

Deploy your app quickly and scale as you grow with our Hobby Tier



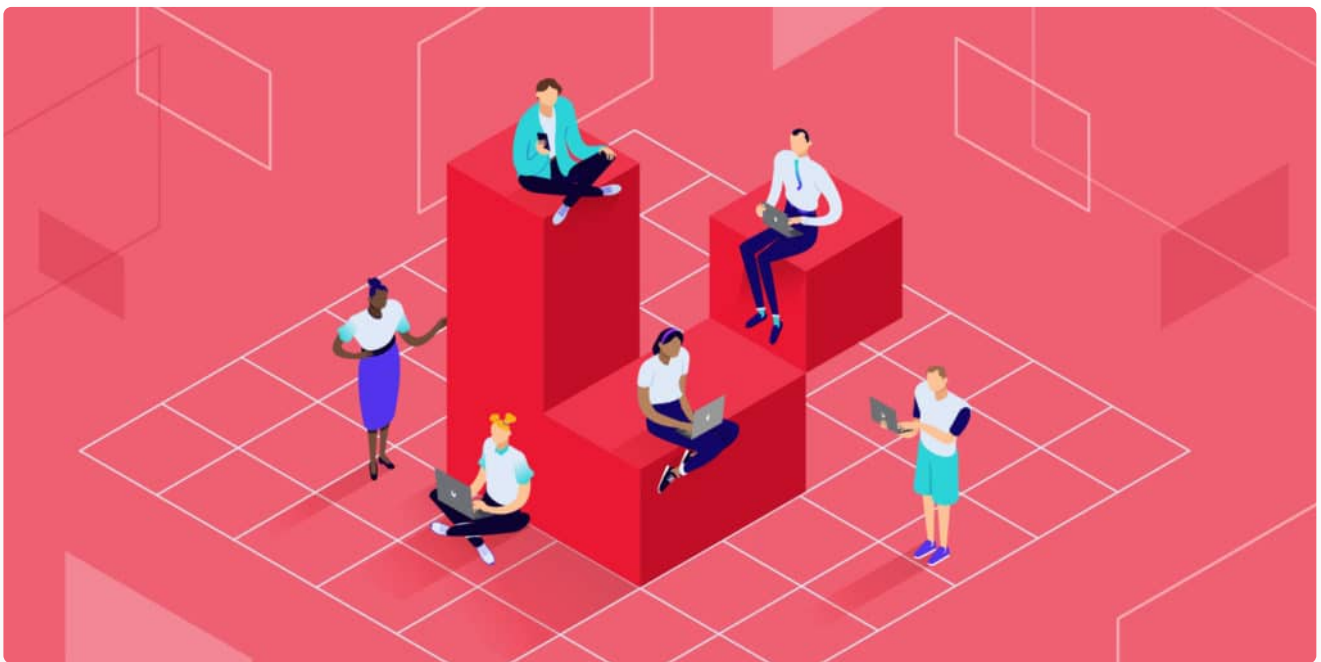
KINSTA



PHP LARAVEL

# Everything You Need to Know About Laravel Caching

Solomon Eseme, September 16, 2022



Shares

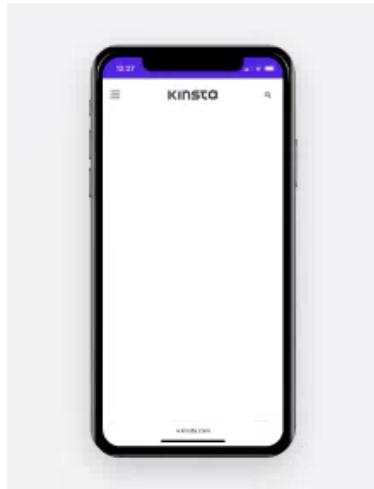


Caching is essential for achieving high performance and scalability. Implementing the proper [caching strategy](#) right from the development phase is critical to avoid lagging APIs and sluggish [page load times](#). Laravel is one of the most popular PHP frameworks, so implementing the optimal Laravel caching strategy is indispensable to a better user experience and greater business impact.

