Note (Very Important): For Storage::disk('disk-name-here') we must understand the path of each disk.

****

```php
// 1--> send request to another server.
    public function testSendRequest1() {
        $response = Http::get('http://jsonplaceholder.typicode.com/posts');

        $jsonData = $response->json();

        return response($jsonData, 200);
    }
```

****

```php
// 2--> send request to another server with post.
    public function testSendRequest2(){
        $response = Http::post(
            'http://jsonplaceholder.typicode.com/posts',
            [
                'title' => 'Jafar Test POST Guzzle',
                'body' => 'This is For Testing only',
            ]
        );

        // This will return a valid object if success
        // with id to it.
        return response($response->json(), 200);

        // $response->successful(); // This will return 1 if success.
    }
```

****

To Make PDF of our views:

- Install dompdf: ***composer require barryvdh/Laravel-dompdf***
- Add packages to ***config/app.php***:
  - In provider array: ***Barryvdh\DomPDF\ServiceProvider::class***.
  - In Aliases array: 'PDF' => ***Barryvdh/DomPDF\Facade::class***.

Then publish the package with: php artisan vendor:publish

After that choose the right number of package.

The Result:

Copied File [\vendor\barryvdh\laravel-dompdf\config\dompdf.php] To [\config\dompdf.php]

Publishing complete.

```php
// 1--> Generate pdf data file
public function testGeneratePDF() {
    $data = [
        'name_1' => 'Jafar Hajji',
        'name_2' => 'Jafar Loka',
        'name_3' => 'Ali Loka',
        'name_4' => 'Khader',
    ];

    $pdf = PDF::loadView('pdf.pdf', $data);

    return $pdf->download('jafar_test.pdf');
}
```

Note: In the View File we access to params using its name not using array.

```
<body>
    <h1>{{ $name_1}} </h1>
    <h1>{{ $name_2}} </h1>
    <h1>{{ $name_3}} </h1>
    <h1>{{ $name_4}} </h1>
</body>
```

****

```
// 1--> Generate pdf data file
    public function testGeneratePDF() {
        $data = [
            'name_1' => 'Jafar Hajji',
            'name_2' => 'Jafar Loka',
            'name_3' => 'Ali Loka',
            'name_4' => 'Khader',
        ];

        $pdf = PDF::loadView('pdf.pdf', $data);

        // This is not working.
        // $pdf->saveAs(public_path('pdf'));

        $content = $pdf->download()->getOriginalContent();

        // This will save in storag/app/pdf
        Storage::put("pdf/jafar_test.pdf", $content);

        return $pdf->download('jafar_test.pdf');
    }
```

****

In your config/queue.php configuration file, there is a connections configuration array. This option defines the connections to backend queue services such as Amazon SQS, Beanstalk, or Redis. However, any given queue connection may have multiple "queues" which may be thought of as different stacks or piles of queued jobs.

****

```
'product_connection_2' => [
        'driver' => 'redis', // we change this from database
        // to redis

        // This is the connection of database.
        // 'connection' => 'default',
        'table' => 'jobs',
        'queue' => 'products',
        'retry_after' => 90,
        'block_for' => null,
    ],
```

```
G:\Web\Laravel\Test_Project\Test1>php artisan make:job BJWithRedisCategory
PHP Warning:  Version warning: Imagick was compiled against ImageMagick version 1799 but versi
on 1808 is loaded. Imagick will run but may behave surprisingly in Unknown on line 0

Warning: Version warning: Imagick was compiled against ImageMagick version 1799 but version 18
08 is loaded. Imagick will run but may behave surprisingly in Unknown on line 0
Job created successfully.
```

```
G:\Web\Laravel\Test_Project\Test1>php artisan make:controller API/BJWithRedis
PHP Warning:  Version warning: Imagick was compiled against ImageMagick version 1799 but versi
on 1808 is loaded. Imagick will run but may behave surprisingly in Unknown on line 0

Warning: Version warning: Imagick was compiled against ImageMagick version 1799 but version 18
08 is loaded. Imagick will run but may behave surprisingly in Unknown on line 0
Controller created successfully.
```

****

Some applications may not need to ever push jobs onto multiple queues, instead preferring to have one simple queue. However, pushing jobs to multiple queues can be especially useful for applications that wish to prioritize or segment how jobs are processed, since the Laravel queue worker allows you to specify which queues it should process by priority. For example, if you push jobs to a high queue, you may run a worker that gives them higher processing priority:

```
php artisan queue:work --queue=high,default
```

****

Finally, don't forget to instruct your application to use the `database` driver by updating the `QUEUE_CONNECTION` variable in your application's `.env` file:

```
QUEUE_CONNECTION=database
```

