🚀 FastAPI Project Refactoring with Routers

> 🌿 Modular Structure · 📁 Folder Setup · 📦 Multiple Routers · 🔁 Clean Architecture

---

🌟 Why Refactor with Routers?

FastAPI lets you split large applications into **independent, reusable router modules**.

✅ Benefits:

- 🧠 Clean and readable `main.py`
- 🧪 Easy to test & scale
- ♻️ Promotes code reuse
- 📚 Organized by features (e.g., users, products, auth)

📁 Suggested Folder Structure (Scalable)

```
fastapi_backend/
│
├── app/
│   ├── main.py               # 🚀 Entry point
│   ├── routers/              # 📦 All route files
│   │   ├── __init__.py
│   │   ├── users.py          # 👤 /users routes
│   │   └── products.py       # 📦 /products routes
│   ├── models/               # 📄 Pydantic schemas
│   │   └── user.py
│   ├── database/             # 🛢️ DB connection/config
│   │   └── db.py
│   └── utils/                # 🛠️ Helpers / validators
│       └── helpers.py
│
├── requirements.txt
└── venv/
```

🔧 Step 1: Refactor `main.py` to Include Routers

```python
# app/main.py

from fastapi import FastAPI
from app.routers import users, products
```

```python
app = FastAPI(title="Modular FastAPI App")

# 👇 Include routers
app.include_router(users.router, prefix="/users", tags=["Users"])
app.include_router(products.router, prefix="/products", tags=["Products"])
```

📦 Step 2: Create First Router – `users.py`

```python
# app/routers/users.py

from fastapi import APIRouter

router = APIRouter()

@router.get("/")
def get_all_users():
    return [{"id": 1, "name": "Alice"}, {"id": 2, "name": "Bob"}]

@router.post("/")
def create_user(name: str):
    return {"msg": f"User '{name}' created"}
```

📌 This exposes:

- `GET /users/`
- `POST /users/`

📦 Step 3: Add Second Router – `products.py`

```python
# app/routers/products.py

from fastapi import APIRouter

router = APIRouter()

@router.get("/")
def list_products():
    return ["Laptop", "Tablet", "Phone"]

@router.post("/")
def add_product(name: str):
    return {"msg": f"Product '{name}' added"}
```

📌 This exposes:

- `GET /products/`
- `POST /products/`

📘 Summary: `include_router` Parameters

Argument	Example Value	Purpose
`router`	`users.router`	Router object
`prefix`	`"/users"`	URL path prefix
`tags`	`["Users"]`	Swagger tag grouping

📚 Swagger UI Structure

When you visit `http://127.0.0.1:8000/docs`, you'll see:

```
📁 Users
  - GET /users/
  - POST /users/

📁 Products
  - GET /products/
  - POST /products/
```

💡 Pro Tip: Reusability

- Each `router` file is like its own **mini app**
- You can **import it into other projects** or mount it under different prefixes
- You can even use **sub-routers** for nested routes

🔁 Bonus: Modularizing Further

Add Pydantic validation, services, and database logic to separate files:

- `/models/user.py`: for request & response schemas
- `/services/user_service.py`: for business logic
- `/database/db.py`: DB connection

📦 Example: Using a Pydantic Model in `users.py`

```python
# app/models/user.py

from pydantic import BaseModel

class UserCreate(BaseModel):
    name: str
    email: str
```

```python
# app/routers/users.py

from fastapi import APIRouter
from app.models.user import UserCreate

router = APIRouter()

@router.post("/")
def create_user(user: UserCreate):
    return {"msg": f"User '{user.name}' created with email '{user.email}'"}
```

✅ Automatically shows fields in Swagger ✅ Auto-validates input and returns `422` on error

---

🗒️ Final Recap: What You've Learned

Concept	Summary
`APIRouter()`	Used to define routes in separate files
`include_router()`	Used in `main.py` to mount routers with prefixes
`prefix`	Base path for all routes in that router
`tags`	Group routes visually in Swagger UI
Refactoring	Makes code scalable, testable, clean
Second Router	Works exactly like the first – just plug & play

