10.06.2026 · SHOPWARE COMMUNITY DAY

Caching What Matters: A Technical Deep Dive into Shopware's 6.7 Cache Rework.

Andrii Havryliuk & Jonas Elfering

WHAT YOU'LL TAKE AWAY

After 30 minutes you can...

- **Identify** the setups and plugins this rework affects.
- **Pick the right migration path** for your project.
- **Reason about** what is in the cache key and how it affects hit rate.
- **Configure cache behaviour** for your projects.
- **Debug** cache misses, wrong variants, and proxy quirks with confidence.

RECAP

In 6.7.0 we made the first steps.

DELAYED INVALIDATION

Fewer invalidations, higher hit rate

Batching invalidations rather than firing them on every write.

ROUTE-LEVEL STORE-API CACHE REMOVED

Less duplicated cache logic

Hit rate was too low to justify the complexity - but the API was left uncached. uncached.

ESI FOR HEADER & FOOTER

Better reuse of common page parts

Header and footer survive personalization on the rest of the page and across page navigation.

…WHICH OPENED THE DOOR FOR MORE

Each fix uncovered the next bottleneck.

6.7.0 cleared a layer of bottlenecks - and that made the next ones visible. This rework picks up from there.

THE PROBLEM

Five issues that drove this rework.

01	Huge amount of cache permutations.
02	Logged-in customers and filled carts bypassed the cache entirely.
03	Store-API sat outside HTTP caching.
04	Cache configuration was not flexible enough.
05	Reverse-proxy setup required Shopware-specific magic.

THE NEW MODEL, IN ONE SENTENCE

Cache behaviour is driven by HTTP, not by Shopware state.

WHAT THE CACHE SEES NOW

- header `Cache-Control` - whether and how long to store
- header `Vary` - which request dimensions split the cache
- header a smaller `sw-cache-hash` - only cache-relevant state

→ Shopware speaks HTTP. Caches do what they always did.

AVAILABLE NOW

Live since **6.7.7.0** behind the `CACHE_REWORK` feature feature flag.

Production-ready today - default in **6.8**.

- ISSUE 01 / 05

Cache permutations.

Too many cache variants, too little reuse.

ISSUE 01 · THE PROBLEM

The hash held every matched rule.

2^N	Rule combinations grow exponentially. **Each combination → its own own cache variant.**
1,024	N = 10 rules **One page, a thousand variants.**
∞	N = 50 **Effectively uncacheable.**

ISSUE 01 · THE SOLUTION

The hash gets shorter.

BEFORE · 6.7

2^N variants per page

sw-cache-hash: en-DE | EUR | net | logged-in |

~~r1+r3+r7+r12+r19+r24+r31+r44+r51+r57+r63+r78+r84+r92+r97+r112+r118+r125+r133+r141~~

AFTER · 6.8

sw-cache-hash: en-DE | EUR | net | logged-in | r7+r19

Every rule is tagged with the areas it's used in.

● product ~~promotion~~ ~~payment~~ ~~shipping~~ ~~flow~~

Only rules tagged for **cache-relevant areas** feed the hash - `product` by default.

Promotion, payment, shipping... rules still run; they just stop fragmenting the cache.

ISSUE 01 · ADOPTION

When should your rule affect the hash?

This applies if you use rules to influence what ends up in cached pages. Anywhere you've built on the flexibility of the rule builder to vary storefront output, you you should check this. Hook into `ResolveCacheRelevantRuleIdsExtension` to decide which rule IDs feed the hash.

ADD TO THE HASH WHEN…

- The rule changes rendered **cacheable storefront content.**
- The content is **public / shared** for that variant.

DON'T ADD TO THE HASH WHEN…

- The route should **not be stored at all.**
- The rule only affects **private or async-loaded** data.

- ISSUE 02 / 05

Logged-in & filled cart.

The bypass that left B2B performance on the table.

ISSUE 02 · THE PROBLEM

The old default was safe – but very expensive.

Anonymous shoppers — MOST REQUESTS HIT THE CACHE — Cached

Logged-in or **filled-cart** — CACHE SKIPPED ON EVERY REQUEST — Bypassed

OLD BEHAVIOUR

`sw-states` bypassed the cache.

