CODE SMART

GUIDA ALLO SVILUPPO DI APPLICAZIONI WEB CON *LARAVEL 5*, PER PRINCIPIANTI

DAYLE REES & FRANCESCO MALATESTA

Laravel: Code Smart (IT)

Sviluppo di Applicazioni Web con il Framework Laravel 5.

Dayle Rees e Francesco Malatesta

Questo libro è in vendita presso http://leanpub.com/codesmart-it

Questa versione è stata pubblicata il 2016-10-20

Indice

Ringraziamenti

Innanzitutto, vorrei ringraziare la mia ragazza Emma, per supportarmi in tutte le mie avventure piÃ¹ nerd… e non solo: ha anche scattato la foto che vedi sulla copertina del libro! Ti amo, Emma!

Grazie anche ai miei genitori, che mi hanno sempre sostenuto durante questi trentadue anni di vita. Ah, e grazie anche per aver comprato un miliardo di copie dei primi libri per tutti i membri della famiglia!

Taylor Otwell: il viaggio che sto facendo con Laravel Ã¨ qualcosa di incredibile. Grazie di cuore per avermi dato l'opportunitÃ di far parte del team, ed anche per la tua amicizia! Grazie soprattutto per aver creato un framework piacevole da usare, che rende il nostro codice poesia. Grazie per averci investito cosÃ¬ tanto tempo e passione nel suo sviluppo.

Grazie anche a te, lettore che ha comprato questo (e magari uno dei precedenti) libri e a tutta la community di Laravel. Senza il vostro supporto, tutto questo non sarebbe possibile.

Errata

Code Smart è il mio quarto libro. Dal primo, per fortuna, sono migliorato notevolmente. Tuttavia, sono assolutamente certo che ci saranno degli errori. Non ho un publisher ed un team per la review. Tantomeno una laurea in inglese. Per questo motivo ho fatto il possibile per rendere presentabile il lavoro, ma ti prego di avere pazienza in caso trovassi un errore. Sentiti libero di mandarmi una mail (in inglese) con tutti i dettagli a me@daylerees.com[1]! Sarò felice di sistemarlo.

Tutti gli errori segnalati verranno sistemati quanto prima, e tutte le piccole sistemazioni verranno quindi rilasciate nelle prossime release del libro.

[1] mailto:me@daylerees.com

Feedback

Non solo errori: per qualsiasi cosa scrivimi senza problemi, in inglese, a me@daylerees.com[2]. Puoi anche contattarmi su twitter: sono @daylerees[3].

[2] mailto:me@daylerees.com
[3] https://twitter.com/daylerees

Traduzioni

Vuoi tradurre Code Smart nella tua lingua? Grande! Mandami una mail all'indirizzo me@daylerees.com[4] con la tua proposta. La mia offerta consiste nel fare a metà (si, cinquanta e cinquanta) con i profitti delle copie tradotte. Il prezzo sarà lo stesso della copia inglese.

Attenzione: il libro è stato scritto in markdown, e tutto il testo è sotto version control su Github[5].

[4] mailto:me@daylerees.com
[5] http://github.com

Introduzione

Ehilà! Ciao! Piacere di conoscerti.

Il mio nome è Dayle, e sono uno sviluppatore. Ho da poco iniziato il mio quarto decennio su questo pianeta, e ti porto i miei omaggi dalle terre del Galles. Alcuni anni fa, sono stato uno dei primi ad usare quello che all'epoca era un framework da poco entrato in circolazione nel mondo PHP: Laravel. Dopo qualche prova e tanto divertimento, ho immediatamente iniziato a costruirci applicazioni. Laravel è pulito, conciso, meraviglioso. Ci ho visto un potenziale, e col senno di poi… avevo ragione!

Da allora, per Laravel sono stato un core contributor, uno speaker, un consulente ed un autore di libri. Svariati libri e si, tutti dedicati a lui. Con ottime reazioni da parte del pubblico, per fortuna, e migliaia di copie vendute! Un'esperienza meravigliosa, davvero. Se ti è già capitato di leggere uno dei miei libri, in precedenza… grazie! In caso contrario, lascia che ti spieghi come funziona il tutto.

