Competency Based Learning Material (CBLM)

Web Application Development with Python

Level-4

Module: Creating API Using Django REST Framework

Code: CBLM- OU-ICT-WADP-07-L4-V1

National Skills Development Authority
Chief Advisor's Office
Government of the People's Republic of Bangladesh

Copyright

National Skills Development Authority
Chief Advisor's Office
Level6-11, Biniyog Bhaban,
E-6 / B, Agargaon, Sher-E-Bangla Nagar Dhaka-1207, Bangladesh.
Emailec@nsda.gov.bd
Websitewww.nsda.gov.bd.
National Skills Portalhttp:\\skillsportal.gov.bd

This Competency Based Learning Materials (CBLM) on "**Creating API Using Django REST Framework**" under the **Web Application Development with Python , Level-4** qualification is developed based on the national competency standard approved by National Skills Development Authority (NSDA)

This document is to be used as a key reference point by the competency-based learning materials developers, teachers/trainers/assessors as a base on which to build instructional activities.

It serves as the document for providing training consistent with the requirements of industry in order to meet the qualification of individuals who graduated through the established standard via competency-based assessment for a relevant job.

This document has been developed by NSDA in association with industry representatives, academia, related specialist, trainer, and related employee. Public and private institutions may use the information contained in this CBLM for activities benefitting Bangladesh.

Approved by the Authority ……. meeting held on .....

How to use this Competency Based Learning Material (CBLM)

The module contains training materials and activities for you to complete. These activities may be completed as part of structured classroom activities or you may be required you to work at your own pace. These activities will ask you to complete associated learning and practice activities in order to gain knowledge and skills you need to achieve the learning outcomes.

1. Review the **Learning Activity** page to understand the sequence of learning activities you will undergo. This page will serve as your road map towards the achievement of competence.

2. Read the **Information sheet s.** This will give you an understanding of the jobs or tasks you are going to learn how to do. Once you have finished reading the **Information sheet s** complete the questions in the **Self-Check.**

3. **Self-**Checks are found after each **Information sheet** . **Self-Checks** are designed to help you know how you are progressing. If you are unable to answer the questions in the **Self-Check** you will need to re-read the relevant **Information sheet** . Once you have completed all the questions check your answers by reading the relevant **Answer Keys** found at the end of this module.

4. Next move on to the **Job Sheets. Job Sheets** provide detailed information about *how to do the job* you are being trained in. Some **Job Sheets** will also have a series of **Activity Sheets**. These sheets have been designed to introduce you to the job step by step. This is where you will apply the new knowledge you gained by reading the Information sheet s. This is your opportunity to practise the job. You may need to practise the job or activity several times before you become competent.

5. Specification **sheets**, specifying the details of the job to be performed will be provided where appropriate.

6. A review of competency is provided on the last page to help remind if all the required assessment criteria have been met. This record is for your own information and guidance and is not an official record of competency

When working though this Module always be aware of your safety and the safety of others in the training room. Should you require assistance or clarification please consult your trainer or facilitator.

When you have satisfactorily completed all the Jobs and/or Activities outlined in this module, an assessment event will be scheduled to assess if you have achieved competency in the specified learning outcomes. You will then be ready to move onto the next Unit of Competency or Module

Table of Contents

Module Content

Unit of Competency	**Create API Using Django REST Framework**
Unit Code	OU-ICT-WADP-07-L4-V1
Module Title	**Creating API Using Django REST Framework**
Module Descriptor	This module covers the knowledge, skills and attitudes required to create API using Django REST Framework It includes the task of applying DRF concepts and creating CRUD project
Nominal Hours	80 Hours
Lerning Outcome	After completing the practice of the module, the trainees will be able to perform the following jobs 1. Apply DRF concepts 2. Create CRUD projec

Assessment Criteria

1. Serializers and views are applied
2. Class-based views are used
3. Mixins and generic class-based views are applied
4. Authentication, Token, Permission is implemented
5. Searching, filtering, and pagination are implemented
6. Project layout is created
7. Application setup
8. Database is set up
9. Blue print and views are applied
10. Project is made installable

Learning Outcome 1: Apply DRF concepts

Assessment Criteria	1. Serializers and views are applied 2. Class-based views are used 3. Mixins and generic class-based views are applied 4. Authentication, Token, Permission is implemented 5. Searching, filtering, and pagination are implemented
Conditions and Resources	1. Real or simulated workplace 2. CBLM 3. Handouts 4. Laptop 5. Multimedia Projector 6. Paper, Pen, Pencil, Eraser 7. Internet facilities 8. White board and marker 9. Audio Video Device
Contents	1. DRF is installed 2. Serializers and views 3. Class-based views 4. Mixins and generic class-based views 5. Authentication, Token, Permission 6. Searching, filtering, and pagination
Activities/job/Task	1. Implement Serializers and Views in Django REST Framework
Training Methods	1. Discussion 2. Presentation 3. Demonstration 4. Guided Practice 5. Individual Practice 6. Project Work 7. Problem Solving 8. Brainstorming
Assessment Methods	Assessment methods may include but not limited to 1. Written Test 2. Demonstration 3. Oral Questioning 4. Portfolio

Learning Experience 1: Apply DRF concepts

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
1. Trainee will ask the instructor ` the learning materials	1. Instructor will provide the learning materials 'Apply DRF concepts'
2. Read the Information sheet and complete the Self Checks & Check answer sheets on "Apply DRF concepts"	2. Read Information sheet 1: Apply DRF concepts 3. Answer Self-check 1: Apply DRF concepts 4. Check your answer with Answer key 1: Apply DRF concepts
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Job Sheet-1: Implement Serializers and Views in Django REST Framework

Information sheet 1: Apply DRF concepts

Learning Objective:

After completion of this Information sheet , the learners will be able to explain, define and interpret the following contents:

1.1. Serializers and views are applied
1.2. Class-based views are used
1.3. Mixins and generic class-based views are applied
1.4. Authentication, Token, Permission is implemented
1.5. Searching, filtering, and pagination are implemented

1.1. Apply Serializers and views

Serializers are used to convert Django QuerySets and model instances to and from JSON. Also, before deserializing the data, for incoming payloads, serializers validate the shape of the data.

Why does the data need to be (de)serialized?

Django QuerySets and model instances are Django-specific and, as such, not universal. In other words, the data structure needs to be converted into a simplified structure before it can be communicated over a RESTful API.

Serializers are one of the main building blocks of the Django REST framework used to define the representation of data records, which are generally based on Django models. As described in the previous section on Introduction to REST services options for Django, Python records can have ambiguous data representations (e.g. a record with a datetime value can be represented as DD/MM/YYYY, DD-MM-YYYY or MM-YYYY) and a serializer removes any uncertainty about how to represent a record. Example illustrates a Django REST framework serializer using one its serializers package. To specify how incoming and outgoing data gets serialized and deserialized, you create a [SomeResource]Serializer class. So, if you have a Task model, you'd create a TaskSerializer class.

For example:

```
# model
class Task(models.Model):
    title = models.CharField(max_length=200)
    description = models.TextField()
    completed = models.BooleanField(default=False)


# basic serializer
class TaskSerializer(serializers.Serializer):
    title = serializers.CharField()
    description = serializers.CharField()
    completed = serializers.BooleanField()
```

Similarly to how Django forms are created, when the serialization is closely coupled to the model, you can extend from ModelSerializer:

```
class TaskSerializer(serializers.ModelSerializer):
    class Meta:
        model = Task
        fields = "__all__"
```

You can easily adapt a ModelSerializer to your needs:

```
class TaskSerializer(serializers.ModelSerializer):
    short_description = serializers.SerializerMethodField()

    class Meta:
        model = Task
        fields = ["title", "description", "completed", "short_description"]
        read_only_fields = ['completed']

    def get_short_description(self, obj):
        return obj.description[:50]
```

Here, we-

1. Explicitly defined the fields the serializer has access to via the fields attribute
2. Set the completed field to read-only
3. Added additional data -- short_description

DRF also allows you to create a hypertext-driven API:

```
class TaskSerializer(serializers.HyperlinkedModelSerializer):
    subtasks = serializers.HyperlinkedRelatedField(
        many=True,
        read_only=True,
        view_name='subtask-detail'
    )

    class Meta:
        model = Task
        fields = ['name', 'subtasks']
```

While these are just basic examples, they should give you a good idea of how serializers work and why they're necessary.

1.2. Use Class-based views

A view is a callable which takes a request and returns a response. This can be more than just a function, and Django provides an example of some classes which can be used as views. These allow you to structure your views and reuse code by harnessing inheritance and mixins. There are also some generic views for tasks which we'll get to later, but you may want to design your own structure of reusable views which suits your use case. For full details, see the class-based views reference documentation.

Basic examples
Django provides base view classes which will suit a wide range of applications. All views inherit from the **View** class, which handles linking the view into the URLs, HTTP method dispatching and other common features. **RedirectView** provides a HTTP redirect, and **TemplateView** extends the base class to make it also render a template.

Usage in your URLconf
The most direct way to use generic views is to create them directly in your URLconf. If you're only changing a few attributes on a class-based view, you can pass them into the **as_view()** method call itself:

```
from django.urls import path
from django.views.generic import TemplateView

urlpatterns = [
    path("about/", TemplateView.as_view(template_name="about.html")),
]
```

Any arguments passed to **as_view()** will override attributes set on the class. In this example, we set **template_name** on the **TemplateView**. A similar overriding pattern can be used for the **url** attribute on **RedirectView**.

Subclassing generic views

The second, more powerful way to use generic views is to inherit from an existing view and override attributes (such as the **template_name**) or methods (such as **get_context_data**) in your subclass to provide new values or methods. Consider, for example, a view that just displays one template, **about.html**. Django has a generic view to do this - **TemplateView** - so we can subclass it, and override the template name:

```
# some_app/views.py
from django.views.generic import TemplateView


class AboutView(TemplateView):
    template_name = "about.html"
```

Then we need to add this new view into our URLconf. **TemplateView** is a class, not a function, so we point the URL to the **as_view()** class method instead, which provides a function-like entry to class-based views:

```
# urls.py
from django.urls import path
from some_app.views import AboutView

urlpatterns = [
    path("about/", AboutView.as_view()),
]
```

1.3. Mixins and generic class-based views

Django's built-in class-based views provide a lot of functionality, but some of it you may want to use separately. For instance, you may want to write a view that renders a template to make the HTTP response, but you can't use **TemplateView**; perhaps you need to render a template only on **POST**, with **GET** doing something else entirely. While you could use **TemplateResponse** directly, this will likely result in duplicate code.

For this reason, Django also provides a number of mixins that provide more discrete functionality. Template rendering, for instance, is encapsulated in the **TemplateResponseMixin**. The Django reference documentation contains full documentation of all the mixins.

Context and template responses

Two central mixins are provided that help in providing a consistent interface to working with templates in class-based views.

TemplateResponseMixin

Every built in view which returns a **TemplateResponse** will call the **render_to_response()** method that **TemplateResponseMixin** provides. Most of the time this will be called for you (for instance, it is called by the **get()** method implemented by both **TemplateView** and **DetailView**); similarly, it's unlikely that you'll need to override it, although if you want your response to return something not rendered via a Django template then you'll want to do it. For an example of this, see the JSONResponseMixin example.

render_to_response() itself calls **get_template_names()**, which by default will look up **template_name** on the class-based view; two other mixins (**SingleObjectTemplateResponseMixin** and **MultipleObjectTemplateResponseMixin**) override this to provide more flexible defaults when dealing with actual objects.

ContextMixin

Every built in view which needs context data, such as for rendering a template (including **TemplateResponseMixin** above), should call **get_context_data()** passing any data they want to ensure is in there as keyword arguments. **get_context_data()** returns a dictionary; in **ContextMixin** it returns its keyword arguments, but it is common to override this to add more members to the dictionary. You can also use the **extra_context** attribute.

Building up Django's generic class-based views

Let's look at how two of Django's generic class-based views are built out of mixins providing discrete functionality. We'll consider **DetailView**, which renders a "detail" view of an object, and **ListView**, which will render a list of objects, typically from a queryset, and optionally paginate them. This will introduce us to four mixins which between them provide useful functionality when working with either a single Django object, or multiple objects.

There are also mixins involved in the generic edit views (**FormView**, and the model-specific views **CreateView**, **UpdateView** and **DeleteView**), and in the date-based generic views. These are covered in the mixin reference documentation.

DetailView: working with a single Django object

To show the detail of an object, we basically need to do two things: we need to look up the object and then we need to make a **TemplateResponse** with a suitable template, and that object as context.

To get the object, **DetailView** relies on **SingleObjectMixin**, which provides a **get_object()** method that figures out the object based on the URL of the request (it looks for **pk** and **slug** keyword arguments as declared in the URLConf, and looks the object up either from the **model** attribute on the view, or the **queryset** attribute if that's provided). **SingleObjectMixin** also overrides **get_context_data()**, which is used across all Django's built in class-based views to supply context data for template renders.

To then make a **TemplateResponse**, **DetailView** uses **SingleObjectTemplateResponseMixin**, which extends **TemplateResponseMixin**, overriding **get_template_names()** as discussed above. It actually provides a fairly sophisticated set of options, but the main one that most people are going to use is **<app_label>/<model_name>_detail.html**. The **_detail** part can be changed by setting **template_name_suffix** on a subclass to something else. (For instance, the generic edit views use **_form** for create and update views, and **_confirm_delete** for delete views.)

ListView: working with many Django objects

Lists of objects follow roughly the same pattern: we need a (possibly paginated) list of objects, typically a **QuerySet**, and then we need to make a **TemplateResponse** with a suitable template using that list of objects.

To get the objects, **ListView** uses **MultipleObjectMixin**, which provides both **get_queryset()** and **paginate_queryset()**. Unlike with **SingleObjectMixin**, there's no need to key off parts of the URL to figure out the queryset to work with, so the default uses the **queryset** or **model** attribute on the view class. A common reason to override **get_queryset()** here would be to dynamically vary the objects, such as depending on the current user or to exclude posts in the future for a blog.

MultipleObjectMixin also overrides **get_context_data()** to include appropriate context variables for pagination (providing dummies if pagination is disabled). It relies on **object_list** being passed in as a keyword argument, which **ListView** arranges for it.

To make a **TemplateResponse**, **ListView** then uses **MultipleObjectTemplateResponseMixin**; as with **SingleObjectTemplateResponseMixin** above, this overrides **get_template_names()** to provide **a range of options**, with the most commonly-used being **<app_label>/<model_name>_list.html**, with the **_list** part again being taken from the **template_name_suffix** attribute. (The date based generic views use suffixes such as **_archive**, **_archive_year** and so on to use different templates for the various specialized date-based list views.)

