Mastering System Design

Section 3: Protocols

Networking & Communication

Section Agenda

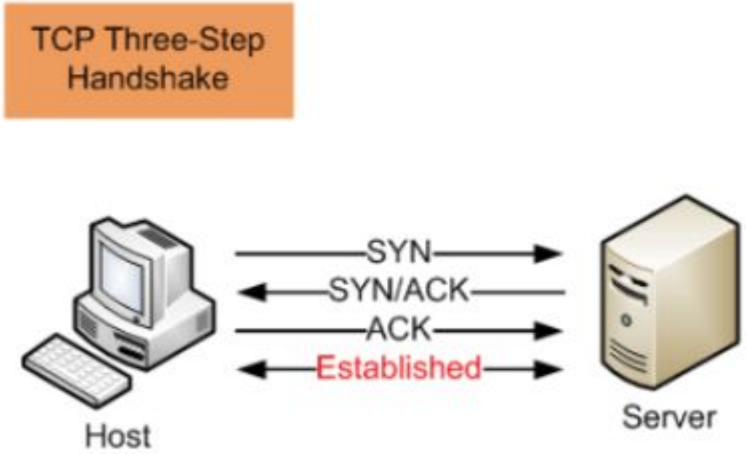
1. Lecture 1: TCP & UDP - Understanding the Basics
2. Lecture 2: HTTP - The Backbone of the Web
3. Lecture 3: REST & RESTfulness - API Design Principles
4. Lecture 4: Real-Time Communication Protocols
5. Lecture 5: Modern API Protocols - Beyond REST
6. Lecture 6: Summary & Practical Applications

TCP & UDP

Protocols

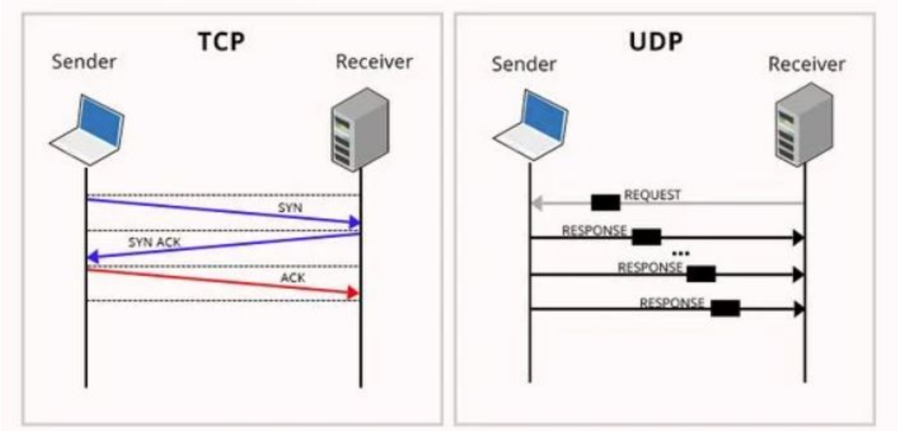
What is Transmission Control Protocol (TCP)?

- Connection-oriented protocol
- Reliable, ordered, and error-checked communication
- Ensures data reaches the destination correctly



What is User Datagram Protocol (UDP)?

- Connectionless protocol
- Faster, but no delivery guarantees
- No retransmission of lost packets



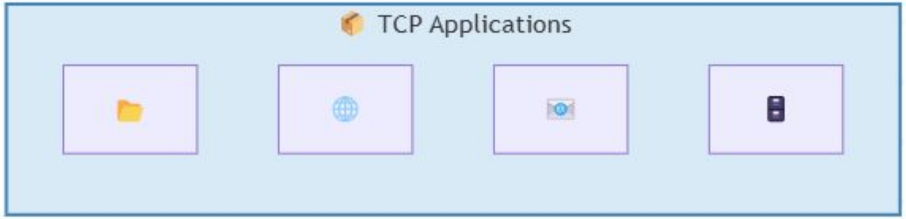
Key Differences Between TCP & UDP

- Reliability: TCP (Yes), UDP (No)
- Speed: TCP (Slower), UDP (Faster)
- Connection Type: TCP (Connection-oriented), UDP (Connectionless)

Feature	TCP (Transmission Control Protocol)	UDP (User Datagram Protocol)
Reliability	✅ Reliable (ensures data delivery)	❌ Unreliable (no guarantee of delivery)
Speed	⏳ Slower (due to error checking & retransmission)	⚡ Faster (no retransmission overhead)
Connection Type	🔗 Connection-oriented (establishes a connection before communication)	🔄 Connectionless (sends data without setup)
Ordering	✅ Ensures packets arrive in order	❌ No guarantee of packet order
Error Handling	✅ Built-in error checking & retransmission	❌ Minimal error checking, no retransmission
Overhead	📦 High (due to handshaking, sequencing, and acknowledgments)	🏗️ Low (minimal protocol overhead)
Use Cases	🌐 Web browsing (HTTP/HTTPS), File transfers (FTP, SFTP), Email (SMTP, IMAP, POP3), Database communication	🎥 Video streaming (YouTube, Netflix), 🎮 Online gaming, 📞 VoIP calls (Skype, Zoom), 🖧 DNS lookups

