

# View parametriche

- Consideriamo una view, con il suo codice e la relativa route:

localhost:8001

Cosa ci piace:

mangiare

```
{{-- layout.blade.php --}}
<html><head></head><body>
@yield('contenuto')
</body></html>
```

```
{{-- welcome.blade.php --}}
@extends('layout')
@section('contenuto')
<h2>Cosa ci piace:</h2>
mangiare
@endsection
```

```
<!-- web.php -->
<?php
// Web Routes
Route::get('/', function () {
    return view('welcome');
});
```

- La view *welcome* è **statica**: ogni volta che si serve *welcome*, il verbo ("mangiare") resta quello scritto nel codice
- Sarebbe utile che la view *welcome* fosse **dinamica**, in modo parametrico, cioè che il verbo cambi, da un rendering all'altro, senza che cambi il codice di *welcome.blade.php*, ma solo il modo in cui la closure in *web.php* invoca *view('welcome',...)*
- Si può, per questo, introdurre un parametro nella view, p.es. *\$azione\_pref* nella view *welcome*
- Ma come si determina il valore di *\$azione\_pref* ogni volta che questa view *welcome* viene servita?

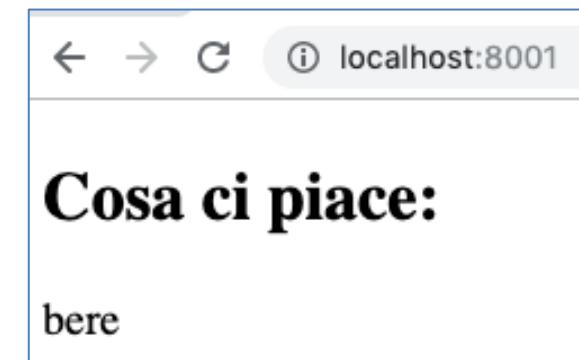
```
 {{-- welcome.blade.php --}}
@extends('layout')
@section('contenuto')
<h2>Cosa ci piace:</h2>
<?= $azione_pref ?>
@endsection
```

# View parametriche: meccanismo

- Dunque: come riceve un valore il parametro `$azione_pref` ogni volta che la view `welcome` viene servita?
- Ricordando che la view `welcome` veniva attivata da una route che invoca la funzione `view('welcome')`, occorre che questa chiamata abbia, per 2° argomento, un array hash con chiave `'azione_pref'`
- Il valore corrispondente alla chiave (qui `'bere'`) verrà assunto da `$azione_pref` nella view da servire
- Ecco che cambia la view servita!
- Se il valore di `$azione_pref` si potesse determinare, p.es., a partire da parti della URL, avremmo view ancor più che *parametriche*, cioè davvero *dinamiche*!

```
{{-- welcome.blade.php --}}  
  
@extends('layout')  
  
@section('contenuto')  
<h2>Cosa ci piace:</h2>  
<?= $azione_pref ?>  
@endsection
```

```
<! -- web.php -->  
  
<?php  
// Web Routes  
  
Route::get('/', function () {  
    return view('welcome',  
        ['azione_pref' => 'bere']);  
});
```



# View parametriche: array

- Il valore passato dalla route al parametro può essere anche un array, sul quale la view potrà eseguire un ciclo con l'appropriato codice PHP

```
<!-- web.php -->

<?php

Route::get('/', function () {
    return view('welcome', [ // view() ha un solo 2° parametro
        'azioni_pref' => // $azioni_pref è chiave e parametro view
        ['bere', 'mangiare'] // valore passato per $azioni_pref alla
    ]); // view welcome
});
```

- Ecco come la view *welcome* sfrutta l'array che le viene passato dalla route come valore del parametro *\$azioni\_pref*

**Cosa ci piace:**

bere  
mangiare

```
{{-- welcome.blade.php --}}

@extends('layout')

@section('contenuto')
<h2>Cosa ci piace:</h2>
<?php foreach ($azioni_pref as $una_azione) : ?>
    <br> <?= $una_azione; ?>
<?php endforeach; ?>
@endsection
```

# Annotazioni/abbreviazioni in blade

- Il componente *blade* di Laravel permette annotazioni equivalenti ma più concise ed eleganti dei tag `<?php... ?>`
- Si confrontino, p.es., due versioni di `welcome.blade.php`:

```
{-- welcome.blade.php --}

@extends('layout')

@section('contenuto')
<h2>Cosa ci piace:</h2>
<?php foreach ($azioni_pref as $una_azione) : ?>
    <br> <?= $una_azione; ?>
<?php endforeach; ?>
@endsection
```

```
{-- welcome.blade.php --}

@extends('layout')

@section('contenuto')
<h2>Cosa ci piace:</h2>
@foreach ($azioni_pref as $una_azione)
<br> {{ $una_azione }}
@endforeach
@endsection
```

- La traduzione di `@foreach` e simili avviene solo la prima volta che la pagina è servita (per maggiore efficienza)
- Si può ottenere ancora più concisione rimpiazzando, come nell'esempio sopra a destra, l'espressione `<?= ... ?>` con `{{...}}` (senza ';' nel PHP!)

# Altre facility sintattiche per il parametro

- Nel file delle rotte il valore del parametro hash, se complesso, si può specificare in modo più chiaro attraverso un'assegnazione a una variabile locale alla closure associata alla route /

```
<!-- web.php -->

<?php

Route::get('/', function () {
    return view('welcome', [ // un solo 2° parametro
        'azioni_pref' => // chiave e parametro view
        ['bere','mangiare']] // valore passato alla view
    ]);
}); // per il parametro
// $azione_pref
```

- Cioè come la variabile *\$lista\_azioni* qui a destra:

```
<!-- web.php -->

<?php

Route::get('/', function () {
    $lista_azioni = ['bere','mangiare'];
    return view('welcome', [ // un solo 2° parametro
        'azioni_pref' => // chiave e parametro view
        $lista_azioni // valore passato alla view
    ]);
}); // per il parametro
// $azione_pref
```

# Due parametri nella view

- Vediamo un esempio, con **due parametri** `$azioni_pref` e `$titol` nella view *welcome* istanza del template *layout*
  - ciò consente al routing che invoca la view *welcome* di rendere anche il titolo parametrico
- Nella gestione delle rotte in *web.php*, nella closure, la funzione *view()* ha ora, rispetto a prima, un 3° argomento: la coppia hash con chiave '`titol`' e valore '`'Benvenuti'`', corrispondente quindi al 2° parametro `$titol` della view *welcome*

```
{{-- layout.blade.php --}}
<html><head><title>
@yield('titolo')
</title></head>
<body>
@yield('contenuto')
</body>
</html>
```

```
{{-- welcome.blade.php --}}

@extends('layout')
@section('titolo', $titol)

@section('contenuto')
<h2>Cosa ci piace:</h2>
@foreach ($azioni_pref as $una_azione)
<br> <?= $una_azione; ?>
@endforeach
@endsection
```

```
<!-- web.php -->

<?php
Route::get('/', function () {
    $lista_azioni = ['bere', 'mangiare'];
    return view('welcome',
        [ 'azioni_pref' => $lista_azioni ],
        [ 'titol' => 'Benvenuti' ]
    );
});
```

# Parametri multipli nella view

- Vediamo un altro esempio, con un terzo parametro **\$quando** nella view *welcome*
- Non lo si può però istanziare dando a *view()*, nella rotta in *web.php*, un 4° argomento (cf., causa errore)!
- Bisogna invece dare a *view()*, come 2° argomento, un **array di coppie hash**: ognuna di queste ha una
  - **chiave** nome di un parametro della view (p.es. '**quando**') e un
  - **valore** che verrà assegnato al parametro (p.es. '**oggi**')
- Così è possibile gestire un numero qualsiasi di parametri!

```
{{-- welcome.blade.php --}}  
  
@extends('layout')  
@section('titolo', $titol)  
  
@section('contenuto')  
<h2>Cosa ci piace {{ $quando }}:</h2>  
@foreach ($azioni_pref as $una_azione)  
<br> <?= $una_azione; ?>  
@endforeach  
@endsection
```

```
<!-- web.php -->  
  
<?php  
Route::get('/', function () {  
    $lista_azioni = ['bere', 'mangiare'];  
    return view('welcome', [  
        'azioni_pref' => $lista_azioni,  
        'titol' => 'Benvenuti',  
        'quando' => 'oggi'  
    ]);  
});
```

# Parametri della URL

- Il valore da assegnare a una variabile come `$quando`, nella gestione della route in `web.php` può anche essere estratto dalla URL
- Per questo si usa la funzione `request('id')`, che va a cercare un parametro *id* nella URL
- Nell'esempio qui, `request('tempo')` estraе dalla URL il parametro `?tempo=oggi`

```
<!-- web.php -->

<?php
Route::get('/', function () {
    $azioni = [ 'bere', 'mangiare'];
    return view('welcome', [
        'azione_pref' => $azioni ,
        'quando' => request('tempo');
    ]);
});
```

```
{{-- welcome.blade.php --}}

@extends('layout')

@section('contenuto')
<h2>Cosa ci piace {{ $quando }}:</h2>
...
```



# `{{...}}` - particolarità

- `{{...}}` non equivale esattamente a `<?= ... ?>` come si vede qui a fianco
- La variabile in `...` potrebbe contenere una stringa con codice javascript!
- `{{...}}` protegge, in quanto non interpreta il codice!
- Invece `<?= ... ?>` interpreta il js in `...`

Così,  
`<h2>Cosa ci piace  
<?= $quando ?>:</h2>`

produce → perché l'*alert()* viene interpretato

The diagram illustrates the execution flow of the code. On the left, a code editor shows a PHP route definition and a corresponding Blade template. A blue arrow points from the Blade template to a browser window. Another blue arrow points from the browser window to a modal dialog at the bottom right.

`<!-- web.php -->`  
`<?php`  
Route::get('/', function () {  
 \$azioni = ['bere', 'mangiare'];  
 return view('welcome', [  
 'azione\_pref' => \$azioni ,  
 'quando' =>  
 '<script>alert(ciao)</script>'  
 ]);  
});

`{-- welcome.blade.php --}`  
`@extends('layout')`  
`@section('contenuto')`  
`<h2>Cosa ci piace {{ $quando }}:</h2>`

localhost:8001

Cosa ci piace `<script>alert(ciao)</script>`:

bere  
mangiare

localhost:8001 dice  
ciao

OK

# `{...}` e `!! ... !!`

- Come detto, `{...}` protegge il proprio argomento (espressione) perché potrebbe essere Javascript non previsto o, peggio, malizioso, con effetti comunque indesiderati
- Cautele simili si dovrebbero avere con il codice che verrà valutato all'interno di tag `<?php ... ?>`
  - di norma dovrebbe essere PHP legale, ma potrebbe non esserlo se proviene in parte da parametri GET (`?XXX=YYY`) o da form inviati con POST
  - in questi casi `{...}` dà maggiore protezione (si dice che "quota" o "protegge" il proprio argomento)
  - ciò non accade, invece, con il costrutto blade `!! ... !!` che presenta quindi dei rischi come `<?php ... ?>`
  - si dice che il codice in `...` "va considerato colpevole finché non lo si dimostra innocente"

# Facility per variabili nelle view: ->with

- Come detto, in una route, la funzione `view()` usa il 2° parametro per associare un valore `V` alla chiave '`xyz`', tale che `$xyz` figuri come variabile nella view blade che è il 1° parametro di `view()`

```
<!-- web.php -->
<?php
Route::get('/', function () {
    $azioni = ['bere', 'mangiare'];
    return view('welcome',
        [ 'quando' => 'oggi',
          'azionepref' => $azioni ]);
});
```

- Lo stesso effetto si può ottenere con l'operatore `->withXyz(V)`, come qui, nel 2° riquadro, equivalente al 1°
- NB: è bene che il nome della chiave (p.es. `xyz`) contenga solo minuscole
- Altra forma equivalente nel 3° riquadro: unico `with` applicato a tutto un hash-array che associa più chiavi ciascuna a un valore

```
<!-- web.php -->
<?php
Route::get('/', function () {
    $azioni = ['bere', 'mangiare'];
    return view('welcome')->withQuando('oggi')
        ->withAzionepref($azioni);
});
```

```
Route::get('/', function () {
    $azioni = ['bere', 'mangiare'];
    return view('welcome')->with(
        [ 'quando' => 'oggi',
          'azionepref' => $azioni ] );
});
```

# Facility per variabili nelle view: *compact()*

- Come già detto, in una route, la funzione *view()* usa il 2° parametro per associare un valore alla chiave '*xyz*', tale che *\$xyz* figuri come variabile nella view blade che è il 1° parametro di *view()*
- Si consideri ora un caso come qui a destra, in cui:
  - il 2° parametro definisce una sola chiave '*xyz*'
  - il valore assegnato alla chiave sia quello di una variabile chiamata anche *\$xyz*
- L'array hash, in questo caso, si può esprimere mediante la funzione PHP *compact()*, come qui a destra:

```
<!-- web.php -->
<?php
Route::get('/', function () {
    $azioni = ['bere', 'mangiare'];
    return view('welcome',
        ['azioni' => $azioni]);
});
```

```
<!-- web.php -->
<?php
Route::get('/', function () {
    $azioni = ['bere', 'mangiare'];
    return view('welcome',
        compact('azioni'));
});
```

# Rotte con "chiusure"

- Come si è visto, nei casi semplici, ogni rotta viene associata con un *callback* che è una "chiusura" (funzione anonima)
  - questo ci risparmia il disturbo di trovare un nome al callback!
- Esistono però delle convenzioni utili sui nomi (e i contenuti) dei callback
- Nell'esempio sopra, si ripete uno schema: ogni callback non fa altro che servire una *Page* (web), sostanzialmente statica (a meno dei template)
- Altre pagine statiche da servire allungherebbero a dismisura la lista delle *Route* e dei relativi callback/closure
- Quindi sarebbe utile ricondurre tutti i callback a metodi di un gruppo di callback, che chiameremo *PagesController* (NB: nome *Page* e *s* plurale)
- Si genera uno (schema, da riempire, di) controller con il tool *php artisan*:

```
my_app $ php artisan make:controller PagesController  
Controller created successfully
```

The screenshot shows a code editor window with a file named 'web.php'. The code defines three routes using the Route::get() method. Each route has a closure (lambda function) as its callback. The first route maps the root path '/' to a view named 'welcome' with parameters 'quando' => 'oggi' and 'azionepref' => ['bere']. The second route maps '/contact' to a view named 'contact'. The third route maps '/about' to a view named 'about'. The code uses color-coded syntax highlighting for keywords like 'Route', 'get', 'return', 'view', and variable names.

```
web.php
<?php

Route::get('/', function () {
    return view('welcome', [
        'quando' => 'oggi',
        'azionepref' => ['bere']
    ]);
});

Route::get('/contact', function () {
    return view('contact');
});

Route::get('/about', function () {
    return view('about');
});
```

# PagesController

- Codice "boilerplate", generato automaticamente
- In `web.php` a ogni route mappata su una pagina statica, si associa il metodo desiderato, sia `f()`, del *PagesController*, con la sintassi `PagesController@f`
- P.es. alla route `'/contact'` la funzione callback `PagesController@contact`
- Questa va definita nella classe *PagesController* e fa ciò che faceva la closure: restituisce la view `contact.blade.php`

The screenshot shows a code editor with a sidebar titled "FOLDERS". The "Http" folder is expanded, showing "Controllers" and "Auth". Inside "Controllers", two files are listed: "Controller.php" and "PagesController.php". The main pane displays the first few lines of the "PagesController.php" file:

```
<?php  
namespace App\Http\Controllers;  
use Illuminate\Http\Request;  
class PagesController extends Controller  
{  
    //  
}
```

```
<?php // Web Routes  
/* eliminiamo, con un commento, la closure per la route /contact  
Route::get('/contact', function () {  
    return view('contact'); });  
* sostituendola con un callback parte di un controller  
*/  
  
Route::get('/contact', 'PagesController@contact');  
  
/* ora, il corpo della closure diventa quello di una funzione  
chiamata contact() che è un metodo del controller  
*/
```

```
class PagesController extends Controller  
{  
    public function contact() {  
        return view('contact');  
    }  
}
```

# Controller

- Ha senso sfruttare il controller per tutte le view:

```
<!-- web.php -->
<?php

// Web Routes

Route::get('/', 'PagesController@home');

Route::get('/contact', 'PagesController@contact');

Route::get('/about', 'PagesController@about');
```

- L'idea, cioè, è che ogni rotta/URL sia associata a un metodo del controller e il codice
- Quando si introdurranno altre "entità", p.es. *Task*, *User* etc, potrà essere utile introdurre dei controller corrispondenti, p.es. *TasksController*, *UsersController* ...
- Probabilmente i metodi del controller non si limiteranno a invocare una view e conterranno anche "business logic"

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

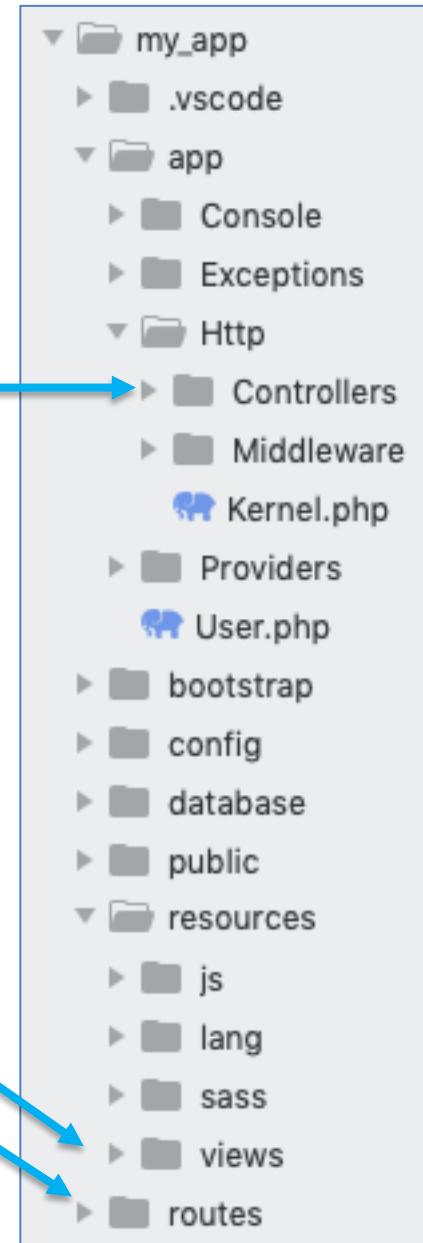
class PagesController extends Controller
{
    public function home() {
        return view('welcome',
            ['azione_pref' => 'bere']);
    }
    public function contact() {
        return view('contact');
    }
    public function about() {
        return view('about');
    }
}
```

# Layer Laravel finora

Finora abbiamo introdotto 3 layer di software:

- *controllers*
- *views*
- *routes*

Ora  
introdurremo  
i **modelli**, per  
accedere ai DB  
backend delle  
nostre web app



# Modelli

- Per costruire modelli, si sfrutta un componente detto *eloquent*
- Si parte, come al solito, con l'*artisan wizard*
- Si noti il file `.env` contiene profilo e configurazione dell'app Laravel
- Si noti il gruppo *DB* – non solo *mysql*

```
~ $ laravel new prog
...
Application ready! Build something amazing.
```

The screenshot shows a Mac OS X desktop environment with a terminal window open. The terminal window title is 'prog' and it is located at the path '~/.Dropbox/SD1handouts/php/laravel/prog/.env'. The window is labeled 'UNREGISTERED'. Inside the terminal, the command 'laravel new prog' is entered, followed by three dots, and the message 'Application ready! Build something amazing.' The terminal's status bar at the bottom indicates 'Line 3, Column 23', 'UTF-8', 'Unix', and a small number '5'.