Using Django's class-based view mixins¶

Now we've seen how Django's generic class-based views use the provided mixins, let's look at other ways we can combine them. We're still going to be combining them with either built-in class-based views, or other generic class-based views, but there are a range of rarer problems you can solve than are provided for by Django out of the box.

Using SingleObjectMixin with View¶

If we want to write a class-based view that responds only to **POST**, we'll subclass **View** and write a **post()** method in the subclass. However if we want our processing to work on a particular object, identified from the URL, we'll want the functionality provided by **SingleObjectMixin**.

```
from django.http import HttpResponseForbidden, HttpResponseRedirect
from django.urls import reverse
from django.views import View
from django.views.generic.detail import SingleObjectMixin
from books.models import Author


class RecordInterestView(SingleObjectMixin, View):
    """Records the current user's interest in an author."""

    model = Author

    def post(self, request, *args, **kwargs):
        if not request.user.is_authenticated:
            return HttpResponseForbidden()

        # Look up the author we're interested in.
        self.object = self.get_object()
        # Actually record interest somehow here!

        return HttpResponseRedirect(
            reverse("author-detail", kwargs={"pk": self.object.pk})
        )
```

1.4. Implement Authentication, Token, Permission

Setting Up The REST API Project

So let's start from the very beginning. Install Django and DRF:

```
pip install django
pip install djangorestframework
```

Create a new Django project:

```
django-admin.py startproject myapi .
```

Navigate to the **myapi** folder:

```
cd myapi
```

Start a new app. I will call my app **core**:

```
django-admin.py startapp core
```

Here is what your project structure should look like:

```
myapi/
 |-- core/
 |    |-- migrations/
 |    |-- __init__.py
 |    |-- admin.py
 |    |-- apps.py
 |    |-- models.py
 |    |-- tests.py
 |    +-- views.py
 |-- __init__.py
 |-- settings.py
 |-- urls.py
 +-- wsgi.py
manage.py
```

Add the **core** app (you created) and the **rest_framework** app (you installed) to the `INSTALLED_APPS`, inside the **settings.py** module:

myapi/settings.py

```
INSTALLED_APPS = [
    # Django Apps
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    # Third-Party Apps
    'rest_framework',

    # Local Apps (Your project's apps)
```

```
    'myapi.core',
]
```

Return to the project root (the folder where the **manage.py** script is), and migrate the database:

```
python manage.py migrate
```

Let's create our first API view just to test things out:

myapi/core/views.py

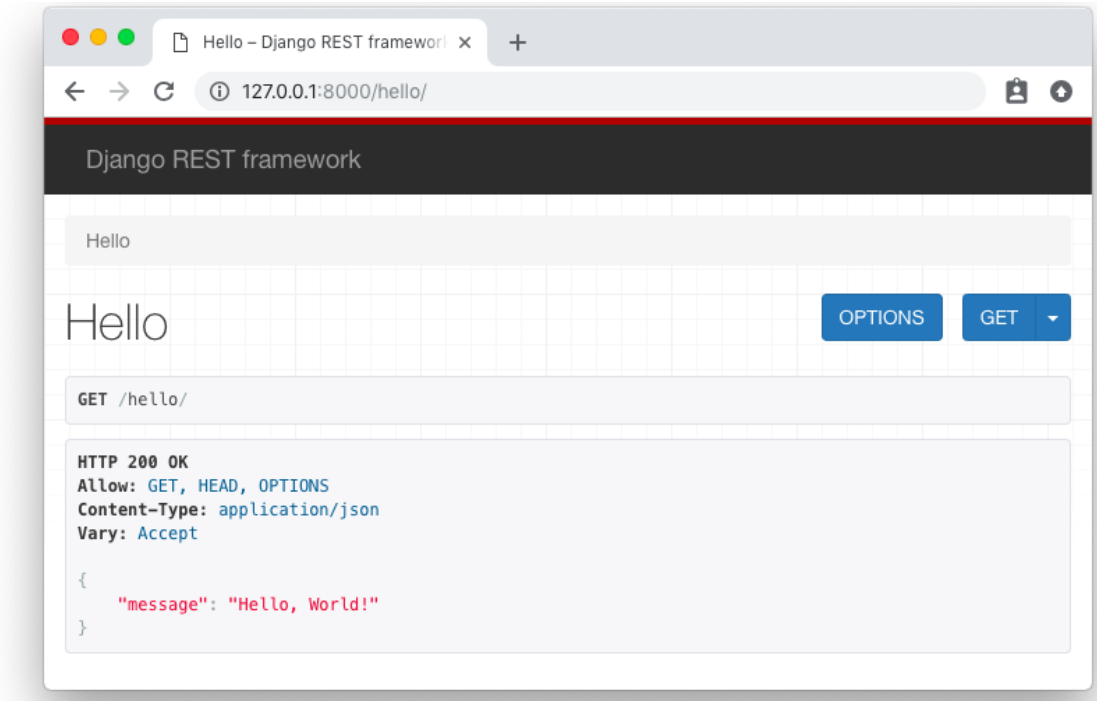
```
from rest_framework.views import APIView
from rest_framework.response import Response

class HelloView(APIView):
    def get(self, request):
        content = {'message': 'Hello, World!'}
        return Response(content)
```

Now register a path in the **urls.py** module:

myapi/urls.py

```
from django.urls import path
from myapi.core import views

urlpatterns = [
    path('hello/', views.HelloView.as_view(), name='hello'),
]
```

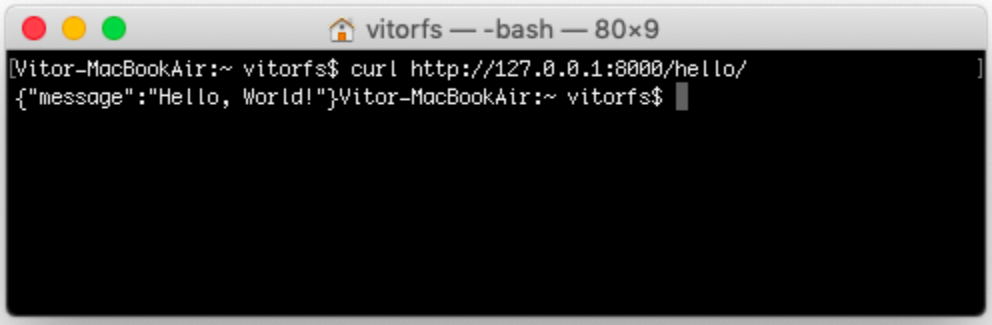
So now we have an API with just one endpoint `/hello/` that we can perform `GET` requests. We can use the browser to consume this endpoint, just by accessing the URL `http://127.0.0.1:8000/hello/`:



We can also ask to receive the response as plain JSON data by passing the `format` parameter in the querystring like `http://127.0.0.1:8000/hello/?format=json`:

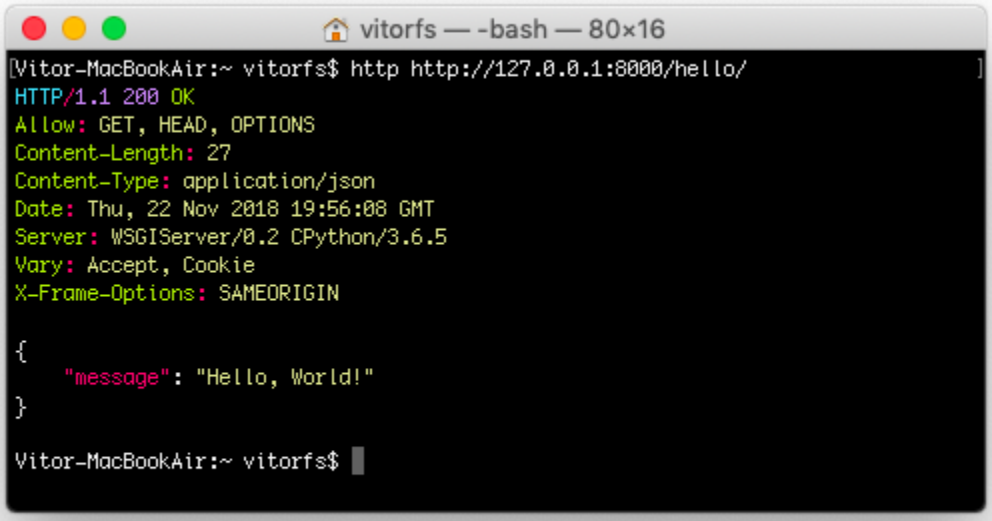
Both methods are fine to try out a DRF API, but sometimes a command line tool is more handy as we can play more easily with the requests headers. You can use cURL, which is widely available on all major Linux/macOS distributions:

```
curl http://127.0.0.1:8000/hello/
```



But usually I prefer to use HTTPie, which is a pretty awesome Python command line tool:

```
http http://127.0.0.1:8000/hello/
```



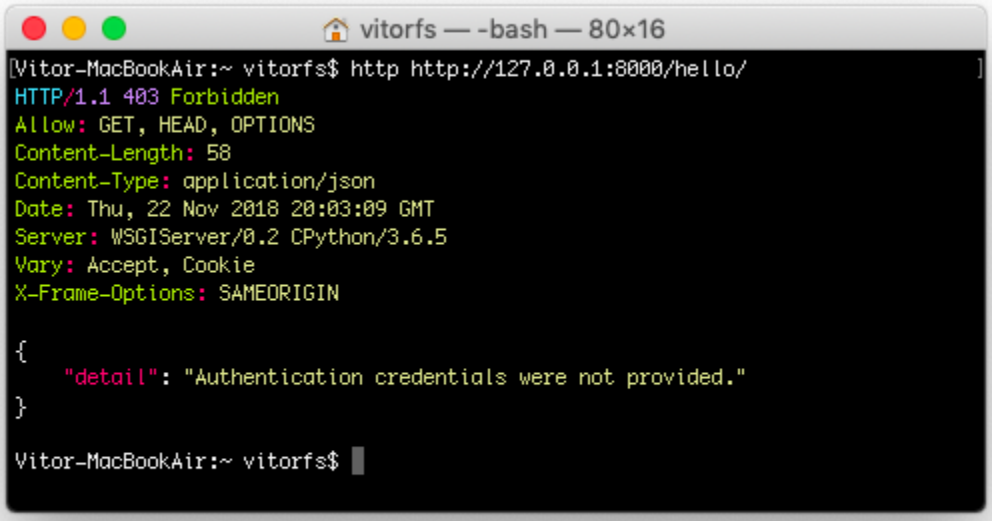
Now let's protect this API endpoint so we can implement the token authentication:

myapi/core/views.py

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework.permissions import IsAuthenticated  # <-- Here


class HelloView(APIView):
    permission_classes = (IsAuthenticated,)             # <-- And
here

    def get(self, request):
        content = {'message': 'Hello, World!'}
        return Response(content)
```

Try again to access the API endpoint:

```
http http://127.0.0.1:8000/hello/
```



And now we get an HTTP 403 Forbidden error. Now let's implement the token authentication so we can access this endpoint.

Implementing the Token Authentication

We need to add two pieces of information in our **settings.py** module. First include **rest_framework.authtoken** to your `INSTALLED_APPS` and include the `TokenAuthentication` to `REST_FRAMEWORK`:

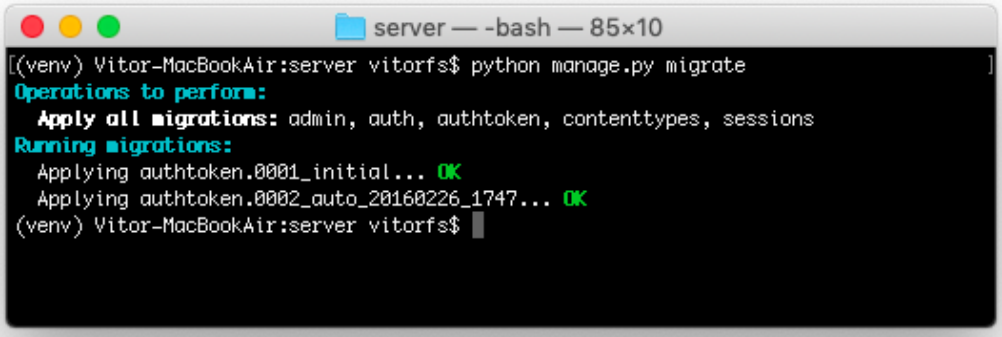
myapi/settings.py

```
INSTALLED_APPS = [
    # Django Apps
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    # Third-Party Apps
    'rest_framework',
    'rest_framework.authtoken',  # <-- Here

    # Local Apps (Your project's apps)
    'myapi.core',
]
```

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.TokenAuthentication',   # <-- And here
    ],
}
```

Migrate the database to create the table that will store the authentication tokens:

```
python manage.py migrate
```

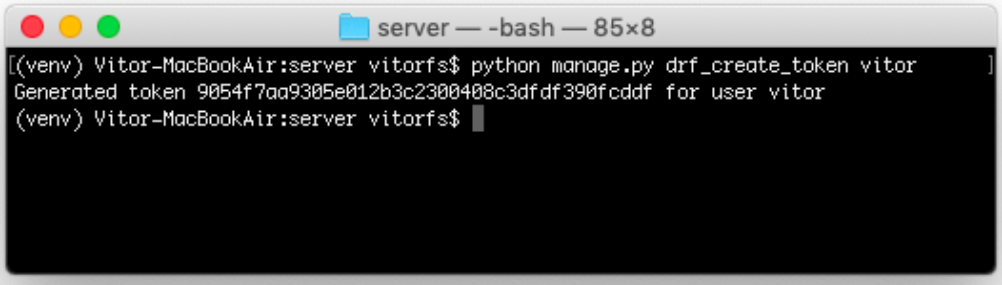


Now we need a user account. Let's just create one using the `manage.py` command line utility:

```
python manage.py createsuperuser --username vitor --email vitor@example.com
```

The easiest way to generate a token, just for testing purpose, is using the command line utility again:

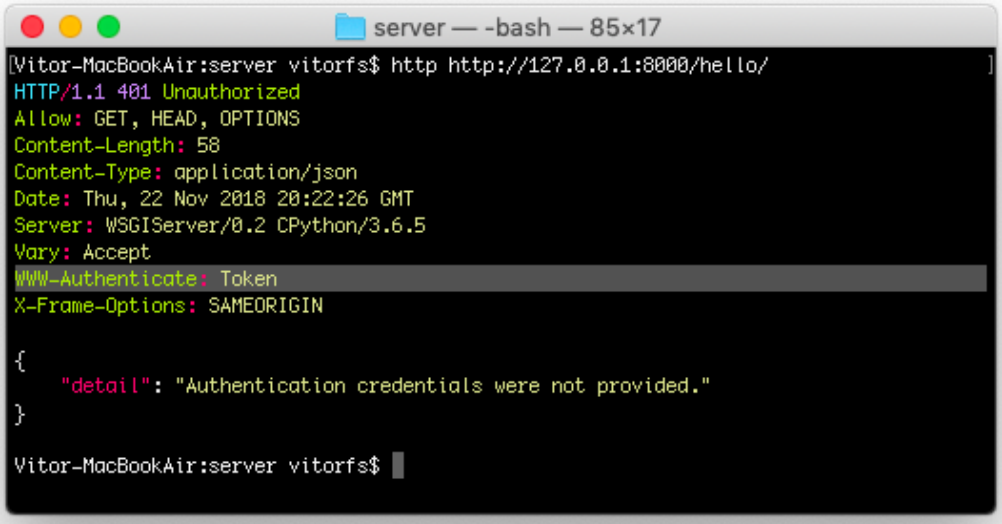
```
python manage.py drf_create_token vitor
```



This piece of information, the random string `9054f7aa9305e012b3c2300408c3dfdf390fcddf` is what we are going to use next to authenticate.