Use Cases: When to Use TCP vs. UDP

- **When to use TCP:** If data integrity is critical, TCP is the way to go.
  - Web browsing (HTTP, HTTPS)
  - File transfers (FTP, SFTP)
  - Email (SMTP, IMAP, POP3)
  - Database communication
- **When to use UDP:** UDP is preferred when speed is more important than reliability
  - Video streaming (YouTube, Netflix)
  - Online gaming (Multiplayer Games)
  - VoIP calls (Skype, Zoom)
  - DNS lookups (Domain Name System)



Interview Questions: TCP & UDP

- Basic Questions:
  - What is the difference between TCP and UDP?
  - Why is TCP considered a reliable protocol?
  - When would you choose UDP over TCP?
- Technical & Deep-Dive:
  - How does TCP ensure reliable data transmission?
  - What is the Three-Way Handshake in TCP?
  - How does UDP handle packet loss?
- Use Case & System Design:
  - Which protocol would you use for a real-time multiplayer game? Why?
  - How does TCP handle congestion control?
  - Why is DNS implemented over UDP instead of TCP?
  - If a video streaming service is buffering too much, would switching from TCP to UDP help?

Pro Tip: Be prepared to compare TCP & UDP in real-world scenarios and discuss trade-offs in speed vs. reliability!

Summary & Key Takeaways

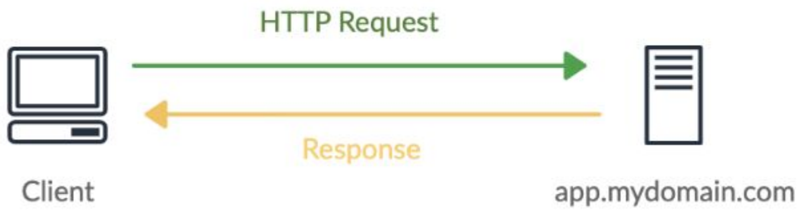
- TCP is reliable but slower; UDP is fast but unreliable.
- TCP is best for web pages, file transfers, and email.
- UDP is best for real-time applications like video calls and gaming.
- System design must balance speed, reliability, and user experience.
- What's Next:
    - HTTP - The Backbone of the Web

HTTP - The Backbone of the Web

Protocols

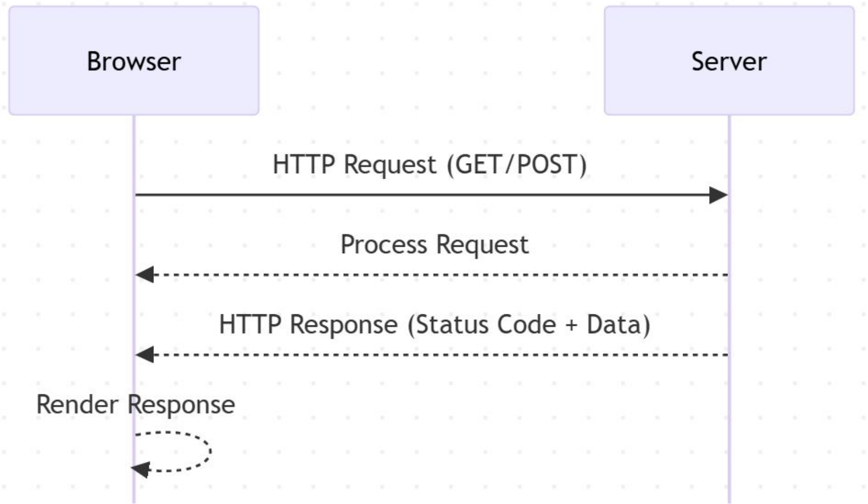
Introduction to HTTP

- What is HTTP?
  - Stands for HyperText Transfer Protocol
  - The foundation of web communication
  - Defines rules for requesting and transferring resources (e.g., web pages, APIs, images)
  - Works over TCP/IP (Port 80 for HTTP, Port 443 for HTTPS)
- Key Features:
  - Text-based protocol (easy to read & debug)
  - Stateless (each request is independent)
  - Supports multiple methods (GET, POST, etc.)



