Contents

Architecture Patterns & State Management

This document covers architectural patterns and state management strategies for system design.

Components

Stateful vs Stateless Design

- **Stateful Architecture**
  - **Characteristics:** Server maintains client session information between requests
  - **Benefits:** Rich user experience, simpler application logic, faster interactions (no repeated auth)
  - **Challenges:** Sticky sessions required, harder horizontal scaling, session failure points
  - **Use Cases:** Gaming, real-time collaboration, complex multi-step workflows
- **Stateless Architecture**
  - **Characteristics:** Each request contains all necessary information, no server-side session storage
  - **Benefits:** Excellent horizontal scalability, fault tolerance, any server handles any request
  - **Challenges:** Larger request payloads, complex client logic, repeated authentication overhead
  - **Implementation:** JWT tokens, client-side storage, external session stores (Redis)

Client-Server Logic Distribution

- **Server-Side Logic (Thick Server, Thin Client)**
  - **Characteristics:** Business logic, validation, processing handled on server
  - **Benefits:** Centralized control, security, easier updates, consistent behavior across clients
  - **Challenges:** Higher server load, increased latency, requires network for all operations
  - **Use Cases:** Financial systems, enterprise applications, security-critical operations
  - **Examples:** Traditional web applications, SaaS platforms, banking systems
- **Client-Side Logic (Thick Client, Thin Server)**
  - **Characteristics:** Business logic, validation, processing handled on client
  - **Benefits:** Responsive UI, reduced server load, offline capability, better user experience
  - **Challenges:** Harder to update, potential security risks, inconsistent behavior across devices
  - **Use Cases:** Gaming, creative tools, offline-capable applications, mobile apps
  - **Examples:** Desktop applications, mobile games, offline-first PWAs, rich SPAs
- **Hybrid Approach**

  - **Strategy:** Distribute logic based on specific requirements and constraints
  - **Server-Side:** Security validation, business rules, data persistence, complex calculations
  - **Client-Side:** UI logic, input validation, caching, user interactions
  - **Examples:** Modern web applications, mobile apps with local caching, collaborative editors

Related Trade-offs

- **Trade-off:** Developer maintainability and centralization vs. better UX responsiveness and reduced server load.
- **Questions to Ask:**
  - What are the capabilities of the frontend vs. backend teams?
  - Are clients resource-constrained (e.g., mobile devices)?
  - How frequently will the logic change?
  - Can the client update easily, or is it distributed?
  - Is the logic sensitive (e.g., security/validation)?
  - Are multiple clients consuming this logic (web, mobile)?

Related Trade-offs

UI Complexity vs. Server Complexity

- **Summary:** Placing logic on the server centralizes control and security, while client-side logic can improve responsiveness and offload server work.
- **Trade-off:** Control and security vs. responsiveness and scalability.
- **Questions to Ask:**
  - How responsive does the UI need to be?
  - Is the logic sensitive (e.g., security/validation)?
  - Are multiple clients consuming this logic (web, mobile)?

Stateful vs. Stateless Architecture

- **Summary:** Stateful architectures maintain client session information on the server between requests, allowing for rich user experiences and simpler application logic. Stateless architectures treat each request independently without storing client state on the server, enabling better scalability and fault tolerance.
- **Trade-off:** Rich user experience and simpler application logic vs. horizontal scalability and fault tolerance.
- **Architecture Comparison:**
  - **Stateful:** Server maintains session data, enables complex workflows, faster user interactions (no repeated authentication/context), but requires sticky sessions, harder to scale horizontally, single point of failure for user sessions
  - **Stateless:** Each request contains all necessary information, excellent horizontal scalability, fault tolerant (any server can handle any request), but potentially larger request payloads, more complex client logic, repeated authentication/authorization overhead
  - **Hybrid Approach:** Use stateless design for core APIs and data operations, stateful components for specific features like real-time collaboration, gaming, or complex multi-step workflows
- **Implementation Strategies:**

  - **Stateful:** In-memory sessions, database-backed sessions, sticky load balancing
  - **Stateless:** JWT tokens, client-side session storage, external session stores (Redis), database lookups per request
- **Questions to Ask:**
  - What's the expected user session complexity and duration?
  - How important is horizontal scalability vs. user experience richness?
  - Can clients reliably store and manage session information?
  - What are the fault tolerance and availability requirements?
  - Do you have complex multi-step workflows or simple CRUD operations?
  - How frequently do users interact with the system?
  - What's the tolerance for request overhead vs. server resource usage?
  - Are you building real-time collaborative features or traditional web applications?