API ENDPOINTS AND CODE EXPLAIN DOCUMENT

Name: Rahul Mahawar

Email: rahul.connects1@gmail.com

Project Stracture

```
maintain_proj/
   accounts/
       models.py
       views.py
       apps.py
       admin.py
       tests.py
       urls.py

  api/
     seralizer.py
     urls.py
     views.py
core/
       models.py
       views.py
       apps.py
       admin.py
       tests.py
       urls.py
maintain_proj
       settings.py
       wsgi.py
       urls.py
```

asgi.py
media
static
manage.py

API Endpoints

1. http://127.0.0.1:8000/api/timesheets/create
2. http://127.0.0.1:8000/api/timesheets/pk/update
3. http://127.0.0.1:8000/api/timesheets/lists/

STEP01:

Create new timesheet entries For Select Multiple Projects Existing

Open Postmen and make request with Post method

URL:http://127.0.0.1:8000/api/timesheets/create

Method:POST

auth:Basic Auth (Username ,password)

Body: raw/json

Demo-data: put projects pk for select project or you can also create a all new project for that you have to create dict inside this list with all attributes or fields like project_name, started_at etc e.g

```
{
   "projects":[
      1,
      2
   ],
   "user":2,
   "week_start_date":"2023-12-12",
   "hours_worked":"55.33"
}
```

STEP02:

Now we are Updating our Created Time Sheet.

For this in Postmen create new request with PUT Method

In this we have to know the timesheet pk so we can access that

URL:http://127.0.0.1:8000/api/timesheets/pk/update

Method:PUT

Auth: same

Body: raw/json

```
{
   "id": 15,
   "projects": [
      3
   ],
   "user": 3,
   "week_start_date": "2025-12-12",
   "hours_worked": "23.33"
}
```

```
{
   "projects":[
      1,
      3
   ],
   "user":2,
   "week_start_date":"2024-12-12",
   "hours_worked":"55.33"
}
```

You can copy response data and put that data to body section with raw/json and change anything like e.g we changing projects from 2,3 to 1,3 selection or you can also change hours_worked and other things

Response:

```
{
    "id": 15,
    "projects": [
        {
            "id": 3,
            "project_id": "54357f5daa1f",
            "project_name": "Project Three",
            "description": "Lorem Ipsume",
            "started_at": "2023-12-21",
            "end_at": "2023-12-30",
            "created_at": "2023-12-13T12:18:39.795124Z",
            "updated_at": "2023-12-13T12:18:39.795124Z"
        }
    ],
    "user": 3,
    "week_start_date": "2025-12-12",
    "hours_worked": "23.33"
}
```

STEP03:

Getting Lists of TimeSheets only related or specific user

We are using basic authentication from DRF

and because of our api/serializer.py ,api/views.py code only

authenticated user see their timesheets and change it not other

user,Only timesheet user or related to time sheet

Method:GET

URL:http://127.0.0.1:8000/api/timesheets/lists/

AUTH:same

Body:Nothing

Response: Getting results

```
[
    {
        "id": 14,
        "projects": [
            {
                "id": 2,
                "project_id": "070ad2c1bb6c",
                "project_name": "Project Two",
                "description": "s",
                "started_at": "2023-12-12",
                "end_at": "2023-12-15",
                "created_at": "2023-12-12T16:23:41.466256Z",
                "updated_at": "2023-12-12T16:23:41.466256Z"
            }
        ],
        "user": 3,
        "week_start_date": "2023-12-12",
        "hours_worked": "35.33"
    },
    {
        "id": 15,
        "projects": [
            {
                "id": 3,
                "project_id": "54357f5daa1f",
                "project_name": "Project Three",
                "description": "Lorem Ipsume",
                "started_at": "2023-12-21",
                "end_at": "2023-12-30",
                "created_at": "2023-12-13T12:18:39.795124Z",
                "updated_at": "2023-12-13T12:18:39.795124Z"
            }
        ],
        "user": 3,
        "week_start_date": "2025-12-12",
        "hours_worked": "23.33"
    }
]
```

CODE Explain:

api/serializer.py

```
from rest_framework import serializers
from core.models import Project , TimeSheet

class ProjectSerializer(serializers.ModelSerializer):
    class Meta:
        model = Project
        fields = "__all__"

class TimeSheetSerializer(serializers.ModelSerializer):
    projects = ProjectSerializer(many=True)
    class Meta:
        model = TimeSheet
        fields =
['id','projects','user','week_start_date','hours_worked']
```

This is a Python code that uses the Django REST framework to create serializers for the `Project` and `TimeSheet` models. Serializers are used to convert complex data such as querysets and model instances to native Python datatypes that can then be easily rendered into JSON, XML or other content types.