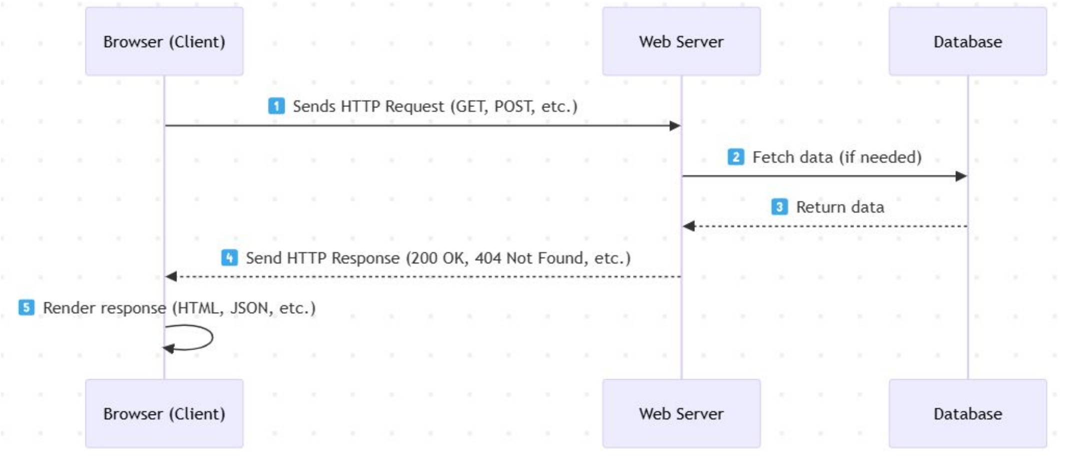
How HTTP Works

- Client-Server Model:
  - Client (browser or mobile app) makes an HTTP request
  - Server (web server, API, etc.) processes the request and sends back a response
- Components of an HTTP Request:
  - Method: Defines the action (GET, POST, etc.)
  - URL: The resource being requested
  - Headers: Metadata (e.g., user-agent, content type)
  - Body (optional): Data sent in POST/PUT requests
- Components of an HTTP Response:
  - Status Code: Indicates success or failure (e.g., 200 OK, 404 Not Found)
  - Headers: Metadata about the response
  - Body (optional): The actual content returned



The HTTP Request-Response Cycle

1. Step 1: The browser (client) sends a request
2. Step 2: The web server processes the request
3. Step 3: The server generates a response and sends it back
4. Step 4: The browser renders the response (for a web page)



Stateless Nature of HTTP

- What does “stateless” mean?
  - HTTP does not retain memory of previous requests
  - Each request is treated as an independent transaction
- Challenges of Statelessness:
  - Hard to maintain user sessions (e.g., login state)
  - Each request must carry all necessary information
- How do we handle state?
  - Cookies – Small pieces of data stored in the browser
  - Sessions – Server-side storage of user state
  - Tokens (JWT, OAuth) – Used for authentication & authorization

HTTP Methods

- Common HTTP Methods & Their Use Cases:
  - GET: Retrieve a resource (e.g., webpage, API data)
  - POST: Send data to create a new resource (e.g., form submission)
  - PUT: Update an existing resource
  - DELETE: Remove a resource
  - PATCH: Partially update a resource

HTTP Status Codes

- 1xx – Informational (Request received, continuing process)
- 2xx – Success (Request successfully processed)
  - 200 OK – Successful response
  - 201 Created – Resource successfully created
- 3xx – Redirection (Further action needed)
  - 301 Moved Permanently – Resource URL changed
  - 304 Not Modified – Use cached version
- 4xx – Client Errors (Mistakes in the request)
  - 400 Bad Request – Incorrect request format
  - 401 Unauthorized – Authentication required
  - 403 Forbidden – No permission
  - 404 Not Found – Resource doesn't exist
- 5xx – Server Errors (Issue with the server)
  - 500 Internal Server Error – Unexpected server failure
  - 503 Service Unavailable – Server overloaded

What About HTTPS?

- HTTPS is the secure version of HTTP, using SSL/TLS encryption.
- It ensures data confidentiality, integrity, and authentication.
- Used for secure websites, online banking, and e-commerce.
- Works on Port 443 instead of Port 80.

HTTP Interview Questions

- Basic Questions:
  - What is HTTP, and how does it work?
  - Why is HTTP considered a stateless protocol?
  - What are the key differences between HTTP and HTTPS?
- Intermediate Questions:
  - Explain the HTTP request-response cycle with an example.
  - What are HTTP methods? When would you use PUT vs. PATCH?
  - What are HTTP status codes? Give examples of 2xx, 3xx, 4xx, and 5xx status codes.
- Advanced Questions:
  - How do cookies, sessions, and tokens help maintain state in HTTP?
  - What is the difference between 301 (Moved Permanently) and 302 (Found) redirections?
  - How does caching work in HTTP, and which headers control it?
  - What security risks are associated with HTTP, and how can they be mitigated?

