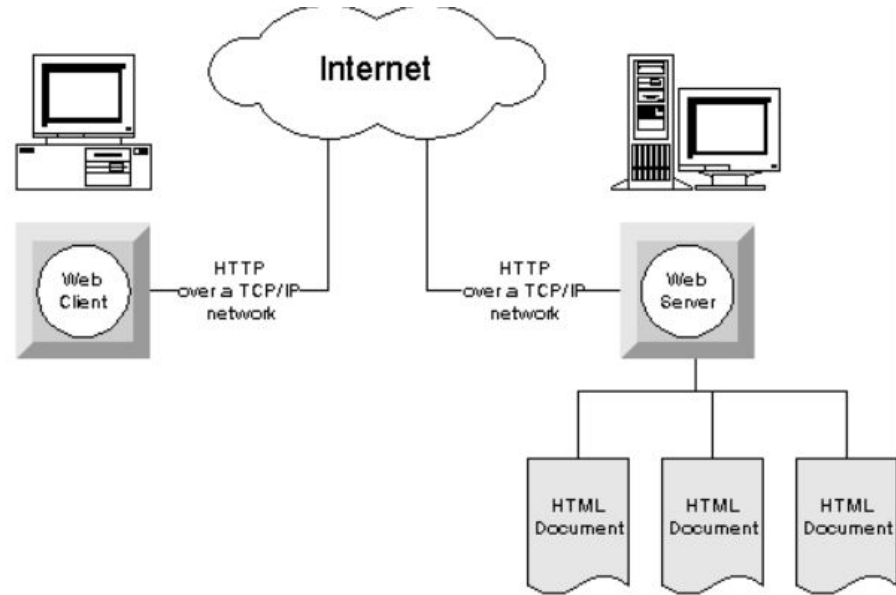

Mastering System Design

Section 5: Web Concepts in System Design

— System Design Fundamentals —

Why Learn Web Concepts?

- Importance of web principles in modern system design
- Impact on scalability, security, and performance
- How understanding these concepts helps in system design interviews



Section Agenda

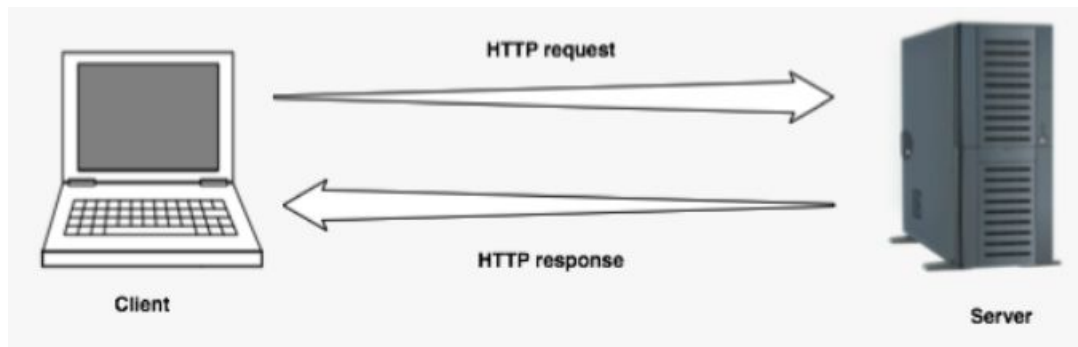
1. Web Sessions: Managing State in Web Applications
2. Serialization: Data Exchange & Storage Formats
3. CORS: Cross-Origin Resource Sharing & Web Security
4. Summary & Practical Applications

