Caching and cache tags

TABLE OF CONTENTS

How to uncache a particular page or node

This will cause Drupal to rebuild the page internally, but won't stop browsers or CDN's from caching.

```
\Drupal::service('page_cache_kill_switch')->trigger();
```

Use this statement in node_preprocess, controller, etc.

Create a custom module to implement setting max-age to 0. For example in ddd.module file:

```
use Drupal\Core\Entity\EntityInterface;
use Drupal\Core\Entity\Display\EntityViewDisplayInterface;

function ddd_node_view_alter(array &$build, EntityInterface $entity, EntityViewDisplayInterface $display) {
  $bundle = $entity->bundle();
  if ($bundle == 'search_home') {
    $build['#cache']['max-age'] = 0;
    \Drupal::service('page_cache_kill_switch')->trigger();
  }
}
```

Don't cache data returned from a controller

From dev1/web/modules/custom/rsvp/src/Controller/ReportController.php

```
  // Don't cache this page.
  $content['#cache']['max-age'] = 0;
  return $content;
}
```

Disable caching for a route in the module.routing.yml file.

```
requirements:
  _permission: 'access content'
options:
  no_cache: TRUE
```

Disable caching for a content type

If someone tries to view a node of content type *search_home* (i.e. an entity of bundle *search_home*) caching is disabled and Drupal and the browser will always re-render the page. This is necessary for a page that is retrieving data from a third party source and you almost always expect it to be different. It wouldn't work for a search page to show results from a previous search.

```
use Drupal\Core\Entity\EntityInterface;
use Drupal\Core\Entity\Display\EntityViewDisplayInterface;

/**
 * Implements hook_ENTITY_TYPE_view_alter().
 */
function ddd_node_view_alter(array &$build, EntityInterface $entity, EntityViewDisplayInterface $display) {
  $bundle = $entity->bundle();
  if ($bundle == 'search_home') {
    $build['#cache']['max-age'] = 0;
    \Drupal::service('page_cache_kill_switch')->trigger();
  }
}
```

Considering caching when retrieving query, get or post parameters

For get variables use:

```
$query = \Drupal::request()->query->get('name');
```

For post variables use:

```
$name = \Drupal::request()->request->get('name');
```

For all items in get:

```
$query = \Drupal::request()->query->all();
$search_term = $query['query'];
$collection = $query['collection'];
```

Be wary about caching. From https://drupal.stackexchange.com/questions/231953/get-in-drupal-8/231954#231954 the code provided only works the first time so it is important to add a '#cache' context in the markup.

```
namespace Drupal\newday\Controller;

use Drupal\Core\Controller\ControllerBase;

class NewdayController extends ControllerBase {
    public function new() {
      $day= [
        "#markup" => \Drupal::request()->query->get('id'),
       ];
      return $day;
    }
}
```

The request is being cached, you need to tell the system to vary by the query argument:

```
$day = [
    '#markup' => \Drupal::request()->query->get('id'),
    '#cache' => [
        'contexts' => ['url.query_args:id'],
    ],
];
```

More about caching render arrays at https://www.drupal.org/docs/8/api/render-api/cacheability-of-render-arrays

Debugging Cache tags

In development.services.yml set these parameters

```
parameters:
  http.response.debug_cacheability_headers: true
```

in Chrome, the network tab, click on the doc and view the Headers. You will see the following two headers showing both the cache contexts and the cache tags

1 **X-Drupal-Cache-Contexts:**

> languages:language_interface route session theme timezone url.path url.query_args url.site user

1 **X-Drupal-Cache-Tags:**

> block_view config:block.block.bartik_account_menu config:block.block.bartik_branding config:block.block.bartik_breadcrumbs config:block.block.bartik_content config:block.block.bartik_footer config:block.block.bartik_help config:block.block.bartik_local_actions config:block.block.bartik_local_tasks config:block.block.bartik_main_menu config:block.block.bartik_messages config:block.block.bartik_page_title config:block.block.bartik_powered config:block.block.bartik_search config:block.block.bartik_tools config:block.block.helloworldsalutation config:block.block.modalblock config:block.block.productimagegallery config:block.block.rsvpblock config:block.block.views_block__aquifer_listing_block_1 config:block.block.views_block__related_videos_block_1 config:block.block.views_block__user_guide_pages_referencing_a_product_block_1 config:block.block.views_block__workshop_count_proposed_workshop_block config:bloc

