imagine 2018

LEAD THE CHARGE

Understand and improve your Magento 2 store caching and performance

imagine 2018
Magento

Renato Cason

Head of Development | END.

Overview

- Introduction
- Application Cache
  - Block caching
  - Cache clean by tag
  - Custom cache types
  - Profiling and troubleshooting
- Page Cache
  - Invalidation
  - ESI
  - Content variation handling
- Results validation

01. Introduction

Performance optimization in the development workflow

Lazy approach

- Implement the functionality
- Deploy to production
- Receive reports of performance deterioration
- Profile the application
- Implement performance fixes
- Deploy them to production
- Check the results on monitoring tools

Proactive approach

- Implement the functionality
- Test performance on production-like environment
  - Performance testing: single thread testing
  - Benchmarking: application behavior under load
- Deploy to production
- Validate live data from monitoring tools

How to be proactive

- Architect your application to
  - Cache everything that make sense to be cached
  - Load efficiently what can’t be cached (i.e. private content)
- Test how much the changes you do affect your store’s performance before pushing them to production

Advanced:

- Automate load testing in your CI/CD pipeline

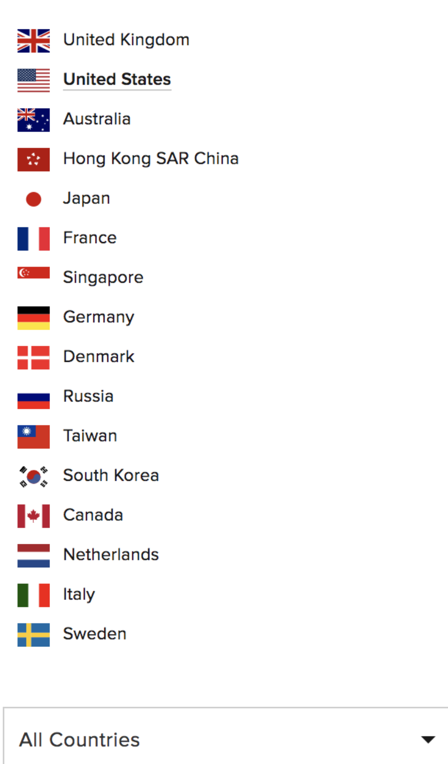
02. Application Cache

How to leverage Magento’s application cache to improve your application performance.

The example

Module for Magento 2.2.x:
https://github.com/renatocason/imagine2018-storepicker-example

- Store picker proof of concept
  (show the list of available country, and link each of them to a specific store view, i.e. US and GB to /en/, ES and AG to /es/, and so on)
- Frontend block
- Grid and form in admin for CRUD operations



Block caching

Use when the block rendering requires:

- Database reads
- API calls
- Complex computations

Activate by setting a cache lifetime to the block:

```
<block class="Magento\Framework\View\Element\Template" name="locations"
      template="Rcason_StorePicker::locations.phtml">
    <arguments>
        <argument name="viewModel" xsi:type="object">
            Rcason\StorePicker\ViewModel\Locations
        </argument>
        <argument name="cache_lifetime" xsi:type="number">84600</argument>
    </arguments>
</block>
```

Under the hood

Class *Magento\Framework\View\Element\AbstractBlock*

Method _*saveCache* is caching the block only if cache is enabled and the cache lifetime for the block is set:

```
if (!$this->getCacheLifetime() || !$this->_cacheState-
>isEnabled(self::CACHE_GROUP)) {
    return false;
}
$cacheKey = $this->getCacheKey();

$this->_cache->save($data, $cacheKey, array_unique(
    $this->getCacheTags()), $this->getCacheLifetime());
```

imagine 2018
Magento

Under the hood

Method *getCacheKey* is using cache key info to generate the cache entry key:

```
$key = $this->getCacheKeyInfo();
$key = array_values($key);
$key = implode('|', $key);
$key = sha1($key);
return static::CACHE_KEY_PREFIX . $key;
```

When using *Magento\Framework\View\Element\Template*, the cache key info is computed this way:

```
return [
    'BLOCK_TPL',
    $this->_storeManager->getStore()->getCode(),
    $this->getTemplateFile(),
    'base_url' => $this->getBaseUrl(),
    'template' => $this->getTemplate()
];
```

Cache clean by tag

The problem. Underlying data might change,
but the block will still show the old content until:

- The cache entry expires
- The cache is flushed manually

The solution:

- Tag each cache entry based on its content
- When an entity is updated, remove all cache entries that are tagged accordingly

Cache clean by tag

Specify a cache tag on your model:

```
const CACHE_KEY = 'storepicker_location';
protected $_cacheTag = self::CACHE_KEY;
```

Add it to the block’s cache tags:

```
<block class="Magento\Framework\View\Element\Template" name="locations"
      template="Rcason_StorePicker::locations.phtml">
    <arguments>
        <argument name="view_model" xsi:type="object">
            Rcason\StorePicker\ViewModel\Locations
        </argument>
        <argument name="cache_lifetime" xsi:type="number">84600</argument>
        <argument name="cache_tags" xsi:type="array">
            <item name="cache_key" xsi:type="string">storepicker_location</item>
        </argument>
    </arguments>
</block>
```

Under the hood

Class *Magento\Framework\Model\AbstractModel*

Method *cleanModelCache*, called from method *afterSave*, is cleaning all the cache tags that match with the model's:

```
$tags = $this->getCacheTags();
if ($tags !== false) {
    $this->_cacheManager->clean($tags);
}
```

While *getCacheTags* returns the object property _cacheTag, which defaults to *false* and can be either a single value or an array:

```
if (is_array($this->_cacheTag)) {
    $tags = $this->_cacheTag;
} else {
    $tags = [$this->_cacheTag];
}
```

And, if set to true, will clean the whole cache:

```
if ($this->_cacheTag === true) {
    $tags = [];
}
```

Advanced - Custom cache type

The problem. Performing the same operations:

- Database reads
- API calls
- Complex computations

But using the results not only in blocks, but for instance in:

- API responses
- CLI scripts

The solution:

- Extend *Magento\Framework\Cache\Frontend\Decorator\TagScope* to create a custom cache type
- Add a *cache.xml* file to the *etc* folder in your module to allow cache flushing from admin interface or CLI

Custom cache type class

```
class Cache extends \Magento\Framework\Cache\Frontend\Decorator\TagScope
{
    const TYPE_IDENTIFIER = 'storepicker';
    const CACHE_TAG = 'STOREPICKER';
    const CACHE_TTL = 84600;

    public function __construct(
        \Magento\Framework\App\Cache\Type\FrontendPool $cacheFrontendPool
    ) {
        parent::__construct(
            $cacheFrontendPool->get(self::TYPE_IDENTIFIER),
            self::CACHE_TAG
        );
    }


    // [...]
}
```

Custom cache type class

```php
public function setLocations($data)
{
    $this->save(
        json_encode($data),
        self::TYPE_IDENTIFIER . '_locations',
        [Location::CACHE_KEY],
        self::CACHE_TTL
    );
}

public function getLocations()
{
    $value = $this->load(self::TYPE_IDENTIFIER . '_locations');
    return $value ? json_decode($value) : false;
}
```

Custom cache type cache.xml

Not necessary, but useful in case you want to be able to clean the cache separately, from the admin or the CLI.

```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="urn:magento:framework:Cache/etc/cache.xsd">
    <type name="storepicker" translate="label,description”
          instance="Rcason\StorePicker\Model\Cache">
        <label>Store Picker</label>
        <description>Store picker locations cache</description>
    </type>
</config>
```

Custom cache type cache.xml

Cache Management

Flush Cache Storage | Flush Magento Cache

☐	Integrations Configuration	Integration configuration file	INTEGRATION	ENABLED
☐	Integrations API Configuration	Integrations API configuration file	INTEGRATION_API_CONFIG	ENABLED
☐	Page Cache	Full page caching	FPC	DISABLED
☐	Translations	Translation files	TRANSLATE	ENABLED
☐	Web Services Configuration	REST and SOAP configurations, generated WSDL file	WEBSERVICE	ENABLED
☐	Store Picker	Store picker locations cache	STOREPICKER	DISABLED

Profiling & Troubleshooting

"Prevention is better than cure"

Hippocrates

But sometimes you need to cure:

- Picking up an existing project
- Using a third-party extension

Profiling

The embedded Magento 2 profiler can be activated via environment variable:

http://devdocs.magento.com/guides/v2.2/config-guide/bootstrap/mage-profiler.html

More advanced options:

- Blackfire - https://blackfire.io/
- Tideways - https://tideways.io/
- Xhprof - https://github.com/preinheimer/xhprof

Troubleshooting

Recommended tool:
https://github.com/magespecialist/m2-MSP_DevTools

Magento 2 module with related Chrome extension that:

- Allows to inspect page elements
- Shows block generation time, cache key and info
- Shows loaded model, to validate caching effectiveness
- Can be used to render the core profiler's output

03. Page Cache

How page cache works and customizations best practices.