Web Sessions: Managing State in Web Applications

System Design Fundamentals - Web Concepts

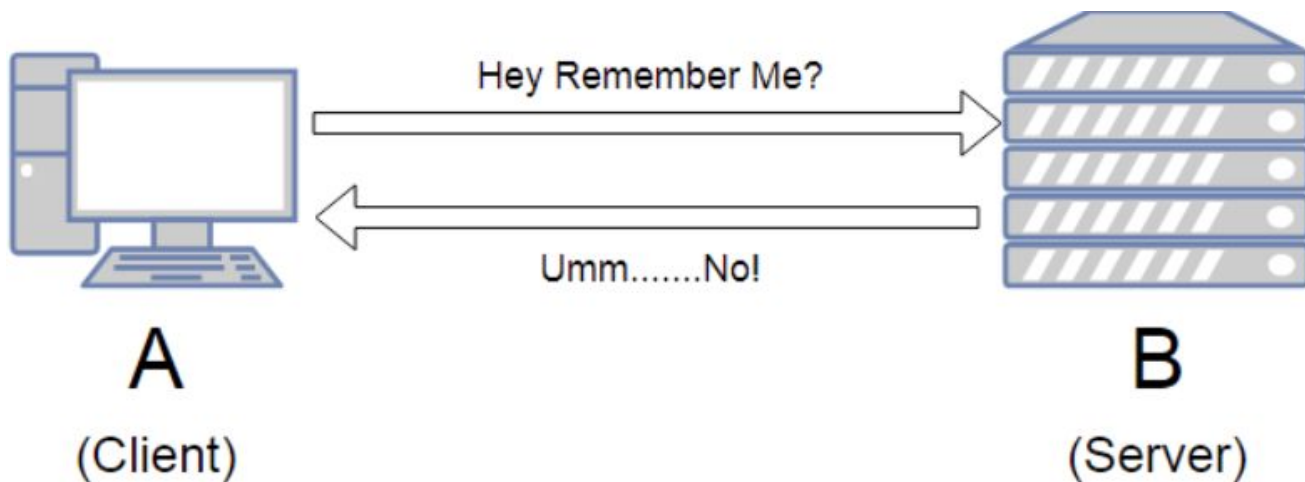
Why Web Sessions Matter?

- Web applications often need to track user state (e.g., login status, shopping cart, user preferences)
- HTTP is stateless, meaning each request is independent
- Goal: Understand how to maintain state in web applications



Understanding Statelessness in HTTP

- HTTP does not retain memory of previous requests
- Each request must contain all necessary information
- Why this is a challenge for user sessions

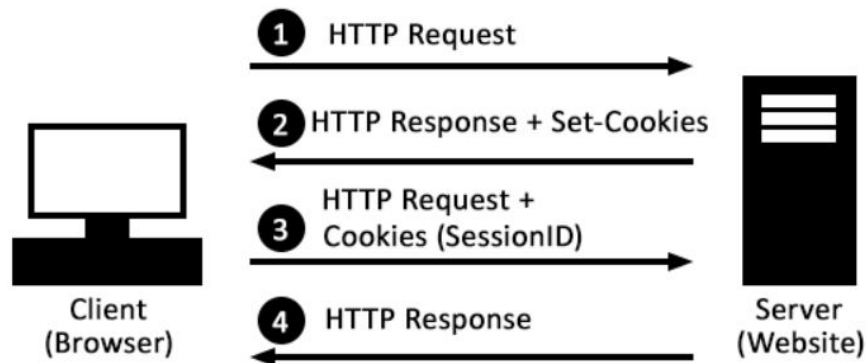


Techniques for Maintaining State

- Session-Based Authentication (Server-side session storage + Cookies for session IDs)
 - The server maintains session state.
 - The client holds only a session ID (usually in a cookie).
- Token-Based Authentication (JWTs, OAuth Tokens)
 - The session state is embedded within the token itself.
 - The server does not need to track user sessions.