****

In order to use the `redis` queue driver, you should configure a Redis database connection in your `config/database.php` configuration file.

```php
'redis' => [

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

    'options' => [
        'cluster' => env('REDIS_CLUSTER', 'redis'),
        //'prefix' => env('REDIS_PREFIX', Str::slug(env('APP_NAME', 'laravel'), '_'
    ],

    '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'),
    ],

    'cache' => [
        '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_CACHE_DB', '1'),
    ],

],
```

****

Redis Cluster

If your Redis queue connection uses a Redis Cluster, your queue names must contain a key hash tag. This is required in order to ensure all of the Redis keys for a given queue are placed into the same hash slot:

```
'redis' => [
    'driver' => 'redis',
    'connection' => 'default',
    'queue' => '{default}',
    'retry_after' => 90,
],
```

****

Blocking

When using the Redis queue, you may use the `block_for` configuration option to specify how long the driver should wait for a job to become available before iterating through the worker loop and re-polling the Redis database.

Adjusting this value based on your queue load can be more efficient than continually polling the Redis database for new jobs. For instance, you may set the value to `5` to indicate that the driver should block for five seconds while waiting for a job to become available:

```
'redis' => [
    'driver' => 'redis',
    'connection' => 'default',
    'queue' => 'default',
    'retry_after' => 90,
    'block_for' => 5,
],
```

****

https://laravel.com/docs/10.x/queues#other-driver-prerequisites

****

Other Driver Prerequisites

The following dependencies are needed for the listed queue drivers. These dependencies may be installed via the Composer package manager:

- Amazon SQS: `aws/aws-sdk-php ~3.0`
- Beanstalkd: `pda/pheanstalk ~4.0`
- Redis: `predis/predis ~1.0` or phpredis PHP extension

****

```php
class ProcessPodcast implements ShouldQueue
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    /**
     * Create a new job instance.
     */
    public function __construct(
        public Podcast $podcast,
    ) {}

    /**
     * Execute the job.
     */
    public function handle(AudioProcessor $processor): void
    {
        // Process uploaded podcast...
    }
}
```

In this example, note that we were able to pass an Eloquent model directly into the queued job's constructor. Because of the SerializesModels trait that the job is using, Eloquent models and their loaded relationships will be gracefully serialized and unserialized when the job is processing.

If your queued job accepts an Eloquent model in its constructor, only the identifier for the model will be serialized onto the queue. When the job is actually handled, the queue system will automatically re-retrieve the full model instance and its loaded relationships from the database. This approach to model serialization allows for much smaller job payloads to be sent to your queue driver.

The handle method is invoked when the job is processed by the queue. Note that we are able to type-hint dependencies on the handle method of

the job. The Laravel service container automatically injects these dependencies.

If you would like to take total control over how the container injects dependencies into the handle method, you may use the container's bindMethod method. The bindMethod method accepts a callback which receives the job and the container. Within the callback, you are free to invoke the handle method however you wish. Typically, you should call this method from the boot method of your App\Providers\AppServiceProvider service provider:

```
use App\Jobs\ProcessPodcast;

use App\Services\AudioProcessor;

use Illuminate\Contracts\Foundation\Application;


$this->app->bindMethod([ProcessPodcast::class, 'handle'], function (ProcessPodcast $job, Application $app) {
    return $job->handle($app->make(AudioProcessor::class));
});
```

Binary data, such as raw image contents, should be passed through the base64_encode function before being passed to a queued job. Otherwise, the job may not properly serialize to JSON when being placed on the queue.

****

Queued Relationships

Because loaded relationships also get serialized, the serialized job string can sometimes become quite large. To prevent relations from being serialized, you can call the withoutRelations method on the model when setting a property value. This method will return an instance of the model without its loaded relationships:

****

Furthermore, when a job is deserialized and model relationships are re-retrieved from the database, they will be retrieved in their entirety. Any previous relationship constraints that were applied before the model was serialized during the job queueing process will not be applied when the job is deserialized. Therefore, if you wish to work with a subset of a given relationship, you should re-constrain that relationship within your queued job.

****

Unique Jobs

Unique jobs require a cache driver that supports locks. Currently, the memcached, redis, dynamodb, database, file, and array cache drivers support atomic locks. In addition, unique job constraints do not apply to jobs within batches.

Sometimes, you may want to ensure that only one instance of a specific job is on the queue at any point in time. You may do so by implementing the ShouldBeUnique interface on your job class. This interface does not require you to define any additional methods on your class:

```php
<?php

use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Contracts\Queue\ShouldBeUnique;

class UpdateSearchIndex implements ShouldQueue, ShouldBeUnique
{
    ...
}
```

In the example above, the UpdateSearchIndex job is unique. So, the job will not be dispatched if another instance of the job is already on the queue and has not finished processing.

