Django CRUD with MySQL example | Django Rest Framework

Last modified: September 28, 2020 (https://bezkoder.com/django-crud-mysql-rest-framework/) bezkoder (https://bezkoder.com/author/bezkoder/) Django (https://bezkoder.com/category/django/), Python (https://bezkoder.com/category/python/)

In this tutorial, we're gonna create Python/Django CRUD with MySQL example that uses Django Rest Framework for building Rest Apis. You'll know:

- How to setup Django to connect with MySQL Database
- How to define Data Models and migrate it to MySQL
- Way to use Django Rest Framework to process HTTP requests
- Way to make Django CRUD Operations with MySQL Database

Related Posts:
– Django & MongoDB CRUD Rest API | Django Rest Framework (https://bezkoder.com/django-mongodb-crud-rest-framework/)
– Django & PostgreSQL CRUD example | Django Rest Framework (https://bezkoder.com/django-postgresql-crud-rest-framework/)

Fullstack:

– Django + Angular 8 (https://bezkoder.com/django-angular-crud-rest-framework/)
– Django + Angular 10 (https://bezkoder.com/django-angular-10-crud-rest-framework/)
– Django + React (https://bezkoder.com/django-react-axios-rest-framework/)
– Django + Vue.js (https://bezkoder.com/django-vue-js-rest-framework/)

Contents [hide]

- Django CRUD with MySQL overview
- Architecture
- Technology
- Project structure
- Install Django REST framework
- Setup new Django project
- Connect Django project to MySQL
- Setup new Django app for Rest CRUD Api
- Configure CORS
- Define the Django Model
- Migrate Data Model to the database
- Create Serializer class for Data Model
- Define Routes to Views functions
- Write API Views
    - Create a new object
    - Retrieve objects (with condition)
    - Retrieve a single object
    - Update an object
    - Delete an object
    - Delete all objects
    - Find all objects by condition
- Test the CRUD with APIs
- Conclusion
- Further Reading
- Source code

Django CRUD with MySQL overview

We will build Rest Apis using Django Rest Framework (https://www.django-rest-framework.org/) that can create, retrieve, update, delete and find Tutorials by title or published status.

First, we setup Django Project with a MySQL Client. Next, we create Rest Api app, add it with Django Rest Framework to the project. Next, we define data model and migrate it to the database. Then we write API Views and define Routes for handling all CRUD operations (including custom finder).

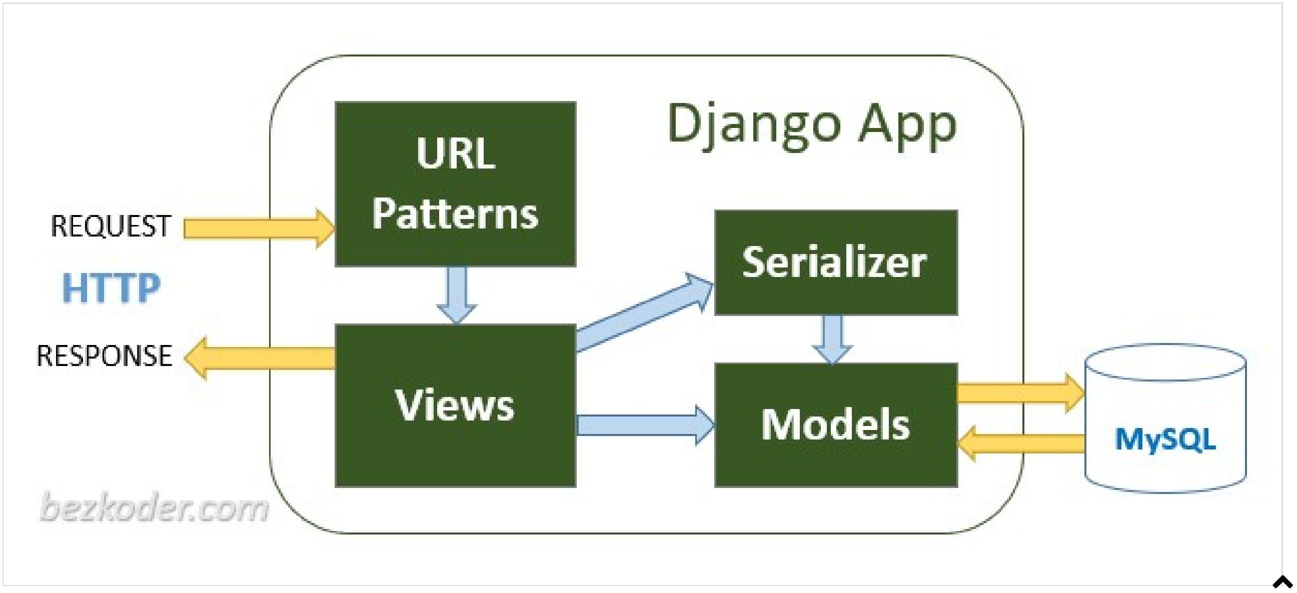
The following table shows overview of the Rest APIs that will be exported:

Methods	Urls	Actions
GET	api/tutorials	get all Tutorials
GET	api/tutorials/:id	get Tutorial by `id`
POST	api/tutorials	add new Tutorial
PUT	api/tutorials/:id	update Tutorial by `id`
DELETE	api/tutorials/:id	remove Tutorial by `id`
DELETE	api/tutorials	remove all Tutorials
GET	api/tutorials/published	find all published Tutorials
GET	api/tutorials?title=[kw]	find all Tutorials which title contains `'kw'`

Finally, we're gonna test the Rest Apis using Postman.

Architecture

Let's look at the diagram below, it shows the architecture of our Django CRUD Rest Apis App with MySQL database:



- HTTP requests will be matched by **Url Patterns** and passed to the **Views**

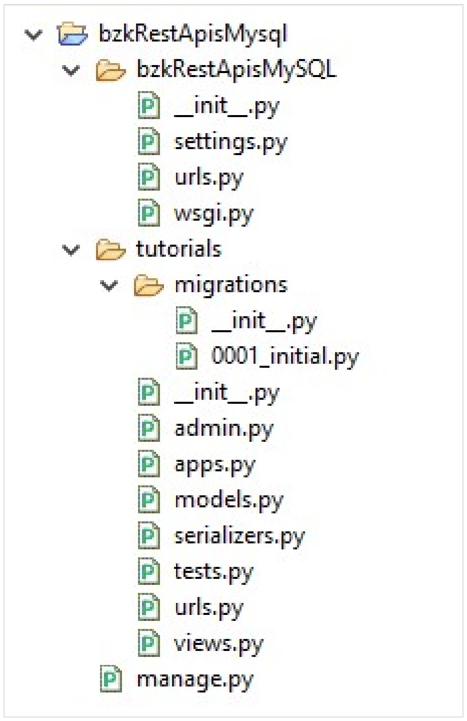
- **Views** processes the HTTP requests and returns HTTP responses (with the help of **Serializer**)
- **Serializer** serializes/deserializes data model objects
- **Models** contains essential fields and behaviors for CRUD Operations with MySQL Database

Technology

- Python 3.7
- Django 2.1.15
- Django Rest Framework 3.11.0
- PyMySQL 0.9.3
- django-cors-headers 3.2.1

Project structure

This is our project structure:



```
bzkRestApisMysql
  bzkRestApisMySQL
    __init__.py
    settings.py
    urls.py
    wsgi.py
  tutorials
    migrations
      __init__.py
      0001_initial.py
    __init__.py
    admin.py
    apps.py
    models.py
    serializers.py
    tests.py
    urls.py
    views.py
  manage.py
```

Let me explain it briefly.

