Indice

1. Capitolo 1 - Premessa
    1. Perchè Laravel?
    2. A chi è rivolto
    3. Cinguetta con Twitter
2. Capitolo 2 - Cenni teorici
    1. Cosa si intende per OOP
    2. Cos'è PHP
    3. Cosa significa MVC
    4. Cos'è un Framework
    5. Perché scegliere Laravel
3. Capitolo 3 - L'ambiente di sviluppo
    1. Node
    2. Terminale
    3. Composer
        1. Installazione per Mac
        2. Installazione per Windows
        3. Installazione per Ubuntu
    4. PHP/MySQL
        1. Auto Setup
        2. Installazione Manuale
            1. Mac
            2. Windows
            3. Ubuntu
4. Capitolo 4 - Installazione di Laravel
    1. Laravel Installer
    2. Installazione via Composer
    3. FAQ in caso di errori post-installazione
    4. La struttura interna
    5. Configurazione del database
5. Capitolo 5 - Le Routes
    1. Url e parametri
    2. Assegna un nome alle Route
    3. Route di gruppo

    4. Raggruppamento in base ai Middleware
    5. Raggruppamento in base al prefisso URL
    6. Raggruppamento che genera il nome delle route
    7. Route cache
6. Capitolo 6 - I Controller
    1. Sintassi
    2. Middleware controller
7. Capitolo 7 - Le View
    1. Blade
    2. Richiamare una View
    3. Passare i dati alle view
8. Capitolo 8 - Response
    1. Stringhe
    2. Array
    3. Oggetti
    4. Redirect
    5. Json
    6. Download
    7. File
9. Capitolo 9 - Request
    1. Path della Request
    2. Url della Request
    3. Il metodo della richiesta
    4. Gestire i dati
    5.
10. Capitolo 10 - Validation
    1. Logica di convalida
    2. Request personalizzate
    3. Regole di Convalida
    4. Visualizza errori nella view
11. Capitolo 11 - Migrations
    1. Genera una Migrations
    2. Comandi utili
    3. Le tabelle
    4. Tabella/Colonna già presenti
    5. Rinominare / eliminare tabelle

    6. Modifica Colonne
12. Capitolo 12 - I Model
    1. Convenzioni Eloquent
13. Capitolo 13 - Blade
    1. Strutturare un layout
    2. Comandi Blade
    3. Form
    4. Direttive
    5. Componenti
    6. Stack
14. Capitolo 14 - Middleware
15. Capitolo 15 - Seeding
    1. Faker
    2. Struttura
    3. Lancia il Seeding
16. Capitolo 16 - Events e Listeners
    1. Il provider di riferimento
    2. Genera un Events e un Listeners
    3. Definire un Events
    4. Definire un Listeners
    5. Dispatching di un evento
17. Capitolo 17 - Gate e Policy
    1. Gates
    2. Policy
18. Capitolo 18 - Resource Controller
19. Capitolo 19 - Sessions
20. Capitolo 20 - Localizzazione
21. Capitolo 21 - Gestione degli Errori
22. Neptune 1 - Crea il progetto
23. Neptune 2 - Gestire i contenuti
24. Neptune 3 - Dashboard
25. Neptune 4 - Ruoli e Autorizzazioni
26. Neptune 5 - Landing Page

Capitolo 1 - Premessa

Negli ultimi 6 anni ho avuto la straordinaria opportunità di avvicinarmi al mondo Laravel e, soprattutto, di lavorarci.

Sono qui per raccontarti di questo fantastico ecosistema e incoraggiarti a diventare un **Web Artisan** a tutti gli effetti.

Inizialmente erano appunti; poi trasformate in mini-guide e, così, circa 3 anni fa ho deciso di trasformare il tutto in una guida completa.

Buona lettura,
Francesco.

Perché Laravel?

Utilizzare Laravel ti permetterà non solo di sviluppare applicazioni web robuste e di grande affidabilità, ma ti regalerà momenti di pura goduria.

Ti sembrerà di utilizzare un altro linguaggio (sto parlando di PHP!).

Entrerai in una famiglia che conta migliaia di sviluppatori, centinaia di package open-source pronti all'uso e infiniti tutorial disponibili per lo più gratuitamente.

Soprattutto avere nel bagaglio tecnico un framework così duttile e diffuso ti renderà un professionista molto apprezzato.

A chi è rivolto

La guida è improntata sull'evidenziare la facilità di sviluppo di Laravel e attirare quella fetta di scettici che, temendo la complessità della parola "Framework", si spostano sul CMS di turno.

ATTENZIONE: Non è mia intenzione screditare o sminuire i CMS.

Cinguetta con twitter

P.S: ai fini della guida l'utilizzo di twitter non è obbligatorio!

La forma mentis di noi italiani è quella di intendere lo sviluppatore come il classico tipo strano con occhiali, brufoli e una spiccata abilità con il computer.

Non ha altri interessi nella vita, non è interessante e per nulla "figo".

In America questo stereotipo è stato sdoganato ormai da anni.

Utilizzando Twitter e seguendo determinati profili che ti andrò a suggerire, ti ritroverai nel tuo feed notizie scritte niente meno che dai protagonisti della scena Laravel e non solo.

Capirai che dietro quelle menti brillanti e apparentemente fuori dal comune si nascondo delle persone "umane" come te.

Ma un social network, senza un network di contatti iniziale, non ha ragione di esistere.
Eccoti quindi una lista di account da seguire:

Blog

- **Laracasts**(*laracst*)
- **Daily Laravel** (*DailyLaravel*)
- **PHP** (*Officialphp*)
- **Laravel.io** (*laravelio*)
- **Laravel News** (*laravelnews*)
- **Laravel** (*laravelphp*)

Sviluppatori

- **Taylor Otwell** (*taylorotwell*): Il creatore di Laravel
- **Jeffrey Way** (_jeffreyway): Fondatore di Laracasts e autore di libri a tema Laravel;
- **Adam Wathan** (*adamwathan*): Ha sviluppato Laravel valet, Jigsaw, Tailwindcss ed è autore di "Refactoring to Collections", "Test-Driven Laravel" e "Refactoring UI";
- **Matt Stauffer** (*stauffermatt*): Autore di "Laravel: Up & Running" e molto attivo nella community Laravel;
- **Nuno Maduro** (*enunomaduro*): Ha sviluppato Laravel Zero e scrive regolarmente sul suo blog articoli su PHP;
- **Caleb Porzio** (*calebporzio*): Molto attivo nel mondo Open-Source, ha creato LaravelLivewire;
- **Freek Van der Herten** (*freekmurze*): Molto attivo nella community Laravel, con la sua Web Agency Spatie ha all'attivo un'infinità di package per Laravel;
- **Bobby Bouwmann** (*bobbybouwmann*): Top Contributor della piattaforma Laracasts e autore di "Laravel secrets";
- **Graham Campbell** (*GrahamJCampbell*): Membro del Team Core di Laravel;
- **Mohamed Said** (*themsaid*): Membro del Team Core di Laravel e creatore di Wink, una piattaforma di pubblicazione open-source;
- **Paul Redmond **(*paulredmond*): Membro staff del blog Laravel News;
- **Rasmus Lerdorf** (*rasmus*): Il creatore di PHP;
- **Francesco Malatesta** (*francescocodes*): Fondatore della community italiana "Laravel Italia" e autore di diversi libri sull'argomento (completamente in Italiano);
- **Francesco Lettera** (*FrancescoLetter*): Sviluppatore italiano, ha all'attivo 3 Libri su Laravel (completamente in Italiano);
- **Enrico Zimuel** (*ezimuel*): Pilastro della community Italiana PHP;
- **Francesco Mansi** (*framansi*): Il mio account, ovviamente se ti va, seguimi.

Community Italiana Facebook

Per quanto riguarda *Facebook* il mio unico consiglio è quello di entrare nel gruppo Laravel Italia, puoi

raggiungerlo a questo link

Capitolo 2 - Cenni teorici

Non spaventarti, i capitoli proposti sono corti e funzionali.
Sentiti libero di saltare i paragrafi che non ti interessano.
Lavorare con un Framework MVC senza cognizione di causa non è mai una scelta saggia.

Cosa si intende per OOP

OOP sta per *Object Oriented Programming, ovvero* Programmazione Orientata ad Oggetti. Il paradigma OOP nasce per migliorare l'astrazione logica della programmazione procedurale, troppo limitata e lenta.

Come? **Introducendo l'elemento Oggetto *(o istanza)* che si occuperà di eseguire un solo compito.**

Gli oggetti menzionati poco sopra sono composti da **Classi,** ovvero dei modelli contenenti variabili e funzioni ancora più specifiche.

Queste variabili prendono il nome di **Attributi** e **Metodi**.

Lo stesso oggetto potrà essere richiamato più volte durante il flusso logico e, a seconda del metodo richiesto, restituirà valori diversi.

Cos'è PHP

Acronimo ricorsivo di Hypertext Preprocessor (in origine *Personal Home Page*), PHP è un linguaggio di scripting interpretato e attualmente utilizzato principalmente per sviluppare applicazioni lato server (backend).

Nel 1994 ***Rasmus Lerdorf***, il suo creatore, aveva concepito PHP come una raccolta di script C.G.I. (Common Gateway Interface)—*adesso passato al FastCGI*—che permettevano una facile gestione delle pagine personali in maniera dinamica e veloce. Con le versioni successive venne introdotto il supporto a mSQL - *il predecessore di MySQL* - e una gestione più semplice dei form.

Cosa significa MVC

MVC — ***M****odel-****V****iew-****C****ontroller* — è uno schema architetturale molto diffuso nello sviluppo di software, visto che si occupa della suddivisione del flusso logico in tre blocchi interconnessi tra loro.

- ***Funzionalità di Business***: Il blocco che gestisce direttamente i dati, i principi logici e le regole dell'applicazione. Questo significa che lo stesso dato può avere un output diverso in base all'interfaccia grafica e al tipo di richiesta, il tutto senza modificare l'informazione sorgente.
- ***Logica di Controllo***: Il nucleo del sistema (Core) è il blocco che contiene materialmente il codice con tutte le istruzioni scritte dallo sviluppatore. Intercetta gli input dell'utente, li elabora, interroga i dati e genera un output non interpretato.
- ***Logica di Presentazione:*** Interfaccia grafica (GUI) o Vista (View), è il blocco che si interfaccia direttamente con l'utente finale. Qui vengono inseriti tutti gli input dell'utente e visualizzati tutti gli

output generati dal sistema.

Seguendo lo schema proposto poco sopra, è facile evincere che l'utente comunicherà sempre e solo con il livello Presentazione.

Cos'è un Framework

Un Framework è un insieme di classi predefinite e facilmente implementabili, su cui un'applicazione può essere progettata e sviluppata.

Definisce uno standard di sviluppo rendendo possibile la realizzazione di librerie e integrazioni ad-hoc.