```
~ $ laravel new prog
...
Application ready! Build something amazing.
```

Line	Setting
1	APP_NAME=Laravel
2	APP_ENV=local
3	APP_KEY=base64:Ujsh7YHoKF6bHBmeT
4	APP_DEBUG=true
5	APP_URL=http://localhost
6	
7	LOG_CHANNEL=stack
8	
9	DB_CONNECTION=mysql
10	DB_HOST=127.0.0.1
11	DB_PORT=3306
12	DB_DATABASE=homestead
13	DB_USERNAME=homestead
14	DB_PASSWORD=secret
15	
16	BROADCAST_DRIVER=log
17	CACHE_DRIVER=file
18	QUEUE_CONNECTION=sync
19	SESSION_DRIVER=file
20	SESSION_LIFETIME=120
21	

# Configurazione

- Le variabili definite nel file `.env` vengono lette da codice *php* che, per lo più, sta in *config*
- Per il DB, in particolare, *database.php*
- e, per il nostro esempio, l'array di array 'mysql'

```
<?php  
use Illuminate\Support\Str;  
return [  
    /*  
     * Default Database Connection Name  
     */  
    'default' => env('DB_CONNECTION', 'mysql'),  
    /*  
     * Database Connections  
     */  
    'mysql' => [  
        'driver' => 'mysql',  
        'url' => env('DATABASE_URL'),  
        'host' => env('DB_HOST', '127.0.0.1'),  
        'port' => env('DB_PORT', '3306'),  
        'database' => env('DB_DATABASE', 'forge'),  
        'username' => env('DB_USERNAME', 'forge'),  
        'password' => env('DB_PASSWORD', ''),  
        'unix_socket' => env('DB_SOCKET', ''),  
        'charset' => 'utf8mb4',  
    ],  
];
```

# phpMyAdmin

- Per interagire con il DB relazionale *mysql* vi sono molti tool
- Useremo *phpMyAdmin*, che è open e... è una web app PHP!
- si può installare in molti modi, secondo la piattaforma
- se installate un pacchetto XAMPP..., avrete già *phpMyAdmin*
- oppure, si può scaricare come .zip dal sito o via GitHub o, essendo un app PHP, si può installare con *composer*, come qui:

```
~ $ composer create-project phpmyadmin/phpmyadmin
Installing phpmyadmin/phpmyadmin (4.8.5)
- Installing phpmyadmin/phpmyadmin (4.8.5): Downloading (100%)
Created project in /Users/gp/phpmyadmin
...
~ $ cd /Users/gp/phpmyadmin
phpmyadmin $ composer update # aggiorna phpmyadmin
...
```

- ora si può eseguire *phpmyadmin* come app PHP **stand-alone**:

```
~ phpmyadmin $ php -S localhost:7777 # o altro port a piacere
```

# phpMyAdmin come pacchetto

- L'altra opzione, per installare phpMyAdmin, è usare il gestore di pacchetti di sistema (brew/OSX, apt/debian...), possibilmente per tutto lo stack *apache/php/phpmyadmin*

```
~ $ brew install httpd / apt install apache2      # httpd = apache  
~ $ brew/apt install php  
~ $ brew/apt install phpmyadmin
```

# a questo punto occorre intervenire sui file di configurazione di apache/php, cf.

```
# http://guide.debianizzati.org/index.php/Installare\_un\_ambiente\_LAMP:\_Linux,\_Apache2,\_SSL,\_MySQL,\_PHP5\_-\_Stretch  
# https://www.digitalocean.com/community/tutorials/how-to-install-the-apache-web-server-on-ubuntu-18-04  
# https://getgrav.org/blog/macros-mojave-apache-multiple-php-versions
```

- NB: le istruzioni alle URL sopra riguardano, *phpmyadmin* a parte, come installare php *come modulo* di Apache.
- In questo caso, la porta 80 è controllata da Apache che, se deve servire una pagina php, si rivolge al modulo PHP

# PHP come modulo di Apache

- Sull'argomento, le URL alla slide precedente forniscono ottime guide
- Una sintesi la dà il tool *brew* di OSX, ma vale per Unix in generale:

```
$ brew info php
```

```
...
```

```
==> Caveats
```

To enable PHP in Apache add the following to httpd.conf and restart Apache:

```
LoadModule php7_module /usr/local/opt/php/lib/httpd/modules/libphp7.so
<FilesMatch \.php$>
    SetHandler application/x-httpd-php
</FilesMatch>
```

Finally, check DirectoryIndex includes index.php

```
DirectoryIndex index.php index.html
```

The `php.ini` and `php-fpm.ini` file can be found in: `/usr/local/etc/php/7.3/`

# Ma PHP può girare come *php-fpm*, anziché come modulo di Apache

To have `launchd` start php now and restart at login:

```
brew services start php
```

Or, if you don't want/need a background service you can just run:

```
php-fpm
```

# Phpmyadmin e PHP con Apache

- Se PHP gira come modulo di Apache, *phpmyadmin* va attivato intervenendo sui file di configurazione di Apache:

```
$ brew info phpmyadmin    # le stesse istruzioni si applicano a Apache/PHP su Linux
...
==> Caveats
To enable phpMyAdmin in Apache, add the following to httpd.conf and restart Apache:
  Alias /phpmyadmin /usr/local/share/phpmyadmin
  <Directory /usr/local/share/phpmyadmin/>
    Options Indexes FollowSymLinks MultiViews
    AllowOverride All
    <IfModule mod_authz_core.c>
      Require all granted
    </IfModule>
    <IfModule !mod_authz_core.c>
      Order allow,deny
      Allow from all
    </IfModule>
  </Directory>
Then open http://localhost/phpmyadmin
The configuration file is /usr/local/etc/phpmyadmin.config.inc.php
```

# Avviare mysql come servizio

- Mysql o il suo fork *mariadb* girano come servizi
- In ambiente Windows conviene avviarlo all'interno di XAMPP/...
- In ambiente Unix si avviare il daemon da shell, così:

```
$ brew services run mariadb      # OSX
```

```
$ sudo systemctl start mariadb   # Linux
```

- Presumendo lo si fosse installato in precedenza:

```
$ brew install mariadb      # OSX
```

```
$ apt install mariadb-server mariadb-client    # Linux
```

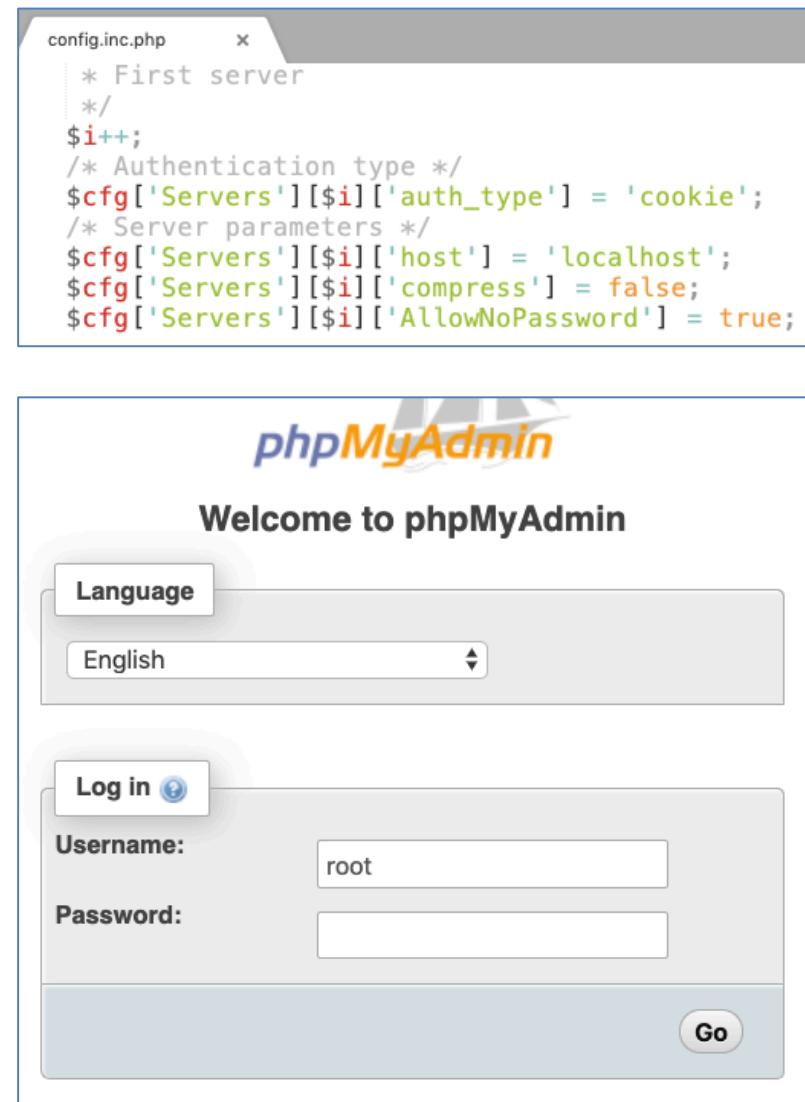
In realtà installare su Ubuntu potrebbe avere dei prerequisiti, vedi:

<https://computingforgeeks.com/install-mariadb-10-on-ubuntu-18-04-and-centos-7/>

- L'uso di password per *root* (NB: *root* di *mysql*) è raccomandato in produzione, ma fonte di problemi in sviluppo, specie all'inizio
- Per eliminare la password o, in generale, ri-inizializzare i metadati di *mysql*: <https://dev.mysql.com/doc/refman/en/data-directory-initialization-mysqld.html>

# mysql e phpmyadmin

- Per accesso a *mysql* senza password, nella home di *phpmyadmin* si crei (o modifichi) il file *config.inc.php* (ultimo rigo qui a destra)
  - se, anziché stand-alone, PHP gira come modulo di Apache, il file è:  
*.../etc/phpmyadmin.config.inc.php*
- A questo punto, se si è già avviato il servizio mysql, si può interagire con esso con *phpmyadmin*
- NB: per problemi di autenticazione con *mysql*, cf.  
<https://stackoverflow.com/questions/11634084>



The image shows two side-by-side screenshots. On the left is a screenshot of a code editor displaying the contents of the file `config.inc.php`. The code defines a server configuration with the following parameters:

```
config.inc.php
/*
 * First server
 */
$i++;
/* Authentication type */
$cfg['Servers'][$i]['auth_type'] = 'cookie';
/* Server parameters */
$cfg['Servers'][$i]['host'] = 'localhost';
$cfg['Servers'][$i]['compress'] = false;
$cfg['Servers'][$i]['AllowNoPassword'] = true;
```

On the right is a screenshot of the *phpMyAdmin* web interface. It features a header with the *phpMyAdmin* logo and the text "Welcome to phpMyAdmin". Below the header is a "Language" dropdown menu set to "English". Underneath is a "Log in" form with fields for "Username" (containing "root") and "Password". At the bottom right of the form is a "Go" button.

# DB mysql di prova

The screenshot shows the phpMyAdmin interface for managing MySQL databases. The left sidebar lists existing databases: information\_schema, mysql, performance\_schema, and portal\_db. A 'Nuovo' (New) button is available for creating a new database. The main area is titled 'Database' and features a 'Crea un nuovo database' (Create a new database) button. Below it, there is an input field for the database name, currently set to 'utf8mb4\_general\_ci', and a 'Crea' (Create) button.

- Creare con phpmyadmin un database chiamato *prova*
- Creare con *laravel new* un nuovo progetto *prova\_db*
- Modificare la sezione *DB\_...* del file *.env* del progetto
- (vedi slide su *mysql*)

```
.env
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=prova
DB_USERNAME=root
DB_PASSWORD=
```

# Laravel: le tabelle

- Per creare le tabelle del progetto *prova\_db* si usa di nuovo il wizard *artisan*, con il comando *migrate*

```
$ cd prova_db/  
prova_db $ php artisan migrate  
Migration table created successfully.  
Migrating: 2014_10_12_000000_create_users_table  
Migrated: 2014_10_12_000000_create_users_table  
Migrating: 2014_10_12_100000_create_password_resets_table  
Migrated: 2014_10_12_100000_create_password_resets_table
```
- Laravel genera automaticamente le tabelle, sfruttando la classe *Blueprint*
  - la API di *Blueprint* ci dà il modo di definire nuove tabelle che vengono poi create nel DB tramite *artisan migrate*
- Ciò rende semplice la generazione dello schema del DB e permette di prescindere sostanzialmente da SQL per la gestione del DB
- Altro vantaggio: per ricreare lo schema del DB, basta clonare il codice del progetto ed eseguire *artisan migrate* (vedi oltre)

# Migrazioni

- NB: per usare *artisan migrate*, deve girare mysql e l'app Laravel deve avere un file *.env* appropriato (DB esistente, user/passwd OK)

```
prova_db $ php artisan migrate:status
Migration table not found.

prova_db $ php artisan migrate          # in realtà migrate include anche il comando
Migration table created successfully      # migrate:install, che crea la Migration table
Migrating: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_000000_create_users_table
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated: 2014_10_12_100000_create_password_resets_table

prova_db $ php artisan migrate:status
+-----+-----+
| Ran? | Migration                                | Batch |
+-----+-----+
| Yes  | 2014_10_12_000000_create_users_table         | 1     |
| Yes  | 2014_10_12_100000_create_password_resets_table | 1     |
+-----+-----+
```

- *artisan migrate* genera **in mysql** la tabella delle migrazioni
- genera anche quelle dell'app, tra cui due "predefinite" (v. oltre)
- le *migrazioni* successive del Database, dopo la prima *artisan migrate*, danno luogo a *versioni* e consentono un certo *version control*

# Migrazioni / 2

- Il wizard *migrate* crea un *model* minimale, con le tabelle per gli utenti e i password\_resets
  - il primo *migrate* creerà anche una tabella *migrations*, per tracciare le migrazioni
- *migrate* successive, se l'app non specifica nuove tabelle (vedremo poi come), non hanno effetto:

```
prova_db $ php artisan migrate:status
+-----+-----+
| Ran? | Migration                                | Batch |
+-----+-----+
| Yes  | 2014_10_12_000000_create_users_table        | 1      |
| Yes  | 2014_10_12_100000_create_password_resets_table | 1      |
+-----+-----+
prova_db $ php artisan migrate
Nothing to migrate.
```

- Usiamo *phpmyadmin* per constatare l'effetto del primo *artisan migrate*
  - ci aspettiamo di trovare tabelle *users* e *password\_resets*

# Dopo la prima migrazione

- L'effetto di *artisan migrate* si può osservare con *phpmyadmin*:

The screenshot shows the phpMyAdmin interface at `localhost:7777/server_databases.php`. The left sidebar lists databases: information\_schema, mysql, performance\_schema, prova (selected), and users. The prova database has three tables: migrations, password\_resets, and users. The main panel shows the 'Structure' tab for the 'prova' database. A message says 'No tables found in database.' Below it, a 'Create table' form is open, showing fields for 'Name:' and 'Number of columns: 4'. The top navigation bar includes links for Structure, SQL, Search, Query, Export, Import, and Operations.

- artisan migrate:rollback* torna indietro di una migrazione

```
prova_db $ php artisan migrate:rollback
Rolling back: 2014_10_12_100000_create_password_resets_table
Rolled back: 2014_10_12_100000_create_password_resets_table
Rolling back: 2014_10_12_000000_create_users_table
Rolled back: 2014_10_12_000000_create_users_table
```

# Rollback di una migrazione

```
# dopo il precedente php artisan migrate:rollback
```

```
prova_db $  
php artisan migrate:status  
+-----+-----+-----+  
| Ran? | Migration | Batch |  
+-----+-----+-----+  
| No | 2014_10_12_000000_create_users_table | |  
| No | 2014_10_12_100000_create_password_resets_table | |  
+-----+-----+-----+
```

The screenshot shows the phpMyAdmin interface. On the left, the database tree is visible with 'prova' selected. Under 'prova', there are 'migrations' and 'migrations' (a new folder). The main panel shows the 'migrations' table with the following structure:

	<b>id</b>	<b>migration</b>	<b>batch</b>

At the top, a message says: "MySQL returned an empty result set (i.e. zero rows). (Query took 0.0001 seconds.)". Below it, the SQL query is shown: "SELECT \* FROM `migrations`".

- Rimane solo la tabella *migrations* ed è vuota!
- Rifacciamo la (prima) migrazione:

# Redo di una migrazione

```
prova_db $ php artisan migrate # di fatto un redo
Migrating: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_000000_create_users_table
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated: 2014_10_12_100000_create_password_resets_table
prova2_db $ php artisan migrate:status
+-----+-----+
| Ran? | Migration | Batch |
+-----+-----+
| Yes | 2014_10_12_000000_create_users_table | 1 |
| Yes | 2014_10_12_100000_create_password_resets_table | 1 |
+-----+-----+
```

- vediamo i sottocomandi di *migrate*:

```
prova_db $ php artisan | grep migrate
Laravel Framework 5.8.18
...
migrate
  migrate:install  Create the migration repository # che è sul DB; install è implicito nel migrate iniziale
  migrate:status   Show the status of each migration
  migrate:rollback Rollback the last database migration
  migrate:fresh    Drop all tables and re-run all migrations
  migrate:refresh  Reset and re-run all migrations # praticamente deprecato, si usi fresh
  migrate:reset    Rollback all database migrations
```

# Struttura di una tabella

The screenshot shows the phpMyAdmin interface with the following details:

- Server:** localhost
- Database:** prova
- Table:** users
- Structure View:** The "Structure" tab is selected.
- Table Structure:** The "Table structure" tab is selected.
- Columns:** The table has 8 columns with the following details:

#	Name	Type	Collation	Attributes	Null	Default	Comments
1	<b>id</b>	bigint(20)		UNSIGNED	No	None	
2	<b>name</b>	varchar(255)	utf8mb4_unicode_ci		No	None	
3	<b>email</b>	varchar(255)	utf8mb4_unicode_ci		No	None	
4	<b>email_verified_at</b>	timestamp			Yes	NULL	
5	<b>password</b>	varchar(255)	utf8mb4_unicode_ci		No	None	
6	<b>remember_token</b>	varchar(100)	utf8mb4_unicode_ci		Yes	NULL	
7	<b>created_at</b>	timestamp			Yes	NULL	
8	<b>updated_at</b>	timestamp			Yes	NULL	

Osserviamo qui la struttura della tabella *users* (creata da *artisan migrate*)

- perché è stata creata e perché con questo schema, da dove proviene?

La risposta è in *prova\_db/database/migrations*, dove troviamo, tra l'altro, la specifica della prima migrazione per le tabelle *users* (e *password\_resets*)

- NB: questi file sono creati già al *laravel new* e preesistono al 1° *artisan migrate*
- In effetti, essi dicono cosa fare al 1° *artisan migrate*

Questo ci dà la chiave per modificare la struttura di *users*, ma, soprattutto, per creare nuove tabelle attraverso *artisan migrate*

# Codice PHP delle migrazioni

Ecco il file per la tabella *users* (schema a destra) in *prova\_db/database/migrations*



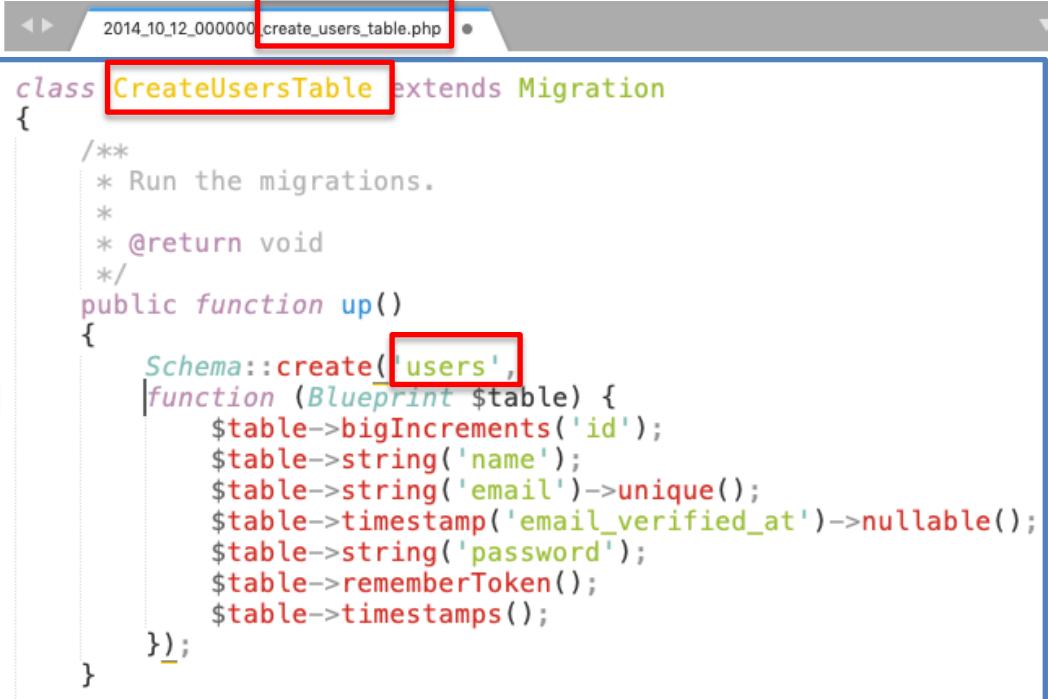
FOLDERS

- prova\_db
- app
- bootstrap
- config
- database
  - factories
  - migrations
- 2014\_10\_12\_000000\_2014\_10\_12\_100000\_
- seeds
- .gitignore
- public
- resources
- routes
- storage
- tests
- vendor
- .editorconfig
- .env
- .env.example
- .gitattributes
- .gitignore
- .styleci.yml

2014\_10\_12\_000000\_create\_users\_table.php

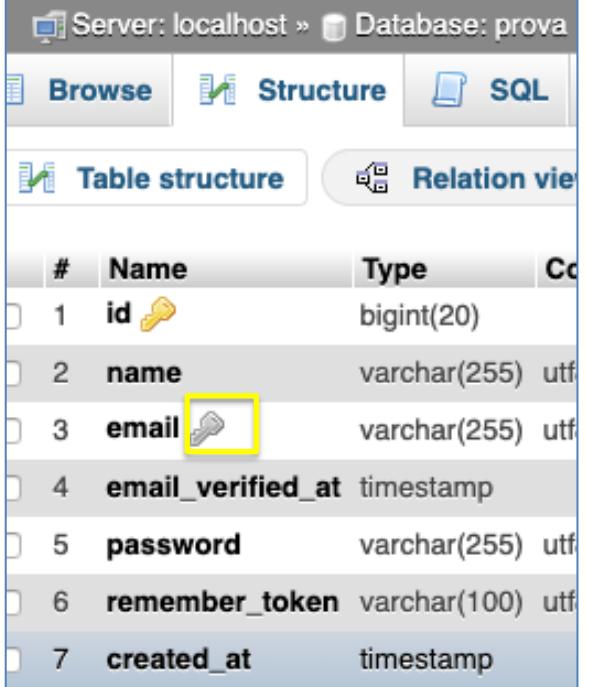
```
<?php  
use Illuminate\Support\Facades\Schema;  
use Illuminate\Database\Schema\Blueprint;  
use Illuminate\Database\Migrations\Migration;  
  
class CreateUsersTable extends Migration  
{  
    /**  
     * Run the migrations.  
     *  
     * @return void  
     */  
    public function up()  
    {  
        Schema::create('users',  
            function (Blueprint $table) {  
                $table->bigIncrements('id');  
                $table->string('name');  
                $table->string('email')->unique();  
                $table->timestamp('email_verified_at')->nullable();  
                $table->string('password');  
                $table->rememberToken();  
                $table->timestamps();  
            }  
        );  
    }  
}
```

# Codice PHP delle migrazioni / 2



```
2014_10_12_000000 create_users_table.php

class CreateUsersTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('users',
            function (Blueprint $table) {
                $table->bigIncrements('id');
                $table->string('name');
                $table->string('email')->unique();
                $table->timestamp('email_verified_at')->nullable();
                $table->string('password');
                $table->rememberToken();
                $table->timestamps();
            });
    }
}
```



#	Name	Type	Collation
1	<b>id</b>	bigint(20)	
2	<b>name</b>	varchar(255)	utf8mb4_unicode_ci
3	<b>email</b>	varchar(255)	utf8mb4_unicode_ci
4	<b>email_verified_at</b>	timestamp	
5	<b>password</b>	varchar(255)	utf8mb4_unicode_ci
6	<b>remember_token</b>	varchar(100)	utf8mb4_unicode_ci
7	<b>created_at</b>	timestamp	

- Si noti la corrispondenza tra nome della tabella *users*, nome del file in *migrations*, nome della classe, argomento di *Schema::create*
- Si riconosce il codice PHP che genera la tabella '*users*' con gli attributi: nome (metodo di *Blueprint*) e tipo (stringa argomento)
  - p.es. all'attributo *varchar name* corrisponde *\$table->string('name')*
- Si notino i constraint *unique* (la ) in *email* e *nullable* in *email\_verified\_at*

# Modifica dello schema del DB

- Per modificare la tabella *users*, p.es. rinominare il campo '*name*' in '*username*' si può intervenire sul codice, nel file *database/migrations/...create\_users\_table.php*
- L'idea è di usare *migrate*, ma non basterà!

```
prova_db $ php artisan:migrate
Nothing to migrate.

prova2_db $ php artisan migrate:status
+-----+-----+
| Ran? | Migration | Batch |
+-----+-----+
| Yes | 2014_10_12_000000_create_users_table | 1 |
| Yes | 2014_10_12_100000_create_password_resets_table | 1 |
+-----+-----+
```

- Questo perché *migrate* osserva il contenuto di *database/migrations*, non i timestamp o i contenuti dei file
- Una soluzione è tornare indietro di una *migration* e poi rifarla
  - si ripeterà la migrazione, con tutte le successive modifiche allo schema del DB, compreso il cambio da '*name*' a '*username*'

# *migrate:rollback+migrate* vs. *migrate:fresh*

- Effettuiamo quindi la modifica in *database/migrations/...create\_users\_table.php* e incorporiamola in *mysql* con *migrate*:

```
prova_db $ php artisan migrate:rollback  
...  
prova_db $ php artisan migrate # si incorpora la modifica nel DB  
...
```

- Un'alternativa radicale è buttar via tutte le migrazioni passate e ricrearne una sola che rispecchia il DB come descritto dal codice PHP in *database/migrations*

```
prova_db $ php artisan migrate:fresh  
...
```

- In realtà con *migrate:fresh* si buttano via (drop) tutte le tabelle, compresa quella delle migrazioni e le si ricreano:

```
prova_db $ php artisan help migrate:fresh  
Description:  
Drop all tables and re-run all migrations  
  
prova_db $ php artisan help migrate:reset # reset non rilancia migrate  
Description:  
Rollback all database migrations
```

# Ancora migrate: vantaggi di Laravel

- A questo punto è chiaro il perché, applicando *artisan migrate* sul codice, più sviluppatori possono facilmente replicare lo stesso schema di DB del loro progetto condiviso
- Apportare numerose modifiche direttamente sul DB, "a mano", anche se con un tool GUI come *phpmyadmin*, è certamente più dispendioso in termini di tempo e più difficile da tracciare per il team di sviluppo
  - soprattutto: la modifica "a mano" del DB non lascerebbe traccia nel codice del progetto Laravel!
  - inoltre si perderebbe il version control con *migrate*

# *artisan make:migration*

- Crea un "file di migrazione"

```
prova_db $ php artisan help make:migration
Description:
  Create a new migration file
Usage:
  make:migration [options] [--] <name>
prova_db $ php artisan make:migration create_projects_table
Created Migration: 2019_05_28_124143_create_projects_table
```

- In generale *make:xxx* crea oggetti di tipo *xxx*

```
prova_db $ php artisan # omettiamo diverse righe...
make:auth          Scaffold basic login and registration views and routes
make:controller    Create a new controller class
make:event         Create a new event class
make:factory        Create a new model factory
make:listener       Create a new event listener class
make:migration      Create a new migration file
make:model          Create a new Eloquent model class
make:notification   Create a new notification class
make:observer        Create a new observer class
make:policy          Create a new policy class
make:provider        Create a new service provider class
make:request         Create a new form request class
make:resource        Create a new resource
make:rule            Create a new validation rule
make:test            Create a new test class
```

# `make:migration create_projects_table`

- L'intento finale è di creare una tabella *projects* nel DB
- In realtà, verrà solo creato un file "migrazione" `20XX..._create_projects_table.php` con un metodo `up()` che invoca `Schema::create('projects', function (Blueprint ...))`

The screenshot shows a code editor with a sidebar containing a file tree. The tree shows a project structure with a `migrations` folder containing several migration files: `2014_10_12_000000_create_users_table.php`, `2014_10_12_100000_create_password_resets_table.php`, and `2019_05_28_124143_create_projects_table.php`. The `2019_05_28_124143_create_projects_table.php` file is currently selected and shown in the main editor area. The code in this file is:

```
1 <?php
2
3 use Illuminate\Support\Facades\Schema;
4 use Illuminate\Database\Schema\Blueprint;
5 use Illuminate\Database\Migrations\Migration;
6
7 class CreateProjectsTable extends Migration
8 {
9     /**
10      * Run the migrations.
11      *
12      * @return void
13      */
14     public function up()
15     {
16         Schema::create('projects', function (Blueprint $table) {
17             $table->bigIncrements('id');
18             $table->timestamps();
19         });
20     }
21 }
```

- Per la nuova tabella, il wizard genera uno schema di default "minimale"
- Ma la migrazione non è ancora avvenuta e il DB non contiene la nuova tabella...

# Convenzione sui nomi dei record del DB

- Se la tabella è destinata a contenere record *Xyz*
- la tabella si chiamerà *xyzs* (plurale, con *-s*)
- la migrazione per la tabella: *create\_xyzs\_table*
- il modello (v. oltre) per il record: *Xyz* (singolare)

Nell'esempio in corso, *Xyz* è *Project*:

```
prova_db $ php artisan make:migration create_projects_table    # già invocato prima,  
# vediamone ora l'effetto  
prova2_db $ php artisan migrate:status  
+-----+-----+-----+  
| Ran? | Migration                                | Batch |  
+-----+-----+-----+  
| Yes  | 2014_10_12_000000_create_users_table        | 1     |  
| Yes  | 2014_10_12_100000_create_password_resets_table | 1     |  
| No   | 2019_05_30_100205_create_projects_table      |       |  
+-----+-----+-----+  
prova2_db $ ls database/migrations/2019_05_30_100205_create_projects_table.php  
database/migrations/2019_05_30_100205_create_projects_table.php  # file migrazione creato!  
# verificare con phpmyadmin che la tabella projects non è (ancora) nel DB
```

# Tabella *projects* con schema di default

```
# verificare con phpmyadmin che la tabella projects non è (ancora) nel DB
```

```
prova2_db $ php artisan migrate # solo ora si avrà un effetto sul DB (e sulla storia delle migrazioni)
```

```
Migrating: 2019_05_30_100205_create_projects_table
```

```
Migrated: 2019_05_30_100205_create_projects_table
```

```
prova2_db $ php artisan migrate:status
```

Ran?	Migration	Batch
Yes	2014_10_12_000000_create_users_table	1
Yes	2014_10_12_100000_create_password_resets_table	1
Yes	2019_05_30_100205_create_projects_table	2

```
prova2_db $ php artisan migrate:rollback
```

```
Rolling back: 2019_05_30_100205_create_projects_table
```

```
Rolled back: 2019_05_30_100205_create_projects_table
```

```
prova2_db $ php artisan migrate:status
```

Ran?	Migration	Batch
Yes	2014_10_12_000000_create_users_table	1
Yes	2014_10_12_100000_create_password_resets_table	1
No	2019_05_30_100205_create_projects_table	

```
# phpmyadmin mostra che scompare projects anche dal DB
```

# Aggiungere attributi a quelli di default

- Per la nuova tabella *projects*, il wizard *artisan make:migration* genera, nella *20XX...\_create\_projects\_table.php*, uno schema di default "minimale", che arricchiremo con campi *title* e *description* (multi-righe):

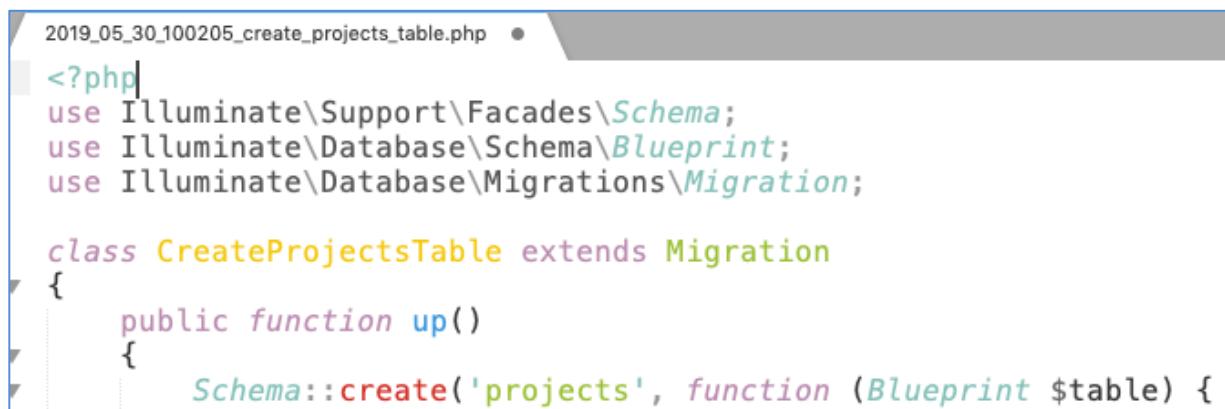
```
class CreateProjectsTable extends Migration
{
    public function up()
    {
        Schema::create('projects', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->string('title');           // aggiunto a mano
            $table->text('description');     // aggiunto a mano
            $table->timestamps();
        });
    }
}
```

- Ma lo schema nel DB non è ancora cambiato (già verificato con *phpmyadmin*) e dovrà avvenire una migrazione perché cambi:

```
prova_db $ php artisan migrate:rollback
. . .
prova_db $ php artisan migrate                      # vedere modifica nel DB con phpmyadmin
. . .
```

# Metodo up()

- A questo punto è chiaro che, per effetto di *migrate*, viene invocato il metodo *up()* della migrazione di una tabella come *projects*
- Come al solito, la dinamica dell'invocazione di *up()* non è immediata da stabilire
- Sarà *up()* della classe *CreateProjectsTable*, attraverso *Schema::create('projects', function (Blueprint ...))* a far sì che venga creata la tabella applicandole lo schema (*Blueprint*) descritto nella closure che fa da 2° argomento di *Schema::create()*



```
2019_05_30_100205_create_projects_table.php •
<?php
use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateProjectsTable extends Migration
{
    public function up()
    {
        Schema::create('projects', function (Blueprint $table) {
```

# ***down(): supporto per il rollback***

- *make:migration* definisce automaticamente, nella migrazione, il metodo *down()*, che verrà invocato per implementare il rollback
- Il default è che *down()* causi un *drop* della tabella
- Alcuni sviluppatori preferiscono introdurre un metodo *down()* vuoto (nel timore di "buttar via" con un rollback del codice utile)
  - cioè vanno solo in avanti!
- Es.: commentare il corpo di *down()* e verificare effetto di *rollback* con *phpmyadmin*
- In questo stile, se si corregge un errore, piuttosto che "tornare indietro", in realtà si fa un fork del progetto che corregga l'errore
- Anche in questo caso, di tanto in tanto, se lo si vuole, si usa *migrate:fresh* per fare "drop" di tutte le tabelle e ricrearle come specificano i file-migrazione

```
/**  
 * Reverse the migrations.  
 *  
 * @return void  
 */  
public function down()  
{  
    Schema::dropIfExists('projects');  
}
```

# **Active Record (pattern), da Wikipedia**

- L'active record pattern fornisce un approccio per l'accesso ai dati di un database (**DB**) da codice in un linguaggio orientato agli oggetti
  - Ogni tabella (o *vista*) del DB corrisponde a (è gestita come) una classe, quindi
    - un oggetto-istanza corrisponde a una riga (o *record*) della tabella
  - L'interfaccia di un oggetto conforme a questo pattern dovrebbe includere:
    - funzioni CRUD: Create, Read, Update, and Delete, nonché
    - proprietà (campi, membri...) corrispondenti (più o meno) direttamente alle colonne/attributi di una sottostante tabella di DB
- Per lo più, la classe wrapper della tabella implementa un *metodo di accesso* o una *proprietà* per ciascun attributo/colonna di una tabella o view nel DB
- Le *relazioni* (x:y) tra tabelle e le chiavi delle tabelle sono rispecchiate da apposite proprietà degli oggetti

# ***Active Record (AR): concetti***

- L'idea di fondo dell'AR pattern è che questo fornisce una visione astratta del DB e nasconde l'interazione con questo
- Dopo la creazione di un nuovo oggetto, un suo *salvataggio*, darà luogo all'aggiunta di una riga (record/individuo) alla tabella sottostante
- *Caricare* un oggetto estrae l'informazione contenuta in esso dal DB
- *Aggiornare* un oggetto fa sì che si aggiornerà la riga corrispondente in tabella
- Tutte le operazioni predette, e altre di interesse, nascondono query SQL più o meno complesse

# ***Active Record (AR): concetti***

Per ribadire i concetti introdotti:

- Il raccordo tra oggetti-AR e tavole del DB è automatico e trasparente
- Il programmatore che usa il pattern AR non ha bisogno di interagire col DB attraverso query SQL
- Queste avvengono "automaticamente" quando si *salva* un oggetto o lo si *carica* e producono gli effetti desiderati nei due sensi (su oggetto o DB)

Il pattern AR è dunque centrale per software che **implementa**:

- la persistenza per sistemi a oggetti
- un ORM (Object-Relational Mapping), cioè lo strato software che rende possibile l'astrazione dell'AR, mappando gli oggetti sulle relazioni (tra dati)

Esempi di ORM: EJB o Hibernate per Java, Eloquent di Laravel per PHP

# ORM basilare in PHP (no Laravel)

N.B.: questo lucido e il prossimo non propongono un esperimento, servono solo a illustrare il concetto di ORM

Si supponga che la libreria [php.activerecord](#) sia installata in una directory *php-activerecord* nell'*include\_path*:

- importiamo la libreria e definiamo la connessione al database:

```
require_once 'php-activerecord/ActiveRecord.php';
ActiveRecord\Config::initialize(function($cfg) {
    $cfg->set_model_directory('models');
    $cfg->set_connections(array(
        'development' => 'mysql://username:password@localhost/database_name'));
});
```

- Per un'ipotetica tabella *Users* creiamo un modello (classe) salvandolo nel file *models/User.php*

```
class User extends ActiveRecord\Model
{
    ...
}
```

# ORM basilare in PHP (no Laravel)

- Ora è semplice interagire con il database attraverso l'ORM

```
# create Tito
$user = User::create(array('name' => 'Tito', 'state' => 'VA'));

# read Tito
$user = User::find_by_name('Tito');

# update Tito
$user->name = 'Tito Jr';
$user->save();

# delete Tito
$user->delete();
```

# Active Record in Laravel

- L'implementazione degli AR è affidata al componente *eloquent*
- Il tool per generare le classi/tabelle per gli oggetti/record è:

```
prova_db $ php artisan make:model Project  
Model created successfully.
```

- è ragionevole (non un obbligo) che il modello generato corrisponda a una tabella del DB e a una *migration* per questa tabella
- il nome del modello è maiuscolo, singolare, come la classe PHP, coincide col nome della tabella, che è però minuscolo e plurale (-s)
- la classe sarà generata nella cartella della app (*prova\_db*)
- per sperimentare con l'active record di Laravel, si usa un tool che è una sorta di interprete dei comandi di Laravel (e PHP)

```
prova_db $ php artisan tinker  
>>> 2 + 2  
=> 4  
>>> echo 3+3;  
6 ↵
```

# Esperimenti con active record

```
$ php artisan tinker
Psy Shell v0.9.9 (PHP 7.3.4 - cli) by Justin Hileman
>>> App\Project::all()                                // Project è una classe/model, la corrispondente
=> Illuminate\Database\Eloquent\Collection {#2954    // tabella projects non è popolata per ora
     all: [],
}
>>> App\Project::first();
App\Project::first()
=> null
>>> App\Project::latest()->first();
=> null
>>>
```

- *all(), first() e latest()*, attraverso l'Active Record *Project* eseguono delle query sul database e precisamente sulla tabella *projects*
- NB: non sono query sulle istanze della classe *Project*, come si vede con delle classi non definite come migrazioni o definite, ma non migrate (quindi senza tabella)

# Query impossibili (non c'è la tabella)

- Classe/modello non esistente:

```
>>> App\Libro::all();
PHP Fatal error: Class 'App\Libro' not found in Psy Shell code on line 1
>>> quit
```

- Classe/modello esistente, ma tabella non presente nel DB

```
prova_db $ artisan make:model Libro
Model created successfully.
prova_db $ artisan tinker
>>> $libro = new App\Libro();
=> App\Libro {#3183}
>>> App\Libro::all();
Illuminate\Database\QueryException with message 'SQLSTATE[42S02]: Base table or view not
found: 1146 Table 'prova.libros' doesn't exist (SQL: select * from `libros`)'
>>> $libro = new App\Libro();
=> App\Libro {#3183}
>>> $libro->autore = 'Dante';
=> "Dante"
>>> $libro->titolo = 'Divina Commedia';
=> "Divina Commedia"
>>> App\Libro::all(); // Il risultato è sempre lo stesso errore: al modello Libro non corrisponde una tabella
Illuminate\Database\QueryException with message 'SQLSTATE[42S02]: Base table or view not
found: 1146 Table 'prova.libros' doesn't exist (SQL: select * from `libros`)'
```

- Istanziamo un *Project*, riempiamolo e salviamolo

```
>>> $project;
PHP Notice: Undefined variable: project in Psy Shell code on line 1
>>> $project = new App\Project;
=> App\Project {#2957}
>>> $project->titolo = 'Primo progetto';
=> "Primo progetto"
>>> $project->description = 'Cantami o diva, del Pelide Achille';
=> "Cantami o diva, del Pelide Achille"
>>> $project
=> App\Project {#2957
    titolo: "Primo progetto",
    description: "Cantami o diva, del Pelide Achille",
}
>>> $project->save();
Illuminate\Database\QueryException with message 'SQLSTATE[42S22]: Column not
found: 1054 Unknown column 'titolo' in 'field list' (SQL: insert into
`projects` (`titolo`, `description`, `updated_at`, `created_at`) values (Primo
progetto, Cantami o diva, del Pelide Achille, 2019-05-30 12:30:39, 2019-05-30
12:30:39))'
>>> $project->titolo = null;
=> null
>>> $project->title = 'Primo progetto';
=> "Primo progetto"
>>> unset($project->titolo)
>>> $project->save();
=> true
```

- Ora diventano possibili le query sulla tabella *projects*, attraverso l'Active Record di classe *Project*

```
>>> App\Project::first();
=> App\Project {#2966
    id: 1,
    title: "Primo progetto",
    description: "Cantami o diva, del Pelide Achille",
    created_at: "2019-05-30 12:30:39",
    updated_at: "2019-05-30 12:30:39",
}
>>> App\Project::first()->title;
=> "Primo progetto"
>>> App\Project::latest();
=> Illuminate\Database\Eloquent\Builder {#2966}
>>> App\Project::latest()->first();
=> App\Project {#2969
    id: 1,
    title: "Primo progetto",
    description: "Cantami o diva, del Pelide Achille",
    created_at: "2019-05-30 12:30:39",
    updated_at: "2019-05-30 12:30:39",
}
```

```

>>> App\Project::all();
=> Illuminate\Database\Eloquent\Collection {#2967
    all: [
        App\Project {#2964
            id: 1,
            title: "Primo progetto",
            description: "Cantami o diva, del Pelide Achille",
            created_at: "2019-05-30 12:30:39",
            updated_at: "2019-05-30 12:30:39",
        },
    ],
}
>>> $project = new App\Project;
=> App\Project {#2966}
>>> $project->title = 'Secondo progetto';
=> "Secondo progetto"
>>> $project->description = 'Canto l'armi e l'uomo... ';
PHP Parse error: Syntax error, unexpected T_STRING on line 1
>>> $project->description = 'Canto l\'armi e l\'uomo... ';
=> "Canto l'armi e l'uomo... "
>>> $project->save();
=> true

```

- sono i *save()* che restituiscono *true* a popolare il DB, come rivelano le prossime query *all()* etc.
- *all()* restituisce una *Collection*, che si può indicizzare (v. oltre)

```

>>> App\Project::all();
=> Illuminate\Database\Eloquent\Collection {#2946
    all: [
        App\Project {#2950
            id: 1,
            title: "Primo progetto",
            description: "Cantami o diva, del Pelide Achille",
            created_at: "2019-05-30 12:30:39",
            updated_at: "2019-05-30 12:30:39",
        },
        App\Project {#2959
            id: 2,
            title: "Secondo progetto",
            description: "Canto l'armi e l'uomo... ",
            created_at: "2019-05-30 12:41:06",
            updated_at: "2019-05-30 12:41:06",
        },
    ],
}
>>> App\Project::all()[1];
=> App\Project {#2949
    id: 2,
    title: "Secondo progetto",
    description: "Canto l'armi e l'uomo... ",
    created_at: "2019-05-30 12:41:06",
    updated_at: "2019-05-30 12:41:06",
}
>>> App\Project::all()[1]->title;
=> "Secondo progetto"
>>> App\Project::all()->map;
=> Illuminate\Support\HigherOrderCollectionProxy {#2963}
>>> App\Project::all()->map->title;
=> Illuminate\Support\Collection {#2948
    all: [
        "Primo progetto",
        "Secondo progetto",
    ],
}

```

# Un controller per il modello

- Per rispondere alle rotte riguardanti il modello, come p.es. '/*projects*', si introduce un controller
- Per convenzione, se il modello si chiama *Project*, il controller avrà classe *ProjectsController* (NB plurale)

```
<!-- web.php -->

<?php

// Routes

Route::get('/projects', 'ProjectsController@index');

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

- Il metodo del controller, invocato come callback per la rotta '/*projects*' si chiama *index*

```
<!-- ProjectsController.php -->

<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
class ProjectsController extends Controller
{
    // questo e` il boilerplate autogenerato
}
```

- Nel controller autogenerato con *artisan* però non compare il metodo *index()*

# Il ProjectsController / 1 (statico)

- Introduciamo nel controller una *index()* che restituisce la view *projects.index*, che, per convenzione, sarà il file .../resources/views/projects/index.php
  - si introduce la subdir *projects* perché altri modelli potranno avere una view chiamata *index*
  - e anche perché serviranno più view per il modello *Project*
- La view mostrata qui è statica, quindi incompleta (ci si aspetta un elenco dei progetti!)

```
<!-- ProjectsController.php -->
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;

class ProjectsController extends Controller
{
    public function index() {
        return view('projects.index');
    }
}
```

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h1>Progetti</h1>
</body>
</html>
```

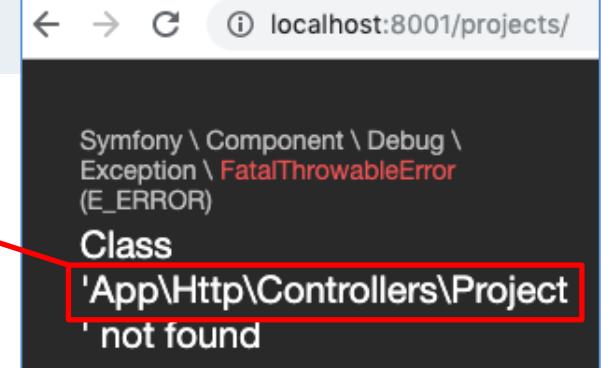
# Il ProjectsController / 2 (dinamico)

- Si può rendere la risposta dinamica restituendo l'output della query/metodo `all()` applicata al modello *Project* (per ora commentiamo via la pagina view nel file)
- Ma dov'è la classe *Project*?
- Bisogna reperirla nel *namespace* giusto, cioè: `\App!`

```
<!-- ProjectsController.php -->
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
class ProjectsController extends Controller
{
    public function index() {
        return \App\Project::all();
    }
    // return view('projects.index');
}
```

```
<!-- ProjectsController.php -->
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;

class ProjectsController extends Controller
{
    public function index() {
        return Project::all();
    }
    // return view('projects.index');
}
```

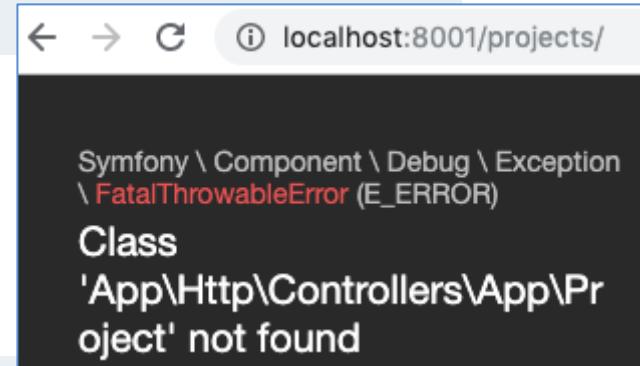


```
<!-- web.php -->
[{"id":1,"title":"Primo prog","description":"Ecco il mio primo prog....","created_at":"2019-06-04 03:48:39","updated_at":"2019-06-04 03:48:39"}, {"id":2,"title":"Secondo prog","description":"Ecco il mio secondo prog....","created_at":"2019-06-04 03:51:02","updated_at":"2019-06-04 03:51:02"}]
```

# Sui namespace e gli alias (use)

- Osserviamo che sarebbe sbagliato omettere il backslash iniziale in \App\Project::all();
- Perché collocherebbe App\Project::all() nel **namespace** App\Http\Controllers in vigore
- Bisogna invece reperirla nel *namespace* giusto (vedi fine slide precedente)!
- Oppure, con: **use** App\Project [*as Project*] si può riferire la classe \App\Project solo mediante l'alias Project
  - NB: l'arg. di **use**, pur non iniziando con "\", è assoluto e non "relativo", cioè non è interpretato rispetto al **namespace** vigente!
- In generale, per i nomi, PHP segue la cosiddetta convenzione PSR-4

```
<!-- ProjectsController.php -->
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
class ProjectsController extends Controller
{
    public function index() {
        return App\Project::all();
        // return view('projects.index');
    }
}
```



```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Project;
class ProjectsController extends Controller
{
    public function index() {
        return Project::all();
    }
}
```

# View *index* dinamica

- Installando la Chrome extension *JSON Formatter* è possibile vedere *pretty-printed* l'output di *Project::all()* (che proviene dalla *SELECT \** sul database)
- Per dare un formato più gradevole all'output (adesso una semplice variabile), il metodo *index()* del controller memorizza l'output di *all()* in **\$progetti**, che passa come parametro alla view **projects.index**, rendendola così dinamica:

```
<?php  
namespace App\Http\Controllers;  
use Illuminate\Http\Request;  
  
class ProjectsController extends Controller  
{  
    public function index()  
    {  
        $progetti = \App\Project::all();  
        return view('projects.index',  
            [ 'progetti' => $progetti ]  
        );  
    }  
}
```

```
<!- projects/index.blade.php -->  
  
<!DOCTYPE html>  
<html>  
<head>  
<title></title>  
</head>  
<body>  
    <h1>Progetti</h1>  
    {{ $progetti }}  
</body>  
</html>
```

# View dinamica / 2

- Un ulteriore miglioramento sintattico nel controller, per passare la variabile `$progetti` alla view, in cui figura il parametro chiamato anch'esso `$progetti`, è usare *compact*
- Infine, nella view, sfruttiamo PHP/blade (`@foreach`) per visualizzare come lista l'informazione dinamica ottenuta dal database con `all()` (estraendo la proprietà/attributo `->title`)

```
<!- ProjectsController.php -->

<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;

class ProjectsController extends Controller
{
    public function index() {
        $progetti = \App\Project::all();
        return view('projects.index',
            compact('progetti'))
    }
}
```

```
<!- projects/index.blade.php -->

<!DOCTYPE html>
<html>
<head><title></title></head>
<body>
    <h1>Progetti</h1>
    @foreach ($progetti as $prog)
        <li> {{$prog->title}} </li>
    @endforeach
</body>
</html>
```

# Creazione di un oggetto/record *Project*

- Si introduce ora una route */projects/create*, mappata su un metodo del controller *create()*
- Si introduce la view *projects.create* che corrisponde al file *projects/create.blade.php*
  - l'idea è di raggruppare le view relative al modello *Project* nella dir .../views/projects

```
<?php  
Route::get('/projects', 'ProjectsController@index');  
  
Route::get('/projects/create', 'ProjectsController@create');  
  
Route::get('/', function () { return view('welcome'); });  
// web.php
```

```
<?php  
namespace App\Http\Controllers;  
use Illuminate\Http\Request;  
  
class ProjectsController extends Controller  
{  
    public function index() {  
        $progetti = \App\Project::all();  
        return view('projects.index', compact('progetti'));  
    }  
    public function create() {  
        return view('projects.create');  
    }  
}  
// ProjectsController.php
```

# View *create.blade.php*

- View con form, POST e *action*
  - viene servita in risposta alla route '/*projects/create*'
  - ciò va bene, ma, quando l'utente, riempito il form, farà clic sul button *submit*
  - si avrà un errore per mancanza di una route per *method* e *action* del form
- occorre una mappatura per una route '/*projects*' e un messaggio HTTP POST
- *store* è un nome convenzionale per la funzione callback del controller

```
<!DOCTYPE html>
<html><head><title></title></head>
<body> <h1>Crea un progetto</h1>
    <form method="POST" action="/projects">
        <div><input type="text" name="title"
            placeholder="Titolo progetto"></div><p>
        <div><textarea name="description"
            placeholder="Descrizione progetto"></textarea></div><p>
        <div><button type="submit">Crea progetto</button></div>
    </form>
</body>
```

Crea un progetto

Progetto 1

Bellissimo!

Crea progetto

localhost:8001/projects/create

localhost:8001/projects

Symfony \ Component \ HttpKernel \ Exception \ MethodNotAllowedHttpException

The POST method is not supported for this route.

Supported methods: GET, HEAD.

```
<?php
Route::get('/projects', 'ProjectsController@index');
Route::post('/projects', 'ProjectsController@store');

Route::get('/projects/create', 'ProjectsController@create');

Route::get('/', function () { return view('welcome'); });
// web.php
```

# Il metodo *store()* del controller

- Prima di inserire nel controller un callback *store()*, si ha un errore **419**, l'errore Laravel di elaborazione di un POST
- Inseriamo ora *store()* vuoto: ancora **419**!
- Tentiamo allora uno *store()* che restituisca *request()*, che dovrebbe restituire una rappresentazione della richiesta, il messaggio HTTP arrivato: ancora l'errore **419**
- Come mai? è vero che il controller dovrebbe accedere ai **parametri** della richiesta POST (e non lo fa), ma perché l'errore **419**?

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
class ProjectsController extends Controller
{
    public function index() {
        $progetti = \App\Project::all();
        return view('projects.index', compact('progetti'));
    }
    public function create() {
        return view('projects.create');
    }
    public function store() {
        return request();
    }
} // ProjectsController.php
```

```
<!- create.blade.php -->
...
<body>
    ...
<form method="POST" action="/projects">
    <div><input type="text" name="title"
        placeholder="Titolo progetto"></div><p>
    ...

```

# Il metodo *store()* del controller

- La richiesta pervenuta dal cliente con POST è considerata "falsa" se non contiene un token unico, con cui il server individua il client
- Aggiungiamo allora, nella view *create*, nel form, la funzione PHP ***csrf\_field()*** (Cross-Site Request Forgery, genera token di sicurezza)
- La richiesta POST è ora sicura!

```
← → ⌂ ⓘ localhost:8001/projects
{
  "_token": "nrCCdq5hv6aBv4tgjQFwmnQB87DN2Nks29DWV6Ce",
  "title": "Progetto 3",
  "description": "Bello"
}
```

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
class ProjectsController extends Controller
{
    public function index() {
        $progetti = \App\Project::all();
        return view('projects.index', compact('progetti'));
    }
    public function create() {
        return view('projects.create');
    }
    public function store() {
        return request();
    }
} // ProjectsController.php
```

```
<!- create.blade.php -->
...
<body>
...
<form method="POST" action="/projects">
  {{ csrf_field() }}
  <div><input type="text" name="title"
    placeholder="Titolo progetto"></div><p>
  ...

```

# Il metodo *store()* del controller / 2

- Quindi *request()* (con o senza *->all()*) restituisce effettivamente la richiesta (in formato JSON)
- è istruttivo riguardare ora la pagina del form e il suo codice sorgente **con il token**:

localhost:8001/projects/create

## Crea un progetto

Titolo progetto

Descrizione progetto

Crea progetto

view-source:localhost:8001/projects/create

```
<!DOCTYPE html>
<html>
<head>
<title></title>
</head>
<body>
<h1>Crea un progetto</h1>
<form method="POST" action="/projects">
<input type="hidden" name="_token" value="nrCCdq5hv6aBv4tgjQFwmnQB87DN2Nks29DWV6Ce">
<div><input type="text" name="title" placeholder="Titolo progetto"></div><p>
<div>
<textarea name="description" placeholder="Descrizione progetto"></textarea>
</div><p>
<div><button type="submit">Crea progetto</button></div>
</form>
</body>
</html>
```

- nella *store()* del controller è possibile prelevare i singoli campi, p.es. *request()->title* o, più breve: *request('title')*:

```
...
public function store() {
    return request('title');
}
// ProjectsController.php
```

localhost:8001/projects

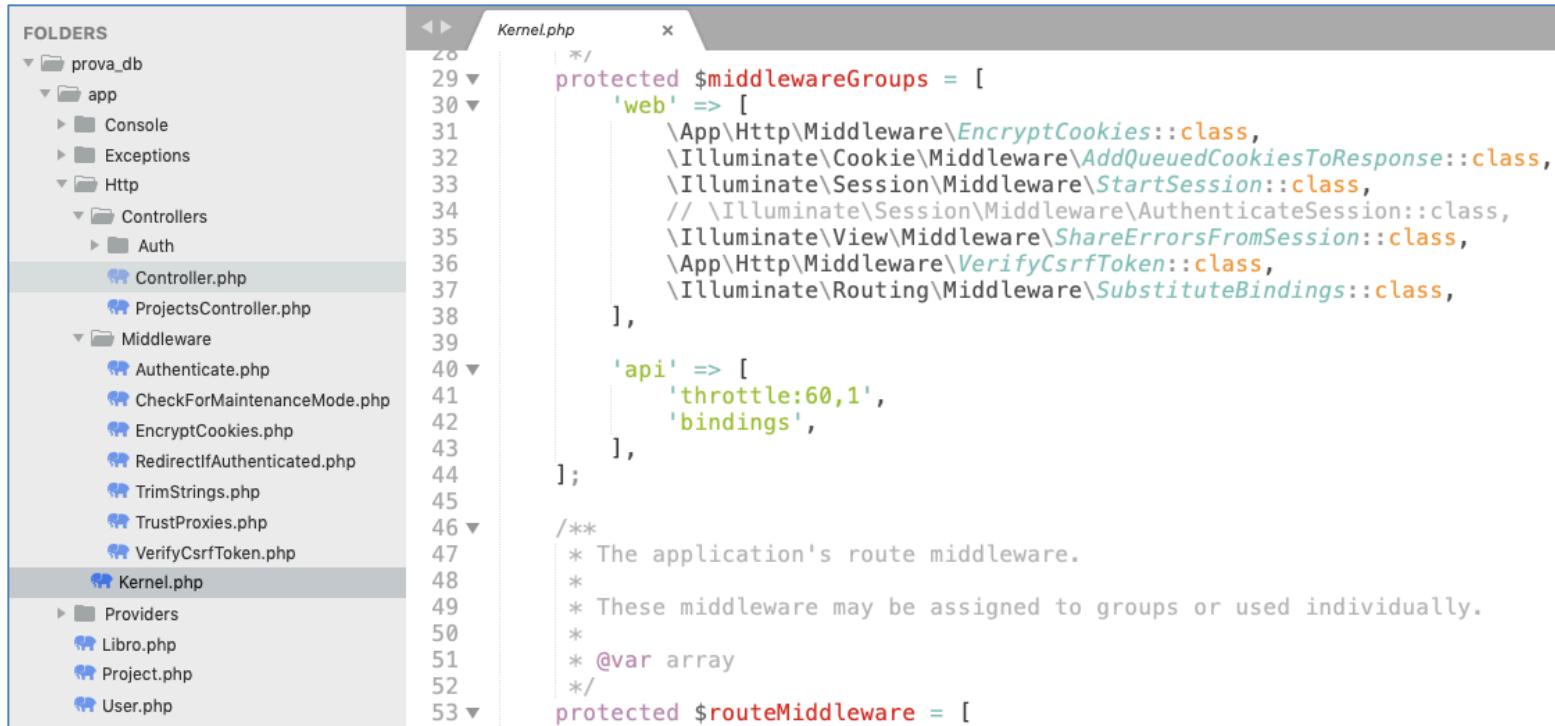
Progetto 4

# Il metodo *store()*: fatto!

- Ora che sappiamo accedere ai campi del form, possiamo istanziare un nuovo *Project*, un Active Record, cioè, e renderlo persistente
- Assegniamo i parametri *title* e *description* estratti dalla richiesta alle rispettive proprietà di *Project*
- Salviamo l'Active Record con *->save()*
- Infine, il metodo *helper redirect()* mostrerà il nuovo elenco progetti (di fatto ripropone la URL */projects* che, via route, richiama la view *index*)

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
class ProjectsController extends Controller
{
    public function index() {
        $progetti = \App\Project::all();
        return view('projects.index', compact('progetti'));
    }
    public function create() {
        return view('projects.create');
    }
    public function store() {
        $project = new \App\Project();
        $project->title = request('title');
        $project->description = request('description');
        $project->save();
        return redirect('/projects');
    }
}
// ProjectsController.php
```

# Middleware attivato dalle route in `web.php`



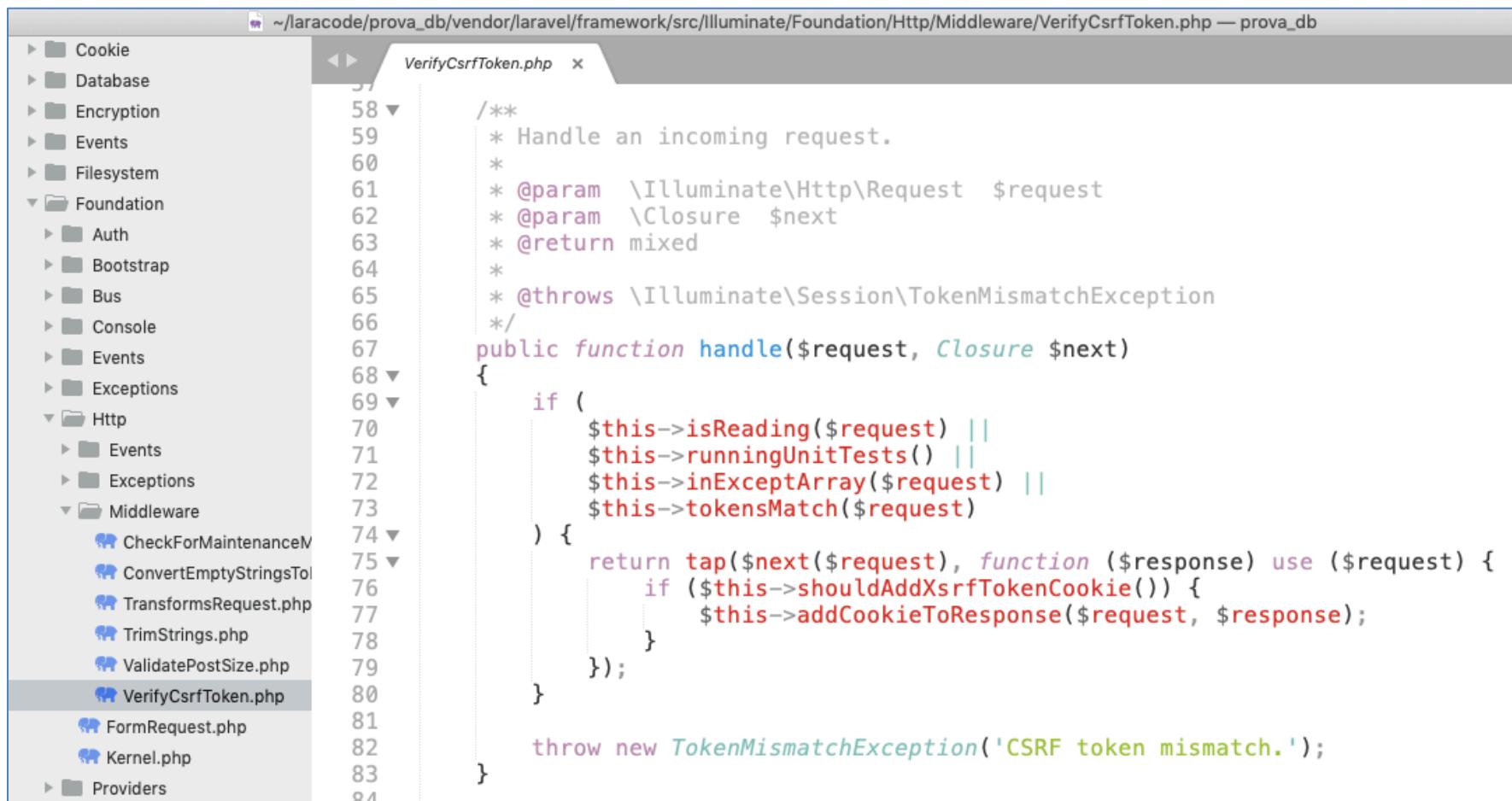
```
Kernel.php
protected $middlewareGroups = [
    'web' => [
        \App\Http\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
        \Illuminate\Session\Middleware\StartSession::class,
        // \Illuminate\Session\Middleware\AuthenticateSession::class,
        \Illuminate\View\Middleware\ShareErrorsFromSession::class,
        \App\Http\Middleware\VerifyCsrfToken::class,
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
    ],
    'api' => [
        'throttle:60,1',
        'bindings',
    ],
];
/** 
 * The application's route middleware.
 *
 * These middleware may be assigned to groups or used individually.
 *
 * @var array
 */
protected $routeMiddleware = [
```



```
VerifyCsrfToken.php
1 <?php
2
3 namespace App\Http\Middleware;
4
5 use Illuminate\Foundation\Http\Middleware\VerifyCsrfToken as Middleware;
6
7 class VerifyCsrfToken extends Middleware
8 {
```

- In realtà, quindi, bisogna guardare `Illuminate/.../VerifyCsrfToken`

# Il controllo del token CSRF



The screenshot shows a code editor with the file `VerifyCsrfToken.php` open. The file is located at `~/laracode/prova_db/vendor/laravel/framework/src/Illuminate/Foundation/Http/Middleware/VerifyCsrfToken.php`. The code implements a middleware to handle incoming requests and verify CSRF tokens.

```
58     /**
59      * Handle an incoming request.
60      *
61      * @param \Illuminate\Http\Request $request
62      * @param Closure $next
63      * @return mixed
64      *
65      * @throws \Illuminate\Session\TokenMismatchException
66      */
67     public function handle($request, Closure $next)
68     {
69         if (
70             $this->isReading($request) ||
71             $this->runningUnitTests() ||
72             $this->inExceptArray($request) ||
73             $this->tokensMatch($request)
74         ) {
75             return tap($next($request), function ($response) use ($request) {
76                 if ($this->shouldAddXsrfTokenCookie()) {
77                     $this->addCookieToResponse($request, $response);
78                 }
79             });
80         }
81     }
82
83     throw new TokenMismatchException('CSRF token mismatch.');
84 }
```

# Modelli e risorse

- Un ente che funge da "modello" per la Web app è detto (considerato) "risorsa" per l'utente
- Nell'esempio in corso, *Project* è un modello/risorsa
- Come si ricorderà, nell'interazione con la web app, il nome della risorsa, p.es. *projects*, è il primo componente di un gruppo di URL/route
- Progettiamo ora il dettaglio l'interazione con la risorsa a partire dalle rotte
- Inseriamo, per ora come commento (qui a destra), un possibile schema di rotte nel router *web.php*
  - il "metodo" della richiesta HTTP (*GET*, *POST*, ...)
  - i **componenti** della rotta dopo */projects* specificano l'operando (**1** **2** ... se c'è) e l'operazione (serve solo per distinguere la natura del *GET*)
  - in parentesi l'**operazione** "logica"
  - i *GET* richiedono dati o form
  - tutti gli altri messaggi HTTP cambiano lo stato
  - *POST* / *PATCH* quasi equivalenti;  
*POST* si usa per memorizzare un nuovo record,  
*PATCH* per un record modificato (*edited*)

```
<?php
```

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

/*
GET /projects (index) // elenco
POST /projects (store) // nel DB
GET /projects/create (create)
GET /projects/ 1 (show #)
PATCH /projects/1 (update)
DELETE /project/1 (destroy)
GET /projects/1/edit (edit)
*/
```

// di seguito vediamo per ora solo le  
// rotte già definite in precedenza

```
Route::get('/projects',
    'ProjectsController@index');

Route::post('/projects',
    'ProjectsController@store');

Route::get('/projects/create',
    'ProjectsController@create');

// web.php
```

# Quali metodi per il controller?

- Nell'esempio in corso, per il modello/risorsa *projects*, la semantica e i nomi scelti per le operazioni non sono stati scelti a caso: sono quelli standard di Laravel
- Per esempio, il metodo associato a un messaggio *GET* con URL ... */projects/* si dovrebbe chiamare *index()* , anche se nulla avrebbe vietato, p.es., *indice()*
- *index()* è la convenzione standard di Laravel e così per gli altri nomi
- Anche le funzioni dei metodi, preciseate meglio nella prossima slide, sono standard
- Si noti il gruppo di 4 metodi *CRUD*
- Comunque vedremo che, di norma, lo sviluppatore non scrive (i nomi dei metodi del) controller a mano/memoria, ma genera un template da riempire con *artisan*

```
<?php

Route::get('/', function () {
    return view('welcome');
});

/*
GET /projects (index) // elenco progetti
POST /projects (store) // nel DB, nuovo progetto

GET /projects/create (Create) // get form
GET /projects/1 (show # ovvero Read)
PATCH /projects/1 (Update) // record # del DB
DELETE /project/1 (Destroy) // record # del DB

GET /projects/1/edit (edit) // get form for
                           // editing record n.#
*/


// di seguito vediamo per ora solo le
// rotte già definite in precedenza

Route::get('/projects',
    'ProjectsController@index');

Route::post('/projects',
    'ProjectsController@store');

Route::get('/projects/create',
    'ProjectsController@create');

// web.php
```

# Route per la gestione di una risorsa

- Modifichiamo ora il file delle rotte inserendo davvero "a mano" le nuove rotte e i corrispondenti callback
- Come si vede, i callback sono altrettanti metodi del *ProjectsController*
- NB: anche se qui non lo prevediamo, può essere utile sapere che un messaggio *HEAD* avrà la risposta con: *header* esattamente come per *GET* e *body* vuoto

```
<?php  
Route::get('/', function () { return view('welcome'); });  
/*  
GET /projects (index)  
POST /projects (store)           // store into DB new project (cf. create)  
  
GET /projects/create (create) // get form da riempire per creare un project  
GET /projects/1   (show ovvero read)  
PATCH /projects/1 (update)    // update in DB record (edited project) n. #  
DELETE /project/1 (destroy)   // delete from DB record (project) n. #  
  
GET /projects/1/edit (edit)   // get form per editing del project n. #  
*/  
  
Route::get('/projects', 'ProjectsController@index');  
  
Route::get('/projects/create','ProjectsController@create');  
  
Route::get('/projects/{project}}','ProjectsController@show');  
  
Route::post('/projects', 'ProjectsController@store');  
  
Route::get('/projects/{project}/edit','ProjectsController@edit');  
  
Route::patch('/projects/{project}}', 'ProjectsController@update');  
  
Route::delete('/projects/{project}}','ProjectsController@destroy');  
  
// web.php
```

# *artisan route:list* e le rotte

```
prova_db $ php artisan | grep '^ *route:'  
route:cache          Create a route cache file for faster route registration  
route:clear          Remove the route cache file  
route:list           List all registered routes
```

# PRIMA dell'introduzione delle nuove route per la risorsa *Project*

```
prova_db $ php artisan route:list
```

Domain	Method	URI	Name	Action	Middleware
	GET HEAD	/		Closure	web
	GET HEAD	projects		App\Http\Controllers\ProjectsController@index	web
	POST	projects		App\Http\Controllers\ProjectsController@store	web
	GET HEAD	projects/create		App\Http\Controllers\ProjectsController@create	web

# DOPO l'introduzione delle route per la risorsa *Project* nel file *web.php*

```
prova_db $ php artisan route:list
```

Domain	Method	URI	Name	Action	Middleware
	GET HEAD	/		Closure	web
	GET HEAD	projects		App\Http\Controllers\ProjectsController@index	web
	POST	projects		App\Http\Controllers\ProjectsController@store	web
	GET HEAD	projects/create		App\Http\Controllers\ProjectsController@create	web
	GET HEAD	projects/{project}/edit		App\Http\Controllers\ProjectsController@edit	web
	GET HEAD	projects/{project}		App\Http\Controllers\ProjectsController@show	web
	PATCH	projects/{project}		App\Http\Controllers\ProjectsController@update	web
	DELETE	projects/{project}		App\Http\Controllers\ProjectsController@destroy	web

# Route "collettiva" per una risorsa

- Sostituiamo le singole *Route* con una sola *Route::resource()*

```
<?php

Route::get('/', function () { return view('welcome'); });

/*
GET /projects (index)
GET /projects/create (create)
POST /projects (store)
GET /projects/1 (show)
GET /projects/1/edit (edit)
PATCH /projects/1 (update)
DELETE /project/1 (destroy)
*/
/*
Route::get('/projects', 'ProjectsController@index');
Route::get('/projects/create', 'ProjectsController@create');
Route::get('/projects/{project}', 'ProjectsController@show');
Route::post('/projects', 'ProjectsController@store');
Route::get('/projects/{project}/edit', 'ProjectsController@edit');
Route::patch('/projects/{project}', 'ProjectsController@update');
Route::delete('/projects/{project}', 'ProjectsController@destroy');
*/

Route::resource('/projects', 'ProjectsController');
// web.php
```

# Route "collettiva" per risorsa / 2

```
# nel file web.php disattiviamo, per un istante, con un commento, la route collettiva per la risorsa Project
prova_db $ php artisan route:list
+-----+-----+-----+-----+-----+
| Domain | Method | URI      | Name | Action | Middleware |
+-----+-----+-----+-----+-----+
|       | GET|HEAD | /       |       | Closure | web        |
+-----+-----+-----+-----+-----+
# riattiviamo in web.php la route collettiva per la risorsa Project
prova_db $ php artisan route:list
+-----+-----+-----+-----+-----+-----+
| Domain | Method | URI          | Name | Action           | Middleware |
+-----+-----+-----+-----+-----+-----+
|       | GET|HEAD | /           |       | Closure          | web        |
|       | GET|HEAD | projects    |       | App\Http\Controllers\ProjectsController@index | web        |
|       | POST     | projects    |       | App\Http\Controllers\ProjectsController@store   | web        |
|       | GET|HEAD | projects/create |       | App\Http\Controllers\ProjectsController@create | web        |
|       | GET|HEAD | projects/{project}/edit |       | App\Http\Controllers\ProjectsController@edit   | web        |
|       | GET|HEAD | projects/{project} |       | App\Http\Controllers\ProjectsController@show    | web        |
|       | PATCH   | projects/{project} |       | App\Http\Controllers\ProjectsController@update | web        |
|       | DELETE  | projects/{project} |       | App\Http\Controllers\ProjectsController@destroy | web        |
+-----+-----+-----+-----+-----+-----+
```

- Esattamente ciò che si era ottenuto definendo a mano, la rotta per ogni operazione sulla risorsa
- Ora si dovrebbero introdurre nel controller per la risorsa i (nuovi) metodi *show()*, *destroy()*, *edit()*, *update()*
- Per *index()* *store()* *create()*, vedi slide n. [67. Il metodo store\(\): fatto!](#)

# Wizard per boilerplate nel controller

```
prova_db $ php artisan route:list
```

Domain	Method	URI	Name	Action	Middleware
	GET HEAD	/	Closure		web
	GET HEAD	projects	App\Http\Controllers\ProjectsController@index		web
	POST	projects	App\Http\Controllers\ProjectsController@store		web
	GET HEAD	projects/create	App\Http\Controllers\ProjectsController@create		web
	GET HEAD	projects/{project}/edit	App\Http\Controllers\ProjectsController@edit		web
	GET HEAD	projects/{project}	App\Http\Controllers\ProjectsController@show		web
	PATCH	projects/{project}	App\Http\Controllers\ProjectsController@update		web
	DELETE	projects/{project}	App\Http\Controllers\ProjectsController@destroy		web

Nel controller, i template *boilerplate* per *index()*, *store()*, *create()*, *edit()*, *show()*, *update()*, *destroy()* possono essere generati da un wizard:

```
prova_db $ php artisan help make:controller
```

**Description:**

Create a new controller class

**Usage:**

`make:controller [options] [--] <name>`

**Arguments:**

`name` The name of the class

**Options:**

`-r, --resource` Generate a resource controller class.

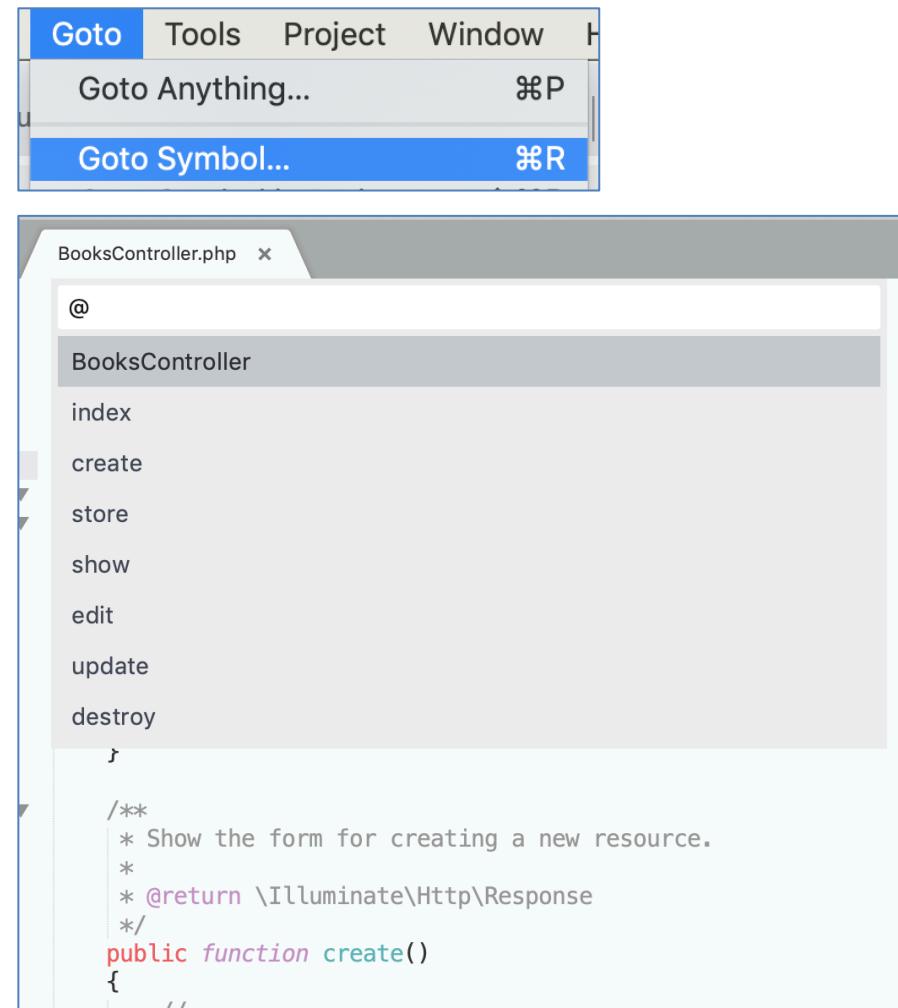
...

# Boilerplate nel controller: esempio (risorsa Book)

```
prova_db $ php artisan make:controller BooksController -r  
Controller created successfully.
```



```
BooksController.php x  
<?php  
  
namespace App\Http\Controllers;  
  
use Illuminate\Http\Request;  
  
class BooksController extends Controller  
{  
    /**  
     * Display a listing of the resource.  
     *  
     * @return \Illuminate\Http\Response  
     */  
    public function index()  
    {  
        //  
    }  
  
    /**  
     * Show the form for creating a new resource.  
     *  
     * @return \Illuminate\Http\Response  
     */  
    public function create()  
    {  
        //  
    }  
}
```



Goto Tools Project Window Help

Goto Anything... ⌘P

Goto Symbol... ⌘R

BooksController.php x

```
@  
BooksController  
index  
create  
store  
show  
edit  
update  
destroy  
    /**  
     * Show the form for creating a new resource.  
     *  
     * @return \Illuminate\Http\Response  
     */  
    public function create()  
    {  
        //
```

# Controller da wizard: vantaggi

I vantaggi del wizard `make:controller -r` sono evidenti

- usa uno schema regolare ed ortogonale per i nomi dei metodi *callback*
- evita che, involontariamente, si diano ai metodi nomi non conformi a questo schema
- evita che proliferino file PHP, ciascuno col codice di uno (o più) callback e nomi più o meno sensati
  - ciò era tipico della app PHP "vecchio stile"
  - ora, con Laravel e il wizard, i callback sono tutti centralizzati nel controller e hanno nomi "regolari", secondo lo schema discusso

# Laravel: dove siamo arrivati / 1

Proviamo a fare il punto su ciò che si è visto finora di Laravel:

- una app è costruita intorno al concetto di *modello/risorsa*, p.es. *Project*
- l'interfaccia utente "REST" sfrutta richieste HTTP che specificano l'operazione e l'eventuale operando attraverso il metodo HTTP e la URL
- Laravel, con *artisan*, propone sintassi e semantica standard per associare richieste REST a URL e operazioni standard, nel file *web.php* delle *route*
- Le 7 rotte standard per un modello/risorsa *Project*
- Laravel, con *artisan make:controller*, propone un'organizzazione standard delle operazioni come metodi di una classe detta controller
- Nel controller standard occorre implementare essenzialmente:
  - le 4 operazioni CRUD (Create, Read, Update, Destroy o Delete) tenendo presente che Laravel chiama "Read" col nome "show"
  - *store* che fa da supporto a *Create* (*Create* mostra un form, *store* memorizza i nuovi dati)
  - *edit* che mostra il form per modificare i dati che *update* memorizzerà
  - *index* che, in risposta alla route base, mostra l'elenco delle risorse disponibili

```
<?php
Route::get('/', function () {
    return view('welcome');
});

/*
GET /projects (index) // elenco progetti
POST /projects (store) // nel DB, nuovo progetto

GET /projects/create (create) // get form
GET /projects/1 (show # ovvero read)
PATCH /projects/1 (update) // record # del DB
DELETE /project/1 (destroy) // record # del DB

GET /projects/1/edit (edit) // get form for
                           // editing record n.#
*/
Route::resource('/projects',
    'ProjectsController');

// web.php
```

# Laravel: dove siamo arrivati / 2

Idea di base: se la route *projects/XXX* è associata all'operazione (*metodo*) *xxx()* del controller:

- *xxx()* esegue la logica di business richiesta sul DB, sfruttando il concetto di *active record*
- poi *xxx()* invoca una view opportuna, detta probabilmente *projects.xxx*

Se la route *projects/1/XXX* ha un operando (1), *xxx()* ha un parametro corrispondente e lo passa alla view

Le view, che sfruttano Blade, mostrano tipicamente:

1. form per l'input dei dati, oppure
2. output di risultati

```
<?php
Route::get('/', function () {
    return view('welcome');
});
/*
GET /projects (index) // elenco progetti
POST /projects (store) // nel DB, nuovo progetto

GET /projects/create (create) // get form
GET /projects/1 (show # ovvero read)
PATCH /projects/1 (update) // record # del DB
DELETE /project/1 (destroy) // record # del DB

GET /projects/1/edit (edit) // get form for
                           // editing record n.#
*/
Route::resource('/projects',
    'ProjectsController');

// web.php
```

# Wizard: anche per il model

- Consideriamo ancora una data risorsa/modello con le operazioni CRUD
- Come detto, Laravel col suo wizard porta ad adottare, per i callback delle operazioni, uno schema di naming comune e una "sede" centralizzata nel file del controller – questo è conveniente!
- Ma può servire anche altro: alcuni callback standard, es. `show()`, hanno un argomento che rappresenta l'istanza di risorsa su cui operano
  - di base quest'argomento è la chiave numerica dell'istanza, ma può convenire (a scelta dello sviluppatore) sia invece l'istanza stessa
  - inoltre, occorre definire il modello (classe) per la risorsa
- Anche per questi fini si può ricorrere al wizard:

```
prova_db $ php artisan help make:controller
Usage: make:controller [options] [--] <name>
Arguments: name          The name of the class
Options:
  -m, --model[=MODEL]    Generate a resource controller for the given model.
  -r, --resource         Generate a resource controller class.
```

# Wizard per controller e model / 2

- Immaginiamo di voler introdurre una risorsa *Post* (p.es. per i post su un blog), ricorrendo al wizard come visto:

```
prova_db $ php artisan make:controller PostsController -r -m Post
A App\Post model does not exist. Want to generate it? (yes/no) [yes]:
> yes
Model created successfully.
Controller created successfully.
```

- comparirà un template per la classe/modello `\App\Post.php` (e `use App\Post` nel `PostsController`)
- nel controller, i boilerplate per i vari callback ricevono un argomento *active record* `Post $post` (cf. `show()` qui a destra)
  - ciò consente interazioni "automatiche", tipo "active record", col DB... (cf. oltre)
- senza l'opzione `-m` del wizard, i callback avranno per argomento l'id numerico dell'istanza, come qui sotto a destra

```
Post.php
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Post extends Model
{
    //
}
```

```
/*
 * Display the specified resource.
 *
 * @param \App\Post $post
 * @return \Illuminate\Http\Response
 */
public function show(Post $post)
{
    //
}
```

```
/*
 * Display the specified resource.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function show($id)
{
    //
}
```

# Problema: HTTP PATCH/PUT/DELETE

- Non tutti i browser gestiscono appieno questi tipi di messaggi HTTP, che però servono per attivare le route *edit/delete*
- Proviamo intanto ad aggiungere i metodi *edit()*, *update()*, *destroy()* al *ProjectsController*
- Per iniziare, riempiamo, p.es., *edit()* *update()* *delete()* in maniera convenzionale, a mano (NB: senza argomento)

```
...
public function create() {
    return view('projects.create');
}

public function store() {
    $project = new Project();
    $project->title = request('title');
    $project->description = request('description');
    $project->save();
    return redirect('/projects');
}

public function edit()
{
    // da aggiungere dopo
}

public function update()
{
    // da aggiungere dopo
}

public function delete()
{
    // da aggiungere dopo
}
...
```

# HTTP PATCH/DELETE: route

- Riconsideriamo le rotte per *edit* e i metodi (messaggi) HTTP

```
prova_db $ php artisan route:list
```

Domain	Method	URI	Name	Action	Middleware
	GET HEAD	projects/{project}/edit		App\Http\Controllers\ProjectsController@edit	web
	PATCH	projects/{project}}		App\Http\Controllers\ProjectsController@update	web
	DELETE	projects/{project}}		App\Http\Controllers\ProjectsController@destroy	web

- L'idea è che in risposta a un *GET* dalla URL *HOST/projects/1/edit* la app invochi il callback *edit()* e questo mostri una vista *projects.edit* con un form per l'editing del *Project* n. 1
- Modificato il form, il browser/utente invia i nuovi valori con messaggio HTTP *PATCH* e URL *HOST/projects/1*
  - ciò attiva il callback *update()* che modifica il record 1
- Similmente per HTTP *DELETE*, URL *HOST/projects/1*, callback *delete()*

```
...
public function edit() // risponde a: HOST/projects/1/edit
{
    return view('projects.edit');
}

public function update() // risponde a: HOST/projects/1 (PATCH)
{
    // modifica il record del DB
    // individuato da callback/view per edit
}

public function delete() // risponde a: HOST/projects/1 (DELETE)
...
```

# Il template layout

- Il file *layout.blade.php* farà da template per *edit.blade.php*:

The screenshot shows a Mac OS X desktop environment. On the left, a file browser window displays the project structure:

- Http
- Providers
- Post.php
- Project.php
- User.php
- bootstrap
- config
- database
- public
- resources
  - js
  - lang
  - sass
- views
  - projects
    - create.blade.php
    - edit.blade.php
    - index.blade.php
    - layout.blade.php
    - welcome.blade.php

- come già visto, *yield('content')* è la parte da istanziare
- adesso la si istanzierà nella view *edit.blade.php*, che, in risposta alla URL *HOST/projects/1/edit*, dovrebbe mostrare un form col record n. 1 e permetterne editing e invio

# View/callback *edit* e tabella *projects*

```
public function edit()      // risponde a: HOST/project/1/edit
{
    // e dovrebbe mostrare record 1
    $project = Project::find(1); // 1 va parametrizzato
    return view('projects.edit'); // solo un abbozzo: la view
    // dovrà dipendere da $project
}

// file ProjectsController.php
```

```
@extends('layout')

@section('content')
<h1>Edit Project</h1>
{{-- MANCA IL FORM! --}}
@endsection
{{--projects/edit.blade.php--}}
```

- Da un precedente esercizio, è già presente la tabella *projects*, lo si può verificare con la Web app *phpmyadmin* (già installata)

SELECT \* FROM `projects`

		id	title	description
<input type="checkbox"/>	<a href="#">Edit</a> <a href="#">Copy</a> <a href="#">Delete</a>	1	My first project	Lorem ipsum
<input type="checkbox"/>	<a href="#">Edit</a> <a href="#">Copy</a> <a href="#">Delete</a>	2	My second project	Lorem ipsum
<input type="checkbox"/>	<a href="#">Edit</a> <a href="#">Copy</a> <a href="#">Delete</a>	3	My new project	Very special description here
<input type="checkbox"/>	<a href="#">Edit</a> <a href="#">Copy</a> <a href="#">Delete</a>	4	Some project	Some text Some more text

- Vediamo ora la risposta alla rotta *HOST.../projects/2/edit*
- è incompleta! Manca il form, perché sono ancora incompleti la view *edit* e il callback



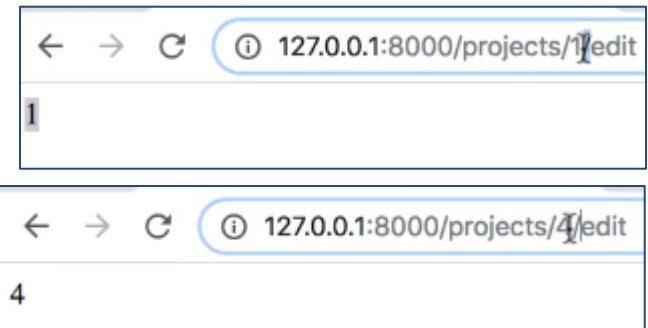
# Callback `edit()` con parametro

Torniamo al controller,  
callback `edit()`,  
bypassiamo la view e  
il `find()`, aggiungiamo un  
parametro `$id` e proviamo a restituirlo...

Come si vede a destra, a run-time, il  
parametro prende il valore dalla URL!  
E questo semplice callback lo restituisce.

Ora complichiamo il callback: cerchiamo  
(con `find($id)`) nel DB l'active record con  
chiave `$id`, lo salviamo  
in `$project` e lo  
restituiamo (in JSON):

```
public function edit($id) // risponde a: HOST/project/1/edit
{
    // e assegna 1 (...) a $id
    return $id;
// $project = Project::find($id);
// return view('projects.edit');
}
```



```
→ C ⓘ localhost:8000/projects/2/edit
{
    "id": 2,
    "title": "My second project",
    "description": "Lorem ipsum",
    "created_at": "2019-06-16 22:28:59",
    "updated_at": "2019-06-16 22:28:59"
}
```

```
public function edit($id)
{
    $project = Project::find($id);
// return view('projects.edit');
    return $project;
}
```

# `edit()`: parametro da URL a DB e view

Lo scopo per cui il callback pone in `$project` l'active record corrispondente nel DB all'`id` nella URL è di passarlo alla view (come parametro `compact` di `blade`)

Nella view `edit` introduciamo ora un form che visualizzi titolo e descrizione del progetto:

- l'HTML si può copiare dalla view `create`, con qualche aggiustamento (si ignori `POST` per ora)
- i campi ora avranno per `contenuto` i valori del record `$project` passato dal callback
- sono cioè i valori correnti da modificare

```
public function edit($id)
{
    $project = Project::find($id);
    return view('projects.edit',
                compact('project'));
}

@extends('layout')

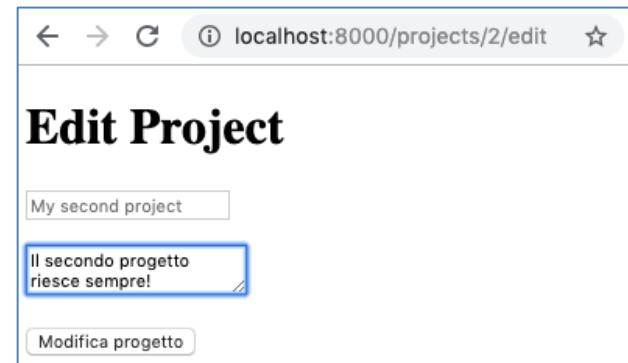
@section('content')
    <h1>Edit Project</h1>
    <form method="POST" action="/projects">
        {{ csrf_field() }}
        <div><input type="text" name="title"
                  value="{{ $project->title }}">
        </div><p>
        <div>
            <textarea name="description"
                      value="{{ $project->description }}></textarea>
        </div><p>
        <div><button type="submit">
            Modifica progetto</button>
        </div>
    </form>
    @endsection

{{-- view projects/edit.blade.php --}}
```

# view edit e messaggio HTTP

Nella view per la URL `.../projects/2/edit` si provi a modificare il form *description*

- ora un clic sul bottone "Modifica progetto" dovrebbe attivare la modifica sul DB
- il meccanismo ci è suggerito da:



```
prova_db $ php artisan route:list
+-----+-----+-----+-----+-----+
| Domain | Method | URI           | Name | Action          | Middleware |
+-----+-----+-----+-----+-----+
|       | GET|HEAD | projects/{project}/edit |      | App\Http\Controllers\ProjectsController@edit | web        |
|       | PATCH   | projects/{project} } |      | App\Http\Controllers\ProjectsController@update | web        |
+-----+-----+-----+-----+-----+
```

Quindi nella view *edit* vanno modificati *method* e *action*

- l'effetto si vedrà nel sorgente
- ... ma non è quello desiderato
- si veda la prossima slide

```
@extends('layout')

@section('content')
<h1>Edit Project</h1>
<form method="PATCH"
      action="/projects/{{ $project->id }}"
      {{ csrf_field() }}>
...
@endsection

{{-- view projects/edit.blade.php --}}
```

# view edit: tentativo di *update()*

The screenshot shows a browser window with the URL `localhost:8000/projects/2/edit`. The page title is "Edit Project". On the left, there's a text area with "My second project" and another with "Il secondo progetto riesce sempre!". Below them is a button labeled "Modifica progetto". A red arrow points from this button to the developer tools' Elements tab, which displays the HTML structure of the page. The form has a method attribute set to "PATCH". In the bottom right corner, there's a terminal window showing a stack trace for a `BadMethodCallException`. The exception message is "Method App\Http\Controllers\ProjectsController@show does not exist." The stack trace points to the `Illuminate\Routing\Route::callAction` method.

```
<!doctype html>
<html>
  <head>...
  </head>
  <body> == $0
    <div class="container">
      <h1>Edit Project</h1>
      <form method="PATCH" action="/projects/2">
        <input type="hidden" name="_token" value="BXBnURS20q0CUeJzvAFkDrXySdAh7IcuwM1lsZey">
        <div>
          <input type="text" name="title" value="My second project">
        </div>
        <p></p>
        <div>
          <textarea name="description">Lorem ipsum</textarea>
        </div>
      </form>
    </div>
  </body>
</html>
```

```
BadMethodCallException
Method
App\Http\Controllers\ProjectsController@show does not exist.

