

Laravel

INTRODUZIONE A LARAVEL

COS'È LARAVEL

Laravel è un framework PHP open source creato nel 2011 da Taylor Otwell.

Si basa sul pattern architettonico **MVC (Model – View – Controller)** e fornisce strumenti per la gestione delle richieste HTTP, l'elaborazione dati, l'accesso al database, la sicurezza e tanto altro.

LARAVEL COME FULL STACK FRAMEWORK

Laravel può essere usato come framework **full stack**: indirizza le richieste all'applicazione e rende il frontend via **Blade templates** o **single-page application** di tecnologia ibrida come **Inertia**.

LARAVEL COME API BACKEND

Laravel può anche servire come **API** ad una **single page application** JavaScript o ad un'app mobile. Ad esempio, si può usare Laravel come **API backend** per un'applicazione Angular o Next.js.

Il pattern MVC

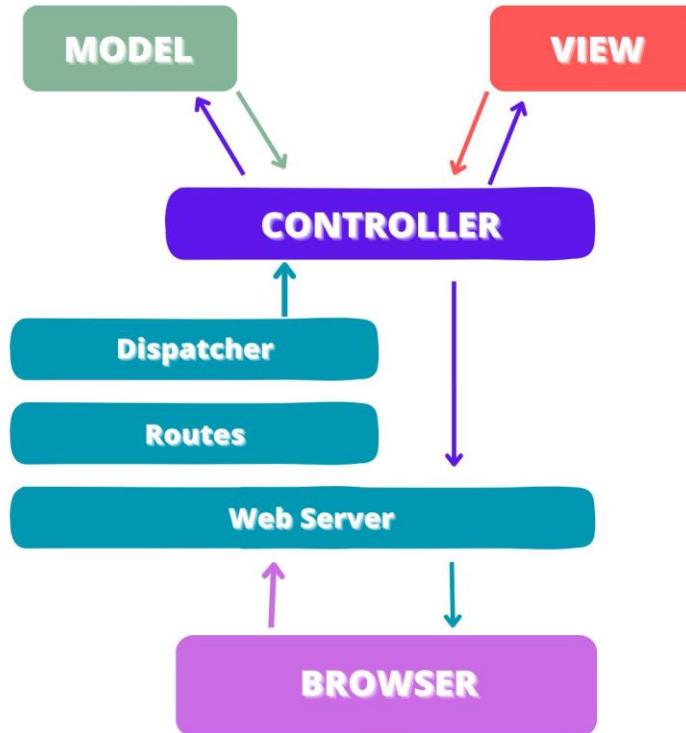
Pattern architettonico molto diffuso per lo sviluppo di applicazioni web.

In estrema sintesi: il **Model** (dati e logica dell'app) viene mostrato tramite la **View** (visualizzazione dei dati ed interfaccia utente) all'utente, il quale genera gli input con cui il **Controller** aggiorna il **Model**.

Questa separazione di funzionalità rende il codice **modulare** e più facilmente **manutenibile**.

Consente inoltre a persone differenti di lavorare contemporaneamente su parti diverse del codice.

ES: un designer può lavorare sulle view mentre il programmatore lavora sul controller.



L'utente, mediante il browser, interagisce con l'applicazione.

Il browser invia al Web server l'azione svolta dall'utente.

L'applicativo deve quindi intercettare la richiesta dell'utente (**Routes**) e valutarla (**Dispatcher**) in modo da richiamare il controller giusto con i giusti parametri.

1- CONTROLLER (gestione degli input)

Sono responsabili della **gestione degli input** dell'utente e della generazione della risposta appropriata. Sono gli INTERMEDIARI tra View e Model

Solitamente il controller riceve la richiesta dell'utente, la **convalida** e la passa ad un modello per l'elaborazione.

Una volta che i dati sono stati elaborati, il controller genera l'output e lo passa alla view.

Cartella Laravel: *app/Http/Controllers*

2- MODELS (dati)

Sono i responsabili del **recupero, creazione, aggiornamento e cancellazione dei dati**. Incapsula tutta la logica necessaria per la **manipolazione dei dati**. Ricevono i dati dal controller con il quale dialogano per il passaggio di dati in ingresso e uscita.

Oltre alle operazioni CRUD di base (comunicherà dunque con il database), un model potrebbe contenere altre logiche come operazioni di casting e relazioni con altri modelli.

Per convenzione, in Laravel i Model sono dichiarati con il nome al **singolare** mentre la relativa **tabella** viene dichiarata usando il plurare.

Cartella Laravel : *app/Models*

3- VIEW (vista)

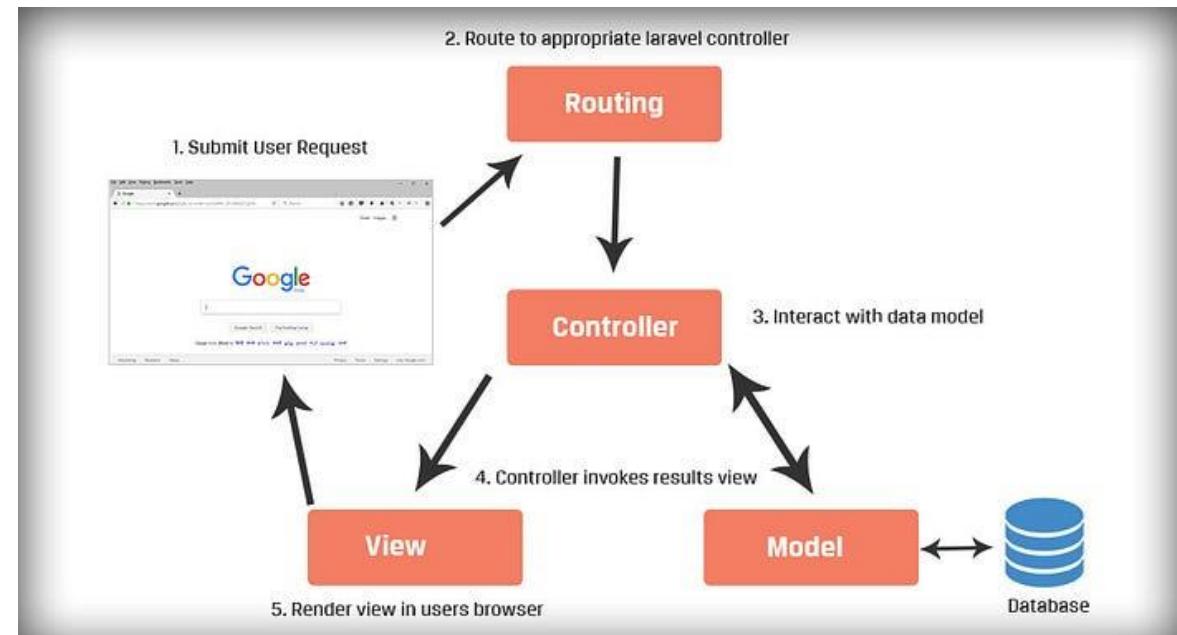
Componente per servire l'**interfaccia all'utente**.

Contengono solitamente HTML, gli assets JavaScript e CSS e dati dinamici passati dal controller.

Laravel utilizza un proprio **template engine**, **BLADE**, con una propria sintassi.

Le viste vengono compilate in PHP e poi memorizzate nella cache fino a quando non vengono modificate.

resources/views.



INSTALLAZIONE DI LARAVEL

PRIMA DI INIZIARE

Bisogna prima di tutto assicurarsi che sulla propria macchina siano installati PHP e Composer.

Se si utilizza macOS, PHP e [Composer](#) possono essere installati via [Homebrew](#).

Si raccomanda inoltre l'installazione di [Node](#) e [NPM](#).

COMPOSER

Composer è un tool per la **gestione delle dipendenze (dependency manager)** per PHP.

Supponiamo di avere un progetto che dipende da **un certo numero di librerie ed alcune di queste librerie dipendono da altre.**

Composer consente di dichiarare le librerie da cui l'app dipende, trova quali versioni di quali pacchetti possono e hanno bisogno di essere installate e le installa (le scarica all'interno del progetto).

Composer non si occupa solo di definire e recuperare le dipendenze di un progetto, ma **anche dell'autoloading delle classi PHP.**

PRINCIPALI COMANDI DI COMPOSER

a. Il file **composer.json**

Un progetto che utilizza Composer è caratterizzato dalla presenza del file **composer.json** nella directory principale. Questo file contiene **tutte le dipendenze per il progetto**. In un progetto Laravel viene generato contestualmente alla creazione dell'app.

```
{  
    "require": {  
        "monolog/monolog": "^3.0.0"  
    }  
}
```

ESEMPIO
MINIMALE DI
COMPOSER.JSON
*Dice che l'unica
dipendenza del
progetto è
Monolog.*

Ogni dipendenza è composta dal **nome del pacchetto (package name)**. Composto a sua volta dal nome del **vendor** e nome del pacchetto separati da /) e da un **selettore di versione (version constraint*)**.

Con il comando

`composer install`

oppure

`php composer.phar update`

i pacchetti specificati all'interno del file verranno **scaricati e installati**, la loro esatta versione verrà trascritta nel file **composer.lock** (file che serve per «bloccare» l'esatta versione delle dipendenze) e le dipendenze saranno inserite nella cartella vendor del progetto.

b. Installare un pacchetto (require)

Il repository principale dal quale composer scarica i pacchetti è **Packagist**. Attraverso il comando **require** la libreria viene cercata automaticamente tra quelle presenti in questo repository.

`composer require mypackage`

c. Cancellare un pacchetto (remove)

`Composer remove mypackage`

Il pacchetto verrà eliminato dalle dipendenze del progetto e dalla cartella vendor

d. Aggiornare uno o più pacchetti (update)

Per aggiornare un pacchetto alla **versione più recente compatibile** con il selettore di versione indicato si usa il comando **update** seguito dal nome del pacchetto.

`Composer update mypackage`

Per **aggiornare tutti i pacchetti** del progetto basta eseguire il comando **update** senza specificare un pacchetto.

`Composer update`

NOTE:

*^3.0.0 vuol dire in pratica “ogni versione maggiore di 3.0.0, ma non la 4.0.0 e successive”

e. Mostrare informazioni su un pacchetto (show)

Per ricevere informazioni su un pacchetto si usa il comando

composer **show** vendor/package

Per informazioni su ulteriori comandi di Composer usare
Composer **list** oppure composer **help**

NPM (Node Package Manager)

Gestore di pacchetti per progetti Node.js che permette l'installazione di librerie, framework e altri strumenti di sviluppo.

L'interfaccia a riga di comando (CLI) consente di installare e disinstallare pacchetti e di accedere ad altre funzionalità.

a. PRINCIPALI COMANDI :

- 1) Npm install
Necessario all'installazione di un pacchetto
- 2) Npm uninstall
disinstalla Pacchetto
- 3) Npm init
usato per inizializzare un progetto. Quando si esegue questo comando, il file package.json viene creato
- 4) Npm update
Aggiorna un pacchetto all'ultima versione

PRIMO PROGETTO IN LARAVEL

E' ora possibile creare un nuovo progetto Laravel via Composer grazie al comando `create-project`

```
composer create-project  
laravel/laravel:^10.0 lezione1
```

Oppure è possibile creare un nuovo progetto installando globalmente Laravel via Composer

```
composer global require  
laravel/installer
```

```
laravel new example-app
```

Dopo la creazione del progetto, avviare il server di sviluppo locale utilizzando il comando `serve` della CLI (Command Line Interface) Artisan di Laravel

```
cd example-app
```

```
php artisan serve
```

Una volta lanciato il server, l'applicazione sarà accessibile nel browser all'indirizzo

```
http://localhost:8000
```

Configurazione iniziale

Tutti i file di configurazione di Laravel si trovano nella cartella **config**.

Questi file di configurazione permettono di impostare, ad esempio, le informazioni per la connessione al database, le info relative al server mail oppure l'encryption key.

Tuttavia spesso Laravel non necessita di configurazioni aggiuntive. Può essere utile rivedere il file **config/app.php** e la sua documentazione. Esso contiene diverse opzioni come la **timezone and locale**.

CONFIGURAZIONE BASATA SULL'AMBIENTE

Molte opzioni di configurazione di Laravel possono cambiare a seconda che l'app stia girando su una macchina locale o su un web server di produzione. Per questo motivo molti importanti valori di configurazione sono definiti usando file **.env** che esistono alla radice dell'app.

DATABASE E MIGRATIONS

Di default l'applicazione interagirà con un **database MySql** al quale accederà da **127.0.0.1**

Qualora non si voglia installare un database MySql sulla macchina locale, si può utilizzare un database **SQLite**. Per creare un database SQLite si crea **un file SQLite vuoto**. Generalmente il file si trova all'interno della cartella **database** dell'applicazione.

```
touch database/database.sqlite
```

Bisognerà **poi aggiornare il file di configurazione .env** per usare il database driver di Laravel sqlite. Si possono rimuovere tutte le altre opzioni di configurazione del database.

```
DB_CONNECTION=sqlite  
DB_CONNECTION=mysql  
DB_HOST=127.0.0.1  
DB_PORT=3306  
DB_DATABASE=laravel  
DB_USERNAME=root  
DB_PASSWORD=
```

Una volta configurato il database, si **lancia la migrazione** del database dell'applicazione. Questa **creerà le tabelle del database**

```
php artisan migrate
```

STRUTTURA DELLE CARTELLE – Root directory

ROOT DIRECTORY

- The app Directory
- The bootstrap Directory
- The config Directory
- The database Directory
- The public Directory
- The resources Directory
- The routes Directory
- The storage Directory
- The tests Directory
- The vendor Directory

- **LA CARTELLA APP**

Contiene il cuore dell'applicazione. Quasi tutte le classi di un'applicazione si trovano in questa cartella

- **CARTELLA BOOTSTRAP**

contiene il file **app.php** che avvia il framework. Al suo interno si ha la **cartella cache** che contiene i files generati dal framework per l'**ottimizzazione delle performances** come i file di cache delle routes e dei services.

- **CARTELLA CONFIG**

Contiene tutti i files di configurazione dell'applicazione.

- **CARTELLA DATABASE**

contiene le **migrazioni del database**, i **model factories** ed i **seeds**. Può essere usata per ospitare un database SQLite.

- **CARTELLA PUBLIC**

Contiene il file **index.php** che è l'**entry point** di tutte le richieste effettuate all'applicazione. Contiene anche gli assets come immagini, JS e CSS

- **CARTELLA RESOURCES**

contiene le **views** e gli **assets** grezzi e non compilati come CSS e JS

- **CARTELLA ROUTES**

contiene la definizione di tutte le rotte dell'applicazione.
Di default sono inclusi i file **web.php**, **api.php**,
console.php e **channels.php**.

- File **web.php**: contiene le rotte che il RouteServiceProvider posiziona nel gruppo **middleware web** il quale fornisce lo stato di sessione, la CSRF protection e la cookie encryption. Se l'applicazione non prevede una Restful API stateless, allora tutte le rotte saranno definite in questo file.
- **(Stateless)**: In un'applicazione stateless, ogni richiesta del client al server contiene tutte le informazioni necessarie per comprendere e soddisfare quella richiesta
- File **api.php**: contiene le **rotte del middleware api**. Queste rotte devono essere **stateless** quindi le richieste che accedono all'app attraverso queste rotte devono essere **autenticate via tokens** e non avranno accesso alla stato della sessione

- File **console.php**: file nel quale definire le closure basate su comandi console. Definisce gli entry point da console

- Cartella **channels.php**: cartella nella quale registrare tutti i canali di broadcasting degli eventi supportati dall'applicazione

- **CARTELLA STORAGE**

contiene i logs, i templates Blade compilati, i file basati sulle sessioni e altri file generati dal framework.
Storage/app/public può essere usata per immagazzinare file generati dall'utente e che dovrebbero essere pubblicamente accessibili.

- **CARTELLA TEST**: contiene i test automatizzati. Ogni classe dovrebbe avere il suffisso Test.

Si possono lanciare i test con i comandi `phpunit`, `php vendor/bin/phpunit` oppure `php artisan test`

- **CARTELLA VENDOR**: contiene le dipendenze di Composer

STRUTTURA DELLE CARTELLE – App directory

APP DIRECTORY

- **Console** Directory
- **Events** Directory
- **Exceptions** Directory
- **Http** Directory
- **Jobs** Directory
- **Listeners** Directory
- **Mail** Directory
- **Models** Directory
- **Notifications** Directory
- **Policies** Directory
- **Providers** Directory
- **Rules** Directory

La maggior parte del codice dell'app si trova all'interno della directory app.

Contiene numerose sottodirectory e molte altre vengono create quando si utilizza il comando Artisan **make** per la generazione di classi.

Per esempio, app/Jobs non esiste finchè non viene eseguito il comando **artisan make:job** con il quale Artisan crea una classe job.

Per una **lista dei comandi Artisan** con i quali creare directories all'interno della cartella App, digitare nel terminale il comando

```
php artisan list make
```

- **CARTELLA CONSOLE**

contiene tutti i comandi Artisan creati dall'utente con il comando

make:command

- **CARTELLA EVENTI**

Non esiste di default. Creata con il primo evento.

Comandi:

event:generate oppure **make:event**

Ospita le classi relative agli eventi. Questi possono essere utilizzati per avvisare un'altra parte dell'applicazione che una data azione si è verificata.

- **CARTELLA ECCEZIONI**

contiene il gestore delle eccezioni (exceptions handlers).

Per personalizzare il modo in cui un'eccezione viene registrata o visualizzata, bisogna modificare la classe Handler

- **CARTELLA HTTP** ★

Cartella fondamentale che **contiene i controllers, i middleware e le richieste dei form**. Quasi tutta la logica di gestione delle requests che arrivano all'app si trova all'interno di questa cartella.

- **CARTELLA JOBS**

Non esiste di default.

Comando per creazione di Jobs è
make:job

- **CARTELLA LISTENERS**

No di default ma generata con

event:generate oppure **make:listener**

Contiene le classi che gestiscono gli eventi: gli **Event listeners** ricevono l'istanza di un evento e in risposta all'evento elaborano delle azioni o logiche. Ad esempio: un evento UserRegistered può essere gestito da un listener SendWelcomeEmail

- **CARTELLA MAIL**

Non esiste di default: Comando Artisan
make:email.

Contiene le classi che rappresentano **le mail spedite dall'applicazione**.

Gli oggetti mail permettono di encapsulare tutta la logica di generazione di una mail in una singola classe. La mail può essere spedita usando il metodo **Mail::send**

- **CARTELLA MODELS** ★

Contiene tutte le classi del **model Eloquent**. L'ORM Eloquent fornisce una semplice implementazione per lavorare con il database: ogni tabella del database ha un **Model** che è usato per interagire con quella stessa tabella. I Models **permettono di eseguire query per i dati nelle tabelle**

- **CARTELLA NOTIFICATIONS**

No di default.

make: notification

Contiene tutte le notifiche «transazionali» che sono inviate dall'applicazione come notifiche circa eventi.

- **CARTELLA POLICIES**

no di default
make: policy

Contiene le **classi di autorizzazione alla policy**. Le policies sono utilizzate per determinare se un utente può eseguire una determinata azione.

- **CARTELLA PROVIDERS**

contiene tutti i service providers per l'applicazione. Di default contiene già diversi providers.

- **CARTELLA RULES**

no di default.

make:rule

Contiene gli oggetti delle regole di validazione customizzate dall'utente. Le regole sono utilizzate per encapsulare logiche di validazione complicate in un semplice oggetto.

FRONTEND

Esistono principalmente due modi per affrontare lo sviluppo del frontend quando si realizza un'applicazione in Laravel:

- Costruire il frontend sfruttando **PHP**
- Usare **framework Javascript** come Vue e React / Angular

Costruire il frontend con PHP e Blade

In passato molte applicazioni Php restituivano HTML al browser usando modelli HTML disseminati di echo che riportavano i dati recuperati dal database durante la request:

In Laravel questo approccio può essere ancora usato usando le views e Blade.

```
<div>
    <?php foreach ($users as $user): ?>
        Hello, <?php echo $user->name; ?> <br />
    <?php endforeach; ?>
</div>
```

BLADE

Blade è un linguaggio di templating estremamente leggero che fornisce sintassi corte e comode per mostrare dati, per iterare sui dati e tanto altro

Quando si costruiscono applicazioni in questo modo le interazioni con la pagina ricevono in genere un documento HTML completamente nuovo dal server e l'intera pagina viene riprodotta dal browser.

```
<div>
    @foreach ($users as $user)
        Hello, {{ $user->name }} <br />
    @endforeach
</div>
```

Ancora oggi, molte applicazioni possono essere perfettamente adatte a essere costruite in questo modo utilizzando semplici modelli Blade.

Tuttavia, con le crescenti aspettative nei confronti delle applicazioni web, molti sviluppatori hanno sentito l'esigenza di costruire frontend più dinamici.

Alla luce di questo alcuni sviluppatori hanno deciso di iniziare a costruire il frontend delle loro applicazioni usando framework JS come Vue e React.

Altri hanno preferito rimanere conformi al linguaggio di backend e hanno sviluppato soluzioni che permettono la realizzazione di interfacce utente moderne usando lo stesso linguaggio di backend.

All'interno del sistema Laravel, la necessità di creare frontend moderni e dinamici usando prevalentemente PHP ha portato alla creazione di Laravel Livewire e Alpine.js

OTTIMIZZAZIONE

OTTIMIZZARE AUTOLOADER

Quando si distribuisce alla produzione, assicurarsi che si stia ottimizzando la classe di Composer autoloader in modo che Composer possa trovare velocemente i file giusti da caricare per una data classe:

```
composer install --optimize-autoloader --no-dev
```

Oltre all'ottimizzazione dell'autoloader, bisogna essere sempre sicuri di includere un file **composer.lock** nel repository sorgente del progetto. In questo modo l'installazione delle dipendenze sarà molto più rapida

CONFIGURAZIONE DEL CACHING

In fase di deployment, accertarsi di lanciare il comando Artisan

```
php artisan config:cache
```

In modo da combinare tutti i file di configurazione di Laravel in un unico file che riduce il numero di movimenti all'interno del filesystem quando si caricano i valori di configurazione.

CACHING DEGLI EVENTS

Se l'applicazione sta utilizzando event discovery, è necessario memorizzare nella cache le mappature degli eventi e dei listeners.

```
php artisan event:cache
```

CACHING DELLE ROTTE

Se si sta realizzando un'applicazione vasta con molte rotte, bisognerebbe accertarsi di lanciare il comando

```
php artisan route:cache
```

Questo comando riduce tutte le registrazioni delle rotte nella chiamata ad un singolo metodo all'interno di un file cached, migliorando le performance della registrazione delle rotte quando se ne devono registrare centinaia.

CACHING DELLE VIEW

```
php artisan view:cache
```

Questo comando precompila tutte le viste Blade così che non siano compilate on demand, migliorando le prestazioni di ogni request che ritorna una view.

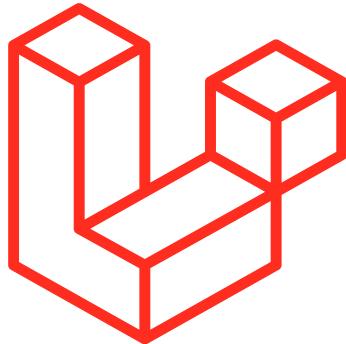
DEBUG MODE

Le opzioni di debug all'interno del file di configurazione config/app.php determinano quante informazioni su un errore vengono mostrate all'utente.

Di default è impostato per rispettare il valore della variabile d'ambiente APP_DEBUG che si trova all'interno del file .env dell'applicazione.

In ambiente di produzione questo valore dovrebbe sempre essere impostato su false.

Se fosse settato su true, il rischio è di esporre importanti valori di configurazione agli utenti finali dell'app.



Laravel

ARCHITECTURE CONCEPTS

COSA SONO LE RICHIESTE HTTP

Le richieste HTTP sono messaggi inviati al server dal client per iniziare una determinata azione.

Le richieste hanno una struttura ben precisa e sono caratterizzate da un metodo (o verbo) che permette di distinguere il tipo di azione richiesta.

STRUTTURA DELLE RICHIESTE HTTP

Le richieste HTTP è composta da:

- Un METODO che indica l'azione richiesta
- Una RISORSA verso cui si sta eseguendo l'azione
- Potrebbero avere uno o più HEADER (facoltativo) che caratterizzano la richiesta
- Un BODY (opzionale) che contiene i dati da inviare alla risorsa

ESEMPIO DI RICHIESTA HTTP:

GET /guide/laravel/index.php HTTP/2 ← Indicazione del metodo e della risorsa (guide/laravel/index.php)

Host: https://example.com

Accept-Encoding: gzip, deflate, br

Accept-Language: it-IT, it;q=0.9, en-US;q=0.8, en;q=0.7

User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/108.0.0.0 Safari/537.36

HEADERS nella forma **Nome-Header:valore**
L'header Host permette di sapere verso quale host indirizzare la richiesta.
Non è presente il body.

COSA SONO LE RICHIESTE HTTP – i metodi

I metodi delle richieste HTTP identificano **l'azione desiderata da compiere sulla risorsa**. Possono essere nomi ma sono di solito indicati come verbi HTTP.

Il set completo dei metodi HTTP è pensato per permettere ai client di caricare o modificare risorse sul server.

I **principali metodi HTTP** sono:

- **GET**: richiede la risorsa indicata che viene servita dal server come **body** della risposta. Essendo usata per richiedere dati al server, **non dovrebbe contenere un body**
- **HEAD**: stesso effetto della GET ma **la risposta del server è senza il body**. Può essere **utile da eseguire prima della GET per ottenere informazioni sulla risorsa** dal server come, ad esempio, la dimensione della risorsa fornite nella risposta con l'header Content-Length
- **POST**: **invia dati al server** usando la risorsa indicata. Viene generalmente inviata tramite form HTML e comporta una modifica sul server. In questo caso il tipo di contenuto (valori possibili sono: application/x-www-form-urlencoded e multipart/form-data) viene gestito dall'elemento <form>. Se la richiesta POST è inviata tramite un metodo diverso da form, ad esempio con XMLHttpRequest, il body può essere di qualsiasi tipo (ad es. application/json)
- **PUT**: **crea una nuova risorsa o sostituisce la risorsa indicata con il body della richiesta**. È un metodo idempotente: chiamarlo una o più volte successivamente ha lo stesso effetto, mentre successive richieste POST identiche possono aggiungere risorse (ad esempio ad ogni POST su un server e-commerce creo un nuovo ordine).
- **DELETE**: **elimina** la risorsa indicata
- **PATCH**: **applica una modifica parziale** alla risorsa. Come il metodo POST può avere effetti aggiuntivi ogni volta che è chiamata su una risorsa. A differenza del metodo PUT, **nel body possono essere indicate «istruzioni» per modificare la risorsa**. Non tutti i server web implementano questo metodo per le varie risorse
- **OPTIONS**: chiede le **opzioni di comunicazione per la risorsa** indicata (ad esempio il server può rispondere quali altri metodi sono applicabili alla risorsa).

CICLO DI VITA DELLA REQUEST – Request Lifecycle

1- FIRST STEP

Il punto di ingresso di tutte le requests ad un'applicazione Laravel è il file **public/index.php**.

Tutte le richieste vengono dirette a questo file dal Web Server.

Il file index.php non contiene molto codice, è piuttosto un punto di inizio per il caricamento di tutto il framework.

Carica la definizione del autoloader di Composer e recupera un'istanza dell'applicazione da **bootstrap/app.php**

La prima azione svolta da Laravel stesso è la creazione di un'istanza dell'applicazione /service container

```
documentazione-laravel > public > index.php > ...
1  <?php
2
3  use Illuminate\Contracts\Http\Kernel;
4  use Illuminate\Http\Request;
5
6  define('LARAVEL_START', microtime(true));
7
8  /*
9  |-----
10 | Check If The Application Is Under Maintenance
11 |-----
12 |
13 | If the application is in maintenance / demo mode via the "down" command
14 | we will load this file so that any pre-rendered content can be shown
15 | instead of starting the framework, which could cause an exception.
16 |
17 */
18
19 if (file_exists($maintenance = __DIR__.'/../storage/framework/maintenance.php')) {
20     require $maintenance;
21 }
22
23 /*
24 |-----
25 | Register The Auto Loader
26 |-----
27 |
28 | Composer provides a convenient, automatically generated class loader for
29 | this application. We just need to utilize it! We'll simply require it
30 | into the script here so we don't need to manually load our classes.
31 |
32 */
33
34 require __DIR__.'/../vendor/autoload.php';
35
36 /*
37 |-----
38 | Run The Application
39 |-----
40 |
41 | Once we have the application, we can handle the incoming request using
42 | the application's HTTP kernel. Then, we will send the response back
43 | to this client's browser, allowing them to enjoy our application.
44 |
45 */
46
47 $app = require_once __DIR__.'/../bootstrap/app.php';
48
49 $kernel = $app->make(Kernel::class);
50
51 $response = $kernel->handle(
52     | $request = Request::capture()
53 )->send();
54
55 $kernel->terminate($request, $response);
56
```

2 – HTTP / CONSOLE KERNEL

Dopo un primo caricamento del framework, la request è inviata o al **kernel HTTP** o al **kernel console**, a seconda del tipo di richiesta.

Questi due kernel servono come **luogo centrale in cui tutte le richieste passano**.

L'HTTP kernel si trova in `app/http/Kernel.php` ed estende la classe `Illuminate\Foundation\Http\Kernel` che definisce un array di «bootstrappers» che saranno lanciati prima che la request venga eseguita.

Questi «bootstrappers» configurano la **gestione degli errori**, il **logging**, rilevano l'**ambiente dell'applicazione** ed eseguono altre azioni che devono essere svolte prima che la richiesta sia effettivamente gestita.

L'HTTP Kernel definisce anche una **lista di HTTP Middleware** attraverso i quali **devono passare tutte le richieste prima di essere prese in gestione**. Questi Middleware gestiscono la lettura e scrittura della sessione http, determinano se l'applicazione è in modalità Manutenzione, verificano il token CSRF (Cross-site Request Forgery) e altro.

Il metodo handle del kernel http è molto semplice:

Riceve una Request e ritorna un Response.

3- SERVICE PROVIDERS

Una delle più importanti azioni del kernel di avvio è il **caricamento dei providers dei servizi** per l'applicazione.

I service providers sono responsabili **dell'avvio di tutti i vari componenti del framework** come il **database**, le **code**, la **validazione** e i **componenti di routing**.

Tutti i service providers sono configurati nel file `config/app.php`

Laravel ciclerà all'interno di una lista di providers e istanzierà ognuno di essi.

Essenzialmente, **ogni maggiore funzionalità di Laravel è avviata e configurata da un service provider**. Questo rende i service providers uno degli aspetti fondamentali dell'intero processo di avvio di Laravel.

I Service Providers di default si trovano all'interno della cartella app/Providers.

4 – ROUTING

Uno dei più importanti service providers in un'applicazione Laravel è [App\Providers\RouteServiceProvider](#).

Esso carica i file delle rotte contenuti nella cartella `routes`.

Come funziona il RouteServiceProvider?

Una volta che l'applicazione è stata inizializzata e tutti i service providers sono stati registrati, la **Request sarà affidata al router**. Questo, a sua volta, la smisterà verso una rotta, un controller oltre ad eseguire qualsiasi specifico middleware della rotta.

Il middleware fornisce un comodo meccanismo per il filtraggio o l'analisi delle richieste HTTP ricevute dall'app.

Ad esempio Laravel include un middleware che consente di verificare se l'utente è autenticato. In caso negativo, il middleware reindirizzerà l'utente alla schermata di login.

Se, al contrario, l'utente è già autenticato, il middleware consentirà alla request di procedere oltre nell'app.

Alcuni middleware sono assegnati a tutte le rotte dell'applicazione mentre altri sono assegnati solo a specifiche rotte o gruppi di esse.