The `ProjectSerializer` class is used to serialize and deserialize data that corresponds to `Project` objects. It inherits from `serializers.ModelSerializer` and specifies the `Project` model as the `model` attribute in the `Meta` class. The `fields` attribute is set to `"__all__"`, which means that all fields in the `Project` model will be included in the serialized representation.

The `TimeSheetSerializer` class is used to serialize and deserialize data that corresponds to `TimeSheet` objects. It also inherits from `serializers.ModelSerializer` and specifies the `TimeSheet` model as the `model` attribute in the `Meta` class. The `fields` attribute is set to `['id','projects','user','week_start_date','hours_worked']`. The `projects` field is a nested serializer that uses the `ProjectSerializer` to serialize the related `Project` objects.

api/views.py

```python
from rest_framework import generics
from core.models import Project, TimeSheet
from .serializers import ProjectSerializer,TimeSheetSerializer
from django.contrib.auth.models import User
from rest_framework import mixins
#Retrieving a list of timesheet entries for a specific user
#Ensure that users can only view and edit their own timesheet entries.
from rest_framework.views import APIView
from rest_framework.response import Response


class TimeSheetCreateView(generics.ListAPIView,mixins.CreateModelMixin):
    queryset = TimeSheet.objects.all()
    serializer_class = TimeSheetSerializer

    def get_queryset(self):
        user = self.request.user
        if user.is_authenticated:
            return TimeSheet.objects.filter(user=user)
        else:
            return TimeSheet.objects.none()

    def post(self, request, *args, **kwargs):
        data = request.data
        projects= request.data.get("projects")

        user = request.user

        week_start_date = request.data.get("week_start_date")

        hours_worked = request.data.get("hours_worked")

        time_sheet = TimeSheet.objects.create(user=user,
week_start_date=week_start_date, hours_worked=hours_worked)


         # Link TimeSheet with Projects

        for project_id in projects:

            try:

                project = Project.objects.get(pk=project_id)

                time_sheet.projects.add(project)

            except Project.DoesNotExist:

                raise serializers.ValidationError({"projects":
f"Project with ID {project_id} not found."})

        # Serialize and return created TimeSheet

        serializer = TimeSheetSerializer(time_sheet)

        return Response(serializer.data)





class TimeSheetListView(generics.ListAPIView):

    queryset = TimeSheet.objects.all()

    serializer_class = TimeSheetSerializer
    def get_queryset(self):

        user = self.request.user

        if user.is_authenticated:

            return TimeSheet.objects.filter(user=user)

        else:

            return TimeSheet.objects.none()




class
TimeSheetDetailView(generics.RetrieveAPIView,mixins.UpdateModelMixin,):

    queryset = TimeSheet.objects.all()

    serializer_class = TimeSheetSerializer


    def get_queryset(self):

        user = self.request.user

        if user.is_authenticated:

            return TimeSheet.objects.filter(user=user)

        else:

            return TimeSheet.objects.none()


    def put(self, request, pk, *args, **kwargs):

        try:

            time_sheet = TimeSheet.objects.get(pk=pk)

        except TimeSheet.DoesNotExist:

            return Response({"error": "TimeSheet not found."},
status=status.HTTP_404_NOT_FOUND)
```

```
            # Update user, week_start_date, and hours_worked if
provided

        data = request.data

        if data.get("user"):

            # time_sheet.user = data["user"]

            time_sheet.user = request.user



        if data.get("week_start_date"):

            time_sheet.week_start_date = data["week_start_date"]

        if data.get("hours_worked"):

            time_sheet.hours_worked = data["hours_worked"]

            # Update projects

        projects = data.get("projects")

        if projects is not None:

             # Clear existing projects

            time_sheet.projects.clear()

            # Link provided projects

            for project_id in projects:

                try:

                    project = Project.objects.get(pk=project_id)

                    time_sheet.projects.add(project)

                except Project.DoesNotExist:

                    return Response({"projects": f"Project with ID
{project_id} not found."}, status=status.HTTP_400_BAD_REQUEST)

                    # Save the updated TimeSheet

        time_sheet.save()
        serializer = TimeSheetSerializer(time_sheet)

        return Response(serializer.data)
```