L'utilizzo di Framework è diventato un requisito fondamentale per restare al passo con i frenetici tempi di sviluppo odierno.

Funzioni basilari come l'autenticazione, la validazione dei dati in un form o il routing delle pagine sono già disponibili e ben documentate.

L'utilizzo di un Framework in un progetto non è obbligatorio, ma è la scelta più sensata che tu possa fare.

Perché scegliere Laravel

Il progetto originariamente si chiamava Bootplant. Nelle cronache di Narnia, Cair Paravel era il nome di un castello in cui vivevano i re e le regine del regno. Ho pensato così a Laravel. Ritenevo che il nome avesse un suono elegante e sofisticato.

É lo stesso creatore di Laravel - **Taylor Otwell** - a raccontare il retroscena dietro al nome con cui rinominò la prima repository pubblica del framework.

Sono passati quasi 10 anni dal quel fatidico Giugno del 2011 e ad oggi, Github alla mano, Laravel è il Framework PHP più popolare ed utilizzato.

Gran parte delle documentazioni dei servizi di terze parti più famosi integrano interi capitoli sul Framework: Laravel è ormai uno standard.

Blade

Blade è il template engine predefinito di Laravel. Un file blade non è altro che una **view** scritta in php. Dovrai aggiungerci soltanto una pre-estensione `view.blade.php`.

A differenza degli altri template engine, Blade non ha alcuna restrizioni sul PHP. Potrai scrivere direttamente porzioni di codice nativo senza problemi. Ma sono sicuro che una volta comprese le reali potenzialità, abbandonerei la malsana idea di scriverci PHP.

Un'applicazione web, solitamente, ha una struttura composta da Header, Body e Footer. Ipotizzando che l'unico elemento dinamico sia il Body, che senso ha compilare ogni volta TUTTA la pagina?

Blade ti permetterà di dividere il layout in **Sections** e **Components**, e grazie all'**ereditarietà** tra views, passare variabili e riferimenti in modo facile e veloce. Potrai costruire uno spicchio di interfaccia per ogni sezione del progetto (sidebar, navbar, footer, hero, form di contatto ecc...) e richiamarla in ogni momento portandoti dietro ogni tipo di dato associato.

Di default tutte le views dovrai inserirle nel percorso `resources/views`.

Da qui, verranno inizialmente compilate e, successivamente, memorizzate nella cache di sistema (storage/framework).

Le views non verranno processate e compilate ogni volta, ma semplicemente caricate e lette in modo statico.

Utilizzare Blade sarà un'esperienza fantastica. Anteponendo il simbolo della chiocciola (@), potrai utilizzare tutte le direttive messe a disposizione dal template engine.

Comandi Artisan

Laravel introduce quasi un centinaio di comandi richiamabili da console utilizzando il prefisso `php artisan <comando>`

Potrai pulire la cache dell'applicazione, lanciare la migrazione delle tabelle, avviare il server locale, programmare dei Job o visualizzare in modo tabellare tutte le Routes/API attive.

Il comando php artisan list ti permetterà di vedere tutti i comandi presenti. Potrai realizzarne di nuovi in base alle tue esigenze e una volta memorizzati, non potrai più farne a meno.

Eloquent

Riassumere in poche righe Eloquent è praticamente impossibile. Eloquent è l'ORM (*Object-Relational Mapping*) di Laravel. Implementa il pattern ActiveRecord e introduce una serie di astrazioni a livello Model che ti permetteranno di cambiare "motore" in modo facile e veloce senza riscrivere la logica dell'applicazione.

Potrai gestire tutto senza utilizzare direttamente il linguaggio del database (SQL o NoSQL che sia), o quanto meno limitarlo. Sarà possibile utilizzare query grezze (RAW), ma con un po di pratica potrai limitarti ad usare semplicemente l'ORM.

Ogni tabella "fisica"avrà una corrispettiva tabella "logica" che prenderà il nome di Model.

Facades

Le *facciate* (traduzione letteraria), non sono altro che un'interfaccia semplificata di un sottosistema più complesso. Tramite l'assegnazione di un Aliases, sarà più facile richiamare e ricordare la classe prescelta.

Enviroment (.env)

Ti capiterà di dover cambiare rapidamente alcune impostazioni di ambiente: credenziali del database di riferimento, server di posta SMTP, driver di cache, token o key per servizi di terze parti. **No Panic**!

Per rendere tutto questo facile come bere un bicchiere d'acqua, Laravel implementa la libreria PHP DotEnv.

Avrai a portata di mano qualsivoglia variabili d'ambiente inizializzata nel file, e tutte quelle di default verranno gestite globalmente da Laravel.

Questo significa, ad esempio, che una volta inserite le credenziali di accesso al database, avrai automaticamente tutte le connessioni attive, non dovrai crearti ulteriori file custom per la connessione!

Ad installazione pulita, avrai dei valori di default già impostati. Vai nella root di sistema, e apri il file `/.env`.

Di seguito ho riassunto le principali variabili che utilizzerai, dividendole in variabili APP e DB:

- `APP_NAME=Laravel`: Farà riferimento al nome del progetto. Potrai utilizzare questa variabile in qualsiasi parte del codice utilizzando la sintassi `{{env('APP_NAME')}}` nei file Blade mentre `env('APP_URL')` nelle classi PHP;
- `APP_ENV=local`: Con riferimento all'Environment (Ambiente) potrai definire valori diversi in base al tipo di fase in cui ti troverai (Production, staging...);
- `APP_KEY=base64:23..43`: La chiave cambierà in ogni tipo di installazione e configurazione, è importantissima per il corretto funzionamento di Laravel. Se non dovessi trovarla, ricordati di generarla utilizzando il comando `php artisan key:generate`;
- `APP_DEBUG=true`:In caso sia true visualizzerai la pagina di errore di Ignition.In caso contrario, una semplice pagina 404;
- `APP_URL=http://localhost`: Permette di definire l'url per i callback delle funzioni interne (generazione email di verifica, reset della password, notifiche ecc.);

Per quanto riguarda quelle relative al database, avrai:

- `DB_CONNECTION=mysql`: Specifica il tipo di driver che verrà utilizzato per la connessione al database. Di default ne troverai alcuni, ma nulla ti vieta di inserirne di nuovi;
- `DB_HOST=127.0.0.1`: Se locale lasciare 127.0.0.1 altrimenti l'host remoto dove è ospitato il database;
- `DB_PORT=3306`: La porta della connessione;
- `DB_DATABASE=lptdatabase`: Qui specificherai il nome del database;
- `DB_USERNAME=root`: Username dell'utente root (o un utente abilitato con privilegi);
- `DB_PASSWORD=root`: Password dell'utente root (o un utente abilitato con privilegi);

Capitolo 3 - L'ambiente di Sviluppo

Avrai bisogno di PHP, Composer, MySQL, Node e un Terminale. Se vuoi avvicinarti al mondo Laravel, o ancora più genericamente a PHP, non puoi fare a meno di introdurre **Composer** nei tuoi tool produttivi.

Composer, una volta installato e configurato, ti permetterà di installare e aggiornare tutte le dipendenze PHP del progetto in modo facile e veloce.

Le dipendenze, sotto forma di pacchetti, verranno installate e compilate automaticamente.

I pacchetti che compongono il core di Laravel sono installati con una semplice riga di comando e subito disponibili nel progetto.

Di seguito ti elencherò i due metodi principali per installare questi pre-requsiti, necessari per lo sviluppo con Laravel:

1. Il primo metodo è quello di utilizzare un software auto-installante con un'interfaccia grafica apposita per gestire PHP,MySQl e Apache/Nginx.
2. Il secondo metodo è leggermente più articolato, ma ti permetterà di gestire meglio tutto l'ambiente di sviluppo.

Inizialmente, per prendere confidenza con il Framework, va benissimo anche il primo metodo.

Ma prima di procedere con i due metodi che si occuperanno di installare PHP e MySQL, dovrai necessariamente prepararti il terreno configurando **Node**, un **Terminale** e **Composer**.

Node

L'installazione di Node prevede il download sul sito ufficiale del pacchetto: `https://nodejs.org/it/`

E l'avvio dell'installazione in base al sistema operativo scelto.

Terminale

Non è richiesto alcun terminale particolare.

- Se sei su macchina **Windows** avrai a disposizione il terminale di default, puoi lanciarlo premendo CTRL + SHIFT + ENTER ;
- Se utilizzi **MacOs** puoi utilizzare la ricerca Spotlight per trovare l'applicazione Terminale;
- Utilizzando **Ubuntu**, invece, lo troverai in Applicazioni > Accessori > Terminale;

Composer

Installazione per MAC

Da terminale copia e incolla queste 4 righe di codice pronte all'uso:

```
php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"

php -r "if (hash_file('sha384', 'composer-setup.php') ===
'a5c698ffe4b8e849a443b120cd5ba38043260d5c4023dbf93e1558871f1f07f58274fc6f4
c93bcfd858c6bd0775cd8d1') { echo 'Installer verified'; } else { echo
'Installer corrupt'; unlink('composer-setup.php'); } echo PHP_EOL;"

php composer-setup.php

php -r "unlink('composer-setup.php');"
```

Il processo sarà automatico e potrai verificare la corretta installazione di Composer digitando nel terminale:

`composer -v`

Installazione per Windows

Copia e incolla questo link nel tuo browser e fai partire il download:

`https://getcomposer.org/Composer-Setup.exe`

Scarica l'eseguibile e avvialo. Segui le classiche istruzioni a video e assicurati di installarlo nello stesso disco locale del software scelto.

Installazione Ubuntu

Per prima cosa aggiorna gli indici del tuo sistema operativo:

`sudo apt update`

Assicurati di avere la curly utility installata e lancia quindi in successione:

`sudo apt-get install curl`

e successivamente:

`sudo curl -s https://getcomposer.org/installer | php`

A download finito sposta il file .phar all'interno della directory di sistema:

`sudo mv composer.phar /usr/local/bin/composer`

PHP/MySQL

Installazione incluso

Utilizzare uno dei software presenti qui sotto ti permetterà di installare i vari PHP, Mysql, Apache o Nginx in un colpo solo.

Eccoti un po di software e relativi link:

- Laragon (Windows) [Download] | [Documentazione]

- WAMPSERVER (Windows) [Download] | [Documentazione]
- MAMP (Windows, MAC) [Download] | [Documentazione]
- XAMPP (Windows, MAC, Unix) [Download]| [Documentazione]

Ognuno ha una sua documentazione e peculiarità, ma la sostanza non cambia: avrai una GUI di gestione per avviare server e database con un click; PHP e MySql auto-installati nella cartella del software in questione e una mini console con tutti gli avvisi di errore del server.

