Caching Entire Systems without Invalidation

SREcon EMEA 2022
October 27, 2022

Peter Sperl
Engineering Manager, Structured Products

TechAtBloomberg.com
Hello

This is a talk about distributed systems architecture
“What could SRE be?”
Product Managers → Feature Pressure, Time Pressure → Feature Developers → The Product

+Features → +Tech Debt

SREs → +Performance, +Reliability, -Tech Debt → SLIs
Being decoupled from this edge is what defines SRE
(according to me)
I’m here to show you how to design and build truly stateless systems
A stateless component always returns the same output for a given input.
Statelessness

- Cacheability
  - Performance
  - Scalability
  - Cost Reduction

- Testability
  - Rapid Iteration
  - Reliability

- Performance
- Scalability
- Cost Reduction
- Rapid Iteration
- Reliability
Caching Entire Systems without Invalidation
Caching Entire Systems without Invalidation
Caching without Invalidation
"There are 2 hard problems in computer science: cache invalidation and naming things."

– Phil Karlton
days_old(username)
Client ➔ cache ➔ days_old(user) ➔ Today’s date ➔ User DB
Whenever the date changes, this cache needs to be cleared.
Whenever a user’s birthday is updated, this cache needs to be **partially** cleared.
Client \rightarrow User Update Service \rightarrow User DB

Client \rightarrow cache \rightarrow days_old(user) \rightarrow Today's date

Date Change cron job

Update birthday
"The only good cache invalidation strategy is no strategy."

– Me, maybe
Client → days_old(user) → User DB
Client → days_old(user) → Today's date
Client → days_old(user) → cache (date_born, date_now)
Client \[\xrightarrow{\text{days_old}(user)}\] Gathering State \[\xrightarrow{\text{User DB}}\] \[\xrightarrow{\text{Today's date}}\] Immutable Zone

\[\begin{array}{l}
\text{days_old} \\
\text{(date_born, date_now)}
\end{array}\]
<table>
<thead>
<tr>
<th>Function</th>
<th>Cache Lifetime</th>
<th>Invalidation Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>days_old(date_born, date_now)</td>
<td>Forever</td>
<td>None</td>
</tr>
<tr>
<td>days_old(date_born)</td>
<td>Until tomorrow</td>
<td>TTL</td>
</tr>
<tr>
<td>days_old(username)</td>
<td>Until birthday changes</td>
<td>Active &amp; TTL</td>
</tr>
</tbody>
</table>
Cache

days_old(1982-09-06, 2022-10-27) = 14,661

days_old
(date_born, date_now)
What happens if I change my birthday?

days_old(6/9/82, 27/10/22) = 14,661
Old birthday is still in cache

Cache

days_old(6/9/82, 27/10/22) = 14,661

days_old(1/9/82, 27/10/22) = 14,666

New entry with new birthday

days_old
(date_born, date_now)
Cache

days_old(pete) = 14,661

days_old(username)
What happens if I change my birthday?

Cache

days_old(pete) = 14,661

days_old(username)
This is now wrong!
It should be 14,666

Cache

days_old(pete) = 14,661

days_old(username)
We need to erase the old entry first to allow the new value to be written.

days_old(pete) = 14,661

days_old(pete) = 14,666

days_old(username)
Cache

days_old(6/9/82) = 14,661
  (expires at midnight)

days_old(1/9/82) = 14,666
  (expires at midnight)

days_old(date_born)
True statelessness reduces total complexity
Any cache invalidation is bad
Interface design drives caching characteristics (among other things)
Stateful interfaces can be converted into stateless ones internally
Factor systems into stateful and stateless layers
Caching Entire Systems
Big Databases

✓ Small Databases  Resolve early into explicit values, replicate to scale
✓ Wall Time  Resolve early into explicit time or date

Software Versions
External Systems
Client 1

API Application Service

Business Logic Service

DB Access Service

Big DB

Gathering State

Client 2

UI Application Service
How can we make access to a large, constantly changing database, stateless?
How can we make access to a large, constantly changing database, stateless?