Using cache tags

If you are generating a list of cached node teasers and you want to make sure your list is always accurate, use cache tags. To refresh the list every time a node is added, deleted or edited you could use a render array like this:

```
$build = [
  '#type' => 'markup',
  '#markup' => $sMarkup,
  '#cache' => [
    'keys' => ['home-all','home'],
    'tags'=> ['node_list'], // invalidate cache when any node content is added/changed etc.
    'max-age' => '36600', // invalidate cache after 10h
  ],
];
```

It is possible to change this so the cache is invalidated only when a content type of *book* or *magazine* is changed in two possible ways:

1. Include all node tags (node:{#id}), if doesn't matter if a new node of a particular type was added.
2. Create and control your own cache tag, and invalidate it when you want.

If you want a block to be rebuilt every time that a term from a particular vocab_id is added, changed, or deleted you can cache the term list. If you need to cache a term list per vocab_id - i.e. every time that a term from a particular vocab_id is added, changed, or deleted the cache tag is invalided using `Cache::invalidateTags($tag_id)` then my block will be rebuild.

```
use Drupal\Core\Cache\Cache;

function filters_invalidate_vocabulary_cache_tag($vocab_id) {
  Cache::invalidateTags(['filters_vocabulary:' . $vocab_id]);
}
```

If you want this to work for nodes, you may be able to just change `$vocab_id` for `$node_type`.

Setting cache keys in a block

If you add some code to a block that includes the logged in user's name, you may find that the username will not be displayed correctly – rather it may show the prior users name. This is because the cache context of user doesn't bubble up to the display of the container (e.g. the node that is displayed along with your custom block.) Add this to bubble the cache contexts up.

```
public function getCacheContexts() {
  Return Cache::mergeContexts(parent::getCacheContexts(),['user']);
}
```

and scrolling down a bit at this link shows some more info about getting cache tags and merging them.
https://drupal.stackexchange.com/questions/145823/how-do-i-get-the-current-node-id

Getting Cache Tags and Contexts for a block

In this file /modules/custom/dart_pagination/src/Plugin/Block/VideoPaginationBlock.php I have a block that renders a form. The form queries some data from the database and will need to be updated depending on the node that I am on.

I added the following two functions:

```
public function getCacheTags() {
  //When my node changes my block will rebuild
  if ($node = \Drupal::routeMatch()->getParameter('node')) {
    //if there is node add its cachetag
    return Cache::mergeTags(parent::getCacheTags(), ['node:' . $node->id()]);
  } else {
    //Return default tags instead.
    return parent::getCacheTags();
  }
}

public function getCacheContexts() {
  //if you depend on \Drupal::routeMatch()
  //you must set context of this block with 'route' context tag.
  //Every new route this block will rebuild
  return Cache::mergeContexts(parent::getCacheContexts(), ['route']);
}
```

Caching REST Resources

Interesting article about caching REST resources at http://blog.dcycle.com/blog/2018-01-24/caching-drupal-8-rest-resource/

We can get Drupal to cache our rest resource e.g. in dev1 /custom/iai_wea/src/Plugin/rest/resource/WEAResource.php where we add this to our response:

```
if (!empty($record)) {
  $response = new ResourceResponse($record, 200);
  $response->addCacheableDependency($record);
  return $response;
}
```

Caching in an API class wrapper

From docroot/modules/custom/cm_api/src/CmAPIClient.php

Here a member is set up in the class

```
/**
 * Custom service to call APIs.
 *
 * @see \Drupal|cm_api|CmAPIClientInterface
 */
class CmAPIClient implements CmAPIClientInterface {
...
/**
 * Internal static cache.
 *
 * @var array
 */
protected static $cache = [];
```

The api call is made and the cache is checked. The index (or key) is built from the api call “getPolicy” and the next key is the policy number with the version number attached. So the $response_data is put in the cache with:

```
self::$cache['getPolicy'][$policy_number . $version] = $response_data;
```

and retrieved with:

```
$response_data = self::$cache['getPolicy'][$policy_number . $version];
```

This relieves the back end load by rather getting the data from the cache if is in the cache. (If the cache is warm.)