Di default, inoltre, i tuoi progetti verranno eseguiti solo se messi nella cartella predefinita del software (www o public_html). Se vorrai modificare la cartella dei progetti, potrai tranquillamente farlo dalle preferenze del software._

Installazione Manuale

Installazione MacOs

1. Avrai bisogno di **Homebrew**, il "composer" per MacOs.
   Per installarlo ti basterà copiare e incollare questa riga sul terminale e seguire le istruzioni da console:
   `/usr/bin/ruby-e"$(curl-fsSL[https://raw.githubusercontent.com/Homebrew/install/master/install](https://raw.githubusercontent.com/Homebrew/install/master/install))"`
2. Ancora, sempre utilizzando il terminale, lancia in successione:
   `brew update`
   `brew upgrade`
3. È il turno di PHP, lancia il comando:
   `brew install php@7.3`
4. Come detto, avrai bisogno di MySQL. Scarica il pacchetto da questo link:
   https://dev.mysql.com/downloads/file/?id=487659
   Decomprimendo il .dmg scaricato ti si aprirà un wizard di installazione.
   A processo finito chiudi la finestra.

Bene, la tua macchina è pronta. Hai appena installato PHP, Composer e MySQL.

IMPORTANTISSIMO: Salva username e password generate con il database per l'utente root!

Installazione Windows

1. Crea una cartella chiamata PHP nel disco che andrai ad utilizzare per i tuoi progetti, ad esempio C:\PHP (per gli esempi successivi utilizzerò questo percorso).
   Copia e incolla questo link e scarica il pacchetto:
   `https://windows.php.net/downloads/releases/php-7.3.27-src.zip`
2. Una volta completato il download, decomprimi tutto il contenuto nella cartella `C:\PHP`
   Avrai diverse cartelle e file binari dall'aspetto terrificante.

Trova il file **php.ini-development** e assicurati di copiarlo e incollarlo nello stesso percorso. Quando il sistema ti chiederà se vorrai rinominarlo, tu accetta e rinominalo come **php.ini**.

3. Per sfruttare tutte le potenzialità di Laravel, dovrai assicurati che queste estensioni siano attive. Per farlo ti basterà rimuovere il punto e virgola all'inizio di ogni riga. Ad esempio:
   `;estensione1 (Estensione non attiva)`
   `estensione1 (Estensione Attiva)`
   Dovrai assicurarti che tutte queste estensioni siano attive:
   - BCMath PHP Extension
   - Fileinfo PHP extension
   - Ctype PHP Extension
   - JSON PHP Extension
   - Mbstring PHP Extension
   - OpenSSL PHP Extension
   - PDO PHP Extension
   - Tokenizer PHP Extension
   - XML PHP Extension

   Nel pannello di controllo, apri la scheda `Sistema/Impostazioni-di-sistema-avanzate/variabili d'ambiente` e scorri fino alla sezione chiamata **Variabili di sistema**.
   Selezionando la riga **PATH**, clicca su **Modifica** e successivamente, nella nuova schermata, su **Nuovo**.Dovrai inserire il percorso dove hai installato PHP in precedenza (C:\PHP).
4. Installato PHP, è il turno di Composer. Copia e incolla questo link nel tuo browser e fai partire il download:
   `https://getcomposer.org/Composer-Setup.exe`
   Avvia l'eseguibile e segui le istruzioni a video.
5. Manca soltanto MySQL. Anche qui, scarica l'eseguibile dal seguente link e segui le istruzioni a schermo:
   `https://dev.mysql.com/downloads/file/?id=485750`Ricordati di memorizzare da qualche parte le credenziali dell'utente root che andrai a creare. Sarà fondamentale per accedere al DB.
6. Riavvia il sistema operativo.
   IMPORTANTISSIMO: Salva username e password generate per l'utente root!

Installazione Ubuntu

1. Avrai bisogno innanzitutto di Apache2. Apri il terminale e incolla i comandi:
   `sudo apt update`
   `sudo apt install apache2`
2. È il turno di PHP. Copia e incolla tutto il seguente comando nella console:

```
sudo apt install php7.3-cli php7.3-fpm php7.3-json php7.3-pdo php7.3-mysql php7.3-zip php7.3-gd php7.3-mbstring php7.3-curl php7.3-xml php7.3-bcmath php7.3-json
```

Hai appena installato PHP 7.3 e tutti i relativi sub-moduli.

3. Finita l'installazione, apri il file php.ini con il comando:
   `sudo nano /etc/php/7.3/apache2/php.ini`
   e modifica le seguenti voci come segue:

```
memory_limit = 256M
upload_max_filesize = 64M
cgi.fix_pathinfo=0```
```

4. Da terminale, copia e incolla il seguente codice per installare MySQL:
   `sudo apt install mysql-server`
5. Infine ricorda di riavviare il server:
   `sudo service apache2 restart`

IMPORTANTISSIMO: Salva username e password generate per l'utente root!

Capitolo 4 - Installazione Laravel

Il framework Laravel può essere utilizzato in diversi modi: attraverso la creazione di un progetto con composer, installandolo globalmente, scaricando un progetto non inizializzato o scaricando un pacchetto già compilato con vendor inclusi.

Laravel Installer

Stai per installare Laravel in tutto il sistema. Potrai richiamarlo in qualsiasi momento e avviare l'installazione del framework utilizzando il comando apposito.

Apri il terminale e digita questo comando:

`composer global require laravel/installer`

Ultimo step è verificare la corretta impostazione del **PATH/Variabili d'ambiente**.
Assicurati di avere il percorso appena creato nelle **Variabili di ambiente** di sistema :

```
MAC: $HOME/.composer/vendor/bin
WINDOWS: %USERPROFILE%\AppData\Roaming\Composer\vendor\bin
GNU/Linux: $HOME/.config/composer/vendor/bin

oppure sempre GNU/LINUX

GNU/Linux:  $HOME/.composer/vendor/bin
```

In qualsiasi momento potrai posizionarti nella cartella dei progetti e lanciare il comando per installare il framework, oppure posizionarti dentro la cartella del singolo progetto e far partire l'installazione:

`laravel new myProject`

Installazione via Composer

Composer permette anche l'installazione diretta. Il comando è molto semplice:

```
composer create-project laravel/laravel nome_progetto

cd nome_progetto
```

Avvia il server

Utilizzando il terminale, entra nella cartella del progetto in modo da lanciare senza grossi problemi:

`php artisan:serve`

Questo comando lancerà il server built-in di laravel.
Ti verrà mostrato un url sulla console del tipo *127.0.0.1:8000* contenente la starter page di Laravel.

FAQ in caso di errori post-installazione

Se hai seguito le istruzioni poco sopra e non sei riuscito nell'impresa, eccoti una FAQ che potrebbe fare al caso tuo.

- **Directory Pubblica**

  Laravel utilizza il file index.php come primo file di lettura , presente in `public/index.php`.

  All'interno di questo index.php c'è lo starter di tutto Laravel, vengono caricati i file del core ed eventuali estensioni aggiunte.
  Normalmente, i server locali o remoti puntano alla più classica root `/index.php`.

  Sarà tua premura modificare il puntamento del file di configurazione del server (solitamente **.htaccess**, **nginx.conf** o **web.config**) alla directory poco fa nominata **public/** .

- **Permessi alle cartelle**

  In caso di problemi o errori di scrittura/lettura di determinate cartelle, dovrai cambiare i permessi delle seguenti cartelle:

  - **storage**;
  - **bootstrap/cache** *(p.s. non ha alcun legame con il Framework CSS Bootstrap!).*

  Laravel immagazzina tutte le sessioni, view, ecc.. nella cartella storage e utilizza bootstrap/cache come directory di sistema per il caching di tutte quelle classi statiche da non richiamare ogni volta.

- **File .env**

  Hai scaricato il progetto da un tutorial online? Sicuramente il file .env non è presente, o quanto meno ci sarà un .env.example . Rinominalo in .env e avrai risolto.
  In alternativa, se dovesse mancare anche il file .env.example, puoi copiare e incollare quello presente nella repository su Github reperibile.

- **Genera la chiave**

  Laravel utilizza una variabile **APPKEY** per gestire tutti i cookies dell'applicazione. In caso di assenza della chiave potrebbe restituirti errori di varia natura.

  Lancia il comando:

  `php artisan key:generate`

  E automaticamente ti verrà generata la chiave nel file .env.

- **URL**

  Ed ecco che entrano in gioco i cosiddetti "Pretty Url" o "Vanity Url" tanto amati dai crawler dei vari motori di ricerca.
  Utilizzare url del tipo:

  `sitoweb.it/index.php`

  `sitoweb.it/utenti/profile.php?id=4`

  Non aiuterà di certo al business.

  Se stai utilizzando un server con Apache dovrai inserire nel .htaccess presente queste regole:

```
Options +FollowSymLinks -Indexes
RewriteEngine On

RewriteCond %{HTTP:Authorization} .
RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]

RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^ index.php [L]
```

In caso di server Nginx, invece, un semplice:

```
location / {
    try_files $uri $uri/ /index.php?$query_string;
}
```

Il risultato sarà che al posto di:

`sitoweb.it/index.php`

`sitoweb.it/utenti/profile.php?id=4`

Otterrai un più sexy:

`sitoweb.it`

`sitoweb.it/utenti/francscomansi`

La struttura interna

Eccoti una breve descrizione delle cartelle e file che più utilizzerai spesso nei tuoi progetti:

- **app**: Ci troverai i Controller, Middleware, i Models e i service provider di sistema.
- **config:** All'interno sono presenti tutti i file di configurazione del framework o di eventuali pacchetti aggiuntivi: Lingua, Timezone, driver di posta, cartella pubblica ecc...
- **database:** La directory principale che ti permetterà di lavorare con: Migrations, Factories e Seeds.
- **public:** Inizialmente ci troverai index.php, htaccess, robots.txt e la favicon. Qui, in un secondo momento, andrai ad inserire i file css e javascript.
- **resources:** La cartella su cui lavorerai maggiormente. Qui dentro sono contenute le views, i file di traduzione e gli assets js e css da implementare.
- **routes:** L'instradamento del progetto passa da qui. Troverai web.php e api.php per gestire gli url e le API; ma anche channels.php e console.php per aiutarti nello sviluppo.
- **storage:** Contiene i file di sessione, la cache, i log e tutti i file generati da Laravel.
- **.env o .env.example:** Soltanto il file .env verrà eseguito da Laravel. La differenza sostanziale è che utilizzando il sistema di versionamento GIT, il file .env non viene condiviso ma ignorato. Rinominandolo .env.example (oppure .env.production), invece, potrai condividere le tue

impostazioni anche fuori dal tuo ambiente locale. Il tuo collega dovrà semplicemente rinominare il file e usarlo.

- **composer.json:** Come dice il nome stesso, è il file di composer. Dentro ci sono tutte le dipendenze da installare per il corretto funzionamento.
- **package.json:** Come sopra, cambia solo l'interprete. Potrai includere tutti i package di NPM per migliorare il frontend del tuo progetto.

