Contents

API Design & Communication Patterns

This document covers API design patterns and communication strategies for system design.

Components

API Design Patterns

- **RESTful APIs**
  - **Resource-oriented, HTTP methods (GET, POST, PUT, DELETE), stateless, cacheable**
  - Standard HTTP status codes and methods
  - Clear resource-based URL structure
  - Stateless communication
- **RPC (Remote Procedure Call)**
  - **Function-oriented, flexible protocols, efficient for complex operations**
  - Direct function call semantics
  - Can use various protocols (HTTP, binary, gRPC)
  - More efficient for complex operations
- **GraphQL**
  - **Query language allowing clients to request specific data, reduces over-fetching**
  - Single endpoint for all operations
  - Client-specified data requirements
  - Strongly typed schema
- **Webhook APIs**
  - **Event-driven, server pushes notifications to client endpoints**
  - Push-based notifications
  - Event-driven architecture
  - Requires public client endpoints

API Versioning Strategies

- **URL Versioning**
  - `/api/v1/users` vs `/api/v2/users`
  - Clear and explicit versioning
  - Easy to implement and understand
  - Can lead to URL proliferation
- **Header Versioning**

  - `Accept: application/vnd.api+json;version=1`
  - Cleaner URLs
  - Version information in HTTP headers
  - More complex client implementation
- **Query Parameter**
  - `/api/users?version=1`
- Simple to implement
  - Easy to test different versions
  - Can clutter query parameters
- **Content Negotiation**
  - Different response formats based on Accept headers
  - Flexible content delivery
  - Leverages HTTP standards
  - More complex server logic

Related Trade-offs

Easy-to-Build APIs vs. Long-Term APIs

- **Summary:** Rapidly built APIs speed up early development but may introduce technical debt and backward compatibility issues. Long-term APIs require more upfront design but are stable and extensible.
- **Trade-off:** Short-term velocity vs. long-term maintainability and ecosystem trust.
- **Questions to Ask:**
  - Who are the consumers (internal, external, third-party)?
  - How likely is the API to change in the next 6–12 months?
  - Do we need versioning from the start?
  - What backward compatibility guarantees are needed?
  - Are we building an MVP or a long-term foundation?

REST vs. RPC (Remote Procedure Call)

- **Summary:** REST (Representational State Transfer) treats system interactions as operations on resources using standard HTTP methods, while RPC (Remote Procedure Call) treats system interactions as function calls that execute remotely. Each approach has different strengths for API design and system integration.
- **Trade-off:** Resource-oriented simplicity and HTTP compatibility vs. action-oriented flexibility and performance.
- **API Comparison:**
  - **REST:** Resource-based URLs, standard HTTP methods (GET, POST, PUT, DELETE), stateless, cacheable, but can be verbose for complex operations and may require multiple round-trips
  - **RPC:** Function-based calls, flexible protocols (HTTP, binary), efficient for complex operations, but tighter coupling, less cacheable, and protocol-dependent tooling
  - **Hybrid Approach:** Use REST for CRUD operations and public APIs, RPC for internal services requiring high performance or complex operations
- **Questions to Ask:**
  - Are you building public APIs or internal service communication?
  - Do your operations map naturally to CRUD on resources?

– What are your performance and latency requirements?
– How important is HTTP compatibility and caching?
– Do you need complex, multi-step operations?
– What's your team's familiarity with each approach?
– Are you building for web clients or diverse client types?
– How important is loose coupling vs. performance optimization?