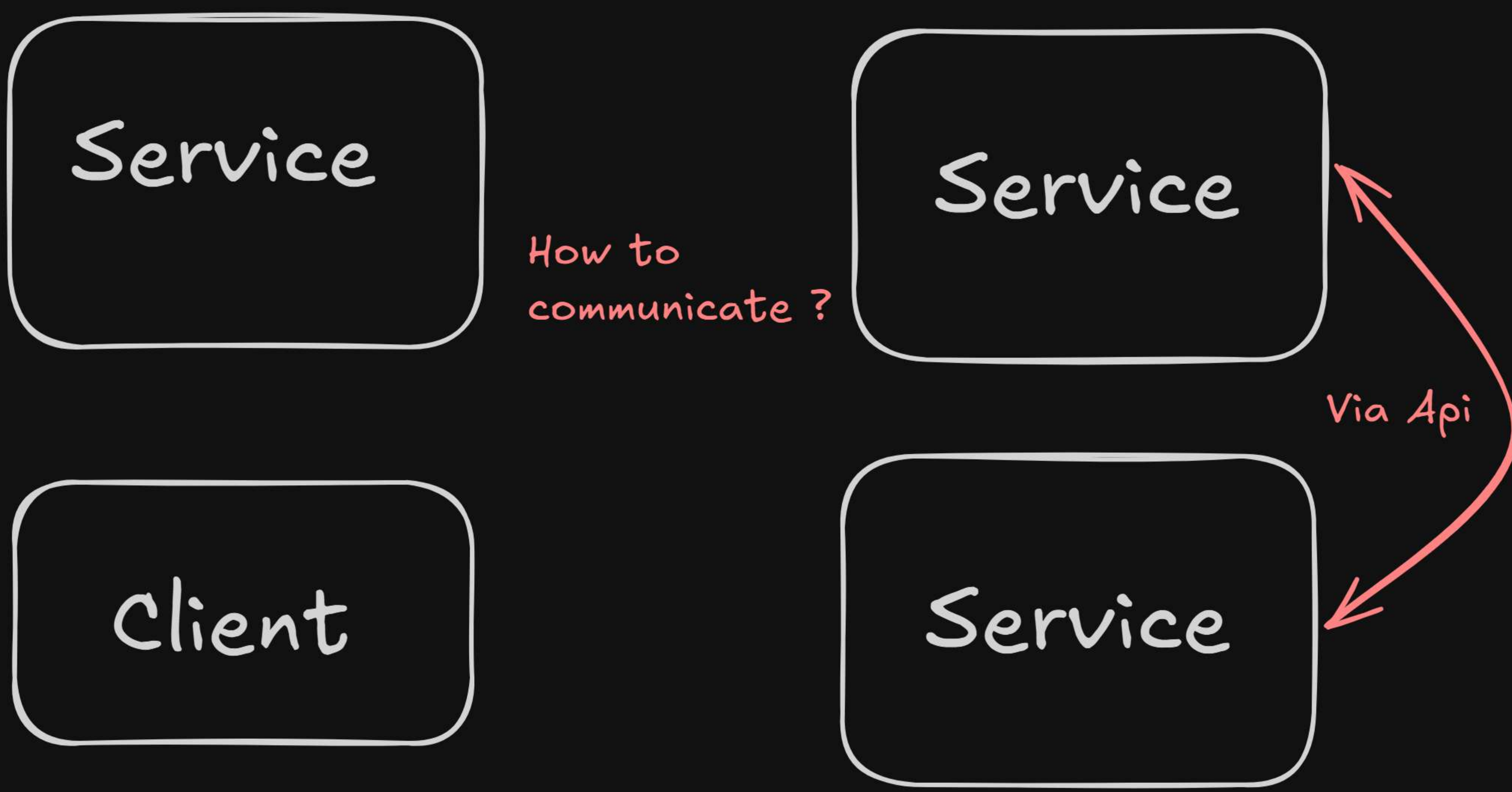


## Application Programming Interface (API)

- With APIs one software/system talk to another without exposing the internals like Waiter brings you the order you want in a restaurant.



### Swiggy/Zomato

When you search "Burger", the app calls an API → returns available items.

### Flight Booking (MakeMyTrip)

Airline APIs to fetch flight schedules, prices, and to book tickets.

### Weather App

Weather apps don't calculate weather → they hit a Weather API (like OpenWeatherMap).

## Types of APIs

REST

GraphQL

gRPC

SOAP (old-school)

## REST APIs

REST = Representational State Transfer.

It's an architectural style (not a protocol, not a library).

Uses HTTP for communication between client ↔ server.

using web URLs + HTTP methods to talk between services.

## Key Principles of REST

### Client-Server Separation

- Client (frontend, mobile app) and Server (backend) are independent.

### Statelessness

- Each API request is independent. Server doesn't remember past requests.

### Uniform Interface

APIs look consistent → HTTP methods used in standard ways.

GET → Fetch data

POST → Create data

PUT/PATCH → Update data

DELETE → Remove data

### Resource-Based URLs

- Everything is treated as a resource with a unique URL.

Example:

GET /users/123 → Get user with ID 123

POST /orders → Create new order

### Use of JSON/XML

REST usually exchanges data in JSON (lightweight).

## Data Returned

- response body
- status code

# Path Parameter

Part of the URL path itself → identifies a specific resource.

Example: `GET /users/123`

# Query Parameter

Key-value

Extra data added to the URL after a `?`.

Used for filtering, sorting, searching, pagination.

Example:

`GET /users?age=25&city=Delhi`

`age=25` and `city=Delhi` are query parameters.

# Response Body

The data server sends back (usually in JSON).

Example response for `GET /users/123`:

```
{
  "id": 123,
  "name": "Aniket",
  "city": "Delhi"
}
```

This is the actual "food" you get back from the kitchen (server).

# Status Code

- A number in HTTP response that tells you whether the request worked or failed.

Common ones:

200 OK → Success

201 Created → New resource created

400 Bad Request → Client error (wrong input)

401 Unauthorized → Need login/authentication

404 Not Found → Resource doesn't exist

500 Internal Server Error → Something broke in server

# DEMO

## When Should We Use REST APIs?

### 1. Client-Server Apps

Mobile app ↔ backend communication.

Example: Swiggy mobile app fetching restaurants via `GET /restaurants?city=Delhi`.

### 2. Web Applications

Frontend (React, Angular, Vue) ↔ backend (Node, Spring Boot, Django).

REST gives a clean way for frontend to fetch/update data.

### 3. Public APIs

When you want third-party devs to use your service.

Example:

Stripe Payments API

Google Maps API

Twitter API

REST is easy to understand + widely supported.

### 4. Microservices Communication

Different services in a system talk over REST.

Example: Order Service calls Inventory Service → `GET /inventory/123`.

### 5. CRUD Applications (Create, Read, Update, Delete)

REST fits perfectly when you're managing resources (users, posts, orders).

Standard HTTP verbs (GET, POST, PUT, DELETE) map directly to CRUD.

## When NOT to Use REST (and What to Use Instead)

### 1. Real-Time Communication Needed

Example: Chat apps, live updates → better with WebSockets or gRPC.

### 2. Over-fetching / Under-fetching Data

REST returns fixed structure → client may get too much or too little data.

Example: A mobile app needs only name and photo, but REST returns full user profile.

Better → GraphQL.

### 3. High-Performance Service-to-Service

REST uses text (JSON) → slower for very high-performance systems.

Better → gRPC (binary, faster).