Summary & Key Takeaways

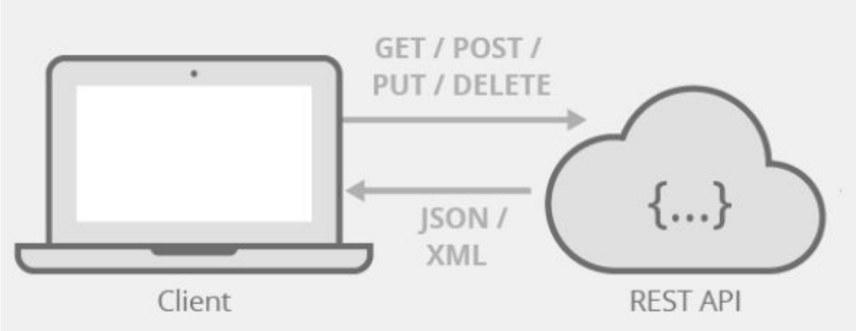
- HTTP is the foundation of web communication
- It follows a request-response model
- HTTP is stateless, requiring mechanisms like cookies & sessions
- HTTP methods define actions, and status codes indicate responses
- What's Next:
    - REST & RESTfulness - API Design Principles

REST & RESTfulness - API Design Principles

Protocols

What is REST?

- **Definition**: REST (Representational State Transfer) is an architectural style for designing networked applications.
- **Key Idea**: Uses standard HTTP methods and stateless communication.
- **Origin**: Coined by Roy Fielding in his 2000 dissertation.

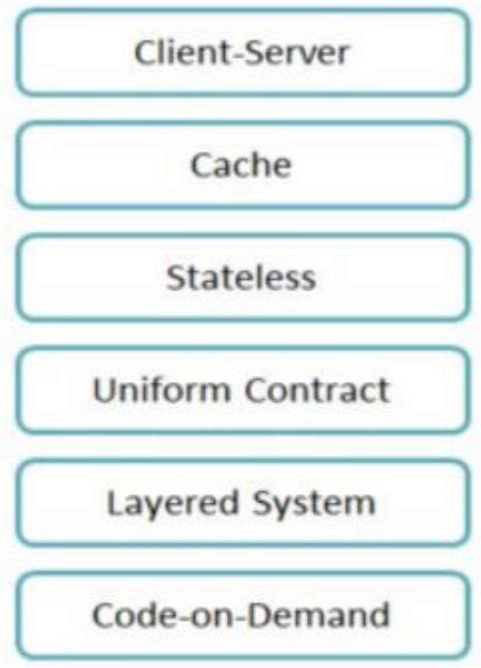


Why REST Matters?

- **Simplicity & Scalability**: Based on standard HTTP.
- **Interoperability**: Works across different platforms.
- **Efficiency**: Uses caching, statelessness for performance.

REST Constraints (Core Principles)

- Client-Server Architecture
- Statelessness
- Cacheability
- Layered System
- Uniform Interface



RESTful API Design Principles

- Resource-Based Approach
  - `GET /users/{id}` to retrieve a user
  - `POST /orders` to create a new order
- Proper HTTP Methods Usage
  - `GET` to **retrieve** data
  - `POST` to **create** new resources
  - `PUT` to **update** existing resources
  - `PATCH` for **partial updates**
  - `DELETE` to **remove** resources
- Stateless Interactions
- Consistency in URL Structure
  - ✅ Use **plural nouns** for collections: `/users`, `/orders`
  - ✅ Avoid including actions in URLs: `/users/{id}/activate` ❌ → `/users/{id}` with `PATCH` ✅
  - ✅ Implement **versioning** for backward compatibility: `/v1/users`

Resources & Endpoints

- Resources: Entities like Users, Orders, Products.
- Endpoint Examples:
  - GET /users/{id}
  - POST /orders
  - DELETE /products/{id}

`GET https://example.com/products` → Get all products

`GET https://example.com/products/{id}` → Get details of a specific product

`POST https://example.com/products` → Add a new product

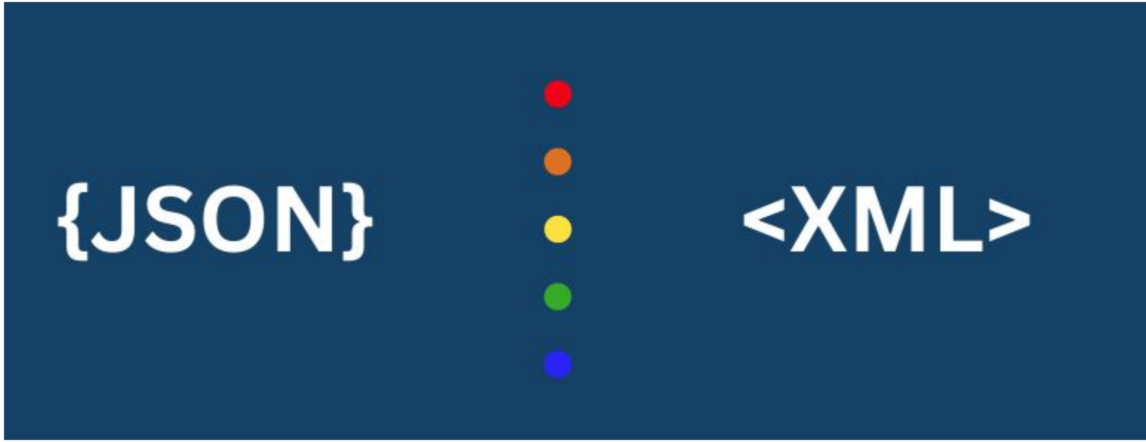
`PUT https://example.com/products/{id}` → Update a product

