Stateful vs. Stateless Architecture

ASHISH PRATAP SINGH
FEB 11, 2025

♡ 166 💬 4 🔁 5

When a client interacts with a server, there are two ways to handle it:

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

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

The choice between stateless and stateful architecture can affect scalability performance, complexity, and cost.

In this article, we'll break down both the approaches, their advantages and and when to use each—with real-world examples.

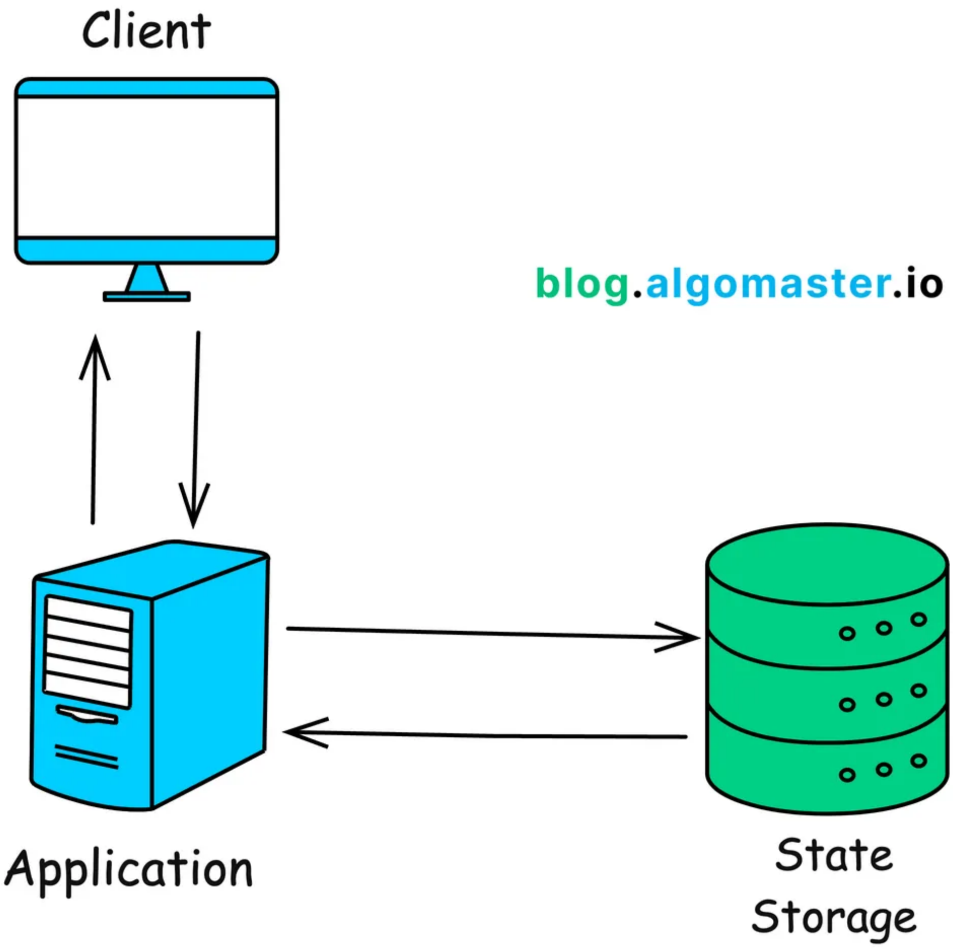
---

1. Stateful Architecture

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

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

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

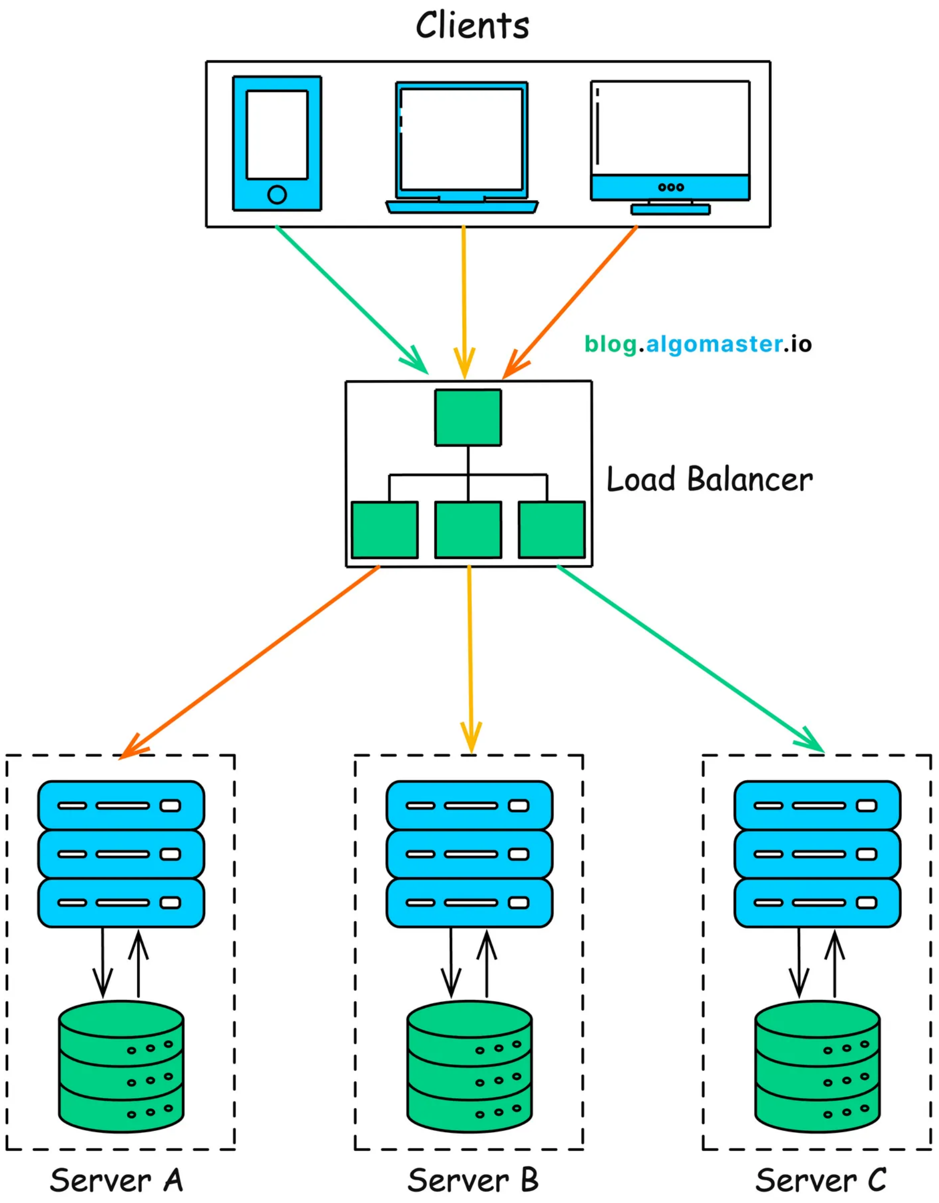


> **Example:** During online shopping, when you add items to your cart, the remembers your selections. If you navigate away to browse more items a return to your cart, your items are still there, waiting for you to check ou

