Purge

The modular external cache invalidation framework.

The Purge module for Drupal 8 enables invalidation of content from external caches, reverse proxies and CDN platforms. The technology-agnostic plugin architecture allows for different server configurations and use cases. Last but not least, it enforces a separation of concerns and should be seen as a **middleware** solution.

Drush commands

The `purge_drush` module adds the following commands for Drush administration:

Command	Alias	Description
`p-diagnostics`	`pdia`	Generate a diagnostic self-service report.
`p-invalidate`	`pinv`	Directly invalidate an item without going through the queue.
`p-processors`	`ppro`	List all enabled processors.
`p-queue-add`	`pqa`	Schedule an item for later processing.
`p-queue-browse`	`pqb`	Inspect what is in the queue by paging through it.
`p-queue-empty`	`pqe`	Clear the queue and reset all statistics.
`p-queue-stats`	`pqs`	Retrieve the queue statistics.
`p-queue-work`	`pqw`	Claim a chunk of items from the queue and process them.
`p-queuers`	`pqrs`	List all enabled queuers.
`p-types`	`ptyp`	List all supported cache invalidation types.

Several commands understand the `--format` parameter allowing you to integrate the commands in external scripts with JSON or YAML output. See the respective `drush help <command>` information for more command detail.

The framework explained

Purge isn't just a single API but made up of several API pillars all driven by plugins, allowing very flexible end-user setups. All of them are clearly defined to enforce a sustainable and maintainable framework over the longer term. This also allows everyone to build, improve and fix bugs in only the plugins they provide and therefore allows everyone to 'scale up' solving external cache invalidation in the best way possible.

Queuer

With Purge, end users can manually invalidate a page with a Drush command or, theoretically, via a "clear this page" button in the GUI. Caches are however meant to be transparent to end users and to only be invalidated when something actually changed - and thus requires external caches to also be transparent.

When editing content of any kind, Drupal will transparently and efficiently invalidate cached pages in Drupal's own **anonymous page cache**. When Drupal renders a page, it can lists all the rendered items on the page in a special HTTP response header named `X-Drupal-Cache-Tags`. For example, this allows all cached pages with the `node:1` Cache-Tag in their headers to be invalidated, when that particular node (node/1) is changed.

Purge ships with the **Core tags queuer**, which replicates everything Drupal core invalidated onto Purge's queue. So, when Drupal clears rendered items from its own page cache, Purge will add a *invalidation* object to its queue so that it gets cleared remotely as well.

Queue

Queueing is an inevitable and important part of Purge as it makes cache invalidation resilient, stable and accurate. Certain reverse cache systems can clear thousands of items under a second, yet others - for instance CDNs - can demand multi-step purges that can easily take up 30 minutes. Although the queue can technically be left out of the process entirely, it will be required in the majority of use cases.

Statistics tracker

The statistics tracker keeps track of queue activity by actively counting how many items the queue currently holds and how many have been deleted or released back to it. This data can be used to report progress on the queue and is easily retrieved, the data resets when the queue is emptied.

Invalidations

Invalidations are small value objects that **describe and track invalidations** on one or more external caching systems within the Purge pipeline. These objects float freely between **queue** and **purgers** but can also be created on the fly and in third-party code.

Invalidation types

Purge has to be crystal clear about what needs invalidation towards its purgers, and therefore has the concept of invalidation types. Individual purgers declare which types they support and can even declare their own types when that makes sense. Since Drupal invalidates its own caches using cache tags, the `tag` type is the most important one to support in your architecture.

- **`domain`** Invalidates an entire domain name.
- **`everything`** Invalidates everything.
- **`path`** Invalidates by path, e.g. `news/article-1`.
- **`regex`** Invalidates by regular expression, e.g.: `\.(jpg|jpeg|css|js)$`.
- **`tag`** Invalidates by Drupal cache tag, e.g.: `menu:footer`.
- **`url`** Invalidates by URL, e.g. `http://site.com/node/1`.
- **`wildcardpath`** Invalidates by path, e.g. `news/*`.
- **`wildcardurl`** Invalidates by URL, e.g. `http://site.com/node/*`.

Purgers

Purgers do all the hard work of telling external systems what to invalidate and do this in the technically required way, for instance with external API calls, through telnet commands or with specially crafted HTTP requests.

Purge **doesn't ship any purger**, as this is context specific. You could for instance have multiple purgers enabled to both clean a local proxy and a CDN at the same time.

Capacity tracker

The capacity tracker is the central orchestrator between limited system resources and a never-ending queue of cache invalidation items.

The tracker actively tracks how much items are invalidated during Drupal's request lifetime and how much PHP execution time has been spent. With this information it can predict how much processing can happen during the rest of request lifetime. It is able to predict this since the capacity tracker also collects timing estimates from the actual purgers. The intelligence it has is used by the queue service and exceeding the limit isn't possible as the purgers service refuses to operate when the limits are near zero.

Diagnostic checks

External cache invalidation usually depends on many parameters, for instance configuration settings such as hostname or CDN API keys. In order to prevent hard crashes during runtime that affect end-user workflow, Purge allows plugins to write preventive diagnostic checks that can check their configurations and anything else that affects runtime execution. These checks can block all purging but also raise warnings and other diagnostic information. End-users can rely on Drupal's status report page where these checks also bubble up.