But now that we have the `TokenAuthentication` in place, let's try to make another request to our `/hello/` endpoint:

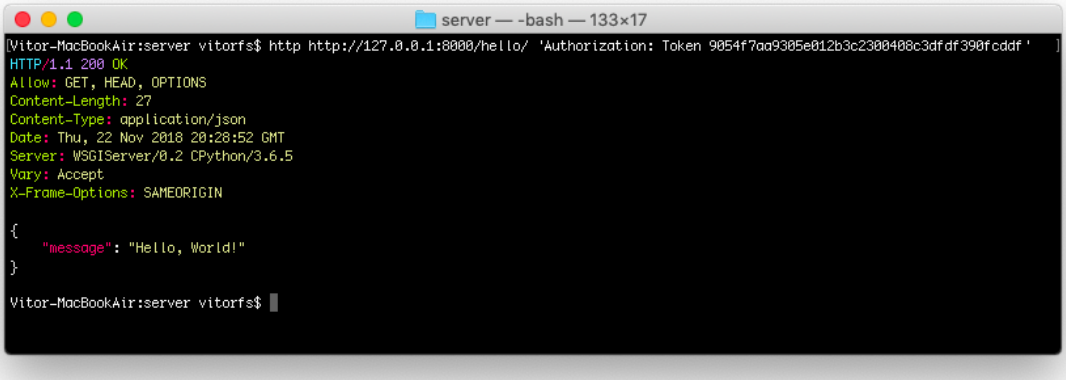
```
http http://127.0.0.1:8000/hello/
```



Notice how our API is now providing some extra information to the client on the required authentication method.

So finally, let's use our token!

```
http http://127.0.0.1:8000/hello/ 'Authorization: Token 9054f7aa9305e012b3c2300408c3dfdf390fcddf'
```



And that's pretty much it. For now on, on all subsequent request you should include the header `Authorization: Token 9054f7aa9305e012b3c2300408c3dfdf390fcddf`.

The formatting looks weird and usually it is a point of confusion on how to set this header. It will depend on the client and how to set the HTTP request header.

For example, if we were using cURL, the command would be something like this:

```
curl http://127.0.0.1:8000/hello/ -H 'Authorization: Token 9054f7aa9305e012b3c2300408c3dfdf390fcddf'
```

Or if it was a Python requests call:

```
import requests

url = 'http://127.0.0.1:8000/hello/'
headers = {'Authorization': 'Token 9054f7aa9305e012b3c2300408c3dfdf390fcddf'}
r = requests.get(url, headers=headers)
```

Or if we were using Angular, you could implement an `HttpInterceptor` and set a header:

```
import { Injectable } from '@angular/core';
import { HttpRequest, HttpHandler, HttpEvent, HttpInterceptor } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const user = JSON.parse(localStorage.getItem('user'));
    if (user && user.token) {
      request = request.clone({
        setHeaders: {
          Authorization: `Token ${user.accessToken}`
        }
      });
    }
    return next.handle(request);
  }
}
```

User Requesting a Token

The DRF provide an endpoint for the users to request an authentication token using their username and password.

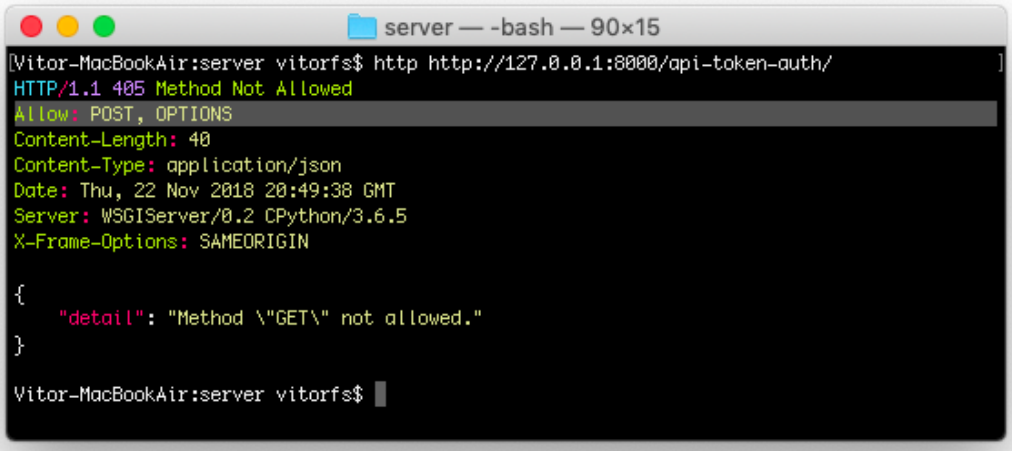
Include the following route to the **urls.py** module:

myapi/urls.py

```
from django.urls import path
from rest_framework.authtoken.views import obtain_auth_token  # <-- Here
from myapi.core import views

urlpatterns = [
    path('hello/', views.HelloView.as_view(), name='hello'),
    path('api-token-auth/', obtain_auth_token, name='api_token_auth'),  # <-- And here
]
```

So now we have a brand new API endpoint, which is `/api-token-auth/`. Let's first inspect it:

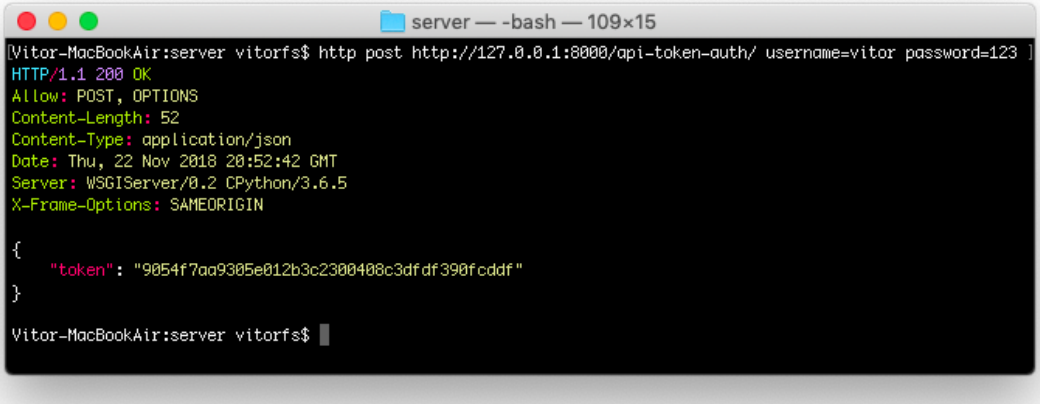
```
http http://127.0.0.1:8000/api-token-auth/
```



It doesn't handle GET requests. Basically it's just a view to receive a POST request with username and password.

Let's try again:

```
http post http://127.0.0.1:8000/api-token-auth/ username=vitor password=123
```



The response body is the token associated with this particular user. After this point you store this token and apply it to the future requests.

Then, again, the way you are going to make the POST request to the API depends on the language/framework you are using.

If this was an Angular client, you could store the token in the `localStorage`, if this was a Desktop CLI application you could store in a text file in the user's home directory in a dot file.

a. Generating Tokens

To automatically generate a token for each user upon creation, you can use Django's signals in combination with Django REST Framework's (DRF) TokenAuthentication. Specifically, Django's post_save signal can trigger token creation whenever a new user instance is saved

1. Import Necessary Modules

In your Django app, create a signals.py file or add these imports in your models.py if you don't want a separate file.

```
#myapp/signals.py

from django.conf import settings
from django.db.models.signals import post_save
from django.dispatch import receiver
from rest_framework.authtoken.models import Token
```

2. Define the Signal

Using the @receiver decorator, define a signal function that listens for the post_save event on the User model. This function will automatically create a token for the user after they are saved to the database

```
#myapp/stonals.py

from django.conf import settings
from django.db.models.signals import post_save
from django.dispatch import receive
from rest framework.authtoken.models import Token

@receiver(post save, sender settings.AUTH_USER_MODEL)
def create auth_token(sender, instance None,created{False, **kwargs):
if created:
    Token.objects.create(user=instance)
```

sender-settings.AUTH_USER_MODEL:

This ensures that the signal is connected to the user model.

instance: Refers to the user instance being saved.

created: Indicates if the user was created (as opposed to updated

3. Connect the Signal in Your App

To make sure the signal is registered, import your signals.py in the app's apps.py file

```
#myapp/apps.py

from django.apps import AppConfig

class MyAppConfig(AppConfig):
     name = 'myapp'
     def ready(self):
     #Import signals module to register it
         import myapp.signals
```

Then, set this custom app config in your settings.py

```
# settings.py

INSTALLED_APPS = [
     'myapp.apps.MyAppConfig',
]
```

4. Run Migrations

Ensure that the token model is properly set up by running migrations if you haven't already

5. Test the Setup

After creating a new user, you should see a token automatically generated for that user in the database

1.5. Implement Searching, filtering, and pagination

When developing APIs for modern web applications, it’s essential to implement features like pagination, search, and filtering to improve user experience and optimize performance. In this blog post, we’ll break down how to implement a **GET API** using Node.js, Express, and MongoDB that handles **pagination**, **search** (filtering), and **sorting** for a test series.

1. Setting Up the API Endpoint

We're using an Express router to define an API route for fetching test series. Below is the basic structure of the route handler for the /api/testSeries endpoint.

```
router.get('/api/testSeries', async (req, res) => {
  try {
    // Pagination, filtering, and sorting logic will go here
  } catch (error) {
    res.status(500).json({ message: 'Internal Server Error' });
  }
});
```

2. Pagination

Pagination is used to break large datasets into smaller chunks, making it easier to load, display, and navigate through large collections of data. In the API, pagination is controlled by two query parameters:

- page: The current page number, starting from 1.
- limit: The number of items to be displayed on each page.

```
// Pagination
const page = parseInt(req.query.page) || 1;   // Default to page 1
const limit = parseInt(req.query.limit) || 10; // Default limit of 10 items
```

If no values for page or limit are provided, the default values of page 1 and limit 10 are used.

- We calculate the **startIndex** and **endIndex** based on these values, which helps in slicing the data for pagination.

3. Search and Filtering

In most cases, users will want to search or filter items based on specific fields, such as the name, category, or sub-category of the test series. For this, we construct a **filter object** based on the query parameter searchTerm.

The filter is built using the MongoDB $regex operator to perform case-insensitive partial text matching. Here's how we set it up:

```
// Filtering
const filter = {};
if (req.query.searchTerm) {
  filter.$or = [
    { name: { $regex: req.query.searchTerm, $options: 'i' } },
    { category: { $regex: req.query.searchTerm, $options: 'i' } },
    { subCategory: { $regex: req.query.searchTerm, $options: 'i' } }
  ];
}
```

- The $or operator allows us to search across multiple fields.
- The $regex operator performs a text search based on the provided searchTerm, with the i option for case-insensitivity.

4. Sorting

Sorting allows users to view data in a specific order, such as alphabetically or by date. Sorting is handled by the sortBy query parameter, which consists of two parts: the field to sort by and the sort order (ascending or descending).

```
// Sorting
const sort = {};
if (req.query.sortBy) {
  const [field, order] = req.query.sortBy.split(':');
  sort[field] = order === 'desc' ? -1 : 1;
}
```

- We split the sortBy query parameter using : to separate the field and the sort order.
- order === 'desc' ? -1 : 1 ensures that the sorting order is descending (-1) or ascending (1), based on the provided query parameter.

5. Fetching the Data

Once the **pagination**, **filtering**, and **sorting** configurations are set, we fetch the data from MongoDB using Mongoose's .find() method. We apply the **filter** and **sort** conditions and then slice the data for pagination.

```
// Fetch all matching test series without pagination for global search
const allMatchingTestSeries = await TestSeries.find(filter).sort(sort);

// Apply pagination to the results
const totalTestSeries = allMatchingTestSeries.length;
const startIndex = (page - 1) * limit;
const endIndex = startIndex + limit;
const paginatedTestSeries = allMatchingTestSeries.slice(startIndex, endIndex);
```

- We first retrieve all matching documents from the database using the .find(filter) method.
- Then, we apply the sorting using .sort(sort).
- Finally, we slice the results using the slice(startIndex, endIndex) method to implement pagination.

6. Sending the Response

After retrieving the data, we send the response to the client in JSON format. The response includes the **paginated test series**, **total number of pages**, **current page**, and **total number of items**.

```
res.json({
  testSeries: paginatedTestSeries,
  totalPages: Math.ceil(totalTestSeries / limit),
  currentPage: page,
  totalItems: totalTestSeries
});
```

- totalPages: The total number of pages, calculated by dividing the total number of items by the page limit.
- currentPage: The current page number.
- totalItems: The total number of matching items in the database.

7. Handling Errors

To handle any potential errors that may arise during the execution of the API (such as database connectivity issues), we wrap the entire code inside a try-catch block. If an error occurs, we send a 500 status code and an error message.

```
catch (error) {
  console.error('Error fetching test series:', error);
  res.status(500).json({ message: 'Internal Server Error' });
}
```

Self-Check Sheet 1: Apply DRF concepts

1. **What is the primary function of a serializer in Django REST framework?**

Answer:

- A. To handle user authentication requests
- B. To define the representation of data records for REST services
- C. To manage database connections and queries
- D. To render HTML templates for web applications

2. **Which of the following statements is most accurate about serializers in Django REST framework?**

Answer:

- A. They are always complex classes requiring extensive coding.
- B. They can inherit from base classes like `serializers.Serializer` for simpler implementations.
- C. They are not necessary for building REST services with Django REST framework.
- D. They are responsible for handling incoming HTTP requests.