/Users/gp/laracode/prova_db/vendor/laravel/framework/src/Illuminate/Routing/
50.     * @return \Symfony\Component\HttpFoundation\Response
51.     */
52.     public function callAction($method, $parameters)
53.     {
54.         return call_user_func_array([$this, $method], $parameters);
```

Ricordiamo ancora una volta le route, e precisamente:

prova_db \$ php artisan route:list					
Domain	Method	URI	Name	Action	Middleware
	GET HEAD	projects/{project}		App\Http\Controllers\ProjectsController@show	web
	PATCH	projects/{project}		App\Http\Controllers\ProjectsController@update	web

Dato che il browser ha ignorato *PATCH* e, per default, inviato *GET*, la app attiverà *show()*, che non abbiamo ancora definito...

Errore:

- callback **show()** assente nel controller
- il problema: il browser ignora *PATCH* e invia *GET*

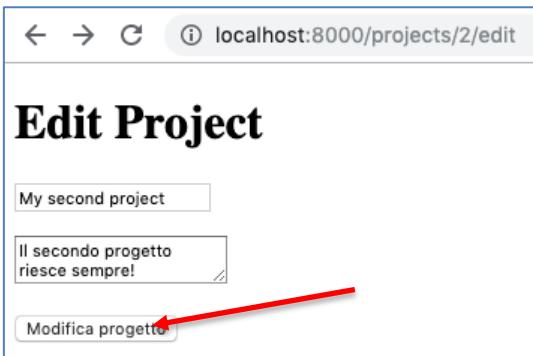
← → ⌂ ⓘ localhost:8000/projects/2/edit

## Edit Project

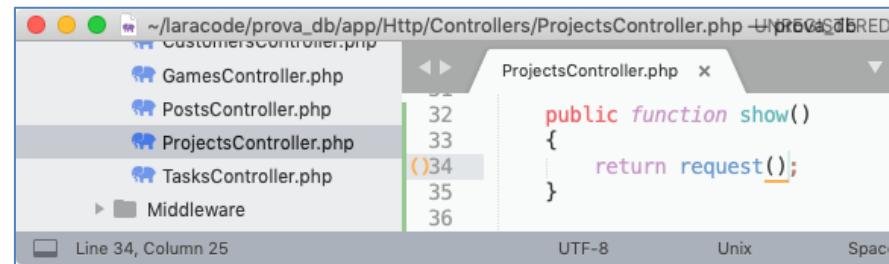
My second project

Il secondo progetto riesce sempre!

[Modifica progetto](#)



# Un rudimentale callback `show()`



```
public function show()
{
    return request();
}
```

Line 34, Column 25      UTF-8      Unix      Space

Ecco l'effetto, prodotto dal nuovo callback `show()`:

Ma occorre inviare una *PATCH*:

- si invia una comune *POST* con tipo *hidden* e parametro di *name* "`_method`" e *value* "*PATCH*"  
- ciò si può anche ottenere con il metodo PHP *method\_field()*
- l'app Laravel saprà così di dover reagire come per una richiesta *PATCH*, cioè con *update()*

← → ⌂ ⓘ localhost:8000/projects/2?title=My+second+project&description=Il+seco

```
{
    "_token": "BXBnURS20qOCUeJzvAFkDrKySdAh7IcuwM1lsZey",
    "title": "My second project",
    "description": "Il secondo progetto riesce sempre!"
}
```

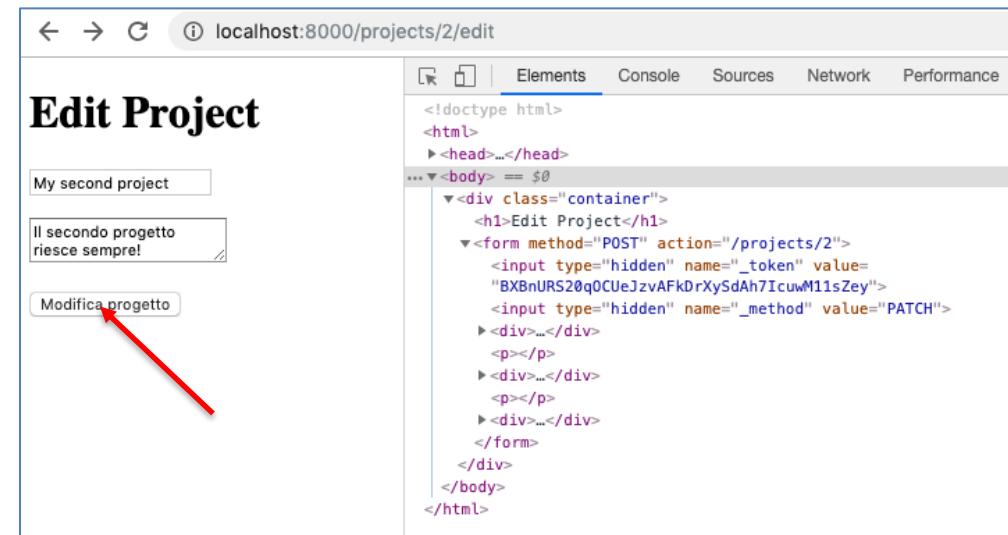
← → ⌂ ⓘ localhost:8000/projects/2/edit

## Edit Project

My second project

Il secondo progetto riesce sempre!

[Modifica progetto](#)



```
<!doctype html>
<html>
  <head>...</head>
  ...<body> == $0
    <div class="container">
      <h1>Edit Project</h1>
      <form method="POST" action="/projects/2">
        <input type="hidden" name="_token" value="BXBnURS20qOCUeJzvAFkDrKySdAh7IcuwM1lsZey">
        <input type="hidden" name="_method" value="PATCH">
      </form>...
    </div>...
  </body>
</html>
```

# Aggiornare un record: il callback *update()*

Iniziamo con un semplice *dump-and-die*, cioè *dd()*, della *request()* che arriva al *ProjectsController*

Come si vede, ora la richiesta è quella che serve.

Ora un *update()* che faccia il lavoro, attraverso il pattern *active record*:

```
// file ProjectsController.php
...
public function update($id) {
    $project = Project::find($id);
    $project->title = request('title');
    $project->description = request('description');
    $project->save();
    return redirect('/projects');
}
```

L'effetto "persistente" si può verificare anche con *phpmyadmin*:

The screenshot illustrates the development process for updating a database record using Laravel's Active Record pattern.

- Code Editor:** Shows the `ProjectsController.php` file with the `update()` method containing a `dd()` statement to dump the `request()` object.
- Browser Output:** Shows the URL `localhost:8000/projects/2`. The response is a JSON dump of the `request` object, which includes the token, method (PATCH), title ('Secondo progetto'), and description ('Il secondo progetto\r\nriesce sempre!').
- Browser View:** Shows the `localhost:8000/projects` page with a list of projects. The second project, titled 'Secondo progetto', is visible.
- phpMyAdmin Screenshot:** Shows the `projects` table with two rows. The first row has `id: 1`, `title: My first project`, and `description: Lorem ipsum`. The second row has `id: 2`, `title: Secondo progetto`, and `description: Il secondo progetto riesce sempre!`.

# Cancellazione: view e HTML

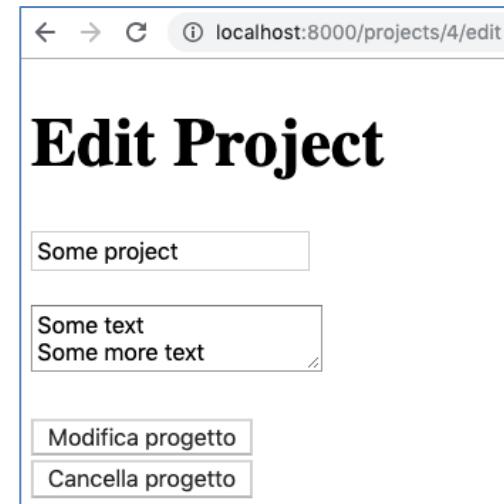
Ricordando ancora una volta le route:

```
prova_db $ php artisan route:list
```

Domain	Method	URI	Name	Action	Middleware
	GET HEAD	projects/{project}/edit		App\Http\Controllers\ProjectsController@edit	web
	PATCH	projects/{project}}		App\Http\Controllers\ProjectsController@update	web
	DELETE	projects/{project}}		App\Http\Controllers\ProjectsController@destroy	web

La cancellazione si può gestire anche dalla view col form attivato da *edit*, introducendo un bottone che invii *POST* con un *DELETE* nascosto; rispetto al codice visto per *PATCH/update*, usiamo *helper blade* più concisi

```
...
<h1>Edit Project</h1>
<form method="POST" action="/projects/{{ $project->id }}">
    {{ csrf_field() }}
    {{ method_field('PATCH') }}
    ...
    <button type="submit">Modifica progetto</button>
</form>
<form method="POST" action="/projects/{{ $project->id }}">
    @csrf
    @method('DELETE')
    <div>
        <button type="submit">Cancella progetto</button></div>
    </form>
{{-- edit.blade.php --}}
```



# Cancellazione: callback

Ricordando ancora una volta le route:

```
prova_db $ php artisan route:list
```

Domain	Method	URI	Name	Action	Middleware
	GET HEAD	projects/{project}/edit		App\Http\Controllers\ProjectsController@edit	web
	DELETE	projects/{project}		App\Http\Controllers\ProjectsController@destroy	web

Ecco il callback e, sotto a destra, il suo effetto, al clic sul bottone "Cancella progetto" nella view *edit* con URL <http://localhost:8000/projects/4/edit>

```
public function destroy($id)
{
    Project::find($id)->delete();
    return redirect('/projects');
}
```

Progetti

- My first project
- Secondo progetto
- My new project
- Some project

Progetti

- My first project
- Secondo progetto
- My new project

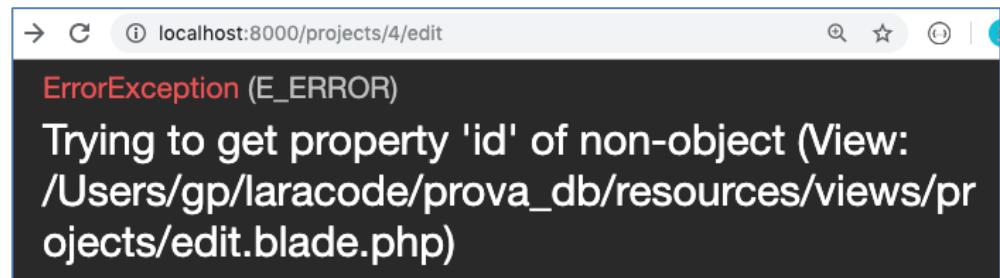
	<input type="text"/> Edit	<input type="button" value="Copy"/>	<input type="button" value="Delete"/>	<b>id</b>	<b>title</b>	<b>description</b>
<input type="checkbox"/>	<input type="button" value="Edit"/>	<input type="button" value="Copy"/>	<input type="button" value="Delete"/>	1	My first project	Lorem ipsum
<input type="checkbox"/>	<input type="button" value="Edit"/>	<input type="button" value="Copy"/>	<input type="button" value="Delete"/>	2	Secondo progetto	Il secondo progetto riesce sempre!
<input type="checkbox"/>	<input type="button" value="Edit"/>	<input type="button" value="Copy"/>	<input type="button" value="Delete"/>	3	My new project	Very special description here
<input type="checkbox"/>	<input type="button" value="Edit"/>	<input type="button" value="Copy"/>	<input type="button" value="Delete"/>	4	Some project	Some text Some more text

	<input type="text"/> Edit	<input type="button" value="Copy"/>	<input type="button" value="Delete"/>	<b>id</b>	<b>title</b>	<b>description</b>
<input type="checkbox"/>	<input type="button" value="Edit"/>	<input type="button" value="Copy"/>	<input type="button" value="Delete"/>	1	My first project	Lorem ipsum
<input type="checkbox"/>	<input type="button" value="Edit"/>	<input type="button" value="Copy"/>	<input type="button" value="Delete"/>	2	Secondo progetto	Il secondo progetto riesce sempre!
<input type="checkbox"/>	<input type="button" value="Edit"/>	<input type="button" value="Copy"/>	<input type="button" value="Delete"/>	3	My new project	Very special description here
	<input type="button" value="Check all"/>	<input type="checkbox"/>	<input type="button" value="With selected:"/>		<input type="button" value="Edit"/>	<input type="button" value="Copy"/>
					<input type="button" value="Delete"/>	<input type="button" value="Export"/>

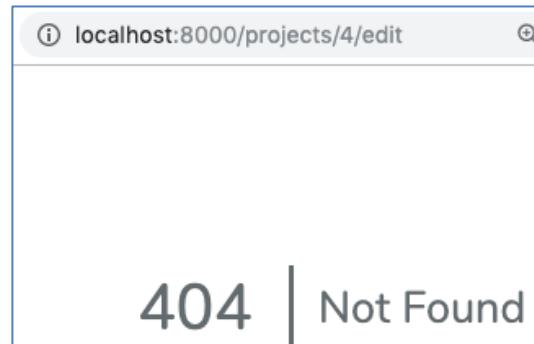
# Accesso a record inesistente: *findOrFail()*

- Chiamate a *find(\$id)*, tipicamente dai callback di un controller, causano errori se non esiste un record con l'*\$id* richiesto (cf. qui a destra in alto)
- Per evitarli, è meglio usare *findOrFail(\$id)*, come qui a destra in basso

```
public function edit($id) // risponde a: HOST/project/1/edit  
{  
    // e mostra form per 1 (in gener. $id)  
    $project = Project::find($id);  
    return view('projects.edit', compact('project'));  
}
```



```
public function edit($id) // risponde a: HOST/project/1/edit  
{  
    // e mostra form per 1 (in gener. $id)  
    $project = Project::findOrFail($id);  
    return view('projects.edit', compact('project'));  
}
```



# Route e Callback `show()`: accesso da `index`

Ripartiamo ancora dalle route, resta il callback `show()`:

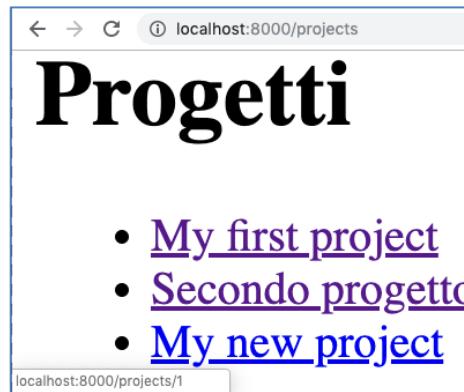
```
prova_db $ php artisan route:list
```

Domain	Method	URI	Name	Action	Middleware
	GET HEAD	projects		App\Http\Controllers\ProjectsController@index	web
	GET HEAD	projects/{project}		App\Http\Controllers\ProjectsController@show	web

Per comodità, rendiamo l'elenco di progetti restituito da `/projects – index()` cliccabile come link verso `/projects/id – show(id)` (con il codice evidenziato):

```
<!-- projects/index.blade.php -->

<!DOCTYPE html>
<html>
<head><title></title></head>
<body>
    <h1>Progetti</h1>
    <ul>
        @foreach ($prog as $progetti)
            <li> <a href="/projects/{{ $prog->id }}">
                {{ $prog->title }} </a>      </li>
        @endforeach
    </ul>
</body>
</html>
```



# Callback `show()` e view

Iniziamo con un esperimento interessante:

- callback `show($id)` che, anziché passare a una view, restituisce l'active record
- callback `show(Project $project)` che si limita a restituire `$project`

In entrambi i casi, il risultato è lo stesso (qui a dx)!

```
public function show($id) {  
    $project = Project::find($id);  
    return $project;  
}
```

```
public function show(Project $project)  
{  
    return $project;  
}
```

```
← → C ⓘ localhost:8000/projects/3  
▼ {  
    "id": 3,  
    "title": "My new project",  
    "description": "Very special description here",  
    "created_at": "2019-06-16 22:29:24",  
    "updated_at": "2019-06-16 22:29:24"  
}
```

Il codice di `show($id)` "finale" presuppone una view omologa `projects.show` cui si passa l'active record di chiave `$id`

La view (a destra) visualizza il record e ha un link verso la route `edit`

```
← → C ⓘ localhost:8000/projects/1  
Progetto: "My first project"  
Descrizione:  
Lorem ipsum  
Modifica questo progetto
```

```
public function show($id)  
{  
    $project = Project::find($id);  
    return view('projects.show', compact('project'));  
}
```

```
@extends('layout')  
  
@section('content')  
  
<h2>Progetto: &ldquo;{{ $project->title }}&rdquo;</h2>  
<div>Descrizione:<p>  
    {{ $project->description }}</p></div>  
<a href="/projects/{{ $project->id }}/edit">  
    Modifica questo progetto </a>  
<endsection>  
<!-- file projects/show.blade.php -->
```

# Model binding nei callback

"*Model binding*" significa che Laravel consente, per una data risorsa/modello, di:

- gestire rotte che contengono *id* numerico
- dare al callback un parametro che, anziché valore numerico (*id*), ha per tipo il modello
- nell'esecuzione del callback il parametro assumerà il valore del record di database che ha per chiave quell'*id*
  - se tale record non esiste, si ha un errore 404

Si può anche personalizzare il model binding, p.es. perché sia pilotato da un attributo diverso dalla chiave *\$id*, cf. documentazione:

<https://laravel.com/docs/routing#route-model-binding>

Sfruttando il model binding si può semplificare il codice dei callback che prevedono un argomento, come mostrato qui a destra:

```
public function edit(Project $project)
{
    return view('projects.edit',
        compact('project'));
}

public function update(Project $project)
{
    $project->title = request('title');
    $project->description =
        request('description');
    $project->save();
    return redirect('/projects');
}

public function show(Project $project)
{
    return view('projects.show',
        compact('project'));
}

public function destroy(Project $project)
{
    $project->delete();
    return redirect('/projects');
}
```

# Metodo callback *create()*

```
public function store() {  
    $project = new Project();  
    $project->title = request('title');  
    $project->description =  
        request('description');  
    $project->save();  
    return redirect('/projects');  
}
```

Un'altra facility dei modelli è il metodo *create()*, che si può impiegare nel callback *store()* (a dx):

```
public function store() {  
    Project::create([  
        'title' => request('title'),  
        'description' =>  
            request('description')  
    ]);  
    return redirect('/projects');  
}
```

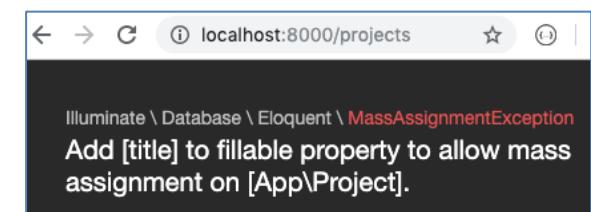
L'argomento array hash di *create()* consente di:

- istanziare il modello, creando l'active record
- assegnare **in massa** tutti gli attributi dell'active record
- e salvarlo nel database (renderlo persistente)

senza che si debba nemmeno memorizzare un riferimento all'istanza creata!

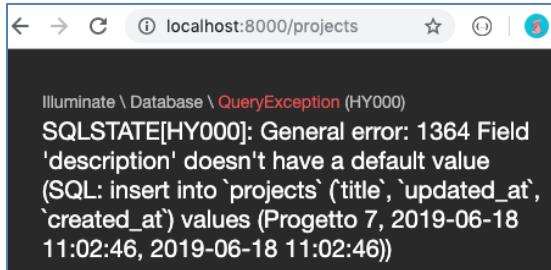
Ma... i **mass assignment** causano un errore, a meno che uno degli attributi-proprietà (qui *[title]*) venga designata come *fillable* nel modello *[App\Project]*

In realtà, si avrà un altro errore se *description* non è anch'essa *fillable* (il valore non verrà passato al DB e, in assenza di un valore di default, la *INSERT* SQL fallirà)

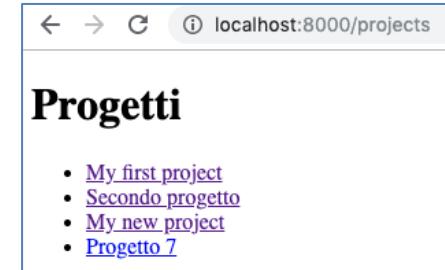


```
// file Project.php  
class Project extends Model {  
    protected $fillable = ['title'];  
}
```

# Modelli: proprietà *fillable* e *guarded*



```
// file Project.php
class Project extends Model
{
    protected $fillable = [
        'title', 'description'];
}
```



L'alternativa a *fillable* è dire al modello di non preoccuparsi del *mass assignment* alle proprietà, a meno che siano dichiarate *guarded*:

Tutto ciò serve a proteggere da richieste HTTP malevole, che cercano di assegnare "di nascosto" ad attributi del modello.

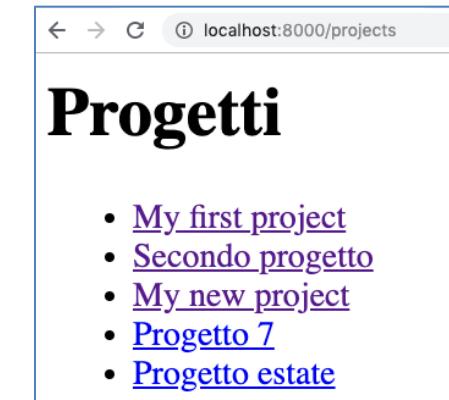
Per capire perché, si visualizzi la richiesta che attiva *store()*

- (NB: si può anche usare *return* al posto di *dd*)

```
// file Project.php
class Project extends Model
{
    protected $guarded = [ ];
}
```



```
public function store() {
    dd(request(['title', 'description']));
    Project::create([
        ...
    ]);
    return redirect('/projects');
}
```



```
array:2 ▾
  "title" => "Progetto autunno"
  "description" => "Tornare al lavoro : -("]
```

# Modelli: richieste con attributi indesiderati

Modifichiamo il callback *store* per vedere tutto (*->all()*) ciò che arriva con la richiesta.

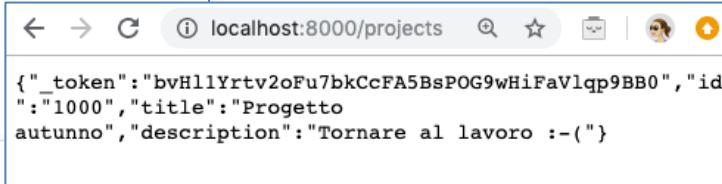
Come si ricorderà, il token era nascosto!

E se si "falsificasse" il form di richiesta sul browser (si usi "*Opzioni per sviluppatori*", "*EDIT as HTML*")?