The timestamped data pattern

-or-

The snapshot pattern
<table>
<thead>
<tr>
<th>Entity</th>
<th>timestamp</th>
<th>some_data</th>
<th>more_data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Step 1
get_most_recent_timestamp(B) = 5

Step 2
get_data(B, 5) = some_data

This is immutable!
Step 1: Stateful call to get timestamp

```sql
select max(timestamp) from table where entity="B"
```

Step 2: Stateless call to get data using said timestamp

```sql
select data from table where entity="B" and timestamp <= 24
```
<table>
<thead>
<tr>
<th>Entity</th>
<th>timestamp</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>datastore.com/3fds80mvdy</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>datastore.com/7xdf8kasnw</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>datastore.com/cjw92kscnsq</td>
</tr>
</tbody>
</table>
Immutable Document Storage

Canonical Database

Snapshot Service

URL to Immutable document

Immutable Document Storage
Gathering State

Client → Application Service → Business Logic Service → DB Access Service → Big DB

Immutable Zone
Resolves stateful entity, like “IBM”, into the URL of an immutable document (db://reports/IBM/4UhJ8gF)
1. “Show me IBM’s analyst reports.”
2. `get_analysis(IBM)`
3. `get_most_recent_url(IBM, now) = db://IBM/4UhJ8gF`
4. `get_analysis(db://IBM/4UhJ8gF)`
5. `db://IBM/4UhJ8gF`
Benefits of Timestamped Data Storage

Database reads can be cached as well as any service call that depends on it.

Point-in-time access is trivially supported.

Batch jobs can freeze the timestamp to ensure consistency, while updates continue unaffected.

Rollbacks can be performed with a system-wide cap on timestamp.

Timed releases are just future-dated timestamps.
✔ Big Slow Databases
Use the timestamped data or snapshot pattern

✔ Small Databases
Resolve early into explicit values, replicate to scale

✔ Wall Time
Resolve early into explicit time or date

Software Versions

? External Systems
Resolve early, use the snapshot pattern, or give up
What if you deploy a new version of this service that changes the output?
Cache

\[ \text{biz\_logic(url)} = X \]
Cache

biz\_logic(url) = X

biz\_logic(url) = X

service v1

biz\_logic(url) = Y

service v2
Cache

\[ \text{hash}(v1, \text{biz\_logic}(\text{url})) = X \]
\[ \text{hash}(v2, \text{biz\_logic}(\text{url})) = Y \]

\[ \text{biz\_logic}(\text{url})=X \]

service v1

\[ \text{biz\_logic}(\text{url})=Y \]

service v2
This cache is “cleared” by the new service version

This cache still works!

Gathering State

Client

Application Service

Immutable Zone

Business Logic Service

New Version

DB Access Service

Big DB

Timestamp DB
<table>
<thead>
<tr>
<th>Topic</th>
<th>Recommendation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big Slow Databases</td>
<td>Use the timestamped data or snapshot pattern</td>
</tr>
<tr>
<td>Small Databases</td>
<td>Resolve early into explicit values, replicate to scale</td>
</tr>
<tr>
<td>Wall Time</td>
<td>Resolve early into explicit time or date</td>
</tr>
<tr>
<td>Software Versions</td>
<td>Include in cache key</td>
</tr>
<tr>
<td>External Systems</td>
<td>Resolve early, use the snapshot pattern, or give up</td>
</tr>
</tbody>
</table>
✔ Big Slow Databases  Use the timestamped data or snapshot pattern
✔ Small Databases  Resolve early into explicit values, replicate to scale
✔ Wall Time  Resolve early into explicit time or date
✔ Software Versions  Include in cache key
? External Systems  Resolve early, use the snapshot pattern, or give up
✖ Write-Heavy DBs  Resolve early, use TTL caching, or give up
True statelessness reduces total complexity

Any cache invalidation should be a non-starter

Interface design drives caching characteristics (among other things)

Stateful interfaces can be converted into stateless ones internally

Factor systems into stateful and stateless layers

Make low-level components stateless and chain upwards

Key generation is the right place to account for state