API

An API is a set of rules that allow different software applications to communicate with each other. It defines how one app can request data from another app and how the data should be returned.

API stands for Application Programming Interface. It allows different software systems to interact by defining how requests and responses should be structured between them.

How do APIs work?

APIs work by having a client (the app making the request) send a request to a server (the app or system providing the data or service). The server then processes the request and sends back a response with the requested data.

Types of APIs

SOAP APIs : Older type, uses XML to exchange data between client and server.

RPC APIs : The client sends a request to the server to perform a task, and the server sends the result back.

REST APIs : The most common type, where the client sends a request with data, and the server responds with data.

WebSocket APIs : Supports two-way communication between client and server, where the server can send updates to the client.

REST Principles

REST (Representational State Transfer) is an architectural style used for designing networked applications. It operates over HTTP and is based on a set of principles and constraints that define how clients and servers communicate. RESTful APIs are commonly used in web development.

Constraints of REST

Client–Server

REST architecture is based on the separation of concerns between the client (requesting the service) and the server (providing the service). This allows for better flexibility and scalability.

Statelessness

Each request from the client to the server must contain all the information the server needs to understand and respond to the request. The server does not store any information about the client's state between requests. Each request is independent

Cacheability

Responses from the server can be marked as cacheable or non-cacheable. If the response is cacheable, the client can reuse it for subsequent requests, improving performance.

Uniform Interface

A uniform interface simplifies the architecture by defining standard conventions for communication. This includes using standard HTTP methods like GET, POST, PUT, DELETE, etc., and standardizing URLs to identify resources.

Layered System

The architecture can be composed of multiple layers, such as intermediaries (e.g., proxies, gateways) that can handle requests and responses to improve performance, security, or scalability. Each layer only knows about the layer directly adjacent to it.

Code on Demand (optional):

Servers can temporarily provide executable code (like JavaScript) to clients to extend the functionality of the client. This is optional in REST.

HTTP verbs (methods) and their meanings

GET

Retrieve a resource.

POST

Create a new resource.

PUT

Update or replace an existing resource.

DELETE

Remove a resource.

PATCH

Partially update an existing resource.

API Versioning

API versioning is the process of managing changes to an API while ensuring that existing clients continue to work even as new functionality or updates are introduced. This allows developers to improve or modify their API without breaking existing integrations.

Why Versioning is Necessary

Backward Compatibility

Backward compatibility ensures older versions of an API continue to work as expected, even when new versions are released. This avoids disruptions for clients who may not be ready to update their code immediately.

Controlled Upgrades

Controlled upgrades allow new features or improvements to be introduced gradually, without affecting existing functionality. This enables users to adopt changes at their own pace, reducing the risk of breaking existing integrations.

Deprecation Strategy

A deprecation strategy gradually removes outdated API features or endpoints while giving users time to transition. This ensures users are informed and can adapt to changes without sudden disruptions.

API versioning strategies

URL-Based Versioning

Uses version numbers in the URL (e.g., /v1/users, /v2/users).

Pros: Simple, explicit, and easy to understand.

Cons: Alters URL structure, can clutter routes if many versions exist.

Header-Based Versioning

Versioning is handled via HTTP headers (e.g., Accept: application/vnd.myapp.v2+json).

Pros: Keeps URLs clean.

Cons: Requires clients to manage headers, slightly more complex setup.

Query Parameter Versioning

Specifies the version as a query parameter (e.g., /users?version=2).

Pros: Easy to implement.

Cons: Less common for large-scale public APIs, can be overlooked or misused by clients.

Controlled upgrades allow new features or improvements to be introduced gradually, without affecting existing functionality. This enables users to adopt changes at their own pace, reducing the risk of breaking existing integrations.

Key Point: Choose a versioning strategy that suits your organization's workflow and is easy for external integrators to adopt.

Designing URL Structures

Designing clean, intuitive, and consistent URL (Uniform Resource Locator) structures is crucial when building RESTful (and other) APIs. A well-designed URL structure makes your API easier to understand, adopt, and maintain.

Resource-Oriented Design

Use nouns instead of verbs.