Configurazione del database

Laravel utilizza il file `.env` per definire tutte quelle variabili poi processate nel core.
Eccoti il file al momento dell'installazione, ho commentato i punti che affronteremo nel corso di questa guida:

Variabili di sistema

`APP_NAME=Laravel`
Nome dell'applicazione, utilizza i doppi apici per parole composte 'App Laravel'

`APP_ENV=local`
Oltre local la variabile può assumere i valori production e testing

`APP_KEY=`
Se non presente va generata con il comando artisan php artisan key:generate

`APP_DEBUG=true`
Se false non mostra la pagina di Ignition relativa agli errori

`APP_URL=http://localhost`
Url dell'applicazione, importante quando vengono generati gli url

Connessione Database

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

Impostazione Server SMTP

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

```
MAIL_FROM_ADDRESS=null
MAIL_FROM_NAME="${APP_NAME}"
```

Neptune: Crea il Progetto

Apri il terminale e posizionati con i comandi bash nella cartella in cui vorrai installare il tuo progetto.

Per lanciare l'installazione vera e propria utilizzeremo il crete-project di composer:

```
composer create-project --prefer-dist laravel/laravel neptune
```

Laravel verrà scaricato, nel percorso che hai scelto, nella sua interezza in pochissimo tempo. Non dovrai fare nient'altro.
Finita l'installazione potrai già accedere alla Starter Page digitando in console:

`php artisan serve`

Comparirà nel terminale l'indirizzo root del progetto. Apri il tuo browser preferito e copia e incolla l'url appena apparso nel terminale:

`http://127.0.0.1:8000`

```
→  neptune php artisan serve
Starting Laravel development server: http://127.0.0.1:8000
```

Scaffolding di autenticazione

Con Laravel 8 ci sono state offerte 3 tipologie di Scaffolding:

Laravel Breeze

Laravel Fortify

Laravel Jetstream

- **Laravel Breeze**: La mia scelta per questa guida. Ti permetterà di vedere come funziona il framework concentrandoti solo su Laravel. Implementa in modo rapido Login, registrazione, recupero password e reset.
- **Laravel Fortify**: La versione evoluta di Breeze ma senza frontend. Avrai a disposizione un sistema di autenticazione più articolato e complesso ma senza una vera e propria interfaccia grafica;
- **Laravel Jetstream**: Utilizzare Jetstream comporta una conoscena pregressa di Livewire o Inertia. Non appena il tuo livello di conoscenza del frameowrk sarà medio-alto diventerà un valore aggiunto. Con un semplice comando Artisan avrai già configurato un sistema SaaS a tutti gli effetti con ruoli, permessi, log di accesso e Team. Un vero goiello tenendo conto che è rilasciato gratuitamente.

Ma prima di procedere con Laravel Breeze, è necessario impostare il db e generare fisicamente le tabelle.
Apri il file .env e compila la parte relativa alla connessione mysql. In particolare il nome del DB e le credenziali per accedere.
Ecco la mia configurazione locale:

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

Prima di lanciare le migration è necessario fare una piccola modifica alla tabella User.

Apri il file:

`database/migrations/2014_10_12_000000_create_users_table.php`

e aggiungi un nuovo campo per gestire l'immagine profilo, in questo modo:

```
//database/migrations/2014_10_12_000000_create_users_table.php

public function up()
    {
        Schema::create('users', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('email')->unique();
            $table->timestamp('email_verified_at')->nullable();
                 //Aggiungi qui il campo avatar
            $table->string('avatar')->nullable();
            $table->string('password');
            $table->rememberToken();
            $table->timestamps();
        });
    }
```

A Migration finita è il turno di Laravel Breeze. Installa il pacchetto composer come segue:

`composer require laravel/breeze --dev`

Il download e lo spacchettamento sarà molto rapido. Una volta completato il processo puoi lanciare il comando artisan appena abilitato:

`php artisan breeze:install`

Breeze utilizza **Tailwindcss** e **Alpine.js** per generare lo scaffolding del frontend, lancia quindi in successione:

```
npm install && npm run dev
```

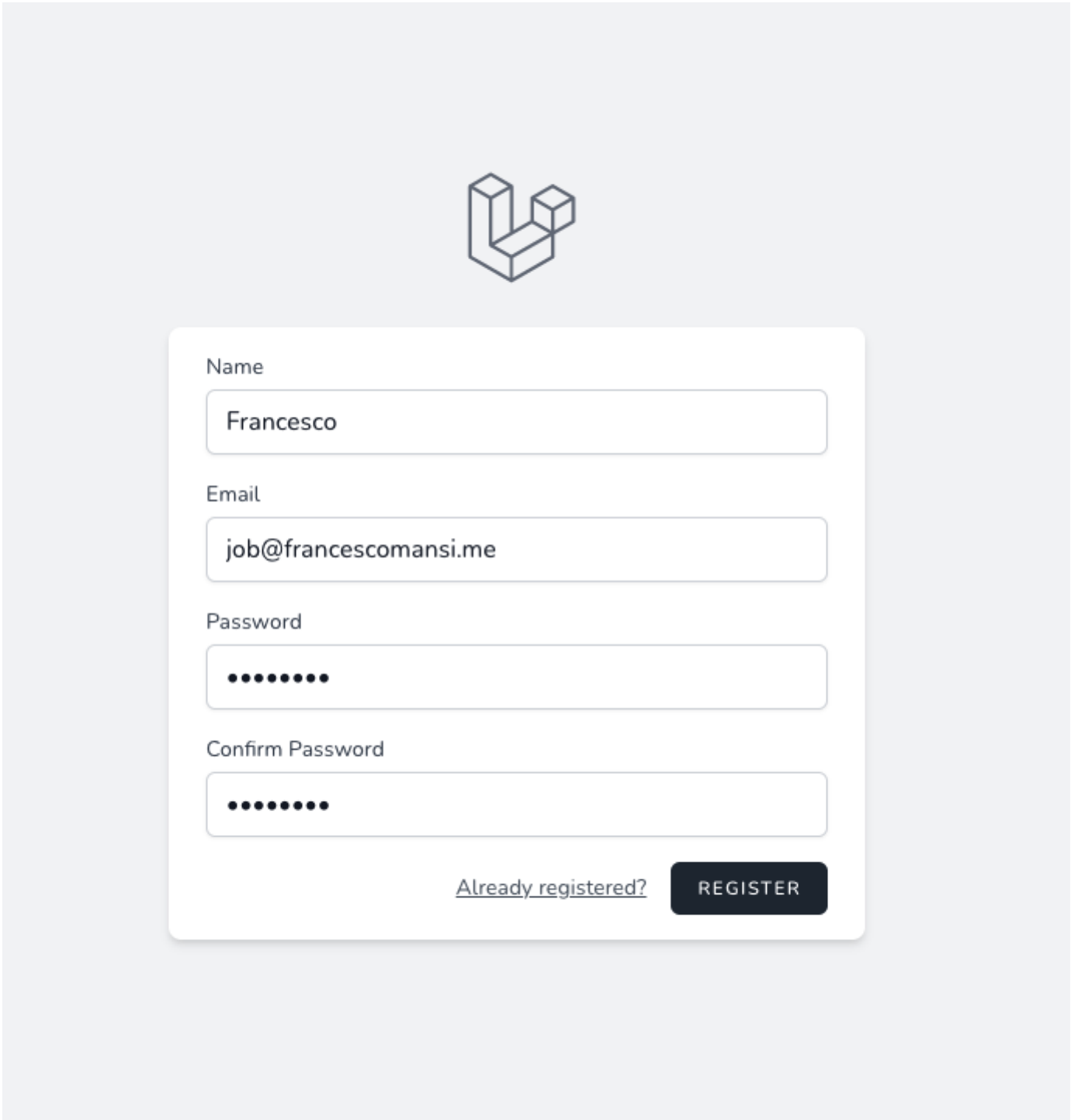
Adesso lancia le migration con il comando:

```
php artisan migrate
```

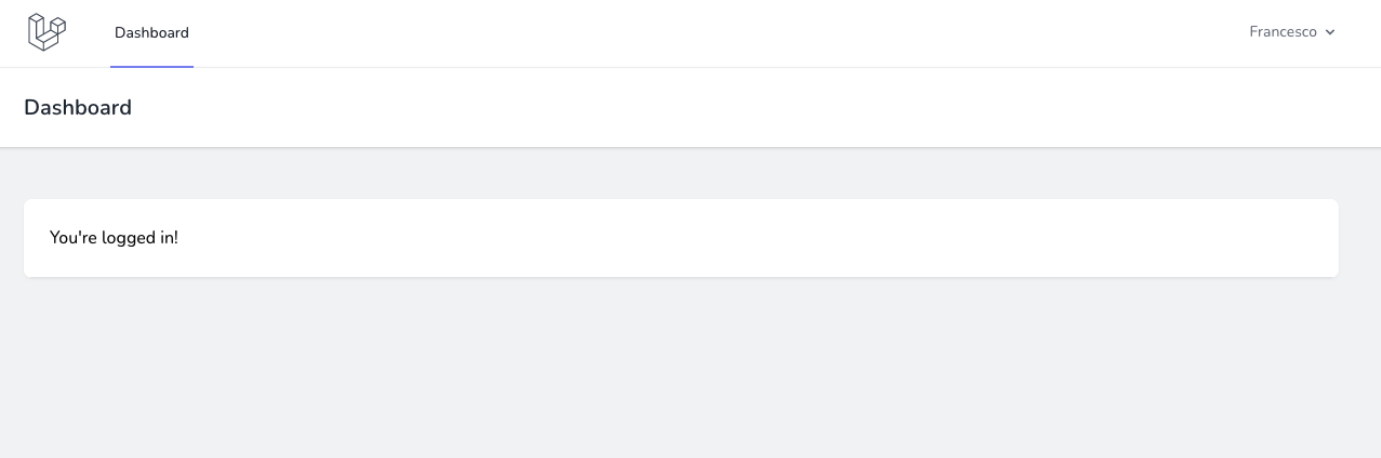
Ebbene si, hai gia configurato il sistema di autenticazione e puoi già provarlo.
Cliccando sul link in alto a destra **register** verrai dirottato su:

```
http://127.0.0.1:8000/register
```

Compila i campi a concludi la registrazione con il tasto **Register**:



Verrai autenticato automaticamente e reindirizzato nella tua dashboard personale:



Navigation

Prendiamo in esame il file blade:

`resources/views/layouts/navigation.blade.php.`

Il file navigation si occupa di gestire la barra superiore dell'interfaccia, sia mobile che responsive.
I componenti che si occupano di generare i link sono due: **nav-link.blade.php** e **responsive-nav-link.blade.php**.

