Part 1: Create a Simple Employee API

Step 1: Setup Django Project

1.1 Create a Virtual Environment

A virtual environment helps isolate dependencies, ensuring your project doesn't conflict with system-wide packages.

Run the following commands:

- `python -m venv env`: Creates a virtual environment named `env`.
- `source env/bin/activate`: Activates the virtual environment (use `env\Scripts\activate` for Windows).

1.2 Install Django and Django REST Framework

bash
CopyEdit

```
pip install django djangorestframework
```

- `django`: The main framework for building web applications.
- `djangorestframework`: Provides tools for building RESTful APIs in Django.

1.3 Create a Django Project

bash
CopyEdit

```
django-admin startproject companyapi .
```

- `django-admin startproject companyapi .`: Creates a Django project named `companyapi`. The `.` ensures it's created in the current directory.

1.4 Start the Development Server

```
python manage.py runserver
```

- This starts Django's built-in server, allowing you to access your application at `http://127.0.0.1:8000/`.

Step 2: Create the Employee API App

Django projects can have multiple apps. Each app is a module with specific functionalities.

2.1 Create the App

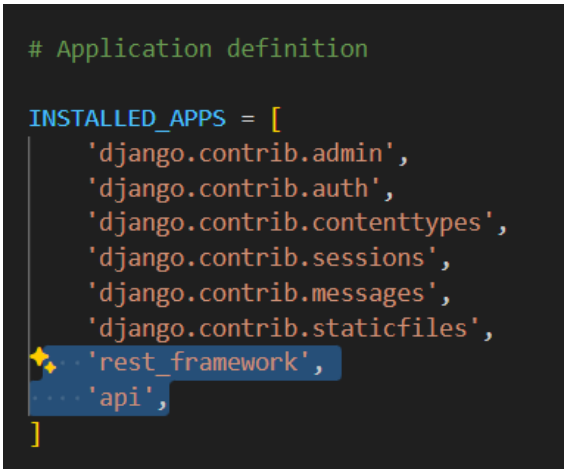
bash
CopyEdit

```
python manage.py startapp employee_api
```

- `startapp employee_api`: Creates an app named `employee_api`.

2.2 Register the App in `settings.py`

Open `companyapi/settings.py` and add `'employee_api'` to `INSTALLED_APPS`:



- This step ensures Django recognizes the new app.

Step 3: Define the Employee Model

A **model** represents a table in the database.

3.1 Open `employee_api/models.py` and add:

```python
# employee_api/models.py
from django.db import models

class Employee(models.Model):
    name = models.CharField(max_length=255)

    age = models.IntegerField()
    department = models.CharField(max_length=255)
    hire_date = models.DateField()
    salary = models.DecimalField(max_digits=10, decimal_places=2,default=0)  # New salary field

    def __str__(self):
        return self.name
```

- `models.Model`: Every model in Django must inherit from this base class.
- `CharField(max_length=255)`: Stores short text values.
- `IntegerField()`: Stores numbers.
- `DateField()`: Stores date values.
- `__str__()`: Returns the employee's name when printed.

3.2 Run Migrations

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

- `makemigrations`: Creates migration files (changes to the database structure).
- `migrate`: Applies migrations to create/update tables in the database.

Step 4: Create a Serializer

Serializers convert Python objects (models) into JSON and vice versa.

4.1 Open `employee_api/serializers.py` and add:

```python
# employee_api/serializers.py
import logging
from rest_framework import serializers
from .models import Employee


class EmployeeSerializer(serializers.ModelSerializer):
    class Meta:
        model = Employee
        fields = '__all__'
```

- `ModelSerializer`: Simplifies the creation of serializers for models.
- `fields = '__all__'`: Includes all fields from the model.

Step 5: Create API Views

5.1 Open `employee_api/views.py` and add:

```python
# employee_api/views.py
from rest_framework import generics
from .models import Employee
from .serializers import EmployeeSerializer

class EmployeeListCreate(generics.ListCreateAPIView):
    queryset = Employee.objects.all()
    serializer_class = EmployeeSerializer

class EmployeeDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Employee.objects.all()
    serializer_class = EmployeeSerializer
```

- `ListCreateAPIView`: Handles `GET` (list all employees) and `POST` (create employee).
- `RetrieveUpdateDestroyAPIView`: Handles `GET` (retrieve one employee), `PUT` (update), and `DELETE`.