3. **What is the advantage of using class-based views in Django REST framework compared to regular Django views decorated with `@api_view`?**

Answer:

- A. Class-based views offer no significant advantages.
- B. Class-based views provide a more readable and organized approach for complex REST services.
- C. Class-based views are mandatory for using serializers in Django REST framework.
- D. Class-based views are not compatible with generic views like `ListCreateAPIView`.

4. **What is the role of mixins in Django REST framework?**

Answer:

- A. Mixins are used to define custom authentication schemes for REST services.
- B. Mixins encapsulate common logic for REST services, promoting code reuse.)
- C. Mixins are a mandatory component for using generic class-based views.
- D. Mixins are primarily used for handling database transactions.

5. **What is the benefit of using viewsets and routers in Django REST framework?**

Answer:

- A. Viewsets and routers simplify the creation of individual views for each CRUD operation.
- B. Viewsets and routers provide a more concise way to define views for complex REST services.
- C. Viewsets and routers are necessary for using authentication with Django REST framework.
- D. Viewsets and routers offer no advantages and are optional for building REST services.

Answer Key 1: Apply DRF concepts

1. **What is the primary function of a serializer in Django REST framework?**
    - A. To handle user authentication requests
    - **B. To define the representation of data records for REST services (Correct)**
    - C. To manage database connections and queries
    - D. To render HTML templates for web applications

2. **Which of the following statements is most accurate about serializers in Django REST framework?**
    - A. They are always complex classes requiring extensive coding.
    - **B. They can inherit from base classes like `serializers.Serializer` for simpler implementations. (Correct)**
    - C. They are not necessary for building REST services with Django REST framework.
    - D. They are responsible for handling incoming HTTP requests.

3. **What is the advantage of using class-based views in Django REST framework compared to regular Django views decorated with `@api_view`?**
    - A. Class-based views offer no significant advantages.
    - **B. Class-based views provide a more readable and organized approach for complex REST services. (Correct)**
    - C. Class-based views are mandatory for using serializers in Django REST framework.
    - D. Class-based views are not compatible with generic views like `ListCreateAPIView`.

4. **What is the role of mixins in Django REST framework?**
    - A. Mixins are used to define custom authentication schemes for REST services.
    - **B. Mixins encapsulate common logic for REST services, promoting code reuse. (Correct)**
    - C. Mixins are a mandatory component for using generic class-based views.
    - D. Mixins are primarily used for handling database transactions.

5. **What is the benefit of using viewsets and routers in Django REST framework?**
    - A. Viewsets and routers simplify the creation of individual views for each CRUD operation.
    - **B. Viewsets and routers provide a more concise way to define views for complex REST services. (Correct)**
    - C. Viewsets and routers are necessary for using authentication with Django REST framework.
    - D. Viewsets and routers offer no advantages and are optional for building REST services.

Job Sheet-1: **Implement Serializers and Views in Django REST Framework**

UoC Cover

OU-ICT-WADP-02- L7-V1: Creating API Using Django REST Framework

Working Procedure / Steps

1. **Create Serializers:**
    - Serializers define the representation of data records in a Django REST framework.
    - They are used to convert Django model instances into other formats like JSON or XML.
2. **Define Serializer Class:**
    - Create a Python class that inherits from `serializers.Serializer` or a more specific
    - Use fields from `serializers.py` to define the data transformation for each model field.
3. **Create Views:**
    - Views handle incoming requests, process data, and return responses.
    - Django REST framework offers several view options:
        - **Regular Django View with `@api_view` decorator:**
            - Use this for basic REST services.
        - **Class-based Views:**
            - More concise and reusable than regular views.
        - **Generic Class-based Views:**
            - Provide pre-built functionalities for common CRUD operations.
        - **View Sets:**
            - Group multiple related views together.

Specification Sheet 1: **Implement Serializers and Views in Django REST Framework**

Technical Requirements

- **Programming Language:** Python 3.7 or later
- **Framework:** Django 3.2 or later
- **Django REST framework:** Additional library for building REST APIs

Tools and Equipment

- **Computer:** A computer with sufficient processing power, RAM, and storage.
- **Terminal:** A command-line interface for executing commands.
- **Text Editor/IDE:** Visual Studio Code, PyCharm, Sublime Text (or any preferred code editor).

Learning Outcome 2: Create CRUD project

Assessment Criteria	1. Project layout is created 2. Application setup 3. Database is set up 4. Blue print and views are applied 5. Project is made installable
Conditions and Resources	1. Real or simulated workplace 2. CBLM 3. Handouts 4. Laptop 5. Multimedia Projector 6. Paper, Pen, Pencil, Eraser 7. Internet facilities 8. White board and marker 9. Audio Video Device
Contents	1. CRUD project 2. API Testing
Activities/job/Task	2. Create a CRUD Django Application
Training Methods	1. Discussion 2. Presentation 3. Demonstration 4. Guided Practice 5. Individual Practice 6. Project Work 7. Problem Solving 8. Brainstorming
Assessment Methods	Assessment methods may include but not limited to 1. Written Test 2. Demonstration 3. Oral Questioning 4. Portfolio

Learning Experience 2: Create CRUD project

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
1. Trainee will ask the instructor about the learning materials	1. Instructor will provide the learning materials 'Create CRUD project'
2. Read the Information sheet and complete the Self Checks & Check answer sheets on "Create CRUD project"	2. Read Information sheet 2: Create CRUD project 3. Answer Self-check 2: Create CRUD project 4. Check your answer with Answer key 2: Create CRUD project
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Job Sheet-2: Create a CRUD Django Application

Information sheet 2: Create CRUD project

Learning Objective:

After completion of this Information sheet , the learners will be able to explain, define and interpret the following contents:

2.1. Project layout is created
2.2. Application setup
2.3. Database is set up
2.4. Blue print and views are applied
2.5. Project is made installable

2.1. Create Project layout

Project Organization:

Your Django project will be organized into several key directories, each serving a specific purpose:

- your_project_name/ (Main Project Directory):
    - **manage.py:** This script acts as the command center for your project. Use it to run various tasks like starting the development server, creating apps, managing database migrations, and more.
    - **your_app_name/ (App Directory):** - Replace your_app_name with the actual name of your application (e.g., blog, products). This directory houses all the code specific to your CRUD functionality.
        - **admin.py (Optional):** This file allows you to register your data models with the Django admin interface, providing a user-friendly way to manage data through a web interface.
        - **apps.py (Optional):** This file (optional) serves as a configuration file for your app. It typically specifies the app name and any dependencies it might have on other Django apps.
        - **migrations/:** This directory stores database migration files. These files track changes made to your data models and facilitate updating your database schema when the models evolve.
        - **models.py:** This core file defines your data models. Here, you'll create model classes representing the data you intend to

manage (e.g., Post model for a blog, Product model for an e-commerce application).

- **tests.py (Optional):** This file is an excellent practice to include unit tests for your app's functionalities. Writing tests ensures your code works as expected and helps prevent regressions during future development.
- **views.py:** This file defines the functions (views) responsible for handling user requests and generating responses. These responses can be HTML pages, JSON data, or other formats depending on your application's needs.
- **urls.py:** This file defines the URL patterns for your app. It maps incoming URLs to specific views within your app, ensuring the correct view handles each user request.
- **templates/ (Optional):** This directory (optional) holds HTML templates used for rendering dynamic content in your views.
    - **your_app_name/ (App-Specific Template Directory):** This subdirectory within templates/ contains the HTML template files specific to your app's views (e.g., list.html for displaying a list of items, detail.html for showing details of a single item, create.html for creating a new item, update.html for editing an existing item).

- **settings.py:** This critical file holds essential configuration settings for your entire Django project. It defines details like the database connection, installed apps (including your your_app_name app), secret keys for security purposes, and other project-wide settings.
- **urls.py:** This file defines the main project-level URL patterns. It typically includes patterns that delegate incoming requests to specific app URL configurations (e.g., your_app_name.urls).
- **wsgi.py:** This file serves as the entry point for the Web Server Gateway Interface (WSGI) server. It provides a way for web servers like Apache or Nginx to interact with your Django application.

2.2. Application setup

Django is a powerful Python web framework that simplifies web development by providing a clean and pragmatic design. One of the most common tasks in web development is creating CRUD (Create, Read, Update, Delete) functionality for your application. In this article, we'll explore how to create a Django CRUD project using function-based views.

Prerequisites

Before we dive into building our CRUD project, make sure you have the following prerequisites in place:

1. Python and Django: Ensure you have Python installed on your system. You can install Django using pip:

pip install django

2. Database: Decide on the database you want to use. By default, Django uses SQLite, but you can configure it to use other databases like PostgreSQL, MySQL, or Oracle.
3. Text Editor or IDE: Choose a code editor or integrated development environment (IDE) of your preference. Popular choices include Visual Studio Code, PyCharm, or Sublime Text.

Setting Up Your Django Project

Let's start by creating a new Django project and a new app within that project. Open your terminal and run the following commands:

django-admin startproject crudproject

cd crudproject

python manage.py startapp crudapp

We've created a new project named "crudproject" and an app named "crudapp."

Application Registration: you need to configure in your settings.py file

Make sure your app (myapp) is included in the INSTALLED_APPS list:

```
INSTALLED_APPS = [
    # ...
    'myapp',
]
```

Defining Models

In Django, models are Python classes that define the structure of your database tables. For our CRUD project, let's assume we want to manage a list of orders. Create a model for the orders in crudapp/models.py:

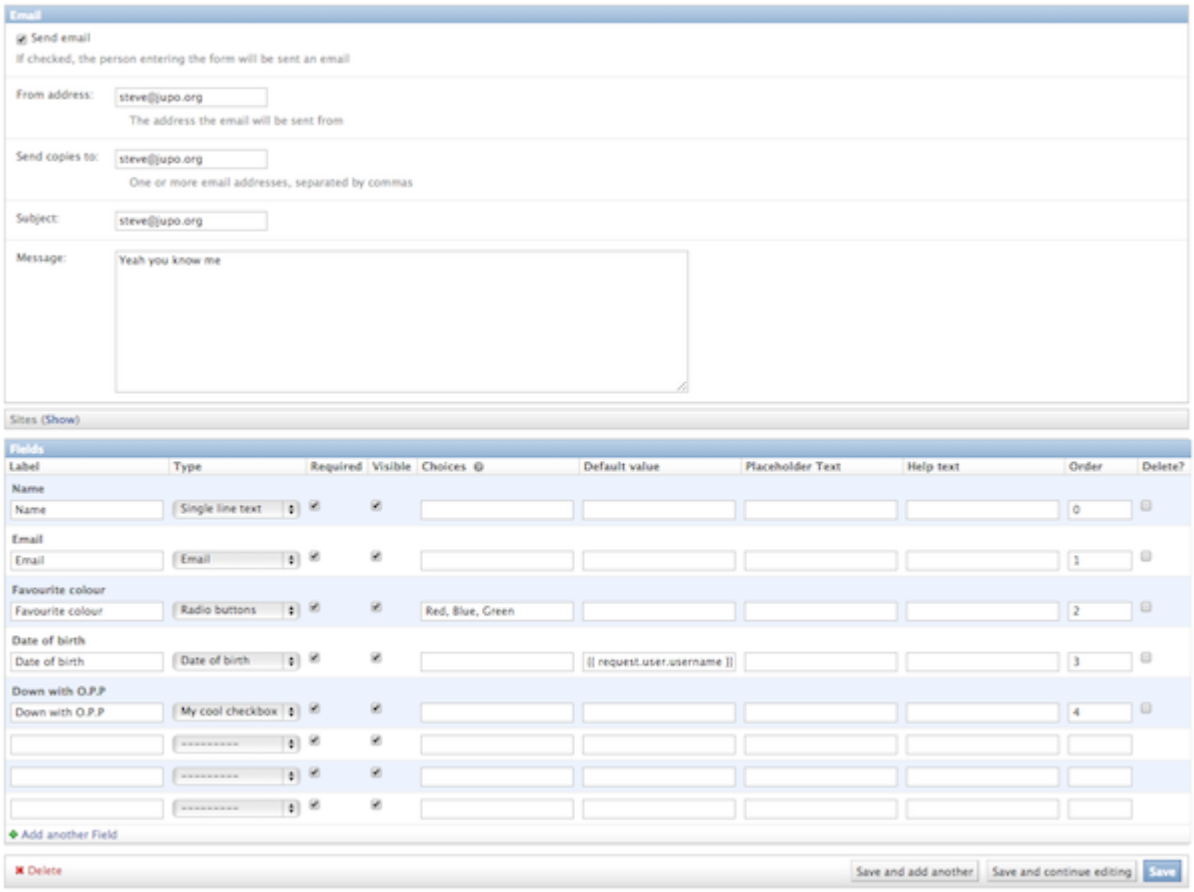
from django.db import models

```
# Create your models here.
class Orders(models.Model):
    oid = models.IntegerField(primary_key=True)
    fname = models.CharField(max_length=20)
    lname = models.CharField(max_length=20)
    price = models.FloatField()
    mail = models.EmailField()
    addr = models.CharField(max_length=50)
```

Now, it's time to create the database tables for our models. Run the following commands to create the migrations and apply them:

```
python manage.py makemigrations
python manage.py migrate
```