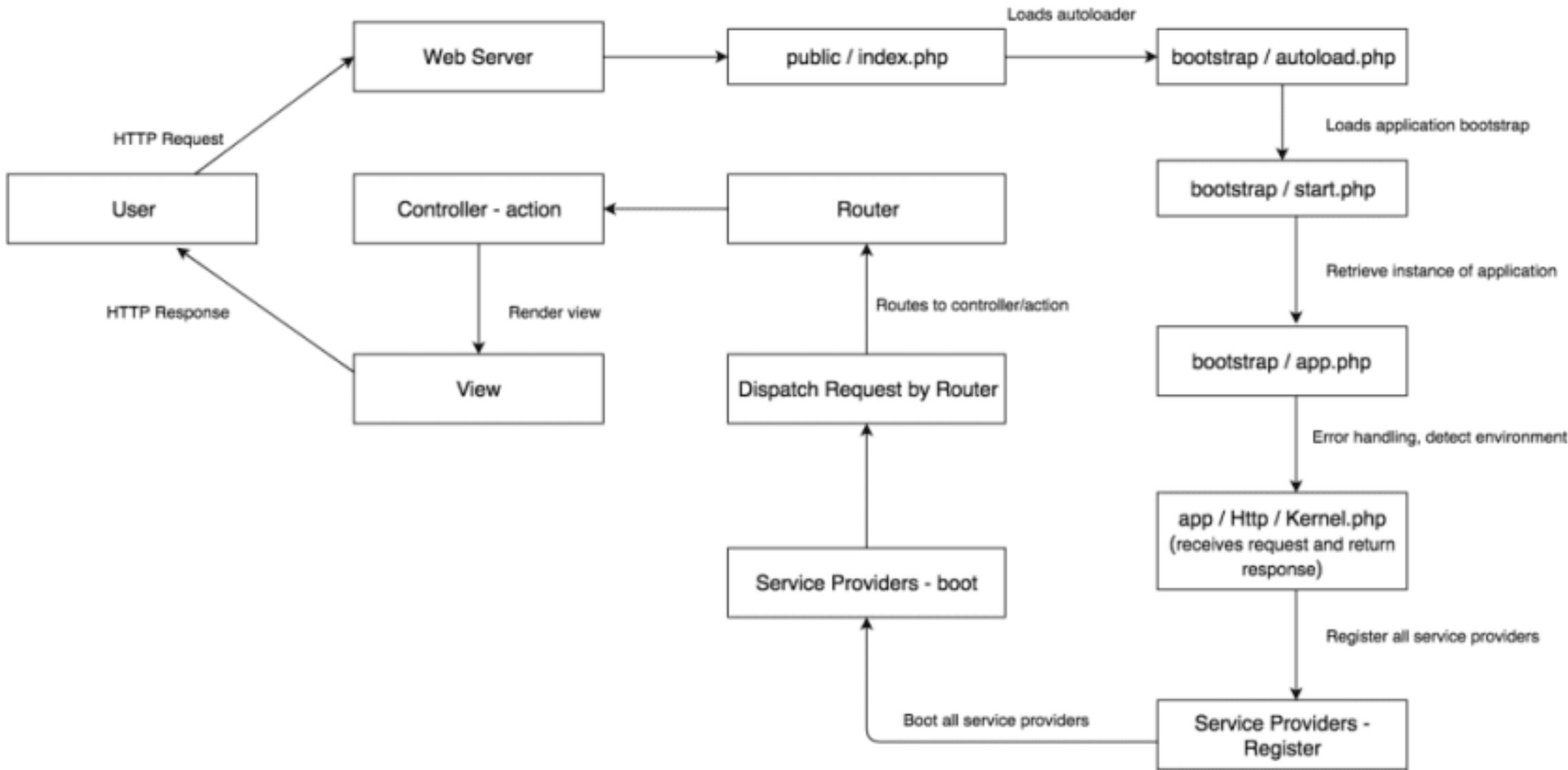
Se la richiesta attraversa tutti i middleware assegnati alla rotta corrispondente, il metodo associato alla rotta o al controller sarà eseguito e la risposta HTTP sarà ritornata attraverso la catena di middleware.

5-TERMINE: RESTITUZIONE RESPONSE

Una volta che la response è stata restituita a seconda del metodo della rotta o del controller, questa viaggia indietro attraverso il middleware della rotta dando all'applicazione la possibilità di modificare o esaminare la risposta in uscita.

Una volta attraversato il middleware, il metodo `handle` del kernel HTTP restituisce l'oggetto risposta e il file `index.php` chiama il metodo `send` sulla risposta restituita. Questo metodo invia il contenuto della risposta al browser web dell'utente.

Request Life Cycle of Laravel



SERVICE CONTAINER

Il service container di Laravel è un potente strumento per la gestione delle dipendenze delle classi e per la dependency injection.

DEPENDANCY INJECTION significa che le dipendenze della classe vengono «iniettate» nella classe tramite il costruttore o, in alcuni casi, con i metodi «setter». Questo sistema ci permette di evitare di istanziare oggetti all'interno della classe.

Nell'esempio a fianco, UserController ha bisogno di recuperare gli utenti da una fonte di dati. Inietteremo quindi un service che è in grado di recuperarli senza dovere istanziare la classe.

```
class UserController extends Controller
{
    protected $userService;

    public function __construct(UserService
        $userService)
    {
        $this->userService = $userService;
    }

    public function index()
    {
        // Utilizzo del servizio utente
        $users = $this->userService-
            >getAllUsers();
        // ...
    }
}
```

FACADES

Le facades forniscono un'interfaccia «statica» alle classi che sono disponibili nel service container dell'applicazione.

Laravel viene fornito con molte facades che consentono l'accesso a quasi tutte le funzionalità di Laravel.

Tutte le facades di Laravel sono definite nel namespace `Illuminate\Support\Facades` quindi possiamo facilmente accedere ad esse nel seguente modo:

```
use Illuminate\Support\Facades\Cache;
use Illuminate\Support\Facades\Route;

Route::get('/cache', function () {
    return Cache::get('key');
});
```

Esempio da tinker:

```
Log::info('ciao')
Config::get('app.name')
Config::get('app.timezone')
URL::to('miosito')
```

HELPER FUNCTIONS

Per completare le facades, Laravel offre una varietà di «helper functions» globali che rendono ancora più semplice interagire con le più comuni funzionalità di Laravel.

Alcune delle helper functions più comuni sono view, response, url, config,...

Ad esempio, invece di usare la facade Illuminate\Support\Facades\Response per generare una risposta JSON, possiamo semplicemente usare la funzione response().

Siccome le helper functions sono disponibili globalmente non c'è bisogno di importare alcuna classe per utilizzarle.

Esempio da tinker:

```
config('app.name')  
config('app.timezone')  
url('miosito')
```

Gli helper di Laravel sono funzioni globali che semplificano alcune operazioni comuni. La maggior parte degli helper in Laravel è definita come funzioni globali, ma alcuni sono accessibili tramite la sintassi della Facade come Arr::sort(\$array).

Alcuni hanno delle sintassi simili alle Facade

[Lista delle helper functions](#)

FACADES – quando usarle

Facades hanno molti benefici tra i quali quello di fornire una sintassi facile da ricordare che permette di utilizzare funzionalità di Laravel senza dover ricordare nomi di classi lunghissimi che dovrebbero essere configurate o iniettate manualmente.

Vanno però utilizzate con cautela.

Il rischio principale è che, non richiedendo injections, è facile che la classe continui a crescere e che usi molte facades in una singola classe.

Bisogna dunque prestare particolare attenzione, quando si usano facades, all'ampiezza della classe e fare in modo che il suo ambito di responsabilità rimanga limitato.

I NAMESPACES

- I namespaces permettono di raggruppare classi, interfacce, funzioni e costanti correlate tra loro al fine di evitare problemi di denominazione.
- In questo modo potranno coesistere classi con lo stesso nome inserite («incapsulate») in namespace differenti (come accade per i file con lo stesso nome ma posizionati in directory differenti)

ESEMPIO:

In una directory «var/www/html/test» abbiamo il file «prova1.php» che contiene la classe Prova

```
class Prova{  
    public function m1(){  
        echo "ciao dal metodo ".__METHOD__. " della classe <b>".__CLASS__;  
    }  
}
```

Avremo poi un file prova2.php contenente anch'esso una classe «Prova» analoga alla precedente.
Abbiamo poi un index.php in cui vengono inclusi entrambi i file

```
<?php  
include "prova1.php";  
include "prova2.php";  
?>
```

Il risultato sarà un fatal error perché i due file hanno lo stesso nome.

```
Fatal error: Cannot declare class Prova, because the name is already in use in /var/www/html/test/prova2.php on line 2
```

- Pensiamo alle due classi come a due file di un filesystem: due file con lo stesso nome non possono coesistere nella stessa directory. Devono essere in directory differenti per mantenere lo stesso nome
- I namespace consentono di superare questo problema di omonimia introducendo un sistema simile a quello che può essere definito quasi come un «filesystem digitale». All'interno dei file che contengono le classi dobbiamo indicare dove le classi contenute al loro interno sono localizzate in questo filesystem virtuale.
- Questo è possibile inserendo la keyword namespace seguita dal nome che viene assegnato a questa directory (spazio).

File prova1.php

```
namespace Spazio1;
class Prova{
    public function m1(){
        echo "ciao dal metodo ".__METHOD__." della classe <b>".__CLASS__;
    }
}
```

La classe Prova è definita all'interno del namespace Spazio1

La classe Prova è definita all'interno del namespace Spazio1

File prova2.php

```
namespace Spazio2;
class Prova(){
    public function m1(){
        echo "ciao dal metodo ".__METHOD__." della classe <b>".__CLASS__;
    }
}
```

- Se rilanciamo index.php noteremo che l'errore è sparito

- Se, all'interno di index.php, creiamo un'istanza della classe «Prova» e lanciamo la index otterremo un errore fatale

```
Fatal error: Uncaught Error: Class 'Prova' not found in /var/www/html/test/index.php:5 Stack trace: #0 {main} thrown in /var/www/html/test/index.php on line 5
```

- Perché nel file delle classi abbiamo dichiarato che risiedono in uno specifico namespace. Se vogliamo istanziare un oggetto della classe Prova dovremo specificare in quale namespace si trova la classe Prova.

```
$obj=new Spazio1\Prova;
```

Se vogliamo istanziare la classe Prova che si trova nel namespace spazio1

```
$obj=new Spazio2\Prova;
```

Se vogliamo istanziare la classe Prova che si trova nel namespace spazio2

- Nel file index.php andiamo ad istanziare due oggetti delle due classi, specificando il relativo namespace

```
include "prova1.php";
include "prova2.php";

$obj=new Spazio1\Prova;
$obj2=new Spazio2\Prova;

$obj->m1();
echo "<br>";
$obj2->m1();
```

Se lanciamo la index il risultato sarà:

```
ciao dal metodo Spazio1\Prova::m1 della classe Spazio1\Prova
ciao dal metodo Spazio2\Prova::m1 della classe Spazio2\Prova
```

- Se pensiamo a tutti quei casi in cui il nostro sito o applicazione hanno già delle librerie con classi ed aggiungiamo una nuova libreria che contiene classi con nomi già presenti nelle librerie: i namespace consentono di superare i conflitti di denominazione.
- Permettono inoltre a più persone di lavorare contemporaneamente allo stesso progetto senza il problema che vengano dichiarate classi con lo stesso nome.
- Abbiamo equiparato i namespace a directories e, come queste, anche i namespaces possono avere un'alberatura come, ad esempio, App\Backend\User\Sicurezza\Controlli

VINCOLI NELL'UTILIZZO DEI NAMESPACES

- I namespaces vanno inseriti subito dopo l'apertura del codice php, nessun codice deve precederli, tranne il caso di commenti ed eventuali declare come nell'esempio seguente:

```
<?php
declare(encoding='UTF-8');
namespace Spazio1;
class Prova(){
    public function m1(){
        echo "ciao dal metodo ".__METHOD__." della classe <b>".__CLASS__;
    }
?>
```

- I nomi dei namespace non possono iniziare con numeri e non possono contenere keywords del linguaggio php come «Class» o «Function»

GLOBAL SPACE E UTILIZZO DEL NAMESPACE

Quando in un file non viene definito un namespace, si dice che la classe è inserita in uno spazio globale, o GLOBAL SPACE.

ESEMPIO: abbiamo un file statistiche. Php in cui è stata definita la classe «statistiche» e ne è stata lanciata un'istanza

```
class Statistiche{
    public function myStat() {
        echo "ciao";
    }
}

$obj = new Statistiche;
```

Creiamo un secondo file index.php dove indichiamo il namespace «App\Backend» e che include il file statistiche.php

Creiamo un'istanza dell'oggetto di classe «Statistiche»

```
<?php  
namespace App\Backend;  
require_once('statistiche.php');  
  
$obj = new Statistiche;  
$obj->myStat();  
?>
```

Se proviamo a lanciare questo file otterremo un errore fatale: la classe Statistiche non è stata trovata perché in questo file ci troviamo nello spazio «App/Backend» mentre la classe Statistiche si trova nella root «\».

```
Fatal error: Class 'App\Backend\Statistiche' not found in /var/www/html/test/index.php on line 5
```

Per accedere ad una classe definita in uno spazio globale dobbiamo indicare un namespace separator prima della classe, cioè una backslash (dobbiamo in pratica entrare nella root).

Aggiungiamo ora il namespace separator nel file index.php e così possiamo accedere alla classe «Statistiche»:

```
<?php  
namespace App\Backend;  
require_once('statistiche.php');  
  
$obj = new \Statistiche;  
$obj->myStat();  
?>
```

Rilanciando il file otterremo la risposta «ciao» del metodo myStat.

In questo caso abbiamo fatto riferimento alla classe Statistiche con un **FULLY QUALIFIED NAME**, cioè utilizzando un namespace separator all'inizio. È come utilizzare un percorso assoluto. Altri esempi di fully qualified names sono:

\Esempio

\Esempio\Prova

\Esempio\Prova\Livello

È possibile fare riferimento al nome di una classe in altri due modi:

- Con un **QUALIFIED NAME** (nome qualificato): paragonabile al percorso relativo rispetto al namespace in cui mi trovo e contiene almeno un namespace, ma non all'inizio
- Con un **UNQUALIFIED NAME** (non contiene namespace)

IL TYPE HINTING (definire il tipo del dato)

- Il Type Hinting («suggerimento del tipo») è una funzionalità introdotta dalla versione 5 di Php che consente di specificare il tipo di oggetto/dato passato come argomento, ad una funzione o ad un metodo. Stiamo quindi dicendo al metodo che tipo di valore stiamo passando, come argomento, ad una funzione o al metodo stesso.
- A partire da php 5 è possibile dichiarare che tipo di dato viene assegnato ad una variabile e passato al metodo o alla funzione.
- Forzando, con il type hinting, il tipo di dato in ingresso, PHP considererà valido solo il dato della tipologia indicata, in caso contrario avremo un «fatal error».

IL TYPE HINTING

1. **int**: per numeri interi.
2. **float**: per numeri in virgola mobile (float).
3. **string**: per stringhe di testo.
4. **bool**: per valori booleani (true/false).
5. **array**: per array.
6. **callable**: per oggetti chiamabili (funzioni, metodi di oggetti, ecc.).
7. **object**: per istanze di una classe specifica.
8. **iterable**: per oggetti iterabili (array o oggetti che implementano l'interfaccia \Traversable).
9. **?Tipo**: per indicare che il parametro può essere di tipo Tipo oppure null (introdotto in PHP 7.1).
10. **self**: per fare riferimento alla stessa classe della classe che dichiara il type hinting.
11. **parent**: per fare riferimento alla classe genitore di quella che dichiara il type hinting.

- `function stampaIntero(int $numero) {`
- `echo $numero;`
- `}`

- `stampalntero(5); // Questo funzionerà correttamente`
- `stampalntero("hello"); // Questo genererà un errore poiché il tipo di dato passato non è un intero`

Artisan

- Artisan è l'interfaccia a riga di comando inclusa con Laravel.

Artisan esiste alla radice della tua applicazione come script artigianale e fornisce una serie di comandi utili che possono aiutarti durante la creazione della tua applicazione. Per visualizzare un elenco di tutti i comandi Artisan disponibili, puoi utilizzare il comando list:

<https://laravel.com/docs/10.x/artisan#main-content>

- `php artisan list`
- `php artisan help migrate`
Every command also includes a "help" screen which displays and describes the command's available arguments and options
- `php artisan tinker`
Laravel Tinker is a powerful REPL for the Laravel framework, powered by the PsySH package.
REPL = Read-Eval-Print Loop

POSTMAN headers

■ Gli headers più comuni di una richiesta

POST		http://localhost:8000/api/get-token				
Params	Authorization	Headers (12)	Body	Pre-request Script	Tests	Settings
Headers <small>8 hidden</small>						
Key	Value					
<input checked="" type="checkbox"/> Content-Type	application/json					
<input checked="" type="checkbox"/> Accept	application/json					
<input checked="" type="checkbox"/> User-Agent	Postman/7.36.0					
<input checked="" type="checkbox"/> Cache-Control	no-cache, max-age=0					
Key	Value					

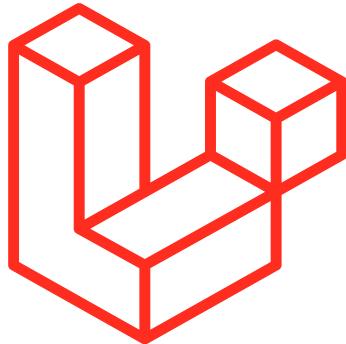
POSTMAN csrf

- Per evitare l'errore 419 passare alle chiamate POST / PUT / PATCH

POST http://127.0.0.1:8000/product

Headers (10)

Key	Value
<input checked="" type="checkbox"/> X-CSRF-TOKEN	h3IcBq0AKu8LJv41EC8jCNYiWPfKSRYwPV85YVjb
<input type="checkbox"/> cont	



Laravel

LE BASI

—

Il routing

ROUTING

Basic routing

Una rotta (Route) consente di configurare l'applicazione per **rispondere ad una chiamata per una determinata URI** in modo che possa rispondere a richieste di risorse che non sono necessariamente file specifici presenti sul server.

La route più semplice che si può definire in Laravel è:

```
use Illuminate\Support\Facades\Route;

Route::get('/greeting', function () {
    return 'Hello World';
});
```

Essa è composta da:

- **Metodo statico** della classe Route, generalmente legato al metodo HTTP della richiesta. In questo caso GET
- La stringa che rappresenta il **path o parte del path** della URI della richiesta (/greeting)
- Una **callback**, ovvero closure (function () {...}) che definisce la risposta che verrà restituita

Queste definizioni di route vanno aggiunte al file **routes/web.php** per le richieste legate alle pagine web. È importante ricordare che **l'ordine in cui le rotte vengono definite è fondamentale per il meccanismo di match**: la **prima route definita** che può gestire la richiesta arrivata sarà quella che, effettivamente, **restituirà la risposta**, anche se dovesse esserne definita una più specifica più avanti nel file. Per ogni richiesta ricevuta, infatti, Laravel scansiona le rotte registrate fino a trovarne una in grado di poter gestire il metodo e il path della richiesta.

ROUTING

Basic routing

• I FILE PREDEFINITI DELLE ROTTE

Tutte le rotte Laravel sono definite nei files che si trovano all'interno della **cartella routes**.

Questi files sono **caricati automaticamente** dal **RouteServiceProvider** dell'app (App\Providers\RouteServiceProvider).

Il file **routes/web.php** definisce le rotte per l'interfaccia web. A queste rotte viene assegnato il **gruppo middleware web** che fornisce funzionalità quali lo stato della sessione (*session state*) e la protezione CSRF (*Cross-Site Request Forgery*).

Si può accedere alle rotte definite in web.php inserendo l'**URL della rotta nel browser**.

```
use App\Http\Controllers\UserController;  
  
Route::get('/user', [UserController::class, 'index']);
```

Si può accedere a questa rotta digitando <http://example.com/user> nella barra degli indirizzi del browser

Le rotte in **routes/api.php** sono **stateless** e hanno assegnato il **middleware api**.

Esse sono **annidate** all'interno di un gruppo di rotte dal RouteServiceProvider. All'interno di questo gruppo, il **prefisso /api** dell'URI è applicato automaticamente in modo che non debba essere aggiunto manualmente ad ogni rotta del file.

Si possono **modificare il prefisso e altre opzioni** del gruppo di rotte **modificando la classe RouteServiceProvider**

ROUTING

Basic routing

• I METODI DEL ROUTER

Il router permette di registrare rotte che rispondono ad ogni metodo HTTP. Le rotte più immediate che si possono registrare sono:

```
Route::get($uri, $callback);
Route::post($uri, $callback);
Route::put($uri, $callback);
Route::patch($uri, $callback);
Route::delete($uri, $callback);
```

In caso si abbia la necessità di registrare una **rotta che risponde a più metodi HTTP**, per ottenere la medesima risposta, si può usare il metodo **match** oppure si può registrare una rotta che risponde a tutti i verbi HTTP usando il metodo **any**

```
Route::match(['get', 'post'], '/', function () {
    // ...
}); // Risposta comune per metodi GET e POST

Route::any('/', function () {
    // ...
}); // Risposta comune per OGNI RICHIESTA
```

!!! L'ORDINE È FONDAMENTALE !!!

Quando si definiscono **rotte multiple** che condividono lo stesso URI, le rotte che usano i metodi **get, post, put, patch, delete e options** dovrebbero essere definite **prima delle rotte che usano i metodi any, match e redirect**. Questo assicura che la richiesta in entrata sia gestita dalla rotta corretta.

ROUTING

Basic routing

• DEPENDENCY INJECTION

Nel caso fosse necessario utilizzare una **dipendenza esterna nella closure** di una route, è possibile sfruttare la **dependency injection** di Laravel.

Esempio: vogliamo che la risposta alla rotta /hello contenga il nome passato opzionalmente come query string alla URI.

Quindi:

- Se l'utente richiede GET/hello la risposta sarà «Hello Laravel»
- Se viene richiesto GET/hello?name=Foo la risposta sarà «Hello Foo»

```
use Illuminate\Http\Request;
Route::get('hello', function (Request $request){
    $name = $request->query('name');
    if ($name == ''){
        $name = 'Laravel';
    }
    return 'Hello ' . $name;
});
```

La funzione associata alla route dichiara di avere una **dipendenza aggiuntiva**, specificando un parametro con il **type-hinting (dichiarazione del tipo di dati)**. Nel momento in cui la funzione verrà eseguita, Laravel inietterà la richiesta arrivata come tipo Illuminate\Http\Request.
Sarà quindi possibile utilizzare questo oggetto all'interno della callback, per esempio per estrarre dati della richiesta necessari ad elaborare la risposta.

NOTA: STRUTTURA DELL'URI

<scheme>://<host><path>[?<query>]

La URI

<https://www.example.com/path/to/foo?param=value&option=none> è dunque composta da:

Schema = https

Host = www.example.com

Path = path/to/foo

Query = param=value&option=none

Una URI ha sempre almeno un **path()** mentre la **query** è opzionale

ROUTING

Basic routing

• REDIRECT ROUTES

Per definire una rotta che redireziona ad un altro URI, si può utilizzare il metodo `Route::redirect`

```
Route::redirect('/here', '/there');
```

Di default il metodo redirect ritorna uno **status code 302**. Si può **personalizzare** lo status code indicandolo come **terzo parametro** opzionale del metodo redirect

```
Route::redirect('/here', '/there', 301);
```

Si può anche utilizzare il metodo `Route::permanentRedirect` per redirezioni permanenti. Esso ritorna uno status code 301:

```
Route::permanentRedirect('/here', '/there');
```

I parametri **destination** e **status** di una rotta sono riservati a Laravel e non possono essere customizzati nelle rotte di redirect

Esempio redirect

- `Route::redirect('/news/{id}', '/news-new/{id}', 302);`
- `Route::get('/news-new/{id}', function(Request $request){`
- `return view('message')->with(['message'=>'news n.'`
- `. $request->segment(2)]);`
- `});`
- `Route::get('/news2/{id}', function(Request $request, int $id){`
- `return view('message')->with(['message'=>'news n.'`
- `. $id]);`
- `});`

ROUTING

Basic routing

• ROTTE DELLA VIEW

Se una rotta deve solamente ritornare una view, si può utilizzare il metodo Route::view.

Anche questo metodo costituisce una scorciatoia che permette di non dover definire tutta una rotta o un controller.

Come primo parametro accetta un URI e il nome di una view come secondo. Inoltre, come terzo argomento, può essere opzionalmente passato alla view un array di dati

```
Route::view('/welcome', 'welcome');
```

```
Route::view('/welcome', 'welcome', ['name' => 'Taylor']);
```

Nelle rotte delle views, i seguenti parametri sono riservati a Laravel e non possono essere utilizzati: view, data, status, headers.

• CSRF PROTECTION

Ogni form HTML che punta a rotte POST, PUT, PATCH o DELETE definite nel file routes/web.php dovrebbero includere un campo per il token CSRF. In caso contrario la richiesta sarà rifiutata.

```
<form method="POST" action="/profile">  
    @csrf  
    ...  
</form>
```

ROUTING

Route Parameters

ROUTES PARAMETRICHE

Laravel permette di gestire con una singola rotta URI, rotte in cui sono presenti elementi variabili.

La possibilità di definire rotte con parti dinamiche del path della URI permette di:

- Creare URI più semplici rispetto a quelle con query string ed in linea con le buone pratiche SEO (per esempio <https://www.example.com/blog/45887> invece di <https://www.example.com/blog/post?id=45887>)
- Essere coerenti con le buone pratiche delle API RESTful
- Gestire porzioni intermedie del path, per esempio <https://www.example.com/blog/topic/lifestyle/posts/> e <https://www.example.com/blog/topic/comedy/posts/>

• PARAMETRI RICHIESTI (Required Parameters)

Nel definire una rottà è possibile istruire Laravel affinchè una porzione della URI sia considerata **variabile** e venga messa a **disposizione delle funzione di callback**.

```
Route::get('/user/{id}', function (string $id) {
    return 'User ' . $id;
});
```

Rispetto ad una rottà statica (rottà che accetta solo un possibile path), si può notare che:

- La **parte variabile è compresa tra parentesi graffe {}**
- Nella funzione di callback è presente un parametro (\$id)

IMPORTANTE:

- la parte variabile del path dovrebbe contenere **solo caratteri alfabetici e underscore (_)**
- È possibile indicare **più parti variabili** che verranno poi **associate ai parametri della callback in base all'ordine** (non è quindi necessario usare gli stessi nomi)

I parametri della rottà saranno iniettati nella callback o nel controller della rottà a seconda del loro ordine.

```
Route::get('/posts/{post_id}/comments/{comment_id}',
    function ($post, $commentId) {
        return 'This is the comment ' . $commentId . ' for post ' . $post;
});
```

ROUTING
Route
Parameters

• PARAMETRI OPZIONALI

Qualora fosse necessario specificare un parametro che non è sempre presente nella URI (parametro opzionale) basterà **aggiungere un punto di domanda (?) dopo il nome del parametro** ed indicare un valore di default alla variabile corrispondente

```
Route::get('/user/{name?}', function (string $name = null) {
    return $name;
});

Route::get('/user/{name?}', function (string $name = 'John') {
    return $name;
});
```

ROUTING
Route
Parameters

ROUTING

Route Parameters

- **PARAMETRI E DEPENDENCY INJECTION**

Per utilizzare una dipendenza esterna nella funzione di callback di una route con parametri, è necessario indicare i **parametri con dependency injection prima dei parametri di rotta**

```
use Illuminate\Http\Request;

Route::get('/user/{id}', function (Request $request, string $id) {
    return 'User '.$id;
});
```

Parametro con dependency injection Parametro di rotta

• VINCOLI REGULAR EXPRESSION e VALIDAZIONE

Nel caso sia necessario **validare il valore ricevuto come parametro**, si può concatenare il metodo **where** alla definizione della rotta indicando, come nome della where, il **nome del parametro** e un'**espressione regolare** di validazione.

Ogni parte variabile della URI può avere la propria validazione

```
Route::get('/user/{name}', function (string $name) {
    // ...
})->where('name', '[A-Za-z]+');

Route::get('/user/{id}', function (string $id) {
    // ...
})->where('id', '[0-9]+');

Route::get('/user/{id}/{name}', function (string $id, string $name) {
    // ...
})->where(['id' => '[0-9]+', 'name' => '[a-z]+']);
```

ROUTING
Route
Parameters

Per comodità, per gli schemi di regular expressions più comuni esistono dei **metodi helper**.

```
Route::get('/user/{id}/{name}', function (string $id, string $name) {
    // ...
})->whereNumber('id')->whereAlpha('name');
```



```
Route::get('/user/{name}', function (string $name) {
    // ...
})->whereAlphaNumeric('name');
```



```
Route::get('/user/{id}', function (string $id) {
    // ...
})->whereUuid('id');
```



```
Route::get('/user/{id}', function (string $id) {
    //
})->whereUlid('id');
```



```
Route::get('/category/{category}', function (string $category) {
    // ...
})->whereIn('category', ['movie', 'song', 'painting']);
```

ROUTING

Route Parameters

ROUTING

Named Routes

Si può anche specificare nomi di rotta per le azioni del controller:

```
Route::get(  
    '/user/profile',  
    [UserProfileController::class, 'show']  
)->name('profile');
```

!!! I NOMI DELLE ROTTE DOVREBBERO SEMPRE ESSERE UNICI!!!

Le **rotte nominate** (a cui viene assegnato un nome) permettono la **generazione di URLs o redirects per rotte specifiche**. Si può specificare il nome di una rotta concatenando il metodo **name** alla definizione della rotta.

```
Route::get('/user/profile', function () {  
    // ...  
})->name('profile');
```

Dopo aver nominato una rotta

```
Route::get('posts/{id}', function($id){  
    return 'id='.$id;  
})->name ('route.name');  
?>
```

è possibile fare riferimento ad essa nel **front-end** in questo modo:

```
<a href=<? route('route.name', ['id'=> 5])?>></a>
```

La funzione **route** è un helper di Laravel che consente di trasformare il nome nell'url della rotta.

I **VANTAGGI** di dare un nome alle rotte sono principalmente due:

- Possibilità di fare **riferimento a rotte anche complesse** in modo rapido
- **Se una rotta cambia, non si deve modificare anche il front-end**

ROUTING

Named Routes

Generazione di rotte per rotte nominate

Una volta assegnato un nome ad una data rotta, si può utilizzare il nome della rotta quando si generano URLs o redirects attraverso le funzioni helper route e redirect

```
Route::get('/user/{id}/profile', function (string $id) {
    // ...
})->name('profile');

$url = route('profile', ['id' => 1]);
```

Se si passano parametri addizionali nell'array, queste coppie chiave/valore saranno automaticamente aggiunte nella query string dell'URL generato

```
// Generating URLs...
$url = route('profile');

// Generating Redirects...
return redirect()->route('profile');

return to_route('profile');
```

Se la rotta ha dei parametri, questi vengono passati come secondo argomento della funzione route. Questi parametri saranno automaticamente inseriti nell'URL generato e nella loro esatta posizione.

```
Route::get('/user/{id}/profile', function (string $id) {
    // ...
})->name('profile');

$url = route('profile', ['id' => 1, 'photos' => 'yes']);

// /user/1/profile?photos=yes
```

ROUTING

Named Routes

CONTROLLARE LA ROTTA CORRENTE

Per determinare se la richiesta corrente è stata indirizzata ad una determinata rotta nominata, si può usare il metodo `named` sull'istanza di una Route. Ad esempio, si può controllare il nome della rotta corrente da un route middleware

```
use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

/**
 * Handle an incoming request.
 *
 * @param \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response) $next
 */
public function handle(Request $request, Closure $next): Response
{
    if ($request->route()->named('profile')) {
        // ...
    }

    return $next($request);
}
```

GRUPPI DI ROTTE

I gruppi di rotte permettono di **condividere attributi**, come i middleware, tra un numero variabile di rotte senza il bisogno di definirli per ogni singola rotta.

I gruppi annidati cercano di «unire» gli attributi con quelli del gruppo genitore:
I middleware e le condizioni **where** sono unificati (merged) mentre i nomi ed i prefissi sono aggiunti (appended).

I delimitatori del namespace e gli slashes nei prefissi dell'URI sono automaticamente aggiunti dove necessario.

ROUTING
Gruppi di
rotte

• MIDDLEWARE

Per assegnare middleware ad un gruppo di rotte si usa il metodo **middleware** prima di definire il gruppo.

I middleware **vengono eseguiti nell'ordine in cui sono elencati nell'array**:

```
Route::middleware(['first', 'second'])->group(function () {
    Route::get('/', function () {
        // Uses first & second middleware...
    });

    Route::get('/user/profile', function () {
        // Uses first & second middleware...
    });
});
```

ROUTING
Gruppi di
rotte

• PREFISSI DELLE ROTTE

Il metodo prefix può essere utilizzato per aggiungere un prefisso ad ogni rotta nel gruppo con un dato URI. Si potrebbe, ad esempio, voler aggiungere il prefisso 'admin' a tutte le URI nel gruppo.

```
Route::prefix('admin')->group(function () {
    Route::get('/users', function () {
        // Matches The "/admin/users" URL
    });
});
```

Si pensi al caso in cui si devono gestire URI con path come /user/orders e /user/profile. Si possono raggruppare insieme usando il metodo prefix:

```
Route::prefix('/user')->group(function () {
    Route::get('/profile', function () {
        // risposta per l'URI "/user/profile"
    });

    Route::get('/orders', function () {
        // risposta per l'URI "/user/orders" URL
    });
});
```

ROUTING
Gruppi di
rotte

• PREFISSI DEL NOME DELLE ROTTE

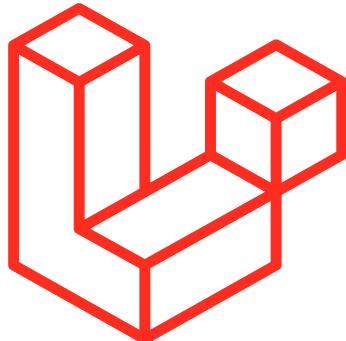
Il metodo name può essere utilizzato per aggiungere una stringa ad ogni nome di rotta nel gruppo. Si potrebbe, ad esempio, voler aggiungere il prefisso ‘admin’ a tutte le URI nel gruppo.

