Azure API Training <> Revanture

Day 7

REST API Architecture and Principles

What is REST API ?

REST is an acronym for **RE**presentational **S**tate **T**ransfer and an architectural style for distributed hypermedia systems.

Roy Fielding first presented it in 2000 in his famous dissertation

REST is not a protocol or a standard, it is an architectural style

Before REST APIs

Before REST became popular (around early 2000s), developers used older communication methods for systems to talk to each other.

- SOAP (Simple Object Access Protocol)
- RPC (Remote Procedure Call)
- CORBA / DCOM

Example of SOAP

```
<soap:Envelope>
  <soap:Body>
    <GetUser>
      <UserId>42</UserId>
    </GetUser>
  </soap:Body>
</soap:Envelope>
```

~120B in size

Example of JSON

{"id":42}

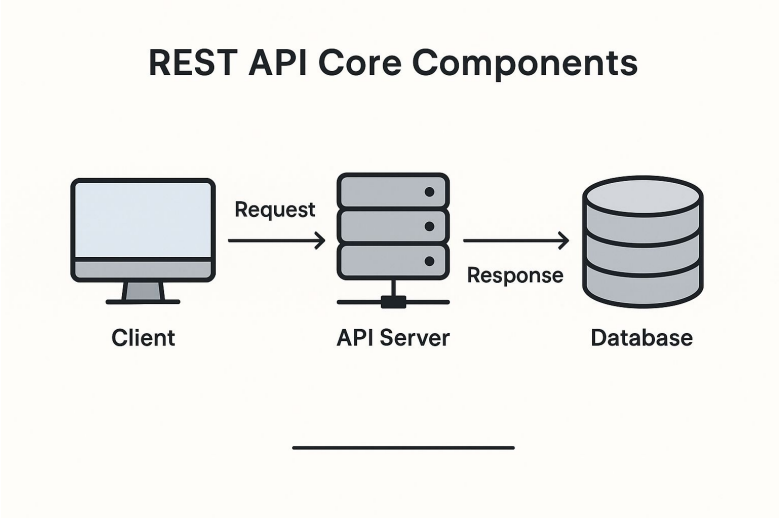
~9B in size

REST API Architecture

REST architecture (Representational State Transfer) is a style of designing networked systems where resources are accessed via standard HTTP methods (GET, POST, PUT, DELETE).

It defines how clients and servers communicate over the internet in a stateless, uniform, and scalable way.

Core Components of REST Architecture



Core Components of REST Architecture

Client

- Sends requests to the server.
  Examples:
  - Web browsers
  - Mobile apps
  - Frontend applications (React, Angular, etc.)

Core Components of REST Architecture

Server

Processes the client’s request.
Performs CRUD operations on resources.
Sends back the response (usually JSON).

Core Components of REST Architecture

Resource

The main thing that’s being operated on (like a User, Product, or Order).
Each resource has a unique URI (Uniform Resource Identifier).

REST API Principles

REST API Principles

Uniform Interface

By applying the principle of generality to the components interface, we can simplify the overall system architecture and improve the visibility of interactions

Client-Server

The client-server design pattern enforces the separation of concerns, which helps the client and the server components evolve independently.

Stateless

Statelessness mandates that each request from the client to the server must contain all of the information necessary to understand and complete the request.

REST API Principles

Cacheable

The cacheable constraint requires that a response should implicitly or explicitly label itself as cacheable or non-cacheable.

Layered System

The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior.

HTTP Methods

What Are HTTP Methods?

HTTP (HyperText Transfer Protocol) defines a set of request methods that tell the server what kind of operation the client wants to perform on a resource (like a user, product, or file).

In REST APIs, we mainly use GET, POST, PUT, PATCH, and DELETE to represent CRUD operations:

What Are HTTP Methods?

CRUD Operation	HTTP Method	Example	Purpose
Create	POST	/users	Add a new user
Read	GET	/users or /users/1	Fetch data
Update	PUT or PATCH	/users/1	Modify existing data
Delete	DELETE	/users/1	Remove data

Installing FAST API

What is FAST API?

FastAPI is a modern, fast (high-performance), web framework for building APIs with Python based on standard Python type hints.