Non sono un autore nel senso tradizionale del termine. Non riesco a complicare le cose, mi piace tenerle semplici. Seguendo questa linea scrivo i miei libri, come se stessi al pub, con un amico, a raccontargli una storia. Tutto qui!

Questo libro è il terzo della mia serie dedicata a Laravel, e copre le versioni 5.x del framework. Libro dopo libro, posso dire di aver imparato qualcosina sulla scrittura: spero per te che quella che stai per iniziare sarà una piacevole avventura. Nel caso in cui dovessero esserci dei problemi, sentiti libero di mandarmi una mail! Risponderò quanto prima.

Detto questo, ci siamo. Eccitato? Laravel è un framework davvero grande, che consente di fare un sacco di cose. Non temere: faremo un passo alla volta, ed in men che non si dica avrai già costruito le tue prime applicazioni!

Iniziamo!

Installazione

Lo so, non vedi l'ora di cominciare ad usare Laravel. Ad ogni modo, visto che stiamo parlando di un web framework, la prima cosa di cui avrai bisogno sarà un web server ed un ambiente di lavoro appropriato. In questo capitolo vedremo un po' come prepararli.

> Ehi, ma io non ho il tempo di preparare tutto questo!

Non temere: non intendevo quello che forse stai pensando. Conosci Vagrant? Beh, è un meraviglioso software che ti consente di creare macchine virtuali velocemente, e da linea di comando. Homestead è una box di Vagrant creata appositamente per Laravel. In poche parole, ti consentirà di creare al volo una macchina su cui lavorare senza andare a "sporcare" l'ambiente locale.

Chiaramente, non è l'unico modo: ma è il mio preferito, onestamente. Ha tutto il necessario per avviare il framework e lavorarci. Certo, si tratta sempre di qualcosa in più da studiare, ma ne varrà la pena, fidati! Se dovesse interessarti, inoltre, il prossimo capitolo riguarda Valet, un eseguibile che ti consentirà di iniziare a lavorare con Laravel senza troppi fronzoli ed in pochissimo tempo, usando la versione di PHP installata in locale. Al momento, tuttavia, funziona solo per Mac. Se non stai usando un Mac o vuoi semplicemente seguire il mio consiglio, continua a leggere qui. In caso contrario, salta questa parte e passa a Valet!

Installare le Dipendenze

Per iniziare a lavorare avrai bisogno di installare alcune "dipendenze". Eccole:

- PHP http://www.php.net/[6]
- Git https://git-scm.com/[7]
- Composer https://getcomposer.org/[8]
- Virtualbox https://www.virtualbox.org/[9]
- Vagrant https://www.vagrantup.com/[10]

[6] http://www.php.net/
[7] https://git-scm.com/
[8] https://getcomposer.org/
[9] https://www.virtualbox.org/
[10] https://www.vagrantup.com/

Non riporterò qui le istruzioni passo passo su come installare tutti questi software: d'altronde, nel nostro settore le cose cambiano di settimana in settimana. Usa invece i link che ti ho lasciato per rintracciare le ultime versioni di questi software seguendo le istruzioni presenti direttamente sui rispettivi siti.

Facciamo comunque un piccolo excursus.

Innanzitutto avremo bisogno di '**PHP**'. È il linguaggio usato da Laravel. Scarica ed installa l'ultima versione disponibile.

Per il version control, useremo '**Git**'. Diciamo che è anche un ottimo modo di scaricare velocemente Laravel. Dopo vedrai come.

Useremo quindi '**Composer**' per gestire le librerie da usare nei nostri progetti. È uno dei miei software preferiti, ed un must-have per qualsiasi sviluppatore PHP nel 2016!

Abbiamo poi '**Virtualbox**'. Forse lo conosci già. Si tratta di un software che consente la virtualizzazione di una macchina. Un po' come quando emulavi la Playstation sul tuo PC. Stavolta lo farai con un altro computer.

Infine, abbiamo '**Vagrant**'. Vagrant serve per il provisioning di ambienti virtuali. È un semplice tool da linea di comando, che useremo dal nostro terminale. Sarà Vagrant ad usare Virtualbox per creare le macchine che adopereremo.

Ci siamo? Bene!