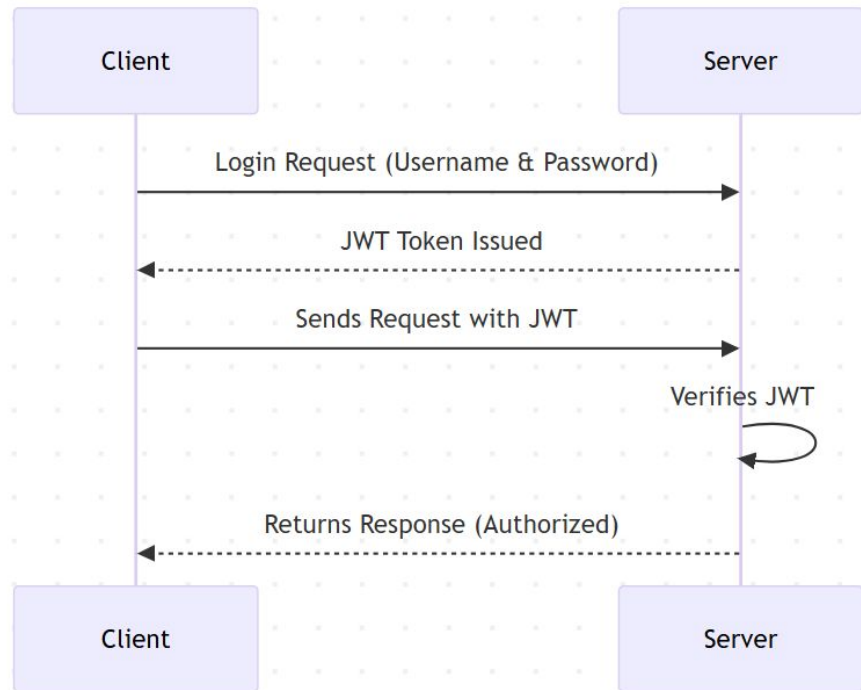
Session-Based Authentication

- Server-side session storage
- Cookies for session IDs
- User logs in → Server creates a session & assigns a session ID
- Session data is stored server-side, while session ID is sent to the client
- The client stores the session ID in the cookie



Token-Based Authentication

- Encodes session data in a self-contained token
- No need for server-side session storage
- Used in modern stateless authentication (e.g., OAuth, API tokens)

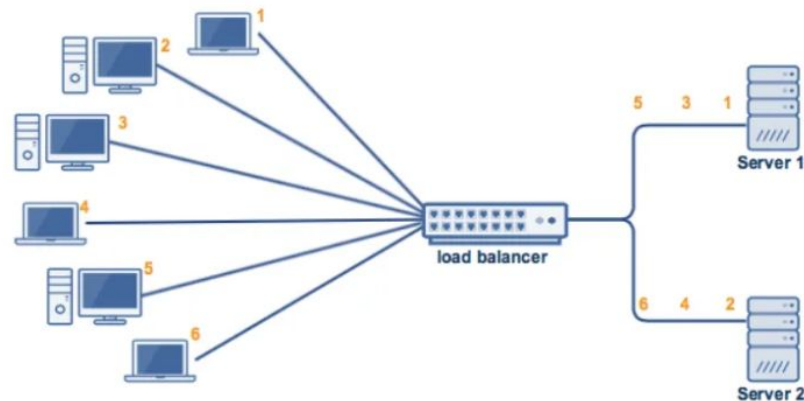


Security Concerns in Session Management





- Session Hijacking: Stolen session IDs
- Cross-Site Request Forgery (CSRF): Unauthorized actions
- Secure Cookie Handling: Avoiding theft via Secure, HttpOnly, and SameSite flags

Best Practices for Scaling Session Management

- Sticky Sessions vs. Distributed Sessions
- Storing session data in Redis, Memcached
- Stateless authentication (JWTs) for scalability



Important Interview Questions on Web Sessions

-  Understanding Statelessness & State Management
 - Why is HTTP considered a stateless protocol?
 - How do web applications maintain state despite HTTP being stateless?
-  Session Management Techniques
 - What are the differences between server-side sessions and client-side tokens?
 - When would you use JWT-based authentication instead of session-based authentication?
 - How do cookies and sessions work together in session-based authentication?
-  Security & Best Practices
 - What is session hijacking, and how can it be prevented?
 - How does CSRF exploit session management, and what are the mitigation strategies?
 - Why should cookies be set with Secure, HttpOnly, and SameSite attributes?
 - How can session management be scaled in distributed systems?
-  Real-World Scenarios & System Design
 - How does a load-balanced system handle session storage?
 - In a microservices architecture, how do you manage authentication across multiple services?
 - How do large-scale applications like Facebook, Amazon, or Google handle user sessions efficiently?

Summary & Key Takeaways

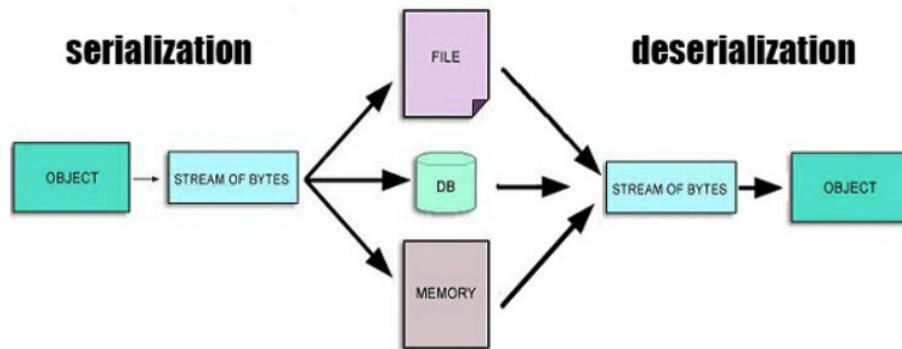
- HTTP is stateless, requiring session management techniques
- Cookies, server-side sessions, JWTs are common methods
- Security risks like session hijacking and CSRF must be mitigated
- Scaling sessions requires distributed storage solutions
- What's next:
 - Serialization: Data Exchange & Storage Formats