In certain cases, you may want to define a specific "key" that makes the job unique or you may want to specify a timeout beyond which the job no longer stays unique. To accomplish this, you may define `uniqueId` and `uniqueFor` properties or methods on your job class:

```
use App\Product;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Contracts\Queue\ShouldBeUnique;
class UpdateSearchIndex implements ShouldQueue, ShouldBeUnique
{
    /**
     * The product instance.
     *
     * @var \App\Product
     */
    public $product;

    /**
     * The number of seconds after which the job's unique lock will be released.
     *
     * @var int
     */
    public $uniqueFor = 3600;

    /**
     * The unique ID of the job.
     */
```

```
    public function uniqueId(): string
    {
        return $this->product->id;
    }
}
```

In the example above, the **UpdateSearchIndex job** is unique by a product ID. So, any new dispatches of the job with the same product ID will be ignored until the existing job has completed processing. In addition, if the existing job is not processed within one hour, the unique lock will be released and another job with the same unique key can be dispatched to the queue.

Note (Very Important): If your application dispatches jobs from multiple web servers or containers, you should ensure that all of your servers are communicating with the same central cache server so that Laravel can accurately determine if a job is unique.

****

Keeping Jobs Unique Until Processing Begins

By default, unique jobs are "unlocked" after a job completes processing or fails all of its retry attempts. However, there may be situations where you would like your job to unlock immediately before it is processed. To accomplish this, your job should implement the `ShouldBeUniqueUntilProcessing` contract instead of the `ShouldBeUnique` contract:

****

Note: If you only need to limit the concurrent processing of a job, use the `WithoutOverlapping` job middleware instead.

****

Job Middleware

Job middleware allow you to wrap custom logic around the execution of queued jobs, reducing boilerplate in the jobs themselves. For example, consider the following handle method which leverages Laravel's Redis rate limiting features to allow only one job to process every five seconds:

```
use Illuminate\Support\Facades\Redis;


/**
 * Execute the job.
 */
public function handle(): void
{
    Redis::throttle('key')->block(0)->allow(1)->every(5)->then(function ()
{
        info('Lock obtained...');

        // Handle job...
    }, function () {
        // Could not obtain lock...

        return $this->release(5);
    });
}
```

****

While this code is valid, the implementation of the `handle` method becomes noisy since it is cluttered with Redis rate limiting logic. In addition, this rate limiting logic must be duplicated for any other jobs that we want to rate limit.

Instead of rate limiting in the handle method, we could define a job middleware that handles rate limiting. Laravel does not have a default location for job middleware, so you are welcome to place job middleware anywhere in your application. In this example, we will place the middleware in an `app/Jobs/Middleware` directory:

```
use Closure;
use Illuminate\Support\Facades\Redis;


class RateLimited
{
    /**
     * Process the queued job.
     *
     * @param  \Closure(object): void  $next
     */
    public function handle(object $job, Closure $next): void
    {
        Redis::throttle('key')
                ->block(0)->allow(1)->every(5)
                ->then(function () use ($job, $next) {
                    // Lock obtained...
```

```
            $next($job);
        }, function () use ($job) {
            // Could not obtain lock...

            $job->release(5);
        });
    }
}
```

As you can see, like route middleware, job middleware receive the job being processed and a callback that should be invoked to continue processing the job.

****

As you can see, like route middleware, job middleware receive the job being processed and a callback that should be invoked to continue processing the job.

****

After creating job middleware, they may be attached to a job by returning them from the job's middleware method. This method does not exist on jobs scaffolded by the make:job Artisan command, so you will need to manually add it to your job class:

```
use App\Jobs\Middleware\RateLimited;


/**
 * Get the middleware the job should pass through.
 *
 * @return array<int, object>
 */
public function middleware(): array
{
    return [new RateLimited];
}
```

****

Rate Limiting

For example, you may wish to allow users to backup their data once per hour while imposing no such limit on premium customers. To accomplish this, you may define a RateLimiter in the boot method of your AppServiceProvider:

```
use Illuminate\Cache\RateLimiting\Limit;

use Illuminate\Support\Facades\RateLimiter;


/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    RateLimiter::for('backups', function (object $job) {
        return $job->user->vipCustomer()
                    ? Limit::none()
                    : Limit::perHour(1)->by($job->user->id);
    });
}
```

you may easily define a rate limit based on minutes using the perMinute method. In addition, you may pass any value you wish to the by method of the rate limit; however, this value is most often used to segment rate limits by customer:

```
return Limit::perMinute(50)->by($job->user->id);
```

****