Ad esempio, si può voler aggiungere come prefisso a tutte le rotte di un dato gruppo, la stringa ‘admin.’.

La stringa è aggiunta al nome della rotta esattamente così come è specificata

```
Route::name('admin.')->group(function () {
    Route::get('/users', function () {
        // Route assigned name "admin.users"...
    })->name('users');
});
```

ROUTING
Gruppi di
rotte



Laravel

LE BASI

—
I middleware

MIDDLEWARE

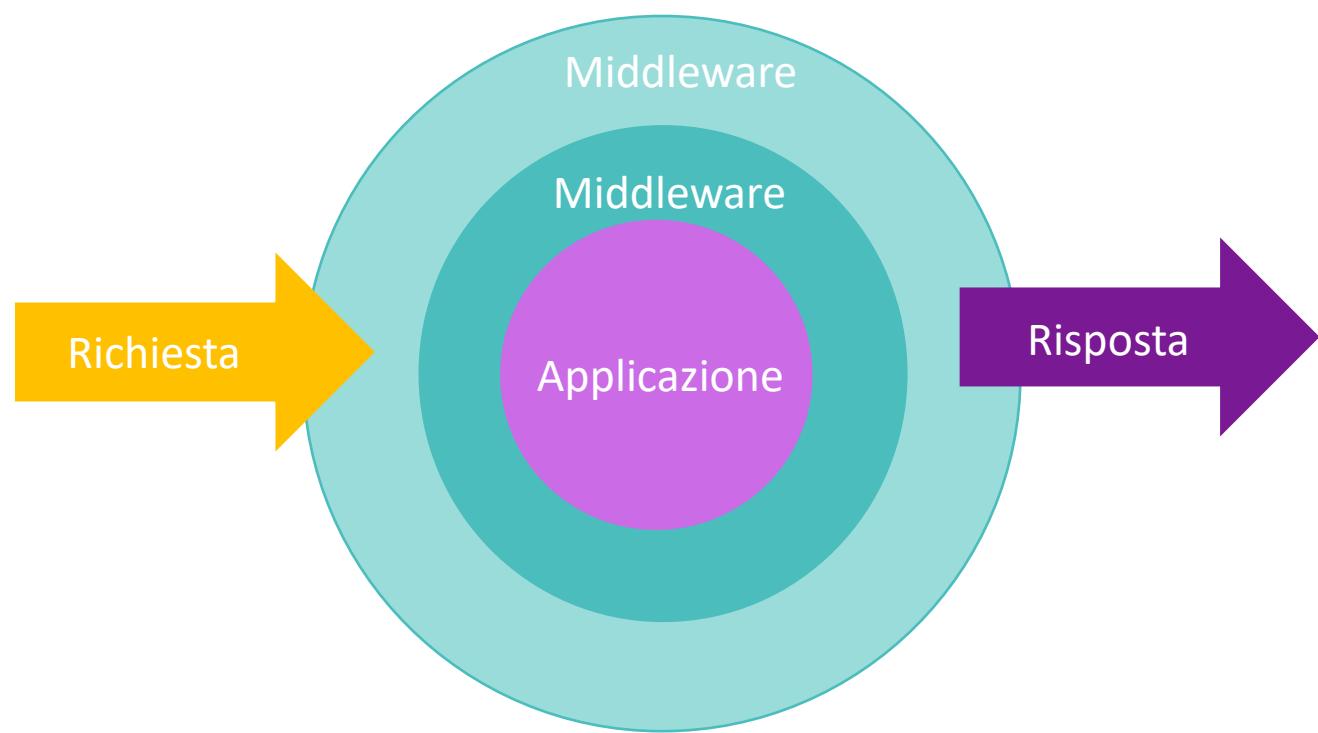
Il Middleware fornisce un comodo meccanismo per **esaminare e filtrare** le **richieste HTTP** che arrivano all'applicazione.

Ad esempio, Laravel include un middleware che verifica se l'utente è autenticato. In caso negativo, il middleware ridirezionerà l'utente alla schermata di login. Invece, se l'utente è autenticato, il middleware consentirà alla richiesta di procedere.

Middleware aggiuntivi possono essere creati per eseguire una varietà di compiti. Per esempio, un middleware di logging, uno per limitare le richieste per secondo all'applicazione, cambiare lingua del sito, ecc.

All'interno di Laravel sono inclusi diversi Middleware, inclusi quelli per l'autenticazione e per la protezione CSRF (Cross Site Request Forgery) e si trovano nella cartella **app/http/Middleware**.

COME SI COMPORTA UN MIDDLEWARE?



1. Il middleware intercetta una Request
2. Verifica delle condizioni
3. Se le condizioni NON sono soddisfatte, allora effettua una **REDIRECT** verso una route predeterminata
4. Se le condizioni sono verificate, la richiesta viene processata normalmente.

MIDDLEWARE

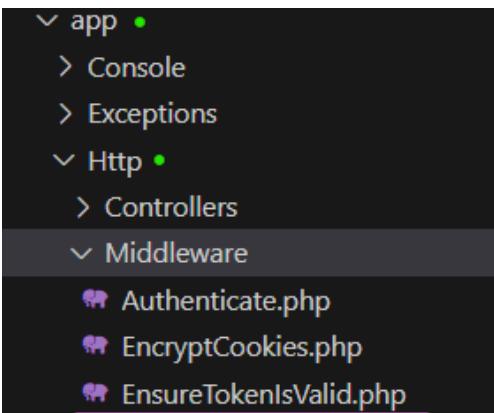
1- DEFINIRE MIDDLEWARE

Per creare un nuovo middleware, si usa il comando Artisan

`make:middleware`

```
php artisan make:middleware EnsureTokenIsValid  
Middleware created successfully.
```

Questo comando posizionerà una nuova classe **EnsureTokenIsValid** all'interno della directory **app/http/Middleware**.



In questo middleware ci limiteremo a consentire l'accesso alla rotta sole se il token fornito in input corrisponde ad un determinato valore. In caso negativo, gli utenti saranno ridirezionati alla home.

Perché la richiesta possa passare il middleware, si deve chiamare la callback **\$next** con la **\$request**.

È bene immaginare un middleware come una **serie di strati attraverso i quali la richiesta HTTP deve passare prima che arrivi all'applicazione vera e propria.**

Ogni strato o livello può esaminare la richiesta e rigettarla.

MIDDLEWARE

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class EnsureTokenIsValid
{
    /**
     * Handle an incoming request.
     *
     * @param \Illuminate\Http\Request $request
     * @param \Closure(\Illuminate\Http\Request): (\Illuminate\Http\Response|\Illuminate\Http\RedirectResponse) $next
     * @return \Illuminate\Http\Response|\Illuminate\Http\RedirectResponse
     */
    public function handle(Request $request, Closure $next): Response
    {
        if ($request->input('token') !== 'my-secret-token'){
            return redirect('home');
        }
        return $next($request);
    }
}
```

Tutto avviene all'interno del metodo pubblico `handle` che prende come parametri la `$request` (contenente l'oggetto `Request`) e la callback `$next` che ha il compito di **processare la request se le condizioni sono soddisfatte.**

MIDDLEWARE

1.1 MIDDLEWARE E RISPOSTE

Un middleware può svolgere delle azioni prima o dopo aver passato la richiesta più internamente all'applicazione. Ad esempio, il middleware seguente esegue certe funzioni prima che la richiesta sia gestita dall'applicazione.

```
namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class BeforeMiddleware
{
    public function handle(Request $request, Closure $next): Response
    {
        // Perform action

        return $next($request);
    }
}
```

Mentre quest'altro middleware eseguirà i suoi compiti dopo che la richiesta è passata all'applicazione

```
namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class AfterMiddleware
{
    public function handle(Request $request, Closure $next): Response
    {
        $response = $next($request);

        // Perform action

        return $response;
    }
}
```

MIDDLEWARE

2- REGISTRARE MIDDLEWARE

Prima di poter utilizzare qualsiasi middleware, questi vanno registrato nel file app/http/Kernel.php.

Ci sono due tipi di middleware: i middleware globali e i route middleware.

Il file app/http/Kernel.php contiene due proprietà usate per registrare i middleware:

- \$middleware: per i Global Middleware
- \$routeMiddleware per registrare middleware specifici per determinate routes.

2.1 MIDDLEWARE GLOBALI

Se si desidera lanciare un middleware durante ogni richiesta HTTP, si elenca la classe middleware nella proprietà \$middleware della classe app/http/Kernel.php.

```
class Kernel extends HttpKernel
{
    /**
     * The application's global HTTP middleware stack.
     *
     * These middleware are run during every request to your application.
     *
     * @var array<int, class-string|string>
     */
    protected $middleware = [
        // \App\Http\Middleware\TrustHosts::class,
        \App\Http\Middleware\TrustProxies::class,
        \Fruitcake\Cors\HandleCors::class,
        \App\Http\Middleware\PreventRequestsDuringMaintenance::class,
        \Illuminate\Foundation\Http\Middleware\ValidatePostSize::class,
        \App\Http\Middleware\TrimStrings::class,
        \Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull::class,
    ];
}
```

MIDDLEWARE

2- REGISTRARE MIDDLEWARE

2.1 ASSEGNARE MIDDLEWARE ALLE ROTTE

Per assegnare il middleware a specifiche rotte, si invoca il metodo middleware quando si definisce la rotta:

```
Route::get('/', function () {
    // ...
})->middleware([First::class, Second::class]);
```

```
use App\Http\Middleware\Authenticate;

Route::get('/profile', function () {
    // ...
})->middleware(Authenticate::class);
```

Si possono assegnare middleware multipli alla rotta passando un array di nomi di middleware al metodo middleware:

MIDDLEWARE

Per comodità, si possono assegnare alias ai middleware nel file app/http/Kernel.php.

Di default, la proprietà \$middlewareAliases di questa classe contiene dati per i middleware inclusi in Laravel. Si può però aggiungere il proprio middleware alla lista e assegnargli un alias a scelta.

Una volta assegnato l'alias al middleware nel kernel HTTP, lo si può utilizzare per assegnare il middleware alla rotta:

```
// Within App\Http\Kernel class...

protected $middlewareAliases = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,
    'cache.headers' => \Illuminate\Http\Middleware\SetCacheHeaders::class,
    'can' => \Illuminate\Auth\Middleware\Authorize::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'signed' => \Illuminate\Routing\Middleware\ValidateSignature::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
    'verified' => \Illuminate\Auth\Middleware\EnsureEmailIsVerified::class,
];
```

```
Route::get('/profile', function () {
    // ...
})->middleware('auth');
```

MIDDLEWARE

2.1.1 ESCLUDERE MIDDLEWARE

Quando si assegnano middleware ad un gruppo di rotte, si può occasionalmente aver bisogno di escludere l'applicazione del middleware ad una singola rotta all'interno del gruppo.

Questo si può ottenere usando il metodo `withoutMiddleware`

```
use App\Http\Middleware\EnsureTokenIsValid;

Route::withoutMiddleware([EnsureTokenIsValid::class])->group(function () {
    Route::get('/profile', function () {
        // ...
    });
});
```

```
use App\Http\Middleware\EnsureTokenIsValid;

Route::middleware([EnsureTokenIsValid::class])->group(function () {
    Route::get('/', function () {
        // ...
    });

    Route::get('/profile', function () {
        // ...
    })->withoutMiddleware([EnsureTokenIsValid::class]);
});
```

E' anche possibile escludere un dato set di middleware da un intero gruppo di rotte.
Il metodo `withoutMiddleware` può solamente rimuovere i middleware della rotta e non si applica ai middleware globali

MIDDLEWARE

2.2 GRUPPI DI MIDDLEWARE

Si possono raggruppare diversi middleware sotto un'unica chiave per rendere più semplice l'assegnazione alle rotte. Questo è possibile grazie alla proprietà `$middlewareGroups` del kernel HTTP.

Di default Laravel include i gruppi middleware web e api che contengono i middleware più comuni applicabili alle rotte web e api. Questi due gruppi di middleware sono automaticamente applicati dal service provider

`App\Providers\RouteServiceProvider` alle rotte nei file `routes/web.php` e `router/api.php`

La sintassi per l'assegnazione di gruppi di middleware a rotte e controller è uguale a quella dei singoli middleware

```
/*
 * The application's route middleware groups.
 *
 * @var array
 */
protected $middlewareGroups = [
    'web' => [
        \App\Http\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
        \Illuminate\Session\Middleware\StartSession::class,
        \Illuminate\View\Middleware\ShareErrorsFromSession::class,
        \App\Http\Middleware\VerifyCsrfToken::class,
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
    ],
    'api' => [
        \Illuminate\Routing\Middleware\ThrottleRequests::class.':api',
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
    ],
];
```

```
Route::get('/', function () {
    // ...
})->middleware('web');

Route::middleware(['web'])->group(function () {
    // ...
});
```

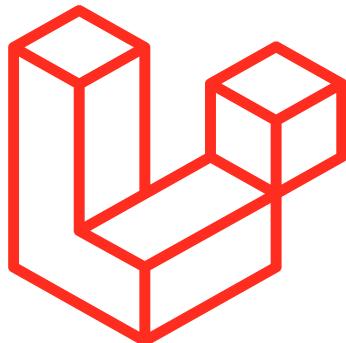
MIDDLEWARE

2.3 ORDINARE I MIDDLEWARE

Quando i middleware vengono assegnati alla rotta, non si ha controllo sul loro ordine. Nel caso si abbia la necessità di eseguire i middleware in un ordine specifico, si può specificare la priorità dei middleware usando la proprietà `$middlewarePriority` nel file `app/http/Kernel.php`

Questa proprietà potrebbe non esistere di default nel file `kernel` HTTP. In questo caso si può copiare la sua definizione predefinita:

```
/**  
 * The priority-sorted list of middleware.  
 *  
 * This forces non-global middleware to always be in the given  
 * order.  
 *  
 * @var string[]  
 */  
protected $middlewarePriority = [  
\Illuminate\Foundation\Http\Middleware\HandlePrecognitiveRe  
quests::class,  
\Illuminate\Cookie\Middleware\EncryptCookies::class,  
\Illuminate\Session\Middleware\StartSession::class,  
\Illuminate\View\Middleware\ShareErrorsFromSession::class,  
\Illuminate\Contracts\Auth\Middleware\AuthenticatesRequests  
::class,  
\Illuminate\Routing\Middleware\ThrottleRequests::class,  
\Illuminate\Routing\Middleware\ThrottleRequestsWithRedis::cl  
ass,  
\Illuminate\Contracts\Session\Middleware\AuthenticatesSessio  
ns::class,  
\Illuminate\Routing\Middleware\SubstituteBindings::class,  
\Illuminate\Auth\Middleware\Authorize::class,  
];
```



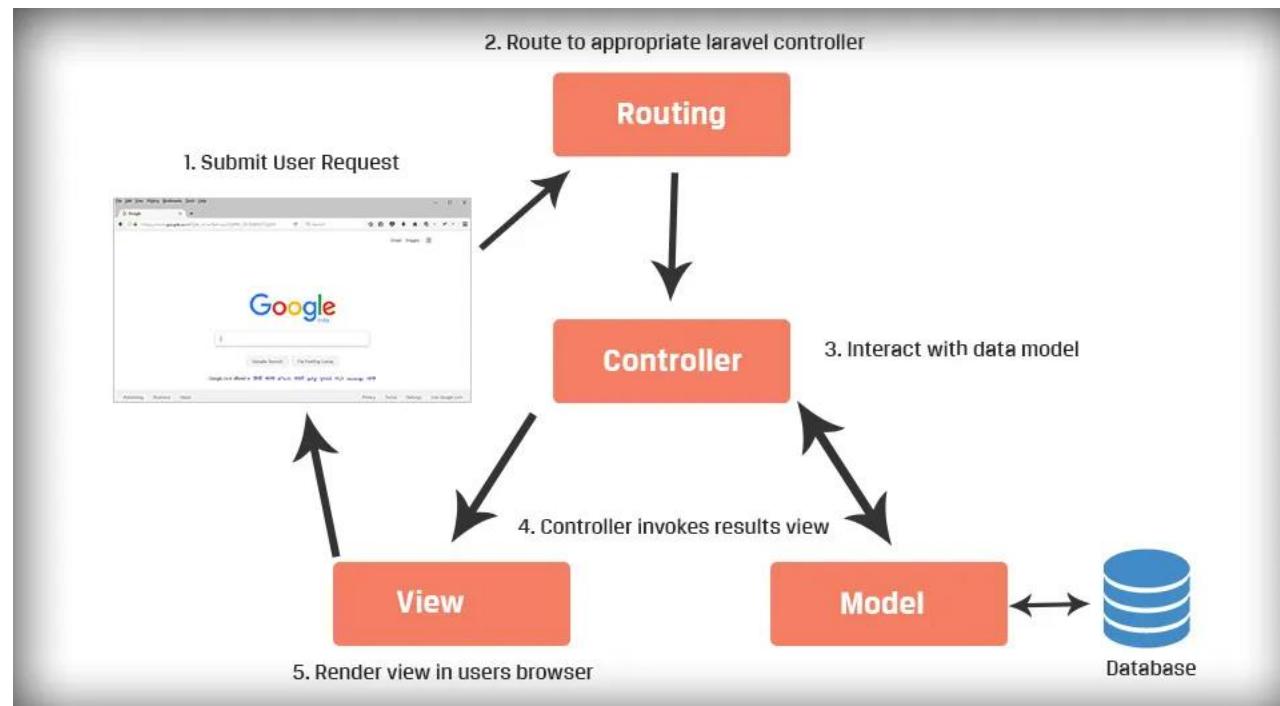
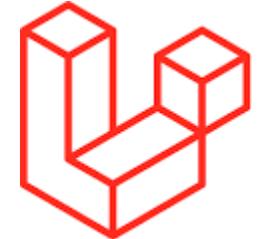
Laravel

LE BASI

—

Controllers

1 - Introduzione ai controller



Laravel è pensato per rendere semplice la **separazione dei compiti (separation of concern)** tra diverse classi.

Per questo motivo è possibile separare la definizione di una rotta (intesa come definizione della URI e del metodo HTTP) dalla gestione del comportamento per generare la risposta.

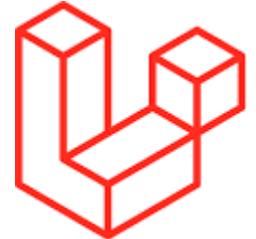
Le classi che si occupano della logica di gestione delle richieste sono i controller.

Una classe UserController, ad esempio, potrebbe gestire tutte le richieste in entrata relative agli utenti, incluse la lettura, creazione, modifica e cancellazione degli utenti.

Di default, tutti i controller si trovano all'interno della cartella **app/http/Controllers**.

2 - CREARE CONTROLLER

2.1 CONTROLLER BASE



```
php artisan make:controller UserController
```

Comando per la creazione di un nuovo controller all'interno della cartella app/http/Controllers.

È bene usare il **suffisso Controller** per il nome del controller e della rispettiva classe.

Un controller può avere **qualsiasi numero di metodi pubblici** che **rispondono alle richieste HTTP in entrata**.

Una volta creato un controller e i suoi metodi, dovremo **definire una rotta (o gruppo di rotte) alla quale collegare il metodo del controller**:

```
use App\Http\Controllers\UserController;

Route::get('/user/{id}', [UserController::class, 'show']);
```

Quando una richiesta in entrata corrisponde alla rotta URI specifica, il metodo show del controller UserController sarà invocato e i **parametri della rotta saranno passati al metodo**.

```
namespace App\Http\Controllers;

use App\Models\User;
use Illuminate\View\View;

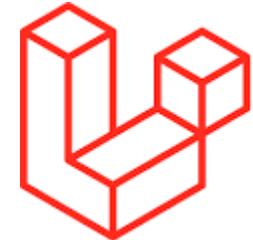
class UserController extends Controller
{
    /**
     * Show the profile for a given user.
     */
    public function show(string $id): View
    {
        return view('user.profile', [
            'user' => User::findOrFail($id)
        ]);
    }
}
```

```
use App\Http\Controllers\PostController;
Route::get('/posts', [PostController::class, 'index']);
    //il metodo index della classe PostController gestirà le richieste GET/post
Route::get('/posts/{id}', [PostController::class, 'show']);
    //il metodo show della classe PostController (che riceverà come parametro $id)
    // gestirà le richieste GET/post/{id}
```

Quando si hanno **più rotte collegate ad un controller**, ad esempio quando si possono compiere più azioni sulla medesima entità, è possibile **raggruppare tali rotte usando il metodo controller per definire il controller comune** ed **indicare solo il metodo del controller da invocare sulla singola rotta**:

```
use App\Http\Controllers\PostController;
Route::controller(PostController::class)->group(function () {
    Route::get('/posts', 'index');
    Route::get('/posts/{id}', 'show');
    Route::post('/posts', 'store');
});|
```

2.1.1 METODI DI UN CONTROLLER E PARAMETRI



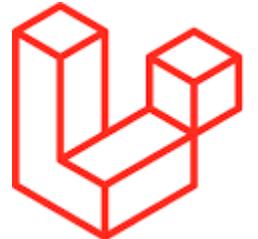
```
namespace App\Http\Controllers;
use Illuminate\Http\Request;
class PostController extends Controller
{
    public function index(): string
    {
        return 'List of blog posts';
    }
    public function show($id): string
    {
        return "Content for blog post $id";
    }
    public function store(Request $request): string
    {
        return 'Created new blog post';
    }
}
```

I controller **POSSONO** estendere la classe Controller, non è necessario.

I nomi dei metodi corrispondono a quanto dichiarato nella definizione della rotta. Se non corrispondessero, il metodo del controller non verrebbe invocato.

I metodi del controller **possono avere parametri a cui verrà passato il valore del o dei parametri della rotta.**

È possibile sfruttare la **dependency injection** per passare istanze specifiche al metodo, come ad esempio, l'istanza della Request.



2.2 SINGLE ACTION CONTROLLERS (invokable controllers)

Se l'azione di un controller è particolarmente complessa, può essere utile dedicare un'intera classe controller a quella azione. Per ottenere questo si può definire un unico metodo `__invoke` all'interno del controller.

Quando si registrano le rotte per controller invocabile, non serve specificare il metodo del controller. Si può semplicemente passare il nome del controller al router:

```
use App\Http\Controllers\ProvisionServer;

Route::post('/server', ProvisionServer::class);
```

Si può creare un **invokable controller** usando l'opzione `--invokable` del comando Artisan make:controller

```
php artisan make:controller ProvisionServer --invokable
```

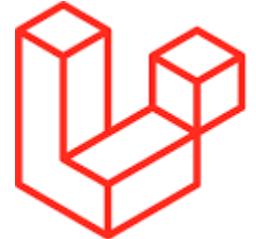
```
namespace App\Http\Controllers;

use App\Models\User;
use Illuminate\Http\Response;

class ProvisionServer extends Controller
{
    /**
     * Provision a new web server.
     */
    public function __invoke()
    {
        // ...
    }
}
```

Si può invocare senza definire il metodo

3 - CONTROLLER MIDDLEWARE



I middleware possono essere assegnati alle rotte dei controller durante la definizione delle rotte:

```
Route::get('profile', [UserController::class, 'show'])->middleware('auth');
```

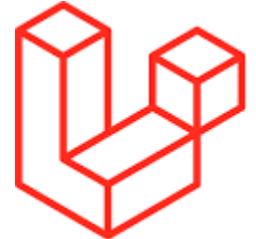
Oppure il middleware può essere **specificato all'interno del costruttore del controller usando il metodo middleware**. Si assegnerà così il middleware alle azioni del controller

```
class UserController extends Controller
{
    /**
     * Instantiate a new controller instance.
     */
    public function __construct()
    {
        $this->middleware('auth');
        $this->middleware('log')->only('index');
        $this->middleware('subscribed')->except('store');
    }
}
```

```
use Closure;  
use Illuminate\Http\Request;  
  
$this->middleware(function (Request $request, Closure $next) {  
    return $next($request);  
});
```

- I controllers consentono inoltre di **registrare i middleware usando una closure**. Questo è un metodo utile per **definire un middleware inline per un singolo controller senza definire un'intera classe middleware**.

4 - RESOURCE CONTROLLERS



Se si pensa ad **ogni modello Eloquent come ad una «risorsa»**, è tipico eseguire lo **stesso set di azioni su ogni risorsa**. Ad esempio, immaginiamo che la nostra applicazione contenga un model Photo ed un model Movie. E' probabile che gli utenti possano creare, leggere, modificare o cancellare entrambe queste risorse.

Per questo il **routing delle risorse** di Laravel **assegna le tipiche rotte create, read, update e delete (CRUD) ad un controller con una singola riga di codice**.

Per iniziare si può usare l'opzione **--resource** del comando `make:controller` per **creare rapidamente un controller che gestisce queste azioni**:

```
php artisan make:controller PhotoController --resource
```

Verrà così creato un controller `app/http/Controllers/PhotoController.php` che **conterrà un metodo per ogni operazione possibile sulle risorse**. Si può in seguito **registrare una rotta che punti a quel controller**:

```
use App\Http\Controllers\PhotoController;  
  
Route::resource('photos', PhotoController::class);
```

Questa singola dichiarazione **crea molteplici rotte per gestire una varietà di azioni sulle risorse.**

Il controller creato avrà già i metodi per ognuna di queste azioni.

Si possono anche **registrare tanti resource controllers uno alla volta passando un array al metodo resource:**

```
Route::resources([
    'photos' => PhotoController::class,
    'posts'  => PostController::class,
]);
```

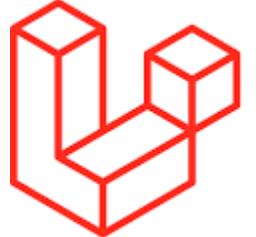
Si ricorda che è possibile dare una veloce occhiata a tutte le rotte disponibili nell'applicazione grazie al comando **route:list**. Esso evidenzierà quale controller e quale metodo sono collegati alla rotta.

```
$ php artisan route:list --except-vendor
GET|HEAD posts ..... PostController@index
POST posts ..... PostController@store
GET|HEAD posts/{id} ..... PostController@show
GET|HEAD user ..... 
GET|HEAD user/orders ..... 
GET|HEAD user/profile .....
```

A - AZIONI GESTITE DAL CONTROLLER DELLE RISORSE

Pur non essendo fondamentale per la connessione tra rotte e controller, è buona pratica usare **alcuni nomi convenzionali per i metodi** del controller in base al tipo di richiesta che stanno gestendo.

VERB (METODO HTTP)	URI	METODO DEL CONTROLLER	ROUTE NAME	AZIONE
GET	/photos	index	photos.index	Elenca tutte le foto
GET	/photos/create	create	photos.create	Mostra il form per la creazione di una nuova risorsa
POST	/photos	store	photos.store	Crea una nuova foto
GET	/photos/{photo}	show	photos.show	Mostra la risorsa con specifico id
GET	/photos/{photo}/edit	edit	photos.edit	Mostra il form per modificare la risorsa specificata
PUT/PATCH	/photos/{photo}	update	photos.update	Modifica ed aggiorna la risorsa con id specificato
DELETE	/photos/{photo}	destroy	photos.destroy	Rimuove la risorsa specificata dal database



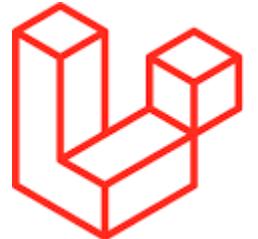
4.1 – PARTIAL RESOURCE ROUTES

Quando si dichiara la rotta di una risorsa, si può specificare un **insieme di azioni che il controller dovrebbe gestire al posto dell'intero set di azioni di default**:

```
use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class)->only([
    'index', 'show'
]);

Route::resource('photos', PhotoController::class)->except([
    'create', 'store', 'update', 'destroy'
]);
```



4.1.1 – ROTTE DELLE RISORSE API

Quando si dichiarano rotte per le risorse che saranno consumate da APIs, si tende a voler escludere le rotte che presentano template HTML come **create** e **edit**.

Il metodo **apiResource** automaticamente **esclude** queste due rotte

Si possono **registrare più API resource controllers in una volta sola** passando un **array** al metodo **apiResources**

Per generare velocemente un **controller per le risorse API che non include i metodi create e edit**, si usa **--api** quando si esegue il comando **make:controller**



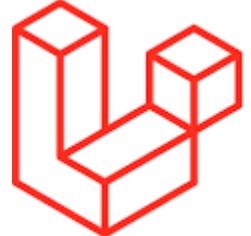
```
use App\Http\Controllers\PhotoController;  
  
Route::apiResource('photos', PhotoController::class);
```



```
use App\Http\Controllers\PhotoController;  
  
use App\Http\Controllers\PostController;  
  
Route::apiResources([  
    'photos' => PhotoController::class,  
    'posts' => PostController::class,  
]);
```



```
php artisan make:controller PhotoController --api
```



4.2 – RISORSE ANNIDATE

Si può avere la necessità di definire rotte a risorse annidate. Ad esempio una risorsa photo potrebbe avere diversi commenti allegati alla photo.

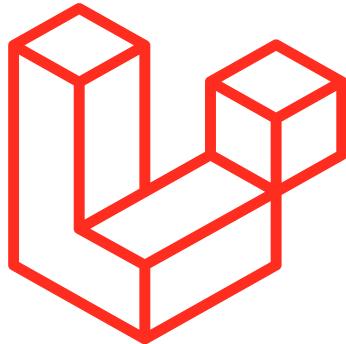
Per annidare i controller delle risorse, si può utilizzare la **dot notation nella dichiarazione della rotta**:

```
use App\Http\Controllers\PhotoCommentController;

Route::resource('photos.comments', PhotoCommentController::class);
```

Questa rotta registrerà una risorsa annidata che potrebbe essere acceduta con gli URI come il seguente:

```
/photos/{photo}/comments/{comment}
```



Laravel

LE BASI

—
HTTP Requests

La classe Illuminate\Http\Request fornisce un modo object-oriented per interagire con l'attuale richiesta HTTP gestita dall'applicazione così come l'input, i cookies ed i file che sono stati inviati con la request.

1 - INTERAGIRE CON LA REQUEST

1.1 – Accedere la Request

Per ottenere un'istanza della request HTTP corrente attraverso la dependency injection, si può usare il type-hint della classe Illuminate\Http\Request nella closure della route o nel metodo del controller.

L'istanza della richiesta in entrata sarà automaticamente iniettata dal service container di Laravel:

```
use Illuminate\Http\Request;

Route::get('/', function (Request $request) {
    // ...
});
```

Closure della route

```
namespace App\Http\Controllers;

use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Store a new user.
     */
    public function store(Request $request): RedirectResponse
    {
        $name = $request->input('name');

        // Store the user...

        return redirect('/users');
    }
}
```

Metodo del controller

1.1.a – Dependency Injection e parametri della Route

Se il metodo del controller si aspetta un input dal parametro della rotta, bisogna elencare i parametri della rotta dopo le altre dipendenze.
Ad esempio, se la nostra rotta è definita così:

```
use App\Http\Controllers\UserController;

Route::put('/user/{id}', [UserController::class, 'update']);
```

Possiamo ancora fare il type-hint della classe Request ed accedere il parametro id della rotta definendo il metodo del controller in questo modo:

```
namespace App\Http\Controllers;

use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Update the specified user.
     */
    public function update(Request $request, string $id): RedirectResponse
    {
        // Update the user...

        return redirect('/users');
    }
}
```

1.2 – PATH, HOST E METODI DELLA REQUEST

L'istanza della Request fornisce una varietà di metodi per esaminare la richiesta HTTP in entata ed estende la classe Symfony\Component\HttpFoundation\Request.

1.2.a – Recuperare il path della Request (metodo path)

Il metodo path ritorna informazioni circa il path della request.

Se la richiesta è diretta a `http://example.come/foo/bar`, il metodo path ritornerà `foo/bar`

```
$uri = $request->path();
```

1.2.b – Ispezionare il path della Request/ Route (metodo is)

Il metodo is permette di verificare che il path delle richiesta corrisponda ad un dato pattern. Si può usare * come jolly:

```
if ($request->is('admin/*')) {  
    // ...  
}
```

Usando il metodo routels si può determinare se la richiesta corrisponde ad una named route.

