Stateless, Proxies, and Caching

Arnav Gupta

December 7, 2024

Contents

1 Stateless

The worker that is chosen to handle request can depend on application code.

A **stateless** thread/process/service remembers nothing from past requests:

- behaviour determined entirely by input request and request handling code
- different copies of service are running same code, so they give exact same response for a given request
- has no local state
- no long term memory (pure function)
- not affected by previous inputs

A **stateful** thread/process/service changes over time as side effect of handling requests:

- persistent, global variables are modified by the request processing code

OOP purposes:

- inheritance: allows strong typing without losing abstraction, creates generic, abstract interfaces, enabling abstraction

- modeling: real world concepts
- grouping sets of related state (memory/variable)
- well-defined, limited side effects
  - a class defines a set of member functions whose side-effects are limited to a small set of variables (object's data members)

Parallelism is easy with stateless code, but difficult with stateful code (related request from the same client must go to the same handler).

DB and cookies can help make stateless applications keep some state.

With cookies, a server returns a cookie (for example, after login), the browser includes the cookie in all future HTTP requests, and the server uses the cookie to determine which user it came from.

Statelessness allows for load balancing, since coordination only happens from shared DBs.

Push state up to client or down to DB.

2 Proxies and Caching

Proxy server: intermediate router for requests

Proxy does not know how to answer requests but knows who to ask. Request relayed to another server and response relayed back.

Proxies can be transparently added to any stateless service.

Load balancer: type of proxy that creates a single point of contact for a large cluster of app servers

Caching proxy: stores recently retrieved items for reuse

Cache: small data storage structure designed to improve performance when accessing a large data store

- stores most recently or most frequently accessed data

When reading data:

1. check cache, if it has data, cache hit
   (a) record in cache that entry was accessed

2. if data not in cache, cache miss
    (a) get data from main data store
    (b) make room by evicting another data element
    (c) store data in cache and repeat step 1

Most common eviction policy is LRU.

Managed cache:

- client has direct access to small and large data store
- client responsible for implementing caching logic

Transparent cache:

- client connects to one data store
- caching implemented inside storage black box

Data writes cause cache to be out of date. Solutions to this are:

- expire cache entries after some TTL (time to live)
- after writes, send new data or invalidation message to all caches, creating coherent cache (adds performance overhead)
- use versioned data: create new filename every time new data added

HTTP is stateless, so same response can be saved and reused for repeat requests:

- GET requests easy to cache (no modification)
- Cache-Control header for both client and server to enable/disable caching and control expiration time
- HTTP caching proxy is compatible with any web server and can be transparently added