Creare il Primo Progetto

Prima di costruire il nostro ambiente di lavoro, assicuriamoci di aver creato un progetto. Scegli una cartella: tipicamente io ho la cartella `Project` in cui inserisco i miei progetti. Entriamoci:

Esempio 01: Entriamo nella cartella dei Progetti

```
1 cd ~/Projects
```

Non è stato difficile, vero? Adesso, usiamo **Composer** per creare un nuovo progetto. Come? Semplicemente come segue ("example" è il nome scelto per la cartella del progetto).

Esempio 02: Creazione di un Progetto Laravel.

```
1 composer create-project laravel/laravel example
```

Probabilmente vedrai un bel po' di output sul terminal. Composer sta scaricando tutte le librerie necessarie al funzionamento di Laravel. Vengono dette 'package dependencies'. Una volta finita la procedura… è fatta! Hai installato Laravel. Adesso installiamo Homestead.

Installare Homestead

Entriamo nella cartella appena creata.

Esempio 03: Entriamo nella directory del progetto.

```
cd ~/Projects/example
```

Adesso installeremo il package di Homestead, che ci consentirà di creare un ambiente di lavoro virutale. Non ti preoccupare, nulla di complesso anche qui:

Esempio 04: Aggiunta di Homestead.

```
composer require laravel/homestead --dev
```

Assisterai all'installazione di altri package composer. Esattamente quello di cui abbiamo bisogno.

A quel punto, eseguiamo il comando "make" per configurare il nostro progetto.

Esempio 05: Lancio dell'installazione.

```
php vendor/bin/homestead make
```

Fatto!

Bene, adesso usa il comando che vedi qui di seguito per avviare la macchina virtuale. Probabilmente stai avviando questo comando per la prima volta, quindi sappi che ci metterà un po' di tempo a finire. Non temere. Deve essere scaricata, infatti, l'immagine della macchina virtuale che userai! Hai tempo per un caffè.

Esempio 06: Boot della macchina virtuale.

```
vagrant up
```

Una volta creata e sistemata la macchina virtuale, rimane soltanto una sola cosa da fare. Dobbiamo aggiungere, infatti, l'hostname della nostra applicazione nel file di hosts, in modo tale da accedere all'applicazione agevolmente, senza doverti ricordare l'indirizzo IP.

Con il tuo editor, modifica il file `/etc/hosts`, come root, aggiungendo questa linea.

Esempio 07: Aggiungere un hostname nel file di hosts.

```
192.168.10.10  homestead.app
```

Per controllare che tutto funzioni, visita `http://homestead.app` nel tuo browser. Dovresti poter vedere il testo "Laravel 5".

Missione completa!

Approfondiamo Vagrant

Il tuo ambiente di lavoro, adesso, esiste su una macchina virtuale. Questa macchina usa ovviamente le risorse della tua macchina fisica. Ad esempio, puoi decidere di rendere disponibile un certo quantitativo di RAM, o di potenza del processore.

Di volta in volta, quindi, sarà una buona idea fermare la macchina virtuale per evitare di sprecare risorse. Inoltre, c'è da tenere a mente che ogni volta che riavviamo la nostra macchina fisica, l'ambiente di sviluppo non viene avviato automaticamente.

Per questa ragione, vediamo un po' di comandi di base di Vagrant. Innanzitutto `up`, che hai visto poco fa.

Esempio 08: Avviare la macchina virtuale.

```
1 vagrant up
```

C'è poi `halt` che serve a fermare la macchina nel momento in cui non ne abbiamo più bisogno.

Esempio 09: Fermare la macchina virtuale.

```
1 vagrant halt
```

Se dovessi decidere di cambiare qualcosa nella configurazione della macchina, aggiornando il `Vagrantfile`, usa il comando `provision` per applicare le modifiche.

Esempio 10: Aggiornare le impostazioni della macchina virtuale.

```
1 vagrant provision
```

In alcune circostanze, potresti aver bisogno di eseguire alcuni comandi dalla macchina virtuale, e non da fuori. Usa il comando SSH per accedere (sudo).

Example 11: Accesso SSH alla macchina virtuale.

```
1 vagrant ssh
```

Infine, una volta finito il nostro progetto, potremo distruggere la macchina ormai inutile. Destroy!

Example 12: Elimina la VM.

```
1 vagrant destroy
```

Et voilà! Nel prossimo capitolo vedremo come avviare un'applicazione Laravel… ma in modo alternativo!

Ciclo di Vita

Se non hai mai usato un framework PHP in precedenza, probabilmente sei abituato ad avere un certo numero di file PHP nella tua directory e stop. Chi userà la tua applicazione dovrà richiedere ogni script, individualmente.

Laravel usa una combinazione di due elementi: **un front controller ed un router**. Che significa? Innanzitutto, c'è un singolo file PHP nella cartella principale che si prende carico di tutte le richieste che arrivano all'applicazione. Tutto passerà da questo front controller. In Laravel, questo file si chiama `index.php` e può essere trovato nella cartella `public`. Che, tra l'altro, è l'unica cartella che va esposta pubblicamente.

> Ma sei pazzo? Non posso mica fare applicazioni con una sola pagina!

Non temere: non è come sembra. Laravel fa uso di svariate tecniche per servire vari tipi di contenuto in base alla richiesta effettuata. Ecco il flusso preciso:

```
1  Richiesta > Servizi > Routing > Logica > Risposta
```

Forse non ancora conosci il significato preciso dei vari termini, ma una cosa è certa: una richiesta ad un webserver non è altro che un qualcosa che parte da un input per arrivare ad un output. Stop.

Vediamo nel dettaglio, ora, i singoli elementi di questo flusso.

Richiesta

Ogni richiesta effettuata da un browser alla tua applicazione ha un sacco di informazioni al suo interno. Ad esempio, l'URL, il metodo HTTP usato, i dati della richiesta, header e così via.

Tocca a Laravel (ed alla tua applicazione) capire cosa fare in base alla richiesta che arriva. Usando Laravel, le informazioni riguardanti la richiesta attuale sono contenute in un'istanza della classe `Illuminate\Http\Request`, che estende `Symfony\Component\HttpFoundation\Request`, che viene creata ad ogni richista.

Hai letto bene, Symfony: ha un'ottima implementazione del protocollo HTTP nel suo package `HttpFoundation`. Laravel fa uso di questo package per non reinventare la ruota.

Ad ogni modo, in questa richiesta avremo un sacco di informazioni a disposizione. Cosa succede ora?

Servizi

Lo step successivo è il processo di bootstrapping del framework. In poche parole, Laravel ha svariati servizi e feature che ad ogni richiesta vengono usati per rendere la vita dello sviluppatore più semplice. Prima di fare uso di questi servizi, però, dobbiamo inizializzarli.

In questa fase, il framework caricherà tutti i servizi e le varie configurazioni, assicurandosi che tutto sia pronto per il nostro codice. Vedremo più avanti, nel dettaglio, come questi servizi vengono caricati.

Il framework è pronto ad accogliere il nostro codice: che succede ora?

Routing

Come scoperto poco fa, c'è solo uno script davvero accessibile quando usiamo Laravel. Come possiamo differenziare le varie pagine ed implementare vari comportamenti, in base a questa o quella condizione?

Ecco che entra in scena il router!

Il solo scopo del router è quello di verificare che una certa richiesta corrisponda ad un certo codice da eseguire. Per capirci, sarà il router a capire che, ad esempio, quando faremo una richiesta GET dell'URI `user/profile` dovremo mostrare la pagina del profilo del nostro utente.

Laravel ha un modo molto semplice di definire le route (o rotte) di un'applicazione: ne parleremo molto presto. Per ora, facciamo un altro piccolo salto in avanti.

Logica

Nel nostro flusso, questo è il momento in cui entra in gioco la logica. Precisamente, possiamo dire che entra in gioco la *tua* logica. Lavoreremo con un database? Mostreremo un messaggio di errore? In questa parte viene deciso il comportamento della nostra applicazione.

In Laravel ci sono vari modi di definire le logiche della propria applicazione. Anche in questo caso ne parleremo a breve, dopo questa panoramica.

Non rimane che l'ultima tessera del domino…

Risposta

Alla fine di tutto, c'è la risposta. L'output, finalmente. Può essere un template HTML, o magari dei dati in formato JSON. O, semplicemente, una stringa. O magari nulla! A volte una schermata di errore, o una pagina 404.

Ad ogni modo, la risposta è quello che il tuo utente vedrà dopo la sua richiesta. Fine!

Facciamo un piccolo riassunto prima di procedere.