```
if ($request->routeIs('admin.*')) {  
    // ...  
}
```

1.2.c – Recuperare l’URL della Request

Si usano i metodi url o fullUrl:

- Metodo url: ritorna l’URL senza la query string
- Metodo fullUrl: ritorna l’URL con la query string

```
$url = $request->url();  
  
$urlWithQueryString = $request->fullUrl();
```

Per aggiungere i dati della query string all’URL corrente, si usa il metodo fullUrlWithQuery che farà il merge dell’array delle variabili della query string con la query string corrente:

```
$request->fullUrlWithQuery(['type' => 'phone']);
```

Per ottenere l’Url corrente senza un dato parametro della query string, si può usare il metodo fullUrlWithoutQuery

```
$request->fullUrlWithoutQuery(['type']);
```

1.2.d – Recuperare l’host della Request

Di può recuperare l’»host» della request con i metodi host, httpHost e schemeAndHttpHost

```
$request->host();  
  
$request->httpHost();  
  
$request->schemeAndHttpHost();
```

1.2.e – Recuperare il metodo della Request

Ritorna il metodo HTTP della request grazie al metodo isMethod che permette di verificare se il metodo http corrisponde ad una data stringa.

```
$method = $request->method();  
  
if ($request->isMethod('post')) {  
    // ...  
}
```

1.3 HEADERS DELLA REQUEST

- Si può ricavare l'header di una request dall'istanza della classe Illuminate\Http\Request usando il metodo header. Se l'header non fosse presente sarà ritornato un null.
- Il metodo header accetta un secondo parametro opzionale che può essere ritornato in caso l'header non fosse presente nella request.
- Il metodo hasHeader può essere usato per determinare se la request contiene un dato header
- Per comodità il metodo bearerToken può essere usato per recuperare un bearer token dall'header Authorization. Se non è presente un token di questo tipo, verrà ritornata una stringa vuota. I bearer tokens sono tipi particolari di Access tokens usati per ottenere l'accesso ad una risorsa protetta da un Authorization Server.

```
$value = $request->header('X-Header-Name');

$value = $request->header('X-Header-Name', 'default');
```

```
if ($request->hasHeader('X-Header-Name')) {
    // ...
}
```

```
$token = $request->bearerToken();
```

1.4 INDIRIZZO IP DELLA REQUEST

- Il metodo ip serve a ricavare l'indirizzo ip del client che ha effettuato la richiesta all'applicazione.

```
$ipAddress = $request->ip();
```

1.5 NEGOZIAZIONE DEL CONTENUTO

- Laravel fornisce diversi metodi per ispezionare il tipo di contenuto richiesto della request attraverso l'header Accept.
 - Metodo getAcceptableContentTypes: ritorna un array contenente tutti i tipi di contenuto accettati dalla request
 - Metodo accepts: accetta un array di content types e ritorna vero se uno qualsiasi dei tipi è accettato dalla request.

```
$contentTypes = $request->getAcceptableContentTypes();
```

```
if ($request->accepts(['text/html', 'application/json'])) {  
    // ...  
}
```

- Metodo `prefers` per determinare quale tipo di contenuto è preferito dalla request all'interno di un dato array di content types. Se nessuno dei tipi forniti è accettato dalla richiesta, viene restituito null.
- Siccome molte applicazioni servono solo HTML o JSON, il metodo `expectsJson` determina se la richiesta si aspetta una risposta JSON

```
$preferred = $request->prefers(['text/html', 'application/json']);
```

```
if ($request->expectsJson()) {  
    // ...  
}
```

2 - INPUT

2.1 RECUPERARE L'INPUT

2.1.1 RECUPERARE TUTTI I DATI DELL'INPUT

- Il metodo all permette di recuperare tutti i dati ricevuti in input con la request.
Si può utilizzare senza prestare troppa attenzione alla provenienza della richiesta (da un form HTML o da una richiesta XHR)

```
$input = $request->all();
```

- Il metodo collect recupera tutti i dati di input della request come una collection.
Consente anche di recuperare un subset di input della request come collection.

```
$input = $request->collect();
```

```
$request->collect('users')->each(function (string $user) {  
    // ...  
});
```

2.1.2 RECUPERARE IL VALORE DI UN INPUT

- Mentre il metodo input recupera i valori dall'intero payload della request (inclusa la query string), il metodo query recupera valori solo dalla query string
- Si può passare un valore di default come secondo argomento del metodo input. Questo valore sarà ritornato se il valore dell'input richiesto non è presente sulla request.
- Quando si lavora con form che contengono array di input, si usa la dot notation per accedere gli array.
- Si usa il metodo input senza parametri per recuperare tutti i valori degli input in un array associativo

```
$name = $request->query('name');
```

```
$name = $request->input('name', 'Sally');
```

```
$name = $request->input('products.0.name');
```

```
$names = $request->input('products.*.name');
```

```
$input = $request->input();
```

2.1.3 RECUPERARE INPUT DALLA QUERY STRING

- Il metodo INPUT può essere usato per recuperare gli input dell'utente indipendentemente dal metodo HTTP utilizzato per la request.
- Se il valore del dato richiesto non è presente, viene ritornato il secondo argomento del metodo, se presente.
- Si usa il metodo query senza argomenti per recuperare tutti i valori degli input in un array associativo

```
$name = $request->input('name');
```

```
$name = $request->query('name', 'Helen');
```

```
$query = $request->query();
```

2.1.4 RECUPERARE VALORI JSON

- Si possono accedere i dati JSON con il metodo `input` sempre che il Content-Type dell'header della request sia opportunamente impostato su `application/json`.
- Si può usare la dot notation per accedere valori annidati negli array o negli oggetti JSON.

```
$name = $request->input('user.name');
```

2.1.5 RECUPERARE VALORI DI INPUT BOOLEANI

- Quando si ha a che fare con elementi HTML come checkboxes, l'applicazione potrebbe ricevere valori che effettivamente sono stringhe. Ad esempio “true” o “on”. Il metodo `boolean` recupera questi valori come booleani.
Ritorna true per 1, “1”, true, “true”, “on” e “yes”.
Tutti gli altri valori ritornano false.

```
$archived = $request->boolean('archived');
```

2.1.6 RECUPERARE DATE COME VALORI DI INPUT

- I valori di input contenenti date ed orari possono essere estratti con il metodo date.

Se la richiesta non contiene un valore di input con il nome indicato, sarà ritornato null

```
$birthday = $request->date('birthday');
```

- Il secondo e terzo argomento accettati dal metodo date possono essere usati per specificare il formato della data e la timezone:

```
$elapsed = $request->date('elapsed', '!H:i', 'Europe/Madrid');
```

- Se il valore dell'input è presente ma ha un formato non valido, verrà lanciata un InvalidArgumentException. È quindi raccomandato validare l'input prima ed invocare il metodo date

2.1.7 RECUPERARE VALORI ENUM

- Il metodo enum permette di recuperare i valori che corrispondono a enums PHP. Il metodo accetta come argomenti il nome del valore di input e la classe enum.

```
use App\Enums>Status;  
  
$status = $request->enum('status', Status::class);
```

- Se la request non contiene un valore di input con il nome dato o se l'enum non ha un backing value che corrisponde al valore di input, sarà ritornato null.

2.1.8 RECUPERARE INPUT ATTRAVERSO PROPRIETA' DINAMICHE

- Se un form contiene un campo name, si può accedere al valore del campo nel seguente modo:

```
$name = $request->name;
```

- Quando si usano proprietà dinamiche, Laravel cercherà prima il valore del parametron nel payload della richiesta. Se non è presente, lo cercherà nei parametri della route.

2.1.9 RECUPERARE PARTE DEI DATI DI INPUT

- Per recuperare solo un sottoinsieme di dati di input, si possono usare i metodi only ed except. Entrambi accettano un singolo array o una lista dinamica di argomenti

```
$input = $request->only(['username', 'password']);  
  
$input = $request->only('username', 'password');  
  
$input = $request->except(['credit_card']);  
  
$input = $request->except('credit_card');
```

- Il metodo only ritorna tutte le coppie chiave/valore che si richiedono ma non ritorna le coppie chiave/valore non indicate nella richiesta.

2.2 DETERMINARE SE L'INPUT È PRESENTE

- Il metodo has consente di determinare se un valore è presente nella richiesta. Il metodo has ritorna true se il valore è presente

```
if ($request->has('name')) {  
    // ...  
}
```

- Dato un array, il metodo has determina se TUTTI i valori specificati sono presenti

```
if ($request->has(['name', 'email'])) {  
    // ...  
}
```

- Il metodo hasAny ritorna true se almeno uno dei valori specificati sono presenti.

```
if ($request->hasAny(['name', 'email'])) {  
    // ...  
}
```

- Il metodo whenHas esegue la closure data se un valore è presente nella richiesta.

Al metodo whenHas può essere passata una seconda closure che sarà eseguita se il valore specificato non è presente nella request

```
$request->whenHas('name', function (string $input) {  
    // ...  
});
```

```
$request->whenHas('name', function (string $input) {  
    // The "name" value is present...  
}, function () {  
    // The "name" value is not present...  
});
```

2.4.3 RECUPERARE VECCHI INPUT

- Per recuperare gli input memorizzati dalla richiesta precedente, si invoca il metodo old su un'istanza di Illuminate\Http\Request.

Il metodo old estrapolerà i precedenti dati di input dalla session:

```
$username = $request->old('username');
```

- Laravel fornisce anche un helper old.

Se si sta visualizzando un vecchio input all'interno di un template Blade, è più comodo usare l'helper old per ripopolare il form.

Se non esiste un vecchio input per il campo richiesto, sarà ritornato null:

```
<input type="text" name="username" value="{{ old('username') }}>
```

1- IL FILE INDEX.PHP

Il file public/index.php è quindi lo script PHP che viene invocato ogni volta che il server riceve una richiesta HTTP. Questo file contiene solamente il codice necessario a caricare il resto del framework.

```
require __DIR__.'/../vendor/autoload.php'; //require dell'autoload -> FONDAMENTALE
```

```
$app = require_once __DIR__.'/../bootstrap/app.php'; //recupera un'istanza dell'applicazione Laravel $app

$kernel = $app->make(Kernel::class); // carica il kernel opportuno (a seconda del tipo di richiesta sarà un kernel HTTP o un kernel console)

$response = $kernel->handle(
    $request = Request::capture()
)->send(); // fai gestire la $request arrivata al kernel e invia una $response

$kernel->terminate($request, $response);
```

Questo ciclo viene ripetuto ad ogni richiesta che arriva al server, passando la richiesta da uno stato all'altro finno a giungere alla parte di codice che gestirà e restituirà il contenuto della risposta.

2- LA SELEZIONE DEL MIDDLEWARE

A seconda del tipo di richiesta, essa viene indirizzata verso un middleware (Gruppo di rotte) specifico.

Quando l'app viene inizializzata, viene caricato un servizio interno per la gestione delle rotte la cui versione di default è la classe App\Providers\RouteServiceProvider contenuta nel file app/Providers/RouteServiceProvider.php, prevede di default che:

```
$this->routes(function () {
    Route::prefix('api')
        ->middleware('api')
        ->namespace($this->namespace)
        ->group(base_path('routes/api.php'));

    Route::middleware('web')
        ->namespace($this->namespace)
        ->group(base_path('routes/web.php'));
});
```

Tutte le richieste di risorse il cui path inizia con '**api**' appartengono al **middleware api** e verranno smistate all'interno del file **routes/api.php** per un ulteriore successivo smistamento.
Ogni altra richiesta va al **middleware web** e verrà successivamente smistata dal file **routes/web.php**

RICHIESTA	MIDDLEWARE
GET/ api /orders	Api
POST/page/form	Web
GET/docs/api	web

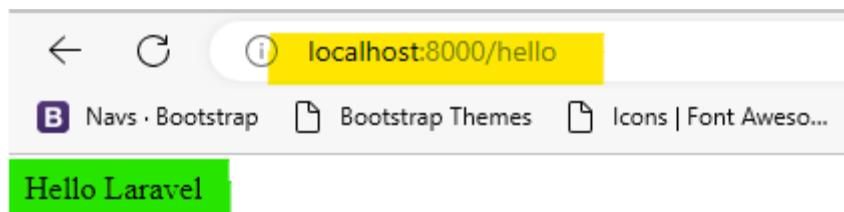
RICHIESTA	MIDDLEWARE
GET /blog/2022/12/23/post.html	Web
DELETE / api /orders/23455	api
DELETE /orders/23455	web

3 - LA SELEZIONE DELLA ROTTA REGISTRATA

I file routes/web.php e routes/api.php, anche se in modo leggermente differente, contengono le **effettive rotte registrate**

FILE routes/web.php

```
Route::get('/', function () {
    return view('welcome');
});
Route::get('/hello', function () {
    return "Hello Laravel";
});
```



Se esiste una **rotta registrata per il path della richiesta**, allora sarà eseguita la **funzione corrispondente**.

ESEMPI:

Se nel middleware web arrivasse una richiesta con path **GET/hello**, verrà ritornato il testo «**Hello Laravel**» che **diventerà il body della risposta HTTP con status code 200 OK**.

Per il path **GET/** verrà invocata una funzione che effettuerà il **rendering della view 'welcome'**: verrà cercato il template associato, elaborato e il risultato restituito come testo (o come HTML) che diventerà il body della risposta HTTP.

Ogni altra richiesta, sia per metodo che per risorsa richiesta, **non avendo una rotta registrata, non potrà essere soddisfatta** e Laravel restituirà una risposta con **status 404**

FILE routes/api.php

```
Route::middleware('auth:sanctum')->get('/user', function (Request $request) {
    return $request->user();
});
```

NOTA: Laravel permette la definizione di una rotta anche **tramite controller**. Per ora lo si può pensare come un modo per raggruppare in una singola classe rotte che fanno parte dello stesso aggregato

LE ROTTE ED ARTISAN

È possibile ottenere un elenco completo delle rotte registrate tramite il comando Artisan:

```
php artisan route:list
```

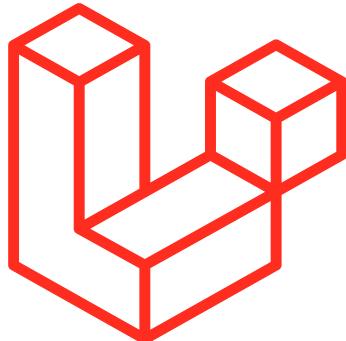
Domain	Method	URI	Name	Action	Middleware
	GET HEAD	/		Closure	web
	GET HEAD	api/user		Closure	api
					App\Http\Middleware\Authenticate:sanctum
	GET HEAD	hello		Closure	web
	GET HEAD	sanctum/csrf-cookie		Laravel\Sanctum\Http\Controllers\CsrfCookieController@show	web

OPZIONI:

- **-v** mostra anche i middleware associati
- **--path** mostra solo le rotte che contengono solo una data stringa nell'URI. Esempio `--path=user`

```
>php artisan route:list --path=user
```

Domain	Method	URI	Name	Action	Middleware
	GET HEAD	api/user		Closure	api



Laravel

LE BASI

—

Protezione CSRF

Il CSRF (Cross-site Request Forgery)

Il CSRF è un tipo di attacco informatico e si tratta fondamentalmente di una falsificazione di richieste tra siti: un utente autenticato viene convinto ad inviare una richiesta specifica verso un app web. Si parla di falsificazione della richiesta perché la richiesta di un utente malintenzionato viene mascherata come una richiesta proveniente da un utente legittimo.

Questo tipo di attacco sfrutta l'incapacità di un'applicazione di distinguere tra richieste legittime e consensuali all'interno di una sessione autenticata di un utente.

La CSRF punta le richieste che modificano il valore dei dati come acquisto di prodotti, modifica password, trasferimento denaro, ecc.

L'utente malintenzionato falsifica una richiesta, come la modifica delle info di accesso, quindi aggiunge un link con la richiesta all'interno di una mail o in un sito. Quando l'utente clicca sul collegamento, la richiesta viene inviata in modo automatico.

Il CSRF – spiegazione della vulnerabilità

Immaginiamo che la nostra app abbia una rotta /user/email che accetta una richiesta POST per modificare l'indirizzo email dell'utente autenticato. È facile che questa rotta si aspetti un campo di input email contenente l'indirizzo che l'utente vorrebbe iniziare ad usare.

Senza protezione CSRF, un sito web malevolo potrebbe creare un form HTML che punta alla rotta /user/mail della nostra app e invia l'indirizzo email dell'utente malintenzionato

```
<form action="https://your-application.com/user/email" method="POST">
    <input type="email" value="malicious-email@example.com">
</form>

<script>
    document.forms[0].submit();
</script>
```

Se il sito malevolo invia automaticamente il form quando la pagina è caricata, basta solo attirare un utente della nostra applicazione a visitare quel sito e il loro indirizzo email sarebbe cambiato nella nostra applicazione.

Per prevenire questo tipo di vulnerabilità, bisogna esaminare ogni richiesta POST, PUT , PATCH o DELETE in ingresso alla nostra app alla ricerca in un valore di sessione segreto a cui l'app dannosa non può accedere.

Il CSRF – prevenire richieste CSRF

Per ogni sessione utente attiva gestita dall'applicazione, Laravel genera un CSRF token usato per verificare che l'utente autenticato sia la persona che effettivamente sta inviando una richiesta all'app.

Dal momento in cui questo token è salvato nella sessione utente e cambia ogni volta che la sessione è rigenerata, un'applicazione malevola sarebbe incapace di accedervi.

Il CSRF token attuale può essere acceduto tramite la **sessione della richiesta** e attraverso la funzione helper **csrf_token**

```
use Illuminate\Http\Request;

Route::get('/token', function (Request $request) {
    $token = $request->session()->token();

    $token = csrf_token();

    // ...
});
```

Il CSRF – prevenire richieste CSRF

Ogni volta che definiamo nella nostra app un form HTML per i **metodi POST, PUT, PATCH, DELETE**, dovremmo includere un **campo nascosto CSRF _token** in modo che il **middleware di protezione CSRF possa convalidare la richiesta**.

Per comodità si può utilizzare la **direttiva Blade @csrf** per generare un campo nascosto per il token.

```
<form method="POST" action="/profile">
    @csrf

    
    <input type="hidden" name="_token" value="{{ csrf_token() }}" />
</form>
```

Il middleware VerifyCsrfToken, incluso nel gruppo dei middleware web di default, verificherà automaticamente che il token nella richiesta corrisponda al token salvato nella sessione.

Quando i due token combaciano, sappiamo che l'utente autenticato è quello che ha inviato la richiesta.

Il CSRF – escludere URIs dalla protezione CSRF

Csrf tokens & SPA – omitted perché solo riferimento a Sanctum

Se si vogliono escludere delle URIs dalla protezione CSRF, magari perché si sta utilizzando il sistema di payment processing Stripe, esistono due modalità:

1. Mettere queste rotte al di fuori del gruppo middleware web che il RouteServiceProvider applica a tutte le rotte web.
2. Aggiungere gli URIs nella proprietà \$except del middleware VerifyCsrfToken

Per comodità il CSRF middleware è automaticamente disattivato per tutte le rotte quando si lanciano i test

```
<?php

namespace App\Http\Middleware;

use Illuminate\Foundation\Http\Middleware\VerifyCsrfToken as Middleware;

class VerifyCsrfToken extends Middleware
{
    /**
     * The URIs that should be excluded from CSRF verification.
     *
     * @var array
     */
    protected $except = [
        'stripe/*',
        'http://example.com/foo/bar',
        'http://example.com/foo/*',
    ];
}
```

Il CSRF X-CSRF-TOKEN e X-XSRF-TOKEN

X-CSRF-TOKEN

Oltre a cercare il token CSRF come parametro POST, il middleware VerifyCsrfToken controllerà anche l'header della request per il token X-CSRF. Questo token potrebbe ad esempio essere memorizzato in un meta tag HTML

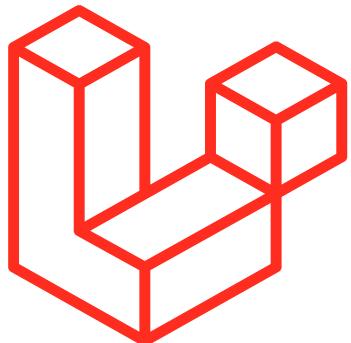
Si può poi istruire jQuery ad aggiungere automaticamente il token a tutte gli header delle richieste. Questo fornisce una semplice protezione per l'app basata su AJAX.

```
<meta name="csrf-token" content="{{ csrf_token() }}>
```

```
$.ajaxSetup({
  headers: {
    'X-CSRF-TOKEN': $('meta[name="csrf-token"]').attr('content')
  }
});
```

X-XSRF-TOKEN

Laravel memorizza il token CSRF attuale in un cookie XSRF-TOKEN criptato che è incluso con ogni risposta generata da Laravel . Si può usare questo cookie per impostare il X-XSRF-TOKEN header della request.



Laravel

LE BASI

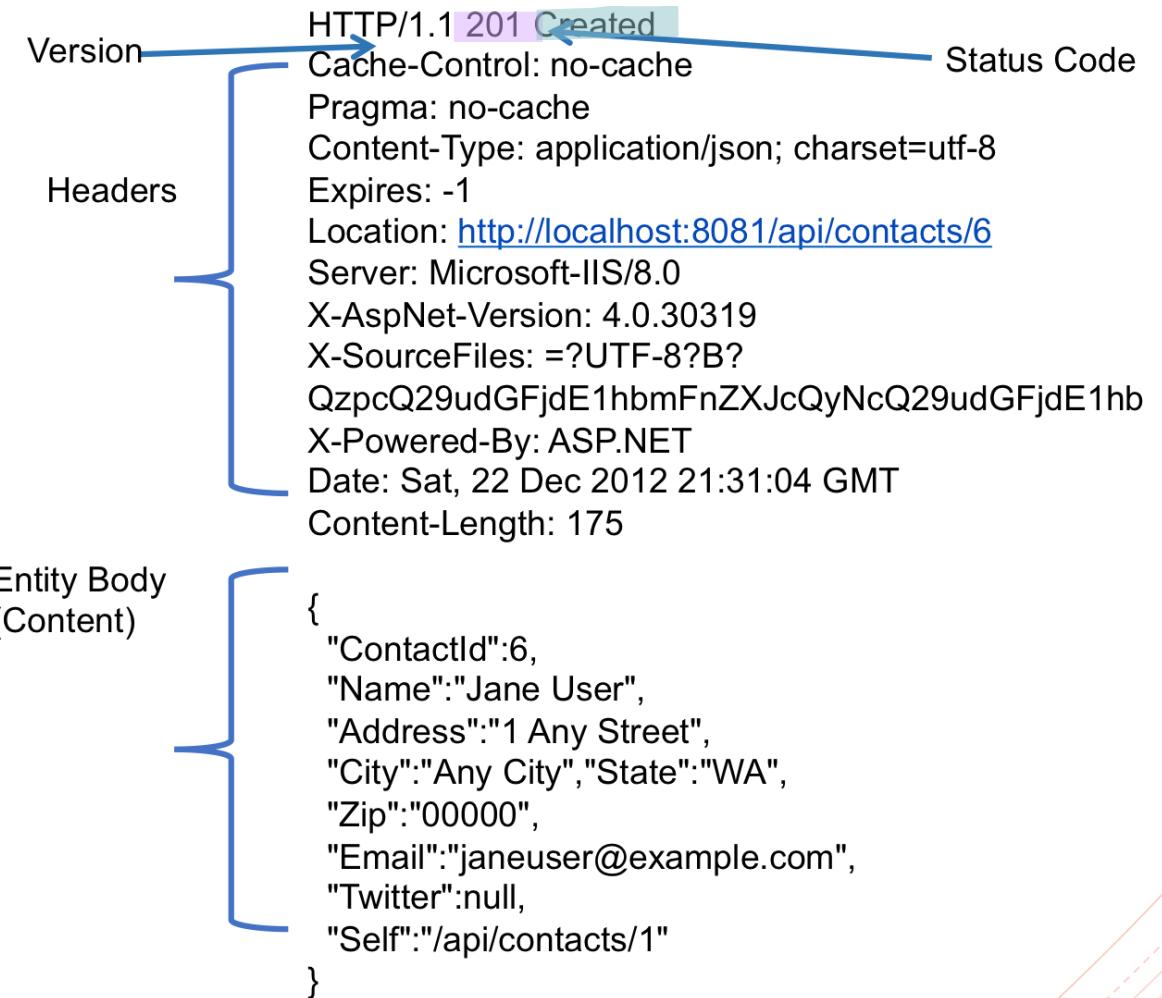
HTTP Responses

COSA SONO LE RISPOSTE HTTP

- Sono messaggi inviati dal server in risposta ad una richiesta inviata dal client.
- Hanno una struttura specifica e sono caratterizzate dal codice di risposta o status code che permette di comprendere l'esito della richiesta.

STRUTTURA DELLA RISPOSTA HTTP

- **Status code** della risposta
- **Status text:** versione human-readable dello status code
- Zero o più **headers**
- Un **body** (opzionale) che contiene i dati associati alla nostra richiesta.



STATUS CODE HTTP

- I codici delle risposte HTTP permettono al client di capire se una specifica richiesta HTTP è andata a buon fine.
- Sapere quali codici di risposta possono essere potenzialmente ricevuti permette di istruire l'applicazione per comportarsi in modi specifici a seconda del codice ricevuto.
- Gli status code sono codici a 3 cifre. La prima cifra permette di capire a che categoria appartiene lo status code:
 - 1xx (informativo): la richiesta è stata ricevuta, elaborazione in corso
 - 2xx (successo): la richiesta è stata ricevuta, compresa ed accettata con successo
 - 3xx (redirezione): per completare la richiesta servono altre azioni
 - 4xx (errore del client): la richiesta non può essere soddisfatta
 - 5xx (errore del server): il server non è riuscito a soddisfare una richiesta apparentemente valida.

Successful Requests

200 OK

La richiesta è avvenuta con successo. Indica un successo generico della richiesta. L'esatto significato di successo dipende dal metodo della richiesta, così come header e body della risposta. Il body di una 200 a seguito di una GET sarà la risorsa richiesta

201 Creato

La richiesta è andata a buon fine ed è stato creata una nuova risorsa il cui URI è di solito indicato nel body. Risposta tipica a seguito di una POST o PUT andata a buon fine

202 Accettato

La richiesta è stata ricevuta con successo, ma è ancora da elaborare.

203 Informazione non-autoritativa

La richiesta ha avuto successo ma le informazioni inviate al client circa la risposta vengono da un server di terze parti.

204 No content

La richiesta ha avuto successo ma non ci sono dati da restituire come risposta ma gli header potrebbero essere utili. IL client può ad esempio aggiornare gli header memorizzati nella cache per questa risorsa con quelle nuove.

205 Reset content

Il server richiede di resettare le informazioni inviate, come i dati del form

206 contenuto parziale

Risposta ad una richiesta solo per parte di un documento

Redirects

300 Scelte multiple

Il contenuto che è stato richiesto è stato spostato oppure ci sono più opzioni che corrispondono alla richiesta.

301 Spostato definitivamente

Il contenuto richiesto è stato spostato permanentemente e la risposta contiene la URI della nuova location. Importante quando si cambia il nome del dominio o gli URLs dei documenti esistenti.

302 Temporaneamente spostato

Il documento richiesto è stato temporaneamente spostato. L'URI della nuova location è inviato con la risposta. 303 e 307 sono versioni più specifiche di questo tipo.

303 Vedi altro (HTTP/1.1)

Il documento è stato trovato e la risposta contiene l'URI dove può essere trovato il documento

304 Non modificato

Il documento non è cambiato dall'ultima volta che è stato richiesto. Il client carica il documento dalla cache

305 Uso Proxy

IL documento richiesto può essere raggiunto solo tramite un proxy specifico

307 Redirect Temporanea (HTTP/1.1)

Il documento richiesto può essere temporaneamente trovato ad un URI differente che è passato nella risposta.

308 Redirect Permanente

La risorsa è stata spostata definitivamente in un altro URI che viene passato nella risposta. La differenza con 301 è che questa consente il cambio dal metodo POST a GET mentre il 308 no.

Errori del client

400 Bad Request

Il server non comprende la richiesta

401 Non autorizzato

UNAUTHORIZED: letteralmente non autorizzato ma semanticamente significa «non autenticato»: il client deve autenticarsi per ottenere la risposta richiesta.

403 Vietato

FORBIDDEN: il client non ha diritti di accesso al contenuto, ovvero non è autorizzato quindi il server si rifiuta di fornire la risorsa richiesta. A differenza del 401, l'identità del client è nota al server.

404 Not found

Il server non riesce a trovare la risorsa richiesta.

405 Metodo non consentito

Il metodo della richiesta non è consentito per la risorsa specificata

406 Non accettabile

La risorsa richiesta non può essere inviata in un modo che il client può comprendere

407 Richiesta autenticazione proxy

Il client deve essere autorizzato dal proxy prima che la risorsa richiesta possa essere spedita.

408 Request Timeout

Il tempo per la richiesta eccede il tempo per il quale il server è impostato per aspettare la richiesta

409 Conflitto

Il documento richiesto non può essere inviato a causa di un conflitto nella richiesta

Errori del client

410 Gone

Simile a 404 ma con motivazione diversa. In questo caso il contenuto richiesto è stato definitivamente eliminato dal server. Il client dovrebbe rimuovere le proprie cache ed i link alla risorsa. Importante per le strategie SEO perché indica ai crawler che quella risorsa può essere rimossa dalla SERP

411 Lunghezza richiesta

Richiesta rifiutata perché la lunghezza del contenuto deve essere specificata dal client.

412 Precondition Failed

Almeno una condizione della richiesta ha fallito

413 Request Entity Too Large

La richiesta è più grande di quanto il server possa gestire..

414 Request URI too long

L'URI era più lunga di quanto il server potesse gestire.

415 Unsupported Media Type

IL formato di almeno parte della richiesta non è supportato

416 Requested Range Not Satisfiable

La richiesta non può essere soddisfatta. Può accadere che la richiesta del client sia parte di un documento che non esiste

417 Expectation Failed

Il server non può soddisfare i requisiti inviati nel campo «Expect» dell'header

Errori del server

500 Internal Server Error

Errore generico, indica che qualcosa non funziona ma non può essere inviato niente di più specifico.

501 Not Implemented

Il server non supporta ciò che serve per restituire la risposta

502 Bad Gateway

Il server che funge da gateway o proxy ha ricevuto una risposta da un server a monte ritenuta non valida.

503 Service Unavailable

Il server non è attualmente disponibile a causa di un carico eccessivo, manutenzione o altre situazioni temporanee

504 Gateway Timeout

IL server che funge da gateway o proxy non ha ricevuto risposte nel tempo prestabilito

505 HTTP Version Not Supported

Il server non supporta la versione HTTP usata dal client per la request.

1 - CREARE RESPONSES

1.a – STRINGHE ED ARRAYS

- Tutte le routes ed i controllers dovrebbero ritornare una risposta da inviare al browser dell'utente.

La risposta base consiste nel ritornare una stringa da una route o da un controller. Laravel convertirà automaticamente la stringa in una risposta HTTP completa.

```
Route::get('/', function () {  
    return 'Hello World';  
});
```

- Se si ritorna un array, Laravel lo convertirà automaticamente in una risposta JSON. Anche le collections Eloquent possono essere ritornate ed automaticamente convertite in JSON.

```
Route::get('/', function () {  
    return [1, 2, 3];  
});
```

1.b RESPONSE OBJECTS

- Tipicamente, come risposte, ritorneremo istanze Illuminate\Http\Response o views.
- Ritornare una Response completa consente di customizzare lo status code e gli headers.
- Un'istanza di Response eredita dalla classe Symfony\Component\HttpFoundation\Response che fornisce diversi metodi per costruire le risposte HTTP

```
Route::get('/home', function () {
    return response('Hello World', 200)
        ->header('Content-Type', 'text/plain');
});
```

1.c MODELLI E COLLECTIONS ELOQUENT

- Si può anche ritornare modelli e collections Eloquent direttamente dalle rotte e dai controller. Quando capita, Laravel converte automaticamente i modelli e le collections in risposte JSON rispettando gli hidden attributes del modello.

```
use App\Models\User;

Route::get('/user/{user}', function (User $user) {
    return $user;
});
```

2 - REDIRECTS

- Le risposte redirect sono istanze della classe Illuminate\Http\RedirectResponse e contengono gli specifici header necessari al redirect dell'utente verso un altro url.
- Ci sono diversi modi di generare una RedirectResponse. Il modo più semplice è usare l'helper globale redirect:

```
Route::get('/dashboard', function () {  
    return redirect('home/dashboard');  
});
```

- Per reindirizzare l'utente alla location precedente, come ad un form invalido, si può usare l'helper back.
- Siccome questa caratteristica di Laravel utilizza le sessioni, assicurarsi che la rotta che chiama la funzione back stia usando il web middleware group.

```
Route::post('/user/profile', function () {  
    // Validate the request...  
  
    return back()->withInput();  
});
```

2.1 – REDIRECTS a NAMED ROUTES

- Quando si chiama l'helper redirect senza parametri, viene ritornata un'istanza della classe Illuminate\Routing\Redirector permettendo di chiamare qualsiasi metodo sull'istanza Redirector.
- Ad esempio, per creare un RedirectResponse ad una rotta con il nome, si può usare il metodo route:

```
return redirect()->route('login');
```

- SE la rotta ha dei parametri, questi possono essere passati come secondo argomento al metodo route.