This is a code Django REST framework to create a view for the `TimeSheet` model. The `TimeSheetCreateView` class inherits from `generics.ListAPIView` and `mixins.CreateModelMixin`. It specifies the `TimeSheet` model as the `queryset` attribute and the `TimeSheetSerializer` as the `serializer_class` attribute.

The `get_queryset` method is used to return a queryset of `TimeSheet` objects filtered by the authenticated user. If the user is not authenticated, an empty queryset is returned.

The `post` method is used to create a new `TimeSheet` object. It first retrieves the data from the request and creates a new `TimeSheet` object with the specified `week_start_date` and `hours_worked`. It then links the `TimeSheet` object with the specified `Project` objects by iterating over the `projects` list and adding each `Project` object to the `TimeSheet` object's `projects` field. If a `Project` object with the specified ID is not found, a `ValidationError` is raised.

Finally, the `TimeSheet` object is serialized using the `TimeSheetSerializer` and returned as a JSON response.

This is a code that uses the Django REST framework to create a view for the `TimeSheet` model. The `TimeSheetListView` class inherits from `generics.ListAPIView`. It specifies the `TimeSheet` model as the `queryset` attribute and the `TimeSheetSerializer` as the `serializer_class` attribute.

The `get_queryset` method is used to return a queryset of `TimeSheet` objects filtered by the authenticated user. If the user is not authenticated, an empty queryset is returned.

This is a Python code that uses the Django REST framework to create a view for the `TimeSheet` model. The `TimeSheetDetailView` class inherits from `generics.RetrieveAPIView` and `mixins.UpdateModelMixin`. It specifies the `TimeSheet` model as the `queryset` attribute and the `TimeSheetSerializer` as the `serializer_class` attribute.

The `get_queryset` method is used to return a queryset of `TimeSheet` objects filtered by the authenticated user. If the user is not authenticated, an empty queryset is returned.

The `put` method is used to update an existing `TimeSheet` object. It first retrieves the `TimeSheet` object with the specified ID. If the object does not exist, a `404 Not Found` response is returned. It then retrieves the data from the request and updates the `user`, `week_start_date`, and `hours_worked` fields if provided. It also updates the related `Project` objects by iterating over the `projects` list and adding or removing each `Project` object to the `TimeSheet` object's `projects` field. If a `Project` object with the specified ID is not found, a `400 Bad Request` response is returned.

Finally, the updated `TimeSheet` object is saved and serialized using the `TimeSheetSerializer`. The serialized data is returned as a JSON response.

api/urls.py

```
path('timesheets/lists/',views.TimeSheetListView.as_view(),name='timesh
eet_list'),

path('timesheets/<pk>/update/',views.TimeSheetDetailView.as_view(),name
='timesheet_detail'),

path('timesheets/create',views.TimeSheetCreateView.as_view(),name='time
sheet_create_entry'),
```

The `path` function in Django is used to define URL patterns for views. It takes two required arguments: a string that defines the URL pattern, and the view function that should be called when the URL pattern is matched.