- *tutorials/apps.py*: declares `TutorialsConfig` class (subclass of `django.apps.AppConfig`) that represents Rest CRUD Apis app and its configuration.
- *bzkRestApisMySQL/settings.py*: contains settings for our Django project: MySQL Database engine, `INSTALLED_APPS` list with Django REST framework, Tutorials Application, CORS and `MIDDLEWARE`.
- *tutorials/models.py*: defines Tutorial data model class (subclass of

`django.db.models.Model`).

- *migrations/0001_initial.py*: is created when we make migrations for the data model, and will be used for generating MySQL database table.
- *tutorials/serializers.py*: manages serialization and deserialization with `TutorialSerializer` class (subclass of `rest_framework.serializers.ModelSerializer`).
- *tutorials/views.py*: contains functions to process HTTP requests and produce HTTP responses (using `TutorialSerializer`).
- *tutorials/urls.py*: defines URL patterns along with request functions in the Views.
- *bzkRestApisMySQL/urls.py*: also has URL patterns that includes `tutorials.urls` , it is the root URL configurations.

Install Django REST framework

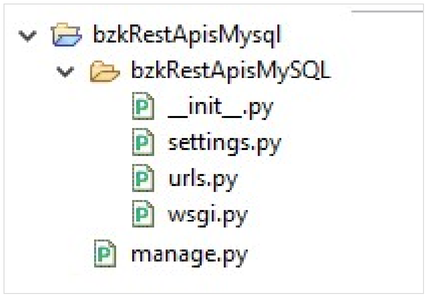
Django REST framework helps us to build RESTful Web Services flexibly.

To install this package, run command:
`pip install djangorestframework`

Setup new Django project

Let's create a new Django project with command:
`django-admin startproject bzkRestApisMySQL`

When the process is done, you can see folder tree like this:



```
bzkRestApisMysql
  bzkRestApisMySQL
    __init__.py
    settings.py
    urls.py
    wsgi.py
  manage.py
```

Now we open *settings.py* and add Django REST framework to the `INSTALLED_APPS` array here.

```
INSTALLED_APPS = [
    ...
    # Django REST framework
    'rest_framework',
]
```

Connect Django project to MySQL

We need a MySQL Client to work with MySQL database.
In this tutorial, we're gonna use *pymysql*.

Run the command to install it: `pip install pymysql` .
Then open *__init__.py* and write following code to import `pymysql` to our Django project:

```
import pymysql

pymysql.install_as_MySQLdb()
```

We also need to setup MySQL Database engine.
So open *settings.py* and change declaration of `DATABASES` :

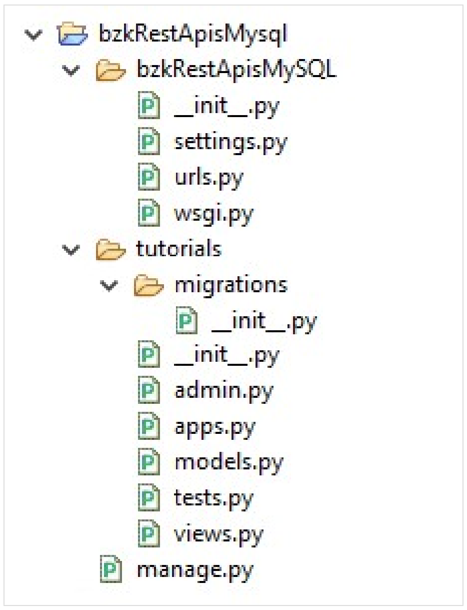
```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'testdb',
        'USER': 'root',
        'PASSWORD': '123456',
        'HOST': '127.0.0.1',
        'PORT': '3306',
    }
}
```

Setup new Django app for Rest CRUD Api

Run following commands to create new Django app **tutorials**:

```
cd bzkRestApisMySQL
python manage.py startapp tutorials
```

Refresh the project directory tree, you can see it now looks like:



```
bzkRestApisMysql
  bzkRestApisMySQL
    __init__.py
    settings.py
    urls.py
    wsgi.py
  tutorials
    migrations
      __init__.py
    __init__.py
    admin.py
    apps.py
    models.py
    tests.py
    views.py
  manage.py
```

Now open *tutorials/apps.py*, you can see `TutorialsConfig` class (subclass of `django.apps.AppConfig`).
This represents the Django app that we've just created with its configuration:

```
from django.apps import AppConfig


class TutorialsConfig(AppConfig):
    name = 'tutorials'
```

Don't forget to add this app to `INSTALLED_APPS` array in *settings.py*:

```
INSTALLED_APPS = [
    ...
    # Tutorials application
    'tutorials.apps.TutorialsConfig',
]
```

Configure CORS

We need to allow requests to our Django application from other origins.
In this example, we're gonna configure CORS to accept requests from `localhost:8081`.

First, install the *django-cors-headers* library:
`pip install django-cors-headers`

In *settings.py*, add configuration for CORS:

```
INSTALLED_APPS = [
    ...
    # CORS
    'corsheaders',
]
```

You also need to add a middleware class to listen in on responses:

```
MIDDLEWARE = [
    ...
    # CORS
    'corsheaders.middleware.CorsMiddleware',
    'django.middleware.common.CommonMiddleware',
]
```

Note: `CorsMiddleware` should be placed as high as possible, especially before any middleware that can generate responses such as `CommonMiddleware`.

Next, set *CORS_ORIGIN_ALLOW_ALL* and add the host to *CORS_ORIGIN_WHITELIST*:

```
CORS_ORIGIN_ALLOW_ALL = False
CORS_ORIGIN_WHITELIST = (
    'http://localhost:8081',
)
```

- *CORS_ORIGIN_ALLOW_ALL*: If `True`, all origins will be accepted (not use the whitelist below). Defaults to `False`.
- *CORS_ORIGIN_WHITELIST*: List of origins that are authorized to make cross-site HTTP requests. Defaults to `[]`.

Define the Django Model

Open **tutorials**/*models.py*, add `Tutorial` class as subclass of `django.db.models.Model`. There are 3 fields: *title*, *description*, *published*.

```
from django.db import models


class Tutorial(models.Model):
    title = models.CharField(max_length=70, blank=False, default='')
    description = models.CharField(max_length=200,blank=False, default='')
    published = models.BooleanField(default=False)
```

Each field is specified as a class attribute, and each attribute maps to a database column. *id* field is added automatically.

Migrate Data Model to the database

Run the Python script: `python manage.py makemigrations tutorials`.

