/), and [this blogpost on using model and manager classes](https://www.dabapps.com/blog/django-models-and-encapsulation/).

## [**Dealing with multiple objects**](https://www.django-rest-framework.org/api-guide/serializers/#dealing-with-multiple-objects)

The Serializer class can also handle serializing or deserializing lists of objects.

#### [**Serializing multiple objects**](https://www.django-rest-framework.org/api-guide/serializers/#serializing-multiple-objects)

To serialize a queryset or list of objects instead of a single object instance, you should pass the many=True flag when instantiating the serializer. You can then pass a queryset or list of objects to be serialized.

queryset = Book.objects.all()

serializer = BookSerializer(queryset, many=True)

serializer.data

# [

# {'id': 0, 'title': 'The electric kool-aid acid test', 'author': 'Tom Wolfe'},

# {'id': 1, 'title': 'If this is a man', 'author': 'Primo Levi'},

# {'id': 2, 'title': 'The wind-up bird chronicle', 'author': 'Haruki Murakami'}

# ]

#### [**Deserializing multiple objects**](https://www.django-rest-framework.org/api-guide/serializers/#deserializing-multiple-objects)

The default behavior for deserializing multiple objects is to support multiple object creation, but not support multiple object updates. For more information on how to support or customize either of these cases, see the [ListSerializer](https://www.django-rest-framework.org/api-guide/serializers/#listserializer) documentation below.

## [**Including extra context**](https://www.django-rest-framework.org/api-guide/serializers/#including-extra-context)

There are some cases where you need to provide extra context to the serializer in addition to the object being serialized. One common case is if you're using a serializer that includes hyperlinked relations, which requires the serializer to have access to the current request so that it can properly generate fully qualified URLs.

You can provide arbitrary additional context by passing a context argument when instantiating the serializer. For example:

serializer = AccountSerializer(account, context={'request': request})

serializer.data

# {'id': 6, 'owner': 'denvercoder9', 'created': datetime.datetime(2013, 2, 12, 09, 44, 56, 678870), 'details': 'http://example.com/accounts/6/details'}

The context dictionary can be used within any serializer field logic, such as a custom .to\_representation() method, by accessing the self.context attribute.

# [**ModelSerializer**](https://www.django-rest-framework.org/api-guide/serializers/#modelserializer)

Often you'll want serializer classes that map closely to Django model definitions.

The ModelSerializer class provides a shortcut that lets you automatically create a Serializer class with fields that correspond to the Model fields.

**The ModelSerializer class is the same as a regular Serializer class, except that**:

* It will automatically generate a set of fields for you, based on the model.
* It will automatically generate validators for the serializer, such as unique\_together validators.
* It includes simple default implementations of .create() and .update().

Declaring a ModelSerializer looks like this:

class AccountSerializer(serializers.ModelSerializer):

class Meta:

model = Account

fields = ['id', 'account\_name', 'users', 'created']

By default, all the model fields on the class will be mapped to a corresponding serializer fields.

Any relationships such as foreign keys on the model will be mapped to PrimaryKeyRelatedField. Reverse relationships are not included by default unless explicitly included as specified in the [serializer relations](https://www.django-rest-framework.org/api-guide/relations/) documentation.

#### [**Inspecting a ModelSerializer**](https://www.django-rest-framework.org/api-guide/serializers/#inspecting-a-modelserializer)

Serializer classes generate helpful verbose representation strings, that allow you to fully inspect the state of their fields. This is particularly useful when working with ModelSerializers where you want to determine what set of fields and validators are being automatically created for you.

To do so, open the Django shell, using python manage.py shell, then import the serializer class, instantiate it, and print the object representation…

>>> from myapp.serializers import AccountSerializer

>>> serializer = AccountSerializer()

>>> print(repr(serializer))

AccountSerializer():

id = IntegerField(label='ID', read\_only=True)

name = CharField(allow\_blank=True, max\_length=100, required=False)

owner = PrimaryKeyRelatedField(queryset=User.objects.all())

## [**Specifying which fields to include**](https://www.django-rest-framework.org/api-guide/serializers/#specifying-which-fields-to-include)

If you only want a subset of the default fields to be used in a model serializer, you can do so using fields or exclude options, just as you would with a ModelForm. It is strongly recommended that you explicitly set all fields that should be serialized using the fields attribute. This will make it less likely to result in unintentionally exposing data when your models change.

For example:

class AccountSerializer(serializers.ModelSerializer):

class Meta:

model = Account

fields = ['id', 'account\_name', 'users', 'created']

You can also set the fields attribute to the special value '\_\_all\_\_' to indicate that all fields in the model should be used.

For example:

class AccountSerializer(serializers.ModelSerializer):

class Meta:

model = Account

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

You can set the exclude attribute to a list of fields to be excluded from the serializer.

For example:

class AccountSerializer(serializers.ModelSerializer):

class Meta:

model = Account

exclude = ['users']

In the example above, if the Account model had 3 fields account\_name, users, and created, this will result in the fields account\_name and created to be serialized.

The names in the fields and exclude attributes will normally map to model fields on the model class.

Alternatively names in the fields options can map to properties or methods which take no arguments that exist on the model class.

Since version 3.3.0, it is **mandatory** to provide one of the attributes fields or exclude.