Andremo ad aggiungere una nuova voce al menu e sarà "Utenti".
Aggiungi quindi la nuova voce utilizzando i due componenti, in questo modo:

```
// resources/views/layouts/navigation.blade.php

<!-- Navigation Links -->
  <div class="hidden space-x-8 sm:-my-px sm:ml-10 sm:flex">
    <x-nav-link :href="route('dashboard')" :active="request()->routeIs('dashboard')">
        {{ __('Dashboard') }}
    </x-nav-link>
    <x-nav-link :href="route('users.index')" :active="request()->routeIs('users.*')">
        {{ __('Utenti') }}
    </x-nav-link>
 </div>
```

E la parte relative alla versione mobile sempre nello stesso file:

```
// resources/views/layouts/navigation.blade.php

<!-- Responsive Navigation Menu -->
<div class="pt-2 pb-3 space-y-1">
  <x-responsive-nav-link :href="route('dashboard')" :active="request()->
```

```
>routeIs('dashboard')">
      {{ __('Dashboard') }}
  </x-responsive-nav-link>
  <x-responsive-nav-link :href="route('users.index')" :active="request()-
>routeIs('users.*')">
      {{ __('Utenti') }}
  </x-responsive-nav-link>
</div>
```

Prima di vedere il risultato sul browser ti anticipo subito cosa sta per accadere: l**a pagina genererà un errore di Route non trovata**.
Ebbene si, utilizzando l'helper **route('dashboard')** stiamo passando a Laravel una variabile non definita.

Niente paura, nel prossimo capitolo vedremo come definire la route.

Routes: Aggiungere una Rotta

Se hai già avuto a che fare con Laravel in passato conoscerai, sicuramente il suo decantato sistema di routing.

Nelle ultime due major release (Laravel 7 e 8) ha subito un potente restyling dal punto di vista sintattico e di performance.
Una cosa non è cambiata però: la semplicità di utilizzo.

Aprendo il file **routes/web.php** avrai modo di constatare tu stesso che esistono già alcune route relative alla root del sito e alla dashboard.

Per il momento ci limiteremo a ricopiare la route già esistente della dashboard facendola ritornare ad una semplice view. Più avanti vedremo come utilizzare i Controller.

Non è tutto. Per sfruttare l'helper **route()** avremo bisogno di denominare la rotta appena creata. Ah, e visto che la lista degli utenti è comunque un dato sensibile, laravel ci regala già un sistema di protezione, andremo ad aggiungere anche un middleware:

```
//routes/web.php

Route::get('/dashboard', function () {
    return view('dashboard');
})->middleware(['auth'])->name('dashboard');

Route::get('/users', function () {
   return view('sections.users.index');
})->middleware(['auth'])->name('users.index');
```

Ecco fatto.
Al momento cliccando sul link **Utenti** vedremo la classica pagina di errore di Ignition: non esiste (ancora) alcun file blade **sections.users.index** .

Nel prossimo capitolo vedremo come creare un file blade.

Blade: Creare una nuova pagina

Crea una nuova cartella chiamata **"sections"** all'interno di **resources/view**. Andremo a posizionare tutti i file blade delle sezioni del backend.

Dentro la cartella appena creata *sections*, come una matrioska, creane ancora una: questa volta chiamala **"users"**. La struttura che avrai sarà più o meno questa:

```
resources/
└── views/
      ├── components/
      ├── layouts/
      └── sections/
            └── users/
```

Per il momento andremo a creare soltanto il file blade index per visualizzare la lista degli utenti, così:
`resources/views/sections/users/index.blade.php`

Dividendo il progetto in piccoli componenti atomici potrai riutilizzarli a tuo piacimento.
Non dovrai creare alcun layout o configurare pagine HTML standard, ci sono già.

Il layout principale da utilizzare, infatti, è quello di Breeze:

`resources/views/layouts/app.blade.php`

Incolla questa porzione di codice nel file index:

```
//resources/views/sections/users/index.blade.php
<x-app-layout>
 <x-slot name="header">
     <h2 class="text-xl font-semibold leading-tight text-gray-800">
         {{ __('Utenti') }}
     </h2>
 </x-slot>

 <div class="py-12">
     <div class="flex flex-wrap mx-auto max-w-7xl sm:px-6 lg:px-8">

     </div>
 </div>
</x-app-layout>
```

Accedendo all'url `http://127.0.0.1:8000/users` ti ritroverai con una pagina vuota e la sola scritta “Utenti”.
L’obiettivo di questa pagina sarà quello di mostrare la lista degli utenti registrati.
Sempre utilizzando Tailwindcss, eccoti una tabella creata ad hoc per l'occasione:

```
//resources/views/sections/users/index.blade.php

<x-app-layout>
 <x-slot name="header">
     <h2 class="text-xl font-semibold leading-tight text-gray-800">
         {{ __('Utenti') }}
     </h2>
 </x-slot>

 <div class="py-12">
     <div class="flex flex-wrap mx-auto max-w-7xl sm:px-6 lg:px-8">
<div class="container flex flex-wrap mx-auto md:p-4 ">
  <div class="inline-block min-w-full px-1 overflow-hidden align-middle
md:shadow-lg md:px-0">
      <table class="min-w-full shadow">
          <thead>
          <tr>
              <th class="w-3/6 px-6 py-3 text-xs font-medium leading-4
tracking-wider text-left text-gray-500 uppercase bg-white border-b border-
gray-200 md:w-2/5">
                 Utenti
              </th>
              <th class="hidden w-1/6 px-6 py-3 text-xs font-medium
leading-4 tracking-wider text-left text-gray-500 uppercase bg-white
border-b border-gray-200 sm:table-cell">
                 Stato
              </th>
              <th class="w-1/6 py-3 bg-white border-b border-gray-200
md:text-right md:px-3 md:w-2/12 sm:table-cell">
                <a href="{{route('users.create')}}" class="inline-flex
items-center h-full px-3 py-2 leading-tight text-white bg-green-400 border
border-green-400 rounded-lg appearance-none focus:outline-none">
                    <span class="block text-sm">Nuovo</span>
                    <svg class="h-5 pl-2"
xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24"
stroke="currentColor">
                        <path stroke-linecap="round" stroke-
```

```
linejoin="round" stroke-width="2" d="M12 9v3m0 0v3m0-3h3m-3 0H9m12 0a9 9 0 11-18 0 9 9 0 0118 0z" />
                    </svg>
                </a>
            </th>
        </tr>
        </thead>
        <tbody class="bg-white">
            @forelse($data['users'] as $user)
                <tr class="cursor-pointer hover:bg-gray-100">
                    <td class="px-4 py-4 border-b border-gray-200 md:whitespace-no-wrap">
                        <div class="flex items-center">
                            <div class="flex-shrink-0 w-10 h-10">
                                <img class="w-10 h-10 rounded-full" src="{{$user->avatar ? $user->avatar: 'https://eu.ui-avatars.com/api/?name='.$user->name}}" alt=""/>
                            </div>
                            <div class="ml-4">
                                <div class="text-sm font-medium leading-5 text-gray-900">{{$user->name}}</div>
                                <div class="text-sm leading-5 text-gray-500 md:block">{{$user->email}}</div>
                            </div>
                        </div>
                    </td>
                    <td class="hidden px-4 py-4 whitespace-no-wrap border-b border-gray-200 sm:table-cell">
                        <span class="inline-flex px-2 text-xs font-semibold leading-5 @if($user->email_verified_at) text-green-800 bg-green-100 @else bg-yellow-100 text-yellow-800 @endif rounded-full">
                            <span class="hidden sm:block">@if($user->email_verified_at)  {{ __('Attivo') }} @else  {{ __('In Attesa') }} @endif</span>
                        </span>
                    </td>
                    <td class="px-2 py-4 text-sm font-medium leading-5 text-right border-b border-gray-200 md:px-6 md:whitespace-no-wrap">
                        <div class="flex justify-end space-x-1">
                            <a href="{{route('users.edit',$user->id)}}" class="p-1 ml-3 border-2 border-indigo-200 rounded-md hover:bg-indigo-100">
                                <svg
```

```
xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24"
stroke="currentColor" class="w-4 h-4 text-indigo-500">
                                        <path stroke-linecap="round"
stroke-linejoin="round" stroke-width="2" d="M15.232 5.232l3.536 3.536m-
2.036-5.036a2.5 2.5 0 113.536 3.536L6.5 21.036H3v-3.572L16.732 3.732z">
</path>
                                    </svg>
                                </a>
                                <form action="{{route('users.destroy',
$user->id)}}" method="POST" class="">
                                    @csrf
                                    @method('DELETE')
                                    <span class="">
                                        <button class="p-1 border-2
border-red-200 rounded-md hover:bg-red-100">

                                            <svg
xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24"
stroke="currentColor" class="w-4 h-4 text-red-500">
                                                <path stroke-
linecap="round" stroke-linejoin="round" stroke-width="2" d="M19 7l-.867
12.142A2 2 0 0116.138 21H7.862a2 2 0 01-1.995-1.858L5 7m5 4v6m4-6v6m1-
10V4a1 1 0 00-1-1h-4a1 1 0 00-1 1v3M4 7h16"></path>
                                            </svg>
                                        </button>
                                    </span>
                                </form>
                            </div>
                        </td>
                    </tr>
                @empty
                    <tr>
                      <td class="px-6 py-4 text-center whitespace-no-wrap
border-b border-gray-200" colspan="3">
                            <div class="flex flex-col">
                                <p class="py-5 text-sm font-medium
leading-4 text-center text-gray-700 md:text-xl">Nessun Risultato</p>
                            </div>
                        </td>
                    </tr>
                @endforelse
            </tbody>
        </table>
```

```
      @if($data['users']->hasPages() )
            {{ $data['users']->links('vendor.pagination.tailwindcss')}}
      @else
            <div class="flex items-center justify-between px-4 py-3 mb-10
bg-white rounded-b-lg shadow sm:px-6 sm:mb-0">
                  <div class="hidden h-4 sm:flex-1 sm:flex sm:items-center
sm:justify-between"></div>
            </div>
      @endif
            </div>
        </div>
        </div>
 </div>
</x-app-layout>
```

Un piccolo trucco per gestire dinamicamente delle collections/array è quello di non utilizzare il classico foreach ma bensì il forelse.

La spettacolare direttiva Blade **@forelse** (invece del tipico **@foreach**) ti permetterà di gestire automaticamente array vuoti senza dover fare controlli ulteriori per verificare la presenza o meno degli elementi:

```
@forelse($users as $user)
    //Se array utenti pieno
@empty
//Se array utenti vuoto
@endforeelse
```

In fondo alla pagina c'è una piccola struttura predefinita

```
@if($data['users']->hasPages() )
   {{ $data['users']->links('vendor.pagination.tailwindcss')}}
@else
 <div class="flex items-center justify-between px-4 py-3 mb-10 bg-white
rounded-b-lg shadow sm:px-6 sm:mb-0">
      <div class="hidden h-4 sm:flex-1 sm:flex sm:items-center sm:justify-
between"></div>
 </div>
@endif
```

Anche in questo caso, la pagina genererà diversi errori. Nel prossimo capitolo ti spiegherò come risolverli.