The console will show:

```
Migrations for 'tutorials':
  tutorials\migrations\0001_initial.py
    - Create model Tutorial
```

Refresh the workspace, you can see new file *tutorials/migrations/0001_initial.py*.
It includes code to create `Tutorial` data model:

```
# Generated by Django 2.1.15

from django.db import migrations, models


class Migration(migrations.Migration):

    initial = True

    dependencies = [
    ]

    operations = [
        migrations.CreateModel(
            name='Tutorial',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True, seri
                ('title', models.CharField(default='', max_length=70)),
                ('description', models.CharField(default='', max_length=200)),
                ('published', models.BooleanField(default=False)),
            ],
        ),
    ]
```

The generated code defines `Migration` class (subclass of the `django.db.migrations.Migration`).
It has operations array that contains operation for creating Customer model table: `migrations.CreateModel()`.

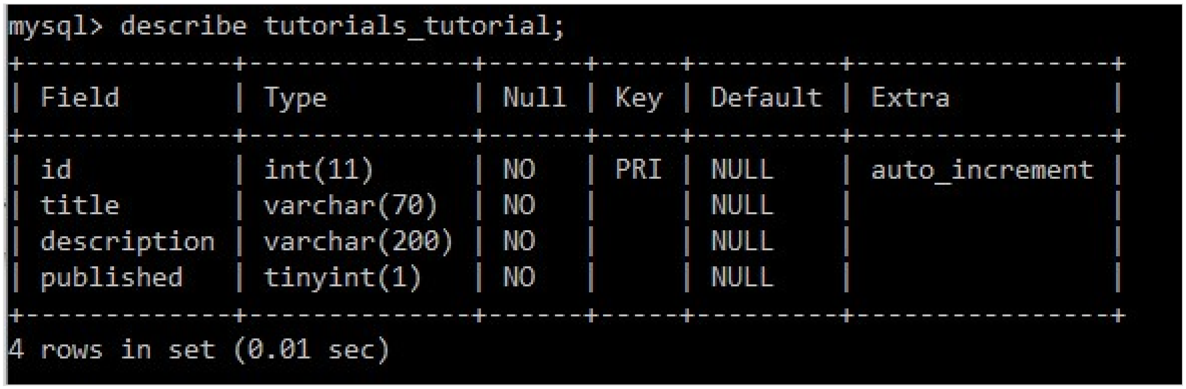
The call to this will create a new model in the project history and a corresponding table in the database to match it.

To apply the generated migration above, run the following Python script:
`python manage.py migrate tutorials`

The console will show:

```
Operations to perform:
  Apply all migrations: tutorials
Running migrations:
  Applying tutorials.0001_initial... OK
```

At this time, you can see that a table for `Tutorial` model was generated automatically with the name: ***tutorials_tutorial***:



```
mysql> describe tutorials_tutorial;
+-------------+--------------+------+-----+---------+----------------+
| Field       | Type         | Null | Key | Default | Extra          |
+-------------+--------------+------+-----+---------+----------------+
| id          | int(11)      | NO   | PRI | NULL    | auto_increment |
| title       | varchar(70)  | NO   |     | NULL    |                |
| description | varchar(200) | NO   |     | NULL    |                |
| published   | tinyint(1)   | NO   |     | NULL    |                |
+-------------+--------------+------+-----+---------+----------------+
4 rows in set (0.01 sec)
```

Create Serializer class for Data Model

Let's create `TutorialSerializer` class that will manage serialization and deserialization from JSON.

It inherit from `rest_framework.serializers.ModelSerializer` superclass which automatically populates a set of `fields` and default `validators`. We need to specify the model class here.

tutorials/*serializers.py*

```
from rest_framework import serializers
from tutorials.models import Tutorial


class TutorialSerializer(serializers.ModelSerializer):

    class Meta:
        model = Tutorial
        fields = ('id',
                  'title',
                  'description',
                  'published')
```

In the inner class `Meta`, we declare 2 attributes:

- `model`: the model for Serializer
- `fields`: a tuple of field names to be included in the serialization

Define Routes to Views functions

When a client sends request for an endpoint using HTTP request (GET, POST, PUT, DELETE), we need to determine how the server will response by defining the routes.

These are our routes:

- `/api/tutorials` : GET, POST, DELETE
- `/api/tutorials/:id` : GET, PUT, DELETE
- `/api/tutorials/published` : GET

Create a *urls.py* inside **tutorials** app with `urlpatterns` containing `url`s to be matched with request functions in the *views.py*:

```
from django.conf.urls import url
from tutorials import views

urlpatterns = [
    url(r'^api/tutorials$', views.tutorial_list),
    url(r'^api/tutorials/(?P<pk>[0-9]+)$', views.tutorial_detail),
    url(r'^api/tutorials/published$', views.tutorial_list_published)
]
```

Don't forget to include this URL patterns in root URL configurations.
Open **bzkRestApisMySQL**/*urls.py* and modify the content with the following code:

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

urlpatterns = [
    url(r'^', include('tutorials.urls')),
]
```

Write API Views

We're gonna create these API functions for CRUD Operations:
– `tutorial_list()` : GET list of tutorials, POST a new tutorial, DELETE all tutorials
– `tutorial_detail()` : GET / PUT / DELETE tutorial by 'id'
– `tutorial_list_published()` : GET all published tutorials

Open **tutorials**/*views.py* and write following code:

```
from django.shortcuts import render

from django.http.response import JsonResponse
from rest_framework.parsers import JSONParser
from rest_framework import status

from tutorials.models import Tutorial
from tutorials.serializers import TutorialSerializer
from rest_framework.decorators import api_view


@api_view(['GET', 'POST', 'DELETE'])
def tutorial_list(request):
    # GET list of tutorials, POST a new tutorial, DELETE all tutorials


@api_view(['GET', 'PUT', 'DELETE'])
def tutorial_detail(request, pk):
    # find tutorial by pk (id)
    try:
        tutorial = Tutorial.objects.get(pk=pk)
    except Tutorial.DoesNotExist:
        return JsonResponse({'message': 'The tutorial does not exist'}, status=sta

    # GET / PUT / DELETE tutorial


@api_view(['GET'])
def tutorial_list_published(request):
    # GET all published tutorials
```

Let’s implement these functions.

Create a new object

Create and Save a new Tutorial:

```
@api_view(['GET', 'POST', 'DELETE'])
def tutorial_list(request):
    ...

    elif request.method == 'POST':
        tutorial_data = JSONParser().parse(request)
        tutorial_serializer = TutorialSerializer(data=tutorial_data)
        if tutorial_serializer.is_valid():
            tutorial_serializer.save()
            return JsonResponse(tutorial_serializer.data, status=status.HTTP_201_
        return JsonResponse(tutorial_serializer.errors, status=status.HTTP_400_BA
```

Retrieve objects (with condition)

Retrieve all Tutorials/ find by `title` from MySQL database:

```
@api_view(['GET', 'POST', 'DELETE'])
def tutorial_list(request):
    if request.method == 'GET':
        tutorials = Tutorial.objects.all()

        title = request.GET.get('title', None)
        if title is not None:
            tutorials = tutorials.filter(title__icontains=title)

        tutorials_serializer = TutorialSerializer(tutorials, many=True)
        return JsonResponse(tutorials_serializer.data, safe=False)
        # 'safe=False' for objects serialization
```

Retrieve a single object

Find a single Tutorial with an `id`:

```
@api_view(['GET', 'PUT', 'DELETE'])
def tutorial_detail(request, pk):
    # ... tutorial = Tutorial.objects.get(pk=pk)

    if request.method == 'GET':
        tutorial_serializer = TutorialSerializer(tutorial)
        return JsonResponse(tutorial_serializer.data)
```

Update an object

Update a Tutorial by the `id` in the request:

```
@api_view(['GET', 'PUT', 'DELETE'])
def tutorial_detail(request, pk):
    # ... tutorial = Tutorial.objects.get(pk=pk)
    # ...

    elif request.method == 'PUT':
        tutorial_data = JSONParser().parse(request)
        tutorial_serializer = TutorialSerializer(tutorial, data=tutorial_data)
        if tutorial_serializer.is_valid():
            tutorial_serializer.save()
            return JsonResponse(tutorial_serializer.data)
        return JsonResponse(tutorial_serializer.errors, status=status.HTTP_400_BA
```

Delete an object

Delete a Tutorial with the specified `id`:

```
@api_view(['GET', 'PUT', 'DELETE'])
def tutorial_detail(request, pk):
    # ... tutorial = Tutorial.objects.get(pk=pk)
    # ...

    elif request.method == 'DELETE':
        tutorial.delete()
        return JsonResponse({'message': 'Tutorial was deleted successfully!'}, st
```

Delete all objects

Delete all Tutorials from the database:

```
@api_view(['GET', 'POST', 'DELETE'])
def tutorial_list(request):
    # ...

    elif request.method == 'DELETE':
        count = Tutorial.objects.all().delete()
        return JsonResponse({'message': '{} Tutorials were deleted successfully!'
```

Find all objects by condition

Find all Tutorials with `published = True`:

```
@api_view(['GET'])
def tutorial_list_published(request):
    tutorials = Tutorial.objects.filter(published=True)

    if request.method == 'GET':
        tutorials_serializer = TutorialSerializer(tutorials, many=True)
        return JsonResponse(tutorials_serializer.data, safe=False)
```

Test the CRUD with APIs

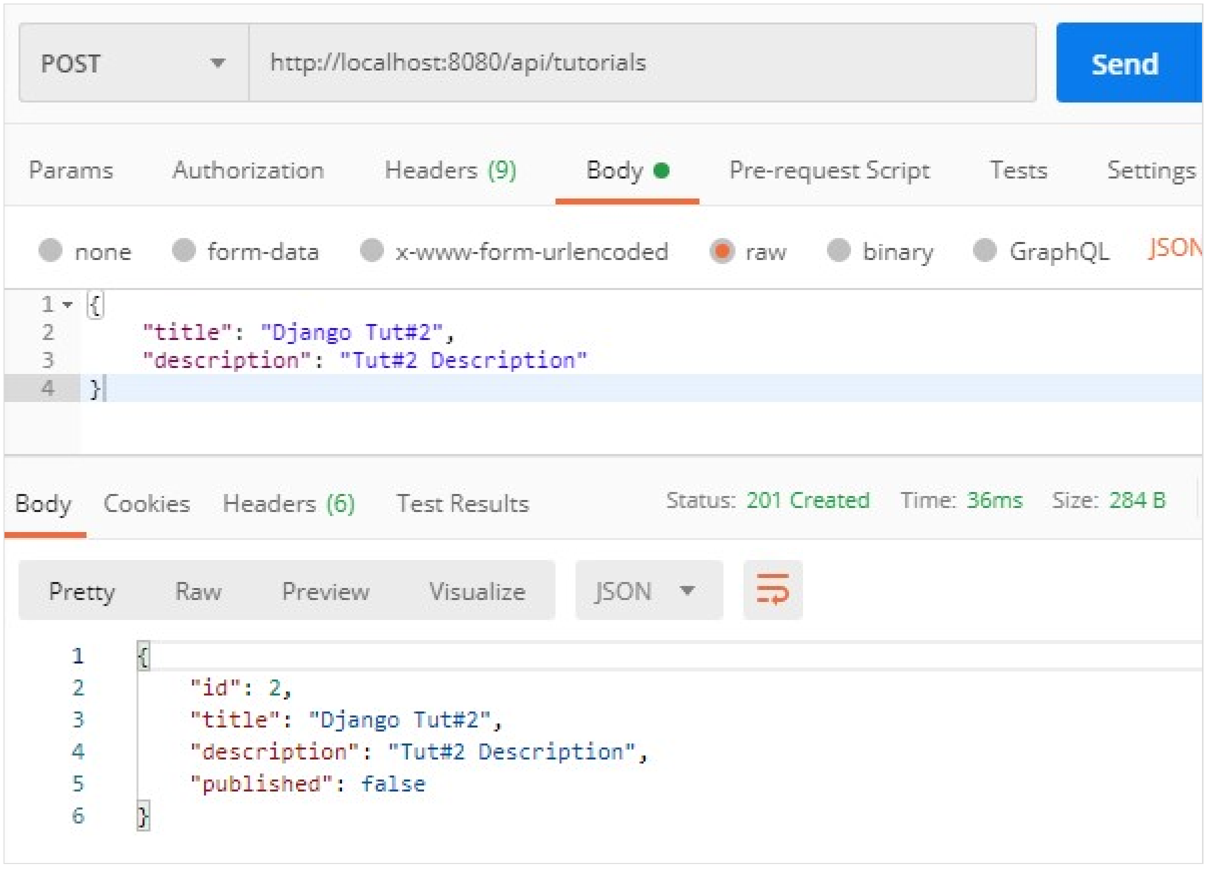
Run our Django Project with command: `python manage.py runserver 8080`.
The console shows:

```
Performing system checks...

