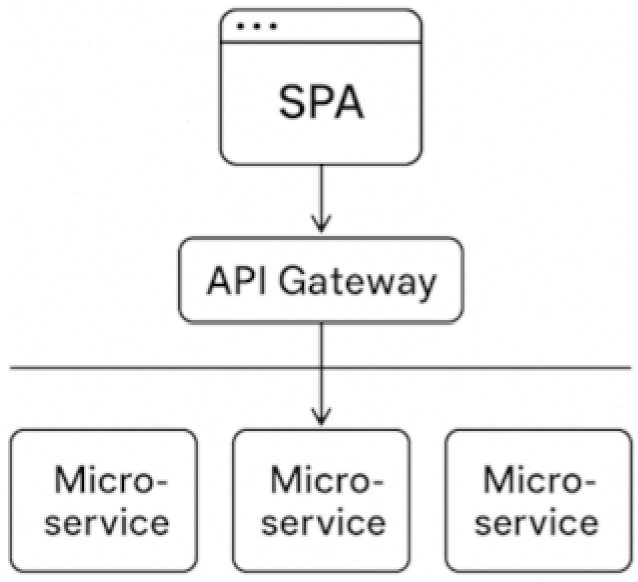
Modern web architecture :

- **SPA (Single Page Application)** built with React, Angular, Vue, Svelte, etc.
- **API Gateway** (Express.js, Kong, NGINX, or BFF).
- **Microservices** for each business domain (users, orders, payments, etc.)



SPA frameworks ?

> A **SPA framework** refers to a framework used to build **Single Page Applications (SPA)**. In a SPA, the entire application runs on a single HTML page and dynamically updates content without reloading the whole page. This leads to faster, smoother user experiences similar to desktop apps.

Key Characteristics of SPAs :

- Only one HTML page is loaded initially.
- Navigation and content updates happen dynamically via JavaScript.
- Often use AJAX or Fetch API to communicate with the backend.

- Uses client-side routing to change views without full-page reloads.

Microservices ?

> **Microservices** are an architectural style where a large application is broken down into many small, loosely coupled services.

Each service:

- Has its own specific function (e.g., user service, payment service, notification service).
- Often runs in its own container (like Docker).
- Communicates with other services via APIs (usually HTTP/REST, gRPC, or messaging systems like Kafka).

Layer	Frontend (client-side)	Backend (server-side)
Technology	Built with JavaScript frameworks (React, Angular, Vue)	Implemented in any language: Node.js, Java, Python, Go, etc.
Deployment artifact	Usually a single JS/CSS/HTML bundle served to the browser	Multiple independently built/deployed services (containers, etc.)
User interaction	Runs in the user’s browser; handles UI events, routing, rendering	Handles data, business logic, databases, authentication, etc.
Communication	Mostly calls backend APIs	May call other microservices over internal APIs

Frontend	React + TypeScript SPA
API Gateway	Express.js, Kong, NGINX, or BFF
Microservices	Node.js, Python, Java, Go services
Databases	PostgreSQL, MongoDB, Redis, etc.
Infrastructure	Docker, Kubernetes, Cloud

How To Approach:

For SPA framework :

1. **Understand the Tech Stack**
    - Use Wappalyzer (browser extension)
2. **Read the Client-side Code**
    - Use browser dev tools (Sources tab) or download source maps (`.js.map`)
    - Search for:
        - Hidden routes
        - Unused features
        - Hardcoded secrets (tokens, API keys)
        - Internal APIs or debug functions
        - Role-based logic (e.g., `if (user.isAdmin)`)
3. **Analyze API Traffic**
    - Use Burp Suite/ZAP/Postman to monitor and replay requests
    - Watch for:
        - Hidden or undocumented API endpoints
        - Insecure direct object references (IDORs)
        - Overly permissive CORS policies
        - Verb tampering (e.g., `GET` vs `POST`)
        - HTTP parameter pollution
4. **Test Authentication and Authorization**
    - Try:
        - Modifying JWT tokens
        - Removing or tampering with headers (Auth token, cookies)
        - Changing user IDs, roles, org IDs in requests
        - Accessing admin-only routes manually (`/admin`, `/users/all`, etc.)
5. **Look for Client-side Vulnerabilities**
    - Test for:
        - **DOM-based XSS** (`innerHTML`, `document.write`, client-side rendering)
        - **Open redirects** (`window.location`)
        - **CSRF** if the API uses cookies
        - **Clickjacking** on key views
6. **Explore the Routing System**
    - SPAs use client-side routing (like React Router).
    - Try:
        - Accessing restricted routes directly
        - Bypassing 403 pages by modifying client-side logic
        - Checking route guards (they can be bypassed in JS)

For Microservices :

Targets when hunting bugs in microservices :

- **API endpoints**: Are they properly authenticated and authorized?
- **Internal APIs**: Sometimes developers assume “internal means safe” → often vulnerable.
- **Service-to-service communication**: Tokens, keys, or secrets might be exposed.
- **Configuration files**: Misconfigured YAML, environment variables, or docker-compose files.
- **CI/CD pipelines**: Can lead to source leaks or deployment flaws.
- **Container security**: Insecure images, outdated dependencies, or excessive privileges.

Tips for bug hunters:

- Map out the architecture: understand what each microservice does and how they communicate.
- Look for differences between external APIs (for users) and internal APIs (for other services).
- Test for classic web bugs (XSS, SQLi) in every microservice.
- Pay special attention to broken access control and IDOR (Insecure Direct Object Reference).
- Scan containers and Kubernetes setups for misconfigurations.