- Il browser manda una richiesta;
- Laravel avvia i propri servizi;
- Viene interrogata la richiesta, ed effettuato il routing di questa;
- La logica della specifica route viene eseguita;
- La risposta derivante da questa logica viene inviata all'utente;

Sembra una cosa semplice, tuttavia non darla mai per scontata: sapere come funzionano le cose ti può rendere un web developer migliore.

Ok, gira pagina!

Configurazione

La configurazione è un aspetto molto importante di qualsiasi applicazione. Infatti, non avremo mai applicazioni con le stesse identiche impostazioni in tutti i possibili ambienti. Soprattutto nel caso in cui dovessimo decidere di condividere con altri la nostra applicazione, avremo il bisogno di poter dare un'ampia possibilità di scelta.

In questo capitolo vedremo insieme come Laravel ci consente di specificare le "opzioni" per la nostra applicazione.

File di Configurazione

Inizieremo dal dare un'occhiata alla cartella `config` presente nella root del nostro progetto. Eccone il contenuto.

Esempio 01: File di configurazione.

```
app.php
auth.php
broadcasting.php
cache.php
compile.php
database.php
filesystems.php
mail.php
queue.php
services.php
session.php
view.php
```

Troverai svariati file PHP. Molti di loro hanno il nome del componente corrispondente nel framework. Sono i file di configurazione di default che vengono usati da un'installazione "fresca" di Laravel.

Adesso, prendi uno di questi file, uno qualsiasi, e guardane il contenuto. Esatto: non sono altro che semplicissimi array associativi. Ogni chiave rappresenta il nome dell'opzione, mentre il valore è appunto… il valore! Per accedere a questi valori, quindi, possiamo usare la Facade `Config` oppure l'helper `config()` come segue:

Esempio 02: Accedere ai valori di configurazione.

```
// Usando una Facade.
$debug = Config::get('app.debug', false);

// Usando un Helper.
$debug = config('app.debug', false);
```

Come puoi vedere, la sintassi è praticamente la stessa, identica! Il primo parametro è la chiave di cui abbiamo bisogno. Il formato di questo primo parametro è, come puoi immaginare, `nome_file.chiave`. In questo caso stiamo richiedendo il valore di `debug` nel file `app`. Il secondo parametro, invece, consiste in un valore di default da ritornare come "cuscinetto" nel caso in cui la chiave specificata non esista.

Nel caso in cui tu voglia usare la Facade, non scordarti di specificare `use Config;` all'inizio della tua classe.

Più in avanti, vedremo un terzo metodo da usare, più pulito, per recuperare i valori delle opzioni senza usare Facade o helper.

Chiaramente, puoi anche creare nuovi file di configurazione, semplicemente inserendoli nella cartella `config`. Laravel infatti provvederà a caricarli automaticamente e ti permetterà di usarli allo stesso modo in cui usi quelli già esistenti.

Un consiglio: aggiungi alle tue applicazioni quante più opzioni puoi (senza esagerare, ovviamente). Renderai tutto più flessibile e più semplice da mantenere.

Variabili d'Ambiente

Girovagando tra i file di configurazione ti sarai sicuramente imbattuto in una piccola funzione chiamata `env()`. Tale funzione è il cuore di un nuovo modo di creare diverse configurazioni in altrettanti diversi ambienti di lavoro. Un esempio: in fase di sviluppo non userai di certo le stesse credenziali che usi in produzione!

Nelle versioni precedenti di Laravel venivano usate delle cartelle per ogni ambiente. Poi le cose sono cambiate, ed in meglio, visto che il progetto ha iniziato a fare uso della libreria `DotEnv`, che permette di caricare i valori di configurazione dalle variabili d'ambiente. Certo, caricare tutte le variabili d'ambiente da una shell può essere noioso:

Esempio 03: Impostare una variabile d'ambiente via shell.

```
export DB_HOST="127.0.0.1"
```

Fidati, tra l'altro, se ti dico che su Windows è ancora più brutto. Tuttavia, c'è una soluzione a tutto questo: usare dei file in cui memorizzare tutte le variabili da usare. I file `.env`, da mettere nella cartella principale del progetto.

Esempio 04: Il nostro file .env.

```
APP_ENV=local
APP_DEBUG=true
APP_KEY=base64:a0hRMNEdGdWW6EUVCn3vIu8VFQiJc113CMciFkJ+pcw=
APP_URL=http://localhost

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret

CACHE_DRIVER=file
SESSION_DRIVER=file
QUEUE_DRIVER=sync

REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379

MAIL_DRIVER=smtp
MAIL_HOST=mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null
```

Questo file `.env` è davvero molto semplice. Le chiavi sono sulla sinistra, i valori sulla destra. Quelli che vedi qui sono una serie di valori di default, usati dal framework non appena viene installato. Sono le basi del lavoro con il database, sistema di invio di email, e server di cache. Se vogliamo fare in modo che alcune opzioni del nostro software cambino in base all'ambiente di lavoro, questo è il posto migliore dove specificarle.

La funzione `env()` può essere quindi usata nei nostri file di configurazione per recuperare i valori presenti nel file `.env`. Certo, stiamo aggiungendo un altro layer al processo di configurazione, ma avremo la possibilità di usare un file `.env` diverso per ogni installazione senza modificare la codebase! Flessibilità al massimo.

Esempio 05: La funzione env().

```
1 $host = env('DB_HOST', '127.0.0.1')
```

Quello che stiamo facendo, qui, è richiedere l'host del database, specificando però un valore di default nel caso in cui tale opzione non sia disponibile nel file `.env` o tra le variabili d'ambiente.

Importante: occorre tenere presente che i valori specificati nelle variabili d'ambiente avranno sempre la priorità su quelli specificati nel file `.env`.

Normalmente, sarà buona regola non mettere mai i file `.env` sotto version control. Ogni installazione infatti avrà un file `.env` con contenuti diversi! Una buona idea però può essere quella di creare un file `.env.dist` con dei valori di default, o di esempio, per creare un boilerplate da cui partire per la costruzione del proprio `.env`. Laravel, ad esempio, non ha un file `.env` di default, ma un file `.env.example` che copia durante l'installazione.

> Mmh, ok, mi piace, ma perchè non usare `env()` ovunque al posto di `config()`?

Ottima domanda! Certo, potresti tranquillamente usare `env()` ovunque, ma così non godresti di nessun vantaggio derivante dal sistema di caching del layer di configurazione di Laravel. Tra l'altro, rifletti: ci sono delle impostazioni che devono rimanere tali ma non cambiano di ambiente in ambiente, mentre alcune si!

> Capisco. Si, mi sembra una spiegazione logica.

Bene. Ricorda comunque: una buona pratica è quella di chiamare `env()` *solo* nei file presenti nella directory `config`.

Se un package che hai installato ha dei file di configurazione propri, ricorda di usare `php artisan vendor:publish` per copiarli nella cartella `config` del tuo progetto. Non devi farlo sempre: ti verrà specifiato nelle istruzioni del package.

Cache e Configurazione

Come detto poco fa, Laravel può mettere in cache i valori dei file di configurazione, in modo tale da caricarli più velocemente nelle richieste successive. Puoi ben immaginare che questa feature trova la sua applicazione ideale in ambiente di produzione.

Per mettere in cache i file di configurazione non devi fare altro che andare nella cartella principale della tua applicazione ed eseguire il comando artisan

```
1 php artisan config:cache
```

I file di configurazione verranno ora messi in cache, venendo bypassati guadagnando in velocità di esecuzione. Nel caso in cui tu voglia pulire questa cache non devi fare altro che eseguire

```
1 php artisan config:clear
```

e via! Probabilmente non avrai bisogno di questa feature in fase di sviluppo locale, ma in produzione potrebbe far comodo, vero?

Per quanto riguarda la configurazione direi che ci siamo, è tutto. Passiamo adesso al routing!

Routing di Base

Bene, iniziamo a muovere i primi passi con Laravel. La prima cosa da fare è capire come viene effettuata e gestita una richiesta all'applicazione.

Esempio 01: Un URL

```
http://homestead.app/my/page
```

In questo esempio stiamo usando il protocollo HTTP (usato da molti browser) per accedere alla nostra applicazione Laravel, su `homestead.app`. La porzione `my/page` è quella che useremo per veicolare la giusta richiesta alla logica corrispondente.