Common Patterns in Stateful Architecture

1. Sticky Sessions

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



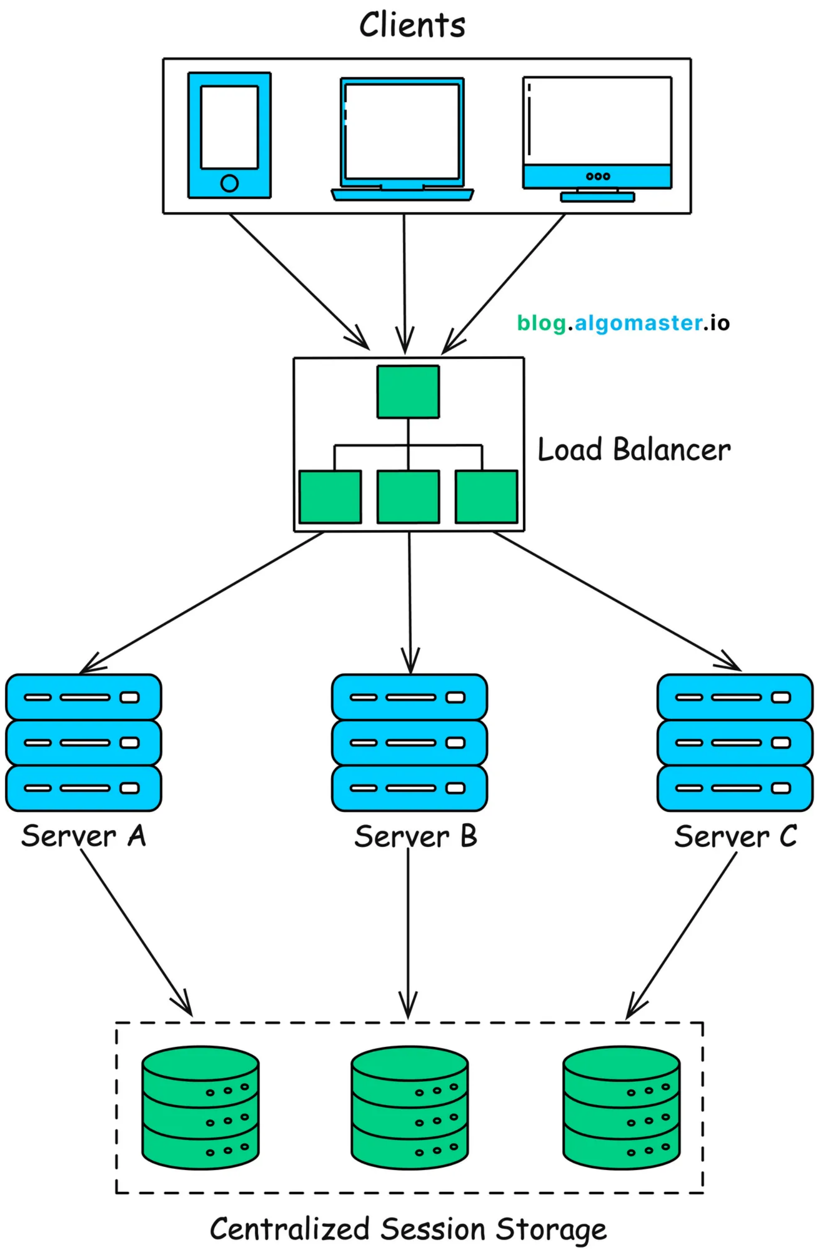
This means: Once a client is assigned to **Server A**, all subsequent requests f client are routed to **Server A**.

> **Trade-Off**: If Server A fails, the user’s session data is lost or the user is fo

> log in. Sticky sessions are also less flexible when scaling because you car
> seamlessly redistribute user traffic to other servers.

2. Centralized Session Store

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



This allows:

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

> **Trade-Off**: You introduce network overhead and rely on an external stor centralized storage fails, you lose session data unless you have a fallback

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 t even if they disconnect and reconnect.
- **Reduced Round Trips:** Certain operations can be faster because the se possesses necessary data.

Challenges:

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

Example Use Cases

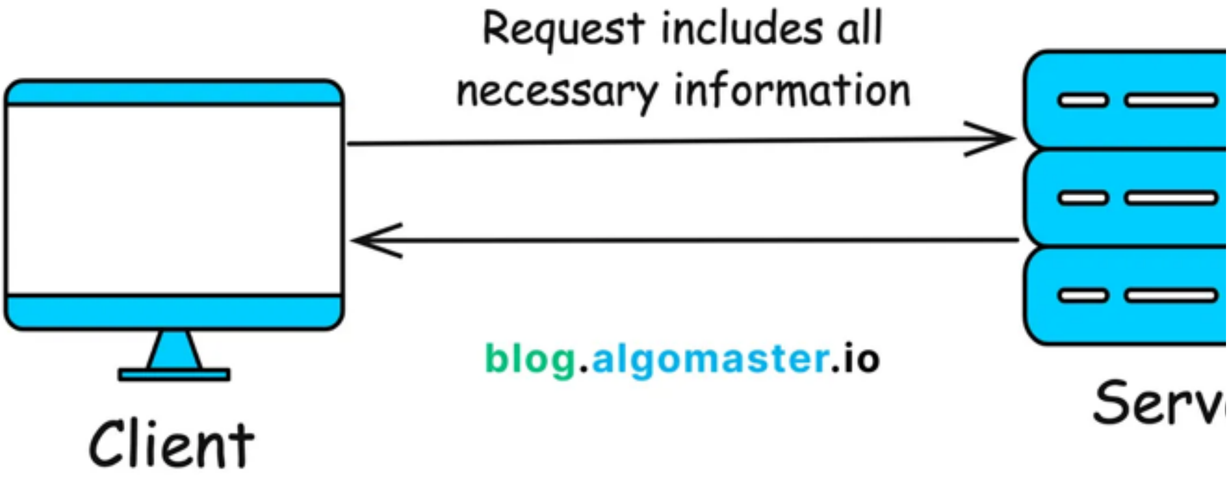
- **E-commerce Shopping Carts** – Stores cart contents and user preferenc multiple interactions, even if the user navigates away and returns.
- **Video Streaming Services** (**Netflix, YouTube**) – Remembers user watch recommendations, and session data for a seamless experience.

