Modular Architecture Implementation for React & Django

Objective:

By the end of this session, students will be able to **structure Django apps and Node.js modules** for **backend reusability** and **organize React components** for frontend scalability using the **React UserList example**.

1. Understanding Modular Architecture

Definition:

- Modular architecture is a **design approach that structures code into independent modules** that handle specific responsibilities.
- It improves **scalability, maintainability, and reusability**.

Key Principles:

- **Backend:** Organize Django apps or Node.js modules.
- **Frontend:** Structure React components for reusability.

2. Backend: Organizing Django Apps for Modular Design

1️⃣ Install Django (if not installed):

```
pip install django djangorestframework
```

2️⃣ Create a Django project:

```
django-admin startproject backend
cd backend
```

3️⃣ Create Django apps:

```
python manage.py startapp users
python manage.py startapp tasks
```

users/ → Handles user authentication and management.

tasks/ → Manages task-related data (for CRUD operations).

Step 2: Configure Django Apps in backend/settings.py

Add the new apps to the INSTALLED_APPS list:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'rest_framework',
    'users',
    'tasks',
]
```

Step 3: Define Models (Database Structure)

users/models.py:

```
from django.contrib.auth.models import AbstractUser
from django.db import models

class CustomUser(AbstractUser):
    email = models.EmailField(unique=True)
```

tasks/models.py:

```
from django.db import models
from users.models import CustomUser

class Task(models.Model):
    title = models.CharField(max_length=255)
    description = models.TextField()
    assigned_to = models.ForeignKey(CustomUser, on_delete=model
```

Run migrations:

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

Step 4: Create API Endpoints with Django REST Framework

users/views.py:

```
from rest_framework import generics
from .models import CustomUser
from .serializers import UserSerializer

class UserListCreate(generics.ListCreateAPIView):
    queryset = CustomUser.objects.all()
    serializer_class = UserSerializer
```

tasks/views.py:

```
from rest_framework import generics
from .models import Task
from .serializers import TaskSerializer

class TaskListCreate(generics.ListCreateAPIView):
    queryset = Task.objects.all()
    serializer_class = TaskSerializer
```

urls.py (backend):

```
from django.urls import path
from users.views import UserListCreate
from tasks.views import TaskListCreate

urlpatterns = [
    path('users/', UserListCreate.as_view(), name='users-list'),
    path('tasks/', TaskListCreate.as_view(), name='tasks-list'),
]
```

Step 5: Testing API with Postman

Start the server:

python manage.py runserver

Test endpoints:

- GET http://127.0.0.1:8000/users/ → List users
- POST http://127.0.0.1:8000/users/ → Create new user
- GET http://127.0.0.1:8000/tasks/ → List tasks
- POST http://127.0.0.1:8000/tasks/ → Create new task

3. Frontend: Structuring React Components for Reusability

Step 1: Project Structure for Modular Design

```
frontend/
│ ─── src/
│     ├── components/
│     │     ├── UserList.js
│     │     ├── TaskList.js
│     ├── hooks/
│     │     ├── useUsers.js
│     │     ├── useTasks.js
│     ├── services/
│     │     ├── userService.js
│     │     ├── taskService.js
│     ├── pages/
│     │     ├── UsersPage.js
│     │     ├── TasksPage.js
│     ├── App.js
```

Assignment: API and UI Setup:

Each student must set up the API and UI for this project following the steps provided.

Each successful step should have a commit message describing the work done.

Let each of you pick out 5 tasks of the following and each student should have atleast 3 tasks that are unique to them:

1. Task A: Implement only the User API (list, create users).
2. Task B: Implement only the Task API (list, create tasks, assign users).
3. Task C: Implement both APIs but only GET requests.
4. Task D: Implement POST requests and authentication only.
5. Task E: Add pagination and filtering to tasks API.
6. Task F: Implement UI for listing users, but UI components must be styled uniquely.
7. Task G: Implement UI for managing tasks, using a different state management approach.
8. Task H: Add unit tests for either users or tasks API.

Example of commit messages:

- `feat: create Django project and user app`
- `feat: add user model and migrations`
- `feat: implement User API with DRF`
- `fix: resolve API response formatting issue`
- `feat: create basic React structure for User List`

Step 6: Run Frontend

npm start to make sure everything runs properly.

Step 7: **Submission**

- Push commits to **individual repositories**.
- Submit repository links along with a brief documentation of challenges faced.
- **Each student must explain their implementation in a short write-up (1-2 paragraphs).**

Conclusion

✅ **Backend follows Django modular apps.**
✅ **Frontend uses reusable components & hooks.**
✅ **Code is scalable & maintainable.**

This ensures a **well-structured, scalable** full-stack app!