```
// For a route with the following URI: /profile/{id}

return redirect()->route('profile', ['id' => 1]);
```

2.1.1–POPOLARE PARAMETRI VIA ELOQUENT MODELS

- Se si sta redirezionando verso una rotta con un parametro ID che è stato popolato da un model Eloquent si può passare il modello stesso. L'ID sarà estratto automaticamente.

```
// For a route with the following URI: /profile/{id}  
  
return redirect()->route('profile', [$user]);
```

- Per personalizzare il valore che si trova nel parametro della rotta, si può specificare la colonna nella definizione del parametro (/profile/({id:slug})) oppure si può sovrascrivere il metodo `getRouteKey` nel modello Eloquent.

```
/**  
 * Get the value of the model's route key.  
 */  
public function getRouteKey(): mixed  
{  
    return $this->slug;  
}
```

3- ALTRI TIPI DI RESPONSE

- L'helper response può essere usato per generare altri tipi di risposte. Quando l'helper è chiamato senza argomenti, viene ritornata un'implementazione del contract ResponseFactory. Questo contract fornisce diversi metodi utili per generare risposte.

3.1 VIEW RESPONSE

- Se si vuole il controllo sullo stato della risposta e gli headers ma si vuole anche ritornare una view come contenuto della risposta, si usa il metodo view:

```
return response()  
    ->view('hello', $data, 200)  
    ->header('Content-Type', $type);
```

Non c'è bisogno di passare uno status code o un header personalizzati, si può usare la funzione helper globale view

3.2 RISPOSTE JSON

- Il metodo json imposta automaticamente il content-type e application/json e converte l'array dato in un JSON usando la funzione php json_encode:

```
return response()->json([  
    'name' => 'Abigail',  
    'state' => 'CA',  
]);
```

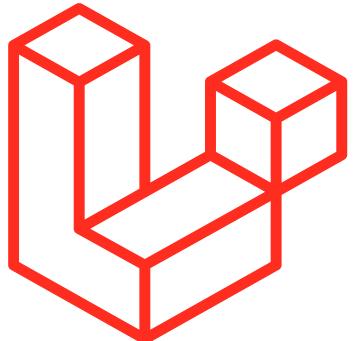
- Per creare una risposta JSONP si usa il metodo json in combinazione con il metodo withCallback

```
return response()  
    ->json(['name' => 'Abigail', 'state' => 'CA'])  
    ->withCallback($request->input('callback'));
```

3.3 FILE DOWNLOADS

- Il metodo download può essere utilizzato per generare una risposta che forza il browser dell'utente a scaricare il file ad un dato path.
- Il metodo accetta un filename come secondo argomento che sarà il nome visualizzato dall'utente che scarica il file.
- Come terzo argomento opzionale si può passare un array di headers HTTP

```
return response()->download($pathToFile);  
  
return response()->download($pathToFile, $name, $headers);
```



Laravel

LE BASI

VALIDATION

Laravel fornisce diversi approcci per **validare i dati in ingresso**.

Comunemente si usa il metodo **validate**, disponibile per tutte le richieste HTTP in entrata.

Laravel include anche una gran varietà di regole di validazione che si possono applicare ai dati, consente addirittura di validare se i valori sono unici all'interno di una determinata tabella del database.

Per conoscere le funzionalità di validazione, faremo un esempio completo di validazione di un form e di visualizzazione di messaggi di errore.

I. DEFINIRE LE ROTTE

Ipotizziamo di avere le seguenti rotte definite nel file `routes/web.php`

```
use App\Http\Controllers\PostController;

Route::get('/post/create', [PostController::class, 'create']);
Route::post('/post', [PostController::class, 'store']);
```

La **rotta GET** mostrerà un **form per la creazione di un nuovo post** da parte dell'utente.
La **rotta POST** **memorizzerà il nuovo post nel database**

II. CREARE IL CONTROLLER

Abbiamo al momento un semplice controller che gestisce le richieste a queste rotte. Per ora lasciamo il metodo `store` vuoto.

```
namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
use Illuminate\View\View;

class PostController extends Controller
{
    /**
     * Show the form to create a new blog post.
     */
    public function create(): View
    {
        return view('post.create');
    }

    /**
     * Store a new blog post.
     */
    public function store(Request $request): RedirectResponse
    {
        // Validate and store the blog post...

        $post = /** ... */

        return to_route('post.show', ['post' => $post->id]);
    }
}
```

III. SCRIVERE LE REGOLE DI VALIDAZIONE

Possiamo ora inserire nel metodo **store** la logica per validare il nuovo post.

Per farlo useremo il metodo **validate** fornito dall'oggetto `Illuminate\Http\Request`.

Se le regole di validazione passano, il codice continuerà ad essere eseguito normalmente; altrimenti, se la validazione fallisce, sarà lanciata **un'eccezione di validazione**

(`Illuminate\Validation\ValidationException`) ed l'opportuno **messaggio di errore** sarà mandato in risposta all'utente.

Se la validazione fallisce durante una normale richiesta HTTP, sarà generata una **risposta di redirect all'URL precedente**.

Se la richiesta in entrata è **una richiesta XHR**, sarà ritornata **una risposta JSON contenente il messaggio di errore di validazione**.

```
/**  
 * Store a new blog post.  
 */  
public function store(Request $request): RedirectResponse  
{  
    $validator = Validator::make($request->all(), [  
        'email' => ['required', 'email'],  
        'password' => ['required'],  
    ]);  
  
    // Se la validazione fallisce, ritorna un JSON di errore  
    if ($validator->fails()) {  
        return response()->json(['errors' => $validator->errors()]),  
400;  
    }  
    // Tentativo di autenticazione  
    // Esegui la validazione e ottieni i dati validati  
    $validatedData = $validator->validate();  
  
    // Tentativo di autenticazione  
    $credentials = [  
        'email' => $validatedData['email'],  
        'password' => $validatedData['password'],  
    ];  
}
```

Le regole di validazione sono passate dentro il metodo validate.
Possono essere specificate anche come **array di regole** invece che come una **stringa singola** delimitata da |

[Vedi elenco regole di validazione](#)

VISUALIZZARE L'ERRORE DI VALIDAZIONE

Se gli input non superano la validazione, Laravel **redirezionerà automaticamente l'utente all'URL precedente**. Inoltre, **tutti gli errori di validazione e gli input della request saranno memorizzati nella session**.

Una variabile **\$errors** è condivisa con tutte le views dell'applicazione attraverso il middleware `Illuminate\View\Middleware\ShareErrorsFromSession` che è fornito dal gruppo **middleware web**. La variabile **\$errors** sarà un'istanza di `Illuminate\Support\MessageBag`.

Nel nostro esempio, l'utente sarà redirezionato al metodo `create` del controller quando la validazione fallisce permettendoci di visualizzare il messaggio di errore nella view:

```
<!-- /resources/views/post/create.blade.php -->

<h1>Create Post</h1>

@if ($errors->any())
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif

<!-- Create Post Form -->
```

PERSONALIZZARE L'ERRORE DI VALIDAZIONE

Ogni regola di validazione di Laravel ha un **messaggio di errore** che si trova nel file **lang/en/validation.php**.

Se l'applicazione non ha una directory lang, questa può essere creata con il comando **lang:publish**

All'interno del file lang/en/validation.php si trova un messaggio per ogni regola di validazione. Questi messaggi possono essere modificati in base alle esigenze dell'applicazione.

Inoltre si può copiare questo file in un'altra directory language per tradurre i messaggi nella lingua dell'applicazione.

```
return [];

/*
| -----
| Validation Language Lines
| -----
|
| The following language lines contain the default error messages used by
| the validator class. Some of these rules have multiple versions such
| as the size rules. Feel free to tweak each of these messages here.
|
*/

'accepted' => 'The :attribute must be accepted.',
'accepted_if' => 'The :attribute must be accepted when :other is :value.',
'active_url' => 'The :attribute is not a valid URL.',
'after' => 'The :attribute must be a date after :date.',
'after_or_equal' => 'The :attribute must be a date after or equal to :date.',
'alpha' => 'The :attribute must only contain letters.',
'alpha_dash' => 'The :attribute must only contain letters, numbers, dashes and underscores.',
'alpha_num' => 'The :attribute must only contain letters and numbers.',
'array' => 'The :attribute must be an array.',
'before' => 'The :attribute must be a date before :date.',
'before_or_equal' => 'The :attribute must be a date before or equal to :date.',
'between' => [
    'numeric' => 'The :attribute must be between :min and :max.',
    'file' => 'The :attribute must be between :min and :max kilobytes.',
    'string' => 'The :attribute must be between :min and :max characters.',
    'array' => 'The :attribute must have between :min and :max items.',
],
```

Porzione del file lang/en/validation.php

RICHIESTE XHR (XML HttpRequest) E VALIDAZIONE

Molte applicazioni ricevono richieste XHR da un frontend JavaScript.

Quando si usa il metodo validate durante una richiesta XHR, Laravel non genererà una redirect response bensì una **risposta JSON che contiene tutti gli errori di validazione**. La risposta JSON sarà inviata con uno **status code 422**.

IV.C – LA DIRETTIVA @error

Si può usare la direttiva Blade **@error** per determinare rapidamente **se un messaggio di errore di validazione esiste per un dato attributo**. All'interno di una direttiva @error si può fare l'echo della variabile **\$message** per visualizzare il messaggio di errore.

Se si sta usando una named error bag, basterà passare il **nome della bag** come secondo argomento della direttiva @error

```
<!-- /resources/views/post/create.blade.php -->

<label for="title">Post Title</label>

<input id="title"
       type="text"
       name="title"
       class="@error('title') is-invalid @enderror">

@error('title')
    <div class="alert alert-danger">{{ $message }}</div>
@enderror
```

```
<input ... class="@error('title', 'post') is-invalid @enderror">
```

V. RIPOPOLARE I FORM

Quando Laravel genera un redirect di risposta a causa di un errore di validazione, il framework memorizza automaticamente tutti gli input della request nella session.

In questo modo si può facilmente **avere accesso agli input** nel corso della request successiva e ripopolare il form che l'utente ha cercato di inviare.

Per recuperare gli input salvati dalla request precedente, si invoca il metodo **old** su un'istanza di **Illuminate\Http\Request**.

Il metodo old estrarrà i dati di input precedenti dalla session

```
$title = $request->old('title');
```

Laravel fornisce anche un **helper globale old**.

Se non esiste un vecchio input per il campo in questione, sarà ritornato **null**:

```
<input type="text" name="title" value="{{ old('title') }}>
```

VI. NOTE SUI CAMPI OPZIONALI

Di default; Laravel include i middleware

TrimStrings e **ConvertEmptyStringsToNull** che sono elencati nello stack dalla classe App\Http\Kernel.

Per questo motivo si avrà spesso la necessità di **indicare i campi opzionali della request come nullable** se non si vuole che i valori null vengano dichiarati invalidi.

```
$request->validate([
    'title' => 'required|unique:posts|max:255',
    'body' => 'required',
    'publish_at' => 'nullable|date',
]);
```

Stiamo specificando che il campo publish_at può essere null.

VII. FORMATO DI RISPOSTA DELL'ERRORE DI VALIDAZIONE

Quando l'applicazione lancia un'eccezione ValidationException e la richiesta HTTP in entrata si aspetta una risposta JSON, Laravel formatterà automaticamente i messaggi di errore e ritornerà una risposta HTTP 422 – Unprocessable Entity.

```
{
  "message": "The team name must be a string. (and 4 more errors)",
  "errors": {
    "team_name": [
      "The team name must be a string.",
      "The team name must be at least 1 characters."
    ],
    "authorization.role": [
      "The selected authorization.role is invalid."
    ],
    "users.0.email": [
      "The users.0.email field is required."
    ],
    "users.2.email": [
      "The users.2.email must be a valid email address."
    ]
  }
}
```

FORM REQUEST VALIDATION

1-Creare Form Request

Per validazioni più complesse si potrebbe voler creare un «form request».

COSA SONO LE FORM REQUEST?

Sono classi request custom che encapsulano le loro proprie logiche di validazione ed autorizzazione.

Per creare una form request si usa il comando **Artisan make:request**

```
php artisan make:request StorePostRequest
```

La classe form request creata sarà posizionata nella directory **app/http/Requests**. Se questa directory non esiste, viene creata quando si lancia il comando `make:request`

Ogni form request generata da Laravel ha **2 metodi: authorize e rules**.

- **Metodo authorize:** si occupa di determinare se l'utente attualmente autenticato può svolgere le azioni presenti nella request
- **Metodo rules:** ritorna le regole di validazione che dovrebbero essere applicate ai dati della request.

```
/**  
 * Get the validation rules that apply to the request.  
 *  
 * @return array<string,  
 \Illuminate\Contracts\Validation\Rule|array|string>  
 */  
public function rules(): array  
{  
    return [  
        'title' => 'required|unique:posts|max:255',  
        'body' => 'required',  
    ];  
}
```

PERSONALIZZARE LA LOCATION DEL REDIRECT

La proprietà `$redirect` della form request permette di personalizzare il redirect conseguente ad un fallimento nella validazione.

```
/**  
 * The URI that users should be redirected to if  
 validation fails.  
 *  
 * @var string  
 */  
protected $redirect = '/dashboard';
```

Per personalizzare una named route si usa invece la proprietà `$redirectRoute`

```
/**  
 * The route that users should be redirected to if  
 validation fails.  
 *  
 * @var string  
 */  
protected $redirectRoute = 'dashboard';
```

2 – Autorizzare le Form Requests

Attraverso il metodo `authorize` della classe `FormRequest` si può determinare su un utente già autenticato ha effettivamente l'autorità di aggiornare una determinata risorsa. Ad esempio, si può determinare se un commento ad un post appartiene all'utente e se questo ha effettivamente il permesso di modificarlo.

All'interno di questo metodo si interagirà con le policies ed i gates di autorizzazione:

```
use App\Models\Comment;  
  
/**  
 * Determine if the user is authorized to make this request.  
 */  
public function authorize(): bool  
{  
    $comment = Comment::find($this->route('comment'));  
  
    return $comment && $this->user()->can('update', $comment);  
}
```

Tutte le form requests estendono la classe Request di Laravel; per questo motivo è possibile usare il metodo **user** per accedere all'attuale utente autenticato.

Il metodo **route** permette invece di accedere ai parametri dell'URI definiti sulla rotta che viene richiamata.

```
Route::post('/comment/{comment}');
```

Inoltre, l'utilizzo del route model binding permette di accedere al model come proprietà della request.

```
return $this->user()->can('update', $this->comment);
```

Se il metodo **authorize** ritorna false, si avrà una risposta HTTP con status code 403 ed il metodo controller non sarà eseguito.

Se si desidera gestire la logica delle autorizzazioni in un'altra parte dell'applicazione, basta ritornare true dal metodo **authorize**:

```
/**  
 * Determine if the user is authorized to make this request.  
 */  
public function authorize(): bool  
{  
    return true;  
}
```

3 – Personalizzare i messaggi di errore

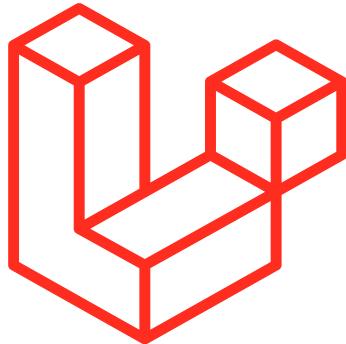
Per personalizzare i messaggi di errore di validazione si dovrà sovrascrivere il metodo **messages** che ritorna un array di coppie attributo/regola e i loro messaggi di errore corrispondenti.

```
/**  
 * Get the error messages for the defined validation rules.  
 *  
 * @return array<string, string>  
 */  
public function messages(): array  
{  
    return [  
        'title.required' => 'A title is required',  
        'body.required' => 'A message is required',  
    ];  
}
```

PERSONALIZZARE GLI ATTRIBUTI DI VALIDAZIONE

Molti dei messaggi di errore di validazione nativi di Laravel possiedono un placeholder :attribute. Se lo si vuole sostituire con un nome di attributo personalizzato, si specifica il nome desiderato sovrascrivendo il metodo attributes. Questo metodo ritorna un array di coppie attributo/nome.

```
/**
 * Get custom attributes for validator errors.
 *
 * @return array<string, string>
 */
public function attributes(): array
{
    return [
        'email' => 'email address',
    ];
}
```



Laravel

LE BASI

GESTIONE DEGLI ERRORI

- Quando si inizia un nuovo progetto Laravel, la gestione di errori ed exceptions è già configurata.
- Tutte le exceptions lanciate dall'applicazione si trovano nella classe `App\Exception\Handler`.

1 - CONFIGURAZIONE

- L'opzione `debug` all'interno del file di configurazione `config/app.php` determina quante informazioni su un errore sono mostrare all'utente. Di default rispetta il valore della variabile d'ambiente `APP_DEBUG` che è all'interno del file `.env`
- **Durante lo sviluppo in locale, la variabile d'ambiente APP_DEBUG dovrebbe essere impostata su true.**
- **Nell'ambiente di produzione dovrebbe essere settata su false.** Se fosse su true si rischierebbe di esporre valori di configurazione importanti e sensibili agli utenti finali.

2 – GESTORE DELLE EXCEPTIONS

2.1 – REPORT DELLE EXCEPTIONS

- Tutte le exceptions sono gestite dalla classe `App\Exceptions\Handler` la quale contiene un metodo `register` dove si possono registrare callbacks personalizzate per il report delle exceptions e il loro rendering.
- Il REPORT DELLE EXCEPTIONS è usato per registrare exceptions o per inviarle ad un servizio esterno come Flare o Bugsnag.
- Per riportare tipi diversi di exceptions in modi differenti, si può ricorrere al metodo `reportable` per registrare una closure che sarà eseguita quando un'exception di un dato tipo ha bisogno di essere segnalata. Laravel determinerà quale tipo di eccezione è riportata dalla closure esaminando il type-hint della closure stessa.

```
use App\Exceptions\InvalidOrderException;

/**
 * Register the exception handling callbacks for the application.
 */
public function register(): void
{
    $this->reportable(function (InvalidOrderException $e) {
        // ...
    });
}
```

- Quando si registra una callback per un'exception personalizzata usando il metodo `reportable`, Laravel registrerà l'exception quando la configurazione di logging di default. Se si vuole terminare la propagazione dell'exception allo stack di logging di default, si usa il metodo `stop` quando si definisce la callback di `report` o si ritorna false dalla callback stessa

```
$this->reportable(function (InvalidOrderException $e) {
    // ...
})->stop();

$this->reportable(function (InvalidOrderException $e) {
    return false;
});
```

3 – HTTP EXCEPTIONS

Alcune exceptions descrivono codici di errore HTTP dal server. Si può avere ad esempio un «page not found» error (404).

Per generare una risposta da qualsiasi punto dell'applicazione, si può usare l'helper abort

```
abort(404);
```

3.1 – PAGINE PERSONALIZZATE PER ERRORI HTTP

- Visualizzare pagine di error personalizzate per i vari HTTP status code è molto semplice in Laravel.
- Ad esempio, per personalizzare la pagina di errore 404, basta creare un template resources/views/errors/404.blade.php
- La view sarà visualizzata per tutti gli errori 404 generati dall'applicazione.

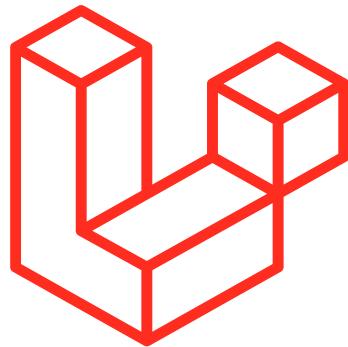
- Il nome del file della view deve corrispondere allo status code HTTP.
- L'istanza `HttpException` creata dalla funzione `abort` sarà passata alla view come variabile `$exception`

```
<h2>{{ $exception->getMessage() }}</h2>
```

- Si può pubblicare la pagina di errore predefinita di Laravel usando il comando Artisan `vendor:publish`. Una volta pubblicati i templates, essi possono essere customizzati a piacimento

```
php artisan vendor:publish --tag=laravel-errors
```

- Per definire **error pages di «fallback»** per un dato status code si definiscono un template `4xx.blade.php` ed un template `5xx.blade.php` della cartella `resources/views/errors`.
- I templates di fallback saranno visualizzati se non esiste una pagina specifica per lo status code specifico.



Laravel

MIGRATIONS

- Una migrazione è un processo che permette di creare o modificare lo schema di una tabella.
- Le migrations possono essere «versionate» rendendo semplici operazioni di rollback e di deploy.
- Laravel offre pieno supporto alle migrations attraverso la facade Schema e vari comandi Artisan per la creare o modificare migrazioni.
- Le migrazioni create si trovano nella cartella database\migration. Ogni filename delle migrazioni contiene un timestamp che consente di ordinare le migrazioni.

GENERARE MIGRATIONS

```
php artisan make:migration create_flights_table
```

```
INFO Migration  
[database/migrations/2023_01_23_193847_create_pos  
ts_table.php] created successfully.
```

- Il nome indicato per la migrazione (create_flights_table) sarà usato da Laravel per cercare di capire il nome della tabella e se la migrazione sta creando una nuova tabella o no.
- Se Laravel è in grado di determinare il nome della tabella dal nome della migration, il file della migrazione sarà precompilato con la tabella specificata. Altrimenti si può specificare la tabella nel file della migrazione manualmente.

STRUTTURA DELLA MIGRATION

- Una classe Migration contiene due metodi: up e down.
- Metodo up: indica le modifiche da applicare quando la migrazione «va su». Le modifiche possono essere aggiunta di tabelle/ colonne/ indici, rimozioni o cambi.
- Metodo down: indica le modifiche da applicare quando si vuole annullare le modifiche apportate dal metodo up
- Nell'esempio a fianco viene creata la tabella flights
- Se la migration interagisce con altri database oltre a quello di default, si imposta la proprietà \$connection della migration

```
/*
 * The database connection that should be used by the migration.
 *
 * @var string
 */
protected $connection = 'pgsql';

/**
 * Run the migrations.
 */
public function up(): void
{
// ...
}
```

Migration file creato con il comando `php artisan make:migration create_flights_table`

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('airline');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     */
    public function down(): void
    {
        Schema::drop('flights');
    }
};
```

LANCIARE MIGRATIONS

- Per lanciare tutte le migrations in sospeso si usa il comando Artisan migrate

```
php artisan migrate
```

- Comando migrate:status per vedere le migrazioni lanciate fino a quel momento
- Per vedere le istruzioni SQL che saranno eseguite dalle migrations senza veramente eseguirle, si può aggiungere il flag --pretend al comando migrate

```
php artisan migrate --pretend
```

ROLLING BACK DELLE MIGRATIONS

- Per eseguire il roll back delle ultime migrazioni, si usa il comando Artisan rollback. Questo comando esegue il rollback dell'ultimo gruppo di migrazioni.

```
php artisan migrate:rollback
```

- L'opzione --step del comando rollback permette di tornare indietro di un dato numero di migrazioni.

```
php artisan migrate:rollback --step=5
```

- Specificando l'opzione --batch si può eseguire il rollback di un determinato gruppo di migrations. Il valore dell'opzione batch corrisponde al valore del batch all'interno della tabella migrations del database

```
php artisan migrate:rollback --batch=3
```

- Il comando migrate:reset eseguirà il roll back di tutte le migrazioni

```
php artisan migrate:reset
```

ROLL BACK E MIGRATE CON UN UNICO COMANDO

- Il comando migrate:refresh farà il roll back di tutte le migrazioni e poi eseguirà il comando migrate
- Questo comando recrea l'intero database

```
php artisan migrate:refresh
```

```
# Refresh the database and run all database seeds...
php artisan migrate:refresh --seed
```

- L'opzione `--step` eseguirà il roll back e migrerà nuovamente un limitato numero di migrations

```
php artisan migrate:refresh --step=5
```

TABELLE

CREARE TABELLE

- Per creare una nuova tabella del database si usa il metodo `create` sulla facade `Schema`. Il metodo accetta due argomenti:
 1. Il nome della tabella
 2. Una closure che riceve un oggetto `Blueprint` usato per definire la nuova tabella

CONTROLLARE L'ESISTENZA DI TABELLE/COLONNE

- Si usano i metodi `hasTable` e `hasColumn`

ELIMINARE TUTTE LE TABELLE E MIGRARE

- Il comando `migrate:fresh` eliminerà tutte le tabelle del database e poi eseguirà il comando `migrate`

```
php artisan migrate:fresh  
php artisan migrate:fresh --seed
```

```
use Illuminate\Database\Schema\Blueprint;  
use Illuminate\Support\Facades\Schema;  
  
Schema::create('users', function (Blueprint $table) {  
    $table->id();  
    $table->string('name');  
    $table->string('email');  
    $table->timestamps();  
});
```

```
if (Schema::hasTable('users')) {  
    // La tabella "users" esiste...  
}  
  
if (Schema::hasColumn('users', 'email')) {  
    // La tabella "users" esiste ed ha una Colonna "email"..."  
}
```

CONNESSIONE AL DATABASE E OPZIONI DELLA TABELLA

- Per effettuare un'operazione su un database che non è quello di default, si usa il metodo **connection**
- Le proprietà **charset** e **collation** sono usate per specificare il set di caratteri e la collation per la tabella che si sta creando
- Il metodo **temporary** indica che la tabella deve essere temporanea: sarà visibile alla sessione database della connessione corrente e sarà eliminata automaticamente quando la connessione viene chiusa
- Il metodo **comment** serve per aggiungere un commento alla tabella del database.

```
Schema::connection('sqlite')->create('users', function (Blueprint $table)  
{  
    $table->id();  
});
```

```
Schema::create('users', function (Blueprint $table) {  
    $table->charset = 'utf8mb4';  
    $table->collation = 'utf8mb4_unicode_ci';  
    // ...  
});
```

```
Schema::create('calculations', function (Blueprint $table)  
{  
    $table->temporary();  
});
```

```
Schema::create('calculations', function (Blueprint $table) {  
    $table->comment('Business calculations');  
    // ...  
});
```

AGGIORNARE TABELLE

- Il metodo `table` della facade `Schema` è usato per aggiornare tabelle esistenti.
Accetta due argomenti: il nome della tabella ed una closure che riceve un oggetto `Blueprint` usato per aggiungere colonne o indici

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('users', function (Blueprint $table) {
    $table->integer('votes');
});
```

RINOMINARE / ELIMINARE TABELLE

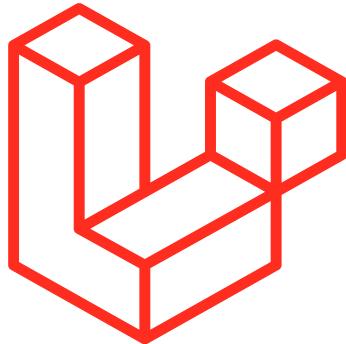
- Il metodo `rename` è usato per rinominare una tabella del database
- Per eliminare una tabella si usano i metodi `drop` o `dropIfExists`

```
use Illuminate\Support\Facades\Schema;

Schema::rename($from, $to);
```

```
Schema::drop('users');

Schema::dropIfExists('users');
```



Laravel

DATABASE

- Praticamente tutte le applicazioni web moderne interagiscono con un database.
- Laravel rende l'interazione con i database estremamente semplice grazie all'utilizzo di SQL, un query builder e l'ORM Eloquent.
- Laravel fornisce supporto per 5 database:
 - MariaDB 10.10+ ([Version Policy](#))
 - MySQL 5.7+ ([Version Policy](#))
 - PostgreSQL 11.0+ ([Version Policy](#))
 - SQLite 3.8.8+
 - SQL Server 2017+ ([Version Policy](#))

CONFIGURAZIONE

- La configurazione database di Laravel si trova nel file di configurazione config/database.php nel quale vengono definite le connessioni con i database e viene indicato quale connessione deve essere usata di default.

OPZIONE STICKY

- L'opzione sticky è un valore opzionale che può essere utilizzato per permettere la lettura immediata dei records che sono stati scritti durante il ciclo della request corrente.

1- ESEGUIRE QUERY SQL

Una volta configurato la connessione database, si possono lanciare le queries usando la facade DB che fornisce metodi per ogni tipo di query: select, update, insert, delete e statement.

QUERY SELECT

Per lanciare una query SELECT base si usa il metodo **select** sulla facade DB.

I vari metodi della facade DB accettano come primo argomento la query SQL da eseguire, il secondo è il bindings dei parametri.

Essi sono di solito i valori della clausola where.

Il binding dei parametri protegge dalla SQL injection.

Il metodo DB::select ritorna sempre un array di risultati, ciascuno dei quali è un oggetto stdClass e rappresenta un record del database.

Il metodo DB::scalar ritorna un singolo valore scalare, utile per le query «count».

```
namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\DB;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * Show a list of all of the application's users.
     */
    public function index(): View
    {
        $users = DB::select('select * from users where active = ?', [1]);
        $count = DB::scalar('select count(1) from posts where active =
:active', ['active' => 1]);

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

UTILIZZO DEL NAMED BINDINGS

Invece di usare ? per il binding dei parametri, si può eseguire la query usando il named binding

LANCIARE UN INSERT

Si usa il metodo DB::insert della facade DB. Come il metodo select, accetta una query SQL come primo argomento e i bindings come secondo

LANCIARE UN UPDATE

Si usa il metodo DB::update per aggiornare records già esistenti nel database. Ritorna il numero delle righe interessate dall'aggiornamento.

LANCIARE UN DELETE

Si usa il metodo DB::delete per eliminare records dal database. Ritorna il numero delle righe eliminate.

```
$results = DB::select('select * from users where id = :id', ['id' => 1]);
```

```
use Illuminate\Support\Facades\DB;  
  
DB::insert('insert into users (id, name) values (?, ?)', [1, 'Marc']);
```

```
use Illuminate\Support\Facades\DB;  
  
$affected = DB::update(  
    'update users set votes = 100 where name = ?',  
    ['Anita'])  
);
```

```
use Illuminate\Support\Facades\DB;  
  
$deleted = DB::delete('delete from users');
```

LANCIARE UNO STATEMENT GENERALE

Alcune istruzioni del database non ritornano alcun valore. Per questo tipo di operazioni si può usare il metodo DB::statement

```
DB::statement('drop table users');
```

LANCIARE UN'ISTRUZIONE UNPREPARED

Se si vuole eseguire un'istruzione SQL senza il binding dei valori, si può utilizzare il metodo DB::unprepared.

Siccome il binding dei parametri non viene eseguito, essi sono vulnerabili ad attacchi di SQL injection.

```
DB::unprepared('update users set votes = 100 where name = "Dries"');
```

ORM ELOQUENT

Object Relational Mapping

- L'ORM (Object Relational Mapping) è una **tecnica di programmazione** che permette di **mettere in relazione una tabella di un database relazionale con un oggetto**.
- Laravel utilizza l'ORM **Eloquent** che si basa sulla corrispondenza tra una tabella del database e una **classe di tipo “Model”**.
- Nello specifico, **una tabella sul database corrisponde ad una classe PHP** e **una singola riga della tabella corrisponde ad una istanza della classe**.
- L'utilizzo di Eloquent e dei Model semplifica molto l'interazione con il database, eliminando, in molti casi, la necessità di scrivere query dedicate.

ELOQUENT – FEATURES PRINCIPALI

1. MODEL DRIVEN: ogni tabella del database è associata ad un modello corrispondente. I modelli definiscono la **struttura** ed il **comportamento dei dati** e permettono di lavorare con i record come oggetti

2. CREAZIONE DI QUERY ESPRESSIVE: fornisce un costruttore di query espressivo che consente la **costruzione di query complesse utilizzando una sintassi semplice** e leggibile evitando SQL puro nella maggior parte dei casi

3. RELAZIONI: Eloquent consente la definire e lavorare con le relazioni del database in modo semplice. Le relazioni sono definite come **metodi del Model** facilitando il recupero dei dati.

4. CONVALIDA: Eloquent include un **supporto integrato per la convalida dei dati**, assicurando che solo i dati validi vengano inseriti nel database contribuendo all'integrità dei dati.

5. EVENTI ELOQUENT: è possibile collegare gli **event listeners ai models** per eseguire azioni prima o dopo determinati eventi del model, come il salvataggio, eliminazione o creazione di un record.