## [**Specifying nested serialization**](https://www.django-rest-framework.org/api-guide/serializers/#specifying-nested-serialization)

The default ModelSerializer uses primary keys for relationships, but you can also easily generate nested representations using the depth option:

class AccountSerializer(serializers.ModelSerializer):

class Meta:

model = Account

fields = ['id', 'account\_name', 'users', 'created']

depth = 1

The depth option should be set to an integer value that indicates the depth of relationships that should be traversed before reverting to a flat representation.

If you want to customize the way the serialization is done you'll need to define the field yourself.

## [**Specifying fields explicitly**](https://www.django-rest-framework.org/api-guide/serializers/#specifying-fields-explicitly)

You can add extra fields to a ModelSerializer or override the default fields by declaring fields on the class, just as you would for a Serializer class.

class AccountSerializer(serializers.ModelSerializer):

url = serializers.CharField(source='get\_absolute\_url', read\_only=True)

groups = serializers.PrimaryKeyRelatedField(many=True)

class Meta:

model = Account

Extra fields can correspond to any property or callable on the model.

## [**Specifying read only fields**](https://www.django-rest-framework.org/api-guide/serializers/#specifying-read-only-fields)

You may wish to specify multiple fields as read-only. Instead of adding each field explicitly with the read\_only=True attribute, you may use the shortcut Meta option, read\_only\_fields.

This option should be a list or tuple of field names, and is declared as follows:

class AccountSerializer(serializers.ModelSerializer):

class Meta:

model = Account

fields = ['id', 'account\_name', 'users', 'created']

read\_only\_fields = ['account\_name']

Model fields which have editable=False set, and AutoField fields will be set to read-only by default, and do not need to be added to the read\_only\_fields option.

**Note**: There is a special-case where a read-only field is part of a unique\_together constraint at the model level. In this case the field is required by the serializer class in order to validate the constraint, but should also not be editable by the user.

The right way to deal with this is to specify the field explicitly on the serializer, providing both the read\_only=True and default=… keyword arguments.

One example of this is a read-only relation to the currently authenticated User which is unique\_together with another identifier. In this case you would declare the user field like so:

user = serializers.PrimaryKeyRelatedField(read\_only=True, default=serializers.CurrentUserDefault())

Please review the [Validators Documentation](https://www.django-rest-framework.org/api-guide/validators/) for details on the [UniqueTogetherValidator](https://www.django-rest-framework.org/api-guide/validators/#uniquetogethervalidator) and [CurrentUserDefault](https://www.django-rest-framework.org/api-guide/validators/#currentuserdefault) classes.

## [**Additional keyword arguments**](https://www.django-rest-framework.org/api-guide/serializers/#additional-keyword-arguments)

There is also a shortcut allowing you to specify arbitrary additional keyword arguments on fields, using the extra\_kwargs option. As in the case of read\_only\_fields, this means you do not need to explicitly declare the field on the serializer.

This option is a dictionary, mapping field names to a dictionary of keyword arguments. For example:

class CreateUserSerializer(serializers.ModelSerializer):

class Meta:

model = User

fields = ['email', 'username', 'password']

extra\_kwargs = {'password': {'write\_only': True}}

def create(self, validated\_data):

user = User(

email=validated\_data['email'],

username=validated\_data['username']

)

user.set\_password(validated\_data['password'])

user.save()

return user

Please keep in mind that, if the field has already been explicitly declared on the serializer class, then the extra\_kwargs option will be ignored.

## [**Relational fields**](https://www.django-rest-framework.org/api-guide/serializers/#relational-fields)

When serializing model instances, there are a number of different ways you might choose to represent relationships. The default representation for ModelSerializer is to use the primary keys of the related instances.

Alternative representations include serializing using hyperlinks, serializing complete nested representations, or serializing with a custom representation.

For full details see the [serializer relations](https://www.django-rest-framework.org/api-guide/relations/) documentation.

## [**Customizing field mappings**](https://www.django-rest-framework.org/api-guide/serializers/#customizing-field-mappings)

The ModelSerializer class also exposes an API that you can override in order to alter how serializer fields are automatically determined when instantiating the serializer.

Normally if a ModelSerializer does not generate the fields you need by default then you should either add them to the class explicitly, or simply use a regular Serializer class instead. However in some cases you may want to create a new base class that defines how the serializer fields are created for any given model.

### [**.serializer\_field\_mapping**](https://www.django-rest-framework.org/api-guide/serializers/#serializer_field_mapping)

A mapping of Django model classes to REST framework serializer classes. You can override this mapping to alter the default serializer classes that should be used for each model class.

### [**.serializer\_related\_field**](https://www.django-rest-framework.org/api-guide/serializers/#serializer_related_field)

This property should be the serializer field class, that is used for relational fields by default.

For ModelSerializer this defaults to PrimaryKeyRelatedField.

For HyperlinkedModelSerializer this defaults to serializers.HyperlinkedRelatedField.

### [**serializer\_url\_field**](https://www.django-rest-framework.org/api-guide/serializers/#serializer_url_field)

The serializer field class that should be used for any url field on the serializer.

Defaults to serializers.HyperlinkedIdentityField

### [**serializer\_choice\_field**](https://www.django-rest-framework.org/api-guide/serializers/#serializer_choice_field)

The serializer field class that should be used for any choice fields on the serializer.

Defaults to serializers.ChoiceField

### [**The field\_class and field\_kwargs API**](https://www.django-rest-framework.org/api-guide/serializers/#the-field_class-and-field_kwargs-api)

The following methods are called to determine the class and keyword arguments for each field that should be automatically included on the serializer. Each of these methods should return a two tuple of (field\_class, field\_kwargs).

