**Cache API** <https://drupal.org/node/145279>

| [**Name[sort descending](https://api.drupal.org/api/drupal/includes%21cache.inc/7.x?order=title&sort=desc)**](https://api.drupal.org/api/drupal/includes%21cache.inc/7.x?order=title&sort=desc) | **Description** |
| --- | --- |
| [**cache\_clear\_all**](https://api.drupal.org/api/drupal/includes%21cache.inc/function/cache_clear_all/7.x) | **Expires data from the cache.** |
| [**cache\_get**](https://api.drupal.org/api/drupal/includes%21cache.inc/function/cache_get/7.x) | **Returns data from the persistent cache.** |
| [**cache\_get\_multiple**](https://api.drupal.org/api/drupal/includes%21cache.inc/function/cache_get_multiple/7.x) | **Returns data from the persistent cache when given an array of cache IDs.** |
| [**cache\_is\_empty**](https://api.drupal.org/api/drupal/includes%21cache.inc/function/cache_is_empty/7.x) | **Checks if a cache bin is empty.** |
| [**cache\_set**](https://api.drupal.org/api/drupal/includes%21cache.inc/function/cache_set/7.x) | **Stores data in the persistent cache.** |
| [**\_cache\_get\_object**](https://api.drupal.org/api/drupal/includes%21cache.inc/function/_cache_get_object/7.x) | **Gets the cache object for a cache bin.** |

**cache\_set($cid, $data, $bin = 'cache', $expire = CACHE\_PERMANENT)**

**Stores data in the persistent cache**

The persistent cache is split up into several cache bins. In the default cache implementation, **each cache bin corresponds to a database table by the same name**. Other implementations might want to store several bins in data structures that get flushed together. While it is not a problem for most cache bins if the entries in them are flushed before their expire time, some might break functionality or are extremely expensive to recalculate. The other bins are expired automatically by core. **Contributed modules can add additional bins and get them expired automatically by implementing hook\_flush\_caches().**

**The reasons for having several bins are as follows:**

Smaller bins mean smaller database tables and allow for faster selects and inserts.

We try to put fast changing cache items and rather static ones into different bins. The effect is that only the fast changing bins will need a lot of writes to disk. The more static bins will also be better cacheable with MySQL's query cache.

Parameters

**$cid: The cache ID of the data to store.**

**$data: The data to store in the cache.** Complex data types will be automatically serialized before insertion. Strings will be stored as plain text and are not serialized. Some storage engines only allow objects up to a maximum of 1MB in size to be stored by default. When caching large arrays or similar, take care to ensure $data does not exceed this size.

**$bin: (optional)** The cache bin to store the data in. Valid core values are:

**cache: (default) Generic cache storage bin** (used for theme registry, locale date, list of simpletest tests, etc.).

**cache\_block: Stores the content of various blocks.**

**cache\_bootstrap: Stores the class registry, the system list of modules, the list of which modules implement which hooks, and the Drupal variable list.**

**cache\_field: Stores the field data belonging to a given object.**

**cache\_filter: Stores filtered pieces of content.**

**cache\_form: Stores multistep forms.** Flushing this bin means that some forms displayed to users lose their state and the data already submitted to them. This bin should not be flushed before its expired time.

**cache\_menu: Stores the structure of visible navigation menus per page.**

**cache\_page: Stores generated pages for anonymous users.** It is flushed very often, whenever a page changes, at least for every node and comment submission. **This is the only bin affected by the page cache setting on the administrator panel.**

**cache\_path: Stores the system paths that have an alias.**

**$expire:** (optional) One of the following values:

**CACHE\_PERMANENT**: Indicates that the item should never be removed unless explicitly told to using cache\_clear\_all() with a cache ID.

**CACHE\_TEMPORARY**: Indicates that the item should be removed at the next general cache wipe.

**A Unix timestamp**: Indicates that the item should be kept at least until the given time, after which it behaves like CACHE\_TEMPORARY.

**cache\_get($cid, $bin = 'cache')**

Returns data from the persistent cache.

Data may be stored as either plain text or as serialized data. cache\_get will automatically return unserialized objects and arrays.

**Parameters**

**$cid: The cache ID of the data to retrieve.**

**$bin: The cache bin to store the data in. Valid core values are 'cache\_block', 'cache\_bootstrap', 'cache\_field', 'cache\_filter', 'cache\_form', 'cache\_menu', 'cache\_page', 'cache\_path', 'cache\_update' or 'cache' for the default cache.**

**Return value**

**The cache or FALSE on failure.**

**cache\_get\_multiple(array &$cids, $bin = 'cache')**

Returns data from the persistent cache when given an array of cache IDs.

**Parameters**

**$cids: An array of cache IDs for the data to retrieve.** This is passed by reference, and will have the IDs successfully returned from cache removed.

**$bin: The cache bin where the data is stored.**

**Return value**

An array of the items successfully returned from cache indexed by cid.

**cache\_clear\_all($cid = NULL, $bin = NULL, $wildcard = FALSE)**

Expires data from the cache.

If called with the arguments $cid and $bin set to NULL or omitted, then expirable entries will be cleared from the cache\_page and cache\_block bins, and the $wildcard argument is ignored.

Parameters

**$cid: If set, the cache ID or an array of cache IDs.** Otherwise, all cache entries that can expire are deleted. The $wildcard argument will be ignored if set to NULL.

**$bin: If set, the cache bin to delete from**. Mandatory argument if $cid is set.

**$wildcard: If TRUE, the $cid argument must contain a string value and cache IDs starting with $cid are deleted in addition to the exact cache ID specified by $cid. If $wildcard is TRUE and $cid is '\*', the entire cache is emptied.**

Building complicated, dynamic content in Drupal is easy, but it can come at a price. A lot of the stuff that makes a site engaging can spell 'performance nightmare' under heavy load, thrashing the database to perform complex queries and expensive calculations every time a user looks at a node or loads a particular page.

**One solution is to turn on page caching on Drupal's performance options administration page. That speeds things up for anonymous users by caching the output of each page, greatly reducing the number of DB queries needed when they hit the site. That doesn't help with logged in users, however: because page level caching is an all-or-nothing affair, it only works for the standardized, always-the-same view that anonymous users see when they arrive.**

Eventually there comes a time when you have to dig in to your code, identify the database access hot spots, and add caching yourself. Fortunately, Drupal's built-in caching APIs and some simple guidelines can make that task easy.

The first rule of optimization and caching is this: **never do something time consuming twice if you can hold onto the results and re-use them**.

function my\_module\_function() {

$my\_data = &drupal\_static(\_\_FUNCTION\_\_);

if (!isset($my\_data)) {

// Do your expensive calculations here, and populate $my\_data

// with the correct stuff..

}

return $my\_data;

}

The important part to look at in this function is the variable named $my\_data; we're initializing it with an odd-looking call to drupal\_static(). The drupal\_static() function is new to Drupal 7, and provides functions with a temporary "storage bin" for data that should stick around even after they're done executing. drupal\_static() will return an empty value the first time we call it, but any changes to the variable will be preserved when the function is called again. That means that our function can check if the variable is already populated, and return it immediately without doing any more work.

This pattern appears all over the place in Drupal -- including important functions like node\_load(). Calling node\_load() for a particular node ID requires database hits the first time, but the resulting information is kept in a static variable for the duration of the page load. That way, displaying a node once in a list, a second time in a block, and a third time in a list of related links (for example) doesn't require three full trips to the database.

**drupal\_static() allows the process to be centralized. When modules need absolutely fresh data, they can call drupal\_static\_reset() to clear out any temporarily cached information.**

You might notice that the static variable technique only stores data for the duration of a single page load. For even better performance, it's often possible to cache data in a more permanent fashion...

**function my\_module\_function() {**

**$my\_data = &drupal\_static(\_\_FUNCTION\_\_);**

**if (!isset($my\_data)) {**

**if ($cache = cache\_get('my\_module\_data')) {**

**$my\_data = $cache->data;**

**}**

**else {**

**// Do your expensive calculations here, and populate $my\_data**

**// with the correct stuff..**

**cache\_set('my\_module\_data', $my\_data, 'cache');**

**}**

**}**

**return $my\_data;**

**}**

This version of the function still uses the static variable, but it adds another layer: database caching. Drupal's APIs provide three key functions you'll need to be familiar with: **cache\_get(), cache\_set(), and cache\_clear\_all().** Let's look at how they're used.

After the initial check of the static variable, this function looks in Drupal's cache for data stored with a particular key. If it finds it, $my\_data is set to $cache->data and we're done. Combined with the static variable, future calls during this page request won't even need to call cache\_get()!

If no cached version is found, the function does the actual work of generating the data. Then it saves it TO the cache so future requests will find it. The key that you pass in as the first parameter can by anything you choose, though it's important to avoid colliding with any other modules' keys. Starting the key with the name of your module is always a good idea.

The end result? A slick little function that saves time whenever it can -- first checking for an in-memory copy of the data, then checking the cache, and finally calculating it from scratch if necessary. You'll see this pattern a lot if you dig into the guts of data-intensive Drupal modules.

KEEPING UP TO DATE

**What happens, though, if the data that you've cached becomes outdated and needs to be recalculated?** By default, cached information stays around until some module explicitly calls the cache\_clear\_all() function, emptying out your record. **If your data is updated sporadically, you might consider simply calling cache\_clear\_all('my\_module\_data', 'cache') each time you save the changes to it.** If you're caching quite a few pieces of data (perhaps versions of a particular block for each role on the site), there's a third 'wildcard' parameter:

**<?php cache\_clear\_all('my\_module', 'cache', TRUE); ?>**

**This clears out all the cache values whose keys start with 'my\_module'.**

**If you don't need your cached data to be perfectly up-to-the-second, but you want to keep it reasonably fresh, you can also pass in an expiration date to the cache\_set() function. For example:**

**<?php cache\_set('my\_module\_data', $my\_data, 'cache', time() + 360); ?>**

The final parameter is a unix timestamp value representing the 'expiration date' of the cache data. The easiest way to calculate it is to use the time() function, and add the data's desired lifetime in seconds. Expired entries will be automatically discarded as they pass that date.

CONTROLLING WHERE CACHED DATA IS STORED

You might have noticed that **cache\_set()'s third parameter is 'cache' -- the name of the table that stores the default cache data.** If you're storing large amounts of data in the cache, you can set up your own dedicated cache table and pass its name into the function. That will help keep your cache lookups speedy no matter what other modules are sticking into their own tables. The Views module uses that technique to maintain full control over when its cache data is cleared.

The easiest **place to set up a custom cache table** is in your module's install file, in the **hook\_schema()** function. It's where all of the custom tables used by your module are defined, and you can even make use of one of Drupal's internal helper functions to simplify the process.

**function mymodule\_schema() {**

**$schema['cache\_mymodule'] = drupal\_get\_schema\_unprocessed('system', 'cache');**

**return $schema;**

**}**

Using the drupal\_get\_schema\_unprocessed() function, the code above retrieves the definition of the System module's standard Cache table, and creates a clone of it named 'cache\_mymodule'. Prefixing the name of custom cache tables with the word 'cache' is common practice in Drupal, and helps keep the assorted cache tables organized.

If you're really hoping to squeeze the most out of your server, Drupal also supports the use of alternative caching systems. By changing a single line in your site's settings.php file, you can point it to different implementations of the standard cache\_set(), cache\_get(), and cache\_clear\_all() functions. The most popular integration is with the open source memcached project, but other approaches are possible (such as a file-based cache or against PHP's APC). As long as you've used the standard Drupal caching functions, your module's code won't have to be altered.

ADVANCED CACHING WITH RENDERABLE CONTENT

In Drupal 7, "renderable arrays" are used extensively when building the contents of each page for display. Modules can define page elements like blocks, tables, forms, and even nodes as structured arrays; when the time comes to render the page to HTML, Drupal automatically uses the drupal\_render() function to process them, calling the theme layer and other helper functions automatically. Some complex page elements, though, can take quite a bit of time to render into HTML. **By adding a special #cache property onto the renderable element, you can instruct the drupal\_render() function to cache and reuse the rendered HTML each time the page element is built.**

**$content['my\_content'] = array(**

**'#cache' => array(**

**'cid' => 'my\_module\_data',**

**'bin' => 'cache',**

**'expire' => time() + 360,**

**),**

**// Other element properties go here...**

**);**

The #cache property contains a list of values that mirror the parameters you would pass to the cache\_get() and cache\_set() if you were calling them manually. For more information on how caching of renderable elements works, check out the detailed documentation for the drupal\_render() function on api.drupal.org.

A FEW CAVEATS

Like all good things, it's possible to overdo it with caching. Sometimes, it just doesn't make sense -- **if you're looking up a single record from a table, saving the result to a database cache is silly.** Using the Devel module is a good way to spot the functions where caching will pay off: it can log the queries that are used on your site and highlight the ones that are slow, or the ones that are repeated numerous times on each page.

Other times, **the data you're using will just be a bad fit for the standard caching system**. If you need to join cached data in SQL queries, for example, cache\_set()'s practice of string data as a serialized string will be a problem. In those cases, you'll need to come up with a solution that's specific to your module. VotingAPI maintains one table full of individual votes and another table full of calculated results (averages, sums, etc.) for quick joining when sorting and filtering nodes.

Finally, it's important to remember that the cache is not long term storage! Since other modules can call cache\_clear\_all() and wipe it out, you should never put something into it if you can't recalculate it again using the original source data.