- Only two hardcoded states existed: **logged-in** and **cart-filled** - no way to add more.
- HTTP cache bypassed sessions in those states by default.
- Projects could opt in - most did not.
- Extensions silently grew to depend on the bypass.

WHY IT WAS SET UP LIKE THAT

Permutations made it unsafe to cache.

- With exploding cache-hash permutations, caching logged-in traffic was inefficient inefficient anyway.
- **Safe:** no login state could leak into the shared cache.
- **Costly:** logged-in B2B accounts, returning customers, anyone mid-bypassed.

ISSUE 02 · THE SOLUTION

Cacheable by default, private by design.

6.8 BEHAVIOUR

- HTTP cache stays **active for logged-in customers and filled carts.**
- Core routes are cacheable by default - if an extension customizes a core route with private or dynamic data, adapt your code.
- **Custom routes opt in:** mark safe custom routes cacheable when they return return shared, non-private data.
- **Flexible:** any extension can contribute state to the hash - no more two-state list.

REMOVED

- `sw-states` cache handling
- `sw-currency` cache handling
- State-based HTTP cache bypass
- `CacheStateValidator` · `CacheStateSubscriber`

ISSUE 02 · ADOPTION

Audit personalized storefront output.

AUDIT THESE PATTERNS

- Customer-specific HTML rendered into cacheable pages
- Cart totals, customer names, account badges in shared markup
- Custom content added to cached core routes that wasn't reflected in the hash

PREFERRED FIXES

- **Mark safe custom routes cacheable** when they return shared, non-data
- Move private data to **async / uncached endpoints**
- Use **bounded cache variants** via `HttpCacheCookieEvent`
- Opt out **explicitly** for truly uncacheable flows

- ISSUE 03 / 05

Store API outside HTTP Cache.

Headless setups paid the highest price.

ISSUE 03 · THE PROBLEM

Store API stayed origin-heavy.

- The **route-level Store API cache was removed in 6.7** - hit rate was too low to justify.
- Store API routes were **not marked cacheable** for the shared HTTP cache either.
- Many read endpoints used **POST** - shared caches and CDNs do not cache POST without special handling.
- Result: Store API stayed **origin-heavy**, reverse proxies needed **project-specific workarounds.**
- Headless setups that lean on Store API were **hit hardest.**

ISSUE 03 · THE SOLUTION

Store API uses HTTP cache semantics.

WHAT CHANGED

- Selected Store API routes are **marked cacheable.**
- Read endpoints gain **GET support** where appropriate.
- Behaviour comes from **policies + headers.**
- Cache tagging supports **automatic invalidation** for selected routes.

AFFECTED ROUTE FAMILIES

- category · navigation · CMS · breadcrumb · seo-url
- product · search · cross-selling · find-variant · suggest
- country · currency · language · salutation
- media · landing-page

Long Criteria? Use the compressed `_criteria` parameter - see addendum A7.

ISSUE 03 · ADOPTION

API clients must carry the context.

The response sets `sw-cache-hash`. Your client carries it back so the shared cache can tell variants apart.

CLIENT RESPONSIBILITIES

- Send `sw-cache-hash` back (cookie or header)
- Prefer cacheable `GET` for read calls where possible
- Keep writes and private actions uncached
- Test all contexts: language, currency, rules, login, cart

RISK IF YOU DON'T

- Wrong cached variants served to the wrong customer, because cache relies on the wrong cache key.
- Cache-poisoning protection kicks in - **hitting cache efficiency**, because responses won't be stored when request and response cache hash differs.
- Subtle bugs (wrong language, wrong currency, wrong prices)

- ISSUE 04 / 05

Cache configuration.

A couple of global knobs and not much else.

ISSUE 04 · THE PROBLEM

One global TTL governed everything.

- `shopware.http.cache.default_ttl` - one number for the whole platform.
- `stale_while_revalidate`, `stale_if_error` - also global.
- Other `Cache-Control` directives - **not configurable.**
- Extension-defined TTLs were hard for operators to override.
- A product listing, a Store API metadata route, and a script endpoint **should not share one number.**

ISSUE 04 · THE SOLUTION

Named HTTP cache policies.

```yaml
# config/packages/shopware.yaml
shopware:
  http_cache:
    policies:
      storefront.cacheable:
        headers:
          cache_control:
            public: true
            s_maxage: 7200
            stale_while_revalidate: 120
            stale_if_error: 360
    default_policies:
      storefront:
        cacheable: storefront.cacheable
        uncacheable: no_cache_private
    route_policies:
      store-api.product.search: store_api.search
```

- **Named policies** define a full `Cache-Control` contract.
- **Defaults per area** - storefront, store_api - pick cacheable +
- **Route overrides** for the cases that materially differ.
- Applies to **browser, reverse proxy, CDN, and Symfony cache**

ISSUE 04 · ADOPTION

Move old config to policies.

