BACK-END DEVELOPMENT

WEEK 4 – REST API Design + Modular Express

CADT
IDT

After Finishing This Lecture:

- ➢ Understand REST principles
- ➢ Design clean API routes
- ➢ Modularize Express apps from top (entry point) to bottom (features)
- ➢ Learn to build a modular Express server
- ➢ Identify good REST API design patterns
- ➢ Apply top-down architecture to Node.js projects

What is an API?

API (Application Programming Interface) is a set of rules that allows two software applications to communicate with each other.

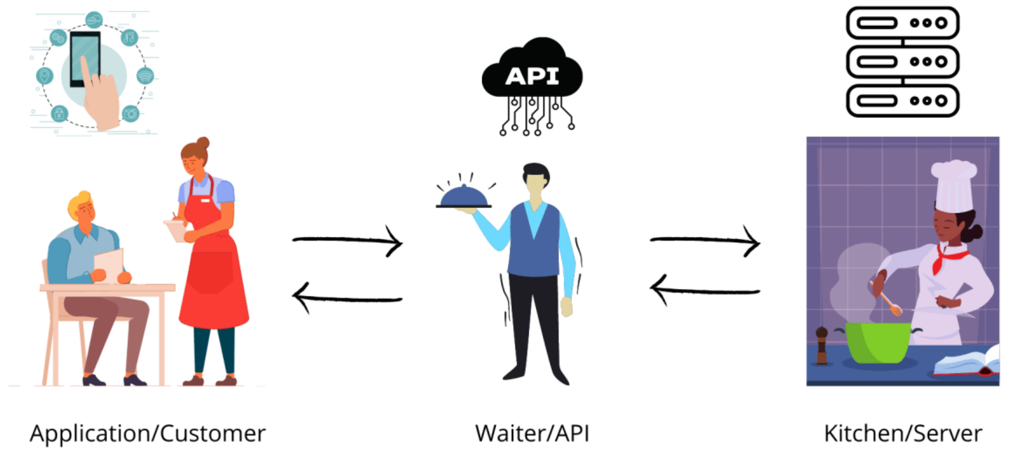
- ✓ Acts as a **messenger** between systems.
- ✓ Hides internal implementation details, exposing only what's necessary.
- ✓ Can be **public** (e.g., Twitter API), **private**, or **partner-only**.

What is an API? (Cont)

Real-World Analogy:

A **restaurant menu** is like an API:

- You (the client) use the menu to request food.
- The kitchen (the server) prepares it without you knowing how.
- The waiter (API) delivers the result.



What is REST?

REST = Representational State Transfer

- Coined by Roy Fielding in his 2000 Ph.D. dissertation
- A design pattern or **architectural style** for building scalable web services
- REST uses standard **HTTP methods** to perform operations on **resources**, which are identified by **URLs**.

Core REST Concepts

Term	Meaning
Resource	Any object or data entity (e.g., user, post, product)
Representation	A format to represent the resource (typically JSON)
State Transfer	Client sends and receives the current state of the resource

Example Resource

GET /posts/123 - **Retrieve a specific post**

```
{
  "id": 123,
  "title": "What is REST?",
  "content": "REST stands for Representational State Transfer..."
}
```

POST /posts/ - **Create a new post**

```
POST /posts
Content-Type: application/json

{
  "title": "Understanding HTTP Methods",
  "content": "Let's learn about GET, POST, PUT, and DELETE..."
}
```

HTTP Methods

Method	**Action**	**Description**
GET	Read	Fetch a resource
POST	Create	Add a new resource
PUT	Update	Replace an existing resource
PATCH	Partial Update	Modify part of a resource
DELETE	Delete	Remove a resource

Key REST Concepts

Resource
A resource represents a logical data entity. Each resource is identified by a URI and manipulated using standard HTTP methods.

Examples:

- User: Represents a person using your app
- Post: A blog article or comment

Key REST Concepts

URI (Uniform Resource Identifier)
URI is the address used to access resources. RESTful APIs use URIs to uniquely identify resources.

Examples:

- `GET /api/users` – list of users
- `GET /api/posts/123` – a specific post with ID 123

Key REST Concepts

Stateless
Each request from a client to the server must contain all necessary information to process it. The server does not retain any session state between requests.

Implications:

- Improves scalability
- Makes APIs easier to debug and cache

Key REST Concepts

Representation
A resource can have multiple representations (e.g., JSON, XML). Clients interact with the representation, not the actual resource.

Most Common: JSON (JavaScript Object Notation)

```
{
  "id": 123,
  "title": "REST Basics",
  "author": "Alice"
}
```

RESTful Endpoint Design

Use Plural Nouns for Routes

- Resources should be treated as collections.
- Stick to nouns, not verbs.
- **Why?** It reflects REST's resource-oriented architecture.