`PATCH https://example.com/products/{id}` → Partially update a product

`DELETE https://example.com/products/{id}` → Remove a product

JSON vs. XML in REST APIs

- Why JSON?
  - Lightweight
  - Faster parsing
  - Readable
- When to Use XML?
  - Legacy systems
  - Data validation needs



HTTP Methods in REST

- **GET**: Retrieve data
- **POST**: Create a resource
- **PUT**: Update a resource
- **PATCH**: Partial update
- **DELETE**: Remove a resource
- **OPTIONS**, **HEAD**: Additional functionalities

`GET https://example.com/orders` → Fetch all orders

`GET https://example.com/orders/{id}` → Fetch a specific order

`POST https://example.com/orders` → Create a new order

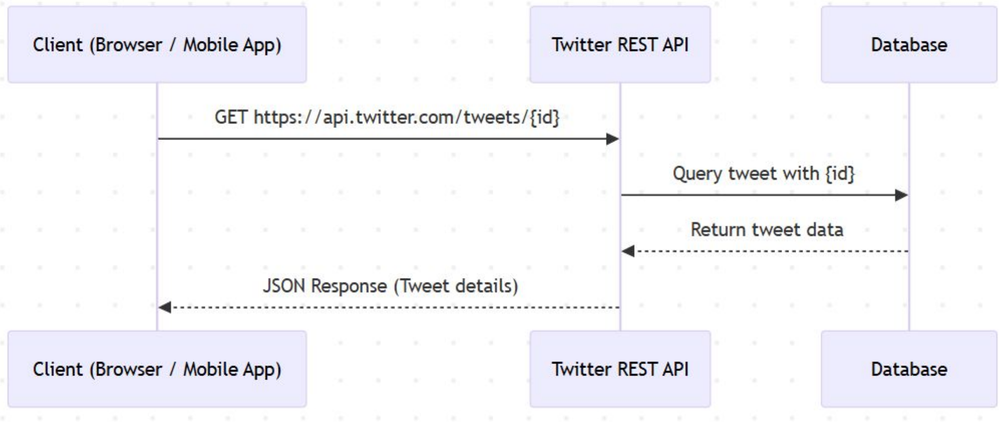
`PUT https://example.com/orders/{id}` → Update an order

`PATCH https://example.com/orders/{id}` → Partially update an order

`DELETE https://example.com/orders/{id}` → Cancel an order

Real-World REST API Examples

- Twitter API
  - Fetch tweets: GET /tweets/{id}
  - Post a tweet: POST /tweets
- GitHub API
  - Get repo details: GET /repos/{owner}/{repo}
  - Create an issue: POST /repos/{owner}/{repo}/issues



Best Practices & Common Pitfalls

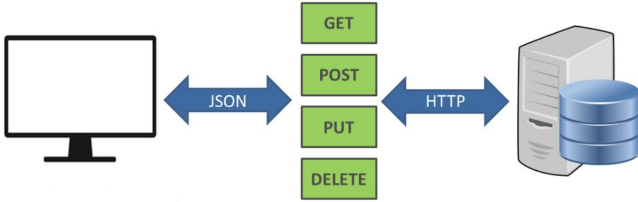
✅ Use Proper Status Codes (200, 201, 400, 404, 500)

✅ Versioning APIs (/v1/resource)

✅ Implement Authentication (OAuth, JWT)

✅ Pagination (?page=2&limit=20)

🚫 Avoid using Verbs in URLs (/createUser → POST /users)



Interview Questions - REST

- Basic Questions
    - What is REST, and how does it differ from SOAP?
    - What are the six constraints of REST architecture?
    - What is the difference between a REST API and a RESTful API?
    - What is a resource in REST, and how is it represented?
    - What are endpoints in a REST API?
- TP Methods & Status Codes
    - Explain the difference between GET, POST, PUT, PATCH, and DELETE.
    - When would you use PUT vs. PATCH?
    - What are the commonly used HTTP status codes in REST APIs?
    - What does the 200 OK, 201 Created, 204 No Content, 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, and 500 Internal Server Error mean?
- RESTful API Design & Best Practices
    - What are the best practices for designing RESTful APIs?
    - How do you design a RESTful API for a blogging platform (e.g., users, posts, comments)?
    - What is HATEOAS (Hypermedia as the Engine of Application State) in REST?
    - How do you handle authentication and authorization in REST APIs?
    - What are API rate limiting and throttling, and why are they important?
    - How do you ensure backward compatibility when updating a REST API?
- Advanced & Real-World Questions
    - How does caching work in RESTful APIs?
    - What is the difference between JSON and XML in REST?
    - How do you implement pagination in REST APIs?
    - How does versioning work in REST APIs?
    - Explain the differences between REST, GraphQL, and gRPC.
    - How would you improve the performance of a REST API?
    - Can a REST API be stateful? Why or why not?
    - How does REST handle security vulnerabilities like SQL injection and CSRF?