Prima di procedere, ricordiamoci che le route si trovano in `app/Http/routes.php`. Trovato il file? Bene. Creiamo la nostra prima route.

Definire una Route

Esempio 02: La nostra prima route.

```php
<?php

// app/Http/routes.php

Route::get('my/page', function () {
    return 'Hello world!';
});
```

Salva il contenuto del file `routes.php` e vai pure sul tuo browser, ovviamente rimpiazzando `homestead.app` con l'indirizzo della tua applicazione.

Se tutto è stato configurato correttamente dovresti poter vedere le parole 'Hello world!', in un fantasmagorico Times New Roman! Meraviglia, vero?

Adesso vediamo come funziona la dichiarazione di una route.

Innanzitutto, una route viene dichiarata usando la classe `Route`. Di questa classe, chiamiamo il metodo `get`, che viene usato per "catturare" le richieste, appunto, GET. Saprai infatti che tutte le richieste HTTP hanno un verb specifico che indica l'intento di quell'azione. Abbiamo GET, POST, PUT e così via...

Ecco tutti i metodi che Laravel mette a disposizione per catturare le richieste:

Esempio 03: Metodi di Route.

```php
<?php

// app/Http/routes.php

Route::get();
Route::post();
Route::put();
Route::patch();
Route::delete();
Route::any();
```

Tutti questi metodi accettano gli stessi identici parametri, per cui sentiti libero di usare il metodo che meglio credi di situazione in situazione. A breve parleremo anche di quello che viene definito `RESTful routing`, mentre per ora tutto quello di cui hai bisogno è il metodo GET per richiedere dei "contenuti", e POST per mandare dei dati nel corpo della richiesta.

Se te lo stai chiedendo si, è una semplificazione, serve solo per rendere un po' le idee.

Il metodo `Route::any()` è molto comodo se devi collegare un determinato comportamento ad una route qualsiasi sia il verb usato. Ad ogni modo, ti consiglio sempre di usare il verb giusto, specifico per quello che devi fare.

Ok, detto questo torniamo al nostro esempio.

Esempio 04: La nostra prima route, di nuovo.

```php
<?php

// app/Http/routes.php

Route::get('my/page', function () {
    return 'Hello world!';
});
```

Il metodo `get()` richiede due parametri. Il primo è l'URI a cui collegare una specifica logica. In questo caso, tale URI è `my/page`. Il secondo parametro invece è proprio la logica che vogliamo eseguire ad ogni chiamata a `my/page`. Qui stiamo usando una `Closure`, conosciuta anche come "funzione anonima". Non spaventarti: non sono altro che semplici funzioni senza nome che possono essere assegnate ad una variabile.

Per intenderci, avremmo potuto scrivere lo stesso snippet così:

Esempio 05: Separazione della Closure.

```php
<?php

// app/Http/routes.php

$logic = function () {
    return 'Hello world!';
};

Route::get('my/page', $logic);
```

Stavolta abbiamo memorizzato la nostra `Closure` in una variabile `$logic`, per poi passarla come parametro di `Route::get()`. Il risultato finale non cambia, prova tu stesso!

Puoi ovviamente definire tutte le route che vuoi:

Esempio 06: Route multiple.

```php
<?php

// app/Http/routes.php

Route::get('first/page', function () {
    return 'First!';
});

Route::get('second/page', function () {
    return 'Second!';
});

Route::get('third/page', function () {
    return 'Potato!';
});
```

Prova a raggiungere le seguenti URL, per verificare come la nostra applicazione, adesso, si comporta.

Esempio 07: URL di route multiple.

```
http://homestead.app/first/page
http://homestead.app/second/page
http://homestead.app/third/page
```

Funziona tutto? Perfetto!

Cosa? Vuoi capire come collegare una route all'indirizzo

Esempio 08: Un URL vuoto?

```
http://homestead.app
```

vero? Nessun problema: la soluzione è la seguente:

Esempio 09: Route senza path.

```php
<?php

// app/Http/routes.php

Route::get('/', function () {
    return 'In soviet Russia, function defines you.';
});
```

> Ehi ehi, aspetta! Ma non abbiamo uno slash in `http://homestead.app`!