REMOVED IN 6.8

- SHOPWARE_HTTP_DEFAULT_TTL
- shopware.http.cache.default_ttl
- shopware.http_cache.stale_while_revalidate
- shopware.http_cache.stale_if_error
- Store API route cache invalidation config

ADOPT

- **Named policies** for shared cache contracts
- **Default policies** per area (storefront, store_api)
- **Route policies** for special cases
- Near 1:1 migration of the global defaults - then refine
- **Extensions:** avoid shipping additional policies without a strong operators tune the defaults.

- ISSUE 05 / 05

Reverse-proxy magic.

Let the proxy be a proxy – it should understand HTTP, not Shopware

ISSUE 05 · FEEL THE DIFFERENCE

A working VCL fits on one screen.

```
# varnish.vcl
sub  vcl_recv {
    cookie.parse(req.http.cookie);
    if (!req.http.sw-cache-hash) {
        set req.http.sw-cache-hash = cookie.get("sw-cache-hash");
    }
    if (req.http.sw-cache-hash == "not-cacheable") {
        return (pass);
    }
    return (hash);
}
```

Our shipped config has more tuning, but you no longer *need* it - the proxy contract is now expressed in standard HTTP.

ISSUE 05 · THE SOLUTION

Standards take over from custom config.

HTTP STANDARD

`Cache-Control`

How long any cache may store the response - and who's allowed to.

REPLACES

- ~~Global `default_ttl` & `stale-while-revalidate` config~~
- ~~Hard-coded `Cache-Control` overrides in the reverse proxy~~

HTTP STANDARD

`Vary`

Which request headers split the cache into separate variants.

REPLACES

- ~~`sw-states` bypass branching~~
- ~~Custom hashing / variant logic on the proxy side~~
- ~~Per-route invalidation config~~

SHOPWARE GLUE - ONE HEADER

`sw-cache-hash`

The only Shopware-specific value the proxy needs to know about - it's just what `Vary` splits on.

REPLACES

- ~~`sw-states` + `sw-currency` cookies~~
- ~~The whole bypass-state model~~
- ~~Proxy-specific hashing mechanics~~

Three headers. **Any cache understands two of them out of the box;** the third is just a value to vary on.

ISSUE 05 · ADOPTION

Verify proxy and browser behaviour.

- Store-API & Storefront clients **forward `sw-cache-hash`** on every request - as a header or cookie.
- Reverse proxy **uses the updated config.**
- Browser-visible `Cache-Control` is **safe for browser caches**, which cannot be invalidated.
- Old reverse-proxy and state-based config are **removed.**

→ Proxy configuration smoke test

1. Create a product with **rule-based pricing.**
2. View it in a session where the **rule does not match** → default price. Reload → *served from cache.*
3. Switch to a session where the **rule matches** → rule price. Reload → *also from cache, different variant.*
4. Change the price in admin and **flush delayed invalidations** → next view shows the *new price.*