---

FastAPI folder structure

✅ Your MERN Stack Folder Recap (`YouTweet/backend/src`):

```
📁 src/
|
```

```
├── components/       → Reusable frontend components
├── config/           → Config files (env, DB, etc.)
├── controllers/      → Route logic (Express handlers)
├── db/               → MongoDB connection
├── middlewares/      → Express middleware (auth, error, etc.)
├── models/           → Mongoose schemas
├── pages/            → (Next.js / frontend routing - N/A in backend)
├── routes/           → Express routes
├── store/            → Redux store (frontend concern)
├── utils/            → Utility functions
├── app.js            → Express app
├── index.js          → Entry point
```

⚙️ Equivalent **FastAPI Backend Structure**

```
📁 app/
│
├── main.py                  # Entry point (like `index.js`)
├── config/                  # Env vars, DB URLs
│   └── settings.py
│
├── core/                    # App configuration, startup logic
│   └── init_db.py
│   └── security.py
│
├── api/                     # Routers (like Express `routes/`)
│   ├── deps.py              # Dependencies for routes
│   ├── v1/                  # Versioned routes
│   │   ├── endpoints/
│   │   │   ├── user.py
│   │   │   └── auth.py
│   │   └── api.py           # Include all v1 routers
│
├── models/                  # SQLAlchemy / Pydantic models
│   ├── user.py
│   └── tweet.py
│
├── schemas/                 # Pydantic schemas (like Mongoose shape + Joi
validation)
│   ├── user.py
│   └── tweet.py
│
├── services/                # Business logic (like controllers)
│   └── tweet_service.py
│
├── db/                      # DB session, connection, migrations
│   ├── base.py
│   └── session.py
```

```
│
├── middlewares/            # Custom middlewares
│   └── error_handler.py
│
├── utils/                   # Helper functions
│   └── hashing.py
│
├── static/                  # Static files (like `public/`)
│
└── .env
```

🧠 Key Mappings Between MERN and FastAPI

MERN (Express.js)	FastAPI Equivalent	Notes
`routes/`	`api/v1/endpoints/`	Use routers and include them
`controllers/`	`services/`	Handles logic for routes
`models/` (Mongoose)	`models/` + `schemas/`	Use SQLAlchemy + Pydantic
`middlewares/`	`middlewares/`	Custom middleware via `add_middleware`
`config/`	`config/` + `.env`	`pydantic.BaseSettings` or `dynaconf`
`utils/`	`utils/`	Same utility concept
`app.js` or `index.js`	`main.py`	Entry file, loads FastAPI app
`db/`	`db/session.py`, `init_db`	For SQLAlchemy sessions

🧰 Tech Stack Used in FastAPI Equivalent

Concern	Tool Used
Routing	FastAPI Routers
Models (DB)	SQLAlchemy
Data Validation	Pydantic
Environment Handling	Python-dotenv or Pydantic Settings
Middleware	FastAPI's `add_middleware`
Static Files	`app.mount("/static", ...)`
ORM Migrations	Alembic (optional)

💡 Tips for Scaling Like MERN

- Use **versioned APIs** in `api/v1/endpoints/`
- Separate **schemas** (validation) from **models** (DB)
- Use `services/` layer to keep logic separate from route files
- Autoload routers using a single `api.py` file in `v1/`

🧩 1. Folder Structure (Express-like in FastAPI)

Here's a folder structure mimicking a typical MERN backend:

```
you_tweet_fastapi_backend/
│
├── app/
│   ├── main.py                 # Entry point
│   ├── models/                 # Pydantic models / SQLAlchemy models
│   │   └── user.py
│   ├── routes/                 # Routers (like Express routes)
│   │   ├── __init__.py
│   │   ├── user.py             # All user-related routes
│   │   └── tweet.py
│   ├── services/               # Business logic
│   └── database/               # DB config/connection
│       └── db.py
│
└── requirements.txt
```

🚀 2. How to Create Modular Routers with Prefix and Tags

📁 `routes/user.py`

```python
from fastapi import APIRouter

router = APIRouter(
    prefix="/users",
    tags=["Users"]
)

@router.get("/")
def get_users():
    return {"message": "Get all users"}

@router.post("/register")
def register_user():
    return {"message": "Register a user"}
```

📁 routes/tweet.py

```
from fastapi import APIRouter

router = APIRouter(
    prefix="/tweets",
    tags=["Tweets"]
)

@router.get("/")
def get_tweets():
    return {"message": "Get all tweets"}

@router.post("/")
def post_tweet():
    return {"message": "Post a tweet"}
```

📄 main.py (Entry Point)

```
from fastapi import FastAPI
from app.routes import user, tweet

app = FastAPI()

# Include routers with shared prefix and tags
app.include_router(user.router)
app.include_router(tweet.router)

@app.get("/")
def root():
    return {"message": "Welcome to YouTweet API"}
```

📘 3. Output in Swagger UI

Once you run: x

```
uvicorn app.main:app --reload
```

Open your browser at http://127.0.0.1:8000/docs — you'll see the API docs grouped under:

- 📁 **Users**
- 📁 **Tweets**

Each containing the relevant endpoints (`/users/`, `/tweets/`), just like you'd expect in Postman or Swagger in a Node.js project.