### [**.build\_standard\_field(self, field\_name, model\_field)**](https://www.django-rest-framework.org/api-guide/serializers/#build_standard_fieldself-field_name-model_field)

Called to generate a serializer field that maps to a standard model field.

The default implementation returns a serializer class based on the serializer\_field\_mapping attribute.

### [**.build\_relational\_field(self, field\_name, relation\_info)**](https://www.django-rest-framework.org/api-guide/serializers/#build_relational_fieldself-field_name-relation_info)

Called to generate a serializer field that maps to a relational model field.

The default implementation returns a serializer class based on the serializer\_related\_field attribute.

The relation\_info argument is a named tuple, that contains model\_field, related\_model, to\_many and has\_through\_model properties.

### [**.build\_nested\_field(self, field\_name, relation\_info, nested\_depth)**](https://www.django-rest-framework.org/api-guide/serializers/#build_nested_fieldself-field_name-relation_info-nested_depth)

Called to generate a serializer field that maps to a relational model field, when the depth option has been set.

The default implementation dynamically creates a nested serializer class based on either ModelSerializer or HyperlinkedModelSerializer.

The nested\_depth will be the value of the depth option, minus one.

The relation\_info argument is a named tuple, that contains model\_field, related\_model, to\_many and has\_through\_model properties.

### [**.build\_property\_field(self, field\_name, model\_class)**](https://www.django-rest-framework.org/api-guide/serializers/#build_property_fieldself-field_name-model_class)

Called to generate a serializer field that maps to a property or zero-argument method on the model class.

The default implementation returns a ReadOnlyField class.

### [**.build\_url\_field(self, field\_name, model\_class)**](https://www.django-rest-framework.org/api-guide/serializers/#build_url_fieldself-field_name-model_class)

Called to generate a serializer field for the serializer's own url field. The default implementation returns a HyperlinkedIdentityField class.

### [**.build\_unknown\_field(self, field\_name, model\_class)**](https://www.django-rest-framework.org/api-guide/serializers/#build_unknown_fieldself-field_name-model_class)

Called when the field name did not map to any model field or model property. The default implementation raises an error, although subclasses may customize this behavior.

# [**HyperlinkedModelSerializer**](https://www.django-rest-framework.org/api-guide/serializers/#hyperlinkedmodelserializer)

The HyperlinkedModelSerializer class is similar to the ModelSerializer class except that it uses hyperlinks to represent relationships, rather than primary keys.

By default the serializer will include a url field instead of a primary key field.

The url field will be represented using a HyperlinkedIdentityField serializer field, and any relationships on the model will be represented using a HyperlinkedRelatedField serializer field.

You can explicitly include the primary key by adding it to the fields option, for example:

class AccountSerializer(serializers.HyperlinkedModelSerializer):

class Meta:

model = Account

fields = ['url', 'id', 'account\_name', 'users', 'created']

## [**Absolute and relative URLs**](https://www.django-rest-framework.org/api-guide/serializers/#absolute-and-relative-urls)

When instantiating a HyperlinkedModelSerializer you must include the current request in the serializer context, for example:

serializer = AccountSerializer(queryset, context={'request': request})

Doing so will ensure that the hyperlinks can include an appropriate hostname, so that the resulting representation uses fully qualified URLs, such as:

http://api.example.com/accounts/1/

Rather than relative URLs, such as:

/accounts/1/

If you do want to use relative URLs, you should explicitly pass {'request': None} in the serializer context.

## [**How hyperlinked views are determined**](https://www.django-rest-framework.org/api-guide/serializers/#how-hyperlinked-views-are-determined)

There needs to be a way of determining which views should be used for hyperlinking to model instances.

By default hyperlinks are expected to correspond to a view name that matches the style '{model\_name}-detail', and looks up the instance by a pk keyword argument.

You can override a URL field view name and lookup field by using either, or both of, the view\_name and lookup\_field options in the extra\_kwargs setting, like so:

class AccountSerializer(serializers.HyperlinkedModelSerializer):

class Meta:

model = Account

fields = ['account\_url', 'account\_name', 'users', 'created']

extra\_kwargs = {

'url': {'view\_name': 'accounts', 'lookup\_field': 'account\_name'},

'users': {'lookup\_field': 'username'}

}

Alternatively you can set the fields on the serializer explicitly. For example:

class AccountSerializer(serializers.HyperlinkedModelSerializer):

url = serializers.HyperlinkedIdentityField(

view\_name='accounts',

lookup\_field='slug'

)

users = serializers.HyperlinkedRelatedField(

view\_name='user-detail',

lookup\_field='username',

many=True,

read\_only=True

)

class Meta:

model = Account

fields = ['url', 'account\_name', 'users', 'created']

**Tip**: Properly matching together hyperlinked representations and your URL conf can sometimes be a bit fiddly. Printing the reprof a HyperlinkedModelSerializer instance is a particularly useful way to inspect exactly which view names and lookup fields the relationships are expected to map too.

## [**Changing the URL field name**](https://www.django-rest-framework.org/api-guide/serializers/#changing-the-url-field-name)

The name of the URL field defaults to 'url'. You can override this globally, by using the URL\_FIELD\_NAME setting.