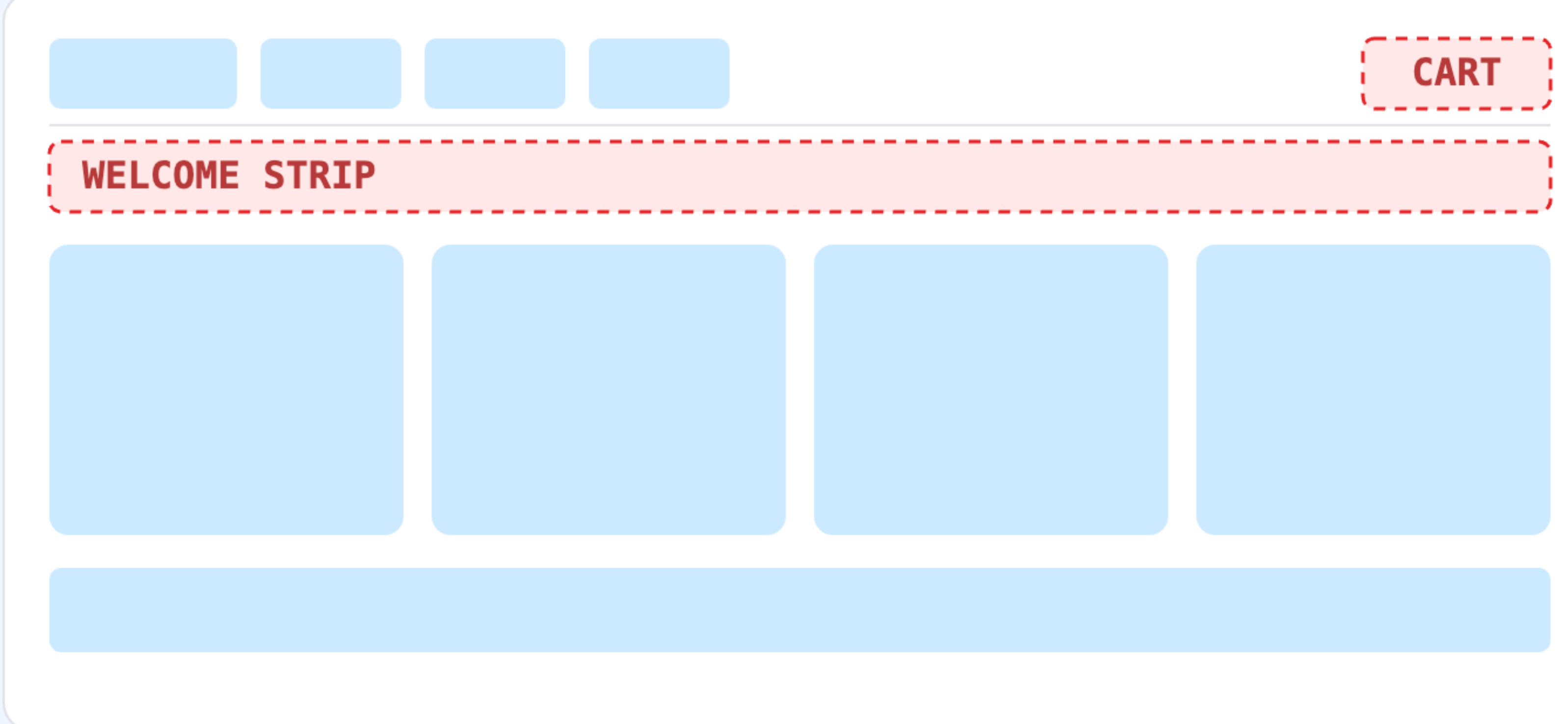
One scenario covers all three: cache hash, policy, proxy invalidation.

DESIGN PRINCIPLES

Designing cache-friendly extensions.

- Load **private data asynchronously**.
- Use **bounded variants** - never per-customer cache keys, keep cardinality low.
- Prefer **GET for cacheable reads.**
- Opt out **explicitly** for truly uncacheable flows.

SHARED SHELL + PERSONAL SLIVERS



Shared shell - **1 cached request**

Personal slivers - **async, uncached**

THE FULL PICTURE · 1 OF 2

A normal cacheable request.

Browser client	Shared cache varnish · fastly · symfony	Shopware origin php / symfony
1 · REQUEST `GET /home`	**2 · READ HASH** Receives `sw-cache-hash`	
	3 · LOOK UP VARIANT URL + `Vary: sw-cache-hash` · MISS	**4 · ROUTE** Apply route + policy
7 · RECEIVE 200 OK · `Age: 0`	**6 · STORE + SERVE** Stores response with policy TTL	**5 · RESPOND** 200 OK · `Cache-Control` · `Vary` · optional new `sw-cache-hash`

On the hot path, Shopware does not run. **The fastest code is still the code that does not run.**

THE FULL PICTURE · 2 OF 2

What happens when context changes.