Installing FAST API

My Versions

Python = 3.14

Pip = 25.2

pip install "fastapi[standard]"

Write a Hello World in Fast API

Write a basic hello world code in fast api

Get the code

Copy the code in your text editor, open terminal and write the following command

fastapi dev filename.py

Follow the instructions

How to Test API?

Write a Hello World in Fast API

We can use many tools for this, but we are going to use postman for testing our api!

Download Postman

Status Code and Response Format

What Are HTTP Status Codes?

Every REST API response includes a status code that indicates the result of the request — success, failure, or error.

Each code is a 3-digit number, grouped by the first digit:

Category	Range	Meaning
1xx	100–199	Informational (rare in APIs)
2xx	200–299	Success
3xx	300–399	Redirection
4xx	400–499	Client error (your request is wrong)
5xx	500–599	Server error (server failed to handle it)

Common HTTP Status Codes in REST APIs

Code	Meaning	When Used
200 OK	Request succeeded	GET, PUT, PATCH, DELETE
201 Created	New resource created	POST (e.g., new user added)
202 Accepted	Request accepted but processing later	Async APIs or queues
204 No Content	Success but no response body	DELETE or empty response

Common HTTP Status Codes in REST APIs

Code	Meaning	When Happens
400 Bad Request	Invalid input / malformed JSON	Missing required field
401 Unauthorized	No valid auth credentials	Missing token
403 Forbidden	Authenticated but not allowed	No permission
404 Not Found	Resource doesn’t exist	Invalid ID or endpoint
405 Method Not Allowed	Wrong HTTP method used	POST to a GET-only route
409 Conflict	Resource already exists	Duplicate email or ID
422 Unprocessable Entity	Validation failed	Wrong data type or format

Common HTTP Status Codes in REST APIs

Code	Meaning	When Happens
500 Internal Server Error	Generic server failure	Bug or crash in backend
502 Bad Gateway	Gateway received invalid response	Reverse proxy issue
503 Service Unavailable	Server down / overloaded	Maintenance or crash
504 Gateway Timeout	Server took too long to respond	Slow backend or DB timeout

What is a Response Format?

Most modern APIs use JSON as the standard response format.

A success response format

```
{
  "status": "success",
  "data": {
    "id": 42,
    "name": "Avinash Seth",
    "email": "avinash@example.com"
  }
}
```

An error response format


```
{
  "status": "error",
  "code": 404,
  "message": "User not found"
}
```

Response format depends on your organization

Good API Response Design Tips

Do	Avoid
Always include proper status codes	Returning 200 for everything
Use consistent JSON structure	Mixing formats (XML + JSON)
Return helpful error messages	Showing raw server errors
Use plural nouns for endpoints	/user instead of /users
Include timestamps and pagination if needed	Huge unpaginated responses

API Versioning Strategies

Why API Versioning?

- Maintain backwards compatibility while adding features or fixing design mistakes.
- Let clients upgrade on their schedule.
- Provide a clear lifecycle (deprecation → removal).
- Avoid breaking integrations in production.

Common versioning strategies

1) URI Path Versioning (aka URL versioning)

Example: GET /v1/users/42 or GET /api/v1/users/42

Pros

- Obvious and discoverable.
- Easy to cache, route, document, and test.
- Simple for clients to switch versions.

Cons

- Versions become part of resource identity (not purely RESTful).
- Encourages duplicate endpoints and more routing work.

When to use: Most widely used; great for public APIs and clear breaking changes.

Query Parameter Versioning

Example: GET /users/42?version=1 or GET /users/42?v=1

Pros

- Simple to implement.
- Keeps URL resource path unchanged.

Cons

- Less visible/explicit than path versioning.
- Some caching proxies may ignore query parameters unless configured.

When to use: Internal or experimental APIs where path cleanliness is preferred.

Header Versioning (Custom header)

Example: GET /users/42 with header X-API-Version: 1

Pros

- Keeps URIs clean and resource-centric.
- Good for enterprise/internal APIs that control clients.

Cons

- Harder to test in browsers (can't easily paste a URL).
- Not as discoverable; proxies and caching may need extra config.
- Slightly more complex to implement and document.

When to use: When you want to separate versioning from URL and are comfortable controlling clients.