# [**ListSerializer**](https://www.django-rest-framework.org/api-guide/serializers/#listserializer)

The ListSerializer class provides the behavior for serializing and validating multiple objects at once. You won't typically need to use ListSerializer directly, but should instead simply pass many=True when instantiating a serializer.

When a serializer is instantiated and many=True is passed, a ListSerializer instance will be created. The serializer class then becomes a child of the parent ListSerializer

The following argument can also be passed to a ListSerializer field or a serializer that is passed many=True:

### [**allow\_empty**](https://www.django-rest-framework.org/api-guide/serializers/#allow_empty)

This is True by default, but can be set to False if you want to disallow empty lists as valid input.

### [**Customizing ListSerializer behavior**](https://www.django-rest-framework.org/api-guide/serializers/#customizing-listserializer-behavior)

There are a few use cases when you might want to customize the ListSerializer behavior. For example:

* You want to provide particular validation of the lists, such as checking that one element does not conflict with another element in a list.
* You want to customize the create or update behavior of multiple objects.

For these cases you can modify the class that is used when many=True is passed, by using the list\_serializer\_class option on the serializer Meta class.

For example:

class CustomListSerializer(serializers.ListSerializer):

...

class CustomSerializer(serializers.Serializer):

...

class Meta:

list\_serializer\_class = CustomListSerializer

#### [**Customizing multiple create**](https://www.django-rest-framework.org/api-guide/serializers/#customizing-multiple-create)

The default implementation for multiple object creation is to simply call .create() for each item in the list. If you want to customize this behavior, you'll need to customize the .create() method on ListSerializer class that is used when many=True is passed.

For example:

class BookListSerializer(serializers.ListSerializer):

def create(self, validated\_data):

books = [Book(\*\*item) for item in validated\_data]

return Book.objects.bulk\_create(books)

class BookSerializer(serializers.Serializer):

...

class Meta:

list\_serializer\_class = BookListSerializer

#### [**Customizing multiple update**](https://www.django-rest-framework.org/api-guide/serializers/#customizing-multiple-update)

By default the ListSerializer class does not support multiple updates. This is because the behavior that should be expected for insertions and deletions is ambiguous.

To support multiple updates you'll need to do so explicitly. When writing your multiple update code make sure to keep the following in mind:

* How do you determine which instance should be updated for each item in the list of data?
* How should insertions be handled? Are they invalid, or do they create new objects?
* How should removals be handled? Do they imply object deletion, or removing a relationship? Should they be silently ignored, or are they invalid?
* How should ordering be handled? Does changing the position of two items imply any state change or is it ignored?

You will need to add an explicit id field to the instance serializer. The default implicitly-generated id field is marked as read\_only. This causes it to be removed on updates. Once you declare it explicitly, it will be available in the list serializer's update method.

Here's an example of how you might choose to implement multiple updates:

class BookListSerializer(serializers.ListSerializer):

def update(self, instance, validated\_data):

# Maps for id->instance and id->data item.

book\_mapping = {book.id: book for book in instance}

data\_mapping = {item['id']: item for item in validated\_data}

# Perform creations and updates.

ret = []

for book\_id, data in data\_mapping.items():

book = book\_mapping.get(book\_id, None)

if book is None:

ret.append(self.child.create(data))

else:

ret.append(self.child.update(book, data))

# Perform deletions.

for book\_id, book in book\_mapping.items():

if book\_id not in data\_mapping:

book.delete()

return ret

class BookSerializer(serializers.Serializer):

# We need to identify elements in the list using their primary key,

# so use a writable field here, rather than the default which would be read-only.

id = serializers.IntegerField()

...

class Meta:

list\_serializer\_class = BookListSerializer

It is possible that a third party package may be included alongside the 3.1 release that provides some automatic support for multiple update operations, similar to the allow\_add\_remove behavior that was present in REST framework 2.

#### [**Customizing ListSerializer initialization**](https://www.django-rest-framework.org/api-guide/serializers/#customizing-listserializer-initialization)

When a serializer with many=True is instantiated, we need to determine which arguments and keyword arguments should be passed to the .\_\_init\_\_() method for both the child Serializer class, and for the parent ListSerializer class.

The default implementation is to pass all arguments to both classes, except for validators, and any custom keyword arguments, both of which are assumed to be intended for the child serializer class.

Occasionally you might need to explicitly specify how the child and parent classes should be instantiated when many=True is passed. You can do so by using the many\_init class method.

@classmethod

def many\_init(cls, \*args, \*\*kwargs):

# Instantiate the child serializer.

kwargs['child'] = cls()

# Instantiate the parent list serializer.

return CustomListSerializer(\*args, \*\*kwargs)

# [**BaseSerializer**](https://www.django-rest-framework.org/api-guide/serializers/#baseserializer)

BaseSerializer class that can be used to easily support alternative serialization and deserialization styles.

This class implements the same basic API as the Serializer class:

* .data - Returns the outgoing primitive representation.
* .is\_valid() - Deserializes and validates incoming data.
* .validated\_data - Returns the validated incoming data.
* .errors - Returns any errors during validation.
* .save() - Persists the validated data into an object instance.

There are four methods that can be overridden, depending on what functionality you want the serializer class to support:

* .to\_representation() - Override this to support serialization, for read operations.
* .to\_internal\_value() - Override this to support deserialization, for write operations.
* .create() and .update() - Override either or both of these to support saving instances.

Because this class provides the same interface as the Serializer class, you can use it with the existing generic class-based views exactly as you would for a regular Serializer or ModelSerializer.