The entire function is shown below. It is from docroot/modules/custom/cm_api/src/CmAPIClient.php:

```
public function getPolicy($policy_number, $version = 'v2') {
  // Api action type.
  $this->params['api_action'] = 'Get policy';
  // Add policy number to display in watchdog.
  $this->params['policynumber'] = $policy_number;
  $base_api_url = $this->getBaseApiUrl();
  if (empty($policy_number) || !is_numeric($policy_number)) {
    $this->logger->get('cm_api_get_policy')
      ->error('Policy number must be a number.');
    return FALSE;
  }
  $endpoint_url = $base_api_url . '/' . $version . '/Policies/group?policies=' . $policy_number . '&include_ref=false&include_hist=true';

  if (isset(self::$cache['getPolicy'][$policy_number . $version])) {
    $response_data = self::$cache['getPolicy'][$policy_number . $version];
  } else {
    $response_data = $this->performRequest($endpoint_url, 'GET', $this->params);
    self::$cache['getPolicy'][$policy_number . $version] = $response_data;
  }
  return $response_data;
}
```

Caching in a .module file

From docroot/modules/custom/ncs_infoconnect/nzz_zzzzconnect.module.

In the hook_preprocess_node function, we are calling an api to get some data.

```
/**
 * Implements hook_preprocess_node().
 */
function nzz_zzzzconnect_preprocess_node(&$variables) {
```

Notice the references to \Drupal::cache(). First we check if this is our kind of node to process. Then we derive the $cid. We check the cache with a call to ->get($cid) and if it fails we:

1. call the api with $client->request('GET')
2. pull out the body with $response->getBody()
3. set the whole body into the cache with:

```
\Drupal::cache()->set($cid, $contents, REQUEST_TIME + (300));
```

In future requests, we can just use the data from the cache.

```
if ($node_type == 'zzzzfeed' && $published) {

  $uuid = $variables['node']->field_uuid->getValue();
  $nid = $variables['node']->id();

  $nzz_auth_settings = Settings::get('nzz_api_auth', []);
  $uri = $ncs_auth_settings['default']['server'] . ':' . $ncs_auth_settings['default']['port'];
  $uri .= '/blahcontent/search';
  $client = \Drupal::httpClient();

  $cid = 'zzzzfeed-' . $nid;
  try {
    if ($cache = \Drupal::cache()->get($cid)) {
      $contents = $cache->data;
    }
    else {
      $response = $client->request('GET', $uri, [
        'auth' => [$nzz_auth_settings['default']['username'], $nzz_auth_settings['default']['password']],
        'query' => [
          'uuid' => $uuid[0]['value'],
        ],
        'timeout' => 1,
      ]);
      $contents = $response->getBody()->getContents();
      \Drupal::cache()->set($cid, $contents, REQUEST_TIME + (300));
    }
  }
  catch (RequestException $e) {
    watchdog_exception('nzz_zzzzconnect', $e);
    return FALSE;
  }
  catch (ClientException $e) {
    watchdog_exception('nzz_zzzzconnect', $e);
    return FALSE;
  }

  $contents = json_decode($contents, TRUE);
  $body = $contents['hits']['hits'][0]['versions'][0]['properties']['Text'][0];
  $variables['content']['body'] = [
    '#markup' => $body,
  ];
}
```

Logic for caching render arrays

From https://www.drupal.org/docs/8/api/render-api/cacheability-of-render-arrays

Whenever you are generating a render array, use the following 5 steps:

1. I'm rendering something. That means I must think of cacheability.
2. Is this something that's expensive to render, and therefore is worth caching? If the answer is yes, then what identifies this particular representation of the thing I'm rendering? Those are the cache keys.
3. Does the representation of the thing I'm rendering vary per combination of permissions, per URL, per interface language, per … something? Those are the cache contexts. Note: cache contexts are completely analogous to HTTP's Vary header.
4. What causes the representation of the thing I'm rendering become outdated? I.e., which things does it depend upon, so that when those things change, so should my representation? Those are the cache tags.
5. When does the representation of the thing I'm rendering become outdated? I.e., is the data valid for a limited period of time only? That is

the max-age (maximum age). It defaults to “permanently (forever) cacheable” (Cache::PERMANENT). When the representation is only valid for a limited time, set a max-age, expressed in seconds. Zero means that it’s not cacheable at all.

