Before getting started with Laravel queues, it is important to understand the distinction between "connections" and "queues". 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.

****

Note that each connection configuration example in the `queue` configuration file contains a `queue` attribute. This is the default queue that jobs will be dispatched to when they are sent to a given connection. In other words, if you dispatch a job without explicitly defining which queue it should be dispatched to, the job will be placed on the queue that is defined in the `queue` attribute of the connection configuration:

****

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

****

Note: to use redis in cache it used db number 2.

****

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,
],
```

****

Warning

Setting `block_for` to `0` will cause queue workers to block indefinitely until a job is available. This will also prevent signals such as `SIGTERM` from being handled until the next job has been processed.

****

php artisan queue:work backups --queue=backups

php artisan queue:retry all

****

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.

****

Note (Very Important):

Warning

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:

```
/**
 * Create a new job instance.
 */
public function __construct(Podcast $podcast)
{
    $this->podcast = $podcast->withoutRelations();
}
```

****

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.

****

If a job receives a collection or array of Eloquent models instead of a single model, the models within that collection will not have their relationships restored when the job is deserialized and executed. This is to prevent excessive resource usage on jobs that deal with large numbers of models.

****

Unique Jobs

Warning
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:

```
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Contracts\Queue\ShouldBeUnique; // this is from Laravel 10

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

****

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:

```
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 to Microservices):

Warning

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:

```
use App\Product;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Contracts\Queue\ShouldBeUniqueUntilProcessing;

class UpdateSearchIndex implements ShouldQueue,
ShouldBeUniqueUntilProcessing
{
    // ...
}
```

****

Unique Job Locks:

Behind the scenes, when a `ShouldBeUnique` job is dispatched, Laravel attempts to acquire a lock with the `uniqueId` key. If the lock is not acquired, the job is not dispatched. This lock is released when the job completes processing or fails all of its retry attempts. By default, Laravel will use the default cache driver to obtain this lock. However, if you wish to use another driver for acquiring the lock, you may define a `uniqueVia` method that returns the cache driver that should be used: (Inside the job we define the method):

```
public function uniqueVia(): Repository
{
    return Cache::driver('redis');
}
```

****

Note

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

****

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:

```
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);
    });
}
```

****

Laravel does not have a default location for job middleware.

****

Job middleware can also be assigned to queueable event listeners, mailables, and notifications.

****

Laravel actually includes a rate limiting middleware that you may utilize to rate limit jobs. Like route rate limiters, job rate limiters are defined using the **RateLimiter** facade's **for** method.

****

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**

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

****

In the example above, we defined an hourly rate limit; however, 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);
```

****

Once you have defined your rate limit, you may attach the rate limiter to your backup job using
the **Illuminate\Queue\Middleware\RateLimited** middleware. Each time the job exceeds the rate limit, this middleware will release the job back to the queue with an appropriate delay based on the rate limit duration.

```php
use Illuminate\Queue\Middleware\RateLimited;

public function middleware(): array
{
    return [new RateLimited('backups')];
}
```

****

Releasing a rate limited job back onto the queue will still increment the job's total number of attempts. You may wish to tune
your tries and **maxExceptions properties** on your job class accordingly. Or, you may wish to use the retryUntil method to define the amount of time until the job should no longer be **attempted**.

If you do not want a job to be retried when it is rate limited, you may use the **dontRelease** method:

```php
public function middleware(): array
{
    return [(new RateLimited('backups'))->dontRelease()];
}
```

****

If you are using Redis, you may use
the **Illuminate\Queue\Middleware\RateLimitedWithRedis** middleware, which is fine-tuned for Redis and more efficient than the basic rate limiting middleware.

****