✓ `/products`

✗ `/getProducts`

Keep it hierarchical to reflect real-world relationships

✓ `/users/{userId}/orders`

✓ `/orders/{orderId}/items`

URL Naming Conventions

Plural vs. Singular

Consistency is key. Decide on one convention and stick to it

e.g., `/users`, `/orders`

Avoid special characters or multiple levels of nesting that become unwieldy.

Filtering, Sorting, and Pagination

Query Parameters

```
/products?sort=price&order=desc&limit=10&page=2
```

Standardize your query parameters across all endpoints

Pagination Strategies

1. Offset-Based Pagination (Simple but Can Be Slow)

Uses offset and limit to fetch a specific set of records.

URL: /users?offset=20&limit=10

This means: Skip the first 20 users, then return the next 10.

Pros: Easy to implement, works well for small datasets.

Cons: Slow for large datasets because skipping many records takes time.

Imagine flipping pages in a book. If you want page 100, you still have to turn every page before it!

2. Cursor-Based Pagination (Efficient for Large Data)

Uses a cursor (a unique ID from the last item) to fetch the next set.

URL: /users?cursor=abc123&limit=10

This means: "Start from user with ID abc123 and return the next 10 users."

Pros: Faster and more efficient for large datasets.

Cons: More complex to implement, requires sorting by a unique column (like ID).

Good vs. Bad URL

Good	Bad
`/users/123/orders`	`/getAllTheUsers`
`/products?category=electronics&limit=10&page=2`	`/deleteUser?id=123`
	`/api/controller.php?action=getProducts`

Key Point: Keep URLs consistent, predictable, and reflect domain logic accurately.

API Documentation Standards

OpenAPI

OpenAPI Specification (formerly Swagger Specification) is an API description format for REST APIs.

A specification that defines how to describe RESTful APIs in a machine-readable format (usually YAML/JSON)

Helps generate human-readable docs, interactive UIs, client libraries, and server stubs.

Benefits of OpenAPI

1. Generate a Server Stub – Automatically create backend server code.

   Tool: Swagger Codegen

2. Generate Client Libraries – Create client SDKs for multiple languages.

   Tool: Swagger Codegen

3. Interactive API Documentation – Provides an easy-to-use interface for API exploration.

   Tool: Swagger UI

4. Automated Testing – Simplifies API testing workflows.

   Tool: SoapUI

Why Use OpenAPI?