System check identified no issues (0 silenced).
March 26, 2020 - 15:56:15
Django version 2.1.15, using settings 'bzkRestApisMySQL.settings'
Starting development server at http://127.0.0.1:8080/
Quit the server with CTRL-BREAK.
```

Using Postman, we're gonna test all the Apis above.

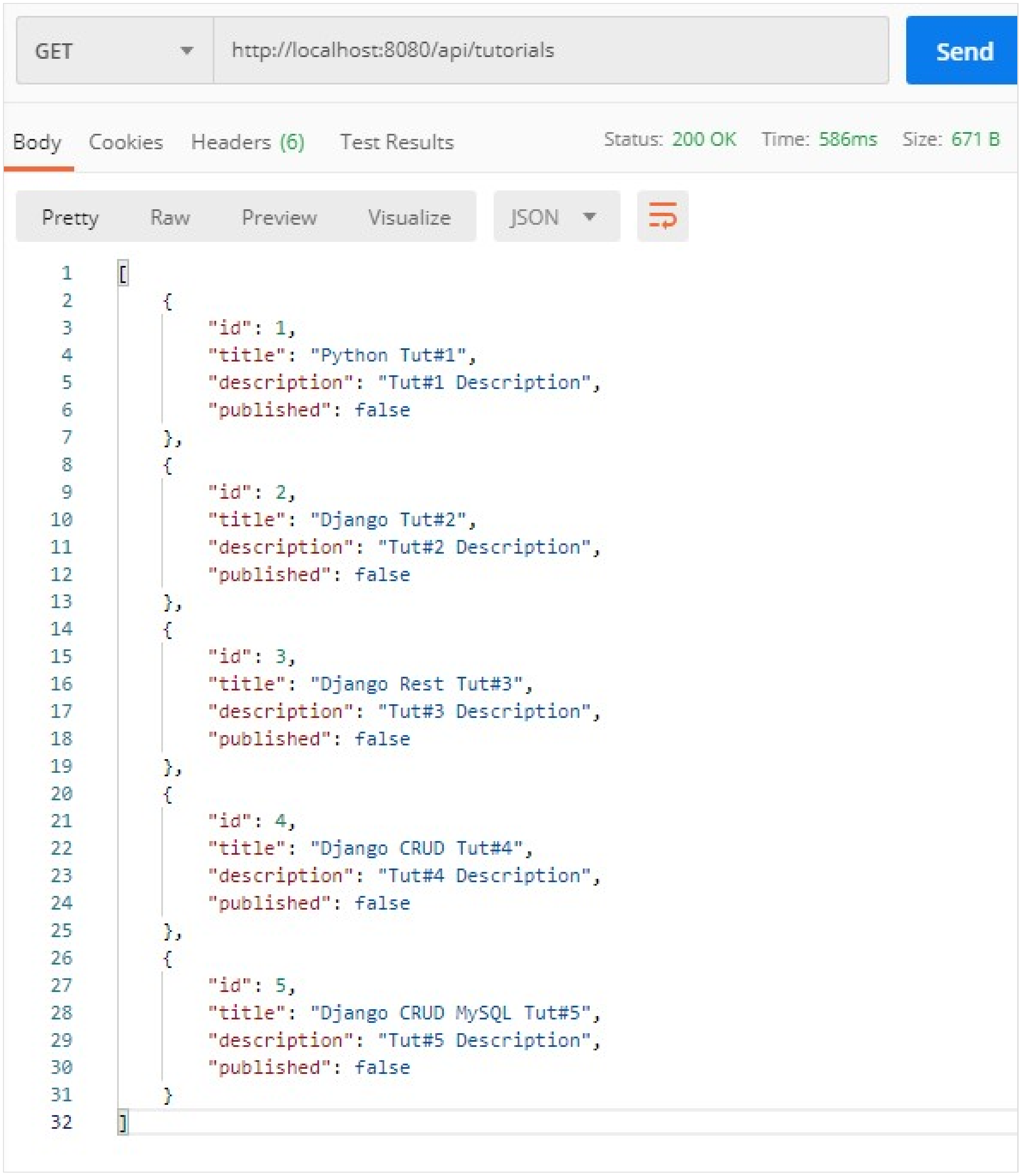
1. **Create a new Tutorial using `POST /tutorials` Api**



```
{
    "title": "Django Tut#2",
    "description": "Tut#2 Description"
}
```

```
{
    "id": 2,
    "title": "Django Tut#2",
    "description": "Tut#2 Description",
    "published": false
}
```

2. **Retrieve all Tutorials using `GET /tutorials` Api**

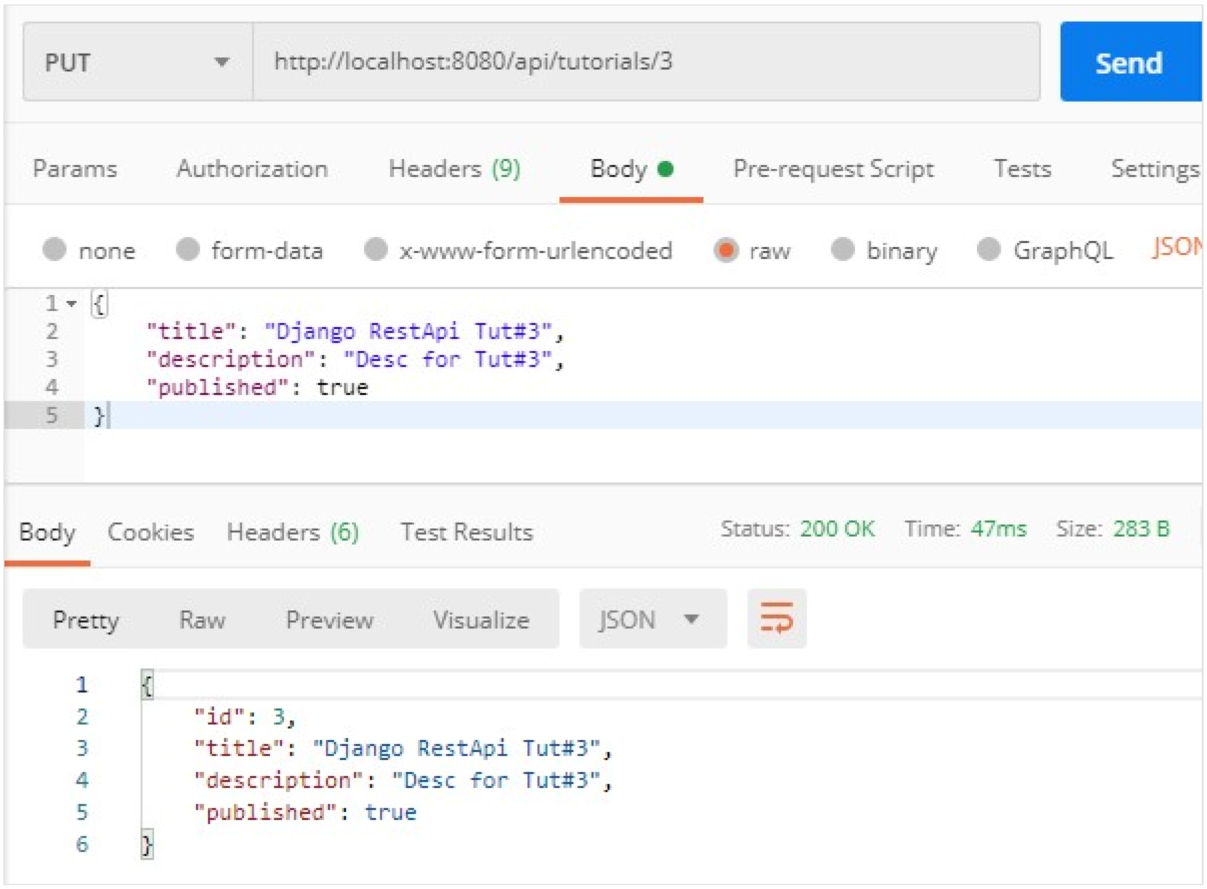


http://localhost:8080/api/tutorials

Status: 200 OK Time: 586ms Size: 671 B