In the code you provided, there are three URL patterns defined:

1. `timesheets/lists/`: This URL pattern maps to the `TimeSheetListView` view, which is used to display a list of all `TimeSheet` objects.
2. `timesheets/<pk>/update/`: This URL pattern maps to the `TimeSheetDetailView` view, which is used to display a detailed view of a single `TimeSheet` object and update it.
3. `timesheets/create`: This URL pattern maps to the `TimeSheetCreateView` view, which is used to create a new `TimeSheet` object.

core/models.py

```
from django.contrib.auth.models import User
import uuid
```

```
class Project(models.Model):
    project_id = models.CharField(max_length=255,
default=uuid.uuid4().hex[:12], editable=False)
    project_name = models.CharField(max_length=50,blank=False)
    description =  models.TextField(blank=False,max_length=540)
    started_at = models.DateField(blank=True,null=True)
    end_at = models.DateField(blank=True,null=True)
    created_at = models.DateTimeField(blank=True,auto_now_add=True)
    #auto_now_add for automatic add firsttime when it object created so
it not change even how many time we
    updated_at = models.DateTimeField(blank=True,auto_now=True)
    #auto_now is for whenever this we chnage anything in object it will
update the date time and store it to updated_at

    def __str__(self):
        return f"{self.project_name},{self.project_id}"

class TimeSheet(models.Model):
    user = models.ForeignKey(User,on_delete=models.CASCADE)
    projects = models.ManyToManyField(Project,blank=True)
    week_start_date = models.DateField(blank=True,null=True)
    hours_worked = models.DecimalField(max_digits=5, decimal_places=2)

    def __str__(self):
        return f"{self.id}{self.user.username}"
```

This code is written in Python and uses the Django web framework. It defines two models: `Project` and `TimeSheet`.

The `Project` model has fields such as `project_id`, `project_name`, `description`, `started_at`, `end_at`, `created_at`, and `updated_at`. The `TimeSheet` model has fields such as `user`, `projects`, `week_start_date`, and `hours_worked`.

The `project_id` field is assigned a unique value using the `uuid4()` method from the `uuid` module. The `created_at` field is automatically set to the current date and time when the object is created, and the `updated_at` field is automatically set to the current date and time whenever the object is updated.

The `TimeSheet` model has a `ForeignKey` to the `User` model and a `ManyToManyField` to the `Project` model. It also has fields such as `week_start_date` and `hours_worked`.

These models can be used to store information about projects and timesheets in a database. For example, you could create a new `Project` object with a name and description, and then create a new `TimeSheet` object for a user that is associated with that project.

I hope this helps! Let me know if you have any further questions.

The code imports the `User` model from `django.contrib.auth.models` and the `uuid` module. It then defines two models: `Project` and `TimeSheet`.

The `Project` model has the following fields:

- `project_id`: A `CharField` with a maximum length of 255 characters. It is assigned a unique value using the `uuid4()` method from the `uuid` module. The `editable` attribute is set to

`False`, which means that the field cannot be edited once it has been created.

- `project_name`: A `CharField` with a maximum length of 50 characters. It is a required field (`blank=False`).
- `description`: A `TextField` with a maximum length of 540 characters. It is a required field (`blank=False`).
- `started_at`: A `DateField` that is not required (`blank=True`) and can be `null`.
- `end_at`: A `DateField` that is not required (`blank=True`) and can be `null`.
- `created_at`: A `DateTimeField` that is not required (`blank=True`) and automatically set to the current date and time when the object is created (`auto_now_add=True`).
- `updated_at`: A `DateTimeField` that is not required (`blank=True`) and automatically set to the current date and time whenever the object is updated (`auto_now=True`).

The `TimeSheet` model has the following fields:

- `user`: A `ForeignKey` to the `User` model. It is a required field (`on_delete=models.CASCADE`).
- `projects`: A `ManyToManyField` to the `Project` model. It is not a required field (`blank=True`).
- `week_start_date`: A `DateField` that is not required (`blank=True`) and can be `null`.

- `hours_worked`: A `DecimalField` with a maximum of 5 digits and 2 decimal places.