CREARE UN MODEL

- Prima di utilizzare Eloquent bisogna **impostare la connessione al database** nei file di configurazione di Laravel. Laravel supporta **connessioni multiple** al database, configurabili nel file config/database.php.
- Una volta configurata la connessione, si creano modelli per rappresentare le tabelle del database.
- Per collegare tabella e model in Eloquent, si utilizza una **convenzione sui rispettivi nomi**.
- La **classe del modello** è indicata al **singolare**
- La **tabella** corrispondente è indicata al **plurale**
- Comando Artisan per creare un nuovo model:
artisan make:model ModelName
- Il comando make:model supporta varie **opzioni** per creare classi collegate al model (controller, form....). Per creare un nuovo modello con annessa migration e, quindi, per creare automaticamente anche la nuova tabella si usa l'opzione **-m**, se si vuole aggiungere il controller **-c**, per indicare se deve essere una risorsa **-r** (CRUD methods)

```
$ php artisan make:model Post -m  
  
INFO Model [app/Models/Post.php] created  
successfully.
```

Viene creato un nuovo file model nella cartella **app**. In questo file si possono **definire le proprietà e le relazioni del modello**.

DEFINIRE PROPRIETA' E RELAZIONI

Esempio di model rappresentante una tabella «users»

```
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    // La tabella associata al model
    protected $table = 'users';

    // La primary key per il model.
    protected $primaryKey = 'id';

    // Gli attributi assegnabili in serie.
    protected $fillable = ['name', 'email', 'password'];

    // definire relazioni
    public function posts()
    {
        return $this->hasMany(Post::class);
    }
}
```

Nell'esempio specifichiamo il **nome della tabella**, la **primary key** e gli **attributi** assegnabili per il model User.

Inoltre definiamo una **relazione one-to-many** tra il model User e il model Post usando il metodo `hasMany`.

Il nostro modello User **estende la classe** astratta **Model di Eloquent** e da essa **eredita metodi** che permettono di interagire con i record del database senza dover scrivere righe di codice.

METODI DELLA CLASSE

Eloquent fornisce dunque metodi e funzionalità che semplificano le comuni operazioni sul database. I vari metodi statici del model restituiscono **istanze della classe** quando vengono recuperate **singole righe** (ad esempio con il metodo `find`) oppure istanze della classe `Illuminate\Database\Eloquent\Collection` nel caso di metodi che recuperano **array di righe**.

RECUPERARE RECORDS

Per recuperare records dal database, si possono usare i metodi del query builder di Eloquent.

Eloquent consente inoltre di concatenare metodi diversi per costruire query complesse in maniera semplice.

```
// Trovare uno user dall' ID  
$user = User::find(1) ;  
  
// Trovare users con uno specifico attributo. Get() Ritorna un array sul quale è possibile ciclare  
$users = User::where('status','active')->get();  
  
// Trovare user con uno specifico attributo. first() ritorna un solo record, il primo che soddisfa la condizione  
$JohnDoe = User::where('name', '=', 'John Doe')->first();  
  
// Recuperare tutti gli users  
$allUsers = User::all();  
  
// Recuperare users con i loro relativi posts  
$userWithPosts = User::with('posts')->find(1)
```

METODI DELLA CLASSE

RECUPERARE RECORDS PER PRIMARY KEY O LANCIARE EXCEPTION

Per lanciare exception se un record non viene trovato si usano i metodi `findOrFail()` o `firstOrFail()`. In questo modo sarà possibile eseguire il log e visualizzare una pagina di errore se necessario.

```
$model = User::findOrFail(1);  
  
$model = User::where('votes', '>', 100)->firstOrFail();
```

CREARE RECORDS

Per creare un nuovo record, si può usare il metodo `create()`

Esistono anche i metodi `firstOrCreate()` o `firstOrNew()` per creare records. Questi cercano un record con determinati attributi; se questo non viene trovato, ne verrà creato uno nel database o verrà istanziata una nuova istanza.

```
$user = User::create([  
    'name' => 'John Doe',  
    'email' => 'john@example.com',  
    'password' => bcrypt('password'),  
]);
```

METODI DELLA CLASSE

SALVARE UN MODEL

Per salvare un model si può utilizzare il metodo `create()`

```
$flight = Flight::create([  
    'name' => 'London to Paris',  
]);
```

MODIFICARE RECORDS

Per modificare un record, si può usare il metodo `save()`

Per modificare records multipli in una sola volta in base ad una condizione, si può usare il metodo `update()`

```
$user = User::find(1);  
$user->name = 'Updated Name';  
$user->save();
```

```
User::where('status', 'inactive')->update(['status' =>  
    'active']);
```

METODI DELLA CLASSE

CANCELLARE RECORDS

Per cancellare un record, si può usare il metodo **delete()** sull'istanza.

```
$user = User::find(1);
$user->delete();

//per cancellare records in base a
//determinate condizioni
User::where('status', 'inactive')->delete();
```

Per cancellare un modello esistente in base alla sua chiave primaria, si può usare il metodo **destroy()**

```
User::destroy(1);

User::destroy([1, 2, 3]);

User::destroy(1, 2, 3);
```

Si può lanciare una query di cancellazione su un **set di record**:

```
$affectedRows = User::where('votes', '>', 100)->delete();
```

METODI DELLA CLASSE

SALVARE UN MODEL

Per salvare un model si può utilizzare il metodo `create()`

```
$flight = Flight::create([  
    'name' => 'London to Paris',  
]);
```

MODIFICARE RECORDS

Per modificare un record, si può usare il metodo `save()`

Per modificare records multipli in una sola volta in base ad una condizione, si può usare il metodo `update()`

```
$user = User::find(1);  
$user->name = 'Updated Name';  
$user->save();
```

```
User::where('status', 'inactive')->update(['status' =>  
    'active']);
```

ALTRE OPERAZIONI

MASS UPDATE

- Flight::where('active', 1)
- ->where('destination', 'San Diego')
- ->update(['delayed' => 1]);

updateOrCreate

- Flight::where('active', 1)
- ->where('destination', 'San Diego')
- ->update(['delayed' => 1]);

SOFT DELETE

Soft Deleting

```
<?php  
  
namespace App\Models;  
  
use Illuminate\Database\Eloquent\Model;  
use Illuminate\Database\Eloquent\SoftDeletes;  
  
class Flight extends Model  
{  
use SoftDeletes;  
}
```

```
if ($flight->trashed()) {  
// ...  
}
```

```
$flight->restore();
```

```
$flight->history()->restore();
```

```
$flight->forceDelete();
```

CAMPI DI DEFAULT DI UN MODELLO

- **\$table:** Questo campo specifica il nome della tabella nel database associata al modello. Se non viene specificato, Laravel assume che il nome della tabella sia il nome del modello al plurale (ad esempio, il modello User si aspetterà una tabella chiamata users)
- **\$fillable:** Questo campo definisce quali attributi del modello possono essere assegnati in massa. Quando si utilizza il metodo create() o fill() per assegnare valori agli attributi del modello, solo gli attributi elencati in \$fillable verranno effettivamente a

```
protected $table = 'my_custom_table';
```

```
protected $fillable = ['name', 'email', 'password'];
```

CAMPI DI DEFAULT DI UN MODELLO

- `$primaryKey`: Questo campo specifica il nome della chiave primaria della tabella associata al modello. Di default, Laravel assume che la chiave primaria sia chiamata `id`, ma è possibile specificare un nome di chiave primaria personalizzato assegnando il nome de

```
protected $primaryKey = 'user_id';
```

LOCAL SCOPES

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Scope a query to only include popular users.
     */
    public function scopePopular(Builder $query): void
    {
        $query->where('votes', '>', 100);
    }

    /**
     * Scope a query to only include active users.
     */
    public function scopeActive(Builder $query): void
    {
        $query->where('active', 1);
    }
}
```

MUTATORS E ACCESSORS

Gli accessors e i mutators consentono di trasformare e/o formattare gli attributi di Eloquent quando questi vengono recuperati o impostati sulle istanze del modello. È possibile, ad esempio, criptare un valore mentre viene memorizzato nel database e poi decriptarlo automaticamente quando vi si accede in un modello Eloquent.

DEFINIRE UN ACCESSOR

Un accessor **trasforma il valore di un attributo Eloquent** nel momento in cui questo viene recuperato.

Per definire un accessor, creare un metodo protetto per rappresentare l'attributo accessibile.

Il nome del metodo deve corrispondere alla rappresentazione "camel case" del vero attributo sottostante o della colonna del database.

Per esempio, definiamo un accessor per l'attributo `first_name`. L'accessor sarà richiamato automaticamente da Eloquent quando si tenterà di recuperare il valore dell'attributo `first_name_ucfirst`.

```
<?php  
  
namespace App\Models;  
  
use Illuminate\Database\Eloquent\Cast\Attribute;  
use Illuminate\Database\Eloquent\Model;  
  
class User extends Model  
{  
    /**  
     * Get the user's first name.  
     */  
    protected function getFirstNameUcfirstAttribute ()  
    {  
        return ucfirst($value);  
    }  
}
```

Tutti i metodi dell'accessor ritornano un'istanza dell'oggetto **Attribute** che definisce come l'attributo viene recuperato e, optionalmente, mutato.

Per accedere al valore dell'accessor, basta semplicemente accedere l'attributo **first_name** sull'istanza del modello.

```
use App\Models\User;

$user = User::find(1);

$firstName = $user->first_name;
```

DEFINIRE UN MUTATOR

I mutators sono funzioni che permettono di modificare il valore di un attributo quanto questo viene impostato all'esterno di un Model.

Per definire un mutator, bisogna fornire l'argomento `set` quando si definisce l'attributo. Questo mutator sarà chiamato automaticamente ogni volta che si tenta di impostare il valore dell'attributo.

La closure del mutator riceve il valore assegnato all'attributo permettendo di manipolarlo e di restituire il valore modificato.

```
use Illuminate\Database\Eloquent\Cast\Attribute;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Interact with the user's first name.
     */
    public function setFirstNameAttribute($value) {
        $this->attributes['firstName'] = strtolower($value);
    }

    public function getFirstNameAttribute($value) {
        return ucfirst($value);
    }
}
```

RELAZIONI

Le relazioni possono essere definite come connessioni tra tabelle e permettono di organizzare e strutturare i dati, migliorandone e semplificandone la gestione.

Nei database esistono 3 tipi di relazioni:

- **Uno a uno**: un record di una tabella è associato **ad uno, e uno solo**, di un'altra tabella. Ad esempio una persona ed il suo codice fiscale.
- **Uno a molti**: un record è associato **a più record** di un'altra tabella. Ad esempio uno scrittore ed i suoi libri.
- **Molti a molti**: **più record** di una tabella sono associati a più record di un'altra tabella. Ad esempio studenti e corsi ai quali sono iscritti.

Per definire relazioni con Eloquent e fare in modo che questo le riconosca automaticamente, è sufficiente seguire alcune **convenzioni nella creazione della foreign key** che associa le tabelle.

Supponiamo di avere due tabelle, books e authors, e di voler creare una relazione tra le due.

La colonna con la foreign key avrà come nome il **nome del modello al singolare** e minuscolo, **seguito dal suffisso _id**.

```
// in Schema::create('books' ...  
$table->foreignId('author_id');  
$table->foreign('author_id')->references('id')->on('authors');
```

RELAZIONI

Una volta creata la colonna con la foreign key, per terminare la relazione tra i due model, serve indicare nel model `App\Model\Book` la **relazione** che lo lega al model `App\Model\Author`. Nel caso dell'esempio, un libro appartiene ad un singolo autore.

Sarà così possibile scrivere una rotta nella quale sarà possibile, a partire dal libro recuperato dal database, estrarre le informazioni sul suo autore, grazie alla proprietà dinamica aggiunta da Eloquent.

Laravel ed Eloquent si occuperanno di costruire ed eseguire le query necessarie per recuperare l'autore del libro che stiamo chiedendo di visualizzare e restituiranno un'istanza del model collegato.

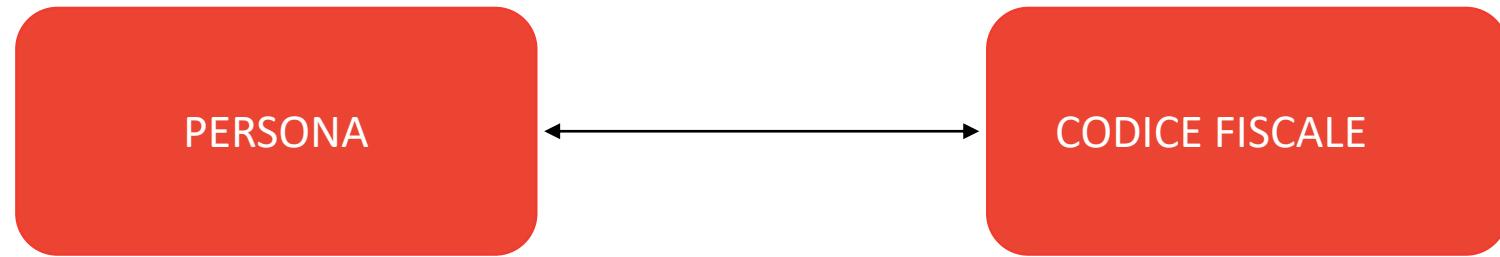
```
class Book extends Model
{
    public function author()
    {
        return $this->belongsTo(Author::class);
    }
}

class Author extends Model
{
    public function books()
    {
        return $this->hasMany(Book::class);
    }
}
```

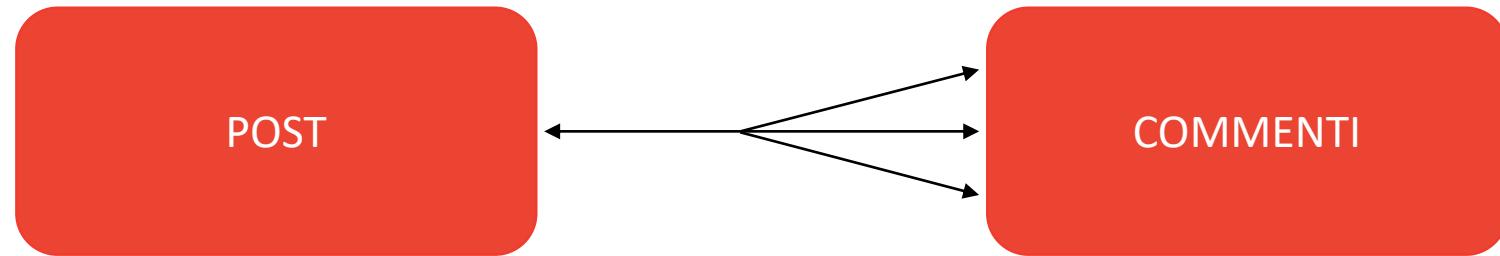
```
Route::get('/books/{book}', function (Book $book) {
    // property dinamica author aggiunta da Eloquent
    $author = $book->author;
    // posso accedere a tutte le property di Author
    return "$book->title - $author->first_name $author->last_name";
});
```

Quindi, alla base di ogni relazione utilizzabile con Eloquent vi sono due condizioni:

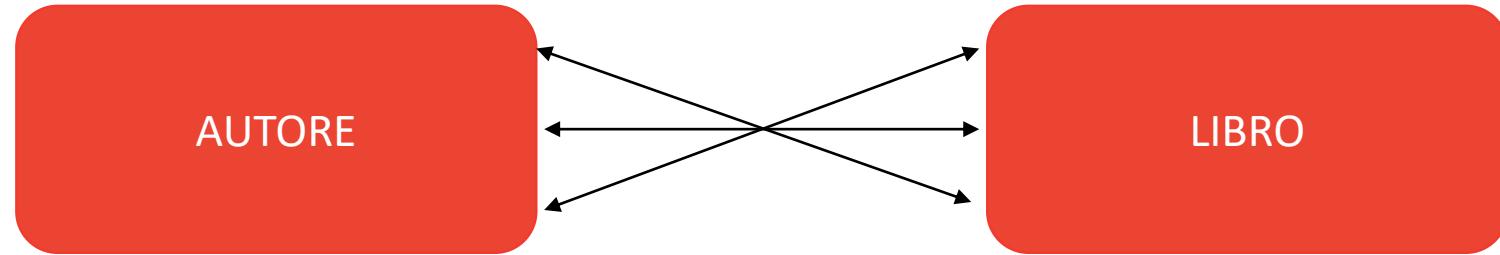
1. Rispetto delle convenzioni su nomi di tabelle e colonne, in particolare per la **foreign key**
2. Aver dichiarato i metodi di relazione nelle classi del model



one-to-one



one-to-many



many-to-many

RELAZIONE UNO A UNO – One to One

Associa due tavole in modo che una riga della prima tabella sia correlata ad **UNA SOLA RIGA** dell'altra tabella.

Ad esempio, un modello `User` può essere associato ad un modello `Phone`. Per definire questa relazione, assegniamo un metodo `phone` al modello `User`. Il metodo `phone` chiamerà il metodo `hasOne` e ritornerà il suo risultato.

Il primo argomento del metodo `hasOne` è il nome del modello collegato.

```
namespace App;  
use Illuminate\Database\Eloquent\Model;  
class User extends Model  
{  
    /** * Recupera il record phone associato allo user. */  
    public function phone() {  
        return $this->hasOne('App\Phone');  
    }  
}
```

Una volta che la relazione è definita, possiamo recuperare il record collegato usando le proprietà dinamiche di Eloquent che permettono di accedere ai metodi della relazione come se fossero proprietà del modello stesso.

```
$phone = User::find(1)->phone;
```

RELAZIONE UNO A UNO – One to One

Eloquent determina la **foreign key** della relazione basandosi sul nome del modello. Nel caso dell'esempio, si presuppone che il modello `Phone` abbia una foreign key `user_id`. Se si vuole sovrascrivere questa convenzione, si può passare un secondo argomento al metodo `hasOne`:

```
return $this->hasOne('App\Phone', 'foreign_key');
```

Inoltre, Eloquent suppone che la foreign key abbia un valore corrispondente nella colonna `id` (o la primary key personalizzata) del parent. In altre parole, Eloquent cercherà il valore della colonna `id` dello user nei record della colonna `user_id` della tabella `Phone`.

Se si desidera che la relazione usi un altro valore invece dell'`id`, si può passare un terzo argomento al metodo `hasOne` specificando la custom key.

```
return $this->hasOne('App\Phone', 'foreign_key', 'local_key');
```

RELAZIONE UNO A UNO – One to One

DEFINIRE LA RELAZIONE INVERSA

È possibile accedere all'utente che possiede un determinato telefono definendo il metodo `belongsTo` (inverso della relazione `hasOne`) nel `Phone` model.

```
namespace App;
use Illuminate\Database\Eloquent\Model;
class Phone extends Model
{
    /** * Get the user that owns the phone. */
    public function user() {
        return $this->belongsTo('App\User');
    }
}
```

Eloquent cercherà una corrispondenza tra il record `user_id` del modello `Phone` e l'`id` del modello `User`.

Eloquent determina il nome di default della foreign key esaminando il nome del metodo della relazione e aggiungendo il suffisso `_id`.

Come il metodo `hasOne`, anche `belongsTo` accetta il secondo ed il terzo parametro per customizzare il nome della key o per indicare che il join avviene su colonne diverse delle tabelle.

```
/** * Get the user that owns the phone. */
public function user() {
    return $this->belongsTo('App\User', 'foreign_key', 'other_key');
}
```

RELAZIONE UNO A MOLTI – One to Many

Definisce la relazione **tra un modello monoparentale e più modelli figli**: una riga della tabella A può essere legata a più righe della tabella B, ma ogni riga della tabella B è legata a una sola riga della tabella A.

Ad esempio, un modello `Post` può avere un infinito numero di commenti.

La relazione uno a molti è definita inserendo il metodo `hasMany` nel modello Eloquent

```
namespace App;
use Illuminate\Database\Eloquent\Model;
class Post extends Model
{
    /** * Get the comments for the blog post. */
    public function comments() {
        return $this->hasMany('App\Comment');
    }
}
```

Una volta che la relazione è stata definita, possiamo accedere alla collection dei commenti accedendo alla proprietà `comments`.

Per convenzione, Eloquent prenderà il nome «snake case» del modello padre ed aggiungerà il suffisso `_id`.
In questo caso Eloquent supporrà che la foreign key sul modello `Comment`, sarà `post_id`.

```
$comments = App\Post::find(1)->comments;
foreach ($comments as $comment) {
    //
}
```

RELAZIONE UNO A MOLTI – One to Many

Poiché tutte le relazioni fungono anche da generatori di query, si possono aggiungere condizioni per definire quali commenti verranno recuperati chiamando il metodo `comments`.

```
$comments = App\Post::find(1)->comments()->where('title', 'foo')->first();
```

Come il metodo `hasOne`, si possono sovrascrivere la `foreign` e la `local` keys passando argomenti aggiuntivi al metodo `hasMany`:

```
return $this->hasMany('App\Comment', 'foreign_key');
return $this->hasMany('App\Comment', 'foreign_key', 'local_key');
```

Per accedere da un commento al post parent, si definisce il modello `belongsTo` (inversa di `hasMany`) nel model del figlio .

```
namespace App;
use Illuminate\Database\Eloquent\Model;
class Comment extends Model
{
    /**
     * Get the post that owns the comment.
     */
    public function post() {
        return $this->belongsTo('App\Post');
    }
}
```

RELAZIONE UNO A MOLTI – One to Many

Una volta che la relazione è stata definita, possiamo recuperare il model Post per un Comment accedendo la proprietà dinamica post.

```
$comment = App\Comment::find(1);  
echo $comment->post->title;
```

In questo esempio, Eloquent cercherà la corrispondenza tra post_id del Comment e l'id del modello Post. Come in precedenza, Eloquent determinerà il nome predefinito della foreign key partendo dal nome del metodo della relazione e aggiungendo il suffisso _id.

Anche in questo caso, se la foreign key non è post_id, si può passare la custom key come secondo argomento al metodo belongsTo.

Allo stesso modo, se si desidera che il join tra le tabelle avvenga su un'altra colonna, si può passare il nome della colonna come terzo argomento.

```
/** * Get the post that owns the comment. */  
public function post()  
{  
    return $this->belongsTo('App\Post', 'foreign_key', 'other_key');  
}
```

RELAZIONE MOLTI A MOLTI – Many to Many

Nel modello autore-libro considerato all'inizio dell'argomento, abbiamo dato per scontato che un libro potesse essere scritto da un solo autore. Nella realtà, però, è possibile che uno stesso libro sia stato scritto da più autori.

In questo caso si avrà una relazione many-to-many poichè **una riga della tabella A può essere legata a più righe della tabella B e, allo stesso tempo, una riga della tabella B può essere collegata a più righe della tabella A.**

Le relazioni many-to-many sono solitamente implementate in un RDBMS utilizzando una terza tabella chiamata «**tabella associativa**» (o **join table** oppure **cross-reference-table**). Questa tabella consente di separare la relazione many-to-many in due diverse relazioni one-to-many.



Anche in Eloquent, l'implementazione di questa relazione richiede la presenza di tre tabelle: books, authors e tabella associativa author_book.

Per convenzione Eloquent richiede che il nome della tabella associativa sia formato dal **nome dei rispettivi modelli al singolare uniti da un _ ed in ordine alfabetico**. Nella tabella associativa devono inoltre essere presenti **le colonne per le due foreign key** che puntano a ciascuna tabella (author_id e book_id)

RELAZIONE MOLTI A MOLTI – Many to Many

A differenza di quanto accade per la relazione one-to-many, in questo caso non saranno presenti le colonne `foreign_key` nelle due tabelle `books` e `authors` poiché non vi è una relazione diretta tra le due.

Si avrà dunque:

author:
- `id = int`
- `name = string`

book:
- `id = int`
- `title = string`

author_book:
- `id = int`
- `author_id = foreign_key(author.id)`
- `book_id = foreign_key(book.id)`

```
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsToMany;

class Book extends Model
{
    public function author(): BelongsToMany
    {
        return $this->belongsToMany(Author::class);
    }
}
```

Una volta create le tabelle e le colonne con i giusti nomi tramite le migrations, si dovranno indicare le relazioni Eloquent nei relativi model. In questo caso avremo due relazioni `belongsToMany` per entrambi i model.

Il primo argomento passato al metodo è il nome del model collegato.

RELAZIONE MOLTI A MOLTI – Many to Many

Sarà poi possibile utilizzare le proprietà dinamiche di Laravel per accedere ai vari model e aggiungere condizioni alle query.

AUTHOR MODEL

```
use App\Models\Book;  
  
$book = Book::find(1);  
  
foreach ($book->authors as $author) {  
    // ...  
}
```

BOOK MODEL

```
use App\Models\Author;  
  
$author = Author::find(124);  
  
foreach ($author->books as $book) {  
    // ...  
}
```

CONDITIONS

```
$books = Author::find(141)->books()->orderBy('title')->get();
```

Abbiamo visto che esistono delle convenzioni per l'assegnazione del nome alla tabella intermedia e alle colonne delle chiavi. E' possibile però personalizzarli passando i nomi scelti come argomenti del metodo belongsToMany:

```
return $this->belongsToMany(Book::class, 'author_book', 'author_id', 'book_id');
```

SECONDO ARGOMENTO: nome personalizzato della tabella intermedia

TERZO ARGOMENTO: nome della foreign key del modello sul quale si sta definendo la relazione

QUARTO ARGOMENTO: nome della foreign key del modello collegato

RELAZIONE MOLTI A MOLTI – Many to Many

Sarà poi possibile utilizzare le proprietà dinamiche di Laravel per accedere ai vari model e aggiungere condizioni alle query.

AUTHOR MODEL

```
use App\Models\Book;  
  
$book = Book::find(1);  
  
foreach ($book->authors as $author) {  
    // ...  
}
```

BOOK MODEL

```
use App\Models\Author;  
  
$author = Author::find(124);  
  
foreach ($author->books as $book) {  
    // ...  
}
```

CONDITIONS

```
$books = Author::find(141)->books()->orderBy('title')->get();
```

Abbiamo visto che esistono delle convenzioni per l'assegnazione del nome alla tabella intermedia e alle colonne delle chiavi. E' possibile però personalizzarli passando i nomi scelti come argomenti del metodo belongsToMany:

```
return $this->belongsToMany(Book::class, 'author_book', 'author_id', 'book_id');
```

SECONDO ARGOMENTO: nome personalizzato della tabella intermedia

TERZO ARGOMENTO: nome della foreign key del modello sul quale si sta definendo la relazione

QUARTO ARGOMENTO: nome della foreign key del modello collegato

COLLECTIONS

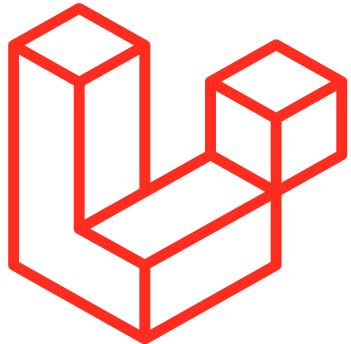
Tutti gli insiemi di risultati multipli sono istanze dell'oggetto [Illuminate\Database\Eloquent\Collection](#), compresi i risultati ottenuti con il metodo get o attraverso le relazioni.

L'oggetto collection di Eloquent estende la collection di base di Laravel, quindi eredita decine di metodi utilizzati per lavorare in modo fluido con l'array sottostante. Ovviamente tutte le collections servono come iteratori permettendo di eseguire cicli come se fossero normali arrays.

Le collections sono però molto più potenti degli array e disponono di una serie di operazioni di mappatura e riduzione che possono essere concatenate tra loro.

Potremmo, ad esempio, rimuovere tutti gli users inattivi e raccogliere il nome di ogni utente rimasto.

```
use App\Models\User;  
  
$users = User::where('active', 1)->get();  
  
foreach ($users as $user) {  
    echo $user->name;  
}
```



Laravel

QUERY BUILDER

- Il query builder di Laravel fornisce una comoda interfaccia per creare e lanciare queries. Funziona con tutti i database supportati da Laravel.
- Il query builder utilizza il binding dei parametri PDO. Non serve sanificare le stringhe passate al query builder come query bindings.

LANCIARE QUERIES

RECUPERARE TUTTE LE RIGHE DA UNA TABELLA

- Il metodo `table` della facade DB serve per iniziare una query. Ritorna un'istanza per la tabella indicata consentendo di concatenare più vincoli ed infine recuperare il risultato della query usando il metodo `get`.

```
class UserController extends Controller
{
    /**
     * Show a list of all of the application's users.
     */
    public function index(): View
    {
        $users = DB::table('users')->get();

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

Il metodo `get` ritorna una collection di risultati della query dove ogni risultato è un oggetto `stdClass`. Si può accedere al valore di ogni colonna accedendo alla colonna stessa come proprietà dell'oggetto:

```
use Illuminate\Support\Facades\DB;

$users = DB::table('users')->get();

foreach ($users as $user) {
    echo $user->name;
}
```

RECUPERARE UNA SINGOLA RIGA/COLONNA

- Si usa il metodo **first** della facade **DB** che ritorna un singolo oggetto **stdClass**

```
$user = DB::table('users')->where('name', 'John')->first();  
return $user->email;
```

- Il metodo **value** serve ad estrarre un singolo valore da un record. Ritorna direttamente il valore della colonna

```
$email = DB::table('users')->where('name', 'John')->value('email');
```

- Con il metodo **find** si recupera una singola riga attraverso il valore della sua colonna id

```
$user = DB::table('users')->find(3);
```

RECUPERARE UNA LISTA DI VALORI DA UNA COLONNA

- Il metodo **pluck** ritorna una Collection contenente i valori di una singola colonna.

```
use Illuminate\Support\Facades\DB;  
  
$titles = DB::table('users')->pluck('title');  
  
foreach ($titles as $title) {  
    echo $title;  
}
```

Si sta recuperando una collection di titles

- Come secondo argomento si può specificare la colonna che la collection dovrà usare come chiave.

```
$titles = DB::table('users')->pluck('title', 'name');  
  
foreach ($titles as $name => $title) {  
    echo $title;  
}
```

CHUNKING DEI RISULTATI

- Il metodo **chunk** della facade DB permette di «spezzettare» i risultati in piccole porzioni che, una alla volta, vengono inviate ad una closure per essere processate.

```
use Illuminate\Support\Collection;
use Illuminate\Support\Facades\DB;

DB::table('users')->orderBy('id')->chunk(100, function (Collection $users) {
    foreach ($users as $user) {
        // ...
    }
});
```

Viene recuperata la tabella users in tranches di 100 records alla volta

```
DB::table('users')->orderBy('id')->chunk(100, function (Collection $users)
{
    // Process the records...

    return false;
});
```

- Per fermare altre porzioni di risultati dall'essere processate, si ritorna false dalla closure

```
DB::table('users')->where('active', false)
->chunkById(100, function (Collection $users) {
    foreach ($users as $user) {
        DB::table('users')
            ->where('id', $user->id)
            ->update(['active' => true]);
    }
});
```

- Se si vuole eseguire l'update dei records recuperati mentre questi vengono suddivisi, è meglio usare il metodo **chunkById** che paginerà i risultati in base alla chiave primaria dei record

SELECT

- **SPECIFICARE UNA CLAUSOLA SELECT**

Con il metodo **select** si può specificare una select clause per la query

Il metodo **distinct** permette di forzare la query a riportare solo risultati distinti:

Per aggiungere una colonna alla select clause di una query già esistente, si usa il metodo **addSelect**

```
use Illuminate\Support\Facades\DB;  
  
$users = DB::table('users')  
    ->select('name', 'email as user_email')  
    ->get();
```

```
$users = DB::table('users')->distinct()->get();
```

```
$query = DB::table('users')->select('name');  
  
$users = $query->addSelect('age')->get();
```

JOINS

- **INNER JOIN**

Per un inner join basica, si usa il metodo **join** su un'istanza del query builder. Primo argomento è il nome della tabella alla quale ci si vuole unire, gli argomenti rimanenti specificano i vincoli delle colonne per la join. Si possono anche unire più tabelle in una singola query.

- **LEFT JOIN / RIGHT JOIN**

Si utilizzano i metodi **leftJoin** o **rightJoin**

- **CROSS JOIN**

La cross join genera un prodotto cartesiano tra la prima tabella e la tabella unita

- **CLAUSOLE JOIN AVANZATE**

Passando una closure come secondo argomento del metodo join si possono specificare clausole di join avanzate.

```
use Illuminate\Support\Facades\DB;

$users = DB::table('users')
    ->join('contacts', 'users.id', '=', 'contacts.user_id')
    ->join('orders', 'users.id', '=', 'orders.user_id')
    ->select('users.*', 'contacts.phone', 'orders.price')
    ->get();
```

```
$users = DB::table('users')
    ->leftJoin('posts', 'users.id', '=', 'posts.user_id')
    ->get();
```

```
$users = DB::table('users')
    ->rightJoin('posts', 'users.id', '=', 'posts.user_id')
    ->get();
```

```
$sizes = DB::table('sizes')
    ->crossJoin('colors')
    ->get();
```

```
DB::table('users')
    ->join('contacts', function (JoinClause $join) {
        $join->on('users.id', '=', 'contacts.user_id')->orOn(/* ... */);
    })
    ->get();
```

Si possono usare i metodi **where** e **orWhere** forniti dall'istanza della JoinClause per usare una clausola **where** sulle joins.