Cache contexts, tags and max-age must always be set, because they affect the cacheability of the entire response. Therefore they “bubble” and parents automatically receive them.

Cache keys must only be set if the render array should be cached.

There are more details at the link above

Development Setup

Disable caching and enable TWIG debugging

Generally I enable twig debugging and disable caching while developing a site.

To enable TWIG debugging output in source, in sites/default/development.services.yml set twig.config debug:true. See core.services.yml for lots of other items to change for development

```
# Local development services.
#
# To activate this feature, follow the instructions at the top of the
# 'example.settings.local.php' file, which sits next to this file.
parameters:
  http.response.debug_cacheability_headers: true
  dino.roar.use_key_value_cache: true
  twig.config:
    debug: true
    auto_reload: true
    cache: false

# To disable caching, you need this and a few other items
services:
  cache.backend.null:
    class: Drupal\Core\Cache\NullBackendFactory
```

to enable put this in settings.local.php:

```
/**
 * Enable local development services.
 */
$settings['container_yamls'][] = DRUPAL_ROOT . '/sites/development.services.yml';
```

You also need to disable the render cache in settings.local.php with:

```
$settings['cache']['bins']['render'] = 'cache.backend.null';
```

Disable Cache for development

From https://www.drupal.org/node/2598914

1. Copy, rename, and move the sites/example.settings.local.php to sites/default/settings.local.php with:

```
$ cp sites/example.settings.local.php sites/default/settings.local.php
```

1. Edit sites/default/settings.php and uncomment these lines:

```
if (file_exists($app_root . '/' . $site_path . '/settings.local.php')) {
  include $app_root . '/' . $site_path . '/settings.local.php';
}
```

This will include the local settings file as part of Drupal's settings file.

1. In settings.local.php make sure development.services.yml is enabled with:

```
$settings['container_yamls'][] = DRUPAL_ROOT . '/sites/development.services.yml';
```

By default development.services.yml contains the settings to disable Drupal caching:

```
services:
  cache.backend.null:
    class: Drupal\Core\Cache\NullBackendFactory
```

NOTE: Do not create development.services.yml, it already exists under /sites. You can copy it from there.

1. In settings.local.php change the following to be TRUE if you want to work with enabled css- and js-aggregation:

```
$config['system.performance']['css']['preprocess'] = FALSE;
$config['system.performance']['js']['preprocess'] = FALSE;
```

1. Uncomment these lines in settings.local.php to disable the render cache and disable dynamic page cache:

```
$settings['cache']['bins']['render'] = 'cache.backend.null';
$settings['cache']['bins']['dynamic_page_cache'] = 'cache.backend.null';
$settings['cache']['bins']['page'] = 'cache.backend.null';
```

If you do not want to install test modules and themes, set the following to FALSE:

```
$settings['extension_discovery_scan_tests'] = FALSE;
```

1. In sites/development.services.yml add the following block to disable the twig cache:

```
parameters:
  twig.config:
    debug: true
    auto_reload: true
    cache: false
```

NOTE: If the parameters block is already present in the yml file, append the twig.config block to it.

Afterwards rebuild the Drupal cache with drush cr otherwise your website will encounter an unexpected error on page reload.

How to specify the cache backend

This information is relevant for using Memcache, Redis and also APCu. By default, Drupal caches information in the database. Tables includes cache_default, cache_render, cache_page, cache_config etc. By using the configuration below, Drupal can instead store this info in memory to increase performance.

Summary

Drupal will no longer automatically use the custom global cache backend specified in $settings['cache']['default'] in settings.php on certain specific cache bins that define their own default_backend in their service definition. In order to override the default backend, a line must be added explicitly to settings.php for each specific bin that provides a default_backend. This change has no effect for users that do not use a custom cache backend configuration like Redis or Memcache, and makes it possible to remove workarounds that were previously necessary to keep using the default fast chained backend for some cache bins defined in Drupal core.