Serialization: Data Exchange & Storage Formats

System Design Fundamentals - Web Concepts

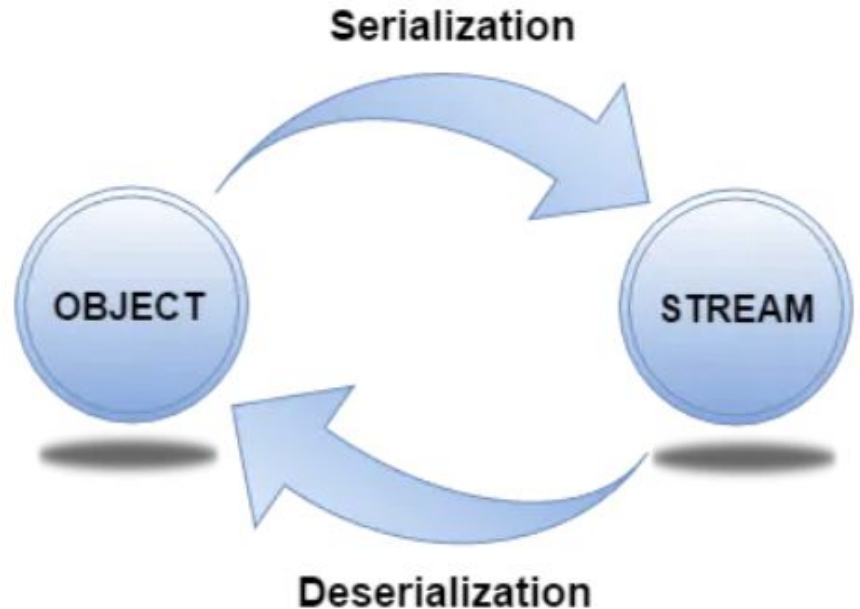
Why Serialization Matters?

- Applications need to exchange & store structured data efficiently.
- Serialization converts complex objects into a format that can be easily transferred.
- Used in APIs, databases, caching, and distributed systems.



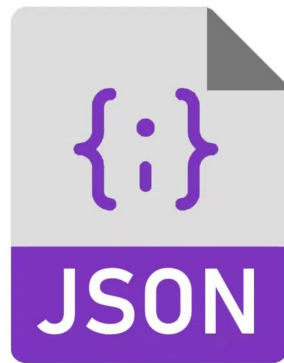
What is Serialization?

- **Serialization:** Converting objects into a format for transmission/storage.
- **Deserialization:** Converting it back into an object.
- Essential for distributed systems & inter-process communication.



Common Serialization Formats

- JSON – Human-readable, widely used in REST APIs.
 - Human-readable, widely used in APIs & web applications.
 - Simple key-value structure, easy to parse.
 - Text-based, so larger in size compared to binary formats.
- XML – Structured but verbose, used in legacy systems.
 - Tag-based markup language, used in legacy systems & configuration files.
 - More complex than JSON, supports rich data structures.
 - Verbose, leading to larger payloads.
- Protocol Buffers (Protobuf) – Compact & efficient, used in gRPC.
 - Binary format developed by Google.
 - Faster & smaller than JSON/XML, but requires schema definition.
 - Used in gRPC for high-performance APIs.



Trade-offs: Readability vs. Efficiency vs. Compatibility

- Readability: JSON & XML are human-readable, but inefficient.
- Efficiency: Protobuf & Avro are compact, reducing bandwidth usage.
- Compatibility: XML support schema evolution, JSON has limited support.







Serialization in action

- Serialization in APIs
 - REST APIs mostly use JSON.
 - gRPC APIs use Protobuf for efficiency.
 - XML is still used in SOAP-based web services.
- Serialization in Caching & Data Storage
 - Redis & Memcached: Store serialized JSON/Protobuf data.
 - Databases: NoSQL databases like MongoDB use BSON (Binary JSON).
 - Big Data: Protobuf used for efficient storage & schema evolution.

