Stateful vs. Stateless Architecture

- **Stateless:** The client includes all necessary data in each request, so the server doesn't store any prior information.
- **Stateful:** The server retains some data from previous requests, making future interactions dependent on past state

> In software systems, **state** refers to any data that persists across requests, such as user sessions, shopping carts, or authentication details.

Stateful Architecture

In a **stateful architecture**, the system remembers client or process data (**state**) across multiple requests.

Once a client connects, the server holds on to certain details—like user preferences, shopping cart contents, or authentication sessions—so the client doesn't need to resend everything with each request.

Stateful systems typically store the state data in a database or in-memory storage.

> Example: During online shopping, when you add items to your cart, the website remembers your selections. If you navigate away to browse more items and then return to your cart, your items are still there, waiting for you to check out.

Common Patterns in Stateful Architecture

1. Sticky Sessions

If you use **in-memory** session storage (i.e., each app server keeps its own sessions locally), you can configure your load balancer for "sticky sessions."

This means: Once a client is assigned to **Server A**, all subsequent requests from that client are routed to **Server A**.

Trade-off: If Server A fails, the user's session data is lost or the user is forced to re-log in. Sticky sessions are also less flexible when scaling because you can't seamlessly redistribute user traffic to other servers.

2. Centralized Session Store

A more robust approach is to store session data in a **centralized** or **distributed** store (e.g., Redis).

This allows:

- **Shared access**: All servers can access and update session data for any user. Any server can handle any request, because the session data is not tied to a specific server's memory.

> **Trade-off**: You introduce network overhead and rely on an external storage. If the centralized storage fails, you lose session data unless you have a fallback strategy.

Advantages:

- **Personalized Experiences:** Stateful systems can deliver highly tailored interactions, as they remember user preferences and past actions.
- **Contextual Continuity:** Users can seamlessly resume activities where they left off, even if they disconnect and reconnect.
- **Reduced Round Trips:** Certain operations can be faster because the server already possesses necessary data.

Challenges:

- **Scalability:** Maintaining state for a large number of users can become resource-intensive and complex, as each server needs to keep track of specific sessions.
- **Complexity:** Managing and synchronizing state across multiple servers (if needed) introduces additional challenges.
- **Failure Points:** If a server holding a user's state fails, their session data might be lost.

Example Use Cases

- **E-commerce Shopping Carts** – Stores cart contents and user preferences across multiple interactions, even if the user navigates away and returns.
- **Video Streaming Services (Netflix, YouTube)** – Remembers user watch progress, recommendations, and session data for a seamless experience.
- **Messaging Apps (WhatsApp, Slack)** – Maintains active user sessions and message history for real-time communication.

Stateless Architecture

In a **stateless** architecture, the server does **not** preserve client-specific data between individual requests.

- Each request is treated as **independent**, with no memory of previous interactions.
- Every request must include **all necessary information** for processing.
- Once the server responds, it **discards any temporary data** used for that request.

> **Example:** Most **RESTful APIs** follow a stateless design. For instance, when you request weather data from a public API, you must provide all required details (e.g., location) in each request. The server processes it, sends a response, and forgets the interaction.

Common Patterns in Stateless Architecture**

Token-Based Authentication (JWT)

A very popular way to implement statelessness is through tokens, particularly **JWTs** (JSON Web Tokens):

1. **Client Authenticates Once:** The user logs in using credentials (username/password) for the first time, and the server issues a signed **JWT**.
2. **Subsequent Requests:** The client includes JWT token in each request (e.g., `Authorization: Bearer <token>` header).
3. **Validation:** The server validates the token's signature and any embedded claims (e.g., user ID, expiry time).
4. **No Server-Side Storage:** The server does **not** need to store session data; it just verifies the token on each request.

> Many APIs, including OAuth-based authentication systems, use JWTs to enable stateless, scalable authentication.

Idempotent APIs

Stateless architectures benefit from **idempotent operations**, ensuring that repeated requests produce the same result. This prevents inconsistencies due to network retries or client errors.

Example: A `PUT /users/123` request with the same payload **always** updates the user's data but doesn't create duplicates.

> Idempotent APIs ensures consistency and reliability, especially in distributed systems where requests might be retried automatically.

Advantages:

- **Scalability:** Stateless systems are inherently easier to scale horizontally. New servers can be added effortlessly, as they don't need to maintain any

specific user sessions.

- **Simplicity:** Since servers don't track state, the architecture is generally simpler and easier to manage.
- **Resilience:** The failure of a single server won't disrupt user sessions, as data isn't tied to specific servers.
- **Lower Memory Footprint:** With no session data stored on the server, you free up memory that would otherwise be reserved for session management.
- **Easier to Cache Responses:** Since requests are self-contained, caching layers (like CDNs) can more easily store and serve responses.

Challenges:

- **Client-Side Complexity:** The client must keep track of the authentication token or relevant data. If it loses the token, it must re-authenticate.
- **Large Payloads:** Every request needs to carry all the required information, potentially leading to larger payloads.

Example Use Cases

1. **Microservices Architecture:** Each service handles requests independently, relying on external databases or caches instead of maintaining session data.
2. **Public APIs (REST, GraphQL):** Clients send tokens with each request, eliminating the need for server-side sessions.
3. **Mobile Apps:** Tokens are securely stored on the device and sent with every request to authenticate users.
4. **CDN & Caching Layers:** Stateless endpoints make caching easier since responses depend only on request parameters, not stored session data. A CDN can cache and serve repeated requests, improving performance and reducing backend load.

When to Choose Stateful Architecture

Stateful systems are ideal when **user context and continuity** are critical.

Consider a stateful approach if your application:

- Requires personalization (e.g., user preferences, session history)
- Needs real-time interactions (e.g., chat applications, multiplayer gaming)
- Manages multi-step workflows (e.g., online banking transactions, checkout processes)
- Must retain authentication sessions for security and convenience

> **Example:** A shopping cart in an e-commerce app should persist, so users don't have to re-add items after refreshing the page.

When to Choose Stateless Architecture

Stateless systems work best when **scalability, simplicity, and resilience** are top priorities.

Use a stateless approach if your application:

- Handles a high volume of requests and needs to scale efficiently
- Doesn't require storing client-specific data between requests
- Needs fast, distributed processing without server dependencies
- Must ensure reliability and failover readiness

> **Example:** A weather API doesn't need to remember previous requests. Each query includes the location, and the response is processed independently.

Hybrid Approaches: The Best of Both Worlds

Many modern applications **blend** stateful and stateless components for flexibility.

This hybrid approach allows:

- Stateless APIs for core functionality, ensuring high scalability
- Stateful sessions for personalization, improving user experience
- External session stores (e.g., Redis) to manage state while keeping app servers stateless

> **Example:** A video streaming platform (e.g., Netflix) uses a stateless backend for streaming but retains stateful user sessions to track watch history and recommendations.