```
[
    {
        "id": 1,
        "title": "Python Tut#1",
        "description": "Tut#1 Description",
        "published": false
    },
    {
        "id": 2,
        "title": "Django Tut#2",
        "description": "Tut#2 Description",
        "published": false
    },
    {
        "id": 3,
        "title": "Django Rest Tut#3",
        "description": "Tut#3 Description",
        "published": false
    },
    {
        "id": 4,
        "title": "Django CRUD Tut#4",
        "description": "Tut#4 Description",
        "published": false
    },
    {
        "id": 5,
        "title": "Django CRUD MySQL Tut#5",
        "description": "Tut#5 Description",
        "published": false
    }
]
```

3. **Update a Tutorial using** `PUT /tutorials/:id` **Api**



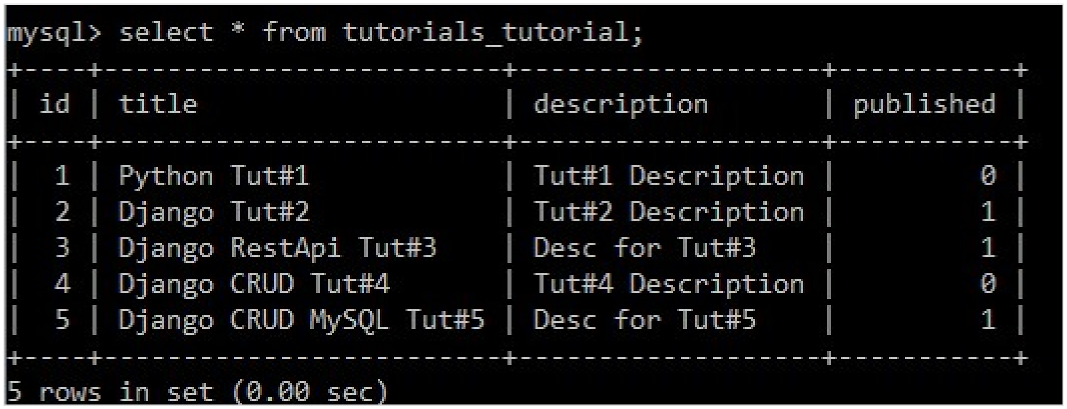
http://localhost:8080/api/tutorials/3

```
{
    "title": "Django RestApi Tut#3",
    "description": "Desc for Tut#3",
    "published": true
}
```

Status: 200 OK Time: 47ms Size: 283 B

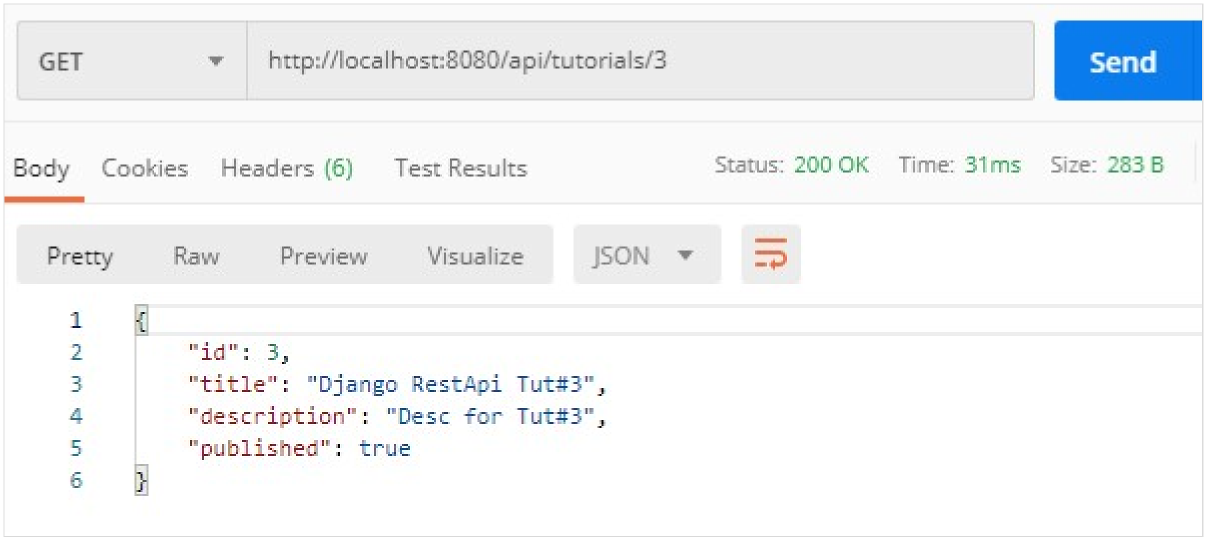
```
{
    "id": 3,
    "title": "Django RestApi Tut#3",
    "description": "Desc for Tut#3",
    "published": true
}
```

Check `tutorials_tutorial` table after some rows were updated:



```
mysql> select * from tutorials_tutorial;
+----+-------------------------+-------------------+-----------+
| id | title                   | description       | published |
+----+-------------------------+-------------------+-----------+
|  1 | Python Tut#1            | Tut#1 Description |         0 |
|  2 | Django Tut#2            | Tut#2 Description |         1 |
|  3 | Django RestApi Tut#3    | Desc for Tut#3    |         1 |
|  4 | Django CRUD Tut#4       | Tut#4 Description |         0 |
|  5 | Django CRUD MySQL Tut#5 | Desc for Tut#5    |         1 |
+----+-------------------------+-------------------+-----------+
5 rows in set (0.00 sec)
```

4. **Retrieve a single Tutorial by id using `GET /tutorials/:id` Api**

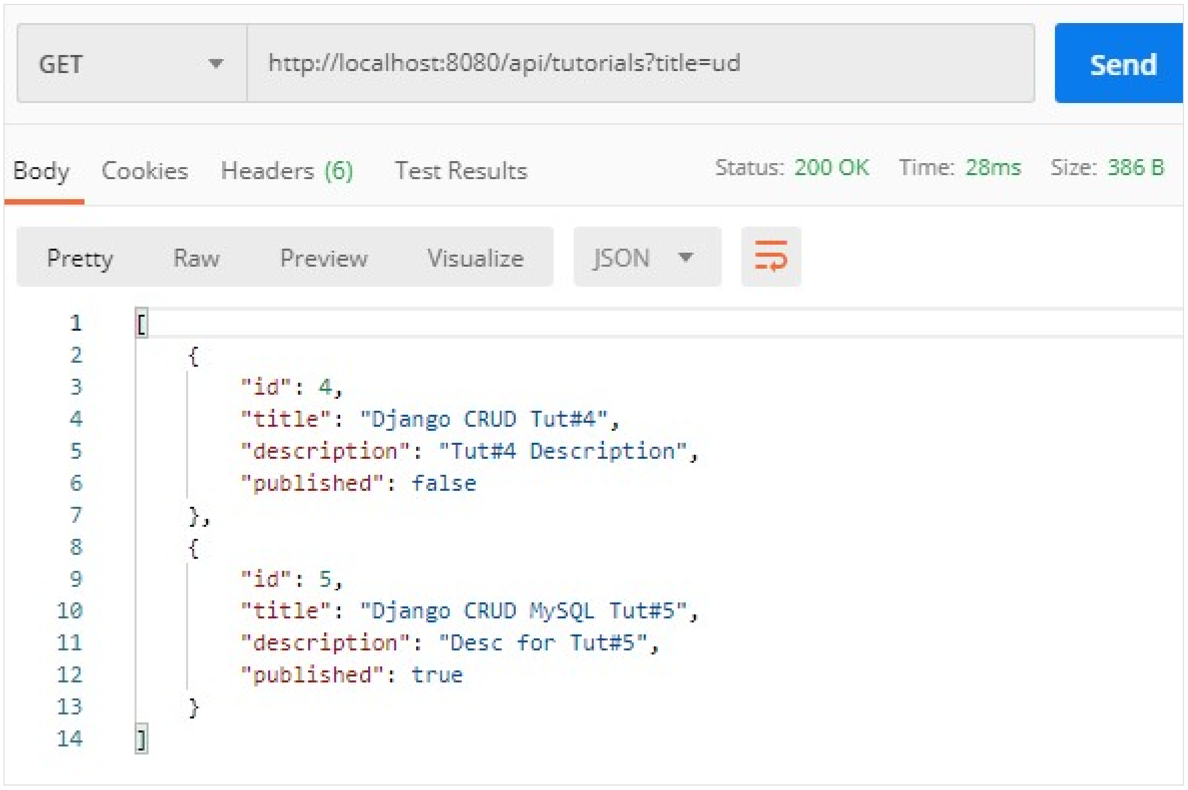

GET http://localhost:8080/api/tutorials/3 Send

Body Cookies Headers (6) Test Results Status: 200 OK Time: 31ms Size: 283 B

Pretty Raw Preview Visualize JSON