Resource Controller: Operazioni CRUD

Per CRUD si fa riferimento all'acronimo di Create Read Update e Delete, le 4 operazioni alla base di una qualsiasi piattaforma web.

Laravel offre la possibilità di utilizzare un **Resource Controller** con tutte le altre funzioni standard per le classiche attività CRUD:

- Index;
- Create;
- Store;
- Show;
- Edit;
- Update;
- Destroy.

Lancia quindi il comando:

`php artisan make:controller UserController --resource`

Verrà creato un nuovo file .php in:

`app/Http/Controllers/UserController.php`

```
//app/Http/Controllers/UserController.php

    public function index()
    {
        //
    }


    public function create()
    {
        //
    }


    public function store(Request $request)
    {
        //
    }


    public function show($id)
    {
        //
```

```
    }

    public function edit($id)
    {
        //
    }


    public function update(Request $request, $id)
    {
        //
    }


    public function destroy($id)
    {
        //
    }
```

Adesso hai 7 nuove Route pronte per essere richiamate nel file **routes/web.php**, ma per non dover scriverle tutte e 7 diverse ti svelerò un piccolo segreto.

Se ricordi hai già avuto modo di scrivere all'interno del file *routes/web.php*, in quell'occasione avevamo definito soltanto la route index.

Esiste una facade che puoi richiamare nel file di route che si occuperà di indirizzare nel modo corretto le chiamate.

```
// routes/web.php

use Illuminate\Support\Facades\Route;
//Importa il controller UserController
use App\Http\Controllers\UserController;

Route::get('/dashboard', function () {
    return view('dashboard');
})->middleware(['auth'])->name('dashboard');


/* Vecchia route
Route::get('/users', function () {
    return view('sections.users.index');
})->middleware(['auth'])->name('users.index');
*/
```

```
Route::resource('users', UserController::class)->middleware(['auth']);
```

(Ricorda da importare il controller).
Ed eccole quindi le nuove route abilitate:

```
| POST   | users           | users.store  |
App\Http\Controllers\UserController@store  |
| GET    | users           | users.index  |
App\Http\Controllers\UserController@index  |
| GET    | users/create    | users.create |
App\Http\Controllers\UserController@create |
| PUT    | users/{id}      | users.update |
App\Http\Controllers\UserController@update |
| GET    | users/{id}      | users.show   |
App\Http\Controllers\UserController@show   |
| DELETE| users/{id}      | users.destroy|
App\Http\Controllers\UserController@destroy|
| GET    | users/{id}/edit | users.edit   |
App\Http\Controllers\UserController@edit   |
```

Il tuo occhio attento avrà sicuramente notato che oltre i comuni metodi HTTP disponibili (POST e GET) sono presenti anche **PUT**, **PATCH** e **DELETE**.

Per utilizzare questi metodi dovrai utilizzare dei campi hidden all'interno del form, così:

```
<form action="" method="POST">
   <input type="hidden" name="_method" value="put" />
   <input type="hidden" name="_method" value="patch" />
   <input type="hidden" name="_method" value="delete" />
</form>
```

Oppure, il metodo che preferisco, utilizzando la direttive Blade **@method()** in questo modo:

```
<form action="" method="POST">
   @method('PUT')

   @method('PATCH')

   @method('DELETE')
</form>
```

Chiusa la parentesi teorica sul Resource Controller, riprendiamo il discorso pratico.

Essendo Laravel un MVC, prima di pensare al Controller ti saresti dovuto occupare di creare un Model. Ma ho una bella notizia per te.

Il model per gli utenti è già presente nell'installazione standard, lo puoi trovare in

`app/Models/User.php`

Nel primo capitolo abbiamo aggiunto il campo Avatar alla tabella utenti, ricordi?
Per poter sfruttare l'inserimento massivo più avanti dovrai inserirlo all'interno dell'array **$fillable** in questo modo:

```php
<?php
//app/Models/User.php

 protected $fillable = [
     'name',
     'email',
     'password',
 ];
```

Ritorniamo a **UserController**.
Per poter usare il Model dovrai semplicemente ricordarti di importarlo sotto al namespace.

Incolla questo codice PHP relativo alla funzione index:

```php
<?php
//app/Http/Controllers/UserController.php

namespace App\Http\Controllers;

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

class UserController extends Controller
{
   public function index()
   {
        $users = User::latest()->paginate(10);

        $data = [
            'users' =>  $users,
        ];
        return view('sections.users.index')->with('data', $data);
   }
}
```

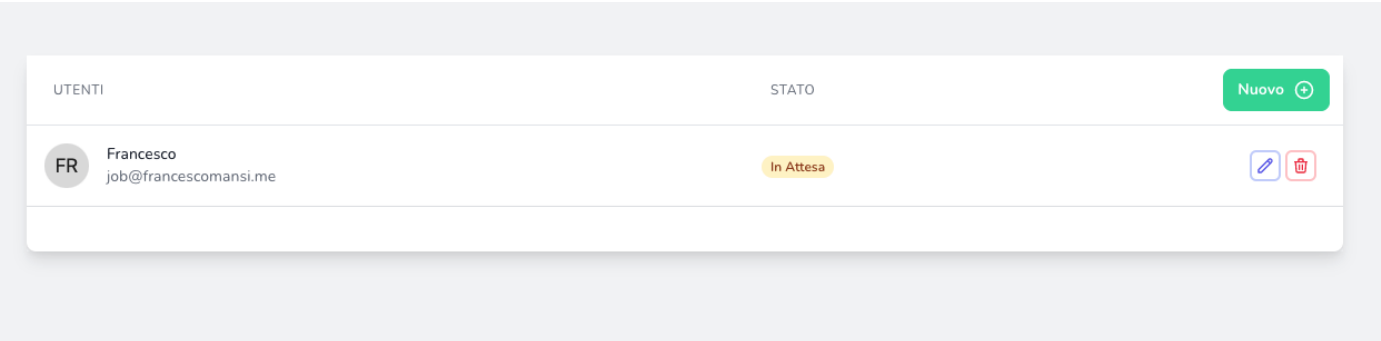
Procediamo per gradi.
Con la riga `$users = User::latest()->paginate(10);` stai assegnando alla variabile **$users** tutti i record presenti all'interno del model, ordinandoli dal più recente (**latest()**) e impostando una paginazione di 10 elementi (**paginate(10)**).

```
$data = [
  'users' =>  $users,
];
```

La variable **$data** sarà un contenitore con all'interno tutte le variabili da trasportare nella view.

Per il momento sarà solo una, ma ti renderai conto che man mano il progetto diventerà più complesso, l'utilizzo di una variabile di trasporto sarà molto utile.

Ritornando alla pagina utenti non vedrai più alcun errore, ma la tabella con il solo utente per il momento inserito.



Nell'interfaccia appena visualizzata ci sono 3 pulsanti:

- Create
- Edit
- Delete

Al momento nessuno di questi darà il risultato sperato. Nel prossimo capitolo vedremo come farli funzionare.

Edit

Quando si procede alla modifica di un record è fondamentale passare il parametro identificativo corretto nell'url. Niente paura, con il codice di poco fa è gia tutto configurato.

Riprendendo il template HTML contenente la tabella degli utenti, soffermati sul collegamento del link edit qui presente:

```
//resources/views/sections/users/edit.blade.php

href="{{route('users.edit',$user->id)}}"
```

L'helper **route()** consente di dichiarare parametri supplementari oltre che il nome della Route. Nella barra degli url avrai una cosa del genere:

`http://127.0.0.1:8000/users/1/edit`

Questo perchè utilizzando la direttiva *Route::resource*, se ricordi la tabella con tutte le route, l'url relativo alla modifica è così composto:

```
users/{id}/edit
```

dove appunto **{id}** è la variabile definita nel secondo parametro di route **$user->id**.

Crea un nuovo file blade:

`resources/views/sections/users/edit.blade.php`

E incollaci il layout base:

```
//resources/views/sections/users/edit.blade.php
<x-app-layout>
 <x-slot name="header">
     <h2 class="text-xl font-semibold leading-tight text-gray-800">
         {{ __('Modifica Utente') }}
     </h2>
 </x-slot>

 <div class="py-12">
     <div class="flex flex-wrap mx-auto max-w-7xl sm:px-6 lg:px-8">

     </div>
 </div>
</x-app-layout>
```

la view è stata creata, adesso è il turno della relative funzione nel controller:

```
//app/Http/Controllers/UserController.php
public function edit($id)
  {
      $user = User::findOrFail($id);
      $data = [
          'user' => $user
      ];
      return view('sections.users.edit')->with('data', $data);
   }
```

Quando cliccherai su modifica, invierai in GET l'id relativo all'utente selezionato (**$id** nel controller). Ti servirà per cercare all'interno della tabella User l'utente corretto.

Inizia a pensare come uno sviluppatore PRO: il form di modifica è un elemento potenzialmente ridondante:

Deve essere un componente.

Con l'installazione di Laravel Breeze hai ricevuto in "dono" dal team Laravel anche altri componenti basilari per la costruzione di un form:

- `resources/views/components/label.blade.php`
- `resources/views/components/input.blade.php`

Li utilizzeremo per snellire ancora di più il codice, senza scrivere le stesse classi e direttive N volte.

Il codice che sto per proporti potrà sembrarti a primo impatto piuttosto contorto: un componente form con all'interno altri componenti input e label.

Inoltre utilizzando la direttiva **{{ $attributes->merge() }}** andrai a passare i vari attributi relativi al method, name e id direttamente nello **{{ $slot }}** del componente stesso.