Detailed description with examples

In Drupal 8 there are several ways to specify which cache backend is used for a certain cache bin (e.g. the discovery cache bin or the render cache bin).

In Drupal, cache bins are defined as services and are tagged with name: cache.bin. Additionally, some cache bins specify a default_backend service within the tags. For example, the discovery cache bin from Drupal core defines a fast chained default backend:

```
  cache.discovery:
    class: Drupal\Core\Cache\CacheBackendInterface
    tags:
      - { name: cache.bin, default_backend: cache.backend.chainedfast }
    factory: cache_factory:get
    arguments: [discovery]
```

Independent of this, the $settings array can be used in `settings.php` to assign cache backends to cache bins. For example:

```
$settings['cache']['bins']['discovery'] = 'cache.backend.memory';
$settings['cache']['default'] = 'cache.backend.redis';
```

Before Drupal 8.2.0, the order of steps through which the backend was selected for a given cache bin was as follows:

- First look for a specific bin definition in settings. E.g., `$settings['cache']['bins']['discovery']`
- If not found, then use the global default defined in settings. I.e., `$settings['cache']['default']`
- If a global default is not defined in settings, then use the default_backend from the tag in the service definition.

This was changed to:

- First look for a specific bin definition in settings. E.g., `$settings['cache']['bins']['discovery']`
- If not found, then use the the `default_backend` from the tag in the service definition.
- If no `default_backend` for the specific bin was provided, then use the global default defined in settings. I.e., `$settings['cache']['default']` The old order resulted in unexpected behaviors, for example, the fast chained backend was no longer used when an alternative cache backend was set as default.

The order has been changed, so that the cache bin services that explicitly set `default_backends` are always used unless explicitly overridden with a per-bin configuration. In core, this means, fast chained backend will be used for `bootstrap`, `config`, and `discovery` cache bins and `memory` backend will be used for the `static` cache bin, unless they are explicitly overridden in settings.

For example, to ensure Redis is used for all cache bins, before 8.2.0, the following configuration would have been enough:

```
$settings['cache']['default'] = 'cache.backend.redis';
```

However, now the following configuration in settings.php would be required to achieve the same exact behavior:

```
$settings['cache']['bins']['bootstrap'] = 'cache.backend.redis';
$settings['cache']['bins']['discovery'] = 'cache.backend.redis';
$settings['cache']['bins']['config'] = 'cache.backend.redis';
$settings['cache']['bins']['static'] = 'cache.backend.redis';
$settings['cache']['default'] = 'cache.backend.redis';
```

Before proceeding to override the cache bins that define fast cached default backends blindly, please also read why they exist, particularly when using multiple webserver nodes. See ChainedFastBackend on api.drupal.org.

The above information is from https://www.drupal.org/node/2754947

Fabian Franz in his article at https://drupalsun.com/fabianx/2015/12/01/day-1-tweak-drupal-8-performance-use-apcu-24-days-performance-goodies suggests that we can configure APCu to be used for caches with the following:

```
$settings['cache']['default'] = 'cache.backend.apcu';
$settings['cache']['bins']['bootstrap'] = 'cache.backend.apcu';
$settings['cache']['bins']['config'] = 'cache.backend.apcu';
$settings['cache']['bins']['discovery'] = 'cache.backend.apcu';
```

> **WARNING**
> Proceed with caution with the above as it seems that APCu may only suitable for single server setups. TODO: I couldn't find any references to using APCu with multi-server setups so I'm not sure if that is a safe configuration.

Pantheon docs ask in their FAQ Can APCu be used as a cache backend on Pantheon? Yes, APCu can be used as a cache backend or a “key-value store”; however, this is not recommended. APCu lacks the ability to span multiple application containers. Instead, Pantheon provides a Redis-based Object Cache as a caching backend for Drupal and WordPress, which has coherence across multiple application containers. This was from Pantheon docs FAQ’s:

Drupal 8 has a so-called fast-chained backend as the default cache backend, which allows to store data directly on the web server while ensuring it is correctly synchronized across multiple servers. APCu is the user cache portion of APC (Advanced PHP Cache), which has served us well till PHP 5.5 got its own zend opcache. You can think of it as a key-value store that is stored in memory and the basic operations are apc_store($key, $data), apc_fetch($keys) and apc_delete($keys). For windows the equivalent on IIS would be WinCache (http://drupal.org/project/wincache).

class ChainedFastBackend

Defines a backend with a fast and a consistent backend chain.