Standardized API Documentation

Enhances Developer Experience

Reduces Development Time

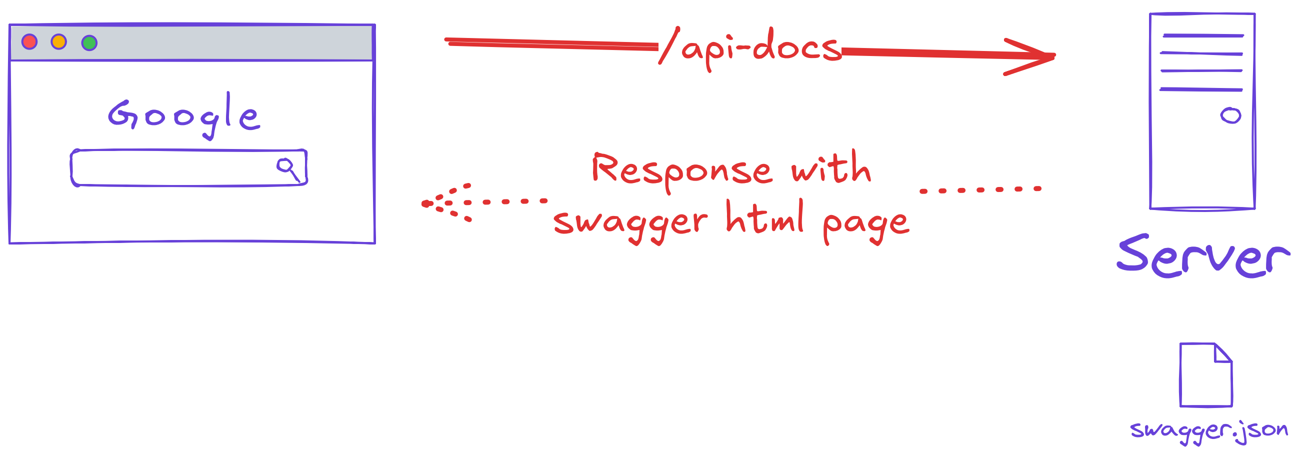
Supports Multiple Programming Languages

Facilitates Automated Testing & Validation

swagger.json Here is a sample swagger.json file

```json
{
"openapi": "3.0.0",
"info": {
"title": "User API",
"version": "1.0.0",
"description": "API for retrieving users"
},
"servers": [
{
"url": "http://localhost:3000",
"description": "Local server"
}
],
"paths": {
"/api/users": {
"get": {
"summary": "Get list of users",
"description": "Returns a list of users",
"responses": {
"200": {
"description": "A list of users",
"content": {
"application/json": {
"schema": {
"type": "array",
"items": {
"type": "object",
"properties": {
"id": { "type": "integer" },
"name": { "type": "string" }
}
}
}
}
}
}
}
}
}
}
}
```

swagger-ui-express
swagger-jsdoc



Request Validation

Request validation is the process of verifying that incoming request data conforms to predefined rules, formats, and constraints before being processed by an application. It helps ensure that only well-structured, expected, and safe data enters the system.

Key Aspects of Request Validation

1. Type Checking – Ensures values match expected data types (e.g., a string for names, integers for IDs).

2. Required Fields – Verifies that all necessary fields are present in the request.

3. Data Format – Ensures values follow the correct format (e.g., email, date, phone number).

4. Business Rules – Validates domain-specific constraints (e.g., age must be greater than 18).

Why is Request Validation Important?

1. Consistency :-

Ensures that the database or business logic only processes well-formed data.

Prevents unexpected errors due to invalid input.

2. Security :-

Protects against SQL injection, XSS, and other malicious attacks by rejecting invalid or harmful input.

Reduces the risk of attackers exploiting weaknesses in your API.

3. Better Client Experience :-

Provides instant feedback to users when they send incorrect or incomplete data.

Helps prevent unnecessary API failures and improves usability.

Validation Approaches

1. Schema-Based Validation

Uses JSON Schema or built-in validation frameworks (e.g., Joi, Zod & Express-Validator etc.)

Automatically enforces data structure, types, and constraints.

2. Custom Validation Middleware

Allows for custom business logic not covered by standard schema validation.

Useful when dealing with complex or conditional validations.

Response Optimization

Response Optimization is the process of refining how your server or API sends responses to clients. The goal is to :-

1. Improve performance
2. Reduce latency
3. Decrease server load and costs
4. Enhance the user experience

Key Techniques for Response Optimization

1. Pagination and Limits

Instead of sending all data at once, responses should include only a subset of records.

2. Data Compression

Compress response payloads using Gzip, Brotli, or Deflate to minimize data transfer size.

3. Caching

Store frequently accessed data to avoid unnecessary processing.

4. Optimized Serialization

Convert objects to efficient data formats before sending them to clients.

Reduce unnecessary fields in responses.

Use lightweight formats like MessagePack or Protocol Buffers instead of JSON where needed.

5. Efficient Data Structures

Choose optimized data structures to store and retrieve data faster.

Avoid sending nested or deeply linked objects when unnecessary.

Error Handling

Error handling in REST APIs is the process of capturing, processing, and communicating errors that occur during API requests.

A well-structured error handling mechanism ensures:

1. Better debugging
2. Improved developer experience
3. Clear and consistent error responses

Key Principles of Error Handling

1. Response Codes

HTTP status codes should accurately reflect the type of error.

Common error codes:

400 Bad Request 401 Unauthorized 404 Not Found

500 Internal Server Error 403 Forbidden → Access denied

2. Error Messages

Provide clear and human-readable error messages.

Avoid vague messages like "Something went wrong!"

Use standardized response structures.

3. Consistency

Maintain a consistent structure for error responses.

Ensure all endpoints follow the same format.

4. Do Not Leak Internal Details

Avoid exposing stack traces, database errors, or sensitive details in responses.

Bad practice (Exposes internal logic):

```
{
"error": "Database connection failed: Connection refused"
}
```

5. Logging and Monitoring

Capture errors using logging tools like Winston, Morgan, or Sentry.

Helps in troubleshooting and debugging API failures.