🏷️ Optional: Add Dependencies or Tags per Operation

```
@router.get("/{user_id}", tags=["Users", "Get User by ID"])
def get_user(user_id: int):
    ...
```

🚦 FastAPI Routing Flow – Full Guide

🧱 1. Folder Structure

A clean and modular FastAPI project typically looks like:

```
fastapi_app/
├── main.py               # Entry point
├── routers/
│   ├── __init__.py
│   ├── blog.py           # Blog-related routes
│   └── user.py           # User-related routes
├── models/
│   └── blog.py           # Pydantic schemas
├── database.py           # (Optional) DB config
└── requirements.txt
```

📌 2. Entry Point: `main.py`

```
from fastapi import FastAPI
from routers import blog, user

app = FastAPI(title="My Blog API 🚀")

# Registering routers
app.include_router(blog.router, prefix="/blog", tags=["Blog"])
app.include_router(user.router, prefix="/user", tags=["User"])

@app.get("/", tags=["Root"])
```

```python
def read_root():
    return {"message": "Welcome to the Blog API!"}
```

📁 3. Blog Router: `routers/blog.py`

```python
from fastapi import APIRouter, Path, Query, Body
from pydantic import BaseModel
from typing import Optional

router = APIRouter()

# Pydantic model
class Blog(BaseModel):
    title: str
    content: str
    author: Optional[str] = "Anonymous"

@router.get("/", summary="Get all blogs")
def get_blogs(limit: int = Query(10, description="Limit number of blogs")):
    return {"message": f"{limit} blog(s) retrieved"}

@router.get("/{id}", summary="Get blog by ID")
def get_blog(id: int = Path(..., gt=0)):
    return {"id": id, "title": "Sample Blog"}

@router.post("/", summary="Create a new blog")
def create_blog(blog: Blog = Body(...)):
    return {"message": "Blog created!", "data": blog}
```

📁 4. User Router: `routers/user.py`

```python
from fastapi import APIRouter
from pydantic import BaseModel

router = APIRouter()

class User(BaseModel):
    username: str
    email: str

@router.get("/", summary="List all users")
def list_users():
    return {"users": ["Darshan", "Alice", "Bob"]}
```

```python
@router.post("/", summary="Register a user")
def register_user(user: User):
    return {"message": f"User {user.username} registered!"}
```

🧠 5. How It All Works

FastAPI Concept	Express Equivalent (MERN)
`APIRouter()`	`express.Router()`
`@app.include_router()`	`app.use('/path', router)`
`@router.get()` / `.post()`	`router.get()`, `router.post()`
`BaseModel` (Pydantic)	`Joi`, `mongoose.Schema`
`Query`, `Path`, `Body`	`req.query`, `req.params`, `req.body`

🔥 Bonus: Swagger Docs

When you run:

```
uvicorn main:app --reload
```

Open in browser:

```
http://127.0.0.1:8000/docs
```

You'll see:

- All routes grouped by **tags**
- Metadata like **summary**, **descriptions**, **examples**
- Built-in testing UI

🧪 Optional: Add Response Model

To enforce return type:

```python
from typing import List

@router.get("/", response_model=List[Blog])
def get_blogs():
```

```
    return [
        {"title": "One", "content": "First blog", "author": "Darshan"},
        {"title": "Two", "content": "Second blog"}
    ]
```

✅ Summary Routing Flow

```
main.py
│
├── Creates FastAPI app
├── Includes modular routers with prefixes
│
├── /blog routes → handled in routers/blog.py
├── /user routes → handled in routers/user.py
│
├── All routes auto-documented via Swagger UI
```

🛠️ Tools/Commands

```
uvicorn main:app --reload          # Start dev server
pip install fastapi uvicorn        # Install deps
```