Hands on

Lets see how we can achieve this using fastapi

Source Code

Best practices & recommendations

- Prefer major-only versioning for public APIs (e.g., v1, v2). Minor features should be additive and not require a new major.
- **Document clearly**: API docs must show available versions, breaking changes, migration guides.
- **Deprecation policy**: Publish clear timelines (e.g., deprecate v1 on Jan 1, 2026; remove 6 months later). Notify clients.
- **Use headers for capability negotiation**: Let clients advertise supported features (e.g., Accept or X-Client-Features).
- **Keep changes additive when possible**: Add fields instead of changing or removing them.
- **Automate testing across versions**: CI should run test suites for all supported versions.
- Version your API surface, not internal implementation: Don’t expose internal package or microservice versions to clients.
- **Backwards-compatible error handling**: Keep error formats stable across non-breaking changes.
- **Support both old and new for a transition period**: Proxy or gateway can route old API traffic to compatibility layers if needed.
- **Think about storage/migration**: If changing resource formats, plan migrations, or use transformation layers.

Resource Naming conventions

What is a Resource?

- In REST, everything that can be operated on — users, products, orders, comments — is a resource.
- Each resource is identified by a URI (Uniform Resource Identifier).
- Examples
  - */users/42*
  - */products/15*
  - */orders/2024-0099*

Use nouns, not verbs

- Use following
  - GET /users
  - POST /users
  - GET /users/1
  - DELETE /users/1
- Avoid
  - GET /getUsers
  - POST /createUser
  - DELETE /deleteUser

Use plural nouns for collections

- Use following
  - /users
  - /users/42
  - /orders
  - /orders/5
- Avoid
  - /user
  - /user/42

Use forward slashes (/) to show hierarchy

- Use following
  - /users/42/orders
  - /orders/5/items
  - /companies/99/employees
- Avoid
  - /getUserOrders?userId=42

Don’t use file extensions

- Use following
  - /users/42
- Avoid
  - /users/42.json
  - /users/42.xml

Use lowercase letters

- Use following
  - /users/42
  - /products/on-sale
- Avoid
  - /Users/42
  - /Products/OnSale

Use hyphens (-) for multi-word names

- Use following
  - /customer-orders
  - /product-reviews
- Avoid
  - /customer_orders
  - /customerOrders

Use query parameters for filtering, sorting, searching

- Use following
  - GET /users?role=admin
  - GET /products?category=coffee&sort=price_desc
  - GET /orders?status=shipped&limit=20&page=2
- Avoid
  - GET /getUsersByRole/admin

Handle actions with subresources, not verbs

- Use following
  - POST /users/42/activate
  - POST /orders/123/cancel
- Avoid
  - POST /orders/123?action=cancel

Use consistent structure for CRUD operations

HTTP Method	Example URI	Purpose
GET	/users	List all users
GET	/users/{id}	Get single user
POST	/users	Create user
PUT	/users/{id}	Replace user
PATCH	/users/{id}	Update partial fields
DELETE	/users/{id}	Delete user

API Design Best Practices

Idempotency and Safety

What is an Idempotent API?

- Idempotency essentially means that the effect of a successfully performed request on a server resource is independent of the number of times it is executed.
- **For example**, in arithmetic, adding zero to a number is an idempotent operation.

When we design the REST APIs, we must realize that API consumers can make mistakes. Consumers can write the client code in such a way that there can be duplicate requests coming to the APIs.

HTTP Method Mapping

Method	Safe? (Read only)	Idempotent?	Typical use
GET	Yes	Yes	Read resource
HEAD	Yes	Yes	Read headers only
POST	No	No	Create resource / non-deterministic actions
PUT	No	Yes	Replace/overwrite resource
PATCH	No	Usually *not* idempotent by default	Partial update
DELETE	No	Yes	Remove resource

Idempotency in practice

- Idempotency key
- Client-generated resource IDs
- Make operations idempotent by design

Pagination and Filtering

Filtering

Filtering

- URL parameters is the easiest way to add basic filtering to REST APIs.

Example

- GET /hired?company=microsoft

However, this only works for exact matches. What if you want to do a range such as a salary range or date range?

LHS Brackets

- One way to encode operators is the use of square brackets [] on the key name.

Example

- GET /buy?price[gte]=10&price[lte]=100

May require more work on server side to parse and group the filters

RHS Colon

- Similar to the bracket approach, you can design an API to take the operator on the RHS instead of LHS.

Example

- GET /buy?price=gte:10&price=lte:100

May require more work on server side to parse and group the filters

Search Query Param

- If you require search on your endpoint, you can add support for filters and ranges directly with the search parameter.

Example

- GET /buy?search=title:mobile phone AND price:[10 TO 100]

May require more work on server side to parse and group the filters

Pagination

Pagination

- Most endpoints that returns a list of entities will need to have some sort of pagination.
- Without pagination, a simple search could return millions or even billions of hits causing extraneous network traffic.

Example

- GET /hired?company=microsoft

Offset Pagination

- This is the simplest form of paging. Limit/Offset became popular with apps using SQL databases which already have LIMIT and OFFSET as part of the SQL SELECT Syntax. Very little business logic is required to implement Limit/Offset paging.

Example

- GET /course?offset=100&limit=20

SQL Example

- SELECT * from `courses` OFFSET=100 LIMIT 20 ORDER BY ID ASC;

Keyset Pagination

- Keyset pagination uses the filter values of the last page to fetch the next set of items. Those columns would be indexed.

Example

- GET /products?limit=20&created=lte:2025-10-13 22:00:00

SQL Example

- SELECT * from `products` WHERE created_at <= ‘2025-10-13 22:00:00’ LIMIT 20;

ERROR Handling Patterns

Importance of Error Handling

Proper error handling is a crucial aspect of any application, but it becomes even more important in REST APIs. Here are some reasons why:

- Better debugging and troubleshooting
- Improved user experience
- Increased maintainability

Best Practices for Error Responses

When your API encounters an error, it should return a response with the appropriate status code and a well-structured error object.

- Use appropriate HTTP status codes
- Provide a consistent error response structure
- Include helpful error messages
- Return relevant error details

Example #1


```
{
  "error": {
    "code": "ERR_INVALID_INPUT",
    "message": "Invalid input data.",
    "details": [
      {
        "field": "username",
        "message": "Username is required."
      },
      {
        "field": "email",
        "message": "Email is invalid."
      }
    ]
  }
}
```

Example X (Twitter)


```
{
  "errors": [
    {
      "code": 215,
      "message": "Bad Authentication data."
    }
  ]
}
```

Example Facebook


```
{
   "error": {
      "message": "Missing redirect_uri parameter.",
      "type": "OAuthException",
      "code": 191,
      "fbtrace_id": "AWswcVwbcqfgrSgjG80MtqJ"
   }
}
```

Rate Limiting Concepts

What is Rate Limiting?

Rate limiting refers to controlling the number of requests a client can make to an API within a specified period.

If the API request count exceeds the threshold defined by the rate limiter, all the excess calls are blocked.

The following rules can be examples of rate limiting an API

- A client can send no more than 20 requests per second.
- A client can send no more than 1000 requests within a minute.
- A client can send no more than 100,000 requests per day.

Best Practices

- Apply Limits at Different Levels
- Implement Graceful Error Handling
- Monitor and Log Requests
- Plan for Burst Control Mechanisms
- Setup Alert

How to Implement Rate Limit

Small Applications

In-Memory Rate Limiting

Description:

- Stores request counters locally within the application memory.
  Simple and fast for single-instance or small APIs.

Tools/Technologies:

- Python Dictionary
- Java HashMap
- Node.js Map

Use Case:

Ideal for prototypes or monolithic applications with one server instance.

Database-Backed Rate Limiting

Database Storage

Description

- Request data and counters stored in a relational database.
- Useful when you want persistence across restarts.

Tools/Technologies

- MySQL
- PostgreSQL
- SQLite

Use Case

Suitable for low-to-medium traffic apps where persistent logs of requests are useful.

Distributed Applications

Distributed Cache Based

Description

- Counters are stored in a distributed cache shared across servers.
- Ensures consistent limits in multi-instance deployments.

Tools/Technologies

- Redis
- Memcached

Use Case

Perfect for horizontally scaled APIs with multiple servers.

Edge / CDN Level

Network-Edge Rate Limiting

Description

- Limits requests before they even reach your API servers.
- Implemented at the CDN or reverse-proxy layer.

Tools/Technologies

- Cloudflare
- Akamai
- Fastly

Use Case

High-traffic public APIs where early request rejection reduces backend load.