```
{
    "id": 3,
    "title": "Django RestApi Tut#3",
    "description": "Desc for Tut#3",
    "published": true
}
```

5. **Find all Tutorials which title contains 'ud':** `GET /tutorials?title=ud`

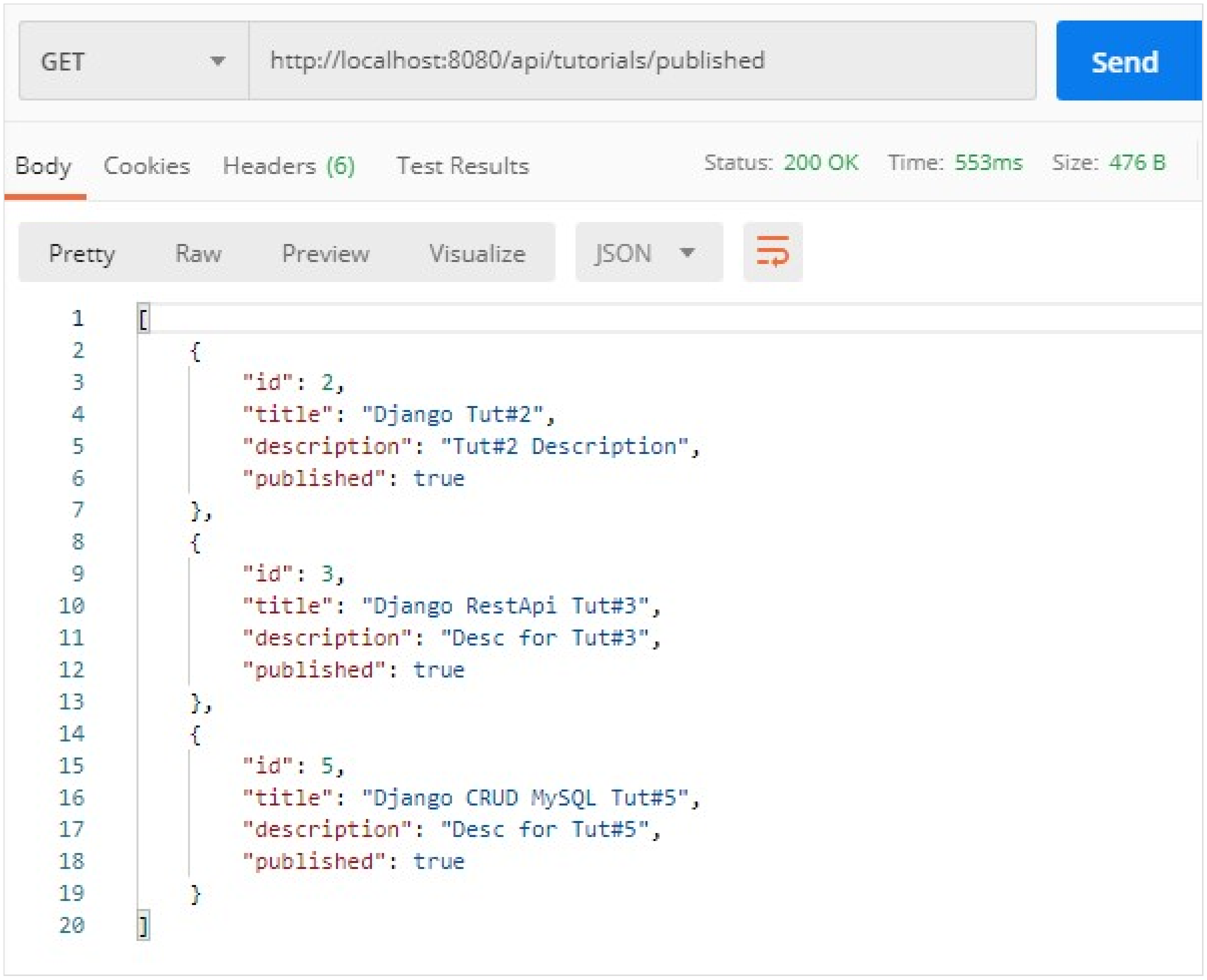

GET http://localhost:8080/api/tutorials?title=ud Send

Body Cookies Headers (6) Test Results Status: 200 OK Time: 28ms Size: 386 B

Pretty Raw Preview Visualize JSON

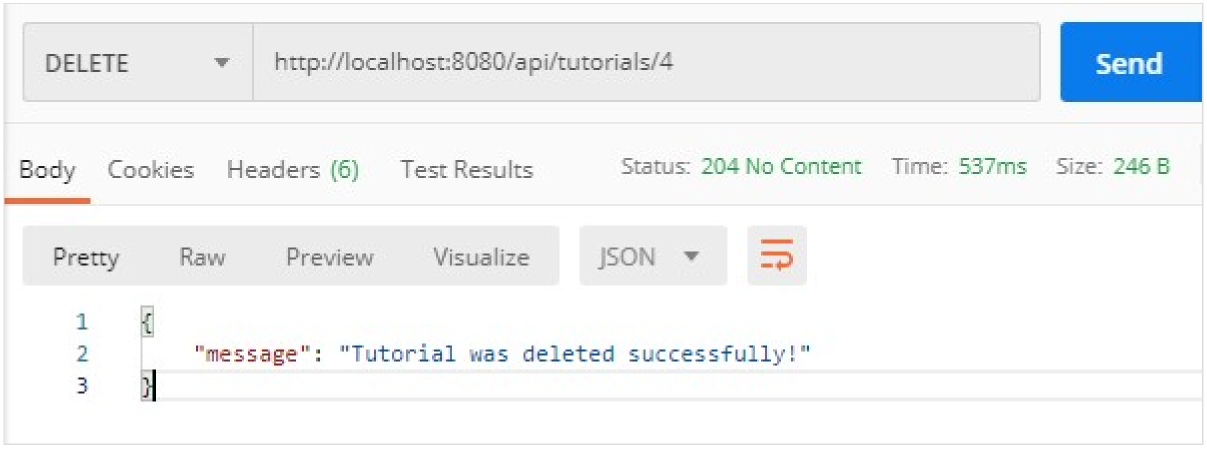
```
[
    {
        "id": 4,
        "title": "Django CRUD Tut#4",
        "description": "Tut#4 Description",
        "published": false
    },
    {
        "id": 5,
        "title": "Django CRUD MySQL Tut#5",
        "description": "Desc for Tut#5",
        "published": true
    }
]
```

6. **Find all published Tutorials using** `GET /tutorials/published` **Api**



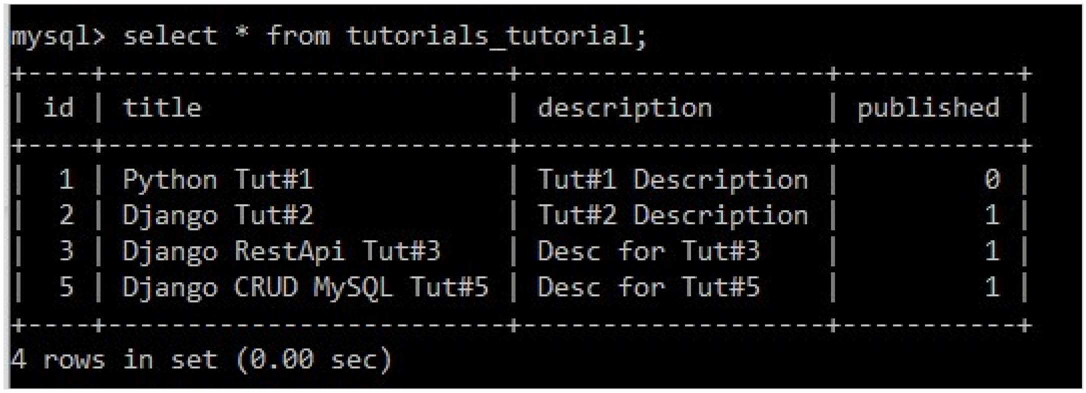
```
[
    {
        "id": 2,
        "title": "Django Tut#2",
        "description": "Tut#2 Description",
        "published": true
    },
    {
        "id": 3,
        "title": "Django RestApi Tut#3",
        "description": "Desc for Tut#3",
        "published": true
    },
    {
        "id": 5,
        "title": "Django CRUD MySQL Tut#5",
        "description": "Desc for Tut#5",
        "published": true
    }
]
```

7. **Delete a Tutorial using `DELETE /tutorials/:id` Api**



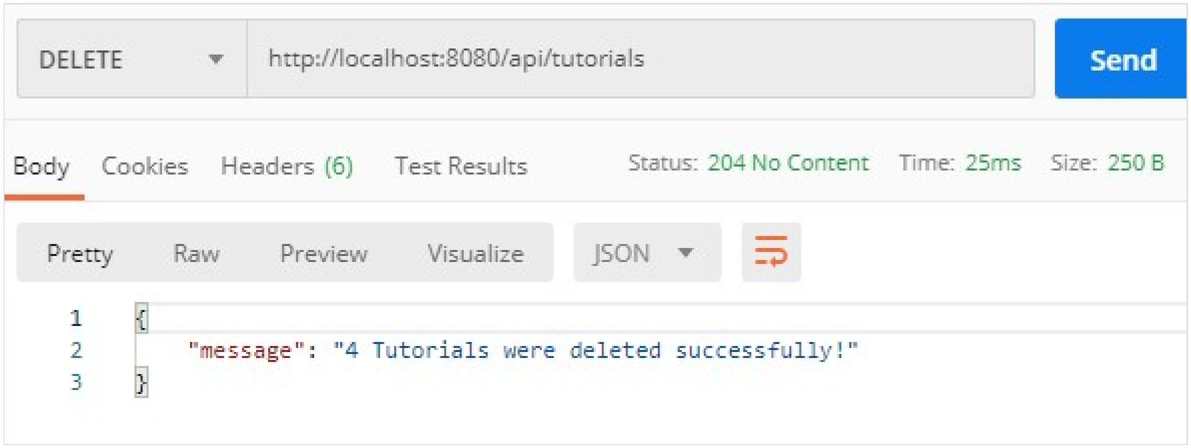
```
{
    "message": "Tutorial was deleted successfully!"
}
```

Tutorial with id=4 was removed from `tutorials` table:



```
mysql> select * from tutorials_tutorial;
+----+------------------------+-------------------+-----------+
| id | title                  | description       | published |
+----+------------------------+-------------------+-----------+
|  1 | Python Tut#1           | Tut#1 Description |         0 |
|  2 | Django Tut#2           | Tut#2 Description |         1 |
|  3 | Django RestApi Tut#3   | Desc for Tut#3    |         1 |
|  5 | Django CRUD MySQL Tut#5 | Desc for Tut#5   |         1 |
+----+------------------------+-------------------+-----------+
4 rows in set (0.00 sec)
```

8. **Delete all Tutorials using `DELETE /tutorials` Api**



```
{
    "message": "4 Tutorials were deleted successfully!"
}
```

Conclusion

Today, we've learned how to create Django CRUD MySQL example Django Rest Framework for Rest Apis. We also know way to connect Django application with MySQL database, create a Django Model, migrate it to database, write the Views and define Url patterns for handling all CRUD operations.

Happy learning! See you again.

Further Reading

- Django Rest Framework quich start (https://www.django-rest-framework.org/tutorial/quickstart/)
- Django Model (https://docs.djangoproject.com/en/2.1/topics/db/models/)
- https://github.com/PyMySQL/PyMySQL (https://github.com/PyMySQL/PyMySQL)
- https://github.com/adamchainz/django-cors-headers (https://github.com/adamchainz/django-cors-headers)

Fullstack CRUD App:

- Django + Angular 8 (https://bezkoder.com/django-angular-crud-rest-framework/)
- Django + Angular 10 (https://bezkoder.com/django-angular-10-crud-rest-framework/)
- Django + React (https://bezkoder.com/django-react-axios-rest-framework/)
- Django + Vue (https://bezkoder.com/django-vue-js-rest-framework/)

Source code

You can find the complete source code for this example on Github (https://github.com/bezkoder/django-rest-api-mysql).

crud (https://bezkoder.com/tag/crud/) django (https://bezkoder.com/tag/django/)

django rest framework (https://bezkoder.com/tag/django-rest-framework/)

mysql (https://bezkoder.com/tag/mysql/) python (https://bezkoder.com/tag/python/)

rest api (https://bezkoder.com/tag/rest-api/)

One thought to “Django CRUD with MySQL example | Django Rest Framework”

Muhammad Ayaz
July 30, 2020 at 8:35 am (https://bezkoder.com/django-crud-mysql-rest-framework/#comment-4080)

For MySql, phpmyadmin should be running
you can use Xampp for that

REPLY

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment