Cache

Configuration

Laravel provides an expressive, unified API for various caching backends. The cache configuration is located at `config/cache.php`. In this file you may specify which cache driver you would like to be used by default throughout your application. Laravel supports popular caching backends like Memcached and Redis out of the box.

The cache configuration file also contains various other options, which are documented within the file, so make sure to read over these options. By default, Laravel is configured to use the `file` cache driver, which stores the serialized, cached objects in the filesystem. For larger applications, it is recommended that you use a more robust driver such as Memcached or Redis. You may even configure multiple cache configurations for the same driver.

Driver Prerequisites

Database

When using the `database` cache driver, you will need to setup a table to contain the cache items. You'll find an example `Schema` declaration for the table below:

```
Schema::create('cache', function ($table) {
    $table->string('key')->unique();
    $table->text('value');
    $table->integer('expiration');
});
```

You may also use the `php artisan cache:table` Artisan command to generate a migration with the proper schema.

Memcached

Using the Memcached driver requires the Memcached PECL package to be installed. You may list all of your Memcached servers in the `config/cache.php` configuration file:

```
'memcached' => [
    [
        'host' => '127.0.0.1',
        'port' => 11211,
        'weight' => 100
    ],
],
```

You may also set the `host` option to a UNIX socket path. If you do this, the `port` option should be set to `0`:

```
'memcached' => [
    [
        'host' => '/var/run/memcached/memcached.sock',
        'port' => 0,
```

```
        'weight' => 100
    ],
],
```

Redis

Before using a Redis cache with Laravel, you will need to either install the PhpRedis PHP extension via PECL or install the `predis/predis` package (~1.0) via Composer.

For more information on configuring Redis, consult its Laravel documentation page.

Cache Usage

Obtaining A Cache Instance

The `Illuminate\Contracts\Cache\Factory` and `Illuminate\Contracts\Cache\Repository` contracts provide access to Laravel's cache services. The `Factory` contract provides access to all cache drivers defined for your application. The `Repository` contract is typically an implementation of the default cache driver for your application as specified by your `cache` configuration file.

However, you may also use the `Cache` facade, which is what we will use throughout this documentation. The `Cache` facade provides convenient, terse access to the underlying implementations of the Laravel cache contracts:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Cache;

class UserController extends Controller
{
    /**
     * Show a list of all users of the application.
     *
     * @return Response
     */
    public function index()
    {
        $value = Cache::get('key');

        //
    }
}
```

Accessing Multiple Cache Stores

Using the `Cache` facade, you may access various cache stores via the `store` method. The key passed to the `store` method should correspond to one of the stores listed in the `stores` configuration array in your `cache` configuration file:

```php
$value = Cache::store('file')->get('foo');

Cache::store('redis')->put('bar', 'baz', 600); // 10 Minutes
```

Retrieving Items From The Cache

The `get` method on the `Cache` facade is used to retrieve items from the cache. If the item does not exist in the cache, `null` will be returned. If you wish, you may pass a second argument to the `get` method specifying the default value you wish to be returned if the item doesn't exist:

```php
$value = Cache::get('key');

$value = Cache::get('key', 'default');
```

You may even pass a `Closure` as the default value. The result of the `Closure` will be returned if the specified item does not exist in the cache. Passing a Closure allows you to defer the retrieval of default values from a database or other external service:

```php
$value = Cache::get('key', function () {
    return DB::table(...)->get();
});
```

Checking For Item Existence

The `has` method may be used to determine if an item exists in the cache. This method will return `false` if the value is `null`:

```
if (Cache::has('key')) {
    //
}
```

Incrementing / Decrementing Values

The `increment` and `decrement` methods may be used to adjust the value of integer items in the cache. Both of these methods accept an optional second argument indicating the amount by which to increment or decrement the item's value:

```
Cache::increment('key');
Cache::increment('key', $amount);
Cache::decrement('key');
Cache::decrement('key', $amount);
```

Retrieve & Store

Sometimes you may wish to retrieve an item from the cache, but also store a default value if the requested item doesn't exist. For example, you may wish to retrieve all users from the cache or, if they don't exist, retrieve them from the database and add them to the cache. You may do this using the `Cache::remember` method:

```
$value = Cache::remember('users', $seconds, function () {
    return DB::table('users')->get();
});
```

If the item does not exist in the cache, the `Closure` passed to the `remember` method will be executed and its result will be placed in the cache.

You may use the `rememberForever` method to retrieve an item from the cache or store it forever:

```
$value = Cache::rememberForever('users', function () {
    return DB::table('users')->get();
});
```

Retrieve & Delete

If you need to retrieve an item from the cache and then delete the item, you may use the `pull` method. Like the `get` method, `null` will be returned if the item does not exist in the cache:

```
$value = Cache::pull('key');
```

Storing Items In The Cache

You may use the `put` method on the `Cache` facade to store items in the cache:

```
Cache::put('key', 'value', $seconds);
```

If the storage time is not passed to the `put` method, the item will be stored indefinitely:

```
Cache::put('key', 'value');
```

Instead of passing the number of seconds as an integer, you may also pass a `DateTime` instance representing the expiration time of the cached item:

```
Cache::put('key', 'value', now()->addMinutes(10));
```

Store If Not Present

The `add` method will only add the item to the cache if it does not already exist in the cache store. The method will return `true` if the item is actually added to the cache. Otherwise, the method will return `false`:

```
Cache::add('key', 'value', $seconds);
```

Storing Items Forever

The `forever` method may be used to store an item in the cache permanently. Since these items will not expire, they must be manually removed from the cache using the `forget` method:

```
Cache::forever('key', 'value');
```

If you are using the Memcached driver, items that are stored "forever" may be removed when the cache reaches its size limit.

Removing Items From The Cache

You may remove items from the cache using the `forget` method:

```
Cache::forget('key');
```

You may also remove items by providing a zero or negative TTL:

```
Cache::put('key', 'value', 0);

Cache::put('key', 'value', -5);
```

You may clear the entire cache using the `flush` method:

```
Cache::flush();
```

Flushing the cache does not respect the cache prefix and will remove all entries from the cache. Consider this carefully when clearing a cache which is shared by other applications.

Atomic Locks

To utilize this feature, your application must be using the `memcached`, `dynamodb`, or `redis` cache driver as your application's default cache driver. In addition, all servers must be communicating with the same central cache server.

Atomic locks allow for the manipulation of distributed locks without worrying about race conditions. For example, Laravel Forge uses atomic locks to ensure that only one remote task is being executed on a server at a time. You may create and manage locks using the `Cache::lock` method:

```
use Illuminate\Support\Facades\Cache;

$lock = Cache::lock('foo', 10);

if ($lock->get()) {
    // Lock acquired for 10 seconds...

    $lock->release();
}
```

The `get` method also accepts a Closure. After the Closure is executed, Laravel will automatically release the lock:

```
Cache::lock('foo')->get(function () {
    // Lock acquired indefinitely and automatically released...
});
```

If the lock is not available at the moment you request it, you may instruct Laravel to wait for a specified number of seconds. If the lock can not be acquired within the specified time limit, an `Illuminate\Contracts\Cache\LockTimeoutException` will be thrown:

```
use Illuminate\Contracts\Cache\LockTimeoutException;

$lock = Cache::lock('foo', 10);

try {
    $lock->block(5);
```

```
    // Lock acquired after waiting maximum of 5 seconds...
} catch (LockTimeoutException $e) {
    // Unable to acquire lock...
} finally {
    optional($lock)->release();
}

Cache::lock('foo', 10)->block(5, function () {
    // Lock acquired after waiting maximum of 5 seconds...
});
```

Managing Locks Across Processes

Sometimes, you may wish to acquire a lock in one process and release it in another process. For example, you may acquire a lock during a web request and wish to release the lock at the end of a queued job that is triggered by that request. In this scenario, you should pass the lock's scoped "owner token" to the queued job so that the job can re-instantiate the lock using the given token:

```
// Within Controller...
$podcast = Podcast::find($id);

$lock = Cache::lock('foo', 120);

if ($result = $lock->get()) {
    ProcessPodcast::dispatch($podcast, $lock->owner());
}

// Within ProcessPodcast Job...
Cache::restoreLock('foo', $this->owner)->release();
```

If you would like to release a lock without respecting its current owner, you may use the `forceRelease` method:

```
Cache::lock('foo')->forceRelease();
```

The Cache Helper

In addition to using the `Cache` facade or cache contract, you may also use the global `cache` function to retrieve and store data via the cache. When the `cache` function is called with a single, string argument, it will return the value of the given key:

```
$value = cache('key');
```

If you provide an array of key / value pairs and an expiration time to the function, it will store values in the cache for the specified duration:

```
cache(['key' => 'value'], $seconds);

cache(['key' => 'value'], now()->addMinutes(10));
```

When the `cache` function is called without any arguments, it returns an instance of the `Illuminate\Contracts\Cache\Factory` implementation, allowing you to call other caching methods:

```
cache()->remember('users', $seconds, function () {
    return DB::table('users')->get();
});
```

When testing call to the global `cache` function, you may use the `Cache::shouldReceive` method just as if you were testing a facade.

Cache Tags

Cache tags are not supported when using the `file` or `database` cache drivers. Furthermore, when using multiple tags with caches that are stored "forever", performance will be best with a driver such as `memcached`, which automatically purges stale records.

Storing Tagged Cache Items

Cache tags allow you to tag related items in the cache and then flush all cached values that have been assigned a given tag. You may access a tagged cache by passing in an ordered array of tag names. For example, let's access a tagged cache and `put` value in the cache:

```
Cache::tags(['people', 'artists'])->put('John', $john, $seconds);

Cache::tags(['people', 'authors'])->put('Anne', $anne, $seconds);
```

Accessing Tagged Cache Items

To retrieve a tagged cache item, pass the same ordered list of tags to the `tags` method and then call the `get` method with the key you wish to retrieve:

```
$john = Cache::tags(['people', 'artists'])->get('John');

$anne = Cache::tags(['people', 'authors'])->get('Anne');
```

Removing Tagged Cache Items

You may flush all items that are assigned a tag or list of tags. For example, this statement would remove all caches tagged with either `people`, `authors`, or both. So, both `Anne` and `John` would be removed from the cache:

```
Cache::tags(['people', 'authors'])->flush();
```

In contrast, this statement would remove only caches tagged with `authors`, so `Anne` would be removed, but not `John`:

```
Cache::tags('authors')->flush();
```

Adding Custom Cache Drivers

Writing The Driver

To create our custom cache driver, we first need to implement the `Illuminate\Contracts\Cache\Store` contract. So, a MongoDB cache implementation would look something like this:

```
<?php

namespace App\Extensions;

use Illuminate\Contracts\Cache\Store;

class MongoStore implements Store
{
    public function get($key) {}
    public function many(array $keys) {}
    public function put($key, $value, $seconds) {}
    public function putMany(array $values, $seconds) {}
    public function increment($key, $value = 1) {}
    public function decrement($key, $value = 1) {}
    public function forever($key, $value) {}
    public function forget($key) {}
    public function flush() {}
    public function getPrefix() {}
}
```

We just need to implement each of these methods using a MongoDB connection. For an example of how to implement each of these methods, take a look at the `Illuminate\Cache\MemcachedStore` in the framework source code. Once our implementation is complete, we can finish our custom driver registration.

```
Cache::extend('mongo', function ($app) {
    return Cache::repository(new MongoStore);
});
```

If you're wondering where to put your custom cache driver code, you could create an `Extensions` namespace within your `app` directory. However, keep in mind that Laravel does not have a rigid application structure and you are free to organize your application according to your preferences.

Registering The Driver

To register the custom cache driver with Laravel, we will use the `extend` method on the `Cache` facade. The call to `Cache::extend` could be done in the `boot` method of the default `App\Providers\AppServiceProvider` that ships with fresh Laravel applications, or you may create your own service provider to house the extension - just don't forget to register the provider in the `config/app.php` provider array:

```php
<?php

namespace App\Providers;

use App\Extensions\MongoStore;
use Illuminate\Support\Facades\Cache;
use Illuminate\Support\ServiceProvider;

class CacheServiceProvider extends ServiceProvider
{
    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Cache::extend('mongo', function ($app) {
            return Cache::repository(new MongoStore);
        });
    }
}
```

The first argument passed to the `extend` method is the name of the driver. This will correspond to your `driver` option in the `config/cache.php` configuration file. The second argument is a Closure that should return an `Illuminate\Cache\Repository` instance. The Closure will be passed an `$app` instance, which is an instance of the service container.

Once your extension is registered, update your `config/cache.php` configuration file's `driver` option to the name of your extension.

Events

To execute code on every cache operation, you may listen for the events fired by the cache. Typically, you should place these event listeners within your `EventServiceProvider`:

```php
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Cache\Events\CacheHit' => [
        'App\Listeners\LogCacheHit',
    ],

    'Illuminate\Cache\Events\CacheMissed' => [
        'App\Listeners\LogCacheMissed',
    ],

    'Illuminate\Cache\Events\KeyForgotten' => [
        'App\Listeners\LogKeyForgotten',
    ],

    'Illuminate\Cache\Events\KeyWritten' => [
        'App\Listeners\LogKeyWritten',
    ],
];
```

Become a Laravel Partner

Laravel Partners are elite shops providing top-notch Laravel development and consulting. Each of our partners can help you craft a beautiful, well-architected project.

Our Partners

Laravel

Highlights

- Release Notes
- Getting Started
- Routing
- Blade Templates
- Authentication
- Authorization
- Artisan Console
- Database
- Eloquent ORM
- Testing

Resources

- Laracasts
- Laravel News
- Laracon
- Laracon EU
- Laracon AU
- Jobs
- Certification
- Forums

Partners

- Vehikl
- Tighten Co.
- Kirschbaum
- Byte 5
- 64Robots
- Cubet
- DevSquad
- Ideil
- Cyber-Duck
- ABOUT YOU

Ecosystem

- Vapor
- Forge
- Envoyer
- Horizon
- Lumen
- Nova
- Echo
- Valet
- Mix
- Spark

Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable and creative experience to be truly fulfilling. Laravel attempts to take the pain out of development by easing common tasks used in most web projects.

Become A Partner

Cashier

Homestead

Dusk

Passport

Scout

Socialite