Vero, ma se provi anche a farlo manualmente, se inserisci un forward slash come nell'esempio seguente:

Esempio 10: Con o senza slash...

```
http://homestead.app
http://homestead.app/
```

il risultato finale non cambia, è sempre la route "di base" che risponde all'appello!

Gli URL possono avere quanti segmenti vuoi (un segmento è quella parte tra uno slash e l'altro, per capirci). Puoi usare tale segmentazione per costruire la gerarchia del tuo sito:

Esempio 11: Un Sito Immaginario.

```
/
/books
    /fiction
    /science
    /romance
/magazines
    /celebrity
    /technology
```

Un sito piuttosto minimale, ma un buon esempio di struttura che spesso troverai sul web. Proviamo a ricrearla, usando delle route Laravel.

Per velocità e chiarezza, come puoi vedere, ho rimosso il contenuto delle singole Closure.

Esempio 12: Route del Sito Immaginario.

```php
<?php

// app/Http/routes.php

// home page
Route::get('/', function () {});


// routes for the books section
Route::get('/books', function () {});
Route::get('/books/fiction', function () {});
Route::get('/books/science', function () {});
Route::get('/books/romance', function () {});

// routes for the magazines section
Route::get('/magazines', function () {});
Route::get('/magazines/celebrity', function () {});
Route::get('/magazines/technology', function () {});
```

Fatto! Abbiamo replicato esattamente la gerarchia vsta prima. Certo, avrai notato un certo numero di ripetizioni nelle varie stringhe: vediamo di risolvere la cosa, aderendo al principio DRY (Don't Repeat Yourself).

Parametri delle Route

Le route possono anche prevedere dei parametri attraverso l'inserimento di "segnaposti" nelle definizioni. I parametri ti permetteranno di creare delle route più interattive e più complesse, visto che potrai passare dei valori variabili dall'esterno all'interno della tua app.

Cosa? Ti ho confuso? Tranquillo, vediamo subito un esempio.

Esempio 13: Parametri per le Route.

```php
<?php

// app/Http/routes.php

// route per la sezione books
Route::get('/books', function () {
    return 'Books index.';
});

Route::get('/books/{genre}', function ($genre) {
    return "Books in the {$genre} category.";
});
```

In questo esempio, abbiamo eliminato il bisogno di tutte le route che rappresentavano uno specifico genere di libro, passando invece ad usare un placeholder `{genre}`, subito dopo la "radice" `/books/`. Laravel riconoscerà automaticamente questo segnaposto e passerà alla Closure un parametro `$genre` da poter usare nella tua logica.

Ad esempio, provando a navigare su

Esempio 14: Parametro nell'URL.

```
http://homestead.app/books/crime
```

riceverai in output...

Esempio 15: Output.

```
Books in the crime category.
```

Volendo, puoi anche rendere un parametro opzionale usando il punto interrogativo `?` alla fine del nome del parametro.

Così:

Esempio 16: Parametro opzionale.

```php
<?php

// app/Http/routes.php

// route per la sezione books
Route::get('/books/{genre?}', function ($genre = null) {
    if ($genre == null) {
         return 'Books index.';
    }
    return "Books in the {$genre} category.";
});
```

Da questo momento, nel caso in cui l'utente non dovesse definire un valore per il parametro `{genre}`, automaticamente si ritroverà nella pagina principale dei libri. In caso contrario, invece, si ritroverà nella pagina della singola categoria.

In un altro caso invece potremmo avere la necessità di un valore di default per un parametro opzionale. Anche qui nessun problema, basta lavorare sulla definizione del parametro che arriva alla Closure.

Esempio 17: Valore di default per il parametro.

```php
<?php

// app/Http/routes.php

// routes for the books section
Route::get('/books/{genre?}', function ($genre = 'Crime') {
    return "Books in the {$genre} category.";
});
```

Adesso, visitando l'URL

Esempio 18: URL senza parametro.

```
http://homestead.app/books
```

riceveremo questo output:

Esempio 19: Output.

```
Books in the Crime category.
```

Visto? Tutto sommato è qualcosa di davvero semplice. Se è la prima volta che ti approcci a Laravel, sentiti sicuro prima di proseguire. Fai tante prove e prendi dimestichezza con il meccanismo.

Dopodichè, prosegui: per ogni richiesta c'è una risposta, ed è esattamente quello che stiamo andando a conoscere. Le risposte.