Page cache

Options:

- Built-in page cache: doesn't have external dependencies
- Varnish integration: strongly recommended for production

Devdocs section:

http://devdocs.magento.com/guides/v2.2/config-guide/varnish/config-varnish.html

Varnish integration

When the Varnish integration is enabled from the configuration:

- HTTP Cache Control and Expires headers are added to all pages that are cacheable, so Varnish will store and send them directly
- In that case, all the tags used by block in the layout are collected, and added to a header called *X-Magento-Tags*
- Varnish will store the tags as property of the cache entry. At each invalidation by tag, Magento will send a ban to Varnish with the header *X-Magento-Tags-Pattern*, that will be used to delete all cache entries with matching *X-Magento-Tags*

Under the hood

Class *Magento\PageCache\Model\Layout\LayoutPlugin.* Method *afterGenerateXml* calls a method that will add the HTTP Cache Control and Expires headers with the given TTL, if the page is cacheable:

```
if ($subject->isCacheable() && $this->config->isEnabled()) {
    $this->response->setPublicHeaders($this->config->getTtl());
}
```

While *afterGetOutput* gather all the blocks' tags and adds them to the *X-Magento-Tags* header:

```
foreach ($subject->getAllBlocks() as $block) {
    if ($block instanceof \Magento\Framework\DataObject\IdentityInterface) {
        // […] ESI and Varnish checks
        $tags = array_merge($tags, $block->getIdentities());
    }
}
$tags = array_unique($tags);
$this->response->setHeader('X-Magento-Tags', implode(',', $tags));
```

Under the hood

Class *Magento\Framework\View\Layout.* Method *isCacheable* checks for blocks with *cacheable="false"* in the layout, and the di-injected *cacheable* parameter:

```
$cacheableXml = !(bool)count($this->getXml()->xpath(
    '//' . Element::TYPE_BLOCK . '[@cacheable="false"]'));
```

To be used only on uncacheable pages (i.e. cart or checkout)

Devdocs:
http://devdocs.magento.com/guides/v2.2/extension-dev-guide/cache/page-caching.html

Under the hood

Class *Magento\CacheInvalidate\Model\PurgeCache*

Method *sendPurgeRequest* sends a ban request to each configured Varnish server:

```
$headers = [self::HEADER_X_MAGENTO_TAGS_PATTERN => $tagsPattern];
foreach ($servers as $server) {
    $headers['Host'] = $server->getHost();
    try {
        $socketAdapter->connect($server->getHost(), $server->getPort());
        $socketAdapter->write('PURGE', $server, '1.1', $headers);
        $socketAdapter->read();
        $socketAdapter->close();
    } catch (\Exception $e) {
        $this->logger->critical($e->getMessage(), compact('server', 'tagsPattern'));
        return false;
    }
}
```

ESI blocks

ESI stands for Edge Side Includes:
https://varnish-cache.org/docs/4.1/users-guide/esi.html

> *Varnish can create web pages by assembling different pages, called fragments, together into one page. These fragments can have individual cache policies. If you have a web site with a list showing the five most popular articles on your site, this list can probably be cached as a fragment and included in all the other pages.*

On Magento 2, all blocks that have a TTL are replaced with *esi:include* tags.

This means that they can be used for any content that has a lower TTL than the whole page.

Use only when needed: the content of ESI blocks is not included in the initial page request, so each one will cause an additional request from Varnish to Magento.

ESI – Request flow

Client	Varnish	Magento

Initial page request (Client → Varnish)

Request forwarded to Magento (Varnish → Magento)

Magento returns a page with ESIs (Magento → Varnish)

Varnish requests the ESIs' content (Varnish → Magento)

Magento returns the ESIs content (Magento → Varnish)

Varnish composes the page and returns it (Varnish → Client)

Under the hood

Class *Magento\PageCache\Observer\ProcessLayoutRenderElement*

Method *execute* calls *_wrapEsi* when the block has a TTL and Varnish integration is enabled:

```
$blockTtl = $block->getTtl();
if (isset($blockTtl) && $this->isVarnishEnabled()) {
    $output = $this->_wrapEsi($block, $layout);
}
```

Which replaces the block with an *esi:include* tag:

```
$url = $block->getUrl(
    'page_cache/block/esi',
    [
        'blocks' => $this->jsonSerializer->serialize([$block->getNameInLayout()]),
        'handles' => $this->base64jsonSerializer->serialize(
            array_values(array_diff($handles, $pageSpecificHandles))
        )
    ]
);
return sprintf('<esi:include src="%s" />', $url);
```

Content variations

The problem: serving different content to different types of users.