In this article, we'll explore strategies for implementing and manipulating caching in Laravel. You'll learn about how Laravel caching works, several Laravel caching queries, and how y



can handle caching on Laravel apps.



**In a hurry? Save this article as a PDF.**

Download

You'll get more out of this article if you already have a basic know-how of the following under your belt:

- Good knowledge of [web development](#)
- Basic [understanding of Laravel](#)
- Building [APIs with Laravel](#)
- Basic [understanding of caching](#)

Let's dive in!

## Table of Contents

- [Why Is Caching Important?](#)
- [What Is Laravel Caching?](#)
- [Laravel Caching Strategies](#)
- [Caching the UI Part of a Laravel App](#)
- [Build a Laravel App](#)

## Check Out Our [Video Guide](#) to Laravel Cache

Everything You Need to Know About Laravel Cachi...



## Why Is Caching Important?

With the recent boom in internet businesses, different companies have statistics showing how website load time and low performance can heavily impact SEO, user engagement, and conversation rates without caching. And that starts with an excellent caching strategy in place.

An [online study found](#), “1 second of load lag time would cost Amazon \$1.6 billion in sales per year.”

Another [Google study reported](#), “Our research shows that if search results are slowed by even a fraction of a second, people search less (seriously: A **400ms delay** leads to a **0.44 percent drop** in search volume, data fans). And this impatience isn’t just limited to search: Four out of five internet users will click away if a video stalls while loading.”



A slight sluggishness on your web page load time can result in a massive impact on your users’ experience and loss of funds at large.

## What Is Laravel Caching?

Laravel provides a robust and easy-to-use implementation of caching and different caching backends. With Laravel cache, you can efficiently and effectively switch between many caching engines without writing any code.

You can find the configuration of the Laravel cache within the `config/cache.php` folder, though you'll likely only need the `.env` file to switch between different cache backends.

Laravel cache also provides many practical methods that we can use to implement different caching strategies.

“ Caching is an essential part of the software development process for anyone who cares more about high performance and scalability.  Dive into everything you need to know about laravel caching here  ”

CLICK TO TWEET

## Laravel Cache Drivers and Comparisons

Laravel cache provides great caching backends and drivers. Depending on your use case, you can switch between them to enhance your application performance and load time.

That said, Laravel cache also provides a seamless way to create a custom backend and use it with Laravel cache, but only if the list below does not fit your use case.

We'll talk about the list of all the backends provided by Laravel cache next.

### 1. File

The **file** driver is the default backend used by the Laravel cache when no driver is specified in the `.env` file.

The **file** backend is designed to store the [cached data](#) in an encrypted file found under `storage/framework/`. Laravel creates an encrypted file with the data and the cache key when new data is cached. The same happens when the user is trying to retrieve the content. Laravel cache searches through the folder for the specified key and, if found, returns the content.

Though the **file** backend works flawlessly and saves time installing and configuring external drivers, it can also be [perfect for development](#). It's faster than accessing the data from the database server directly.

To use the **file** driver, add the following code to your `.env` file:

```
CACHE_DRIVER=file
```

## 2. Array

The **array** driver is a perfect caching backend for running automated tests and is configured easily with Github Actions, Jenkins, etc.

The **array** backend stores the cached data in an array in PHP and does not require you to install or configure any drivers. It works perfectly for automated tests, and it's a bit faster than the file cache backend.

To use the **array** driver, add the following code to your **.env** file:

```
CACHE_DRIVER=array
```

## 3. Database

When using the **database** driver, data is stored in memory for the current PHP process. Therefore, you need to create a database table to store the cached data. In addition, [database](#) caching improves scalability by distributing query workload from the backend to multiple frontends.

You can run this Artisan command — `php artisan cache:table` — to auto-generate the database schema needed by the database driver.

The **database** driver is used mainly in situations where you can install any software on your hosting platform.

For example, let's say you are using a free hosting plan with limited options. For that, we'd suggest sticking with the **file** driver because the **database** driver is, in most cases, the weakest point of your application, and trying to push more data into that bottleneck is not a good idea.

To use the **database** driver, add the following code to your **.env** file:

```
CACHE_DRIVER=database
```

#### 4. Redis

The **redis** driver uses the in-memory-based caching technology called [Redis](#). Though it's swift compared to the other cache drivers discussed above, it requires installing and configuring external technology.

To use the **redis** driver, add the following code to your **.env** file:

```
CACHE_DRIVER=redis
```

#### 5. Memcached

Memcached is known to be the most popular in-memory-based cache-store. If you don't mind a bit of extra server maintenance (having to install and maintain additional services), the memory-based cache drivers Memcached are great options.

Using the **memcached** driver requires the [Memcached PECL package](#) to be installed.

To use the **memcached** driver, add the following code to your **.env** file.

```
CACHE_DRIVER=memcached
```

The [best caching driver to use and the performance](#) of the cache driver depend on your project use case and the quantity of data to be retrieved.

## Laravel Cache Usage and Methods

Laravel cache provides many valuable methods used to implement many caching strategies.

Below we'll list and explain the different methods (categorized as per their use case):

1. `put()`
2. `get()`
3. `many()`
4. `putMany()`
5. `increment()`
6. `decrement()`
7. `forever()`
8. `forget()`
9. `flush()`
10. `remember()`
11. `rememberForever()`

## Storing Cache

Storing new data in the cache is very simple using the different methods, each with several use cases.

### 1. `Cache::put()`

This method accepts three key parameters, **duration**, and the **data** to be cached.

Let's take a look at how to use `Cache::put()`:

```
Cache::put(key, data, duration)

$post = Post::find(1);

Cache::put('post_1', $post, 20);
```

The code above will cache the post with the unique **key** for 20 seconds.

### 2. `Cache::putMany()`

This method stores an array of data in the cache at once with the same duration. It accepts two parameters which are **data** and **seconds**.

Let's take a look at how to use `Cache::putMany()` :

```
Cache::putMany(data, duration) // syntax

$posts = Post::all();

Cache::putMany($posts, 20);
```

### 3. Cache::remember()

This method is another excellent way to implement the cache Aside strategy. The

`Cache::remember()` method accepts three parameters, a **key**, **seconds**, and **closure** used to retrieve data from the database if not found.

Let's take a look at how to use `Cache::remember()` :

```
Cache::remember(key, duration, closure) // syntax

Cache::remember('posts', 20, function(){
    return Post::all();
});
```

Laravel cache also has the `Cache::rememberForever()` method, which does not accept the **seconds** parameter and stores the **data** forever.

### 4. Cache::forever()



This method stores data in the cache server forever without specifying any duration. You can implement it with the following code:

```
Cache::forever(key, data)

$post = Post::find(1);

Cache::forever('post_1', $post);
```

## Retrieving Cache Data

The methods in this category retrieve data from the cache. Some of these methods can behave differently depending on if the data is found or not.

### 1. Cache::get()

This method retrieves data from the cache server with a specific key. You can retrieve an item by using the code below:

```
Cache::get(key) // syntax

$posts = Cache::get('posts');
```

### 2. Cache::many()

This method is similar to `Cache::putMany()`. It is used to retrieve an array of cached data at once using an array of the cached keys. You can retrieve an array of cache using the following code:

```
Cache::many(keys) // syntax

const $keys = [
    'posts',
    'post_1',
    'post_2'
];

$posts = Cache::many($keys);
```

### 3. Cache::remember()

You can also use this method to retrieve cached data by checking the cache server using the key provided. If the data is stored in the cache, it will retrieve it. Otherwise, it will retrieve the data from the Database Server and cache it. This method is the same as the

`Cache::rememberForever()` method with just an extra **seconds** parameter in the `Cache::remember()` method.

## Removing Items From Cache

The methods under this category are used to remove items from the cache, grouped by functionality.

### 1. Cache::forget()

This method removes a single item from the cache with a specified key parameter:

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

### 2. Cache::flush()

This method clears all the cache engines. It deletes all the items stored anywhere in the cache:

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

### Incrementing or Decrementing Cache Values

You can adjust the values of an integer value stored in your cache by using the increment and decrement methods, respectively:

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

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

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

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

Laravel cache has many great methods that we have not discussed above, but the above methods are popular. You can get an overview of all the methods in the official [Laravel cache documentation](#).

### Deploy your application to Kinsta. Get started now with a free trial.

Run your Node.js, Python, Go, PHP, Ruby, Java, and Scala apps, (or almost anything else if you use your own custom Dockerfiles), in three, easy steps!

Start free trial

### Cache Commands Explained

Laravel provides commands to make working with Laravel cache easy and fast. Below is the list of all the [commands](#) and their functionalities.

### Clear Laravel Cache

This command is used to clear the Laravel cache before it even expires using the terminal/console. For example, you can run the following command:

```
php artisan cache:clear
```

### Clear Route-Cache

This command is used to clear the route cache of your Laravel application. For example, run the following command to clear your route cache:

```
php artisan config:cache
```

### Clear Compiled View Files

This command is used to clear the compiled view files of your Laravel application. You can achieve it with the following command:

```
php artisan view:clear
```

### Database Table

When using the database driver, you need to create a database schema called **cache** to store the cache data. You may also use the Artisan command to generate a migration with the

proper schema:

```
php artisan cache:table
```

## Laravel Caching Strategies

Depending on your application use case and data structure, several different cache strategies are likely available to you. You can even create a custom strategy to fit your needs. Below we'll go over the list of popular caching strategies you can implement in your Laravel project.

### Info

Kinsta allows you to run Laravel for your products, even though it's not officially supported by our team.

### writeThrough

In the **writeThrough** strategy, the cache server sits between the requests and the database server, making every **write** operation go through the cache server before going to the Database Server. Thus, the **writeThrough** caching strategy is similar to the **readThrough** strategy.

You can implement this strategy with the Laravel cache with the following code:

```
public function writeThrough($key, $data, $minutes) {  
    $cachedData = Cache::put($key, $data, $minutes)  
  
    // Database Server is called from(after) the Cache Server.  
    $this->storeToDB($cachedData)
```

```
        return $cacheData;
    }

    private function storeToDB($data){
        Database::create($data)
        return true
    }
}
```

## writeBack (writeBehind)

This strategy is a more advanced way of implementing the **writeThrough** strategy by adding writing operations delays.

You can also call this the **writeBehind** strategy because of the delay in time applied to the cache server before writing the data to the database server.

You can implement this strategy with the Laravel cache with the following code:

```
$durationToFlush = 1; // (in minute)
$tempDataToFlush = [];

public function writeBack($key, $data, $minutes){
    return $this->writeThrough($key, $data, $minutes);
}

public function writeThrough($key, $data, $minutes) {
    $cacheData = Cache::put($key, $data, $minutes);
    $this->storeForUpdates($cacheData);
    return $cacheData;
}

// Stores new data to temp Array for updating
private function storeForUpdates($data){
```

```

        $tempData = {};
        $tempData['duration'] = this.getMinutesInMilli();
        $tempData['data'] = data;
        array_push($tempDataToFlush, data);
    }

    // Converts minutes to millisecond
    private function getMinutesInMilli(){
        $currentDate = now();
        $futureDate = Carbon\Carbon::now()->timestamp + $this->durationToFlush
        return $futureDate->timestamp
    }

    // Calls to update the Database Server.
    public function updateDatabaseServer(){
        if($this->tempDataToFlush){
            foreach($this->tempDataToFlush as $index => $obj){
                if($obj->duration timestamp){
                    if(Database::create($obj->data)){
                        array_splice($this->tempDataToFlush, $index, 1);
                    }
                }
            }
        }
    }
}

```

The **writeBack** method calls to the **writeThrough** method, which stores the data to the cache server and a temporary array to be pushed later to the database server using the **updateDatabaseServer** method. You can set up a [CronJob](#) to update the database server every five minutes.

## writeAround

This strategy allows all the **write** operations to go directly to the database server without updating the cache server — only during the **read** operations is the cache server updated.

Assuming a user wants to create a new **Article**, the **Article** stores directly to the database server. When the user wants to read the **Article**'s content for the first time, the **Article** is retrieved from the database server and updates the cache server for subsequent requests.

You can implement this strategy with the Laravel cache with the following code:

```
public function writeAround($data) {  
    $storedData = Database::create($data);  
    return $storedData;  
}  
  
public function readOperation($key, $minutes){  
    $cacheData = Cache::remember($key, $minutes, function() {  
        return Article::all();  
    })  
    return $cacheData;  
}
```

## Cache Aside (Lazy Loading)

The database is sitting aside in this strategy, and the application requests data from the cache server first. Then, if there's a hit (found), the data is returned to the client. Otherwise, if there's a miss (not found), the database server requests the data and updates the cache server for subsequent requests.

You can implement this strategy with the Laravel cache with the following code:

```
public function lazyLoadingStrategy($key, $minutes, $callback) {  
    if (Cache::has($key)) {  
        $data = Cache::get($key);  
    }
```



```
        return $data;
    } else {
        // Database Server is called outside the Cache Server.
        $data = $callback();
        Cache::set($key, $data, $minutes);
        return $data;
    }
}
```

The code above shows the implementation of the cache Aside Strategy, which is equivalent to implementing the **Cache::remember** method.

## Read Through

This strategy is the direct opposite of the cache Aside Strategy. In this strategy, the cache Server sits between the Client Request and the Database Server.

Requests go directly to the cache server, and the cache server is responsible for retrieving the data from the [database server](#) if not found in the cache server.

You can implement this strategy with the Laravel cache with the following code:

```
public function readThrough($key, $minutes) {
    $data = Cache::find($key, $minutes);
    return $data;
}

private function find($key, $minutes){
    if(Cache::has($key)){
        return Cache::get($key);
    }

    // Database Server is called from the Cache Server.
```

```
$DBdata = Database::find($key);  
Cache::put($key, $DBdata, $minutes);  
return $DBdata;  
}
```

There you have it! We've now discussed a few popular caching strategies for your next Laravel application. Remember, you can even use a custom caching strategy that best suits your project requirements.

## Caching the UI Part of a Laravel App

Caching the UI of our Laravel App is a concept known as Full Page cache FPC. The term refers to the process of caching the HTML response from an application.

It's excellent for applications where the dynamic HTML data doesn't change frequently. You can cache the HTML response for a faster overall response and rendering of the HTML.

We can implement FPC with the following line of code:

```
class ArticlesController extends Controller {  
    public function index() {  
        if ( Cache::has('articles_index') ) {  
            return Cache::get('articles_index');  
        } else {  
            $news = News::all();  
            $cachedData = view('articles.index')->with('articles', $news)  
            Cache::put('articles_index', $cachedData);  
            return $cachedData;  
        }  
    }  
}
```

```
}  
  
}
```

At first glance, you might have noticed that we check if that **articles\_index** page already exists in our cache server. Then we return the page by rendering it with Laravel's **view()** and **render()** methods.

Otherwise, we render the page and store the output in our cache server for subsequent requests before returning the rendered page to the browser.

## Build a Laravel App

Now we're going to apply what we've learned so far by creating a new Laravel project and implementing Laravel cache.

If you haven't used Laravel, you can read through [what Laravel is](#) and peek at our list of [excellent Laravel tutorials](#) to get started.

## Setting Up Laravel

First, we're going to create a fresh Laravel instance using the below command. You can look up the official documentation for more.

Open your console and navigate to where you store your PHP projects before running the commands below. Make sure to have [Composer](#) installed and configured correctly.

```
composer create-project laravel/laravel fast-blog-app
```

```
// Change directory to current Laravel installation
```

```
cd fast-blog-app
```

```
// Start Laravel development server.  
php artisan serve
```

## Configuring and Seeding the Database

Next, we will set up our database, create a new **Article** model, and seed 500 fake data points for testing.

Open your [database client](#) and create a new database. We'll do the same with the name **fast\_blog\_app\_db** and then fill up our **.env** file with the database credentials:

```
DB_CONNECTION=mysql  
DB_HOST=127.0.0.1  
DB_PORT=3306  
DB_DATABASE=fast_blog_app_db  
DB_USERNAME=//DB USERNAME HERE  
DB_PASSWORD=//DB PASSWORD HERE
```

Next, we'll run the following command to create the migration and the **Article** model simultaneously:

```
php artisan make:model Article -m
```

Open the newly created migration found `database/migrations/xxx-create-articles-xxx.php` and paste in the following code:

```
<?php
use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;
class CreateArticlesTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('articles', function (Blueprint $table) {
            $table->id();
            $table->string('title');
            $table->text('description');
            $table->timestamps();
        });
    }
    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('articles');
    }
}
```

Next, run the command below to create a new seeder:

```
php artisan make:seeder ArticleSeeder
```

Open the newly created seeder file found in `database/seeder/ArticleSeeder.php` and paste in the following code:

```
<?php
namespace Database\Seeders;
use App\Models\Article;
use Illuminate\Database\Seeder;
class ArticleSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        Article::factory()->count(500)->create();
    }
}
```

Open the **DatabaseSeeder.php** file in the same directory and add the following code:

```
<?php
namespace Database\Seeders;
use Illuminate\Database\Seeder;
```

```
class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {
        $this->call(ArticleSeeder::class);
    }
}
```

Next, run the command below to create a new factory:

```
php artisan make:factory ArticleFactory
```

Open the newly built factory file found in `database/factories/ArticleFactory.php` and paste in the following code:

```
<?php
namespace Database\Factories;
use App\Models\Article;
use Illuminate\Database\Eloquent\Factories\Factory;
class ArticleFactory extends Factory
{
    /**
     * The name of the factory's corresponding model.
     */
}
```

```
*  
* @var string  
*/  
protected $model = Article::class;  
/**  
 * Define the model's default state.  
 *  
 * @return array  
 */  
public function definition()  
{  
    return [  
        'title' => $this->faker->text(),  
        'description' => $this->faker->paragraph(20)  
    ];  
}
```

Now, run the command below to migrate our newly created schema and also seed our fake data for testing:

```
php artisan migrate --seed
```

## Creating the Article Controller

Next, we will create our controller and set up our routes to handle our request and retrieve data using the above model.

Run the following command to create a new **ArticlesController** inside the `app/Http/Controllers` folder:



```
php artisan make:controller ArticlesController --resource
```

Open the file and add the following code to the class:

```
// Returns all 500 articles with Caching
public function index() {
    return Cache::remember('articles', 60, function () {
        return Article::all();
    });
}

// Returns all 500 without Caching
public function allWithoutCache() {
    return Article::all();
}
```

After that, open the **api.php** file found inside the `routes/` folder and paste in the following code to create an endpoint we can call to retrieve our data:

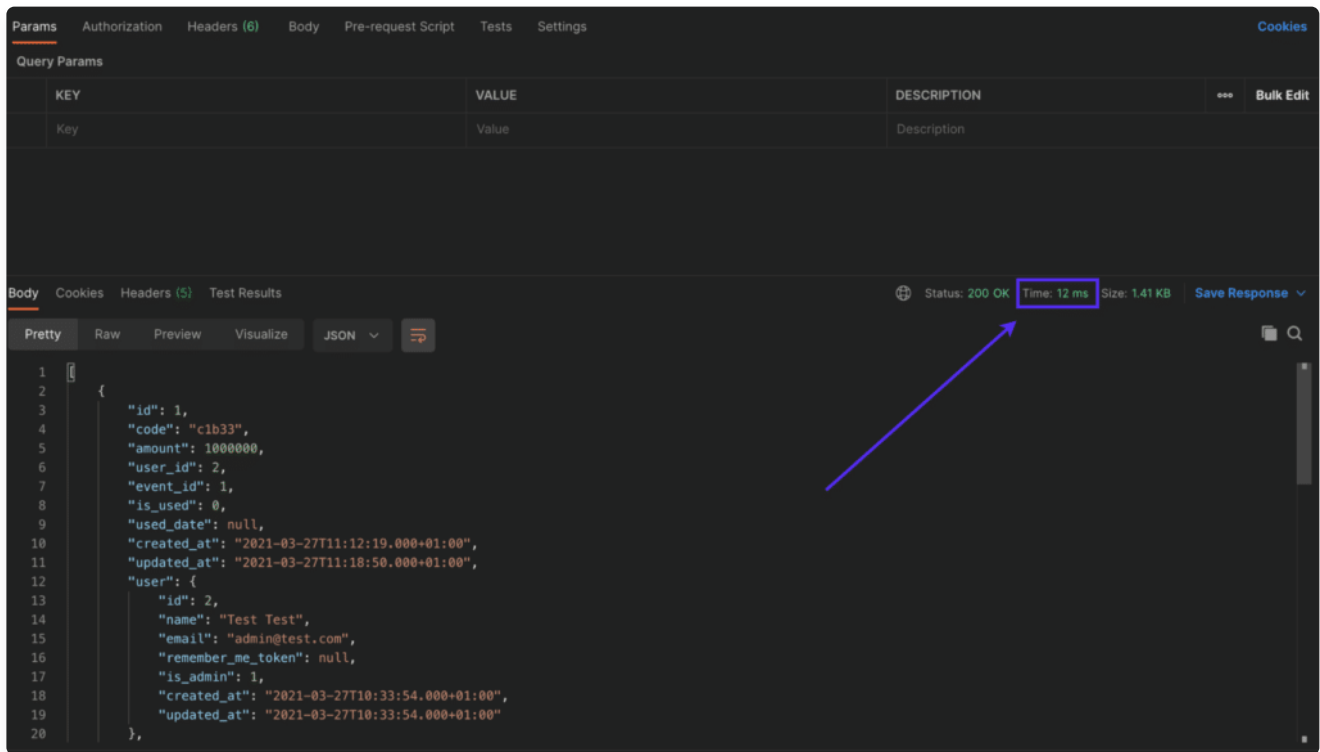
```
Route::get('/articles', 'ArticlesController@index');

Route::get('/articles/withoutcache', 'ArticlesController@allWithoutcache')
```

## Testing the Performance

Lastly, we will [test the performance](#) of our app's response with or without the implementation of the Laravel cache.

This screenshot shows the response time of the API with cache implemented:



— Laravel API response time with cache.

The following screenshot shows the response time of the API without cache implemented — note that the response time has increased over the cached instance by over **5,000%**:

The screenshot shows the 'Body' tab of a web browser's developer tools. The status bar at the top right indicates 'Status: 200 OK', 'Time: 643 ms', 'Size: 1.41 KB', and a 'Save Response' button. The JSON response is as follows:

```

{
  "id": 1,
  "code": "c1b33",
  "amount": 1000000,
  "user_id": 2,
  "event_id": 1,
  "is_used": 0,
  "used_date": null,
  "created_at": "2021-03-27T11:12:19.000+01:00",
  "updated_at": "2021-03-27T11:18:50.000+01:00",
  "user": {
    "id": 2,
    "name": "Test Test",
    "email": "admin@test.com",
    "remember_me_token": null,
    "is_admin": 1,
    "created_at": "2021-03-27T10:33:54.000+01:00",
    "updated_at": "2021-03-27T10:33:54.000+01:00"
  }
}

```

— Laravel API response time without cache.

“ Have a solid grasp of web development and laravel, but want to learn more about laravel caching and how you can implement it? This post is for you

CLICK TO TWEET

## Summary

We’ve explored various strategies for implementing and manipulating Laravel caching by building a new project, benchmarking its responses, and comparing the results.

You’ve also learned how to use the different Laravel caching drivers and methods. In addition, we implemented different caching strategies to help you figure out which one might be right for you.

For more Laravel, explore our hand-picked selection of [the best laravel tutorials](#). Whether you’re a beginner or an advanced Laravel developer, there’s something for everyone in there!

*If you still have questions about Laravel caching, please let us know in the comments section.*

Get all your [applications](#), [databases](#) and [WordPress sites](#) online and under one roof. Our feature-packed, high-performance cloud platform includes:

- Easy setup and management in the MyKinsta dashboard
- 24/7 expert support
- The best Google Cloud Platform hardware and network, powered by Kubernetes for maximum scalability
- An enterprise-level Cloudflare integration for speed and security
- Global audience reach with up to 35 data centers and 275 PoPs worldwide

Get started with a free trial of our [Application Hosting](#) or [Database Hosting](#). Explore our [plans](#) or [talk to sales](#) to find your best fit.

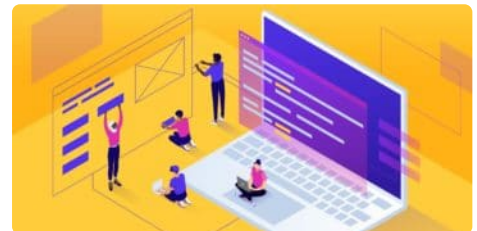
#### Hand-picked related articles

##### BLOG

### 23 Best PHP Editors and IDEs (Free and Premium)

Are you looking for a PHP editor or IDE? Here's a detailed comparison of their key features and some useful advice on how to find the best.

39 MIN READ JANUARY 25, 2021 LEARN PHP



##### BLOG

### 19 Best Laravel Tutorials (Free and Paid Resources in 2023)

Are you thinking of learning Laravel? This guide includes everything you need to find the best Laravel tutorial for any knowledge level.

28 MIN READ OCTOBER 5, 2020 PHP FRAMEWORKS PHP LARAVEL



##### BLOG

### 27 Best Tutorials to Learn PHP in 2023 (Free and Paid Resources)

Now's the time to expand your skill set. This collection of PHP tutorials will help you become a more well-rounded developer.

15 MIN READ MARCH 27, 2020 LEARN PHP



# Comments

[Leave A Comment](#)

## Leave a Reply

**Comment policy:** We love comments and appreciate the time that readers spend to share ideas and give feedback. However, all comments are manually moderated and those deemed to be spam or solely promotional will be deleted.

**Comment**

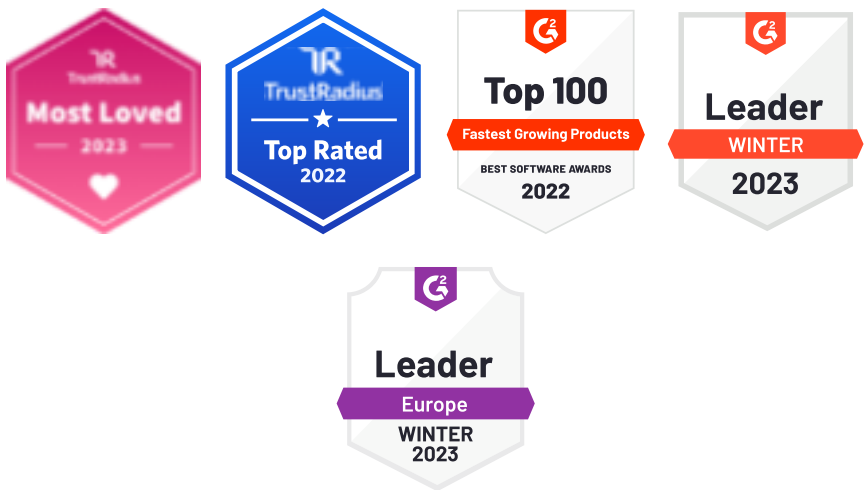
**Name**

**Email**

By submitting this form: You agree to the processing of the submitted personal data in accordance with Kinsta's Privacy Policy, including the transfer of data to the United States.

- ☐ You also agree to receive information from Kinsta related to our services, events, and promotions. You may unsubscribe at any time by following the instructions in the communications received.

Post Comment



Subscribe to our Newsletter


Get premium content from an [award-winning](#) cloud hosting platform.

Subscribe

- Kinsta Hosting
- Platform
- Resources
- Company

Compare Kinsta



 English 

© 2023 Kinsta Inc. All rights reserved. Kinsta® and WordPress® are registered trademarks.  
Legal information.

