Using Redis as a Cache in Laravel

One of the easiest ways to speed up your web app is to cache slow-to-compute data, so it doesn't have to be recomputed the next time the page loads. In this article, Keneth Ekandem shows us how to use Redis to cache DB results in a Laravel App.

author photo By Kenneth Ekandem Author Twitter #php May 31, 2021

Laravel is a web application framework built with PHP. It’s a framework that uses providers and dependency injections for code organization. It follows a model-view-controller design pattern. Laravel reuses the existing components of different frameworks, which helps in creating a web application. The web application thus designed is more structured and pragmatic.

Redis is a disk-persistent key-value database with support for multiple data structures or data types, which means that while it supports mapped key-value-based strings to store and retrieve data (analogous to the data model supported in traditional kinds of databases), it also supports other complex data structures, such as lists and sets.

Laravel supports the use of Redis, which uses caches for temporary data storage to speed up the process of performing database queries and getting feedback, which will, in turn, reduce the amount of time spent pulling up data.

What is a Cache?

A cache is a component that stores data so that future requests for the data can be fulfilled quickly. The data stored in a cache might be the result of an earlier computation or duplicates of data stored elsewhere. Therefore, whenever the same result is needed, one can use the cached value, which has already been computed and stored elsewhere, so there is no need to perform repeated computations of the same thing.

Redis uses server-assisted client-side caching, which allows it to exploit the available memory of the application servers, which are usually distinct computers compared to the database nodes, to store some subset of database information directly on the application side. Usually, when some data are required, the application servers ask the database about such information, but in Redis, the record is retrieved from the local cache.

Before using Redis, it must be set up on your Windows or Mac operating system. If you do not have it installed, you can follow the instructions given in the MacOS or Windows version of the tutorial to set it up.

Note: In this tutorial, we will be using Postman as our API client. You can go ahead and use Laravel’s resources to build the view for your application, but it will not be covered in this article.

System Requirements

Composer

Postman

Benefits of Redis

Redis acts as a temporary database for storing data up to 512 MB in size and allows data lookup to be performed more efficiently and quickly, which will prove useful in large-scale applications.

Redis uses the master-slave data replication structure, which means that slave nodes always listen to the master node, and when data are updated in the master node, the data are also updated in the slave node. All of this can be done asynchronously.

Redis uses a built-in hashing mechanism called Redis Hashing, which stores information in the form of a key and a map.

There are many other benefits to using Redis, but let’s stick to the listed ones for now. More information about Redis is available on its official documentation page

Redis Get and Set Methods

Get: This gets a key example blogs from the local cache and return its value. The Get method only stores strings and will return an error—the key stores hashes or others.

Set: This method sets a key example blogs to hold string values. If a key already exists, then it will override the value(s) of that key, regardless of its type, unless instructed otherwise.

Setting up Laravel

To get started, we need to create a new instance of Laravel on our local machine. We create a new Laravel project from our command line with the name laravel-redis, go into the newly created directory containing our files, and run the application:

composer create-project laravel/laravel laravel-redis

cd laravel-redis

php artisan serve

Next, we need to install the predis package in our application using composer prior to using Redis on our application. Predis is a Redis client written entirely in PHP and does not require any additional extensions. Run the following command on your terminal to install Predis:

Installing Redis

composer require predis/predis

When the installation is complete, we can find our Redis configuration settings in config/database.php. In the file, you will see a redis array containing the Redis server. By default, the client is set to phpredis, and since predis is being used in this tutorial, the client will be changed to predis:

'redis' => [

'client' => env('REDIS\_CLIENT', 'predis'),

'options' => [

'cluster' => env('REDIS\_CLUSTER', 'redis'),

'prefix' => env('REDIS\_PREFIX', Str::slug(env('APP\_NAME', 'laravel'), '\_').'\_database\_'),

],

'default' => [

'url' => env('REDIS\_URL'),

'host' => env('REDIS\_HOST', '127.0.0.1'),

'password' => env('REDIS\_PASSWORD', null),

'port' => env('REDIS\_PORT', '6379'),

'database' => env('REDIS\_DB', '0'),

],

],

You can modify the default settings to your own custom settings depending on your URL and add a password, but for this tutorial, we won’t be doing that.