Summary & Key Takeaways

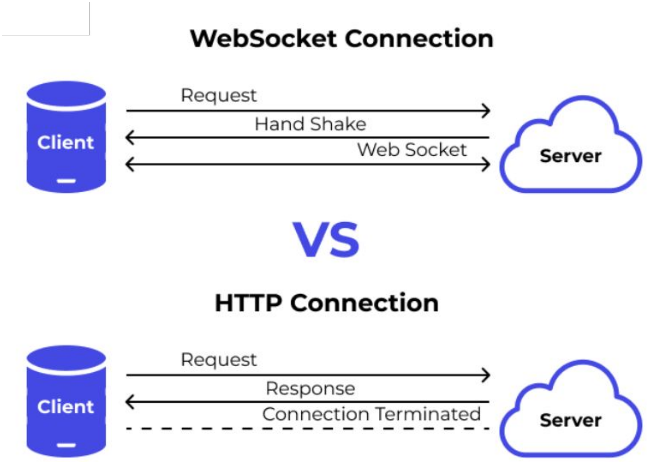
- REST is a stateless, scalable, and widely adopted API design style.
- Proper endpoint structure, HTTP methods, and response formats matter.
- Security, versioning, and best practices improve API quality.
- What's next:
  - Real-Time Communication Protocols

Real-Time Communication Protocols

Protocols

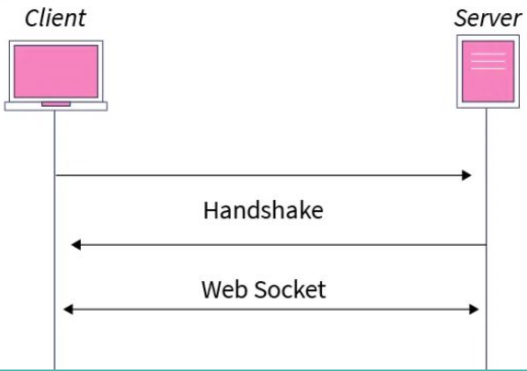
Introduction to Real-Time Communication

- **Definition**: Real-time communication refers to the continuous exchange of data with minimal latency.
- Why real-time is important (e.g., instant chat, live stock updates, multiplayer games).
- Challenges of traditional request-response HTTP model.
- Traditional HTTP vs. Real-Time Communication
- Alternatives to improve real-time data exchange:
    - Polling
    - WebSockets
    - Server-Sent Events (SSE)
    - Long Polling



WebSockets: Persistent Full-Duplex Communication

- Definition: WebSockets provide a persistent, full-duplex connection between the client and server over a single TCP connection.
- How they work:
  - WebSocket handshake using HTTP upgrade request.
    - **Step 1:** Client requests an upgrade to WebSockets
    - **Step 2:** Server accepts and keeps the connection open.
  - Connection remains open for continuous data exchange.
  - Either client or server can send messages at any time.
    - **Step 3:** Data is exchanged in real-time using frames.
    - **Step 4:** Either party can close the connection when done.

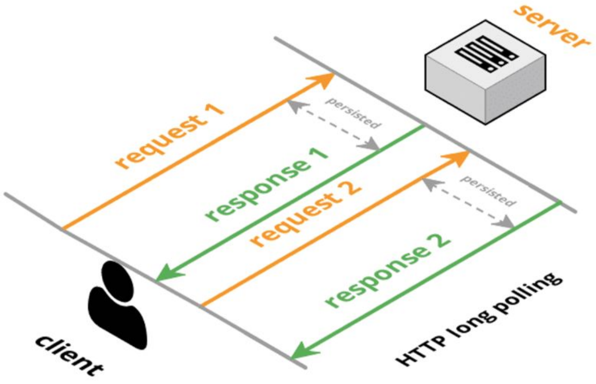


Advantages and use cases of WebSockets

- Advantages:
  - Persistent connection = lower latency.
  - Reduces overhead compared to HTTP polling.
  - Efficient for real-time applications.
- Use cases:
  - Live Chat Applications (WhatsApp, Slack, Discord)
  - Stock Market Price Updates (NASDAQ, Robinhood)
  - Multiplayer Online Games (Fortnite, Call of Duty)
  - Collaborative Tools (Google Docs, Figma)

Long Polling: Simulating Real-Time with HTTP

- **Definition**: A technique where the client sends a request to the server and waits until the server has new data to respond with.
- How it differs from regular polling:
- Instead of immediate responses, the server holds the request until new data is available.
- How Long Polling Works (Step-by-Step)
  a. Client makes an HTTP request.
  b. Server holds the request until data is available.
  c. Server responds with new data.
  d. Client immediately sends another request.



When to Use WebSockets vs. Long Polling?