Crea quindi un file users-form come segue:

`resources/views/components/users-form.blade.php`

E impostiamo il layout seguente nel componente appena creato:

```
//resources/views/components/users-form.blade.php
@props(['user'])
<form {{ $attributes->merge() }}>
    {{ $slot }}
<div class="border-b">
    <div class="flex items-center justify-end mr-5 md:justify-between">
        <div class="flex-1 block min-w-0">
            <div class="flex flex-col p-6">
                <ol class="inline-flex p-0 list-none">
                    <li class="flex items-center">
                        <a href="{{ route('users.index') }}"
                           class="font-semibold leading-relaxed text-gray-
600"> {{ __('Utenti') }}</a>
                    </li>
                    @isset($user)
                    <li class="flex items-center">
                        <svg class="w-3 h-3 mx-3 text-gray-600 fill-
current" xmlns="http://www.w3.org/2000/svg" viewBox="0 0 320 512">
                            <path  d="M285.476 272.971L91.132 467.314c-
9.373 9.373-24.569 9.373-33.941 0l-22.667-22.667c-9.357-9.357-9.375-
24.522-.04-33.901L188.505 256 34.484 101.255c-9.335-9.379-9.317-24.544.04-
```

```
33.901l22.667-22.667c9.373-9.373 24.569-9.373 33.941 0L285.475
239.03c9.373 9.372 9.373 24.568.001 33.941z"/>
                        </svg>
                        <span class="font-semibold leading-relaxed text-
black ">{{$user->name}}</span>

                    </li>
                    @else
                    <li class="flex items-center">
                        <svg class="w-3 h-3 mx-3 text-gray-600 fill-
current"
                        xmlns="http://www.w3.org/2000/svg"
                        viewBox="0 0 320 512">
                       <path
                           d="M285.476 272.971L91.132 467.314c-9.373
9.373-24.569 9.373-33.941 0l-22.667-22.667c-9.357-9.357-9.375-24.522-.04-
33.901L188.505 256 34.484 101.255c-9.335-9.379-9.317-24.544.04-
33.901l22.667-22.667c9.373-9.373 24.569-9.373 33.941 0L285.475
239.03c9.373 9.372 9.373 24.568.001 33.941z"/>
                    </svg>
                        <span class="font-semibold leading-relaxed text-
gray-600"> {{ __('Nuovo Utente') }}
                        </span>
                    </li>
                    @endisset
                </ol>
            </div>
        </div>
        <div class="py-5">
          <span class="ml-3 rounded-md shadow-sm">
            <button type="submit"
                    class="inline-flex items-center px-4 py-2 text-sm
font-medium leading-5 text-gray-700 transition duration-150 ease-in-out
bg-green-300 border border-green-100 rounded-md hover:bg-green-400
hover:text-gray-500 focus:outline-none focus:shadow-outline-blue
focus:border-blue-300 active:text-gray-800 active:bg-gray-50">
                <svg fill="currentColor" class="w-5 h-5 text-green-100
fill-current"
                  viewBox="0 0 20 20">
                  <path fill-rule="evenodd"
                        d="M6.267 3.455a3.066 3.066 0 001.745-.723 3.066
3.066 0 013.976 0 3.066 3.066 0 001.745.723 3.066 3.066 0 012.812
2.812c.051.643.304 1.254.723 1.745a3.066 3.066 0 010 3.976 3.066 3.066 0
```

```
00-.723 1.745 3.066 3.066 0 01-2.812 2.812 3.066 3.066 0 00-1.745.723
3.066 3.066 0 01-3.976 0 3.066 3.066 0 00-1.745-.723 3.066 3.066 0 01-
2.812-2.812 3.066 3.066 0 00-.723-1.745 3.066 3.066 0 010-3.976 3.066
3.066 0 00.723-1.745 3.066 3.066 0 012.812-2.812zm7.44 5.252a1 1 0 00-
1.414-1.414L9 10.586 7.707 9.293a1 1 0 00-1.414 1.414l2 2a1 1 0 001.414
0l4-4z"
                        clip-rule="evenodd">
                    </path>
                </svg>
            </button>
          </span>
        </div>
    </div>
</div>
<div class="p-6">
    <div class="mb-5 -mt-1 text-center">
        <div class="relative w-32 h-32 mx-auto mb-2 bg-gray-100 border
rounded-full">
            <img alt="" id="cover" class="object-cover w-full h-32
rounded-full" src="@isset($user->avatar)/{{$user->avatar}} @else
https://eu.ui-avatars.com/api/?name={{isset($user->name) ? $user->name :
'Nuovo Utente' }} @endisset">
        </div>
        <label for="uploadcover" class="items-center justify-between px-4
py-2 font-medium text-left text-gray-600 bg-white border rounded-lg
shadow-sm cursor-pointer inine-flex focus:outline-none hover:bg-gray-100">
            <svg xmlns="http://www.w3.org/2000/svg" class="inline-flex
flex-shrink-0 w-6 h-6 mr-1 -mt-1"
                viewBox="0 0 24 24" stroke-width="2"
stroke="currentColor" fill="none"
                stroke-linecap="round" stroke-linejoin="round">
                <rect x="0" y="0" width="24" height="24" stroke="none">
</rect>
                <path
                    d="M5 7h1a2 2 0 0 0 2 -2a1 1 0 0 1 1 -1h6a1 1 0 0 1 1
1a2 2 0 0 0 2 2h1a2 2 0 0 1 2 2v9a2 2 0 0 1 -2 2h-14a2 2 0 0 1 -2 -2v-9a2
2 0 0 1 2 -2"></path>
                <circle cx="12" cy="13" r="3"></circle>
            </svg>
           {{ __('Carica Avatar') }}
        </label>
        <input name="avatar" id="uploadcover" accept="image/*"
               onchange="document.getElementById('cover').src =
```

```
window.URL.createObjectURL(this.files[0])"
                class="hidden" type="file">
                @error('avatar')
                <p class="mt-4 text-xs italic text-red-500">
                    {{ $message }}
                </p>
                @enderror
    </div>
    <div class="mb-6 md:flex md:items-center">
        <div class="md:w-1/3">
            <x-label class="px-1 text-sm text-gray-600 md:flex md:justify-
end md:px-5" for="name" :value="__('Username')" />
        </div>
        <div class="md:w-2/3">
             <div>
                <x-input type="text"
                id="name"
                class="block w-full md:w-3/5 "
                name="name"
                :value="$user->name ?? old('name')"
                autofocus />
            </div>
        </div>
    </div>
    <div class="mb-6 md:flex md:items-center">
        <div class="md:w-1/3">
            <x-label class="px-1 text-sm text-gray-600 md:flex md:justify-
end md:px-5" for="email" :value="__('Email')" />
        </div>
        <div class="md:w-2/3">
            <div>
                <x-input type="email"
                id="email"
                class="block w-full md:w-3/5"
                name="email"
                :value="$user->email ?? old('email')"
                autofocus />
            </div>
        </div>
    </div>
    <div class="mb-6 md:flex md:items-center">
        <div class="md:w-1/3">
            <x-label class="px-1 text-sm text-gray-600 md:flex md:justify-
```

```
end md:px-5" for="password" :value="__('Password')" />
        </div>
        <div class="md:w-2/3">
            <div>
                <x-input type="password"
                id="password"
                class="block w-full md:w-3/5"
                name="password"
                :value="old('password')"
                autofocus />
            </div>
        </div>
    </div>
</div>
</form>
```

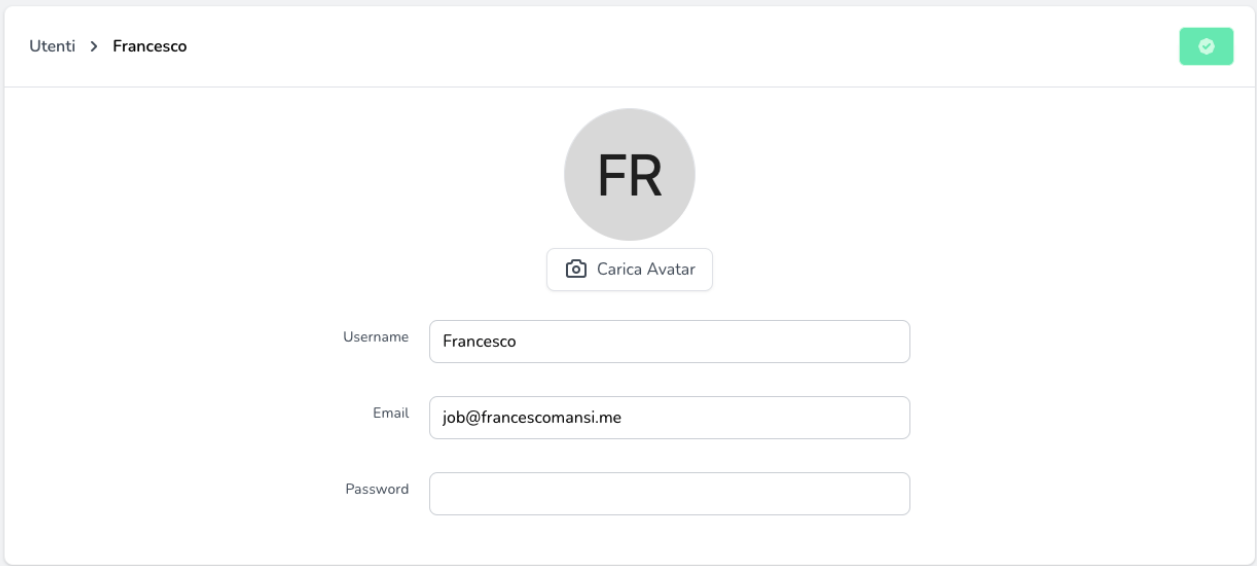
Non lasciarti spaventare dalla lunghezza del codice.

Ritorniamo sul file blade **users.edit**.
il componente appena creato è un `<form>` senza metodi, action e tutto il resto. Dovremo specificare gli attributi del form direttamente nel tag html del componente.
Sarà la direttiva `{{ $attributes->merge() }}` a fare il resto:

```
//resources/views/sections/users/edit.blade.php
<x-app-layout>
  <x-slot name="header">
      <h2 class="text-xl font-semibold leading-tight text-gray-800">
          {{ __('Modifica Utente') }}
      </h2>
  </x-slot>

  <div class="py-12">
    <div class="flex flex-row justify-center px-2 mt-3">
            <div class="w-full mx-auto max-w-7xl sm:px-6 lg:px-8">
                <x-auth-validation-errors class="mb-4" :errors="$errors"
/>
                <x-auth-session-status class="mb-4"
:status="session('status')" />
                <x-users-form class="overflow-hidden bg-white rounded-lg
shadow" method="POST"
                        action="{{route('users.update', $data['user']->id)
}}" enctype='multipart/form-data'
                        autocomplete="off"
```

```
                        :user="$data['user']">
                        @csrf
                        @method('PUT')
                    </x-users-form>
                </div>
            </div>
        </div>
    </x-app-layout>
```

Tutto quello presente tra i tag `<x-users-form>` e `</x-users-form>` (cioè **@csrf** e **@method('PUT')**) verrà passato al componente e renderizzato nella variabile **{{ $slot }}** inserita in precedenza.



In dettaglio vediamo che:

- **action**: Utilizzando sempre l'helper route('') stiamo richiamando un url relativo ad users definito nelle route (vedi lista delle route poco sopra). Inoltre ho aggiunto anche il parametro id, fondamentale per la query successiva;
- **@csrf** : Fondamentale per evitare attacchi di tipo Cross-site request forgery. La direttiva in questione genererà un token univoco per verificare la bontà dell'origine della richiesta.
- **@method('')** : Il metodo PUT non può essere definito mediante la direttiva classica, ecco quindi entrare in scena @method().

Per completare il processo di modifica dovremo configurare anche il comportamento della funzione update richiamata nella action del form.