Processors

With queuers adding `tag` invalidation objects to the queue, this still leaves the processing of it open. Since different use cases are possible, it is up to you to configure a stable processing policy that's suitable for your use case.

Possibilities:

- **`cron`** claims items from the queue & purges during cron.
- **`ajaxui`** AJAX-based progress bar working the queue after a piece of content has been updated.
- **`lateruntime`** purges items from the queue on every request (**SLOW**).

API examples

Queueing

Adding invalidations to the queue is the simplest use case and requires a queuer object so that the queue knows who is adding the given items.

```
$purgeInvalidationFactory = \Drupal::service(&#39;purge.invalidation.factory&#39;);
$purgeQueuers = \Drupal::service(&#39;purge.queuers&#39;);
$purgeQueue = \Drupal::service(&#39;purge.queue&#39;);

$queuer = $purgeQueuers->get(&#39;myqueuer&#39;);
$invalidations = [
  $purgeInvalidationFactory->get(&#39;tag&#39;, &#39;node:1&#39;),
  $purgeInvalidationFactory->get(&#39;tag&#39;, &#39;node:2&#39;),
  $purgeInvalidationFactory->get(&#39;path&#39;, &#39;contact&#39;),
  $purgeInvalidationFactory->get(&#39;wildcardpath&#39;, &#39;news/*&#39;),
];

$purgeQueue->add($queuer, $invalidations);
```

What happens now depends on the **processors you configured**, as some might purge very quickly after adding items to the queue whereas others might need a time-based delay before this occurs. Items enter the queue in state `FRESH` and normally leave the processor in the states `SUCCEEDED`, `FAILED`, `PROCESSING` or when no single plugins supported it: `NOT_SUPPORTED`. Items that don't succeed, cycle back to the queue until it gets manually cleared.

Invalidation without queue

Processing invalidations without going through the queue is possible, but not the recommended workflow when your invalidations cannot fail. All it takes is to instantiate invalidation objects and to feed them to the purgers service.

```
use Drupal\purge\Plugin\Purge\Purger\Exception\CapacityException;
use Drupal\purge\Plugin\Purge\Purger\Exception\DiagnosticsException;
use Drupal\purge\Plugin\Purge\Purger\Exception\LockException;
$purgeInvalidationFactory = \Drupal::service(&#39;purge.invalidation.factory&#39;);
$purgeProcessors = \Drupal::service(&#39;purge.processors&#39;);
$purgePurgers = \Drupal::service(&#39;purge.purgers&#39;);

$processor = $purgeProcessors->get(&#39;myprocessor&#39;);
$invalidations = [
  $purgeInvalidationFactory->get(&#39;tag&#39;, &#39;node:1&#39;),
  $purgeInvalidationFactory->get(&#39;tag&#39;, &#39;node:2&#39;),
  $purgeInvalidationFactory->get(&#39;path&#39;, &#39;contact&#39;),
  $purgeInvalidationFactory->get(&#39;wildcardpath&#39;, &#39;news/*&#39;),
];

try {
  $purgePurgers->invalidate($processor, $invalidations);
}
catch (DiagnosticsException $e) {
  // Diagnostic exceptions happen when the system cannot purge.
}
catch (CapacityException $e) {
  // Capacity exceptions happen when too much was purged during this request.
}
catch (LockException $e) {
  // Lock exceptions happen when another code path is currently processing.
}
```

When this code finished successfully, the `$invalidations` array holds the objects it had before, but now each object has changed its state. You can now verify this by iterating over the objects and by calling `getState` or `getStateString()` on them (the latter is only intended for UI presentation):

```
foreach ($invalidations as $invalidation) {
  var_dump($invalidation->getStateString());
}
```

Which could then look like this:

```
string(6) "FAILED"
string(6) "FAILED"
string(9) "SUCCEEDED"
string(10) "PROCESSING"
```

The results reveal why you should **normally not invalidate without going through the queue**, because items can fail or need to run again later to finish entirely. The most common use case for direct invalidation is manual UI purging.

Queue processing

Processing items from the queue is handled by processors, which users can add and configure according to their configuration. In essence, processors invoke the following code to retrieve a dynamically calculated chunk of items from the queue and feed those to the purgers service:

```
use Drupal\purge\Plugin\Purge\Purger\Exception\CapacityException;
use Drupal\purge\Plugin\Purge\Purger\Exception\DiagnosticsException;
use Drupal\purge\Plugin\Purge\Purger\Exception\LockException;
$purgePurgers = \Drupal::service(&#39;purge.purgers&#39;);
```

```
$purgeProcessors = \Drupal::service(&amp;#39;purge.processors&amp;#39;);
$purgeQueue = \Drupal::service(&amp;#39;purge.queue&amp;#39;);

$claims = $purgeQueue->claim();
$processor = $purgeProcessors->get(&amp;#39;myprocessor&amp;#39;);
try {
  $purgePurgers->invalidate($processor, $claims);
}
catch (DiagnosticsException $e) {
  // Diagnostic exceptions happen when the system cannot purge.
}
catch (CapacityException $e) {
  // Capacity exceptions happen when too much was purged during this request.
}
catch (LockException $e) {
  // Lock exceptions happen when another code path is currently processing.
}
finally {
  $purgeQueue->handleResults($claims);
}
```