Creating Forms

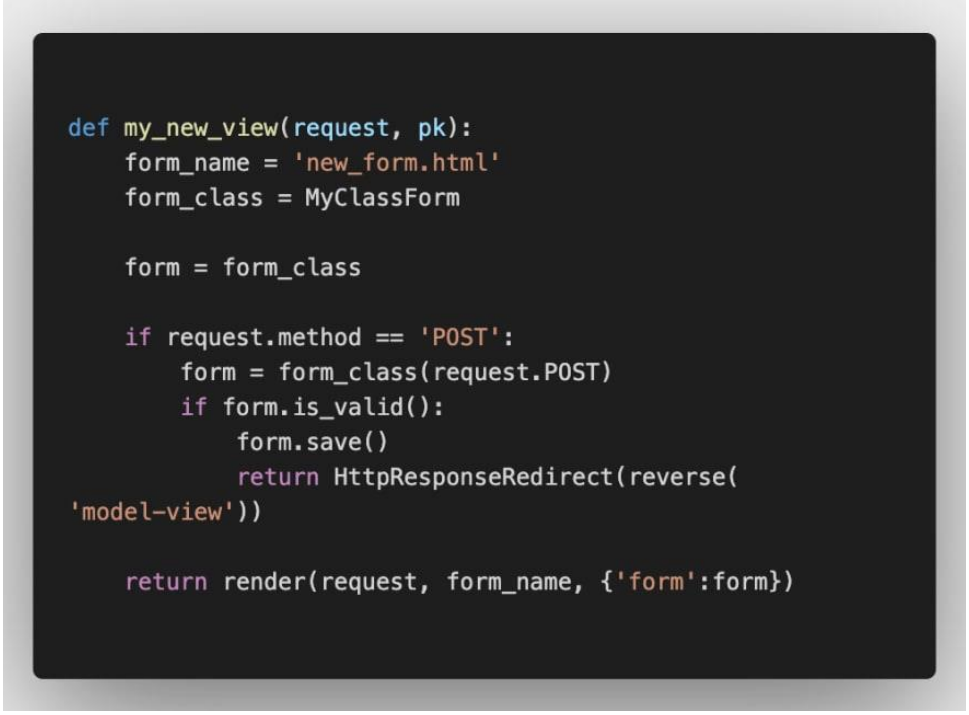


We mentioned using a form for creating and updating orders. You can define the form in crudapp/forms.py:

```
from django import forms
```

```
from .models import Orders

class OrderForm(forms.ModelForm):
   class Meta:
      model = Orders
      fields = '__all__'

      labels = {
         'oid': 'Order ID',
         'fname' : 'First Name',
         'lname' : 'Last Name.' ,
         'price' : 'Price' ,
         'mail' : 'Email ID',
         'addr' : 'Address' ,
      }

      widgets  ={
         'oid' : forms.NumberInput(attrs={'placeholder': 'eg. 101'}),
         'fname' : forms.TextInput(attrs={'placeholder': 'eg. Prosenjeet'}),
         'lname' : forms.TextInput(attrs={'placeholder': 'eg. Shil'}),
         'price' : forms.NumberInput(attrs={'placeholder': 'eg. 10000'}),
         'mail' : forms.EmailInput(attrs={'placeholder': 'eg. abc@xyz.com'}),
         'addr' : forms.Textarea(attrs={'placeholder': 'eg. IN'}),
      }
```

Creating Function-Based Views



```
def my_new_view(request, pk):
    form_name = 'new_form.html'
    form_class = MyClassForm

    form = form_class

    if request.method == 'POST':
        form = form_class(request.POST)
        if form.is_valid():
            form.save()
            return HttpResponseRedirect(reverse(
'model-view'))

    return render(request, form_name, {'form':form})
```

Function-based views are a simple and straightforward way to handle CRUD operations in Django. In this example, we'll create views for creating, reading, updating, and deleting orders.

Create a Order (Create View) In crudapp/views.py, define a view function for creating new order:

```
from django.shortcuts import redirect, render
from .forms import OrderForm
from .models import Orders

# Create your views here.
def orderFormView(request):
    form = OrderForm()
    if request.method == 'POST':
        form = OrderForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('show_url')
    template_name = 'crudapp/order.html'
    context = {'form': form}
    return render(request, template_name, context)
```

In this view, we handle both GET and POST requests. If it's a GET request, we render a form for creating a new order. If it's a POST request, we validate the form data and save the new order if it's valid.

- Read Orders (List View) Now, let's create a view to display a list of all books in crudapp/views.py:

```
def showView(request):
    obj = Orders.objects.all()
    template_name = 'crudapp/show.html'
    context = {'obj': obj}
    return render(request, template_name, context)
```

This view retrieves all orders from the database and renders them using a template.

- Update a Order (Update View) To update a order, create a view in crudapp/views.py:

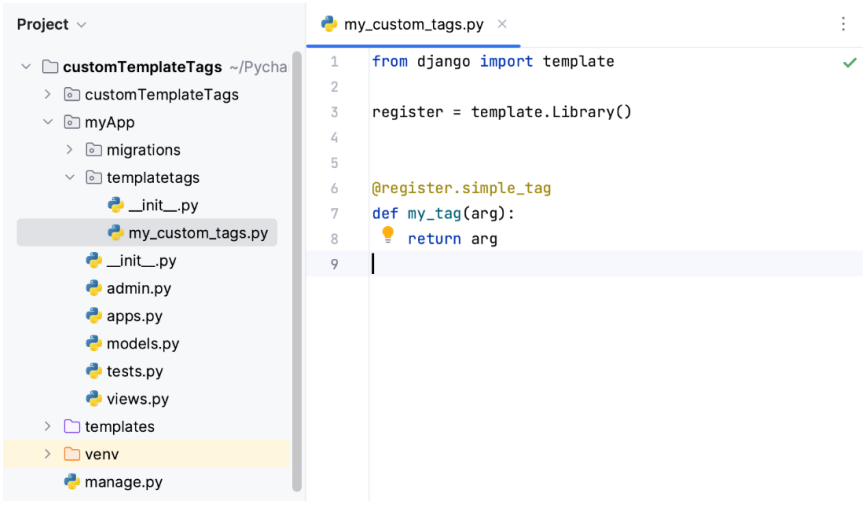
```
def updateView(request, f_oid):
    obj = Orders.objects.get(oid=f_oid)
    form = OrderForm(instance=obj)
    if request.method == 'POST':
        form = OrderForm(request.POST, instance=obj)
        if form.is_valid():
            form.save()
            return redirect('show_url')
    template_name = 'crudapp/order.html'
    context = {'form': form}
    return render(request, template_name, context)
```

- Delete a Order (Delete View) Finally, let's create a view to delete a order in crudapp/views.py:

```
def deleteView(request, f_oid):
    obj = Orders.objects.get(oid=f_oid)
    if request.method == 'POST':
        obj.delete()
        return redirect('show_url')
    template_name = 'crudapp/confirmation.html'
    context = {'obj': obj}
    return render(request, template_name, context)
```

In this view, we confirm the order deletion with a confirmation page.

Creating Templates



Now, create HTML templates for the views in the crudproject/templates directory. You'll need templates for the following views:

layout.html: for creating base html file with navbar.

Similarly, create HTML templates for the views in the crudproject/templates/crudapp directory. You'll need templates for the following views:

order.html: For the create and update forms. show.html: For listing all orders. confirmation.html: For confirming order deletion.

Below, are the templates for base file and the three views we discussed earlier:

crudproject/templates/layout.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset='utf-8'>
  <meta http-equiv='X-UA-Compatible' content='IE=edge'>
  {% block title %}
  <title>Layout Page</title>
  {% endblock %}
  <meta name='viewport' content='width=device-width, initial-scale=1'>
  <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@4.0.0/dist/css/bootstrap.min.css"
integrity="sha384-
Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6J
Xm" crossorigin="anonymous">
</head>
<body>
  <nav class="navbar navbar-expand-lg navbar-light bg-light">
    <a class="navbar-brand" href="#">CRUD APP</a>
    <button class="navbar-toggler" type="button" data-toggle="collapse" data-
target="#navbarSupportedContent" aria-controls="navbarSupportedContent" aria-
expanded="false" aria-label="Toggle navigation">
      <span class="navbar-toggler-icon"></span>
    </button>
```

```
    <div class="collapse navbar-collapse" id="navbarSupportedContent">
      <ul class="navbar-nav mr-auto">
        <li class="nav-item active">
          <a class="nav-link" href="{% url 'order_url' %}">Add Orders<span class="sr-only">(current)</span></a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="{% url 'show_url' %}">Show Orders</a>
        </li>
      </ul>
    </div>
  </nav>

  {% block content %}
  {% endblock %}

  <script src="https://code.jquery.com/jquery-3.2.1.slim.min.js" integrity="sha384-KJ3o2DKtIkvYIK3UENzmM7KCkRr/rE9/Qpg6aAZGJwFDMVNA/GpGFF93hXpG5KkN" crossorigin="anonymous"></script>
  <script src="https://cdn.jsdelivr.net/npm/popper.js@1.12.9/dist/umd/popper.min.js" integrity="sha384-ApNbgh9B+Y1QKtv3Rn7W3mgPxhU9K/ScQsAP7hUibX39j7fakFPskvXusvfa0b4Q" crossorigin="anonymous"></script>
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@4.0.0/dist/js/bootstrap.min.js" integrity="sha384-JZR6Spejh4U02d8jOt6vLEHfe/JQGiRRSQQxSfFWpi1MquVdAyjUar5+76PVCmYl" crossorigin="anonymous"></script>
</body>
</html>
```

crudproject/templates/crudapp/orders.html

```
{% extends 'layout.html' %}
{% load crispy_forms_tags %}

{% block title %}
  <title>Add Page</title>
{% endblock %}
```

```
{% block content %}
<center><h1>Order Form</h1></center>
<div class="container">
  <form method="post" class="jumbotron">
     {% csrf_token %}
     {{form|crispy}}
     <input type="submit" value="Place Order" class="btn btn-success">
  </form>
</div>
{% endblock %}
```

crudproject/templates/crudapp/show.html

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

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

{% block content %}
<center><h1>Show Orders</h1></center>
<table class="table">
  <thead>
    <tr>
      <th scope="col">Order ID</th>
      <th scope="col">First Name</th>
      <th scope="col">Last Name</th>
      <th scope="col">Price</th>
      <th scope="col">Email ID</th>
      <th scope="col">Address</th>
      <th scope="col">Actions</th>
    </tr>
  </thead>
  <tbody>
  {% for i in obj %}
    <tr>
      <td>{{i.oid}}</td>
      <td>{{i.fname}}</td>
```

```
      <td>{{i.lname}}</td>
      <td>{{i.price}}</td>
      <td>{{i.mail}}</td>
      <td>{{i.addr}}</td>
      <td>
         <button class="btn btn-warning"><a href="{% url 'update_url' i.oid
%}">Update</a></button>  
         <button class="btn btn-danger"><a href="{% url 'delete_url' i.oid
%}">Delete</a></button>
      </td>
    </tr>
  {% endfor %}
  </tbody>
 </table>
{% endblock %}
```

crudproject/templates/crudapp/confirmation.html

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

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

{% block content %}
   <div class="container">
      <form class="jumbotron" method="post">
         {% csrf_token %}
         <h2>Are you sure you want to delete this data? </h2>
         <input type="submit" value="YES" class="btn btn-danger">
         <button class="btn btn-success"><a href="{% url 'show_url'
%}">No</a></button>
      </form>
   </div>
{% endblock %}
```

Add Bootstrap Crispy forms for styling form and navbar

```
INSTALLED_APPS = [
    # ...
    'crispy_forms',
    'crispy_bootstrap5',
]

CRISPY_TEMPLATE_PACK = 'bootstrap5'
```

Wiring Up URLs

Finally, configure the URLs for your views. In your project's crudproject/urls.py file, include the URLs for the crudapp app:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('crudapp.urls'))
]
```

Then, in your app's crudpp/urls.py file, define the URLs for your views:

```
from django.urls import path
from . import views

urlpatterns = [
    path('ofv/', views.orderFormView, name='order_url'),
    path('sv/', views.showView, name='show_url'),
    path('up/<int:f_oid>', views.updateView, name= 'update_url'),
    path('del/<int:f_oid>', views.deleteView, name= 'delete_url'),
]
```

Testing Your CRUD Project

With everything set up, you can start your Django development server:

```
python manage.py runserver
```

2.3. Setup Database

Create a database **djangodb** in mysql, and configure into the **settings.py** file of django project. See the example.

// **settings.py**

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

2.4. Blueprint and views

Create a Model

Put the following code into **models.py** file.

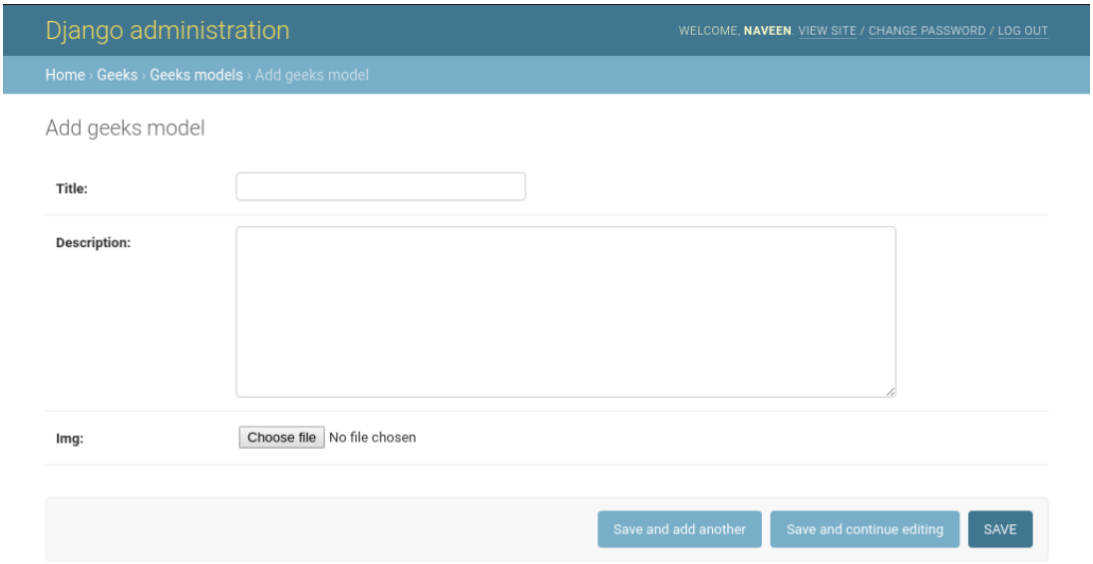
// **models.py**

```
from django.db import models
class Employee(models.Model):
    eid = models.CharField(max_length=20)
    ename = models.CharField(max_length=100)
    eemail = models.EmailField()
    econtact = models.CharField(max_length=15)
    class Meta:
        db_table = "employee"
```

Create a ModelForm

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

```
# declare a new model with a name "GeeksModel"
class GeeksModel(models.Model):
        # fields of the model
    title = models.CharField(max_length = 200)
    description = models.TextField()
    last_modified = models.DateTimeField(auto_now_add = True)
    img = models.ImageField(upload_to = "images/")

        # renames the instances of the model
        # with their title name
    def __str__(self):
        return self.title
```



,

To create a form directly for this model, dive into geeks/forms.py and Enter following code,

```
// forms.py
    from django import forms
    from employee.models import Employee
    class EmployeeForm(forms.ModelForm):
        class Meta:
            model = Employee
            fields = "__all__"
```

Create View Functions

// **views.py**

```
from django.shortcuts import render, redirect
from employee.forms import EmployeeForm
from employee.models import Employee
# Create your views here.
def emp(request):
    if request.method == "POST":
        form = EmployeeForm(request.POST)
        if form.is_valid():
            try:
                form.save()
                return redirect('/show')
            except:
                pass
    else:
        form = EmployeeForm()
    return render(request,'index.html',{'form':form})
def show(request):
    employees = Employee.objects.all()
    return render(request,"show.html",{'employees':employees})
def edit(request, id):
    employee = Employee.objects.get(id=id)
    return render(request,'edit.html', {'employee':employee})
def update(request, id):
    employee = Employee.objects.get(id=id)
    form = EmployeeForm(request.POST, instance = employee)
    if form.is_valid():
        form.save()
        return redirect("/show")
    return render(request, 'edit.html', {'employee': employee})
def destroy(request, id):
    employee = Employee.objects.get(id=id)
    employee.delete()
    return redirect("/show")
```

Provide Routing

Provide URL patterns to map with views function.