TRIGGERS A NEW HASH

- Currency change
- Language change
- Login / logout
- Cart state change
- Cache-relevant rule match

FLOW

01 Context-changing request hits the **origin.**

02 Shopware **calculates the new hash.**

03 Response sets / updates `sw-cache-hash.`

04 When request and response sw-cache-hash differ, Shopware sets Cache-Control: no-Control: no-store (cache poisoning protection)

05 Next request **carries the new hash.**

06 Shared cache uses `Vary` to **select the right variant.**

The variant is the safety mechanism. The cache holds more traffic **because** context changes are deliberate.

FOR EXTENSION AUTHORS

Four extension points to know.

`ResolveCacheRelevantRuleIdsExtension`

Decide which rule IDs feed the cache hash. Add rule areas or specific IDs for rules that change cacheable output.

`CacheHashRequiredExtension`

Decide whether a hash needs to be calculated for this request at all - the gate before gate before `HttpCacheCookieEvent` fires.

`HttpCacheCookieEvent`

Contribute to the hash, or mark the session uncacheable (`isCacheable = false`) or this response only (`doNotStore = true`).

`HttpCacheKeyEvent`

Customize Symfony's HTTP cache key only.

⚠ **Discouraged - no effect on reverse proxies.**

RECAP

What you need to remember.

01

Smaller `sw-cache-hash` - only rules that change output.

02

Logged-in & filled-cart traffic **can be cached.**

03

Store API uses **HTTP caching**, not its own layer.

04

Policies replace **global cache config.**

05

Reverse proxies follow **headers + Vary.**

YOUR 6.8 MIGRATION

Migration checklist.

- [ ] Remove `sw-states` / `sw-currency` assumptions.
- [ ] Move old cache config to policies.
- [ ] Update the reverse-proxy config.
- [ ] Audit personalized storefront output.
- [ ] Update API clients to carry `sw-cache-hash`.
- [ ] Test browser + proxy headers in production-like setup.

Details, code, removed APIs, and Varnish reference → **addendum (next).**

- Q&A · Keep the conversation going

Cache what matters – let's keep talking.

Find us on Discord · #shopware-cache →

or grab us in person - **Andrii & Jonas**

- REFERENCE

Addendum.

Migration code, removed APIs, Varnish notes
- the slides you'll want offline.

ADDENDUM A1

Refresher: Cache vocabulary.

Cache-Control

`max-age`

Browser / private cache TTL

`s-maxage`

Shared cache TTL

`public / private / no-store`

Who may store this response

`stale-while-revalidate`

Serve stale while refreshing

`stale-if-error`

Serve stale on origin failure

Age

`Age: 1200`

How long this response has lived in a cache. Lets

Lets multiple caches stack TTLs correctly.

Vary

`Vary: sw-cache-hash`

Which request headers create separate variants of the same URL.

HIT / MISS

`HIT`

Cache could serve the response without going to to origin.

`MISS`

Cache had no matching variant - request was forwarded.

Example

```
Cache-Control: public,
  s-maxage=3600,
  stale-while-revalidate=120
Age: 1200
Vary: sw-cache-hash
```

ADDENDUM A2

Debugging the new cache model.

INSPECT

- `Cache-Control` on the response
- `Age` - how long it has lived in cache
- `Vary` - what differentiates variants
- `sw-cache-hash` cookie + header
- `sw-dynamic-cache-bypass`
- Proxy forwarding + normalization

ASK

- **Should** this response be stored?
- Is this the **right variant**?
- Did the request reach origin **because it had to**?
- Did the hash change **when it should**?
- Is the proxy **passing headers through**?

ADDENDUM A3

Removed Store API cache events.

REMOVED IN 6.8 · NO REPLACEMENT

- `StoreApiRouteCacheKeyEvent`
- `StoreApiRouteCacheTagsEvent`
- Child events for: category · navigation · country · product · currency · landing-page · language · payment · salutation · shipping · sitemap · AI search

Already unused since 6.7 when the Store API route cache layer went away.

MIGRATION

- Remove listeners / subscribers for these events.
- Move customizations to **HTTP cache extension points + policies.**
- Do **not** try to recreate the old route cache layer - the replacement is the **shared HTTP cache.**

ADDENDUM A4

Add rule relevance to the hash.

```php
use Shopware\Core\Framework\Adapter\Cache\Http\Extension\ResolveCacheRelevantRuleIdsExtension;
use Symfony\Component\EventDispatcher\EventSubscriberInterface;

final class CacheRuleSubscriber implements EventSubscriberInterface
{
    public static function getSubscribedEvents(): array {
        return [
            ResolveCacheRelevantRuleIdsExtension::NAME . '.post' => 'addRuleAreas',
        ];
    }

    public function addRuleAreas(ResolveCacheRelevantRuleIdsExtension $extension): void {
        $extension->ruleAreas[] = 'my-custom-rule-area';
    }
}
```

Only add rule IDs / areas when they change cacheable output. For custom rule associations use the `RuleAreas` DAL flag.