In order to mitigate a network roundtrip for each cache get operation, this cache allows a fast backend to be put in front of a slow(er) backend. Typically the fast backend will be something like APCu, and be bound to a single web node, and will not require a network round trip to fetch a cache item. The fast backend will also typically be inconsistent (will only see changes from one web node). The slower backend will be something like Mysql, Memcached or Redis, and will be used by all web nodes, thus making it consistent, but also require a network round trip for each cache get.

In addition to being useful for sites running on multiple web nodes, this backend can also be useful for sites running on a single web node where the fast backend (e.g., APCu) isn’t shareable between the web and CLI processes. Single-node configurations that don’t have that limitation can just use the fast cache backend directly.

We always use the fast backend when reading (get()) entries from cache, but check whether they were created before the last write (set()) to this (chained) cache backend. Those cache entries that were created before the last write are discarded, but we use their cache IDs to then read them from the consistent (slower) cache backend instead; at the same time we update the fast cache backend so that the next read will hit the faster backend again. Hence we can guarantee that the cache entries we return are all up-to-date, and maximally exploit the faster cache backend. This cache backend uses and maintains a “last write timestamp” to determine which cache entries should be discarded.

Because this backend will mark all the cache entries in a bin as out-dated for each write to a bin, it is best suited to bins with fewer changes.

Note that this is designed specifically for combining a fast inconsistent cache backend with a slower consistent cache back-end. To still function correctly, it needs to do a consistency check (see the “last write timestamp” logic). This contrasts with \Drupal\Core\Cache\BackendChain, which assumes both chained cache backends are consistent, thus a consistency check being pointless. This information is from https://api.drupal.org/api/drupal/core%21lib%21Drupal%21Core%21Cache%21ChainedFastBackend.php/class/ChainedFastBackend/9

APCu

APCu is the official replacement for the outdated APC extension. APC provided both opcode caching (opcache) and object caching. As PHP versions 5.5 and above include their own opcache, APC was no longer compatible, and its opcache functionality became useless. The developers of APC then created APCu, which offers only the object caching (read “in memory data caching”) functionality (they removed the outdated opcache). Read more at https://www.php.net/manual/en/book.apcu.php

> **NOTE**
> APCu is not the same as apc!

APCu support is built into Drupal Core. From this Change record Sep 2014:

In order to improve cache performance, Drupal 8 now has:

A cache.backend.apcu service that site administrators can assign as the backend of a cache bin via $settings[‘cache’] in settings.php for sites running on a single server, with a PHP installation that has APCu enabled, and that do not use Drush or other command line scripts.

> **WARNING**
> This references single-server sites not needing Drush. TODO: I couldn’t find any references to using APCu with multi-server setups so I’m not sure if that is a safe configuration.

A cache.backend.chainedfast service that combines APCu availability detection, APCu front caching, and cross-server / cross-process consistency management via chaining to a secondary backend (either the database or whatever is configured for $settings[‘cache’][‘default’]).

A `default_backend` service tag (the value of which can be set to a backend service name, such as cache.backend.chainedfast) that module developers can assign to cache bin services to identify bins that are good candidates for specialized cache backends.

The above tag assigned to the `cache.bootstrap`, `cache.config`, and `cache.discovery` bin services.

This means that by default (on a site with nothing set for $settings['cache'] in settings.php), the bootstrap, config, and discovery cache bins automatically benefit from APCu caching if APCu is available, and this is compatible with Drush usage (e.g., Drush can be used to clear caches and the web process receives that cache clear) and multi-server deployments.

APCu will act as a very fast local cache for all requests. Other cache backends can act as bigger, more general cache backend that is consistent across processes or servers.