// **urls.py**

```
from django.contrib import admin
from django.urls import path
from employee import views
urlpatterns = [
   path('admin/', admin.site.urls),
   path('emp', views.emp),
   path('show',views.show),
   path('edit/<int:id>', views.edit),
   path('update/<int:id>', views.update),
   path('delete/<int:id>', views.destroy),
]
```

Organize Templates

Create a **templates** folder inside the **employee** app and create three (index, edit, show) html files inside the directory. The code for each is given below.

// **index.html**

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Index</title>
   {% load staticfiles %}
   <link rel="stylesheet" href="{% static 'css/style.css' %}"/>
</head>
<body>
<form method="POST" class="post-form" action="/emp">
      {% csrf_token %}
   <div class="container">
<br>
```

```
 <div class="form-group row">
 <label class="col-sm-1 col-form-label"></label>
 <div class="col-sm-4">
 <h3>Enter Details</h3>
 </div>
</div>
 <div class="form-group row">
 <label class="col-sm-2 col-form-label">Employee Id:</label>
 <div class="col-sm-4">
  {{ form.eid }}
 </div>
</div>
<div class="form-group row">
 <label class="col-sm-2 col-form-label">Employee Name:</label>
 <div class="col-sm-4">
  {{ form.ename }}
 </div>
</div>
 <div class="form-group row">
 <label class="col-sm-2 col-form-label">Employee Email:</label>
 <div class="col-sm-4">
  {{ form.eemail }}
 </div>
</div>
 <div class="form-group row">
 <label class="col-sm-2 col-form-label">Employee Contact:</label>
 <div class="col-sm-4">
  {{ form.econtact }}
 </div>
</div>
 <div class="form-group row">
 <label class="col-sm-1 col-form-label"></label>
 <div class="col-sm-4">
 <button type="submit" class="btn btn-primary">Submit</button>
 </div>
</div>
 </div>
```

```
</form>
</body>
</html>
```

// show.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Employee Records</title>
    {% load staticfiles %}
    <link rel="stylesheet" href="{% static 'css/style.css' %}"/>
</head>
<body>
<table class="table table-striped table-bordered table-sm">
    <thead class="thead-dark">
    <tr>
        <th>Employee ID</th>
        <th>Employee Name</th>
        <th>Employee Email</th>
        <th>Employee Contact</th>
        <th>Actions</th>
    </tr>
    </thead>
    <tbody>
{% for employee in employees %}
    <tr>
        <td>{{ employee.eid }}</td>
        <td>{{ employee.ename }}</td>
        <td>{{ employee.eemail }}</td>
        <td>{{ employee.econtact }}</td>
        <td>
            <a href="/edit/{{ employee.id }}"><span class="glyphicon glyphicon-
pencil" >Edit</span></a>
            <a href="/delete/{{ employee.id }}">Delete</a>
        </td>
```

```
    </tr>
  {% endfor %}
    </tbody>
  </table>
  <br>
  <br>
  <center><a href="/emp" class="btn btn-
  primary">Add New Record</a></center>
  </body>
  </html>
```

// edit.html

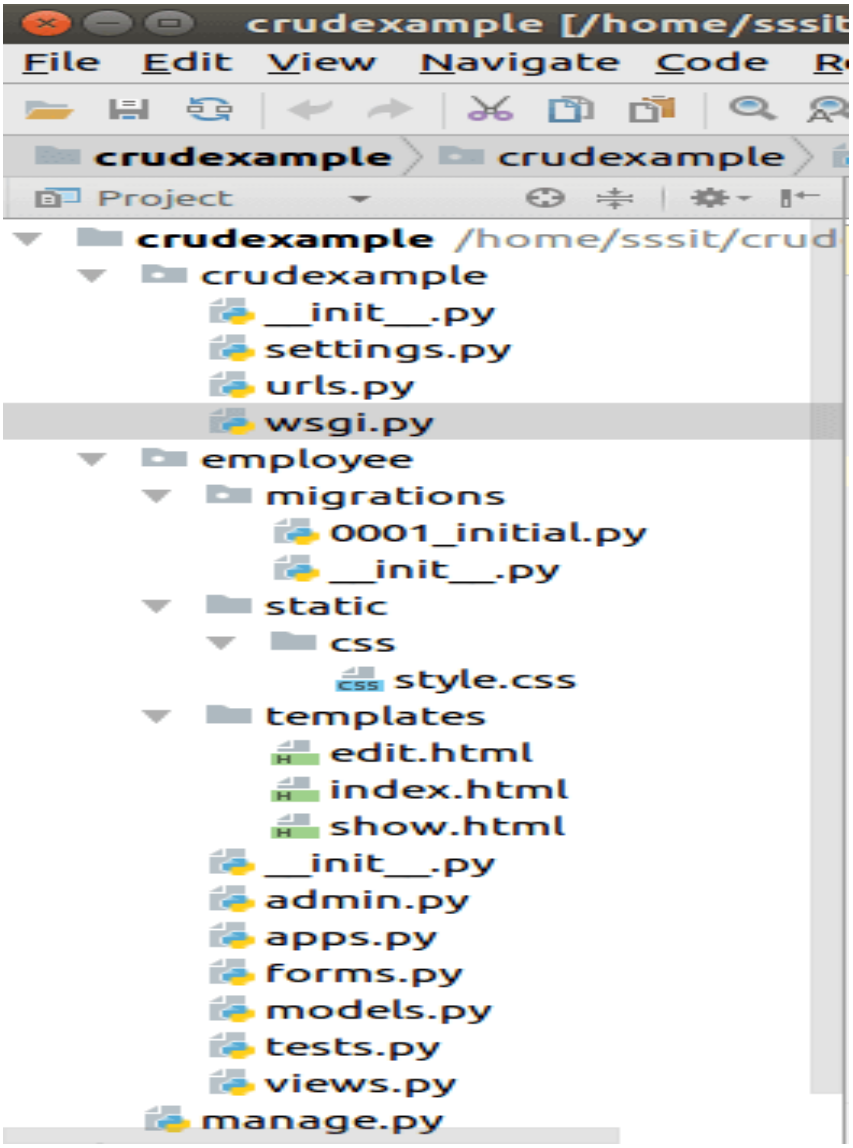
```
  <!DOCTYPE html>
  <html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Index</title>
    {% load staticfiles %}
    <link rel="stylesheet" href="{% static 'css/style.css' %}"/>
  </head>
  <body>
  <form method="POST" class="post-form" action="/update/{{employee.id}}">
      {% csrf_token %}
    <div class="container">
  <br>
    <div class="form-group row">
    <label class="col-sm-1 col-form-label"></label>
    <div class="col-sm-4">
    <h3>Update Details</h3>
    </div>
   </div>
    <div class="form-group row">
    <label class="col-sm-2 col-form-label">Employee Id:</label>
    <div class="col-sm-4">
      <input type="text" name="eid" id="id_eid" required maxlength="20" value
  ="{{ employee.eid }}"/>
```

```
    </div>
  </div>
  <div class="form-group row">
    <label class="col-sm-2 col-form-label">Employee Name:</label>
    <div class="col-sm-4">
      <input type="text" name="ename" id="id_ename" required maxlength="10
0" value="{{ employee.ename }}" />
    </div>
  </div>
    <div class="form-group row">
    <label class="col-sm-2 col-form-label">Employee Email:</label>
    <div class="col-sm-4">
      <input type="email" name="eemail" id="id_eemail" required maxlength="
254" value="{{ employee.eemail }}" />
    </div>
  </div>
    <div class="form-group row">
    <label class="col-sm-2 col-form-label">Employee Contact:</label>
    <div class="col-sm-4">
      <input type="text" name="econtact" id="id_econtact" required maxlength=
"15" value="{{ employee.econtact }}" />
    </div>
  </div>
    <div class="form-group row">
    <label class="col-sm-1 col-form-label"></label>
    <div class="col-sm-4">
    <button type="submit" class="btn btn-success">Update</button>
    </div>
  </div>
    </div>
</form>
</body>
</html>
```

Static Files Handling

Create a folder **static/css** inside the **employee** app and put a css inside it. Download the css file here Click Here.

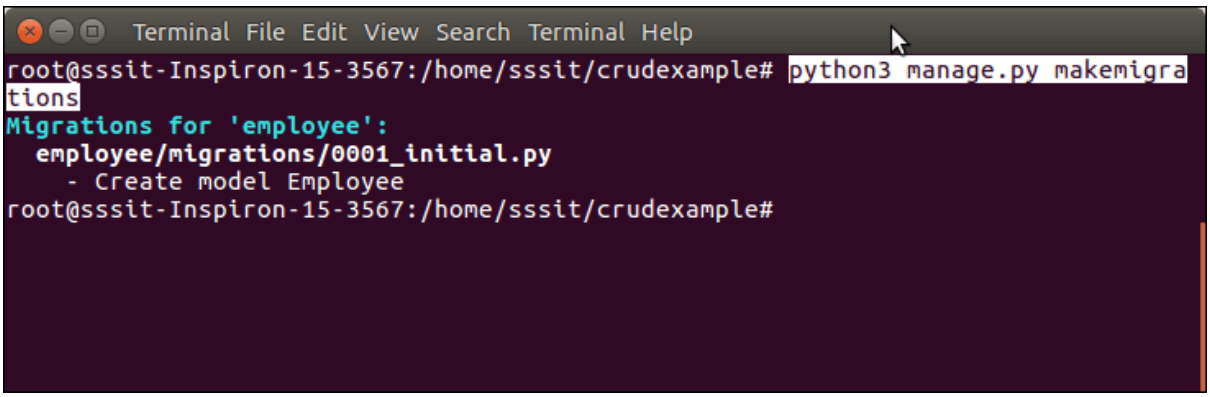
Project Structure



Create Migrations

Create migrations for the created model employee, use the following command.

1. $ python3 manage.py makemigrations



After migrations, execute one more command to reflect the migration into the database. But before it, mention name of app (employee) in INSTALLED_APPS of settings.py file.

// **settings.py**

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'employee'
]
```

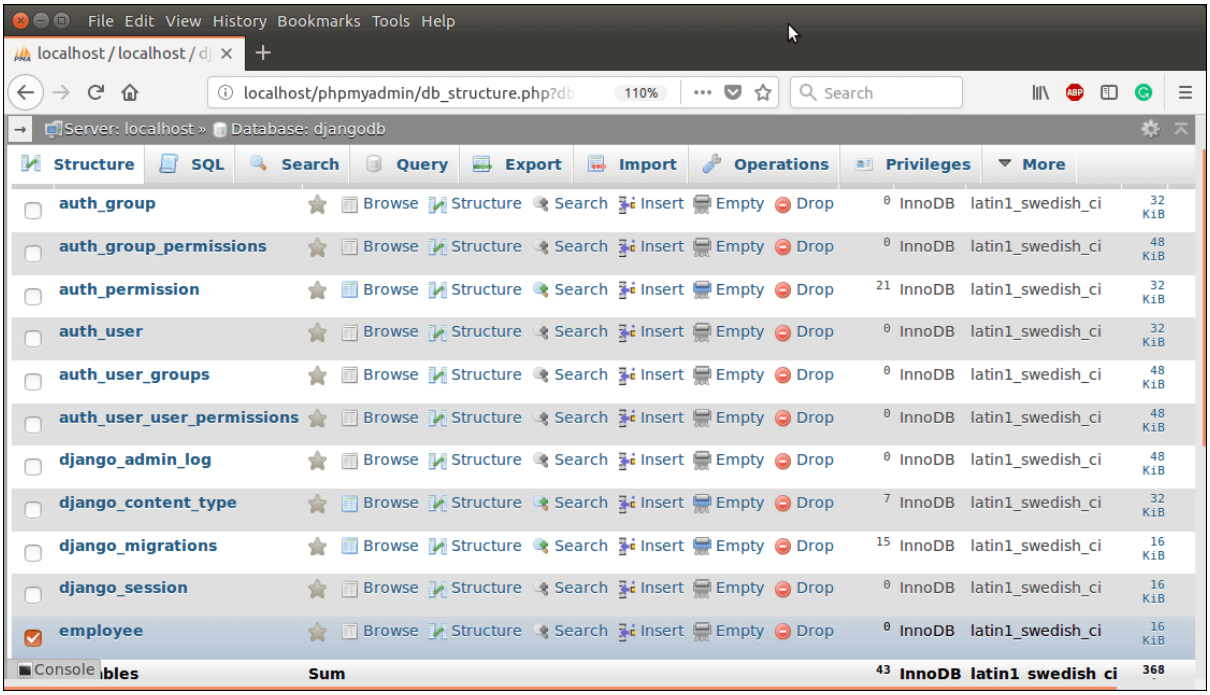
Run the command to migrate the migrations.

$ *python3* manage.py migrate