After that is done, go ahead to register your REDIS\_CLIENT in the .env file:

REDIS\_CLIENT=predis

Seeding

I’ve gone ahead and created a database connection in my .env file and seeded a couple of blogs in the database.

Note: This is not a compulsory step but rather a maneuver to avoid going the long route to make a create request.

Use the command below to create a seeder. Then, add a couple of blogs, which we run with the artisan command to seed the data:

php artisan make:seeder BlogSeeder

This will create a new BlogSeeder in your database folder. Next, we will add a couple of blogs:

DB::table('blogs')->insert([

'title' => 'First blog',

'sub\_header' => 'This is the first sub header',

'content' => 'BLOG\_CONTENT'

]);

...

Then, we run the artisan command to seed the data to the database under the blogs table:

php artisan db:seed --class=BlogSeeder

Fetching

To fetch a blog from the database, we create a blog controller and add a function to fetch a blog using id from the database. On the first request, the blog will be fetched from the database and cached in Redis. However, on subsequent requests, the blog will be retrieved directly from Redis and formatted in JSON.

First, we will create a new controller using the following command:

php artisan make:controller BlogController

Then, we add our function to get all blogs:

use App\Models\Blog;

use Illuminate\Support\Facades\Redis;

public function index($id) {

$cachedBlog = Redis::get('blog\_' . $id);

if(isset($cachedBlog)) {

$blog = json\_decode($cachedBlog, FALSE);

return response()->json([

'status\_code' => 201,

'message' => 'Fetched from redis',

'data' => $blog,

]);

}else {

$blog = Blog::find($id);

Redis::set('blog\_' . $id, $blog);

return response()->json([

'status\_code' => 201,

'message' => 'Fetched from database',

'data' => $blog,

]);

}

}

In the above code, we are first checking Redis for the key with blog\_ + id and returning it in JSON format if it exists. If the key has not been set, then we go to the database, get the key, and set it in Redis using blog\_ + id.

Next, we create a route in routes/web.php that our API can interact with when used in Postman. This route will point to our BlogController and, specifically, our function that is fetching the blog:

Route::get('/blogs/{id}', 'BlogController@index');

Updating

In this process, we will get a single blog and update it in the database. When this is done, we will look up the key in Redis and delete it, and then create a new key with the id of the updated blog.

Before we proceed, I commented out \App\Http\Middleware\VerifyCsrfToken::class, in the app/http/kernel.php file to disable CSRF verification. You shouldn’t do this in an application ready for production, as this is necessary for the protection of the application routes. You can read more about CSRF verification and its usefulness here.

First, we create our update function in the BlogController:

public function update(Request $request, $id) {

$update = Blog::findOrFail($id)->update($request->all());

if($update) {

// Delete blog\_$id from Redis

Redis::del('blog\_' . $id);

$blog = Blog::find($id);

// Set a new key with the blog id

Redis::set('blog\_' . $id, $blog);

return response()->json([

'status\_code' => 201,

'message' => 'User updated',

'data' => $blog,

]);

}

}

In the above code, we got the id of the blog, updated it in the database, and then deleted the key with blog\_ + id. After the key is deleted, we then fetch the blog from the database using the id and create a new key in Redis.

After this is done, we can create our route:

Route::post('/blogs/update/{id}', 'BlogController@update');

Delete

For the last part, we will delete a blog from the database and then delete the key from Redis. First, add the delete function:

public function delete($id) {

Blog::findOrFail($id)->delete();

Redis::del('blog\_' . $id);

return response()->json([

'status\_code' => 201,

'message' => 'Blog deleted'

]);

}

This will get the id and delete the blog from the database. Then, it will delete the key from Redis. After adding our function, we can then create a route.

Route::delete('/blogs/delete/{id}', 'BlogController@delete';

Wrapping Up

In this article, we talked about caching and what it is, Redis and its benefits, and how to use Redis in Laravel to temporarily cache data to speed up database queries. We also went through the steps to update and delete data in Redis using keys and implemented the Get/Set methods effectively. I hope this tutorial is helpful.

Honeybadger has your back when it counts.

We're the only error tracker that combines exception monitoring, uptime monitoring, and cron monitoring into a single, simple to use platform. Our mission: to tame production and make you a better, more productive developer.