**Rest Apis:**

Representational state transfer 🡪 Application programming interface (universal translator)

Direct interaction with a server involves managing low-level details like network protocols, sockets, and server configurations. REST APIs abstract these complexities, offering a structured way to communicate with servers. Let's break down why REST APIs are needed and what configurations would be necessary for direct interaction without them.

**Why We Need REST APIs:**

1. **Simplified Communication**:
   * REST APIs use HTTP methods (GET, POST, PUT, DELETE) to define clear operations.
   * They abstract the complexities of socket programming and raw protocol handling.
2. **Consistency**:
   * APIs define a consistent interface to interact with the server, making integration predictable and maintainable.
3. **Cross-Platform Compatibility**:
   * REST APIs work over HTTP/HTTPS, which is universally supported, enabling communication between diverse platforms (e.g., mobile apps, web apps).
4. **Security**:
   * APIs often include built-in security layers like authentication and authorization (e.g., API keys, OAuth).
   * They protect servers from direct, unauthorized access.
5. **Scalability**:
   * APIs provide a layer that allows load balancing, caching, and other optimizations without affecting client applications.

**Configurations Needed for Direct Interaction:**

If you bypass REST APIs, you need to manage the following manually:

1. **Network Details**:
   * **IP Address**: The server's public or private IP address.
   * **Port Number**: The specific port the server is listening on (e.g., 80 for HTTP, 443 for HTTPS, or custom ports).
2. **Protocol**:
   * Define whether you'll use HTTP, WebSocket, FTP, or other communication protocols.
   * For HTTP/HTTPS, you need to structure your requests (e.g., headers, body, etc.) manually.
3. **Authentication**:
   * Directly implement mechanisms like Basic Auth, Token-based Auth, or SSL certificates.
4. **Error Handling**:
   * Define how you'll handle timeouts, connection errors, and server responses.
5. **Data Serialization**:
   * Serialize data into the required format (e.g., JSON, XML) and deserialize server responses.
6. **Concurrency and Threading**:
   * Handle simultaneous requests using threads or async programming.
7. **Caching**:
   * Manually implement caching mechanisms for repeated data requests.
8. **Security Configurations**:
   * Manage SSL/TLS certificates for secure communication.
   * Protect the server from direct access by unauthorized users.
9. **State Management**:
   * Handle session or token management for stateful interactions.
10. **Firewall and Networking Rules**:
    * Configure firewalls to allow specific IPs and ports.
    * Define rules to prevent malicious attacks (e.g., DDoS protection).

**Challenges of Direct Interaction:**

1. **Complexity**:
   * Writing low-level networking code is time-consuming and error-prone.
2. **Security Risks**:
   * Directly exposing server ports and endpoints increases vulnerability.
3. **Maintenance Overhead**:
   * Every client must implement these configurations, leading to inconsistency and maintenance challenges.

**REST API Example:**

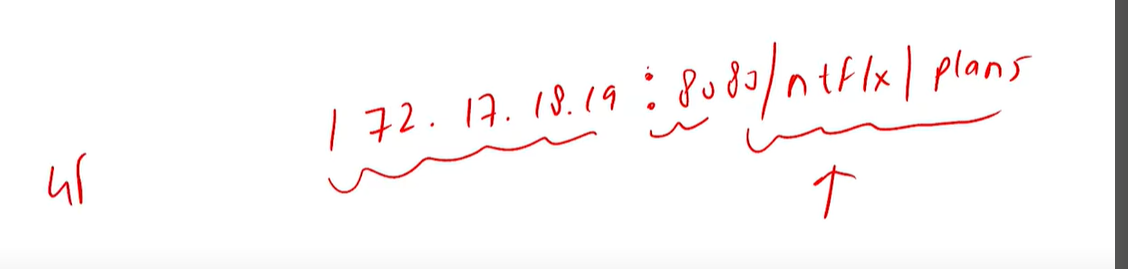
With a REST API, you can simply send an HTTP request:

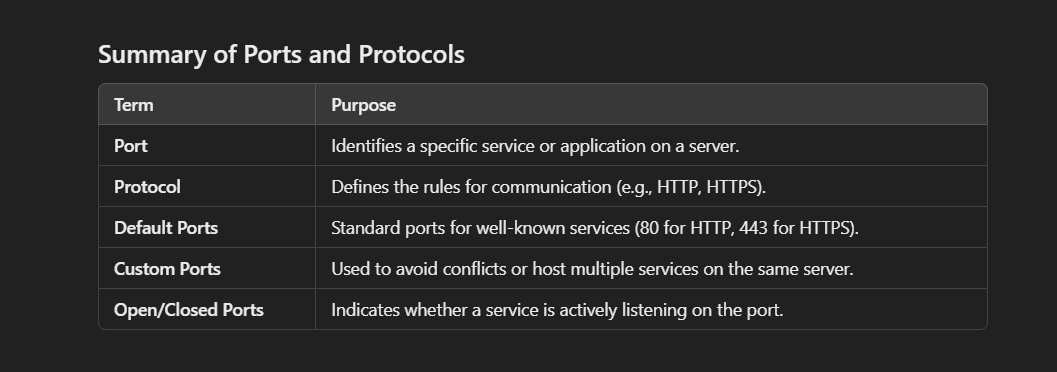
GET https://example.com/api/v1/users

Authorization: Bearer <token>

Instead of configuring sockets, ports, and protocols, you focus on the request's content and structure.

We need to access the REST API using URL + HTTP Verb 🡪 GET, POST, PUT, DELETE





**Controller Class:**

Special Type of Classes, Special Type of Components

localhost:8080/health-check

**Post using Body (postman)**