ADDENDUM A5

HttpCacheCookieEvent – bypass + don't-store.

```
public static function getSubscribedEvents(): array {
    return [
        CacheHashRequiredExtension::NAME . '.post' => 'hashRequired',
        HttpCacheCookieEvent::class                => 'configureHash',
    ];
}

public function hashRequired(CacheHashRequiredExtension $e): void {
    if ($this->featureChangesOutput()) {
        $e->result = true;
    }
}

public function configureHash(HttpCacheCookieEvent $e): void {
    if ($this->shouldBypass()) {
        $e->isCacheable = false;
        return;
    }

    if ($this->skipThisResponse()) {
        $e->doNotStore = true;
    }
}
```

`isCacheable = false`

Permanent bypass for the session. **Example:** B2B customer with customer-customer-specific prices.

`doNotStore = true`

Skip storing this one response. **Example:** a page that just rendered a flash message.

ADDENDUM A6

Mark your Store API route cacheable.

```
use Shopware\Core\Framework\Routing\PlatformRequest;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Routing\Attribute\Route;

#[Route(
    path: '/store-api/example',
    name: 'store-api.example',
    methods: [Request::METHOD_GET],
    defaults: [
        PlatformRequest::ATTRIBUTE_HTTP_CACHE => true,
    ],
)]
public function load(Request $request): Response
{
    // Return shared, non-private data only.
}
```

- Use for **safe read endpoints** returning shared, non-private data. data.
- Do **not** include customer-specific or private data.
- Same mechanism used by category, product, currency, language, language, salutation, media…

ADDENDUM A7

GET + compressed _criteria.

WHY GET

- Shared caches + CDNs understand GET
- POST read requests need special proxy handling
- Store API Criteria can exceed query-string limits

_CRITERIA PARAMETER

- Query parameter: `?_criteria=…`
- Format: `base64url(gzip(json_encode(criteria)))`
- Custom fields are copied to the query bag for plugin filters

Future: HTTP QUERY could replace this - not standardized yet.

ADDENDUM A8

App-script cache API changes.

REMOVED

```
{% do response.cache.maxAge(3600) %}

{% do response.cache.invalidationState(
    'logged-in', 'cart-filled'
) %}
```

USE

```
{% do response.cache.sharedMaxAge(3600) %}

{% do response.cache.clientMaxAge(300) %}

{# State-based invalidation: no replacement #}
```

- `sharedMaxAge()` → `s-maxage` · `clientMaxAge()` → browser `max-age`
- Admins can override script policies via `route_policies` with the `route#hook` pattern.

ADDENDUM A9

Defining policies.

```yaml
shopware:
  http_cache:
    policies: # list of named policies
      storefront.cacheable: # policy name
        headers:
          cache_control: # Cache-Control header directives
            public: true
            s_maxage: 7200
            stale_while_revalidate: 120
            stale_if_error: 360
      no_cache_private:
        headers:
          cache_control:
            private: true
            no_store: true
```

A policy is a named bag of `Cache-Control` directives. Shopware ships three defaults - `storefront.cacheable`, `store_api.cacheable`, and `no_cache_private`. You can override any of them, but **no merging happens** - the whole policy has to be redefined.

ADDENDUM A10

Applying policies.

```yaml
shopware:
  http_cache:
    default_policies: # policies to apply per area
      storefront: # area name
        cacheable: storefront.cacheable
        uncacheable: no_cache_private
      store_api: # area name
        cacheable: store_api.cacheable
        uncacheable: no_cache_private
    route_policies: # policies to apply per route or app hook
      store-api.product.search: store_api.search
      frontend.script_endpoint#my-hook: storefront.short
```

PRECEDENCE - LOWEST TO HIGHEST

01 **Default policy** for the area.