- **Messaging Apps (WhatsApp, Slack)** – Maintains active user sessions ai history for real-time communication.

2. Stateless Architecture

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

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



> **Example**: Most **RESTful APIs** follow a stateless design. For instance, wh request weather data from a public API, you must provide all required de location) in each request. The server processes it, sends a response, and f interaction.

Common Patterns in Stateless Architecture

1. Token-Based Authentication (JWT)

A very popular way to implement statelessness is through tokens, particular (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 **JW**
2. **Subsequent Requests**: The client includes JWT token in each request (e `Authorization: Bearer <token>` header).
3. **Validation**: The server validates the token's signature and any embedde (e.g., user ID, expiry time).
4. **No Server-Side Storage**: The server does **not** need to store session data verifies the token on each request.

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

2. Idempotent APIs

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

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

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

Advantages:

- **Scalability:** Stateless systems are inherently easier to scale horizontally

servers can be added effortlessly, as they don't need to maintain any spe
sessions.

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

Challenges:

- **Less Context:** Stateless systems can't provide the same level of persona context awareness as stateful systems without additional effort (like usi or tokens).
- **Client-Side Complexity:** The client must keep track of the authenticati relevant data. If it loses the token, it must re-authenticate.
- **Large Payloads:** Every request needs to carry all the required informati potentially leading to larger payloads.

Example Use Cases

1. **Microservices Architecture:** Each service handles requests independen 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 ev request to authenticate users.

4. **CDN & Caching Layers:** Stateless endpoints make caching easier since depend only on request parameters, not stored session data. A CDN ca serve repeated requests, improving performance and reducing backend

---

Choosing the Right Approach

There's no one-size-fits-all answer when choosing between stateful and sta architectures.

The best choice depends on your application's needs, scalability goals, and experience expectations.

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 gamin
- Manages multi-step workflows (e.g., online banking transactions, checl processes)
- Must retain authentication sessions for security and convenience

> **Example:** A shopping cart in an e-commerce app should persist, so users to re-add items after refreshing the page.

When to Choose Stateless Architecture

Stateless systems work best when **scalability, simplicity, and resilience** are 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. E 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 flex

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 s stateless

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

Thank you for reading!

If you found it valuable, hit a like ❤️ and consider subscribing for more suc every week.

If you have any questions or suggestions, leave a comment.

This post is public so feel free to share it.

P.S. If you’re enjoying this newsletter and want to get even more value, cons becoming a paid subscriber.

As a paid subscriber, you'll unlock all **premium articles** and gain full access premium courses on algomaster.io.

There are group discounts, gift options, and referral bonuses available.

Checkout my Youtube channel for more in-depth content.

Follow me on LinkedIn and X to stay updated.

Checkout my GitHub repositories for free interview preparation resources.

I hope you have a lovely day!

See you soon,

Ashish

166 Likes · 5 Restacks

← Previous

Discussion about this post

Comments Restacks

Write a comment...

Simon Chan 20 Mar

♥ Liked by Ashish Pratap Singh

Hi Ashish, your article is definitely clear and straightforward. very useful, Appreciate

♡ LIKE (1) REPLY

Venkat Sure 5 Aug

Thanks

♡ LIKE REPLY

2 more comments...