Step 6: Define API Routes

6.1 Open `employee_api/urls.py` and add:

```python
from django.urls import path
from .views import EmployeeListCreate, EmployeeDetail

urlpatterns = [
    path('employees/', EmployeeListCreate.as_view(), name='employee-list-create'),
    path('employees/<int:pk>/', EmployeeDetail.as_view(), name='employee-detail'),
]
```

- Defines routes for listing, creating, retrieving, updating, and deleting employees.

6.2 Include API URLs in the Main `companyapi/urls.py`

```python
from django.contrib import admin
from django.urls import path,include
from django.conf import settings
from django.conf.urls.static import static

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

Step 7: Run the Server and Test API

```
python manage.py runserver
```

Part 2: Add Logging

Logging helps track API activity.

Step 8: Configure Logging in `settings.py and serializer`

In sereializer.py

```python
# employee_api/serializers.py
import logging
from rest_framework import serializers
from .models import Employee

logger = logging.getLogger('employee_api')

class EmployeeSerializer(serializers.ModelSerializer):
    class Meta:
        model = Employee
        fields = '__all__'

    def validate_salary(self, value):
        # Validate that the salary is a positive number
        if value <= 0:
            logger.error(f"Invalid salary value: {value}. Salary must be positive.")
            raise serializers.ValidationError("Salary must be a positive number.")
        return value
```

```python
LOGGING_DIR = os.path.join(BASE_DIR, 'logs')
# Create logs directory if it doesn't exist
if not os.path.exists(LOGGING_DIR):
    os.makedirs(LOGGING_DIR)

LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'file': {
```

```
            'level': 'INFO',
            'class': 'logging.FileHandler',
            'filename': os.path.join(LOGGING_DIR, 'employee_api.log'),
        },
        'csv_file': {
            'level': 'INFO',
            'class': 'logging.FileHandler',
            'filename': os.path.join(LOGGING_DIR, 'employee_api.csv'),
            'formatter': 'csv_formatter',
        },
    },
    'formatters': {
        'csv_formatter': {
            'format': '{asctime},{levelname},{message}',
            'style': '{',
        },
    },
    'loggers': {
        'django': {
            'handlers': ['file', 'csv_file'],
            'level': 'INFO',
            'propagate': True,
        },
    },
}
```

- Logs API activities in `logs/employee_api.log`.

Step 9: Add Logging to Views

Modify `employee_api/views.py`:

```
# employee_api/views.py
import logging
```

```python
from rest_framework import generics
from .models import Employee
from .serializers import EmployeeSerializer
from rest_framework.response import Response
from rest_framework import status

logger = logging.getLogger('employee_api')

class EmployeeListCreate(generics.ListCreateAPIView):
    queryset = Employee.objects.all()
    serializer_class = EmployeeSerializer

    def perform_create(self, serializer):
        logger.info(f"Creating new employee:
{serializer.validated_data}")
        try:
            serializer.save()
            logger.info("Employee created successfully.")
        except Exception as e:
            logger.error(f"Error creating employee: {e}")
            raise e

    def get_queryset(self):
        logger.debug("Fetching list of employees.")
        return Employee.objects.all()

class EmployeeDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Employee.objects.all()
    serializer_class = EmployeeSerializer

    def get_object(self):
        logger.debug("Fetching employee details.")
        try:
            return super().get_object()
        except Exception as e:
            logger.error(f"Error fetching employee: {e}")
            raise e
```

- Logs employee creation.

Create a file logging_handlers.py which has a class CSVLoggingHandler

```
# logging_handlers.py

import csv

import logging



class CSVLoggingHandler(logging.Handler):

    def __init__(self, filename):

        super().__init__()

        self.filename = filename



        # Ensure the file exists and is ready for appending

        with open(self.filename, 'a', newline='') as file:

            writer = csv.writer(file)

            writer.writerow(['Time', 'Level', 'Message'])  # Write
header only once



    def emit(self, record):

        try:

            log_entry = self.format(record)

            # Split log entry into time, level, and message for CSV

            log_parts = log_entry.split(',')

            with open(self.filename, 'a', newline='') as file:

                writer = csv.writer(file)

                writer.writerow(log_parts)

        except Exception:
```

```
            self.handleError(record)
```

Import it to settings.py

```
import os
from .logging_handlers import CSVLoggingHandler
```