Performance Considerations

- Serialization choice impacts Bandwidth, CPU, and Memory Usage.
- JSON/XML → Larger payloads, slower parsing.
- Protobuf → Smaller payloads, faster parsing but needs schema.

Important Interview Questions on Serialization

-  Understanding Serialization
 - What is serialization, and why is it needed in system design?
 - How does serialization impact data exchange and storage?
-  Comparing Serialization Formats
 - What are the key differences between JSON, XML, Protocol Buffers, and Avro?
 - When would you choose Protobuf over JSON?
-  Serialization in APIs and Distributed Systems
 - How does serialization impact API performance and efficiency?
 - Why is Protocol Buffers commonly used in gRPC instead of JSON?
 - How does serialization affect caching strategies in systems like Redis?
-  Performance and Trade-offs
 - What are the trade-offs between readability, efficiency, and compatibility in serialization formats?
 - How does serialization impact CPU and memory usage?
-  Real-World Applications
 - How is Avro beneficial in big data systems?
 - Why do some databases like MongoDB use BSON instead of JSON?
-  Security Considerations
 - What security risks are associated with serialization?
 - How can improper deserialization lead to vulnerabilities?

Summary & Key Takeaways

- Serialization enables efficient data exchange & storage.
- Choose the right format based on performance, readability, and compatibility.
- JSON for APIs, Protobuf for efficiency, Avro for big data.
- Impacts bandwidth, processing speed, and storage efficiency.
- What's next:
 - CORS: Cross-Origin Resource Sharing & Web Security

CORS – Cross-Origin Resource Sharing & Web Security

System Design Fundamentals - Web Concepts

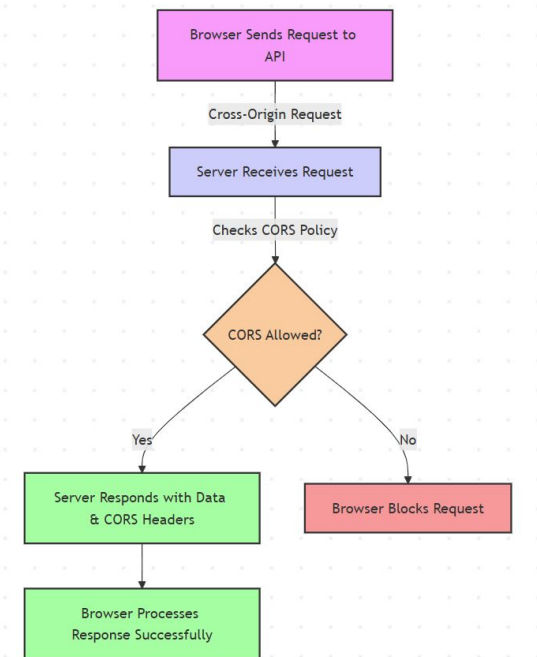
Why CORS Matters? (The Problem & Solution)

- **The Problem:** Browsers enforce Same-Origin Policy (SOP), blocking cross-origin requests by default.
- The Need for CORS:
 - Modern web apps rely on APIs hosted on different domains (e.g., Frontend on app.com, API on api.com).
 - CORS is a mechanism that allows secure cross-origin communication.

✗ Access to XMLHttpRequest at 'http://localhost:5000/global_config' step1:1 from origin 'http://localhost:8080' has been blocked by CORS policy: Response to preflight request doesn't pass access control check: No 'Access-Control-Allow-Origin' header is present on the requested resource.

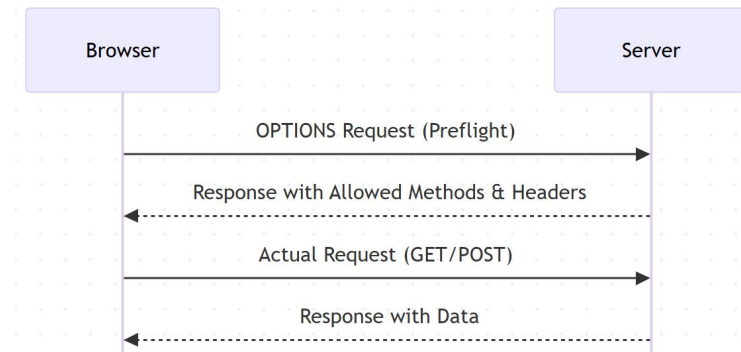
How CORS Works: Requests & Responses

- CORS is server-driven – the server must explicitly allow access.
- Two types of requests:
 - Simple requests: GET, POST (without custom headers).
 - Preflight requests: Needed for PUT, DELETE, or custom headers.
- CORS Headers control:
 - Access-Control-Allow-Origin (which origins can access).
 - Access-Control-Allow-Methods (allowed HTTP methods).
 - Access-Control-Allow-Headers (custom headers that can be sent).