The only difference you'll notice when doing so is the BaseSerializer classes will not generate HTML forms in the browsable API. This is because the data they return does not include all the field information that would allow each field to be rendered into a suitable HTML input.

##### [Read-only BaseSerializer classes](https://www.django-rest-framework.org/api-guide/serializers/#read-only-baseserializer-classes)

To implement a read-only serializer using the BaseSerializer class, we just need to override the .to\_representation()method. Let's take a look at an example using a simple Django model:

class HighScore(models.Model):

created = models.DateTimeField(auto\_now\_add=True)

player\_name = models.CharField(max\_length=10)

score = models.IntegerField()

It's simple to create a read-only serializer for converting HighScore instances into primitive data types.

class HighScoreSerializer(serializers.BaseSerializer):

def to\_representation(self, obj):

return {

'score': obj.score,

'player\_name': obj.player\_name

}

We can now use this class to serialize single HighScore instances:

@api\_view(['GET'])

def high\_score(request, pk):

instance = HighScore.objects.get(pk=pk)

serializer = HighScoreSerializer(instance)

return Response(serializer.data)

Or use it to serialize multiple instances:

@api\_view(['GET'])

def all\_high\_scores(request):

queryset = HighScore.objects.order\_by('-score')

serializer = HighScoreSerializer(queryset, many=True)

return Response(serializer.data)

##### [Read-write BaseSerializer classes](https://www.django-rest-framework.org/api-guide/serializers/#read-write-baseserializer-classes)

To create a read-write serializer we first need to implement a .to\_internal\_value() method. This method returns the validated values that will be used to construct the object instance, and may raise a serializers.ValidationError if the supplied data is in an incorrect format.

Once you've implemented .to\_internal\_value(), the basic validation API will be available on the serializer, and you will be able to use .is\_valid(), .validated\_data and .errors.

If you want to also support .save() you'll need to also implement either or both of the .create() and .update() methods.

Here's a complete example of our previous HighScoreSerializer, that's been updated to support both read and write operations.

class HighScoreSerializer(serializers.BaseSerializer):

def to\_internal\_value(self, data):

score = data.get('score')

player\_name = data.get('player\_name')

# Perform the data validation.

if not score:

raise serializers.ValidationError({

'score': 'This field is required.'

})

if not player\_name:

raise serializers.ValidationError({

'player\_name': 'This field is required.'

})

if len(player\_name) > 10:

raise serializers.ValidationError({

'player\_name': 'May not be more than 10 characters.'

})

# Return the validated values. This will be available as

# the `.validated\_data` property.

return {

'score': int(score),

'player\_name': player\_name

}

def to\_representation(self, obj):

return {

'score': obj.score,

'player\_name': obj.player\_name

}

def create(self, validated\_data):

return HighScore.objects.create(\*\*validated\_data)

#### [**Creating new base classes**](https://www.django-rest-framework.org/api-guide/serializers/#creating-new-base-classes)

The BaseSerializer class is also useful if you want to implement new generic serializer classes for dealing with particular serialization styles, or for integrating with alternative storage backends.

The following class is an example of a generic serializer that can handle coercing arbitrary objects into primitive representations.

class ObjectSerializer(serializers.BaseSerializer):

"""

A read-only serializer that coerces arbitrary complex objects

into primitive representations.

"""

def to\_representation(self, obj):

output = {}

for attribute\_name in dir(obj):

attribute = getattr(obj, attribute\_name)

if attribute\_name.startswith('\_'):

# Ignore private attributes.

pass

elif hasattr(attribute, '\_\_call\_\_'):

# Ignore methods and other callables.

pass

elif isinstance(attribute, (str, int, bool, float, type(None))):

# Primitive types can be passed through unmodified.

output[attribute\_name] = attribute

elif isinstance(attribute, list):

# Recursively deal with items in lists.

output[attribute\_name] = [

self.to\_representation(item) for item in attribute

]

elif isinstance(attribute, dict):

# Recursively deal with items in dictionaries.

output[attribute\_name] = {

str(key): self.to\_representation(value)

for key, value in attribute.items()

}

else:

# Force anything else to its string representation.

output[attribute\_name] = str(attribute)

return output

# [**Advanced serializer usage**](https://www.django-rest-framework.org/api-guide/serializers/#advanced-serializer-usage)

## [**Overriding serialization and deserialization behavior**](https://www.django-rest-framework.org/api-guide/serializers/#overriding-serialization-and-deserialization-behavior)

If you need to alter the serialization or deserialization behavior of a serializer class, you can do so by overriding the .to\_representation() or .to\_internal\_value() methods.

Some reasons this might be useful include...

* Adding new behavior for new serializer base classes.
* Modifying the behavior slightly for an existing class.
* Improving serialization performance for a frequently accessed API endpoint that returns lots of data.

The signatures for these methods are as follows:

#### [**.to\_representation(self, obj)**](https://www.django-rest-framework.org/api-guide/serializers/#to_representationself-obj)

Takes the object instance that requires serialization, and should return a primitive representation. Typically this means returning a structure of built-in Python datatypes. The exact types that can be handled will depend on the render classes you have configured for your API.

May be overridden in order modify the representation style. For example:

def to\_representation(self, instance):

"""Convert `username` to lowercase."""

ret = super().to\_representation(instance)

ret['username'] = ret['username'].lower()

return ret

#### [**.to\_internal\_value(self, data)**](https://www.django-rest-framework.org/api-guide/serializers/#to_internal_valueself-data)

Takes the unvalidated incoming data as input and should return the validated data that will be made available as serializer.validated\_data. The return value will also be passed to the .create() or .update() methods if .save() is called on the serializer class.