La prima cosa da fare e controllare che i dati passati siano corretti, integri e non vuoti. Per fare questo possiamo utilizzare la funzione **validate()** ai dati in post:

```
$request->validate([
            'name' => 'required|string|max:191|alpha_dash',
```

```
            'email' => 'required|string|email|max:191|unique:users,email,'
. $id,
            'avatar' =>
'nullable|image|mimes:jpeg,jpg,gif,png,svg|max:2048',
            'password' => 'nullable|min:8'
        ]);
```

Tutte le regole proposte sono abbastanza intuitive, tranne (forse) una: **unique:users,email,' . $id**. Stiamo specificando che il campo email e id, insieme, dovranno identificare in modo univoco il record. Non saranno ammessi duplicati, in poche parole saranno **Unici**.

Per quanto riguarda le altre regole di validazione puoi trovarle direttamente sulla documentazione, ti basti sapere che solitamente per questo tipo di dati queste sono le più indicate.

Successivamente si procederà con l'eventuale aggiornamento dei dati.
Più che Laravel, il codice che seguirà è puro PHP:

```
empty($request['password'])
            ? $request->merge(['password' => $user->password])
            : $request->merge(['password' =>
Hash::make($request['password'])]);
        $user->update($request->except('avatar'));
        if ($request->file('avatar') != $currentAvatar && $request-
>hasFile('avatar')) {

            $name = Str::of($user->name)->lower()->slug() . '-' . $user->id
. '.' . request()->avatar->getClientOriginalExtension();;
            $request->file('avatar')->move('img/avatars', $name);

            $user->avatar = '/img/avatars/' . $name;
            $user->update();

        }
```

Andremo a controllare se è stata aggiunta una nuova password (che verrà cryptata mediante Hash) e l'avatar.

Essendo campi nullable non sarà obbligatorio mandarli in ogni modifica utente.

Ecco quindi il codice completo:

```
//app/Http/Controllers/UserController.php
public function update(Request $request, $id)
    {
        $request->validate([
            'name' => 'required|string|max:191|alpha_dash',
```

```
            'email' => 'required|string|email|max:191|unique:users,email,'
. $id,
            'avatar' =>
'nullable|image|mimes:jpeg,jpg,gif,png,svg|max:2048',
            'password' => 'nullable|min:8'
        ]);

        $user = User::findOrFail($id);
        $currentAvatar = $user->avatar;

        empty($request['password'])
            ? $request->merge(['password' => $user->password])
            : $request->merge(['password' =>
Hash::make($request['password'])]);
        $user->update($request->except('avatar'));
        if ($request->file('avatar') != $currentAvatar && $request-
>hasFile('avatar')) {
            $destinationPath = 'img/avatar/';
            $name = Str::of($user->name)->lower()->slug() . '-' . $user->id
. '.' . request()->avatar->getClientOriginalExtension();;
            $request->file('avatar')->move($destinationPath, $name);

            $user->avatar = $destinationPath . $name;
            $user->update();

        }

         return redirect()->back()->with('status', 'Profilo Modificato con
successo.');
    }
```

Per evitare errori, sarà fondamentale importare anche le librerie relative a Str e Hash sotto al namespace:

```
<?php

namespace App\Http\Controllers;

use App\Models\User;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Str;
```

Bene, potrai modificare a tuo piacimento l'utente in questione.
Ma se volessimo crearne uno nuovo? Lo scoprirai nel prossimo capitolo.

Create

Per la creazione di un nuovo, invece, dobbiamo procedere in modo speculare alla modifica.

Crea un nuovo file blade e crea le due funzioni di GET/POST

Quindi:

```
resources/views/sections/users/create.blade.php
```

Il layout sarà sempre il medesimo, e soprattutto il form sarà quello del componente creato in precedenza.

A differenza della modifica, dovrai richiamare una route diversa e il metodo non sarà più PUT ma POST:

```
//resources/views/sections/users/create.blade.php
<x-app-layout>
  <x-slot name="header">
      <h2 class="text-xl font-semibold leading-tight text-gray-800">
          {{ __('Crea Utente') }}
      </h2>
  </x-slot>

  <div class="py-12">
    <div class="flex flex-row justify-center px-2 mt-3">
        <div class="w-full mx-auto max-w-7xl sm:px-6 lg:px-8">
            <x-auth-validation-errors class="mb-4" :errors="$errors" />
            <x-auth-session-status class="mb-4"
:status="session('status')" />
            <x-users-form class="overflow-hidden bg-white rounded-lg
shadow" method="POST"
                    action="{{ route('users.store') }}"
enctype='multipart/form-data'
                    autocomplete="off">
                    @csrf
                    @method('POST')

            </x-users-form>
        </div>
    </div>
  </div>
 </x-app-layout>
```

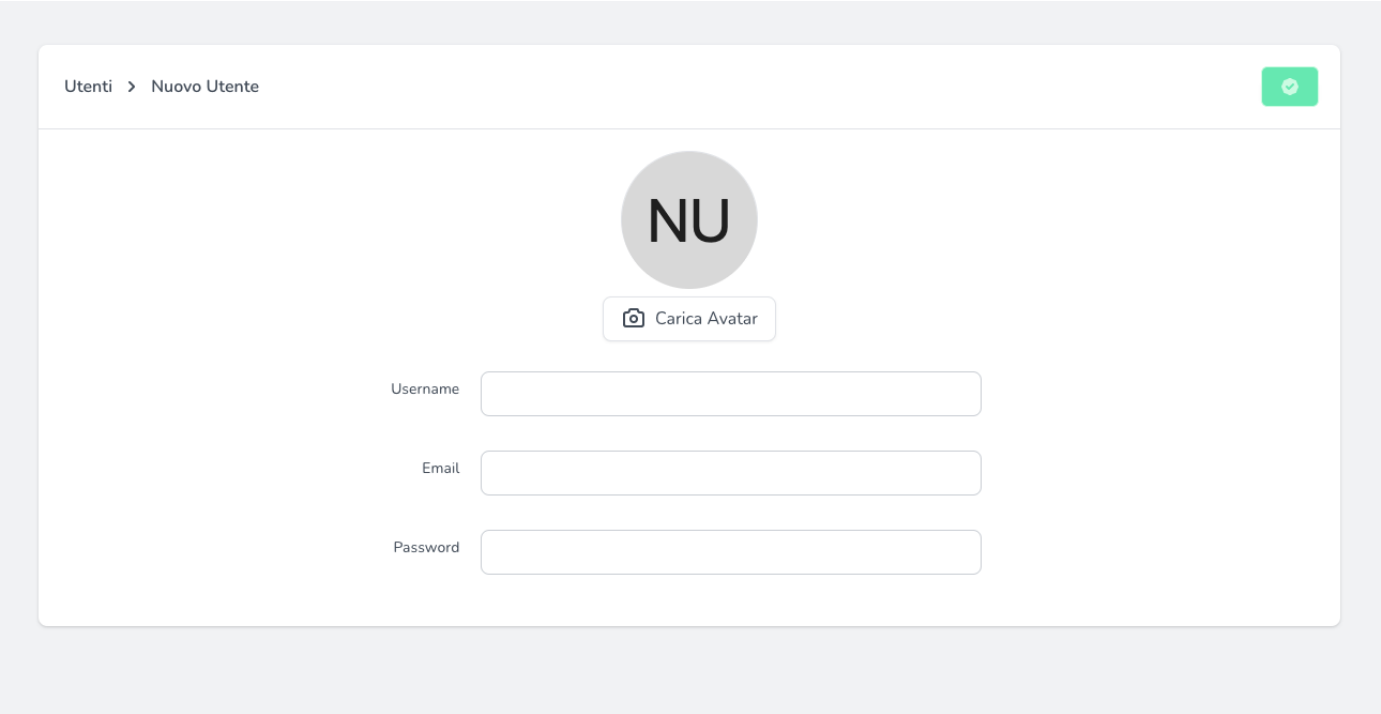
la view è stata creata, adesso è il turno della relative funzione nel controller:

```
//app/Http/Controllers/UserController.php
public function create()
  {
      return view('sections.users.create');
  }
```

Successivamente dovrai configurare la chiamata store(), che servirà per creare il nuovo record nel Database.
La logica di validazione sarà questa volta diversa rispetto al capitolo precedente. La password ora è un campo obbligatorio.
Non può esistere un utente senza password.

```
//app/Http/Controllers/UserController.php
public function store(Request $request)
   {
       $request->validate([
           'name' => 'required|string|max:191|alpha_dash',
           'email' => 'required|string|email|max:191|unique:users',
           'avatar' =>
'nullable|image|mimes:jpeg,jpg,gif,png,svg|max:2048',
           'password' => 'required|string|min:8'
       ]);

       $request->merge(['password' => Hash::make($request['password'])]);

       $user = User::create($request->except('avatar'));

       if ($request->hasFile('avatar')) {
                $destinationPath = 'img/avatar/';
           $name = Str::of($user->name)->lower()->slug() . '-' . $user->id
. '.' . $request->file('avatar')->getClientOriginalExtension();;

           $request->file('avatar')->move($destinationPath, $name);
           $user->avatar = $destinationPath . $name;

       }
       $user->update();

       return redirect()->route('users.index');
   }
```

Questo è il risultato:



Sentiti libero di creare un nuovo utente e salva con il pulsante verde in alto.
Crea un nuovo utente sia per testare il funzionamento ma anche perché nel prossimo capitolo ci servirà per testare la cancellazione.

Delete

Ultima atto di questo viaggio alla scoperta del Resource Controller è la funzione delete.

Per poter richiamare correttamente la funzione destroy() sarà necessario utilizzare il metodo DELETE, richiamabile con la direttiva **@method('DELETE')**.
Questa volta non dovrai creare componenti o layout, lo troverai già presente nella view:

```
//resources/views/sections/users/index.blade.php

<form action="{{route('users.destroy', $user->id)}}"
    method="POST" class="">
       @csrf
       @method('DELETE')
      <button class="p-1 border-2 border-red-200 rounded-md hover:bg-red-
100">
          <svg xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0
0 24 24" stroke="currentColor" class="w-4 h-4 text-red-500">
              <path stroke-linecap="round" stroke-linejoin="round"
stroke-width="2" d="M19 7l-.867 12.142A2 2 0 0116.138 21H7.862a2 2 0 01-
1.995-1.858L5 7m5 4v6m4-6v6m1-10V4a1 1 0 00-1-1h-4a1 1 0 00-1 1v3M4 7h16">
</path>
          </svg>
```

```
        </button>
</form>
```

Sulla falsa riga della modifica, avrai bisogno dell'id dell'elemento (*$user->id*) e che il tutto sia gestito attraverso un form in grado di comunicare attraverso DELETE.

Vediamo quindi la funzione vera e propria.

Apri il controller **UserController.php** e spostati fino alla funzione **destroy($id)**.
Un pò come per l'edit, dovrai cercare il record nella tabella users e, se la ricerca avrà dato esito positivo, eliminarla.

```
//app/Http/Controllers/UserController.php
public function destroy($id)
{
    $user = User::findOrFail($id);
    $user->delete();
    return redirect()->route('users.index');
}
```