Preflight Requests & CORS Headers

- Some requests require preflight checks (sent via OPTIONS request).
- Browser first sends a preflight request before making the actual request.
- If the server responds with valid CORS headers, the browser allows the actual request.
- Key Headers:
 - Access-Control-Allow-Origin: <https://example.com>
 - Access-Control-Allow-Methods: GET, POST
 - Access-Control-Allow-Headers: Authorization, Content-Type



Security Risks & Common Misconfigurations




- Common security risks:
 - Overly permissive CORS (Access-Control-Allow-Origin: *) → Allows any website to access sensitive data.
 - Allowing credentials with * (Access-Control-Allow-Credentials: true with * origin) → Leads to security vulnerabilities.
 - Exposing sensitive APIs via improper CORS settings.
- Mitigation Strategies:
 - Use a whitelist of trusted origins instead of *.
 - Set correct CORS policies for different API endpoints.
 - Use Reverse Proxies or API Gateways to handle CORS securely.



Handling CORS in APIs (REST & GraphQL)

- CORS in REST APIs:
 - JSON-based APIs commonly use CORS.
 - Configure headers properly in backend frameworks (e.g., Express.js, Spring Boot, Django).
- CORS in GraphQL APIs:
 - GraphQL APIs also need CORS handling since they operate over HTTP.
 - Preflight requests occur due to complex queries & mutations.

Alternatives to CORS & Role of API Gateways

- Alternatives to CORS for Cross-Origin Requests
 - Reverse Proxy:
 - Forwards client requests to the backend, bypassing browser CORS restrictions.
 - Example: Nginx proxying requests to a backend API.
 - API Gateway Handling CORS:
 - Centralized control of CORS policies for multiple services.
 - Example: AWS API Gateway configuring allowed origins.
- How API Gateways & Reverse Proxies Help?
 -  Reverse Proxies handle cross-origin requests internally to avoid CORS issues.
 -  API Gateways enforce CORS policies centrally, ensuring security & consistency.
 -  Both improve performance by reducing unnecessary browser preflight requests.

Interview Questions on CORS & Web Security

1. What is the Same-Origin Policy, and why does it exist?
2. How does CORS enable cross-origin requests?
3. What is a preflight request, and when is it required?
4. How do you configure CORS headers on a server?
5. What are common security risks associated with CORS?
6. What are alternatives to CORS for handling cross-origin requests?
7. How do API Gateways and Reverse Proxies help with CORS?

Summary & Key Takeaways

- Same-Origin Policy (SOP) restricts cross-origin requests for security reasons.
- CORS is a server-side mechanism that allows controlled cross-origin access.
- Properly configured CORS prevents security risks.
- Alternatives like backend proxying can bypass CORS when needed.
- What's next:
 - Moving to Summary & Practical Applications of Web Concepts.

Section Summary - Web Concepts in System Design

- Introduction to Web Concepts
 - How web applications work (Client-Server Model, Request-Response Cycle)
 - Stateless vs. Stateful interactions
 - Importance of security, scalability & performance
- Web Sessions: Managing State
 - Cookies, Server-Side Sessions, JWTs, Token-Based Authentication
 - Security concerns (Session Hijacking, CSRF)
 - Best practices for scaling session management
- Serialization: Data Exchange & Storage
 - Formats: JSON, XML, Protocol Buffers,
 - Trade-offs: Readability, Efficiency, Compatibility
 - Serialization in APIs, caching, and databases
- CORS & Web Security
 - Same-Origin Policy & why CORS is needed
 - How CORS works (Preflight Requests, Allowed Headers & Methods)
 - Common misconfigurations & security risks
 - API Gateways & Reverse Proxies for CORS handling
- What's next:
 - Scalability in System Design