02 TTLs set by an app script via `sharedMaxAge()` / `clientMaxAge()`.

03 **Route** or **app-script hook** policy.

App-script TTLs are **merged into** the area default. A route policy **overwrites every** directive of the default.

`default_policies` sets the per-area baseline. `route_policies` overrides specific routes (or app-script hooks) by name.

ADDENDUM A11

Hit-for-miss - caching the fact that a response wasn't cacheable.

WITHOUT A MARKER

Concurrent requests pile up.

When the backend returns an uncacheable response, Varnish has nothing stored to remember that fact - so the next request for the same URL still tries the cache, request-coalesces with the in-flight one, and serializes.

Each origin hit blocks the next. Under load the queue stalls.

HIT-FOR-MISS

A tiny “skip me” marker.

Varnish stores a stand-in object for the URL that says *“this returned returned uncacheable - for the next N seconds, send requests straight to origin without going through cache lookup or coalescing.”*

After the TTL elapses, the URL is treated as potentially cacheable again. The marker only suppresses cache lookup - for a window.

Reference: Varnish - Hit-for-miss and why a null TTL is bad for you.

ADDENDUM A12

Passing a single response, keeping the URL cacheable.

```
# vcl_backend_response
if (beresp.http.sw-dynamic-cache-bypass == "1") {
    set beresp.uncacheable = true;
    set beresp.ttl         = 1s;
    unset beresp.http.sw-dynamic-cache-bypass;
    return (deliver);
}
```

`sw-dynamic-cache-bypass`

A custom response header Shopware sets when a single request turns out personal mid-flight. We **strip it here** so it never reaches clients - the URL itself stays cacheable for everyone else.

`beresp.uncacheable = true`

Tell Varnish not to store this response. The personal HTML never lands in the shared cache.

`beresp.ttl = 1s`

Caps the hit-for-miss window at one second - just long enough to skip cache for skip cache for in-flight peers, short enough that other visitors aren't locked out locked out of the cacheable URL.

ADDENDUM A13

SystemConfig – the `$silent` default flips in 6.8.

→ Every `set()` / `setMultiple()` / `delete()` call invalidated the HTTP cache tag `system.config-{salesChannelId}` - chains of admin writes triggered **cache-invalidation storms.**

→ A `silent` parameter was added - when `true`, the internal cache cache is still cleared, but the HTTP cache tag is **not invalidated.**

→ **6.8:** the default switches to `silent: true.` Pass `silent: false` explicitly when a config change has immediate effect on rendered pages and and therefore needs the HTTP cache invalidated.

```
$this->systemConfigService->set(
    key: 'MyPlugin.config.showBanner',
    value: true,
    salesChannelId: $salesChannelId,
    silent: false,
);
```

ADDENDUM A14

Outlook: What's next.

CDN-CACHE-CONTROL

CDN-specific directives.

A draft `CDN-Cache-Control` header - similar to `Cache-Control`, but only applies to CDNs and reverse proxies. Today, `s-maxage` directive is used as a replacement.

HTTP QUERY

"GET with a body."

A cleaner future for large Store API Criteria - eventually replacing the compressed `_criteria` workaround. Not standardized yet.

Not migration requirements for 6.8 - just the direction. **Keep moving toward standards. Keep Shopware-specific behaviour small behaviour small and explicit.**

ADDENDUM A15

Removed & deprecated APIs.

SW-STATES

- `HttpCacheKeyGenerator::SYSTEM_STATE_COOKIE`
- `HttpCacheKeyGenerator::INVALIDATION_STATES_HEADER`
- `CacheStateValidator`
- `CacheStateSubscriber`
- `CacheAttribute::$states`

SW-CURRENCY

- `HttpCacheKeyGenerator::CURRENCY_COOKIE`

SYSTEMCONFIGSERVICE TRACING

- `SystemConfigService::trace()`
- `SystemConfigService::getTrace()`

OLD STORE API ROUTE CACHE CONFIG

- All `shopware.cache.invalidation.*_route` options

ADDENDUM A16

Useful links

- → Docs HTTP-Cache Manipulation
- → Docs HTTP-Cache Policies
- → Store-API Caching ADR
- → Cache Rework ADR
- → Varnish VCL Config
- → Fastly Config
- → Cache Rework Blog Post
- → Cache-Control Header on MDN
- → RFC 9111 - HTTP Caching