HTTP Verb	**Endpoint**	**Description**
GET	/api/posts	Get all blog posts
GET	/api/posts/:id	Get a single post by ID
POST	/api/posts	Create a new blog post
PATCH	/api/posts/:id	Update an existing post
DELETE	/api/posts/:id	Delete a post

RESTful Endpoint Design

Avoid Verbs in the URI

Bad:

- `/api/getPosts`
- `/api/createPost`

Good:

Use HTTP methods to convey actions:

```
GET /api/posts (not /getPosts)
POST /api/posts (not /createPost)
```

RESTful Endpoint Design

Sub-resources / Nested Resources

For relationships between resources:

- `GET /api/posts/123/comments` – Comments on post #123
- `POST /api/posts/123/comments` – Add comment to post #123

HTTP Status Codes

HTTP status codes are 3-digit responses sent by the server to indicate the result of a client's request.

Success Codes

- **200 OK** – Request succeeded
Example: Successfully retrieved a list of posts with
`GET /api/posts`

- **201 Created** – Resource was successfully created
Example: A new post was created with
`POST /api/posts`

HTTP Status Codes

Client Error Codes

- **400 Bad Request** – The request is malformed or missing required data
  Example: Missing title field in `POST /api/posts`

- **401 Unauthorized** – Client must authenticate before accessing the resource
  Example: Accessing a protected route without a valid token

- **404 Not Found** – The requested resource does not exist
  Example: `GET /api/posts/9999` when post ID 9999 doesn't exist

Server Error Code

- **500 Internal Server Error** – Something went wrong on the server
  Example: An unexpected exception or database crash

Traditional Express Setup

One File = All Logic

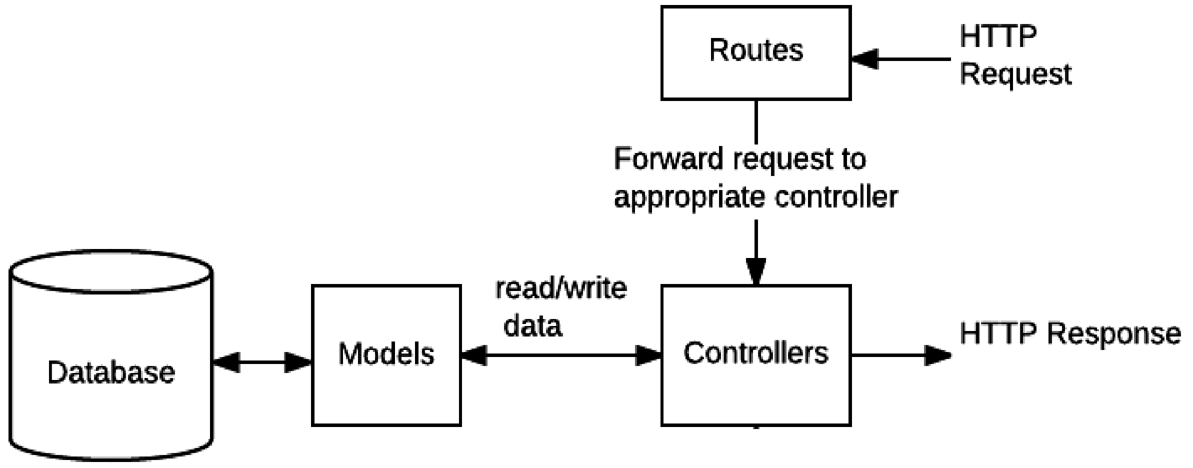
- Typical beginners write everything in a single file: `server.js`
- Routes, middleware, controllers, and DB logic are all jammed together

Problems with This Approach:

- **Hard to maintain**: Adding new features becomes messy
- **No separation of concerns**: Logic for different parts of the app is mixed
- **Difficult to test**: You can't easily isolate logic
- **Poor scalability**: Hard to onboard new developers or grow the codebase

Separation of Concern

- Each layer handles a single responsibility:
  - **Routes**: Handle URLs and method mapping
  - **Controllers**: Handle request/response logic
  - **Models**: Handle data interaction
- Cleaner and more understandable code



Sample Source Code:
https://github.com/KimangKhenng/express-separate

Additional Reading

- Rest API Design Guideline: https://restfulapi.net/
- Sample Rest API Design: https://petstore.swagger.io

Questions

Q1: Can I return HTML from a REST API?

Q2: What’s the role of models if I’m not using a database yet?

Q3: How should I name nested resources?

Answers

A1: Technically yes, but REST APIs usually return **JSON**. Returning HTML is more typical in traditional server-rendered web apps.

A2: You can still define mock data or schema structure to simulate interactions and prepare for later DB integration.

A3: Use logical nesting, e.g.:
`GET /api/posts/:postId/comments`
This shows that comments belong to a specific post.

WHAT WE HAVE LEARNT

- Understand REST principles
- Design clean API routes
- Modularize Express apps from top (entry point) to bottom (features)
- Learn to build a modular Express server
- Identify good REST API design patterns
- Apply top-down architecture to Node.js projects