For module developers creating custom cache bins

If you are defining a cache bin that is:

- relatively small (likely to have few enough entries to fit within APCu memory), and
- high-read (many cache gets per request, so reducing traffic to the database or other networked backend is worthwhile), and
- low-write (because every write to the bin will invalidate the entire APCu cache of that bin)

then, you can add the default_backend tag to your bin, like so:

```
#example.services.yml
services:
  cache.example:
    class: Drupal\Core\Cache\CacheBackendInterface
    tags:
      - { name: cache.bin, default_backend: cache.backend.chainedfast }
    factory_method: get
    factory_service: cache_factory
    arguments: [example]
```

For site administrators customizing $settings['cache'] Any entry for $settings['cache']['default'] takes precedence over the default_backend service tag values, so you can disable all APCu caching by setting $settings['cache']['default'] = 'cache.backend.database'. If you have $settings['cache']['default'] set to some alternate backend (e.g., memcache), but would still like to benefit from APCu front caching of some bins, you can add those assignments, like so:

```
$settings['cache']['bins']['default'] = 'cache.backend.memcache';
$settings['cache']['bins']['bootstrap'] = 'cache.backend.chainedfast';
$settings['cache']['bins']['config'] = 'cache.backend.chainedfast';
$settings['cache']['bins']['discovery'] = 'cache.backend.chainedfast';
// ...
```

The bins set to use cache.backend.chainedfast will use APCu as the front cache to the default backend (e.g., memcache in the above example).

For site administrators of single-server sites that don't need Drush or other CLI access

> **WARNING**
> This references single-server sites not needing Drush. TODO: I couldn't find any references to using APCu with multi-server setups so I'm not sure if that is a safe configuration.

Pantheon docs ask in their FAQ Can APCu be used as a cache backend on Pantheon? Yes, APCu can be used as a cache backend or a "key-value store"; however, this is not recommended. APCu lacks the ability to span multiple application containers. Instead, Pantheon provides a Redis-based Object Cache as a caching backend for Drupal and WordPress, which has coherence across multiple application containers. This was from Pantheon docs FAQ's:

You can optimize further by using APCu exclusively for certain bins, like so:

```
$settings['cache']['bins']['bootstrap'] = 'cache.backend.apcu';
$settings['cache']['bins']['config'] = 'cache.backend.apcu';
$settings['cache']['bins']['discovery'] = 'cache.backend.apcu';
```

For site administrators wanting a different front cache than APCu

You can copy the cache.backend.chainedfast service definition from core.services.yml to sites/default/services.yml and add arguments to it. For example:

```
#services.yml
services:
  cache.backend.chainedfast:
    class: Drupal\Core\Cache\ChainedFastBackendFactory
    arguments: ['@settings', , 'cache.backend.eaccelerator']
    calls:
      - [setContainer, ['@service_container']]
```

Reference

- Drupal: cache tags for all, regardles of your backend From Matt Glaman 22, August 2022 https://mglaman.dev/blog/drupal-cache-tags-all-regardless-your-backend
- Debugging your render cacheable metadata in Drupal From Matt Glaman 14, February 2023 https://mglaman.dev/blog/debugging-your-render-cacheable-metadata-drupal
- Cache contexts overview on drupal.org https://www.drupal.org/docs/drupal-apis/cache-api/cache-contexts
- Caching in Drupal 8 a quick overview of Cache tags, cache context and cache max-age with simple examples https://zu.com/articles/caching-drupal-8
- Nedcamp video on caching by Kelly Lucas from November 2018 https://www.youtube.com/watch?v=QCZe2K13bd0&list=PLgfWMnl57dv5KmHaK4AngrQAryjO_ylaM&t=0s&index=16
- #! code: Drupal 9: Debugging Cache Problems With The Cache Review Module, September 2022 https://www.hashbangcode.com/article/drupal-9-debugging-cache-problems-cache-review-module
- #! code: Drupal 9: Using The Caching API To Store Data, April 2022 https://www.hashbangcode.com/article/drupal-9-using-caching-api-store-data
- #! code: Drupal 8: Custom Cache Bin, September 2019 https://www.hashbangcode.com/article/drupal-8-custom-cache-bins
- New cache backend configuration order, per-bin default before default configuration (How to specify cache backend), June 2016 https://www.drupal.org/node/2754947
- Cache API Drupal Core

Back to top

Page last modified: May 31 2023.

Edit this page on GitHub

This site uses Just the Docs, a documentation theme for Jekyll.