- Use WebSockets when:
  - ✅ High-frequency, bi-directional data exchange is needed.
  - ✅ Low latency is critical (e.g., gaming, chat).
- Use Long Polling when:
  - ✅ WebSockets are not supported or overkill.
  - ✅ Periodic updates are sufficient (e.g., notifications).
- Real-world example:
  - Slack: WebSockets for chat.
  - Twitter: Long Polling for notifications.
  - Stock Exchanges: WebSockets for real-time data feeds.
  - IoT devices: Long Polling for intermittent updates.

Interview Questions

- Basic Questions
  - What is real-time communication, and why is it important?
  - How does WebSockets work, and how does it differ from traditional HTTP?
  - Explain the WebSocket handshake process.
  - What is long polling, and how does it work?
- Intermediate Questions
  - What are the advantages of WebSockets over long polling?
  - In what scenarios would you prefer long polling over WebSockets?
  - How does WebSockets handle connection failures or network interruptions?
  - Can you use WebSockets with load balancers? If yes, how?
  - What are some challenges of scaling WebSockets in a distributed system?

Summary & Final Takeaways

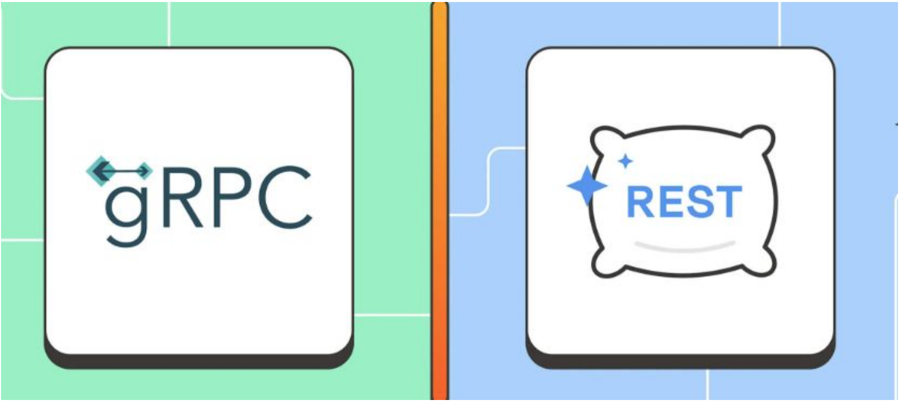
- WebSockets = Persistent, full-duplex communication.
- Long Polling = Simulated real-time via HTTP requests.
- Choose based on latency needs and infrastructure.
- What’s next:
    - Modern API Protocols - Beyond REST(gRPC, GraphQL)

Modern API Protocols - Beyond REST(gRPC, GraphQL)

Protocols

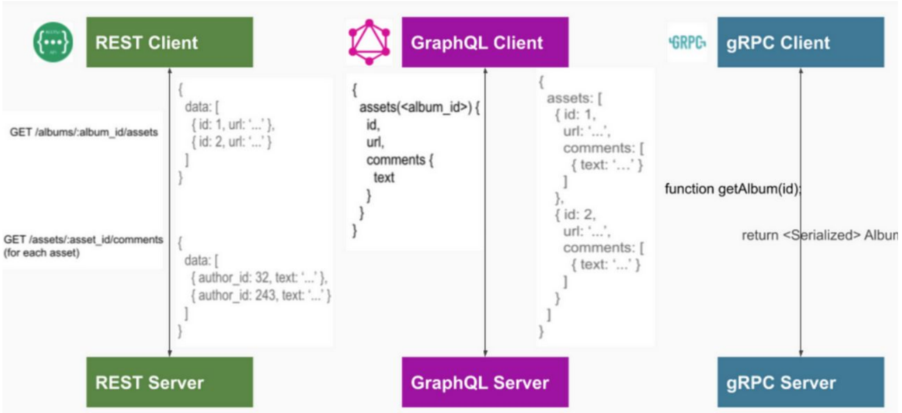
Why Do We Need More Than REST?

- Limitations of REST APIs:
  - Over-fetching & Under-fetching (Clients get too much or too little data).
  - High Latency (Multiple requests needed for complex data).
  - Not optimized for real-time communication (Polling required).
- Need for modern solutions like gRPC & GraphQL to handle efficiency, flexibility, and performance.