If any of the validation fails, then the method should raise a serializers.ValidationError(errors). The errors argument should be a dictionary mapping field names (or settings.NON\_FIELD\_ERRORS\_KEY) to a list of error messages. If you don't need to alter deserialization behavior and instead want to provide object-level validation, it's recommended that you instead override the [.validate()](https://www.django-rest-framework.org/api-guide/serializers/#object-level-validation) method.

The data argument passed to this method will normally be the value of request.data, so the datatype it provides will depend on the parser classes you have configured for your API.

## [**Serializer Inheritance**](https://www.django-rest-framework.org/api-guide/serializers/#serializer-inheritance)

Similar to Django forms, you can extend and reuse serializers through inheritance. This allows you to declare a common set of fields or methods on a parent class that can then be used in a number of serializers. For example,

class MyBaseSerializer(Serializer):

my\_field = serializers.CharField()

def validate\_my\_field(self, value):

...

class MySerializer(MyBaseSerializer):

...

Like Django's Model and ModelForm classes, the inner Meta class on serializers does not implicitly inherit from it's parents' inner Meta classes. If you want the Meta class to inherit from a parent class you must do so explicitly. For example:

class AccountSerializer(MyBaseSerializer):

class Meta(MyBaseSerializer.Meta):

model = Account

Typically we would recommend not using inheritance on inner Meta classes, but instead declaring all options explicitly.

Additionally, the following caveats apply to serializer inheritance:

* Normal Python name resolution rules apply. If you have multiple base classes that declare a Meta inner class, only the first one will be used. This means the child’s Meta, if it exists, otherwise the Meta of the first parent, etc.
* It’s possible to declaratively remove a Field inherited from a parent class by setting the name to be None on the subclass.
* class MyBaseSerializer(ModelSerializer):
* my\_field = serializers.CharField()
* class MySerializer(MyBaseSerializer):

my\_field = None

However, you can only use this technique to opt out from a field defined declaratively by a parent class; it won’t prevent the ModelSerializer from generating a default field. To opt-out from default fields, see [Specifying which fields to include](https://www.django-rest-framework.org/api-guide/serializers/#specifying-which-fields-to-include).

## [**Dynamically modifying fields**](https://www.django-rest-framework.org/api-guide/serializers/#dynamically-modifying-fields)

Once a serializer has been initialized, the dictionary of fields that are set on the serializer may be accessed using the .fieldsattribute. Accessing and modifying this attribute allows you to dynamically modify the serializer.

Modifying the fields argument directly allows you to do interesting things such as changing the arguments on serializer fields at runtime, rather than at the point of declaring the serializer.

### [**Example**](https://www.django-rest-framework.org/api-guide/serializers/#example)

For example, if you wanted to be able to set which fields should be used by a serializer at the point of initializing it, you could create a serializer class like so:

class DynamicFieldsModelSerializer(serializers.ModelSerializer):

"""

A ModelSerializer that takes an additional `fields` argument that

controls which fields should be displayed.

"""

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)

for field\_name in existing - allowed:

self.fields.pop(field\_name)

This would then allow you to do the following:

>>> class UserSerializer(DynamicFieldsModelSerializer):

>>> class Meta:

>>> model = User

>>> fields = ['id', 'username', 'email']

>>>

>>> print(UserSerializer(user))

{'id': 2, 'username': 'jonwatts', 'email': 'jon@example.com'}

>>>

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

{'id': 2, 'email': 'jon@example.com'}

## [**Customizing the default fields**](https://www.django-rest-framework.org/api-guide/serializers/#customizing-the-default-fields)

REST framework 2 provided an API to allow developers to override how a ModelSerializer class would automatically generate the default set of fields.

This API included the .get\_field(), .get\_pk\_field() and other methods.

Because the serializers have been fundamentally redesigned with 3.0 this API no longer exists. You can still modify the fields that get created but you'll need to refer to the source code, and be aware that if the changes you make are against private bits of API then they may be subject to change.

# [**Third party packages**](https://www.django-rest-framework.org/api-guide/serializers/#third-party-packages)

The following third party packages are also available.

## [**Django REST marshmallow**](https://www.django-rest-framework.org/api-guide/serializers/#django-rest-marshmallow)

The [django-rest-marshmallow](https://marshmallow-code.github.io/django-rest-marshmallow/) package provides an alternative implementation for serializers, using the python [marshmallow](https://marshmallow.readthedocs.io/en/latest/)library. It exposes the same API as the REST framework serializers, and can be used as a drop-in replacement in some use-cases.

## [**Serpy**](https://www.django-rest-framework.org/api-guide/serializers/#serpy)

The [serpy](https://github.com/clarkduvall/serpy) package is an alternative implementation for serializers that is built for speed. [Serpy](https://github.com/clarkduvall/serpy) serializes complex datatypes to simple native types. The native types can be easily converted to JSON or any other format needed.

## [**MongoengineModelSerializer**](https://www.django-rest-framework.org/api-guide/serializers/#mongoenginemodelserializer)

The [django-rest-framework-mongoengine](https://github.com/umutbozkurt/django-rest-framework-mongoengine) package provides a MongoEngineModelSerializer serializer class that supports using MongoDB as the storage layer for Django REST framework.