Both models define a `__str__` method that returns a string representation of the object.

accounts/forms.py

```
from django import forms

from django.contrib.auth.models import User


class UserRegisterForm(forms.ModelForm):

    password =
forms.CharField(label='Password',widget=forms.PasswordInput(attrs={'cla
ss':'form-control  ','id':'pass'}))

    password2 = forms.CharField(label='Repeat Password',
widget=forms.PasswordInput(attrs={'class':'form-control   '}))


    class Meta:

        model = User

        fields = ('username', 'first_name','last_name', 'email',)

        widgets = {

                'username':forms.TextInput(attrs={'class':'form-control
'}),


'first_name':forms.TextInput(attrs={'class':'form-control   '}),
'last_name':forms.TextInput(attrs={'class':'form-control   '}),
                'email':forms.TextInput(attrs={'class':'form-control
'}),}


    def clean_password2(self):
        cd = self.cleaned_data
        if cd['password'] != cd['password2']:
            raise forms.ValidationError('Passwords don\'t match.')
        return cd['password2']



class LoginForm(forms.Form):
    username =
forms.CharField(label='Username',widget=forms.TextInput(attrs={'class':
'form-control'}))
    password =
forms.CharField(widget=forms.PasswordInput(attrs={'class':'form-control
','id':'pass'}))


class UserEditForm(forms.ModelForm):
        class Meta:
            model = User
            fields = ('first_name', 'last_name','email')
            widgets ={

'first_name':forms.TextInput(attrs={'class':'form-control  '}),
'last_name':forms.TextInput(attrs={'class':'form-control  '}),

                 'email':forms.TextInput(attrs={'class':'form-control
'}),



             }
```

It defines three classes: `UserRegisterForm`, `LoginForm`, and `UserEditForm`.

The `UserRegisterForm` class has fields such as `password` and `password2`, which are used to create a new user account. The `password` field is a required field that accepts a password input from the user. The `password2` field is also a required field and is used to confirm the password entered in the `password` field. If the two passwords do not match, a validation error is raised.

The `LoginForm` class has fields such as `username` and `password`, which are used to authenticate a user. The `username` field is a required field that accepts a username input from the user. The `password` field is also a required field and is used to authenticate the user.

The `UserEditForm` class has fields such as `first_name`, `last_name`, and `email`, which are used to edit a user's profile. These fields are not required and can be left blank.

All three classes use the `forms` module from Django to create form fields and widgets. The `Meta` class is used to specify the model and fields for the form.

accounts/views.py

```
# Create your views here.

from django.shortcuts import render,redirect,HttpResponse,
get_object_or_404

# Create your views here.

from django.http import request

from django.http import HttpResponse

from django.contrib.auth import authenticate, login, logout

from .forms import UserRegisterForm,LoginForm,UserEditForm

from django.contrib import auth, messages

from django.contrib.auth.decorators import login_required

from django.contrib.auth.models import Group, Permission, User


def register(request):

    if request.user.is_authenticated:
        messages.success(request,"You already have logged-in in your
current account.")

        return redirect('')

    elif request.method == 'POST':

        user_form = UserRegisterForm(request.POST)

        if user_form.is_valid ():

            new_user = user_form.save(commit=False)

            new_user.set_password(user_form.cleaned_data['password'])

            new_user.save()

            #Automate Logged in new Signup user

            login(request, new_user)

            #return redirect("accounts:signup_done")

            return redirect('core:home')

    else:

        user_form = UserRegisterForm()

    return render(request,'accounts/register.html',{'user_form':
user_form})











def user_login(request):

    if request.user.is_authenticated:

        msg = messages.success(request,"You Already have account.")
        return redirect('')
    elif request.method == 'POST':
        form = LoginForm(request.POST)
        if form.is_valid():
            cd = form.cleaned_data
            user = authenticate(request,
                                        username=cd['username'],
                                        password=cd['password'])
            if user is not None:
                if user.is_active:
                    login(request, user)
                    return redirect('core:home')
                else:
                    return HttpResponse('Disabled account')
            else:
                return HttpResponse('Invalid login')
    else:
        form = LoginForm()
    return render(request, 'accounts/login.html', {'form': form})


def user_logout(request):
    logout(request)
    return redirect("accounts:login")
```

```python
@login_required
def edit(request):
    if request.method == 'POST':
        user_form = UserEditForm(instance=request.user,
                                 data=request.POST)
        if user_form.is_valid():
            user_form.save()
            messages.success(request, 'User Details updated '\
                                      'successfully')
            #return redirect('accounts:dashboard')
            return redirect('core:home')
        else:
            messages.error(request, 'Error updating your profile')
    else:
        user_form = UserEditForm(instance=request.user)
    return render(request,
                  'accounts/edit.html',
                  {'user_form': user_form,})
```

```
@login_required
def dashboard(request):
    return render(request,
                  'accounts/dashboard.html')
```

It defines two functions: `register` and `user_login`.