Invece di confrontare due colonne, questi metodi confronteranno una colonna ed un valore

```
DB::table('users')
    ->join('contacts', function (JoinClause $join) {
        $join->on('users.id', '=', 'contacts.user_id')
            ->where('contacts.user_id', '>', 5);
    })
    ->get();
```

- **SUBQUERY DELLE JOINS**

I metodi **joinSub**, **leftJoinSub** e **rightJoinSub** servono per unire una query ad una sottoquery.

Ognuno di questi metodi riceve tre argomenti: la sottoquery, l'alias della sua tabella ed una closure che definisce le colonne correlate.

```
$latestPosts = DB::table('posts')
    ->select('user_id', DB::raw('MAX(created_at) as last_post_created_at'))
    ->where('is_published', true)
    ->groupBy('user_id');

$users = DB::table('users')
    ->joinSub($latestPosts, 'latest_posts', function (JoinClause $join) {
        $join->on('users.id', '=', 'latest_posts.user_id');
    })
    ->get();
```

Recuperiamo una collection di utenti dove ogni record contiene anche un timestamp created_at del post più recente

UNIONI

Il metodo **union** permette di unire due o più query insieme.

Il metodo **unionAll** non rimuoverà i risultati duplicati formatisi dall'unione di due queries.

```
use Illuminate\Support\Facades\DB;

$first = DB::table('users')
    ->whereNull('first_name');

$users = DB::table('users')
    ->whereNull('last_name')
    ->union($first)
    ->get();
```

CLAUSOLA WHERE BASICA

La più basica chiamata al metodo del query builder **where** richiede tre argomenti:

- Il nome della colonna
- Un operatore (se omesso, Laravel userà l'operatore =)
- Il valore da confrontare con il valore della colonna

Si può usare qualsiasi operatore supportato dal database

È anche possibile passare al metodo where un array di condizioni. Ogni elemento deve essere un array contenente i tre argomenti passati al metodo where.

```
$users = DB::table('users')->where([
    ['status', '=', '1'],
    ['subscribed', '<>', '1'],
])->get();
```

```
$users = DB::table('users')
->where('votes', '=', 100)
->where('age', '>', 35)
->get();
```

La query recupera gli utenti che hanno voto uguale a 100 ed età superiore a 35 anni

```
$users = DB::table('users')
->where('votes', '>=', 100)
->get();
```

```
$users = DB::table('users')
->where('votes', '<>', 100)
->get();
```

```
$users = DB::table('users')
->where('name', 'like', 'T%')
->get();
```

```
$users = DB::table('users')
->where('votes', '>', 100)
->orWhere('name', 'John')
->get();
```

CLAUSOLA ORWHERE

Il metodo **orWhere** permette di unire una clausola alla query usando l'operatore or.
Accetta gli stessi argomenti del metodo where.

Si può raggruppare una condizione or tra parentesi **passando una closure come primo argomento del metodo orWhere**.

Questo esempio produrrà il seguente SQL:

```
select * from users where votes > 100 or (name = 'Abigail' and votes > 50)
```

```
$users = DB::table('users')
->where('votes', '>', 100)
->orWhere(function (Builder $query) {
    $query->where('name', 'Abigail')
        ->where('votes', '>', 50);
})
->get();
```

CLAUSOLA WHERE NOT

I metodi **WhereNot** e **orWhereNot** negano un dato gruppo di condizioni. Ad esempio, la query a fianco esclude i prodotti che sono in liquidazione e che hanno un prezzo inferiore a 10

```
$products = DB::table('products')
    ->whereNot(function (Builder $query) {
        $query->where('clearance', true)
            ->orWhere('price', '<', 10);
    })
    ->get();
```

LE CLAUSOLE WHERE JSON

Laravel supporta le query verso colonne di tipo JSON su database che le supportano. Al momento sono MySQL 5.7+, PostgreSQL, SQL Server 2016, and SQLite 3.39.0 (with the [JSON1 extension](#)).

Per rivolgere una query ad una Colonna JSON si usa l'operatore `->`

Per gli array JSON si usa `whereJsonContains`

```
$users = DB::table('users')
    ->where('preferences->dining->meal', 'salad')
    ->get();
```

```
$users = DB::table('users')
    ->whereJsonContains('options->languages', 'en')
    ->get();
```

Oppoore il metodo `whereJsonLength` per query di JSON array a seconda della loro lunghezza.

```
$users = DB::table('users')
    ->whereJsonLength('options->languages', 0)
    ->get();

$users = DB::table('users')
    ->whereJsonLength('options->languages', '>', 1)
    ->get();
```

CLAUSOLE WHERE AGGIUNTIVE

- **whereBetween / orWhereBetween** verificano che il valore di una colonna sia compreso tra due valori
- **whereNotBetween / orWhereNotBetween**: valore della colonna non compreso tra due valori
- **whereBetweenColumns / whereNotBetweenColumns / orWhereBetweenColumns / orWhereNotBetweenColumns** : il valore della colonna è o non è tra due valori di due colonne nella stessa tabella
- **whereIn / whereNotIn / orWhereIn / orWhereNotIn**: il valore di una colonna è (o non è) contenuto nell'array dato
- **whereNull / whereNotNull / orWhereNull / orWhereNotNull**: il valore della colonna è (o non è) null.
- **whereDate / whereMonth / whereDay / whereYear / whereTime**: confronta il valore di una Colonna con una data, mese, un giorno, un anno o un orario
- **whereColumn / orWhereColumn**: verifica se due colonne sono uguali. Si può specificare un operatore come secondo parametro oppure passare un array di comparatori.

```
$users = DB::table('users')
    ->whereBetween('votes', [1, 100])
    ->get();
```

```
$patients = DB::table('patients')
    ->whereBetweenColumns('weight', ['minimum_allowed_weight',
    'maximum_allowed_weight'])
    ->get();
```

```
$users = DB::table('users')
    ->whereIn('id', [1, 2, 3])
    ->get();
```

```
$users = DB::table('users')
    ->whereNull('updated_at')
    ->get();
```

```
$users = DB::table('users')
    ->whereDate('created_at', '2016-12-31')
    ->get();
```

```
$users = DB::table('users')
    ->whereColumn('first_name', 'last_name')
    ->get();
```

```
$users = DB::table('users')
    ->whereColumn([
        ['first_name', '=', 'last_name'],
        ['updated_at', '>', 'created_at'],
    ])->get();
```

CLAUSOLE WHERE AVANZATE

- **whereExists:** accetta una closure che riceve un'istanza del query builder che permette di definire la query che deve essere inserita nella clausola exists

```
$users = DB::table('users')
    ->whereExists(function (Builder $query) {
        $query->select(DB::raw(1))
            ->from('orders')
            ->whereColumn('orders.user_id', 'users.id');
    })
    ->get();
```

- In alternativa, si può passare al metodo whereExists un oggetto invece di una closure

```
$orders = DB::table('orders')
    ->select(DB::raw(1))
    ->whereColumn('orders.user_id', 'users.id');

$users = DB::table('users')
    ->whereExists($orders)
    ->get();
```

Il risultato di entrambi gli esempi riportati è il seguente SQL:

```
select * from users
where exists (
    select 1
    from orders
    where orders.user_id = users.id
)
```

- **Where clause di una subquery**

Se si avesse la necessità di costruire una where clause che confronta i risultati di una subquery ad un dato valore, si possono passare una closure ed un value al metodo where.

```
use App\Models\User;
use Illuminate\Database\Query\Builder;

$users = User::where(function (Builder $query) {
    $query->select('type')
        ->from('membership')
        ->whereColumn('membership.user_id', 'users.id')
        ->orderByDesc('membership.start_date')
        ->limit(1);
}, 'Pro')->get();
```

Recupera tutti gli utenti che hanno una recente membership di un certo tipo

Se, invece, si avesse la necessità di costruire una where clause che confronta una colonna con i risultati di una subquery, si possono passare una colonna, un operatore ed una closure al metodo where.

```
use App\Models\Income;
use Illuminate\Database\Query\Builder;

$incomes = Income::where('amount', '<', function (Builder $query) {
    $query->selectRaw('avg(i.amount)')->from('incomes as i');
})->get();
```

Recupera i records income dove l'importo è minore della media

ORDINARE, RAGGRUPPARE, LIMIT E OFFSET

ORDINARE

- **Metodo orderBy:** ordina i risultati in base ad una data colonna. Primo argomento è la colonna, secondo è la direzione dell'ordinamento che può essere asc o desc
- Per ordinare più colonne, si può semplicemente richiamare orderBy tante volte quante ne occorrono
- **Metodi latest e oldest:** ordina i risultati in base alla data. Di default il risultato sarà ordinato in base alla colonna created_at. Oppure si passa il nome della colonna che si desidera usare per l'ordinamento.
- **Ordine casuale:** Si usa il metodo inRandomOrder
- **Metodo reorder:** rimuove tutti gli orderBy che sono stati precedentemente applicati alla query.

```
$users = DB::table('users')  
    ->orderBy('name', 'desc')  
    ->get();
```

```
$users = DB::table('users')  
    ->orderBy('name', 'desc')  
    ->orderBy('email', 'asc')  
    ->get();
```

```
$user = DB::table('users')  
    ->latest()  
    ->first();
```

```
$randomUser = DB::table('users')  
    ->inRandomOrder()  
    ->first();
```

```
$query = DB::table('users')->orderBy('name');  
  
$unorderedUsers = $query->reorder()->get();
```

Passando una colonna ed una direzione al metodo reorder, si rimuovono tutti gli orderBy precedenti e si applica un nuovo ordinamento alla query.

```
$query = DB::table('users')->orderBy('name');  
  
$usersOrderedByEmail = $query->reorder('email', 'desc')->get();
```

RAGGRUPPARE I RISULTATI

- **Metodi groupBy e having:** il metodo **having** ha una signature simile al metodo where

```
$users = DB::table('users')
    ->groupBy('account_id')
    ->having('account_id', '>', 100)
    ->get();
```

- **Metodo havingBetween** Per filtrare i risultati all'interno di un certo range

```
$report = DB::table('orders')
    ->selectRaw('count(id) as number_of_orders, customer_id')
    ->groupBy('customer_id')
    ->havingBetween('number_of_orders', [5, 15])
    ->get();
```

- Passando più argomenti al metodo **groupBy**, si possono raggruppare i risultati per colonne multiple

```
$users = DB::table('users')
    ->groupBy('first_name', 'status')
    ->having('account_id', '>', 100)
    ->get();
```

- Per costruire istruzioni **having** più avanzate, vedere il metodo **havingRaw**

LIMIT E OFFSET

- **Metodi skip e take:** questi due metodi vengono usati per limitare il numero dei risultati di una query o per saltare un dato numero di risultati

```
$users = DB::table('users')->skip(10)->take(5)->get();
```

- **Metodi limit e offset:** si possono usare in alternativa ai metodi skip e take.

```
$users = DB::table('users')
    ->offset(10)
    ->limit(5)
    ->get();
```

ISTRUZIONE INSERT

- Il metodo **insert** è usato per inserire record nelle tabelle del database.
Accetta un array di nome di colonne e valori.

Si possono inserire più records alla volta passando un array di array.
Ogni array rappresenta un record che deve essere inserito in tabella.

- Il metodo **insertOrIgnore** ignorerà gli errori che si potrebbero verificare inserendo il record nel database.
- Il metodo **insertUsing** inserirà nuovi records nella tabella mentre una subquery per determinare i dati che devono essere inseriti

ID AUTO-INCREMENTALI

Se la tabella ha id auto-incrementali, usare il metodo `insertGetId` per inserire un record e recuperarne l'ID

```
DB::table('users')->insert([
    'email' => 'kayla@example.com',
    'votes' => 0
]);
```

```
DB::table('users')->insert([
    ['email' => 'picard@example.com', 'votes' => 0],
    ['email' => 'janeway@example.com', 'votes' => 0],
]);
```

```
DB::table('users')->insertOrIgnore([
    ['id' => 1, 'email' => 'sisko@example.com'],
    ['id' => 2, 'email' => 'archer@example.com'],
]);
```

```
DB::table('pruned_users')->insertUsing([
    'id', 'name', 'email', 'email_verified_at'
], DB::table('users')->select(
    'id', 'name', 'email', 'email_verified_at'
)->where('updated_at', '<=', now()-
>subMonth()));
```

```
$id = DB::table('users')->insertGetId(
    ['email' => 'john@example.com', 'votes' => 0]
);
```

ISTRUZIONE INSERT

- Il metodo **insert** è usato per inserire record nelle tabelle del database.
Accetta un array di nome di colonne e valori.

Si possono inserire più records alla volta passando un array di array.
Ogni array rappresenta un record che deve essere inserito in tabella.

- Il metodo **insertOrIgnore** ignorerà gli errori che si potrebbero verificare inserendo il record nel database.
- Il metodo **insertUsing** inserirà nuovi records nella tabella mentre una subquery per determinare i dati che devono essere inseriti

ID AUTO-INCREMENTALI

Se la tabella ha id auto-incrementali, usare il metodo `insertGetId` per inserire un record e recuperarne l'ID

```
DB::table('users')->insert([
    'email' => 'kayla@example.com',
    'votes' => 0
]);
```

```
DB::table('users')->insert([
    ['email' => 'picard@example.com', 'votes' => 0],
    ['email' => 'janeway@example.com', 'votes' => 0],
]);
```

```
DB::table('users')->insertOrIgnore([
    ['id' => 1, 'email' => 'sisko@example.com'],
    ['id' => 2, 'email' => 'archer@example.com'],
]);
```

```
DB::table('pruned_users')->insertUsing([
    'id', 'name', 'email', 'email_verified_at'
], DB::table('users')->select(
    'id', 'name', 'email', 'email_verified_at'
)->where('updated_at', '<=' , now()-
>subMonth()));
```

```
$id = DB::table('users')->insertGetId(
    ['email' => 'john@example.com', 'votes' => 0]
);
```

ISTRUZIONE DELETE

Il metodo **delete** elimina i record dal database. Si può vincolare il delete usando la clausola **where** prima di chiamare il metodo delete. Viene ritornato il numero di righe interessate dalla cancellazione

```
$deleted = DB::table('users')->delete();  
  
$deleted = DB::table('users')->where('votes', '>', 100)->delete();
```

Per svuotare un'intera tabella rimuovendo tutti i record e resettando a zero l'ID auto-increment, si usa il metodo **truncate**

```
DB::table('users')->truncate();
```

PESSIMISTIC LOCKING

Il pessimistic locking evita che la riga selezionata venga modificata prima che la transazione con il database sia completata.
Si usano i metodi **sharedLock** o **lockForUpdate**

```
DB::table('users')  
    ->where('votes', '>',  
    100)  
    ->sharedLock()  
    ->get();
```

```
DB::table('users')  
    ->where('votes', '>', 100)  
    ->lockForUpdate()  
    ->get();
```

Il lock «for update» evita che i record selezionati siano modificati o selezionati con un altro shared lock

DEBUGGING

Metodi dd e dump

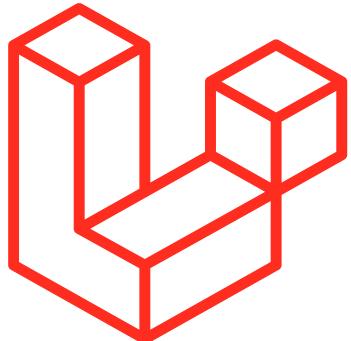
Utilizzati per eseguire il dump dei binding della query e dell'SQL corrente.
Il metodo **dd** visualizza le informazioni di debug e interrompe l'esecuzione della richiesta.
Il metodo **dump** visualizza le informazioni di debug ma continua l'esecuzione della richiesta

```
DB::table('users')->where('votes', '>', 100)->dd();  
  
DB::table('users')->where('votes', '>', 100)->dump();
```

Metodi dumpRawSql e ddRawSql

Utilizzati per eseguire il dump dell'SQL con tutti i bindings opportunamente sostituiti.

```
DB::table('users')->where('votes', '>', 100)->dumpRawSql();  
  
DB::table('users')->where('votes', '>', 100)->ddRawSql();
```



Laravel

LE BASI

VIEWS

- Sicuramente non è pratico ritornare intere stringhe di un documento HTML dalle rotte o dai controllers.
- Le views forniscono un sistema utile per dividere il nostro HTML in file separati.
- Separano la logica dei controller / applicazione dalla logica della presentazione e si trovano all'interno della cartella resources/views.
- Quando si usa Laravel, i templates delle views sono spesso scritti il linguaggio di templating Blade.

Esempio di una semplice view

```
<!-- View stored in resources/views/greeting.blade.php -->

<html>
    <body>
        <h1>Hello, {{ $name }}</h1>
    </body>
</html>
```

- Siccome la view dell'esempio si trova in resources/views/greeting.blade.php, possiamo ritornarla usando l'helper globale view in questo modo:

```
Route::get('/', function () {
    return view('greeting', ['name' => 'James']);
});
```

1

CREARE E VISUALIZZARE LE VIEWS

- Si può creare una view creando o inserendo un file .blade.php nella directory resources/views.
- Il formato .blade.php informa il framework che il file contiene un template Blade che, a sua volta, contiene HTML e direttive Blade che permettono, in maniera molto semplificata, di fare l'echo di valori, creare «if» statements, iterare sui dati e molto altro
- Una volta creata una view, si può ritornarla da una delle rotte dell'applicazione o da un controller usando l'helper globale view

```
Route::get('/', function () {
    return view('greeting', ['name' => 'James']);
});
```

- Le views possono anche essere ritornate usando la facade View

```
use Illuminate\Support\Facades\View;

return View::make('greeting', ['name' => 'James']);
```

Il primo argomento passato è il nome del file della view nella cartella resources/views. Il secondo argomento è un array di dati che dovrebbe essere visibile alla view. In questo caso stiamo passando alcune variabili name, che sono visualizzate nella view usando la sintassi Blade

1.1

DIRECTORIES DI VIEW ANNIDATE

1.2

CREARE LA PRIMA VIEW DISPONIBILE

- Le views possono essere annidate all'interno di sottocartelle della directory resources/views.
- La dot notation può essere utilizzata per referenziare le views annidate. Se, ad esempio, la view si trova in resources/views/admin/profile.blade.php , possiamo ritornarla da una delle rotte o dei controller in questo modo:

```
return view('admin.profile', $data);
```

- I nomi delle cartelle delle views non dovrebbero contenere il carattere . (punto)
- Il metodo first della facade View permette di creare la prima view che esiste in un dato array di views. Questo può risultare utile se l'applicazione o il pacchetto permette di personalizzare e sovrascrivere le views:

```
use Illuminate\Support\Facades\View;  
  
return View::first(['custom.admin', 'admin'], $data);
```

1.3

DETERMINARE SE UNA VIEW ESISTE

- Per determinare se una view esiste, si può utilizzare la facade View
- Il metodo `exists` ritorna true se la view esiste:

```
use Illuminate\Support\Facades\View;

if (View::exists('emails.customer')) {
    // ...
}
```

2

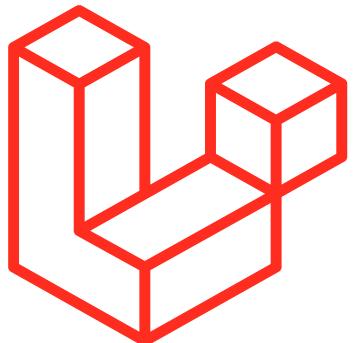
PASSARE DATI ALLE VIEWS

- È possibile passare un array di dati alle views per rendere quei dati disponibili alla view

```
return view('greetings', ['name' => 'Victoria']);
```

- Quando si passano informazioni in questo modo, i dati dovrebbero essere nella forma di array di coppie chiave/valore.
- Dopo aver fornito i dati alla view, possiamo accedere ogni valore all'interno della view usando la key del dato. Ad esempio <?php echo \$name ?>
- Un'alternativa al passaggio di un array completo di dati alla funzione helper view, si può usare il metodo with per aggiungere singoli dati alla view.
Il metodo view ritorna un'istanza dell'oggetto view in modo che si possa continuare a concatenare metodi prima di ritornare la view:

```
return view('greeting')  
    ->with('name', 'Victoria')  
    ->with('occupation', 'Astronaut');
```



Laravel

LE BASI

SESSION

- Poichè le applicazioni HTTP driven sono stateless, le sessioni rappresentano un modo per immagazzinare informazioni sull'utente attraverso richieste multiple.
- Queste informazioni sono tipicamente memorizzate in un archivio persistente/backend al quale si può accedere con richieste successive
- Laravel offre una serie di backend per le sessioni a cui si accede tramite un'API unificata.

CONFIGURAZIONE

Il file di configurazione della sessione dell'applicazione si trova in **config/session.php**.

Di default, Laravel è configurato per utilizzare il driver del file di sessione.

L'opzione di configurazione del driver di sessione definisce **dove saranno salvati i dati di sessione per ogni richiesta**.

Laravel viene fornito con **diversi drivers** eccellenti:

- file – le sessioni sono salvate in storage/framework/sessions
- Cookie – sessioni salvate in cookies sicuri e criptati
- Database – sessioni salvate in database relazionali
- Memcached /redis – sessioni salvate in uno di questi archive veloci e basati sulla cache
- Dynamob – sessioni salvate in AWS DynamoDB
- Array – sessioni memorizzate in un array PHP e non saranno conservate. Usato principalmente durante i test.

PREREQUISITI DEL DRIVER

DATABASE

Quando si usa il driver di sessione database, si dovrà creare una tabella per contenere i records della sessione.

ESEMPIO di dichiarazione di Schema per la tabella:

Si può usare il comando Artisan
session:table per generare questa
migration.

```
php artisan session:table
```

```
php artisan migrate
```

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::create('sessions', function (Blueprint $table) {
    $table->string('id')->primary();
    $table->foreignId('user_id')->nullable()->index();
    $table->string('ip_address', 45)->nullable();
    $table->text('user_agent')->nullable();
    $table->text('payload');
    $table->integer('last_activity')->index();
});
```

1- INTERAGIRE CON LA SESSION

1.1 – Recuperare dati

Esistono principalmente due modi per lavorare con i dati della session:

- L'helper globale session
- Un'istanza della classe Request

I. ISTANZA DELLA REQUEST

L'istanza della classe Request può essere type-hinted nella closure di una route o nel metodo del controller. Si ricorda che le dipendenze del metodo del controller sono automaticamente iniettate attraverso il service container.

```
namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * Show the profile for the given user.
     */
    public function show(Request $request, string $id): View
    {
        $value = $request->session()->get('key');

        // ...

        $user = $this->users->find($id);

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

Quando si recupera un elemento dalla session, si può passare un valore di default come secondo argomento del metodo get.

Questo valore di default sarà ritornato se la chiave specificata non esiste nella session.

Se si passa una closure come secondo argomento del metodo get e la chiave richiesta non esiste, la closure sarà eseguita ed il risultato ritornato

```
$value = $request->session()->get('key', 'default');

$value = $request->session()->get('key', function () {
    return 'default';
});
```

II. L'HELPER GLOBALE SESSION

Per recuperare e memorizzare i dati nella session, si può anche usare la funzione globale session.

Quando l'helper session è chiamato con un singolo argomento, ritorna il valore di quella chiave di session.

Quando è chiamato con un array di coppie chiave/valore, questi valori saranno salvati nella session

```
Route::get('/home', function () {
    // Retrieve a piece of data from the session...
    $value = session('key');

    // Specifying a default value...
    $value = session('key', 'default');

    // Store a piece of data in the session...
    session(['key' => 'value']);
});
```

1.1.1 – Recuperare tutti i dati della session

Per recuperare TUTTI i dati nella session, si usa il metodo **all**

```
$data = $request->session()->all();
```

1.1.2 – Determinare se un elemento esiste nella session

- Si usa il metodo **has** che ritorna un booleano:
true se l'elemento esiste e non è null.
- Per determinare **se un elemento esiste** nella session, **anche se il suo valore è null**, si usa il metodo **exists**
- Al contrario, per **verificare se un elemento è mancante nella session**, si usa il metodo **missing** che ritorna true se l'elemento non è presente

```
if ($request->session()->has('users')) {  
    // ...  
}
```

```
if ($request->session()->exists('users')) {  
    // ...  
}
```

```
if ($request->session()->missing('users')) {  
    // ...  
}
```

1.2 – Memorizzare dati

Per memorizzare i dati nella session, si usa tipicamente il metodo **put** dell'istanza della Request oppure si usa **l'helper globale session**.

I. Pushing nell'array dei valori della Request

Usato per **inserire un nuovo valore** in un valore di sessione che è un array.

Ad esempio, se la chiava user.teams contiene un array di nomi di team, si può inserire un nuovo valore nell'array.

```
// Via a request instance...
$request->session()->put('key', 'value');

// Via the global "session" helper...
session(['key' => 'value']);
```

```
$request->session()->push('user.teams', 'developers');
```

```
$value = $request->session()->pull('key', 'default');
```

```
$request->session()->increment('count');

$request->session()->increment('count', $incrementBy = 2);

$request->session()->decrement('count');

$request->session()->decrement('count', $decrementBy = 2);
```

II. Recuperare ed eliminare un elemento

Il metodo **pull** **recupera ed elimina** un elemento dalla sessione con una singola istruzione

III. Incrementare e decrementare valori di sessione

Si usano i metodi **increment** e **decrement** per **aumentare o diminuire** un intero

1.3 – FLASH DATA

Per memorizzare gli elementi **nella session per la richiesta successiva**, si può usare il metodo **flash**.

I dati memorizzati nella session con questo metodo saranno disponibili subito e durante la successiva richiesta HTTP.

In seguito alla richiesta successiva, i dati saranno cancellati.

I dati flash sono utili principalmente per messaggi di stato di breve durata

Per conservare i dati flash per più requests, si può utilizzare il metodo **reflash** che terrà i dati per un'altra request.

Se si ha bisogno solo di specifici dati, si può utilizzare il metodo **keep**.

Per conservare i dati solo per la request corrente, esiste il metodo **now**.

```
$request->session()->flash('status', 'Task was successful!');
```

```
$request->session()->reflash();  
$request->session()->keep(['username', 'email']);
```

```
$request->session()->now('status', 'Task was successful!');
```

1.4 – ELIMINARE DATI

Il metodo **forget** rimuove parte dei dati dalla session

Per eliminare **TUTTI I DATI** si usa il metodo **flush**

```
// Forget a single key...
$request->session()->forget('name');

// Forget multiple keys...
$request->session()->forget(['name', 'status']);

$request->session()->flush();
```

1.5 – RIGENERARE L'ID DELLA SESSION

Rigenerare l'ID della session è spesso fatto per prevenire un tipo di attacco di session fixation.

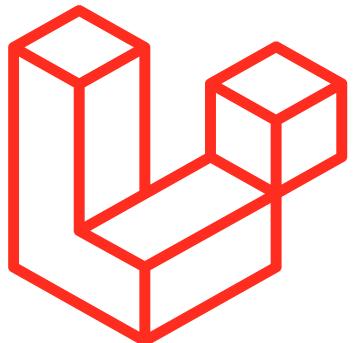
Se si utilizza Laravel Fortify o uno dei starter kits di Laravel, l'ID della session sarà automaticamente rigenerato.

Per farlo manualmente, invece, serve il metodo **regenerate**.

```
$request->session()->regenerate();

$request->session()->invalidate();
```

Se si vuole **cancellare l'ID della session e rimuovere tutti i dati dalla session in un'unica istruzione**, si usa il metodo **invalidate**



Laravel

LE BASI

BLADE TEMPLATES

- Blade è il **motore di templating** incluso con Laravel. A differenza di altri motori di templating PHP, Blade non impedisce di usare puro codice PHP all'interno dei template. Blade infatti compila i templates in **puro codice PHP** e li mette in cache finché il template è modificato.
- I file di template Blade hanno estensione **.blade.php** e si trovano nella directory **resources/views**
- Le views di Blade possono essere **ritornate dalle rotte** o dai **controllers** usando l'helper globale **view**.
- I **dati** possono essere passati alla view usando il **secondo argomento dell'helper view**.

```
Route::get('/', function () {  
    return view('greeting', ['name' => 'Finn']);  
});
```

1 - VISUALIZZARE DATI

- Si possono visualizzare i dati passati dalle views Blade inserendo la variabile tra parentesi graffe.
Ad esempio, data la seguente rotta:

```
Route::get('/', function () {  
    return view('welcome', ['name' => 'Samantha']);  
});
```

Si può visualizzare il contenuto della variabile name in questo modo:

```
Hello, {{ $name }}.
```

- Le echo {{ }} di Blade sono inviate automaticamente attraverso la funzione **htmlspecialchars** di PHP per prevenire gli attacchi XSS.
- Oltre a poter visualizzare il contenuto delle variabili passate alle views, si può anche fare l'echo dei risultati di qualsiasi funzione PHP. Si può infatti inserire qualsiasi codice PHP all'interno dell'echo statement di Blade.

```
The current UNIX timestamp is {{ time() }}.
```

1.2 – HTML ENTITY ENCODING

VISUALIZZARE UNESCAPED DATA

Di default gli statements {{ }} di Blade sono automaticamente inviati tramite la funzione `htmlspecialchars` per prevenire attacchi XSS. Per disabilitare questa opzione si usa la sintassi `{!! !!}`

```
Hello, {!! $name !!}.
```

- Di default, Blade effettuerà la doppia codifica delle entities HTML.
- Per disabilitare la doppia codifica, si chiama il metodo `Blade::withoutDoubleEncoding` dal metodo `boot` dell'AppServiceProvider

```
namespace App\Providers;

use Illuminate\Support\Facades\Blade;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     */
    public function boot(): void
    {
        Blade::withoutDoubleEncoding();
    }
}
```

Bisogna prestare molta attenzione nel mostrare i contenuti forniti dagli utenti. Si dovrebbe sempre usare la sintassi `escaped`, con doppie parentesi graffe per prevenire attacchi XSS.

- Siccome anche molti frameworks JavaScript usano le parentesi graffe per indicare che una data espressione deve essere visualizzata nel browser, si può usare il simbolo @ per informare Blade che un'espressione deve rimanere inalterata.
- Ad esempio:

```
<h1>Laravel</h1>
```

```
Hello, @{{ name }}.
```

In questo esempio, il simbolo @ sarà eliminato da Blade mentre l'espressione {{ name }} rimarrà inalterata, permettendole di essere visualizzata dai frameworks JavaScripts

Il simbolo @ può essere usato anche per l'escape delle direttive Blade

```
<script>
  var name = "John";
  var greeting = "Hello, @{{name}}!";
  console.log(greeting); // Stampa "Hello, John!"
</script>
```

```
--> Blade template -->
@&if()
<!-- HTML output -->
@if()
```

1.3 – BLADE & JAVASCRIPT FRAMEWORKS

- Se si stanno visualizzando variabili Javascript in una larga porzione del template, si può comprendere l'HTML nella direttiva @verbatim in modo da non dover aggiungere il prefisso @ ad ogni echo statement Blade.

```
@verbatim
    <div class="container">
        Hello, {{ name }}.
    </div>
@endverbatim
```

1.3.2 – LA DIRETTIVA @verbatim

2- DIRETTIVE BLADE

- Si può costruire uno statement condizionale usando le direttive **@if**, **@elseif**, **@else** ed **@endif** che funzionano come le controparti PHP.

```
@if (count($records) === 1)  
    I have one record!  
@elseif (count($records) > 1)  
    I have multiple records!  
@else  
    I don't have any records!  
@endif
```

- Oltre all'ereditarietà dei template e alla visualizzazione dei dati, Blade fornisce delle scorcatoie per le strutture di controllo PHP più comuni, come gli if statements ed i cicli.