## [**GeoFeatureModelSerializer**](https://www.django-rest-framework.org/api-guide/serializers/#geofeaturemodelserializer)

The [django-rest-framework-gis](https://github.com/djangonauts/django-rest-framework-gis) package provides a GeoFeatureModelSerializer serializer class that supports GeoJSON both for read and write operations.

## [**HStoreSerializer**](https://www.django-rest-framework.org/api-guide/serializers/#hstoreserializer)

The [django-rest-framework-hstore](https://github.com/djangonauts/django-rest-framework-hstore) package provides an HStoreSerializer to support [django-hstore](https://github.com/djangonauts/django-hstore) DictionaryField model field and its schema-mode feature.

## [**Dynamic REST**](https://www.django-rest-framework.org/api-guide/serializers/#dynamic-rest)

The [dynamic-rest](https://github.com/AltSchool/dynamic-rest) package extends the ModelSerializer and ModelViewSet interfaces, adding API query parameters for filtering, sorting, and including / excluding all fields and relationships defined by your serializers.

## [**Dynamic Fields Mixin**](https://www.django-rest-framework.org/api-guide/serializers/#dynamic-fields-mixin)

The [drf-dynamic-fields](https://github.com/dbrgn/drf-dynamic-fields) package provides a mixin to dynamically limit the fields per serializer to a subset specified by an URL parameter.

## [**DRF FlexFields**](https://www.django-rest-framework.org/api-guide/serializers/#drf-flexfields)

The [drf-flex-fields](https://github.com/rsinger86/drf-flex-fields) package extends the ModelSerializer and ModelViewSet to provide commonly used functionality for dynamically setting fields and expanding primitive fields to nested models, both from URL parameters and your serializer class definitions.

## [**Serializer Extensions**](https://www.django-rest-framework.org/api-guide/serializers/#serializer-extensions)

The [django-rest-framework-serializer-extensions](https://github.com/evenicoulddoit/django-rest-framework-serializer-extensions) package provides a collection of tools to DRY up your serializers, by allowing fields to be defined on a per-view/request basis. Fields can be whitelisted, blacklisted and child serializers can be optionally expanded.

## [**HTML JSON Forms**](https://www.django-rest-framework.org/api-guide/serializers/#html-json-forms)

The [html-json-forms](https://github.com/wq/html-json-forms) package provides an algorithm and serializer for processing <form> submissions per the (inactive) [HTML JSON Form specification](https://www.w3.org/TR/html-json-forms/). The serializer facilitates processing of arbitrarily nested JSON structures within HTML. For example, <input name="items[0][id]" value="5"> will be interpreted as {"items": [{"id": "5"}]}.

## [**DRF-Base64**](https://www.django-rest-framework.org/api-guide/serializers/#drf-base64)

[DRF-Base64](https://bitbucket.org/levit_scs/drf_base64) provides a set of field and model serializers that handles the upload of base64-encoded files.

## [**QueryFields**](https://www.django-rest-framework.org/api-guide/serializers/#queryfields)

[djangorestframework-queryfields](https://djangorestframework-queryfields.readthedocs.io/) allows API clients to specify which fields will be sent in the response via inclusion/exclusion query parameters.

## [**DRF Writable Nested**](https://www.django-rest-framework.org/api-guide/serializers/#drf-writable-nested)

The [drf-writable-nested](https://github.com/beda-software/drf-writable-nested) package provides writable nested model serializer which allows to create/update models with nested related data.

## Create the serializer



## Create the View that will work with the serializer



### URLS.py for Project

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

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

path('api/', include('profiles\_api.urls')),

]

