Event Loop

Event -> Listener -> Job -> Command - 2

Sidrit Trandafili

@sidis405

@strandafili

- Sviluppatore PHP 15+ anni
- Laravel da 4 anni
- Back-end
  - php
  - java
  - node
  - python
- Front-end
  - VueJS
  - Vanilla JS
  - TailwindCSS
  - Sass
- Devops + Sysops da 13 anni

Events, Listener, Job, Command - 2

Events, Listener, Job, Command - 2

“ Nella prima sezione riguardo all'Event Loop di Laravel, abbiamo parlato di Events e Listeners. Adesso parleremo di Jobs. Per spiegare al meglio questo concetto, dobbiamo inevitabilmente parlare delle queues (code di elaborazione).

*“ Le code di elaborazione in **Laravel** vengono implementate per rinviare i task impegnativi a livello computazionale (Jobs) o che richiedono tempo (es: invio mail) a dei meccanismi paralleli, permettendo un'esperienza più fluida nell'uso dell'applicazione*

“ Le code di elaborazione supportate da Laravel di default sono 'sync', 'database', 'beanstalkd', 'sqs', 'redis' e 'null' e sono gestite in 'config/queue.php'.

“ La tipologia attiva al momento dell'istallazione di una nuova copia di Laravel è sync

Events, Listener, Job, Command - 2

Riprendiamo il nostro esempio (parte 1) e creiamo una classe Job per inviare un'email agli amministratori dell'applicazione che un post è stato aggiornato.

Creiamo un job NootifyAdmins

```
php artisan make:job NotifyAdmins // app/Jobs/NotifyAdmins.php
```

Nel nostro PostUpdateListener lanciamo questo Job appena creato passandogli il post da processare

```
<?php

namespace App\Listeners;

use App\Jobs\NotifyAdmins;
use App\Events\PostWasUpdated;
class PostUpdateListener
{
   [..]
    public function handle(PostWasUpdated $event)
    {
        NotifyAdmins::dispatch($event->post);
    }
}
```

Events, Listener, Job, Command - 2

Vediamo la nuova classe job che abbiamo appena creato

1. indichiamo che il job deve essere messo in coda
2. definiamo un $post localmente
3. riceviamo il post nel costruttore e lo assegniamo al post locale
4. nella funzione handle() eseguiamo il lavoro

```php
<?php

namespace App\Jobs;

use App\Post;
use Illuminate\Bus\Queueable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;

class NotifyAdmins implements ShouldQueue
{
    protected $post;
    use Dispatchable, InteractsWithQueue,
        Queueable, SerializesModels;

    public function __construct(Post $post)
    {
        $this->post = $post;
    }

    public function handle()
    {
        echo("Dentro la classe Job - " . date('h:i:s') . "\n");
        sleep(3); //simulazione
        echo("Job Eseguito " . date('h:i:s') . "\n");
    }
}
```

Events, Listener, Job, Command - 2

Per poter provare che il nostro sistema stia infatti completando il loop che parte dall'evento fino al job appena creato, possiamo usare tinker

```
>>> $post = Post::find(2); // troviamo un post da aggiornare
```

```
event(new App\Events\PostWasUpdated($post)); // emettiamo evento di aggiornamento
```

```
Dentro la classe Job - 12:53:09 // inizio esecuzione

//passa il tempo di elaborazione come definito da noi, 3 secondi

Job Eseguito 12:53:12 // esecuzione completa
=> [
     null,
   ]
>>>
```

L'osservazione più importante da fare è la seguente: l'evento PostWasUpdated parte appena il post viene aggiornato nel controller o repo, ma più importante è il fatto che il Job scatterà prima che l'utente ricarichi la pagina. Conoscendo il modo in cui funziona il call stack in php, tutto ciò implica che prima di vedere la pagina ricaricare, l'utente dovrà aspettare ben 3 secondi.

Events, Listener, Job, Command - 2

Quello che abbiamo definito succede appunto perché la tipologia di coda di elaborazione attivata di default su Laravel, è 'sync'. Questo parametro si trova nel file .env

Implementiamo una coda 'redis' e una 'database'.

> *Redis è un database open-source di tipo chiave-valore e salva i dati in memoria*

```
QUEUE_DRIVER=redis
```

Per una coda redis, Laravel richiede l'istallazione di Predis e un server redis accessibile tramite parametri definiti nel file .env

```
composer require predis/predis
```

```
REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379
```

Per una coda database, Laravel richiede la presenza di una tabella 'jobs' (nome configurabile in config/queue.php). Per la creazione di questa tabella ci aiuta artisan

```
php artisan queue:table
```

Events, Listener, Job, Command - 2

Da questo punto in poi, tutti i comandi e le considerazioni valgono per tutti e due i tipi di code.

Per la gestione delle queues, artisan ci mette a disposizione vari comandi

```
queue
  queue:failed          List all of the failed queue jobs
  queue:failed-table    Create a migration for the failed queue jobs database table
  queue:flush           Flush all of the failed queue jobs
  queue:forget          Delete a failed queue job
  queue:listen          Listen to a given queue
  queue:restart         Restart queue worker daemons after their current job
  queue:retry           Retry a failed queue job
  queue:table           Create a migration for the queue jobs database table
  queue:work            Start processing jobs on the queue as a daemon
```

Per la gestione delle queues, artisan ci mette a disposizione vari comandi. Il comando che ci interessa è work.

```
php artisan queue:work
```

Events, Listener, Job, Command - 2

Rivediamo il flusso del nostro loop adesso con questa nuova configurazione

```
php artisan tinker

>>> $post = Post::find(2);

>>> event(new App\Events\PostWasUpdated($post));

=> [ // LA RISPOSTA È IMMEDIATA
     null,
   ]
>>>
```

```
php artisan queue:work

[2018-01-31 13:19:38] Processing: App\Jobs\NotifyAdmins

Dentro la classe Job - 01:19:38

Job Eseguito 01:19:41

[2018-01-31 13:19:41] Processed:  App\Jobs\NotifyAdmins
```

Per tenere le code di elaborazione attive, lo standard è usare supervisord

https://laravel.com/docs/5.5/queues#supervisor-configuration