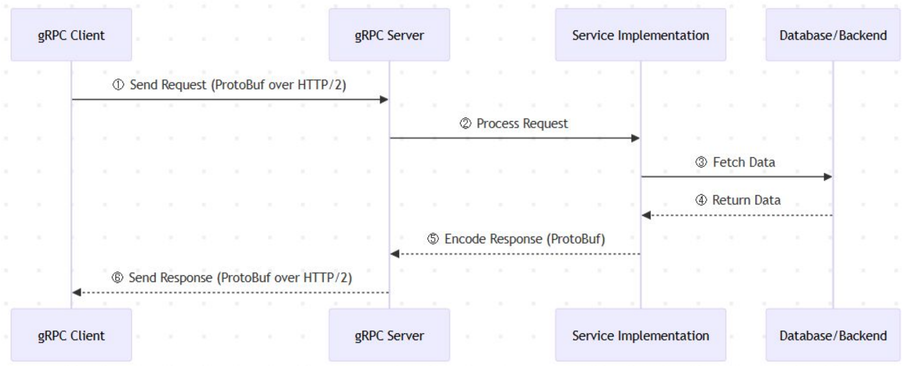
Introducing gRPC & GraphQL

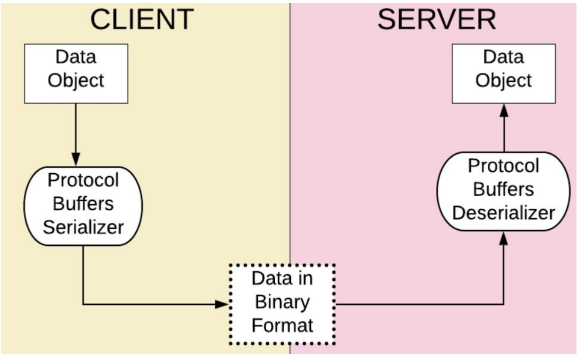
- **gRPC**: A high-performance, binary protocol optimized for microservices & real-time communication.
- **GraphQL**: A flexible query language that allows clients to fetch only the data they need.
- Key Differences:
  - gRPC = Fast, binary, microservices-focused
  - GraphQL = Flexible, JSON-based, frontend-optimized



gRPC - How It Works?

- gRPC is built on HTTP/2, allowing:
  - Multiplexed requests (multiple calls over one connection).
  - Compression (smaller payload sizes).
  - Full-duplex streaming (real-time bidirectional communication).
- Uses Protocol Buffers (ProtoBuf) for fast serialization (smaller & faster than JSON).



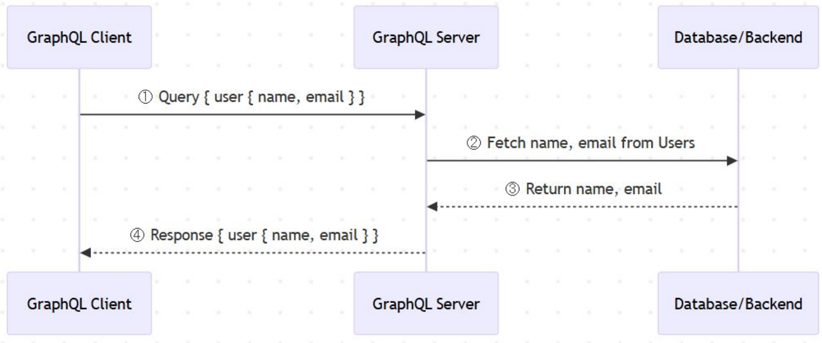


gRPC - Use Cases & When to Use

- Microservices – Fast inter-service communication.
- Real-time streaming – Full-duplex bidirectional data transfer.
- IoT & Low-bandwidth applications – Efficient binary communication.
- Multi-language ecosystems – Auto-generated client & server code.

GraphQL - How It Works?

- Instead of multiple REST endpoints, GraphQL has one endpoint where clients specify the data they need.
- GraphQL Schema defines types & relationships between data.
- Clients send queries → GraphQL server resolves fields dynamically.




```
query {
  user(id: 123) {
    name
    email
  }
}
```

```
{
  "data": {
    "user": {
      "name": "John Doe",
      "email": "john.doe@example.com"
    }
  }
}
```

GraphQL - Use Cases & When to Use

- Frontend Optimization – Clients fetch exactly what they need.
- Reducing API Requests – One query replaces multiple REST calls.
- Mobile & Web Apps – Handles slow networks & multiple data sources.
- Aggregating Data from Multiple Services – Simplifies fetching data from different databases or APIs.

Interview Questions on API Protocols

- How would you compare REST, gRPC, and GraphQL?
- When would you use gRPC over REST?
- What are the trade-offs of using GraphQL in a large-scale system?
- How does gRPC handle authentication & security?
- How do you scale GraphQL APIs efficiently?

Summary & Key Takeaways

- gRPC → Best for microservices, real-time streaming, high-performance APIs.
- GraphQL → Best for frontend-driven APIs, flexible data fetching, and reducing over-fetching.
- Choose based on use case – No one-size-fits-all!
- Be ready to justify your API choices in system design interviews!
- What's next:
    - Summary & Practical Applications of protocols

Summary - Protocols

- In this section, we explored key communication and API design protocols essential for system design:
    - TCP & UDP - Understanding the Basics
    - HTTP - The Backbone of the Web
    - REST & RESTfulness - API Design Principles
    - Real-Time Communication Protocols
    - Modern API Protocols - Beyond REST
- Next Up:
    - Architectural Patterns
    - We now shift focus to architectural patterns, which define how components interact in scalable systems.