URLS.py for the application

from django.urls import path

from profiles\_api import views

urlpatterns = [

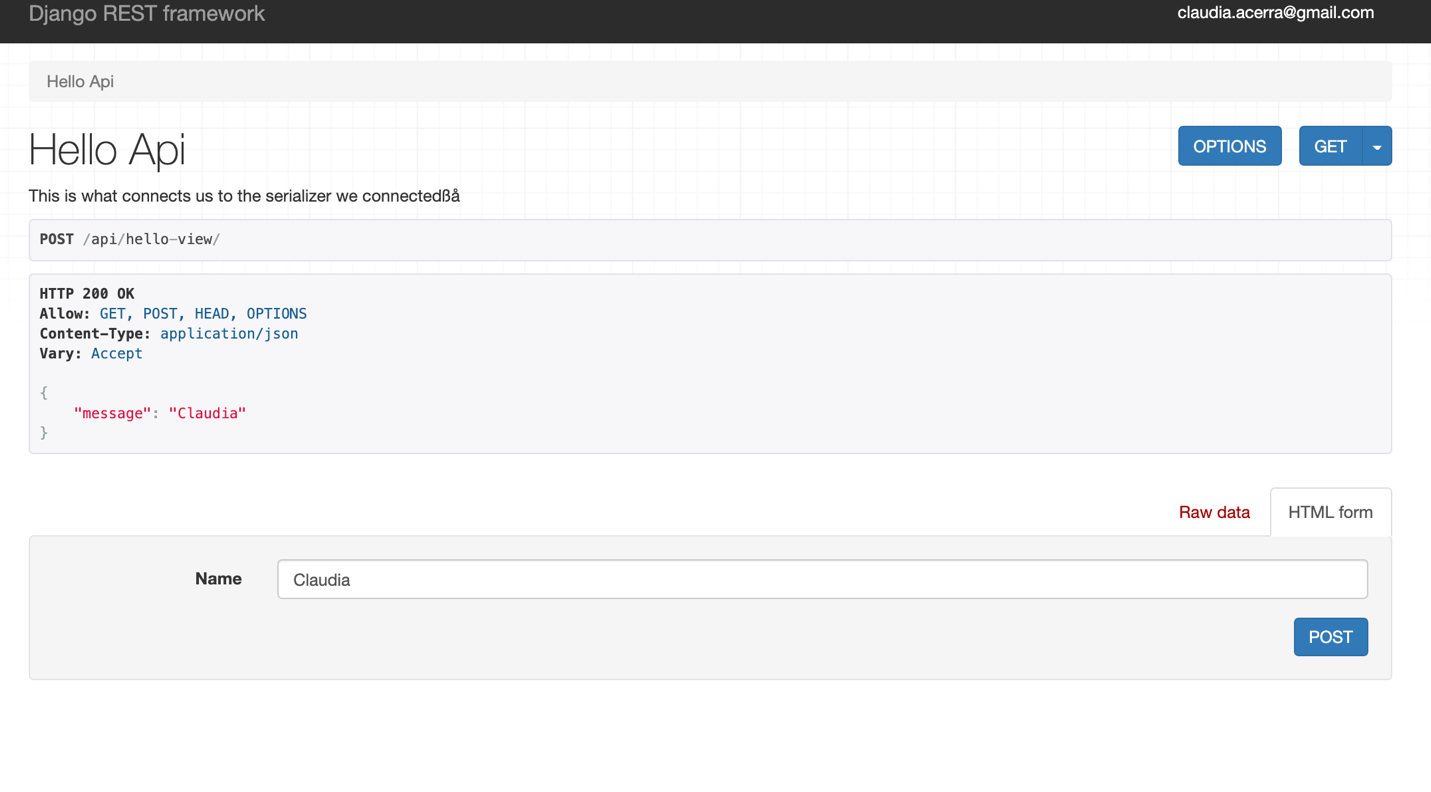
path('hello-view/', views.HelloApiView.as\_view()),

]

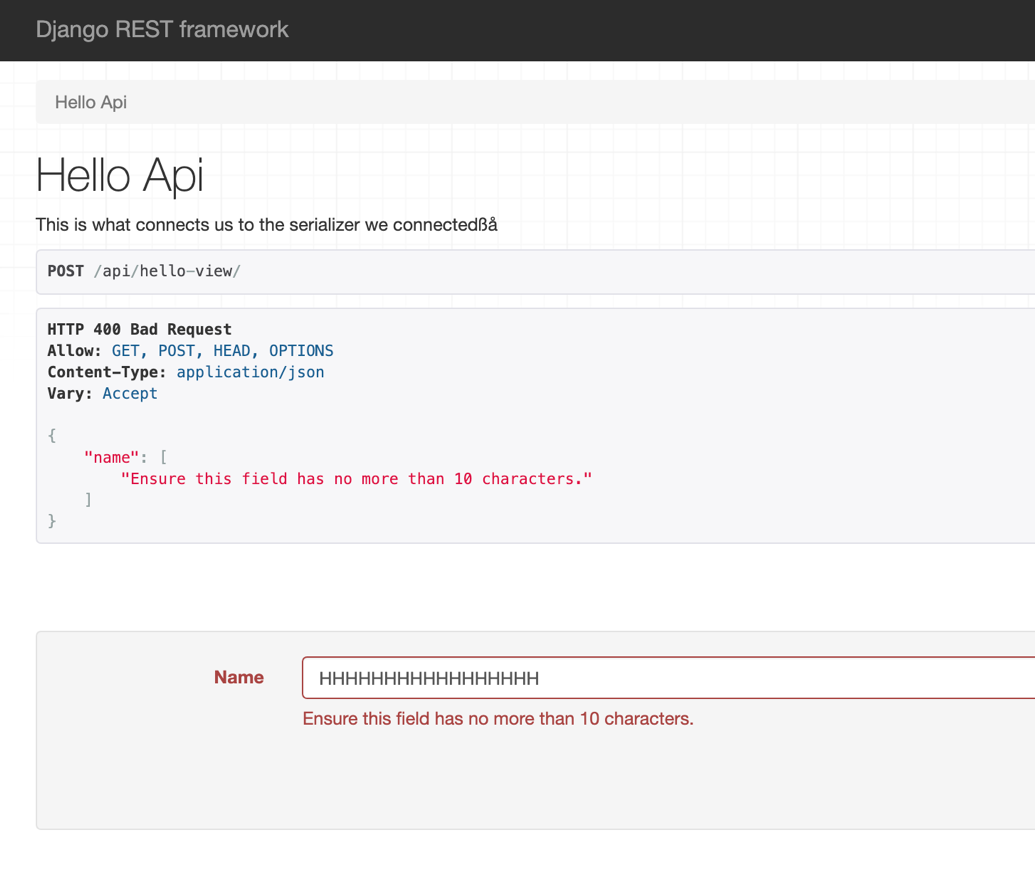
See how these two URLS work together to make the URL we need to access below

Now I can got to the URL <http://127.0.0.1:8000/api/hello-view/>

And I will get this below. It is a form where I can add my name. If I do that, the data is valid and I get what I get below



I can add non-valid data and I will get



### Put Patch and Delete Methods