The solutions:

- Load the data from browser storage or API via Javascript/Knockout
  - To be used with private data
  - Used in the core, for instance, to render the minicart
- Serve a different version of the page and cache it separately with the *X-Magento-Vary* cookie
  - To be used with public content
  - Used in the core, for instance, with currency conversions

Under the hood

Class *Magento\Framework\App\Response\Http*

Method *sendVary* sets a cookie based on the vary string:

```
$varyString = $this->context->getVaryString();
if ($varyString) {
    $sensitiveCookMetadata = $this->cookieMetadataFactory->createSensitiveCookieMetadata()
        ->setPath('/');
    $this->cookieManager->setSensitiveCookie(
        self::COOKIE_VARY_STRING, $varyString, $sensitiveCookMetadata);
}
```

Which is generated in *Magento\Framework\App\Http\Context*:

```
$data = $this->getData();
if (!empty($data)) {
    ksort($data);
    return sha1($this->serializer->serialize($data));
}
```

Method *setData* class can be used to add custom variations.

04. Results validation

Evaluate performance improvements in an objective and reliable way.

Results validation

- Requires the correct caching settings, based on the objectives
  - To evaluate the base code, disable all types of cache
  - To evaluate application cache, disable page cache
  - To evaluate hitrate and private content load speed, enable all cache
- Can be a complex task
  - A few manual page loads or a single waterfall test are, usually, not good representations the load time distribution

Useful tools

Siege

https://www.joedog.org/siege-home/

Shows failure rate and average page generation time.

Taurus

http://gettaurus.org/

Based on jMeter, it can run more complex scenarios, verify all in page content and show the data in graphs.

Gatling

https://gatling.io/

Based on jMeter, like Taurus, and allows scenarios do be defined in Scala.

Siege

Benchmark the application for 20 seconds:

```
$ siege –b –t20S https://magento2.dev/

** SIEGE 4.0.4
** Preparing 25 concurrent users for battle.
The server is now under siege...

[...]

Lifting the server siege...
Transactions:             479 hits
Availability:             96.57 %
Elapsed time:             19.82 secs
Data transferred:         10.47 MB
Response time:            1.03 secs
Transaction rate:         24.17 trans/sec
Throughput:               0.53 MB/sec
Concurrency:              24.86
Successful transactions:  471
Failed transactions:      17
Longest transaction:      17.64
Shortest transaction:     0.06
```

Taurus example

test.yml file content:

```
execution:
- concurrency: 2
  hold-for: 30s
  scenario: example-test

scenarios:
 example-test:
   retrieve-resources: true
   requests:
   - https://magento2.dev/
```

Terminal:

$ bzt test.yml

Taurus results example

1 VU	0.41 Hits/s	0 %	2.57 s	2.96 s	864.68 KiB/s
Max Users	Avg. Throughput	Errors	Avg. Response Time	90% Response Time	Avg. Bandwidth

Duration	1 minute	Test Type	External
Started	7:36:24 PM	Response Codes	2xx
Ended	7:37:27 PM		

Load

5 vu
0.25
04/04/2018 - 19:36:00
• Users: **1**
• Hits/s: **0.22**
• Error(s): **0.22**
0
19:36
19:37
Users — Hits/s — Error(s)

Response Time

5 vu
3000 ms
0 ms
19:36
19:37
Users — Response Time

Advanced – Magento Performance Toolkit

Useful to test how much a custom extension impacts on a vanilla installation of Magento.

Further details and usage instructions:
https://github.com/magento/magento2/tree/2.2/setup/performance-toolkit

Provides:

- A command to generate fixtures for a performance test, with different profile sizes (small, medium, large, extra large)
- A set of jMeter scenarios to test and benchmark the core functionalities

Thanks for listening.

Any question?

renato.cason@endclothing.com

@renatocason