VERSION

7.x

Search Docs

Redis

 # Introduction

Configuration

Predis

PhpRedis

 # Interacting With Redis

Pipelining Commands

 # Pub / Sub

Introduction

Redis is an open source, advanced key-value store. It is often referred to as a data structure server since keys can contain strings, hashes, lists, sets, and sorted sets.

Before using Redis with Laravel, we encourage you to install and use the PhpRedis PHP extension via PECL. The extension is more complex to install but may yield better performance for applications that make heavy use of Redis.

Alternatively, you can install the `predis/predis` package via Composer:

```
composer require predis/predis
```

Predis has been abandoned by the package's original author and may be removed from Laravel in a future release.

Configuration

The Redis configuration for your application is located in the `config/database.php` configuration file. Within this file, you will see a `redis` array containing the Redis servers utilized by your application:

```
'redis' => [

    'client' => env('REDIS_CLIENT', 'phpredis'),

    'default' => [
        'host' => env('REDIS_HOST', '127.0.0.1'),
        'password' => env('REDIS_PASSWORD', null),
        'port' => env('REDIS_PORT', 6379),
        'database' => env('REDIS_DB', 0),
    ],

    'cache' => [
        'host' => env('REDIS_HOST', '127.0.0.1'),
        'password' => env('REDIS_PASSWORD', null),
        'port' => env('REDIS_PORT', 6379),
        'database' => env('REDIS_CACHE_DB', 1),
    ],

],
```

The default server configuration should suffice for development. However, you are free to modify this array based on your environment. Each Redis server defined in your configuration file is required to have a name, host, and port.

Configuring Clusters

If your application is utilizing a cluster of Redis servers, you should define these clusters within a `clusters` key of your Redis configuration:

```
'redis' => [

    'client' => env('REDIS_CLIENT', 'phpredis'),

    'clusters' => [
        'default' => [
            [
                'host' => env('REDIS_HOST', 'localhost'),
                'password' => env('REDIS_PASSWORD', null),
                'port' => env('REDIS_PORT', 6379),
                'database' => 0,
            ],
        ],
    ],

],
```

By default, clusters will perform client-side sharding across your nodes, allowing you to pool nodes and create a large amount of available RAM. However, note that client-side sharding does not handle failover; therefore, is primarily suited for cached data that is available from another primary data store. If you would like to use native Redis clustering, you should specify this in the `options` key of your Redis configuration:

```
'redis' => [

    'client' => env('REDIS_CLIENT', 'phpredis'),

    'options' => [
        'cluster' => env('REDIS_CLUSTER', 'redis'),
    ],
```

```
    'clusters' => [
        // ...
    ],

],
```

Predis

To utilize the Predis extension, you should change the `REDIS_CLIENT` environment variable from `phpredis` to `predis`:

```
'redis' => [

    'client' => env('REDIS_CLIENT', 'predis'),

    // Rest of Redis configuration...
],
```

In addition to the default `host`, `port`, `database`, and `password` server configuration options, Predis supports additional connection parameters that may be defined for each of your Redis servers. To utilize these additional configuration options, add them to your Redis server configuration in the `config/database.php` configuration file:

```
'default' => [
    'host' => env('REDIS_HOST', 'localhost'),
    'password' => env('REDIS_PASSWORD', null),
    'port' => env('REDIS_PORT', 6379),
    'database' => 0,
    'read_write_timeout' => 60,
],
```

PhpRedis

The PhpRedis extension is configured as default at `REDIS_CLIENT` env and in your `config/database.php`:

```
'redis' => [

    'client' => env('REDIS_CLIENT', 'phpredis'),

    // Rest of Redis configuration...
],
```

If you plan to use PhpRedis extension along with the `Redis` Facade alias, you should rename it to something else, like `RedisManager`, to avoid a collision with the Redis class. You can do that in the aliases section of your `app.php` config file.

```
'RedisManager' => Illuminate\Support\Facades\Redis::class,
```

In addition to the default `host`, `port`, `database`, and `password` server configuration options, PhpRedis supports the following additional connection parameters: `persistent`, `prefix`, `read_timeout` and `timeout`. You may add any of these options to your Redis server configuration in the `config/database.php` configuration file:

```
'default' => [
    'host' => env('REDIS_HOST', 'localhost'),
    'password' => env('REDIS_PASSWORD', null),
    'port' => env('REDIS_PORT', 6379),
    'database' => 0,
    'read_timeout' => 60,
],
```

The Redis Facade

To avoid class naming collisions with the Redis PHP extension itself, you will need to delete or rename the `Illuminate\Support\Facades\Redis` facade alias from your `app` configuration file's `aliases` array. Generally, you should remove this alias entirely and only reference the facade by its fully qualified class name while using the Redis PHP extension.

Interacting With Redis

You may interact with Redis by calling various methods on the `Redis` facade. The `Redis` facade supports dynamic methods, meaning you may call any Redis command on the facade and the command will be passed directly to Redis. In this example, we will call the Redis `GET` command by calling the `get` method on the `Redis` facade:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\Redis;

class UserController extends Controller
{
    /**
     * Show the profile for the given user.
     *
     * @param  int  $id
     * @return Response
     */
    public function showProfile($id)
    {
        $user = Redis::get('user:profile:'.$id);

        return view('user.profile', ['user' => $user]);
    }
}
```

As mentioned above, you may call any of the Redis commands on the `Redis` facade. Laravel uses magic methods to pass the commands to the Redis server, so pass the arguments the Redis command expects:

```
Redis::set('name', 'Taylor');

$values = Redis::lrange('names', 5, 10);
```

Alternatively, you may also pass commands to the server using the `command` method, which accepts the name of the command as its first argument, and an array of values as its second argument:

```
$values = Redis::command('lrange', ['name', 5, 10]);
```

Using Multiple Redis Connections

You may get a Redis instance by calling the `Redis::connection` method:

```
$redis = Redis::connection();
```

This will give you an instance of the default Redis server. You may also pass the connection or cluster name to the `connection` method to get a specific server or cluster as defined in your Redis configuration:

```
$redis = Redis::connection('my-connection');
```

Pipelining Commands

Pipelining should be used when you need to send many commands to the server. The `pipeline` method accepts one argument: a `Closure` that receives a

Redis instance. You may issue all of your commands to this Redis instance and they will all be streamed to the server thus providing better performance:

```
Redis::pipeline(function ($pipe) {
    for ($i = 0; $i < 1000; $i++) {
        $pipe->set("key:$i", $i);
    }
});
```

Pub / Sub

Laravel provides a convenient interface to the Redis `publish` and `subscribe` commands. These Redis commands allow you to listen for messages on a given "channel". You may publish messages to the channel from another application, or even using another programming language, allowing easy communication between applications and processes.

First, let's setup a channel listener using the `subscribe` method. We'll place this method call within an Artisan command since calling the `subscribe` method begins a long-running process:

```
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;
use Illuminate\Support\Facades\Redis;

class RedisSubscribe extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
```

```
    protected $signature = 'redis:subscribe';

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Subscribe to a Redis channel';

    /**
     * Execute the console command.
     *
     * @return mixed
     */
    public function handle()
    {
        Redis::subscribe(['test-channel'], function ($message) {
            echo $message;
        });
    }
}
```

Now we may publish messages to the channel using the `publish` method:

```
Route::get('publish', function () {
    // Route logic...

    Redis::publish('test-channel', json_encode(['foo' => 'bar']));
});
```

Wildcard Subscriptions

Using the `psubscribe` method, you may subscribe to a wildcard channel, which may be useful for catching all messages on all channels. The `$channel` name will be passed as the second argument to the provided callback `Closure`:

```php
Redis::psubscribe(['*'], function ($message, $channel) {
    echo $message;
});

Redis::psubscribe(['users.*'], function ($message, $channel) {
    echo $message;
});
```


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 +

Resources +

Partners +

Ecosystem +

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.

Laravel is a Trademark of Taylor Otwell.
Copyright © 2011-2020 Laravel LLC.