2.1- IF STATEMENTS

@unless

- Per comodità, Blade fornisce anche una direttiva **@unless**.
- Il codice viene eseguito se la condizione restituisce false. Nell'esempio Auth::check() verifica se un utente è loggato restituendo true se lo è.

```
@unless (Auth::check())
    You are not signed in.
@endunless
```

```
@isset($records)
    // $records is defined and is not null...
@endisset

@empty($records)
    // $records is "empty"...
@endempty
```

@isset / @empty

- **@isset** funziona come iset() di php
- **@empty** controparte della funzione php empty()

2.1.1 - DIRETTIVE DI AUTENTICAZIONE

@auth/ @guest

- Possono essere utilizzate per determinare rapidamente se l'utente corrente è autenticato oppure è un ospite (guest)

```
@auth('admin')  
    // The user is authenticated...  
@endauth  
  
@guest('admin')  
    // The user is not authenticated...  
@endguest
```

```
@auth  
    // The user is authenticated...  
@endauth  
  
@guest  
    // The user is not authenticated...  
@endguest
```

Se necessario possono essere specificate le guard di autenticazione che devono essere controllate

@production

- Serve a controllare se l'applicazione sta girando in ambiente di produzione

```
@production  
    // Production specific content...  
  
@endproduction
```

```
@env('staging')  
    // The application is running in "staging"...  
@endenv  
  
@env(['staging', 'production'])  
    // The application is running in "staging" or "production"...  
@endenv
```

2.1.2- DIRETTIVE D'AMBIENTE

@env

- Serve a determinare se l'applicazione sta girando in uno specifico ambiente

@yield

- @yield serve per posizionare il punto in cui verrà inserita una section

```
<html>
    <head>
        <title>App Name - @yield('title')</title>
    </head>
    <body>
        @section('sidebar')
            This is the master sidebar.
        @show

        <div class="container">
            @yield('content')
        </div>
    </body>
</html>
```

```
@extends('layouts.app')

@section('title', 'Page Title')

@section('sidebar')
    @parent

    <p>This is appended to the master sidebar.</p>
@endsection

@section('content')
    <p>This is my body content.</p>
@endsection
```

@stack

Blade ti consente di eseguire il push su stack con nome che possono essere renderizzati altrove in un'altra vista o layout. Ciò può essere particolarmente utile per specificare eventuali librerie JavaScript richieste dalle visualizzazioni secondarie.

```
<head>
    <!-- Head Contents -->

    @stack('scripts')
</head>
```

```
@push('scripts')
    <script src="/example.js"></script>
@endpush
```

```
@prepend('scripts')
    This will be first...
@endprepend
```

@extends estendere un layout

Quando si definisce una vista secondaria, utilizzare la direttiva @extends Blade per specificare quale layout la vista secondaria dovrebbe "ereditare". Le viste che estendono un layout Blade possono inserire contenuto nelle sezioni del layout utilizzando le secondarie:

```
<!-- resources/views/child.blade.php -->

@extends('layouts.app')

@section('title', 'Page Title')

@section('sidebar')
    @parent

    <p>This is appended to the master sidebar.</p>
@endsection

@section('content')
    <p>This is my body content.</p>
@endsection
```

```
@hasSection('navigation')  
  <div class="pull-right">  
    @yield('navigation')  
  </div>  
  
  <div class="clearfix"></div>  
@endif
```

@hasSection

- Verifica se una sezione possiede dei contenuti

```
@sectionMissing('navigation')  
  <div class="pull-right">  
    @include('default-navigation')  
  </div>  
@endif
```

@sectionMissing

- Determina se una sezione non ha contenuto

2.1.3- DIRETTIVE DI SEZIONE

2.2- SWITCH STATEMENTS

- Possono essere costruiti usando le direttive:

@switch
@case
@break
@default
@endswitch

```
@switch($i)
  @case(1)
    First case...
    @break

  @case(2)
    Second case...
    @break

  @default
    Default case...

@endswitch
```

2.3 - CICLI (LOOPS)

- Oltre alle strutture condizionali, Blade fornisce direttive per lavorare con i cicli.
- Ognuna di queste direttive funziona come la controparte PHP

@for
@foreach
@forelse
@while



Quando si itera in un loop **foreach**, si può usare la variabile **\$loop** per raccogliere informazioni sul ciclo come, ad esempio, se si è nella prima o ultima iterazione

```
@for ($i = 0; $i < 10; $i++)  
    The current value is {{ $i }}  
@endfor
```

@for

```
@foreach ($users as $user)  
    <p>This is user {{ $user->id }}</p>  
@endforeach
```

@foreach

```
@forelse ($users as $user)  
    <li>{{ $user->name }}</li>  
@empty  
    <p>No users</p>  
@endforelse
```

@forelse

```
@while (true)  
    <p>I'm looping forever.</p>  
@endwhile
```

@while

```
@foreach ($users as $user)
    @if ($user->type == 1)
        @continue
    @endif

    <li>{{ $user->name }}</li>

    @if ($user->number == 5)
        @break
    @endif
@endforeach
```

Quando si usano i cicli, si può saltare l'iterazione corrente o terminare il ciclo usando le direttive **@continue** e **@break**

- Si può anche includere la **condizione di continuazione** o di **interruzione** all'interno della dichiarazione della direttiva.

```
@foreach ($users as $user)
    @continue($user->type == 1)

    <li>{{ $user->name }}</li>

    @break($user->number == 5)
@endforeach
```

2.4- LA VARIABILE LOOP (\$loop)

- Come già accennato, mentre si itera in un ciclo foreach, la variabile \$loop è disponibile all'interno del loop.
- La variabile \$loop consente di accedere ad alcune informazioni come l'indice dell'iterazione corrente oppure se quella attuale è la prima o ultima iterazione

```
@foreach ($users as $user)
    @foreach ($user->posts as $post)
        @if ($loop->parent->first)
            This is the first iteration of the parent loop.
        @endif
    @endforeach
@endforeach
```

CICLO INNESTATO

- Se ci si trova all'interno di un ciclo innestato, si può accedere alla variabile \$loop del parent loop attraverso la proprietà parent

```
@foreach ($users as $user)
    @if ($loop->first)
        This is the first iteration.
    @endif

    @if ($loop->last)
        This is the last iteration.
    @endif

    <p>This is user {{ $user->id }}</p>
@endforeach
```

■ PROPRIETA' DELLA VARIABILE LOOP

PROPRIETA'	DESCRIZIONE
\$loop->index	Indice dell'iterazione corrente (inizia a 0)
\$loop->iteration	L'iterazione corrente del ciclo (inizia ad 1)
\$loop->remaining	Le iterazioni rimanenti nel ciclo
\$loop->count	Il numero totale di elementi nell'array che viene iterato
\$loop->first	Se quella attuale è la prima iterazione del ciclo
\$loop->last	Se quella attuale è l'ultima iterazione del ciclo
\$loop->even	Se quella attuale è un'iterazione pari
\$loop->odd	Se l'iterazione attuale è dispari
\$loop->depth	Il livello di annidamento del ciclo corrente
\$loop->parent	Quando ci si trova in un loop annidato, la variabile loop del ciclo genitore

2.5 - LE CLASSI CONDIZIONALI E GLI STILI

@style

Usata per aggiungere condizionalmente uno stile CSS inline ad un elemento HTML

```
@php
    $isActive = true;
@endphp

<span @style([
    'background-color: red',
    'font-weight: bold' => $isActive,
])></span>

<span style="background-color: red; font-weight: bold;"></span>
```

@class

Questa direttiva compila in maniera condizionale una stringa per definire una classe CSS.

La direttiva accetta un array di classi nel quale la chiave dell'array contiene la classe o le classi che si vogliono aggiungere, mentre il valore è un'espressione booleana che stabilisce se aggiungere tale classe.

Se l'elemento dell'array ha una chiave numerica, sarà incluso ugualmente nella lista di classi visualizzata.

```
@php
    $isActive = false;
    $hasError = true;
@endphp

<span @class([
    'p-4',
    'font-bold' => $isActive,
    'text-gray-500' => ! $isActive,
    'bg-red' => $hasError,
])></span>

<span class="p-4 text-gray-500 bg-red"></span>
```

2.6- ATTRIBUTI AGGIUNTIVI

@checked

Direttiva utilizzata per indicare che un dato checkbox HTML è checked.

Farà l'echo di checked se la condizione fornita è uguale a true

```
<input type="checkbox"  
       name="active"  
       value="active"  
       @checked(old('active', $user->active)) />
```

```
<select name="version">  
  
    @foreach ($product->versions as $version)  
        <option value="{{ $version }}" @selected(old('version') == $version)>  
            {{ $version }}  
        </option>  
    @endforeach  
</select>
```

@selected

Usata per indicare se l'opzione di una select deve essere «selected»

@disabled

Indica se un dato elemento deve essere «disabled»

```
<button type="submit" @disabled($errors->isNotEmpty())>Submit</button>
```

@readonly

Usata per indicare se un dato elemento deve essere «readonly»

```
<input type="email"  
       name="email"  
       value="email@laravel.com"  
       @readonly($user->isNotAdmin()) />
```

```
<input type="text"  
       name="title"  
       value="title"  
       @required($user->isAdmin()) />
```

@required

Per indicare se un elemento deve essere required

@include

Consente di includere una view Blade dall'interno di un'altra view. Tutte le variabili che sono disponibili alla vista genitore, saranno rese disponibili anche alla view inclusa

```
<div>
    @include('shared.errors')

    <form>
        <!-- Form Contents --&gt;
    &lt;/form&gt;
&lt;/div&gt;</pre>
```

Anche se la view inclusa erediterà tutti i dati disponibili nella vista genitore, si può passare un array di dati addizionali che devono essere resi disponibili alla vista figlia

```
@include('view.name', ['status' => 'complete'])
```

2.7 – INCLUDERE SUBVIEWS

```
@includeIf('view.name', ['status' => 'complete'])
```

@includeIf

Se si cerca di includere una view che non esiste, Laravel lancerà un errore.

Se si vuole includere una vista che può essere o no presente, si usa la direttiva @includeIf

@each

Si possono **combinare i cicli e includes** in una singola riga usando la direttiva @each

```
@each('view.name', $jobs, 'job')
```

Il **primo argomento** della direttiva @each è il **nome della view** da mostrare per ogni elemento dell'array o della collection.

Il **terzo argomento** è il **nome** che sarà assegnato all'**iterazione corrente** all'interno della view.

Il **secondo argomento** è l'**array o collection** sul quale si desidera iterare

Si può passare anche un quarto argomento. Questo determina la view che deve essere **visualizzata se l'array è vuoto**.

```
@each('view.name', $jobs, 'job', 'view.empty')
```

2.7.1 – MOSTRARE VIEWS PER LE COLLECTIONS @each

Le views mostrate attraverso la direttiva @each **non ereditano le variabili dalla vista padre**.

@once

Consente di definire una porzione del template che sarà **valutata una volta per ciclo** di rendering.

Può essere utile per inserire un dato script JavaScript nell'header della pagina usando stacks.

Ad esempio, se si sta mostrando un dato componente all'interno di un loop, si potrebbe voler inserire lo script JavaScript nell'header solo la prima volta che componente è renderizzato.

```
@once
  @push('scripts')
    <script>
      // Your custom JavaScript...
    </script>
  @endpush
@endonce
```

```
@pushOnce('scripts')
  <script>
    // Your custom JavaScript...
  </script>
@endPushOnce
```

2.8 – LA DIRETTIVA @once

La direttiva **@once** è spesso usata insieme alle direttive **@push** o **@prepend**. Per comodità sono disponibili le direttive **@pushOnce** e **@prependOnce**.

In alcune situazioni può essere utile incorporare del codice PHP nelle views.

Per eseguire un blocco di PHP puro si può utilizzare la direttiva
@php

```
@php  
    $counter = 1;  
@endphp
```

```
@php($counter = 1)
```

Se si ha bisogno di inserire un singolo statement PHP, si può includere lo statement stesso all'interno della direttiva @php

2.9 – PHP PURO

2.10 – COMMENTI

Blade permette di definire dei commenti nelle views. A differenza dei commenti HTML, quelli di Blade non sono incluse nell'HTML ritornato dall'applicazione

```
{{-- This comment will not be present in the rendered HTML --}}
```

3 - FORMS

3.1 – CSRF FIELD

Ogni volta che si costruisce un form HTML bisognerebbe inserire un **campo nascosto CSRF token** in modo che il **middleware CSRF protection possa validare la richiesta.**

Per generare il token field si può usare la direttiva **@csrf**

```
<form method="POST" action="/profile">  
    @csrf  
    ...  
</form>
```

3.2 – METHOD FIELD

Poiché i moduli HTML non possono effettuare richieste **PUT, PATCH o DELETE**, è necessario aggiungere un campo **_method** nascosto per «falsificare» questi verbi HTTP. La direttiva **@method** Blade può creare questo campo.

```
<form action="/foo/bar" method="POST">  
    @method('PUT')  
    ...  
</form>
```

3.3 – ERRORI DI VALIDAZIONE @error

La direttiva **@error** può essere utilizzata per controllare velocemente **se esistono messaggi di errore di validazione** per un dato attributo.

All'interno della direttiva **@error**, si può fare l'echo della variabile **\$message** per mostrare l'errore:

```
<!-- /resources/views/post/create.blade.php -->

<label for="title">Post Title</label>

<input id="title"
       type="text"
       class="@error('title') is-invalid @enderror">

@error('title')
    <div class="alert alert-danger">{{ $message }}</div>
@enderror
```

La direttiva **@error** si compila come un'istruzione **«if»**. Per questo si può usare la direttiva **@else** nel caso in cui non ci sia un errore per un determinato attributo.

```
<!-- /resources/views/auth.blade.php -->

<label for="email">Email address</label>

<input id="email"
       type="email"
       class="@error('email') is-invalid @else is-valid @enderror">
```

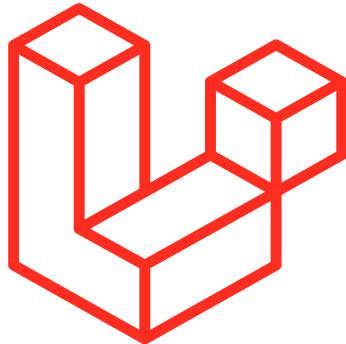
Come secondo argomento della direttiva @error, si può passare il nome di uno specifico **insieme di errori** per recuperare i **messaggi di errore di validazione su pagine che contengono più forms**.

```
<!-- /resources/views/auth.blade.php -->

<label for="email">Email address</label>

<input id="email"
       type="email"
       class="@error('email', 'login') is-invalid @enderror">

@error('email', 'login')
    <div class="alert alert-danger">{{ $message }}</div>
@enderror
```



Laravel

LE BASI

GENERAZIONE DEGLI
URL

- Laravel fornisce una **serie di helpers per la generazione di URLs** per l'applicazione.
- Questi helpers sono particolarmente utili quando si costruiscono link nei templates e nelle risposte API oppure quando si generano redirect di risposta ad un'altra parte dell'applicazione.
- L'helper url può essere utilizzato per generare URLs arbitruserà automaticamente lo **schema** (HTTP o HTTPS) e l'host della richiesta corrente. L'URL generato

```
$post = App\Models\Post::find(1);

echo url("/posts/{$post->id}");

// http://example.com/posts/1
```

ACCEDERE L'URL CORRENTE

Se non è fornito alcun path all'url helper, sarà ritornata un'istanza di Illuminate\Routing\UrlGenerator che permette di **accedere ad informazioni circa l'URL attuale**.

```
// Get the current URL without the query string...
echo url()->current(); ★
```

```
// Get the current URL including the query string...
echo url()->full(); ★
```

```
// Get the full URL for the previous request...
echo url()->previous(); ★
```

Si può accedere ad ognuno di questi metodi anche attraverso la **URL facade**

```
use Illuminate\Support\Facades\URL;

echo URL::current();
```

URLs PER ROTTE NOMINATE

- Route helper

- L'helper route può essere usato per generare URLs alle rotte nominate.
- Le rotte nominate permettono di generare URLs senza essere accoppiate all'effettivo URL definito sulla rottta.
- Quindi, se l'URL della rottta cambia, non serve cambiare la chiamata alla funzione route.

ESEMPIO: supponiamo che l'applicazione contenga una rottta definita come segue:

Per generare un URL a questa rottta, si deve usare l'helper nel seguente modo

```
Route::get('/post/{post}', function (Post $post) {  
    // ...  
})->name('post.show');
```

```
echo route('post.show', ['post' => 1]);  
  
// http://example.com/post/1
```

- Ovviamente l'helper può essere usato anche per generare URLs per rotte con parametri multipli

```
Route::get('/post/{post}/comment/{comment}', function (Post $post,  
Comment $comment) {  
    // ...  
})->name('comment.show');  
  
echo route('comment.show', ['post' => 1, 'comment' => 3]);  
  
// http://example.com/post/1/comment/3
```

URLs PER ROTTE NOMINATE

- Route helper

- Ogni elemento addizionale dell'array che non corrisponde alla definizione dei parametri della route, sarà aggiunto alla query string dell'URL:

```
echo route('post.show', ['post' => 1, 'search' => 'rocket']);  
  
// http://example.com/post/1?search=rocket
```

MODELLO ELOQUENT

- Si dovranno spesso generare URLs usando le chiavi delle rotte (solitamente la chiave primaria) dei modelli Eloquent.
Si possono passare i modelli Eloquent come valori dei parametri.
- L'helper route estrarrà automaticamente la chiave della rotta del modello.

```
echo route('post.show', ['post' => $post]);
```

URLS PER ROTTE NOMINATE

- Signed URLs

- Laravel permette di creare molto facilmente degli URLs «firmati» alle rotte nominate.
- Questi URLs hanno una firma (signature) hash alla fine della query string. Questa permette a Laravel di verificare che l'URL non è stato modificato da quando è stato creato.
Gli URLs firmati sono particolarmente utili per le rotte che sono pubblicamente accessibili e che hanno bisogno di un livello di protezione contro la manipolazione dell'URL.

ESEMPIO:

Si potrebbero usare gli URLs firmati per implementare un link pubblico «disiscriviti» che viene inviato per email all'utente.

- Per creare una URL firmata verso una rotta nominata, si usa il metodo `signedRoute` della facade URL

```
use Illuminate\Support\Facades\URL;  
  
return URL::signedRoute('unsubscribe', ['user' => 1]);
```

- Si possono generare URL temporanee firmate che scadono dopo un periodo di tempo specificato usando il metodo `temporarySignedRoute`.
Quando Laravel valida queste URLs, si assicura che il timestamp di scadenza che è codificato nell'URL firmato non sia scaduto.

VITE

BUILD E ASSET BUNDLING IN LARAVEL

Indipendentemente da come si sia implementato il frontend del proprio progetto Laravel, sarà necessario caricare JS e CSS quando il browser apre una pagina.

Dalla versione 9.x Laravel utilizza Vite per eseguire il **bundle degli asset JS e CSS**.

Vantaggi di Vite:

- **Ricarica al volo le modifiche** ai sorgenti durante la fase di **sviluppo**
- **Ottimizza gli asset** quando l'applicazione è **in produzione**
- Offre pieno supporto a tecnologie come React e Vue

Lo starter kit di Laravel già include la libreria Vite.

Nel caso non si disponga di tale libreria ed occorra installarla, bisogna assicurarsi che Node.js (16+) e npm siano installati.

```
node -v  
npm -v
```

Se questi sono installati si può procedere con l'installazione delle dependencies via NPM

```
npm install
```

CONFIGURAZIONE DI VITE

Vite è configurabile nel file `vite.config.js` nella root del progetto Laravel.

La configurazione di default prevede che Vite venga usato per indicare quali sono gli **entry point della propria applicazione**. Da impostazioni predefinite, vengono utilizzati tutti i file `resources/css/app.css` e `resources/js/app.js`.

È possibile però indicare anche file preprocessati come Typescript, JSX, TSX e Sass.

Nel caso si stesse lavorando ad una single page application, Vite lavora meglio senza entry points.

Bisognerebbe invece importare il CSS via JavaScript. Solitamente questo viene fatto all'interno del file `resources/js/app.js`

```
import './bootstrap';
import '../css/app.css';
```

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
    plugins: [
        laravel([
            'resources/css/app.css',
            'resources/js/app.js',
        ]),
    ],
});
```

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
    plugins: [
        laravel([
            'resources/css/app.css',
            'resources/js/app.js',
        ]),
    ],
});
```

CARICAMENTO DI SCRIPTS E STILI

Una volta configurati gli entry points, è possibile referenziarli in una direttiva Blade del tipo `@vite()` aggiunta al tag `<head>` del root template dell'applicazione che si sta sviluppando.

Se si sta importando il CSS via JavaScript, servirà solo includere l'entry point JavaScript

La direttiva `@vite` individuerà automaticamente il **server di sviluppo** Vite ed inietterà il client Vite per abilitare **l'Hot Module Replacement** (genera in automatico gli opportuni CSS e JS da caricare nella pagina). In modalità **build**, la direttiva **caricherà gli assets compilati e versionati**, incluso qualsiasi CSS importato.

Se necessario, è possibile specificare il path build degli assets compilati quando si invoca la direttiva `@vite`.

```
<!doctype html>
<head>
  {{-- ... --}}
  @vite(['resources/css/app.css', 'resources/js/app.js'])
</head>
```

```
<!doctype html>
<head>
  {{-- ... --}}
  @vite('resources/js/app.js')
</head>
```

```
<!doctype html>
<head>
  {{-- Given build path is relative to public path. --}}
  @vite('resources/js/app.js', 'vendor/courier/build')
</head>
```

Inline assets

A volte può essere necessario includere il contenuto grezzo delle risorse piuttosto che linkare all'URL della risorsa.

Ad esempio, può servire includere il contenuto direttamente nella pagina quando si passa il contenuto HTML ad un generatore PDF.

È possibile produrre il contenuto degli assets Vite utilizzando il metodo `content` fornito dalla facade Vite.

```
@php
use Illuminate\Support\Facades\Vite;
@endphp

<!doctype html>
<head>
    {{-- ... --}}


    <style>
        {!! Vite::content('resources/css/app.css') !!}
    </style>
    <script>
        {!! Vite::content('resources/js/app.js') !!}
    </script>
</head>
```

LANCIARE VITE

Esistono due modi per lanciare Vite:

1. Lanciare il server di sviluppo con il comando `npm run dev` (utile quando si sviluppa in locale). Il server di sviluppo **individuerà automaticamente i cambiamenti ai files** e li mostrerà nel browser.
2. Eseguire il comando `build`. Ciò consente di **versionare e raggruppare le risorse** dell'applicazione, rendendole pronte per la distribuzione in produzione. Vite creerà i file `public/build/assets/app<HASH_UNIVOCO>.css` e `public/build/assets/app<HASH_UNIVOCO>.js`

Il template Blade includerà questi file per fare il rendering della view e verranno erogati direttamente dall'applicazione Laravel, in quanto inclusi nel public.

```
# Run the Vite development server...
npm run dev

# Build and version the assets for production...
npm run build
```

ELABORAZIONE DEGLI URL

Quando si usa Vite e si fa riferimento alle risorse HTML, CSS o JS bisogna prestare attenzione ai percorsi utilizzati:

- Se si indicano **percorsi assoluti**, Vite **non includerà la risorsa nella compilazione**. Assicurarsi, dunque, che la risorsa sia disponibile nella directory pubblica
- I **percorsi relativi** sono relativi al file a cui si fa riferimento. Qualsiasi risorsa cui si fa riferimento con un percorso relativo sarà **riscritta, versionata e distribuita da Vite**.

```
public/  
    taylor.png  
resources/  
    js/  
        Pages/  
            Welcome.vue  
        images/  
            abigail.png
```

STRUTTURA DI UN PROGETTO

```
<!-- This asset is not handled by Vite and will not be included in the build -->  
  
  
<!-- This asset will be re-written, versioned, and bundled by Vite -->  

```

LAVORARE CON BLADE E ROTTE

Vite elabora e inietta automaticamente risorse in JavaScript o CSS.

Quando si costruiscono applicazioni usando **Blade**, Vite può anche **elaborare e versionare le risorse statiche** a cui si fa riferimento esclusivamente nei template di Blade.

Bisogna però importare queste risorse statiche nell'entry point dell'applicazione.

Ad esempio, se si desidera elaborare e versionare tutte le immagini memorizzate in `resources/images` e tutti i font memorizzati in `resources/fonts`, è necessario aggiungere quanto segue nel punto di ingresso `resources/js/app.js` dell'applicazione:

```
import.meta.glob([
  './images/**',
  './fonts/**',
]);
```

Queste risorse verranno processate da Vite quando si lancerà `npm run build`.

Si può fare riferimento a queste risorse nei templates Blade usando il metodo `Vite::asset` che ritornerà l'URL versionato per una data risorsa

```

```

REFRESH ON SAVE

Quando si utilizza il rendering lato server con Blade, **Vite aggiorna automaticamente il browser quando si modificano i file di visualizzazione dell'applicazione.**

Perché ciò avvenga occorre impostare l'opzione di aggiornamento su true.

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
  plugins: [
    laravel({
      // ...
      refresh: true,
    }),
  ],
});
```

E' possibile **specificare una lista personalizzata di paths:**

Quando l'opzione refresh è true, il salvataggio di file nelle seguenti directories, attuerà un refresh dell'intera pagina quando si lancia il comando npm run dev:

- app/View/Components/**
- lang/**
- resources/lang/**
- resources/views/**
- routes/**

```
export default defineConfig({
  plugins: [
    laravel({
      // ...
      refresh: ['resources/views/**'],
    }),
  ],
});
```

ALIASES

Nelle applicazioni Javascript è comune **referenziare directories creando degli alias**.

E' possibile creare alias da usare in Blade con il metodo macro della classe `Illuminate\Support\Facades\Vite`. Tipicamente le «macro» dovrebbero essere definite all'interno del metodo boot di un service provider.

```
/**  
 * Bootstrap any application services.  
 */  
public function boot(): void  
{  
    Vite::macro('image', fn (string $asset) => $this->asset("resources/images/{$asset}"));  
}
```

Una volta che la macro è stata definita, può essere chiamata all'interno dei templates.

Ad esempio, si può usare la macro `image` definita per referenziare una risorsa che si trova in `resources/images/logo.png`

```

```

CUSTOM BASE URLs

Se le risorse compilate di Vite vengono fornite da un dominio separato dalla nostra applicazione, come nel caso di un CDN, bisogna specificare la variabile d'ambiente `ASSET_URL` all'interno del file `.env`

```
ASSET_URL=https://cdn.example.com
```

Dopo aver configurato l'URL della risorsa, tutti gli URLs alle risorse verranno riscritti aggiungendo il prefisso con il valore configurato (**gli URLs assoluti non verranno riscritti da Vite**):

```
https://cdn.example.com/build/assets/app.9dce8d17.js
```

ARGOMENTI VITE NON INSERITI:

- *Variabili d'ambiente*
- *Disabilitare Vite nei test*
- *Server side rendering*
- *cspNonce*
- *Vite-plugin-manifest-sri*
- *Arbitrary attributes (es: data-turbo-track)*
- *Advanced customisation*

ROUTE MODEL BINDING

È la possibilità di **associare automaticamente uno o più modelli a determinate rotte web**, evitando di lavorare a livello di ‘id’ ma potendo **lavorare direttamente a partire dai modelli** che questi id solitamente rappresentano.

Spesso quello che succede nel web è avere una rotta che identifica un modello, per esempio un blog potrebbe avere una rotta simile a /articolo/1 dove 1 rappresenta proprio l’id del Post da visualizzare.

Soltamente all’interno del controller che gestisce questa rotta, il primo metodo che viene invocato è un **findOrFail** a partire, appunto, dall’id per recuperare l’oggetto o per segnalare errore all’utente.

Usando invece il Route Model Binding delegiamo questa operazione ripetitiva a Laravel.

ROUTE MODEL BINDING

BINDING IMPLICITO

Questa funzionalità sfrutta il type-hinting di PHP per capire l'intenzione dell'utente e consente di risolvere automaticamente le istanze dei modelli basandosi sui parametri delle routes.

```
Route::get('/posts/{post}', function (Post $post) {
    // Laravel risolve automaticamente un'istanza di Post basandosi sul valore dell'ID passato nel parametro {post} nella route.
    // $post sarà già un'istanza del modello Post corretto.
    return view('post.show', ['post' => $post]);
});
```

In questo esempio viene definita una route con parametro {post}. Quando un utente accede a questa rotta con un valore specifico per {post}, Laravel utilizzerà automaticamente l'implicit Route Model Binding per risolvere l'istanza corretta del modello 'Post' basandosi sull'ID fornito.

Per utilizzare l'implicit Model Binding, il nome del parametro nella definizione della route ({post}) deve corrispondere al parametro del metodo di callback (function (Post \$post)) dove si desidera utilizzare il modello associato.

ROUTE MODEL BINDING

BINDING IMPLICITO – modelli Soft Deleted

Tipicamente, l'implicit model binding non recupera modelli che sono stati soft deleted. Si può in ogni caso istruire l'implicit binding per recuperare questi modelli concatenando il metodo `withTrashed` nella definizione delle rotte

```
use App\Models\User;

Route::get('/users/{user}', function (User $user) {
    return $user->email;
})->withTrashed();
```

ROUTE MODEL BINDING

BINDING IMPLICITO – personalizzare la Key

A volte si possono voler risolvere i modelli Eloquent usando una colonna diversa dall'id. Per farlo basta specificare la colonna nei parametri della definizione della rottta.

```
use App\Models\Post;

Route::get('/posts/{post:slug}', function (Post $post) {
    return $post;
});
```

Se si vuole usare sempre una colonna diversa dall'id quando si recupera una determinata classe, si può sovrascrivere il metodo `getRouteKeyName` nel modello Eloquent

```
/**
 * Get the route key for the model.
 */
public function getRouteKeyName(): string
{
    return 'slug';
}
```

DATABASE

- TRANSAZIONI

TRANSAZIONI

È possibile eseguire le varie query tramite transazione.

Una transazione consente di eseguire diverse operazioni sul database.

Viene automaticamente eseguito il rollback di tutte quelle eseguite fino ad un certo punto se una delle operazioni non dovesse riuscire.

Il secondo argomento opzionale del metodo transaction indica il numero di volte per cui una transazione deve essere rieseguita in caso di blocco.

```
use Illuminate\Support\Facades\DB;

DB::transaction(function () {
    DB::update('update users set votes = 1');

    DB::delete('delete from posts');
}, 5);
```

Secondo argomento: numero di tentativi in caso di deadlock

TRANSAZIONI

È possibile eseguire le varie query tramite transazione. Una transazione consente di eseguire diverse operazioni sul database.

Viene automaticamente eseguito il rollback di tutte quelle eseguite fino ad un certo punto se una delle operazioni non dovesse riuscire.

Il secondo argomento opzionale del metodo transaction indica il numero di volte per cui una transazione deve essere rieseguita in caso di blocco.

```
use Illuminate\Support\Facades\DB;  
  
DB::transaction(function () {  
    DB::update('update users set votes = 1');  
  
    DB::delete('delete from posts');  
}, 5);
```

Secondo argomento: numero di tentativi in caso di deadlock

CONNESSIONE ALLA CLI DEL DATABASE

Si usa il comando Artisan db per connettersi alla command line del database.

```
php artisan db
```

Per la connessione ad un database non predefinito, si specifica il suo nome

```
php artisan db mysql
```

Numerosi sottocomandi permettono di ispezionare il database e ricavarne informazioni anche sulle singole tabelle.

COMANDO SHOW: fornisce una panoramica sul database includendone la dimensione, il tipo, il numero di connessioni aperte ed un sommario delle sue tabelle.

```
$ php artisan db:show  
SQLite .....  
Database ..... database/database.sqlite  
Host .....  
Port .....  
Username .....  
URL .....  
Open Connections .....  
Tables ..... 6  
Table ..... Size (MiB)  
failed_jobs ..... —  
migrations ..... —  
password_resets ..... —  
personal_access_tokens ..... —  
posts ..... —  
users ..... —
```

COMANDO TABLE: fornisce una panoramica su una singola tabella del database: colonne, tipo, attributi, chiavi e indici:

```
$ php artisan db:table posts
posts ..... .
Columns ..... 3
Column ..... Type
id autoincrement, integer ..... integer
title text ..... text
active integer ..... 1 integer
Index .....
primary id ..... unique, primary
```

db:monitor ?

Listening for query events

RIASSUMENDO....

- Il routing è il modo con cui una richiesta HTTP di una certa URI viene indirizzata al file o ad una porzione di codice deputato alla creazione del contenuto della risposta.
- È sufficiente registrare una rotta per gestire le richieste
- Una rotta registrata è individuata dal metodo HTTP e dal path della risorsa ed indica cosa fare per costruire la risposta da mandare al cliente
- Tutte le rotte registrate sono presenti nel file routes/web.php
- La maggior parte della gestione di basso livello del routing è nascosta agli sviluppatori i quali, però, possono sempre customizzare il contenuto di specifici file e classi.

FONTI

- [Gestore di dipendenze Composer \(aulab.it\)](#)
- [Laravel e il ciclo di vita delle richieste HTTP \(aulab.it\)](#)
- [PHP Laravel Project Example for Beginners – Phppot](#)
- [PHP Laravel Project Example for Beginners – Phppot](#)
- [Laravel: risorse gratuite per sviluppatori Web e non solo \(laramind.com\)](#)
- [Laravel - Middleware | Tutorialspoint](#)
- [Laravel Tutorial: Step by Step Guide to Building Your First Laravel Application | Laravel News \(laravel-news.com\)](#)
- [The Best Laravel Tutorials !\[\]\(bb75de1d64f098072fde46ba53faeb1d_img.jpg\) For Beginners 2023 - DEV Community](#)
- [Laravel Tutorial for Beginners \(guru99.com\)](#)
- [Laravel View Composers Vs View Creators | Medium](#)

TERMINI UTILI

- **HELPERS:** gli helpers in Laravel sono funzioni built-in che possono essere chiamate in qualsiasi punto dell'applicazione. Se l'helper di cui si ha bisogno non esiste, si può creare il proprio helper.