```
Terminal File Edit View Search Terminal Help
root@sssit-Inspiron-15-3567:/home/sssit/crudexample# python3 manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, employee, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying contenttypes.0002_remove_content_type_name... 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 employee.0001_initial... OK
  Applying sessions.0001_initial... OK
root@sssit-Inspiron-15-3567:/home/sssit/crudexample#
```

Now, our application has successfully connected and created tables in database. It creates 10 default tables for handling project (session, authentication etc) and one table of our model that we created.

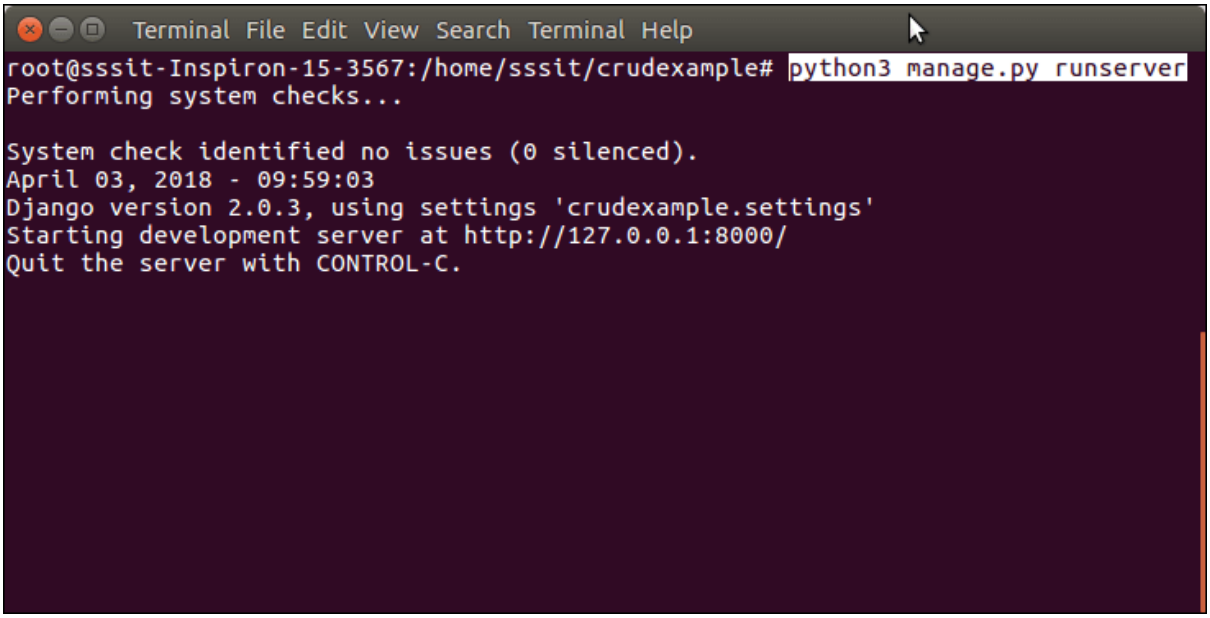
See list of tables created after migrate command.



Run Server

To run server use the following command.

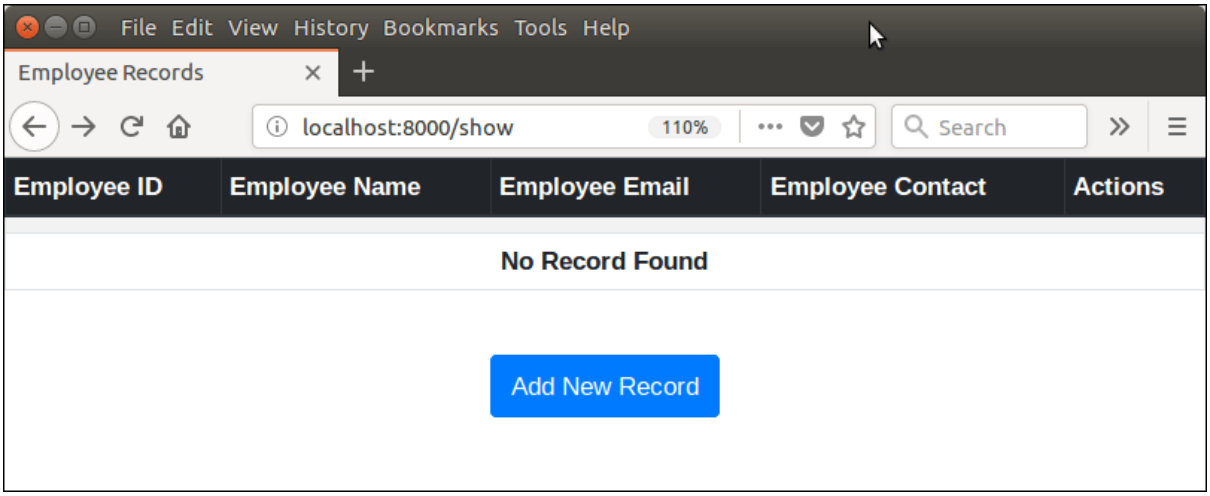
1. $ python3 manage.py runserver



Access to the Browser

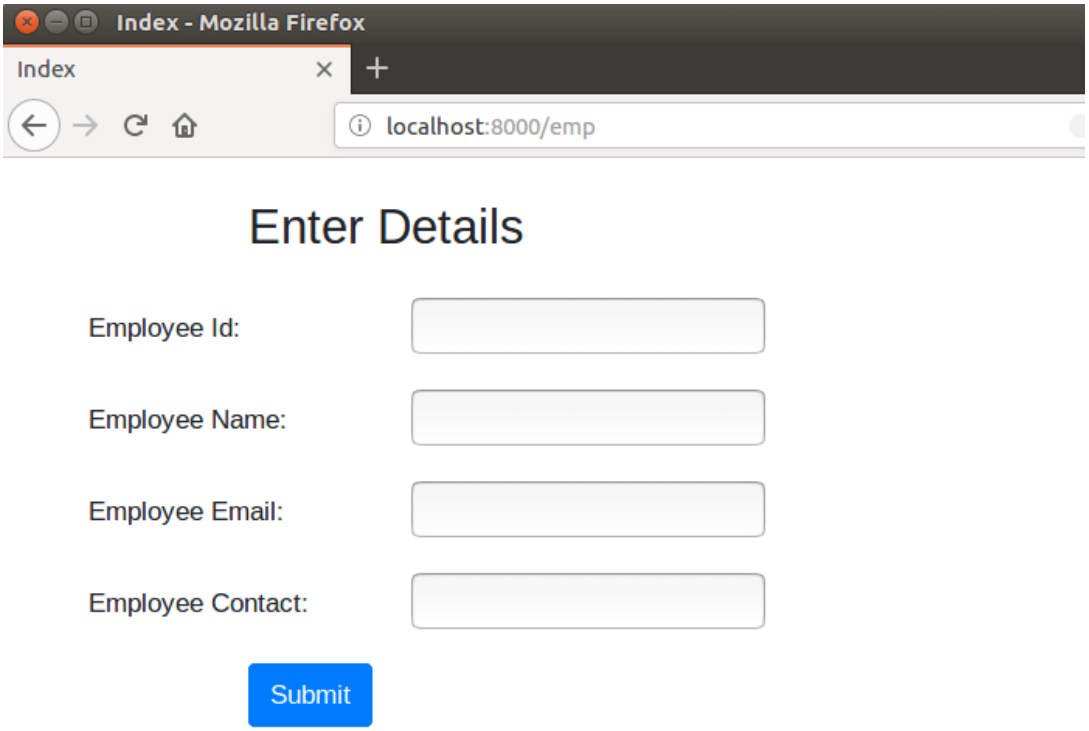
Access the application by entering **localhost:8000/show**, it will show all the available employee records.

Initially, there is no record. So, it shows no record message.

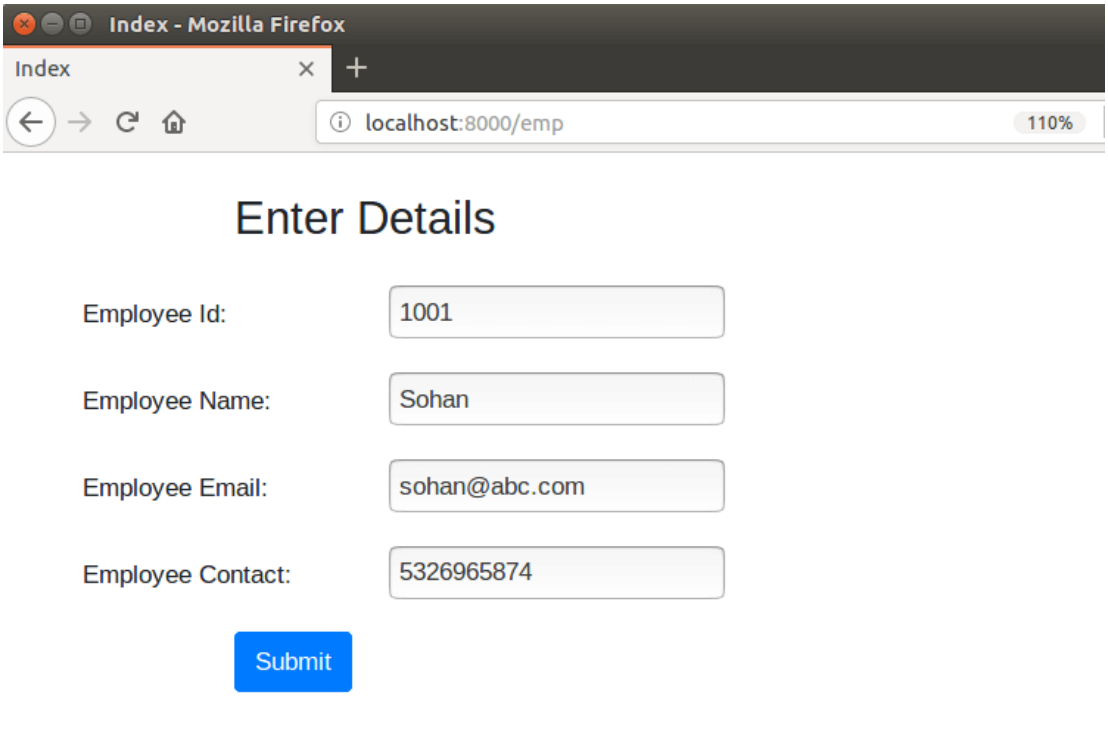


Adding Record

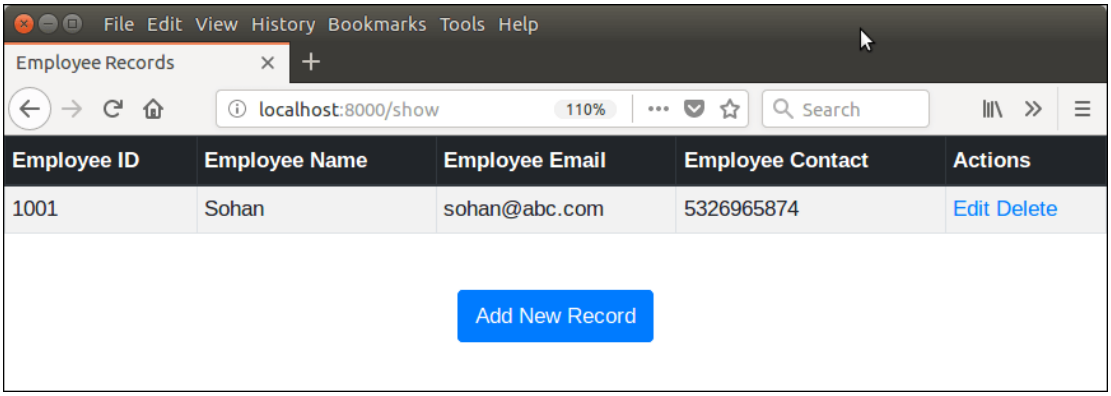
Click on the **Add New Record** button and fill the details. See the example.



Filling the details.

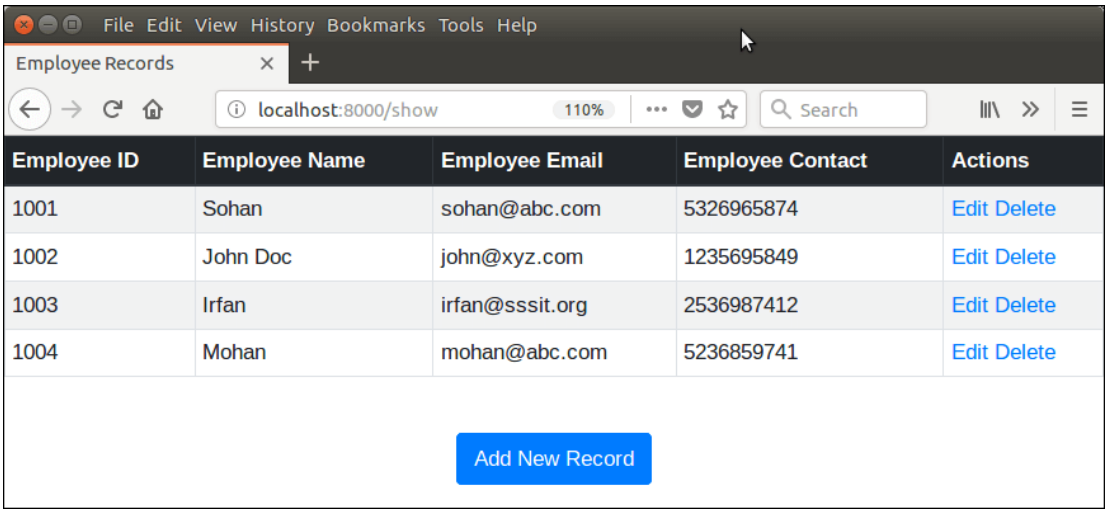


Submit the record and see, after submitting it shows the saved record.



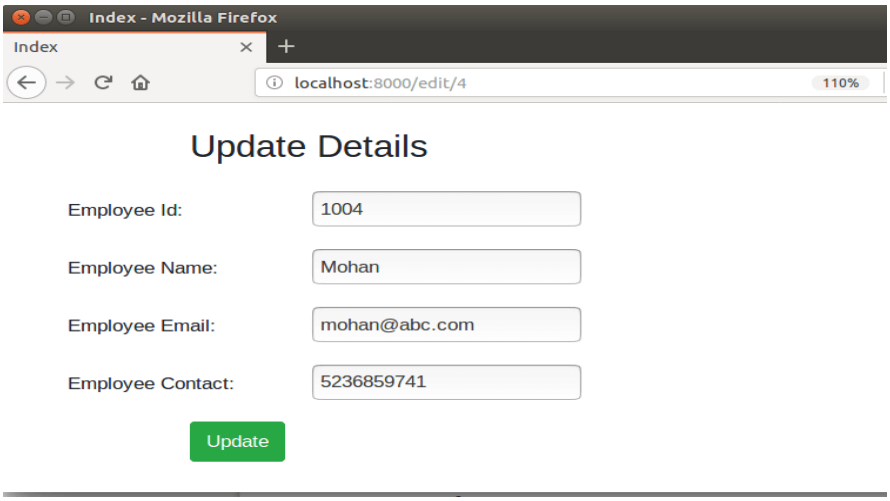
This section also allows, update and delete records from the **actions** column.

After saving couple of records, now we have following records.

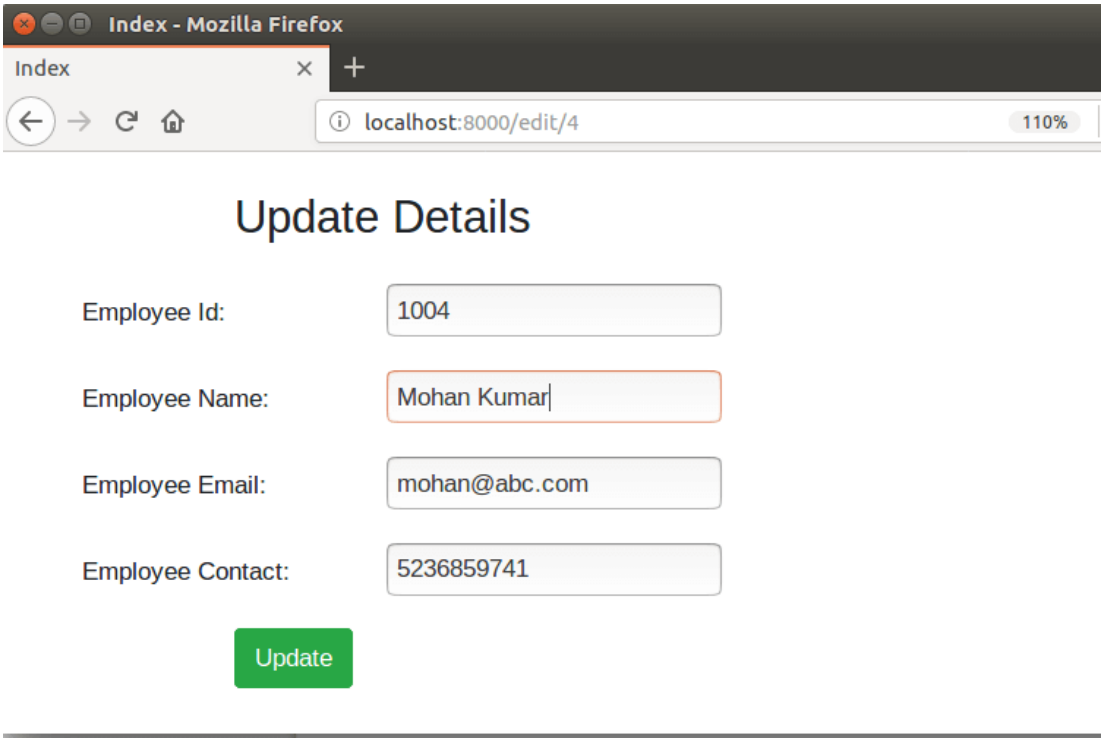


Update Record

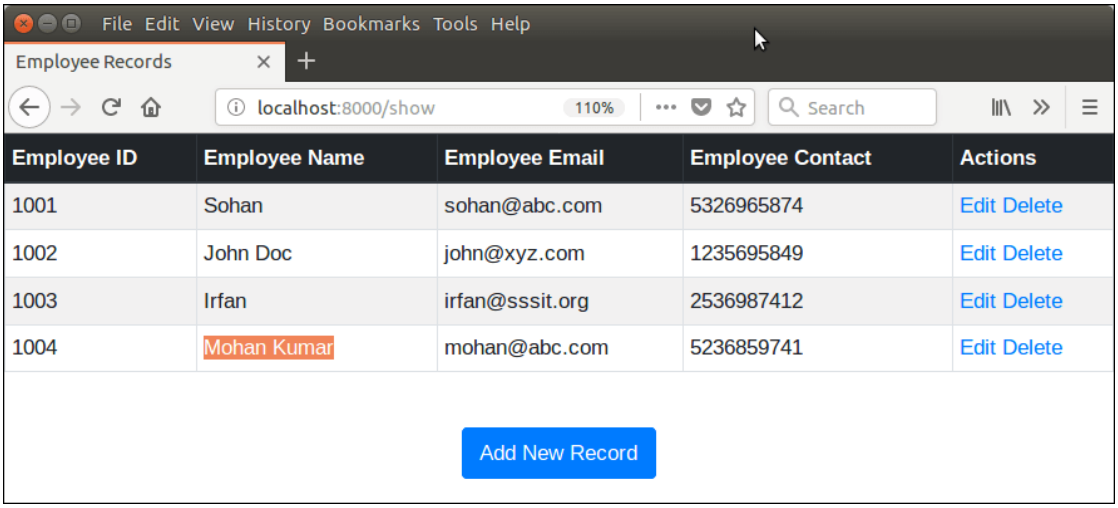
Lets update the record of **Mohan** by clicking on **edit** button. It will display record of Mohan in edit mode.



Lets, suppose I update **mohan** to **mohan kumar** then click on the update button. It updates the record immediately. See the example.



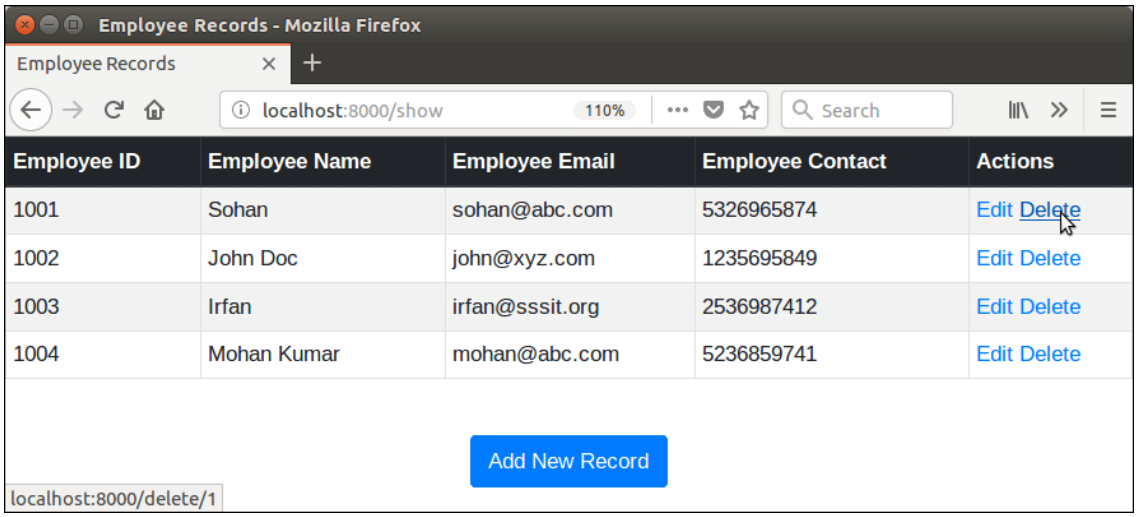
Click on update button and it redirects to the following page. See name is updated.



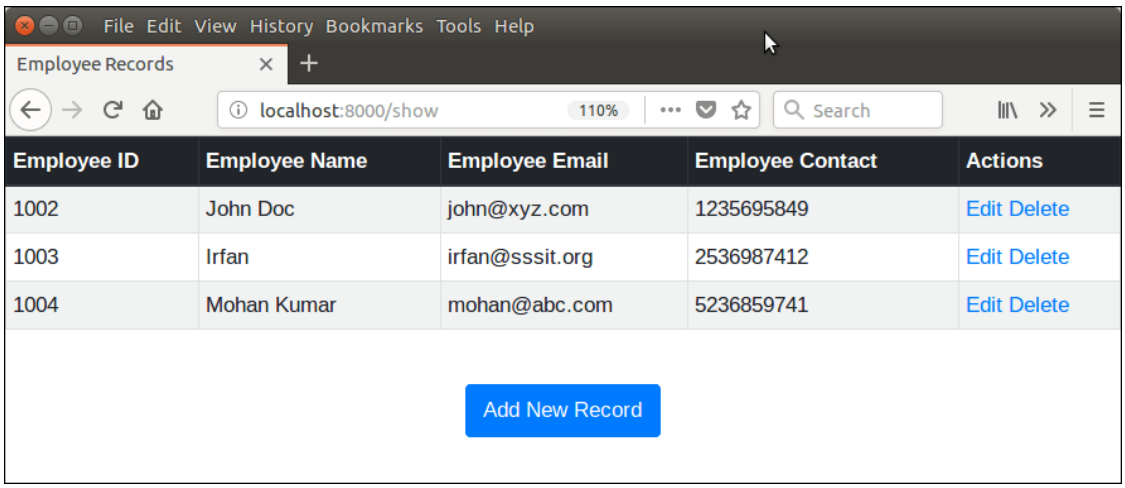
Same like, we can delete records too, by clicking the **delete** link.

Delete Record

Suppose, I want to delete **Sohan**, it can be done easily by clicking the delete button. See the example.



After deleting, we left with the following records.



Well, we have successfully created a CRUD application using Django.

2.5. Project installable

You can create a distributable package for your app using setuptools:

- Install setuptools (pip install setuptools).
- Create a setup.py file in the project root:

```
from setuptools import setup, find_packages

setup(
    name='your-app-name',
    version='0.1.0',
    packages=find_packages(),
    include_package_data=True,
    install_requires=[
        'django',
        # Any other dependencies
    ],
)
```

un python setup.py sdist bdist_wheel to create source and wheel distributions.

Install Your App in Another Project

- Navigate to the project where you want to use your CRUD app.
- Install the app from the distribution file (if you packaged it) or from the Git repository using pip:

Bash

```
pip install path/to/your-app-name-0.1.0.tar.gz  # For distribution file
pip install git+https://github.com/your-username/your-app-name.git  # From Git
```

Register Your App in the Main Project (myproject/settings.py)

- In myproject/settings.py, add your app to the INSTALLED_APPS list:

Python

```
INSTALLED_APPS = [
    # ... other apps
    'crudapp',
```

Self-Check Sheet Create CRUD project

1. Which of the following is NOT a part of the core directory structure of a Django project?

Answer:

- a) your_project_name/
- b) manage.py
- c) urls.txt
- d) your_app_name/

2. What command is used to create a new Django project?

Answer:

- a) django-createproject project_name
- b) django-initproject project_name
- c) django-admin startproject project_name
- d) django startproject project_name

3. In which file are database models defined in a Django application?

Answer:

- a) views.py
- b) models.py
- c) forms.py
- d) urls.py

4. What does the Django ORM (Object Relational Mapper) do?

Answer:

- a) Defines the URL patterns for a Django application.
- b) Provides a way to interact with a database using Python objects.
- c) Handles form validation in Django views.
- d) Manages the development server for a Django project.

5. What purpose does a template serve in a Django application?

Answer:

- a) Defines the logic for handling user requests.
- b) Holds the HTML structure and content displayed to the user.
- c) Specifies the database schema for a Django application.
- d) Defines the forms used for user input in a Django application.

Answer Key Create CRUD project

1. Which of the following is NOT a part of the core directory structure of a Django project?

Answer: c) urls.txt **Answer**

2. What command is used to create a new Django project?

Answer: c) django-admin startproject project_name

3. In which file are database models defined in a Django application?

Answer: b) models.py

4. What does the Django ORM (Object Relational Mapper) do?

Answer: b) Provides a way to interact with a database using Python objects.

5. What purpose does a template serve in a Django application?

Answer: b) Holds the HTML structure and content displayed to the user.

Job Sheet-2: **Create a CRUD Django Application**

UoC Cover

OU-ICT-WADP-02- L7-V1: Creating API Using Django REST Framework

Working Procedure / Steps

1. Project Setup:

1.1. **Create a Django Project:** Open your terminal and run the following command:

1.2. **Create a Django App:** Navigate to your project directory

2. Application Configuration:

2.1. **Register Your App:** In your project's settings.py file, ensure your app (crudapp) is included in the INSTALLED_APPS list

3. Model Definition:

Define your data model in crudapp/models.py. This model represents the data you want to manage (e.g., Order, Product, Employee).

4. Database Setup:

4.1. **Choose a Database:**

Decide on the database you want to use. Django supports various databases like MySQL, PostgreSQL, SQLite (default).

4.2. **Configure Database Settings:**

In your project's settings.py file, define the database connection details in the DATABASES dictionary

5. Forms: Forms are useful for creating user-friendly interfaces for adding and editing data. Define them in crudapp/forms.py.

6. Views: Views handle user requests and generate responses. Define them in crudapp/views.py.

Specification Sheet 2: Create a CRUD Django Application

Technical Requirements

- **Programming Language:** Python 3.7 or later
- **Framework:** Django 3.2 or later
- **Database:** Choose a compatible database backend (e.g., MySQL, PostgreSQL, SQLite)

Tools and Equipment

- **Computer:** A computer with sufficient processing power, RAM, and storage.
- **Terminal:** A command-line interface for executing commands.
- **Text Editor/IDE:** Visual Studio Code, PyCharm, Sublime Text (or any preferred code editor).

Reference

1. https://forum.djangoproject.com/
2. https://www.webforefront.com/django/

Review of Competency

Below is yourself assessment rating for module “Creating API Using Django REST Framework”

Assessment of performance Criteria	Yes	No
Serializers and views are applied	☐	☐
Class-based views are used	☐	☐
Mixins and generic class-based views are applied	☐	☐
Authentication, Token, Permission is implemented	☐	☐
Searching, filtering, and pagination are implemented	☐	☐
Project layout is created	☐	☐
Application setup	☐	☐
Database is set up	☐	☐
Blue print and views are applied	☐	☐
Project is made installable	☐	☐

I now feel ready to undertake my formal competency assessment.

Signed:

Date:

Development of CBLM

The Competency based Learning Material (CBLM) of 'Creating API Using Django REST Framework' (Occupation: Web Application Development with Python , Level-4) for National Skills Certificate is developed by NSDA with the assistance of SAMAHAR Consultants Ltd.in the month of June, 2024 under the contract number of package SD-9C dated 15th January 2024.

SL No.	Name and Address	Designation	Contact Number
1	Khan Mohammad Mahmud Hasan	Writer	Cell: 01714087897 Email: kmmhasan@gmail.com
2	A K M Mashuqur Rahman Mazumder	Editor	Cell: 01676323576 Email : mashuq.odelltech@odell.com.bd
3	Khan Mohammad Mahmud Hasan	Co-Ordinator	Cell: 01714087897 Email: kmmhasan@gmail.com
4	Md. Saif Uddin	Reviewer	Cell:01723004419 Email: engrbd.saif@gmail.com