The `register` function is used to register a new user. If the user is already logged in, a message is displayed indicating that they are already logged in. If the request method is `POST`, the form data is validated using the `UserRegisterForm` class. If the form is valid, a new user is created with the specified username, password, first name, last name, and email. The user is then logged in and redirected to the home page. If the request method is not `POST`, a new `UserRegisterForm` instance is created and rendered in the `register.html` template.

The `user_login` function is used to authenticate a user. If the user is already logged in, a message is displayed indicating that they are already logged in. If the request method is `POST`, the form data is validated using the `LoginForm` class. If the form is valid, the user is authenticated using the specified username and password. If the user is authenticated and their account is active, they are logged in and redirected to the home page. If the user is not authenticated or their account is disabled, an error message is displayed.

These functions use various modules and classes from Django, such as `render`, `redirect`, `HttpResponse`, `get_object_or_404`, `request`, `authenticate`, `login`, `logout`, `UserRegisterForm`, `LoginForm`, `UserEditForm`, `auth`, `messages`, `login_required`, `Group`, `Permission`, and `User`.

accounts/urls.py

```
from django.urls import path, include
from django.contrib.auth import views as auth_views
from . import views
from django.contrib import admin
from django.views.generic import TemplateView
from django.contrib.auth import views as auth_views

app_name='accounts'
urlpatterns = [ #reg
    path('register/', views.register, name='register'),
    path("register_done/",
    TemplateView.as_view(template_name="account/register_done.html"),
    name="signup_done"),
    path("login/", views.user_login, name="login"),
    path('logout/', views.user_logout, name='logout'),
    path("dashboard/",views.dashboard,name="dashboard"),
    path("edit-user/",views.edit,name="edit-user"),


]
```

This code is written in Python and uses the Django web framework. It defines a list of URL patterns for the `accounts` app.

The `path` function is used to define URL patterns for various views, such as `register`, `user_login`, `user_logout`, `dashboard`, and `edit`. The `include` function is used to include URL patterns from other apps, such as the `django.contrib.auth` app.

The `views` module is imported and used to define the views for the `register`, `user_login`, `user_logout`, `dashboard`, and `edit` URL patterns.

The `TemplateView` class is imported from `django.views.generic` and used to define the view for the `register_done` URL pattern.

The `admin` module is imported from `django.contrib` but not used in this code.

The `auth_views` module is imported from `django.contrib.auth` and used to define the views for the `login` and `logout` URL patterns.

The app_name variable is defined to specify the namespace for the app.

project/urls.py

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

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

    path('accounts/', include("accounts.urls",namespace='accounts')),

    #path('api-auth/', include('rest_framework.urls')),

    path('api/', include('api.urls', namespace='api')),
```

- The first URL pattern maps to the Django admin site, which is used to manage the project's data.
- The second URL pattern maps to the core app's URLs. The core app is a part of the project and contains the main functionality of the project.
- The third URL pattern maps to the accounts app's URLs. The accounts app is a part of the project and contains the user authentication functionality.
- The fourth URL pattern maps to the api app's URLs. The api app is a part of the project and contains the REST API functionality.

project/settings.py

Set global Authentication Permissions:

```
REST_FRAMEWORK = {

    # Use Django's standard `django.contrib.auth` permissions,

    # or allow read-only access for unauthenticated users.

    'DEFAULT_PERMISSION_CLASSES': [


        'rest_framework.permissions.IsAuthenticated',

    ]

}
```

This is a code that configures the default permission classes for the Django REST framework. The `DEFAULT_PERMISSION_CLASSES` setting is used to specify a list of permission classes that should be applied to all views in the project by default.

In this case, the `IsAuthenticated` permission class is used to restrict access to authenticated users only. This means that unauthenticated users will not be able to access any views that use this permission class.

If you want to allow read-only access for unauthenticated users, you can use the `IsAuthenticatedOrReadOnly` permission class instead.