```
public function store() {
    return request()->all();
}
```



```
<!-- projects/create.blade.php -->
<!DOCTYPE html>
<html>
  <head>...</head>
  <body>
    <h1>Crea un progetto</h1>
    <form method="POST" action="/projects">
      <input type="hidden" name="_token" value="bvH1lYrtv2oFu7bkCcFA5BsPOG9wHiFaVlqp9BB0">
      <form method="POST" action="/projects">
        <input type="hidden" name="id" value="1000">
        <div><input type="text" name="title" placeholder="Titolo progetto"></div><p>
```



Nella richiesta è stato iniettato un *id* nascosto!

Ripristiniamo lo *store()* che salva il record, dando a *Project::create()* direttamente *request* come argomento)

Siamo ancora protetti, però, perché gli attributi da salvare ('*title*', '*description*') sono specificati esplicitamente

```
public function store() {
    Project::create(request(['title',
                           'description']));
    return redirect('/projects');
}
```

# Modelli: richieste con attributi indesiderati /2

Ecco invece una versione di *store()* accattivante (perché breve), ma assai pericolosa se usata con *guarded* vuoto nel modello *Project!*

```
public function store() {  
    Project::create(request()->all());  
    return redirect('/projects');  
}
```

Essa salva infatti **tutti (*all()*)** gli attributi inviati con la richiesta

Se riproviamo a iniettare un *id* nascosto, lo ritroveremo nel database!

Tutti questi sono evidentemente trabocchetti da evitare!

# Accorgimenti per callback ancora più concisi

I successivi refactoring visti in precedenza introducono vari accorgimenti:

- Si utilizzi ovunque il model binding
- Nel callback *store()*, il metodo *Project::create()*, per istanziazione e "mass assignment"
- Nel callback *update()*, il metodo omonimo (di istanza di *Project*) *update()*, per un "mass update"

La funzione *request()* della classe ??? ha i vari argomenti, inserire slide ad hoc, cercando *request* nelle precedenti per decidere dove inserirla

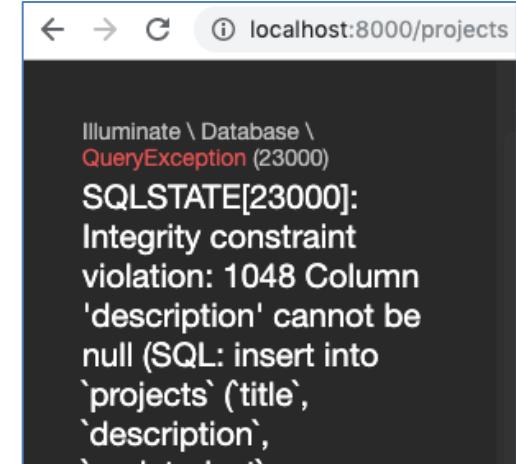
```
class ProjectsController extends Controller
{
    public function index()
    {
        $progetti = Project::all();
        return view('projects.index',
                    compact('progetti'));
    }
    public function create()
    {
        return view('projects.create');
    }
    public function store()
    {
        Project::create( request()->all() );
        return redirect('/projects');
    }
    public function edit(Project $project)
    {
```

```
        return view('projects.edit', compact('project'));
    }
    public function update(Project $project)
    {
        $project->update( request(['title','description']) );
        return redirect('/projects');
    }
    public function show(Project $project)
    {
        return view('projects.show', compact('project'));
    }
    public function destroy(Project $project)
    {
        $project->delete();
        return redirect('/projects');
    }
}
```

// ProjectsController.php

# Validazione input

- Creare progetto con descrizione vuota: errore!
- Rimedio client-side: *required* nei form, ma questo non protegge da manipolazioni, anche maliziose, dell'HTML (chrome developer)
- Rimedio server-side: l'helper *request()* ha una variante *validate()*:
  - specifica parametri *required* nella *request* HTTP
  - se richiesta non è valida, ridirige automaticamente alla pagina di provenienza, e, per, di più, le passa un parametro *\$errors*
  - nella view *create* la gestione di *\$errors* potrebbe essere meno spartana che qui a destra...



A screenshot of a web browser window showing an error message. The URL is 'localhost:8000/projects'. The error message is: 'Illuminate \ Database \ QueryException (23000) SQLSTATE[23000]: Integrity constraint violation: 1048 Column 'description' cannot be null (SQL: insert into `projects` ('title', 'description', 'created\_at', 'updated\_at') values ('Progetto test', NULL, now(), now()))'

```
public function store() {  
    request()->validate([  
        'title' => 'required',  
        'description' => 'required'  
    ]);  
    ...  
}
```

```
<!-- projects/create.blade.php -->  
<!DOCTYPE html>  
...  
<form method="POST" action="/projects">  
    {{ csrf_field() }}  
    ...  
    <div><button type="submit">  
        Crea progetto</button></div>  
</form>  
<div><p>{{ $errors }}</p>  
...  
}
```

# Validazione: ripartire dal testo errato

- Una comodità importante è ritrovare il testo inserito prima della (eventuale) validazione negativa
- Si usa la helper function `old()`
- NB per la *textarea* non è definito *value*

```
<!-- projects/create.blade.php -->

@extends('layout')
@section('content')
<h1>Crea un progetto</h1>

<form method="POST" action="/projects">
    {{ csrf_field() }}
    <div><input type="text" name="title"
        value="{{ old('title') }}"
        placeholder="Titolo progetto" required></div><p>
    <div>
        <textarea name="description"
            placeholder="Descrizione progetto"
            >{{ old('description') }}</textarea>
    </div><p>
    <div><button type="submit">Crea progetto</button></div>
</form>

@if ($errors->any)
<div>
    <ul>
        @foreach ($errors->all() as $err)
            <li>{{ $err }}</li>
        @endforeach
    </ul>
</div>
@endif

@endsection
```

# Validazione: altre regole

- Vi sono davvero molte possibilità
- Si veda <https://laravel.com/docs/validation#available-validation-rules>

The screenshot shows a browser window displaying the Laravel documentation at <https://laravel.com/docs/5.8/validation#available-validation-rules>. The page title is '# Available Validation Rules'. Below the title, a sub-section header '# Available Validation Rules' is shown. A note states: 'Below is a list of all available validation rules and their function:'. To the right of the list, there are two callout boxes with rounded corners.

Accepted	Distinct	Not Regex
Active URL	E-Mail	Nullable
After (Date)	Ends With	Numeric
After Or Equal (Date)	Exists (Database)	Present
Alpha	File	Regular Expression
Alpha Dash	Filled	Required
Alpha Numeric	Greater Than	Required If
Array	Greater Than Or Equal	Required Unless
Bail	Image (File)	Required With
Before (Date)	In	Required With All
Before Or Equal (Date)	In Array	Required Without
Between	Integer	Required Without All
Boolean	IP Address	Same
Confirmed	JSON	Size
Date	Less Than	Sometimes
Date Equals	Less Than Or Equal	Starts With
Date Format	Max	String
Different	MIME Types	Timezone
Digits	MIME Type By File Extension	Unique (Database)
Digits Between	Min	URL
Dimensions (Image Files)	Not In	UUID

**required\_if:anotherfield,value,...**

The field under validation must be present and not empty if the `anotherfield` field is equal to any value.

If you would like to construct a more complex condition for the `required_if` rule, you may use the `Rule::requiredIf` method. This methods accepts a boolean or a Closure. When passed a Closure, the Closure should return `true` or `false` to indicate if the field under validation is required:

**confirmed**

The field under validation must have a matching field of `foo_confirmation`. For example, if the field under validation is `password`, a matching `password_confirmation` field must be present in the input.

# Validazione: refactoring conciso del controller

- Nel callback `store()`, replicare i nomi degli attributi sia nell'argomento di `validate()` che di `Project::create()` è noioso (e causa di errori)
- In realtà, `validate()` restituisce ciò che serve, lo si vede con `dd()`
- Da cui i due possibili refactoring concisi, qui a destra
- NB: nel codice più in basso, se la validazione fallisce, `validate()` ridirige sulla pagina HTML di provenienza, la `store()` termina e l'esecuzione di `create()` non inizia nemmeno
- NB: altri constraint per i campi validati!

```
public function store() {  
    $validated = request()->validate([  
        'title' => ['required', 'min:3', 'max:255'],  
        'description' => ['required', 'min:8']  
    ]);  
    dd($validated);  
    Project::create(request(['title', 'description']));  
}
```

```
array:2 [▼  
    "title" => "Nuovo progetto"  
    "description" => "Descrizione lunga"  
]
```

```
public function store() {  
    $validated = request()->validate([  
        'title' => ['required', 'min:3', 'max:255'],  
        'description' => ['required', 'min:8']  
    ]);  
    Project::create($validated);  
}
```

```
public function store() {  
    Project::create(request()->validate([  
        'title' => ['required', 'min:3', 'max:255'],  
        'description' => ['required', 'min:8']  
    ]));  
}
```

# Relazioni tra modelli

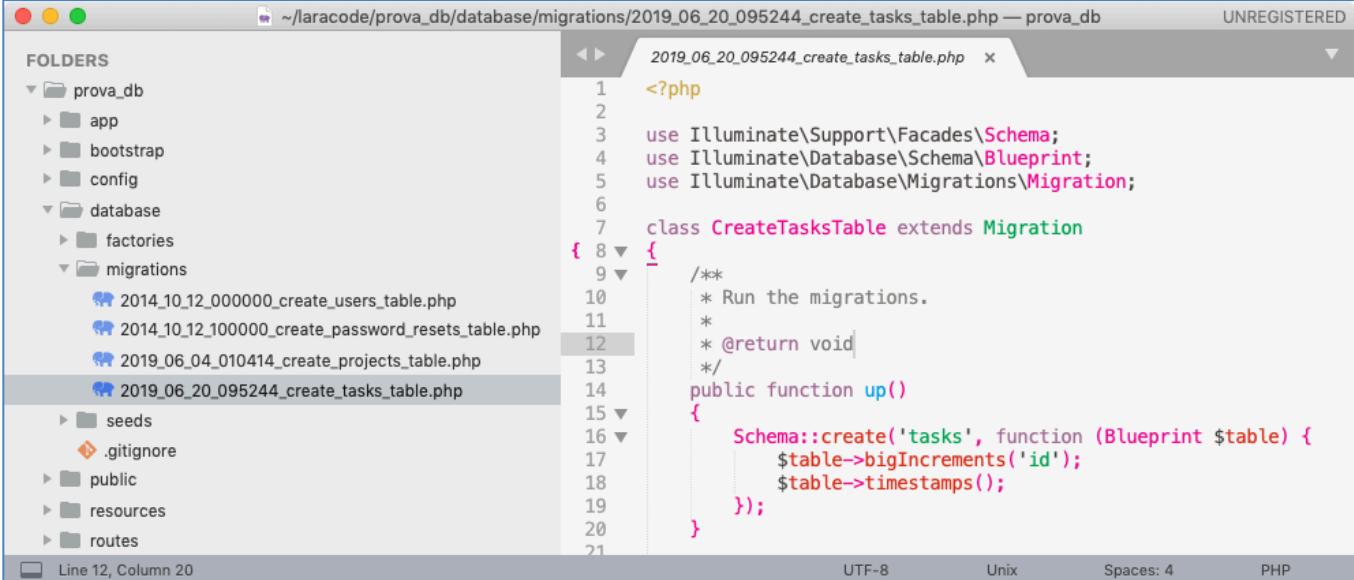
- A livello di modellazione *Entity-Relationship*, decidiamo che un *Project* è fatto di *Task*
- Usiamo *artisan* per generare stub del modello e delle tabelle (attraverso le migrazioni)
- Per il momento, nessun controller

```
prova_db $ php artisan help make:model
Description:
Create a new Eloquent model class
Usage:
make:model [options] [--] <name>
Arguments:
  name          The name of the class
Options:
  -a, --all      Generate a migration, factory, and resource controller for the model
  -c, --controller Create a new controller for the model
  -f, --factory   Create a new factory for the model
  -m, --migration Create a new migration file for the model
```

```
prova_db $ artisan make:model Task -m -f
Model created successfully.
Factory created successfully.
Created Migration: 2019_06_20_095244_create_tasks_table
```

# Design dell'entità Task e migrazione

- *artisan* ci ha dato uno "stub" dell'entità *Task*
- aggiungiamo attributi *description*, *completed* e, soprattutto *project\_id*



The screenshot shows a Mac OS X desktop with a code editor window open. The title bar says: `~/laracode/prova_db/database/migrations/2019_06_20_095244_create_tasks_table.php — prova_db`. The status bar at the bottom right shows: `UNREGISTERED`, `2019_06_20_095244_create_tasks_table.php`, `Line 12, Column 20`, `UTF-8`, `Unix`, `Spaces: 4`, and `PHP`.

The left sidebar shows the project structure:

- `prova_db`
- `app`
- `bootstrap`
- `config`
- `database`
  - `factories`
  - `migrations`
    - `2014_10_12_000000_create_users_table.php`
    - `2014_10_12_100000_create_password_resets_table.php`
    - `2019_06_04_010414_create_projects_table.php`
    - `2019_06_20_095244_create_tasks_table.php` (highlighted)
  - `seeds`
  - `.gitignore`
- `public`
- `resources`
- `routes`

The main editor area displays the `2019_06_20_095244_create_tasks_table.php` migration file:

```
<?php  
use Illuminate\Support\Facades\Schema;  
use Illuminate\Database\Schema\Blueprint;  
use Illuminate\Database\Migrations\Migration;  
  
class CreateTasksTable extends Migration  
{  
    /**  
     * Run the migrations.  
     *  
     * @return void  
     */  
    public function up()  
    {  
        Schema::create('tasks', function (Blueprint $table) {  
            $table->bigIncrements('id');  
            $table->timestamps();  
        });  
    }  
}
```

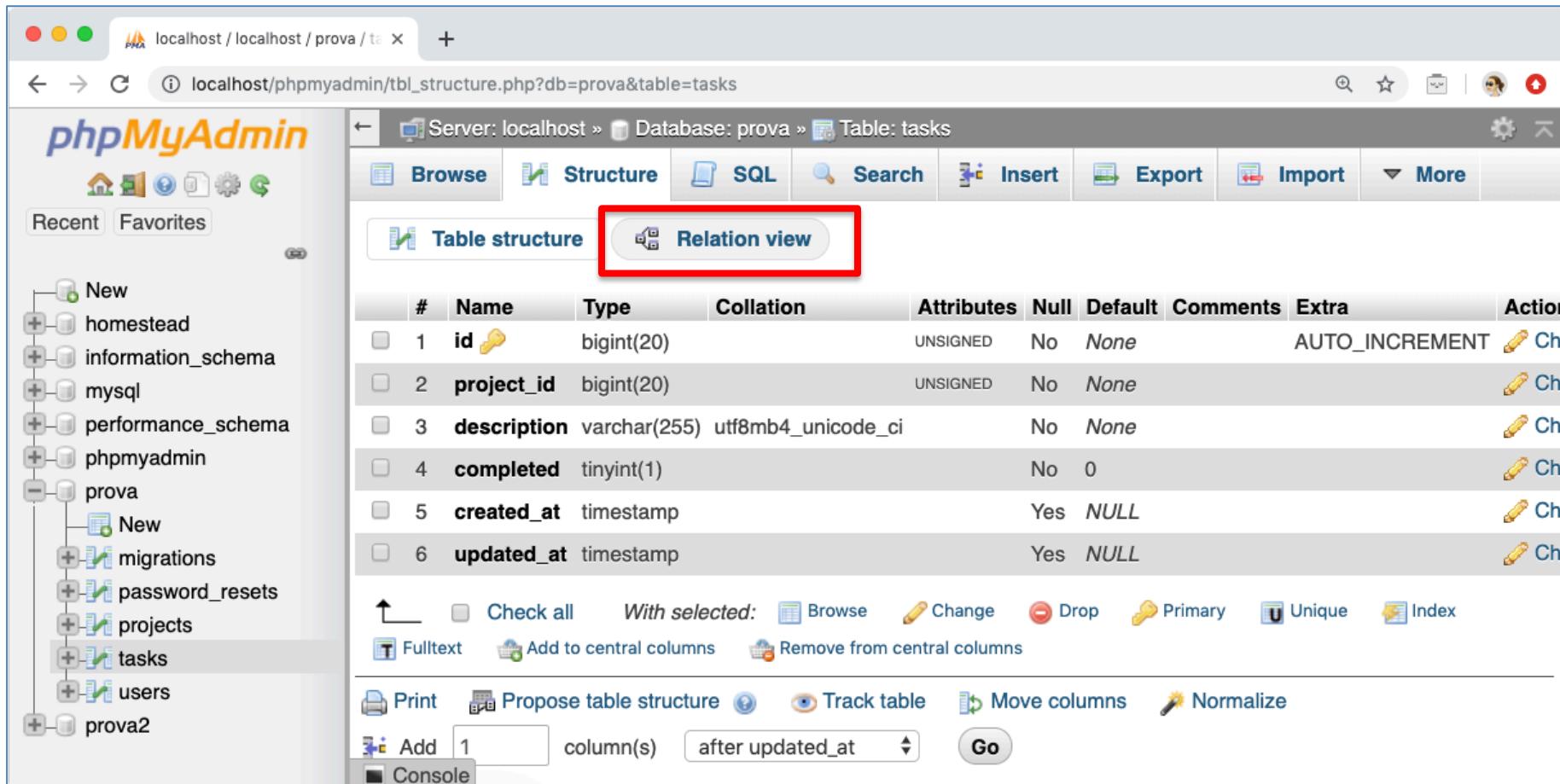
- l'intento è che *project\_id* stabilisca una relazione con *Project*
- ora si possono eseguire le migrazioni:

```
prova_db $ php artisan migrate  
Migrating: 2019_06_20_095244_create_tasks_table  
Migrated: 2019_06_20_095244_create_tasks_table
```

```
public function up()  
{  
    Schema::create('tasks', function (Blueprint $table) {  
        $table->bigIncrements('id');  
        $table->unsignedInteger('project_id');  
        $table->string('description');  
        $table->boolean('completed')->default(false);  
        $table->timestamps();  
    });  
}
```

- e verificare l'effetto sul DB con *phpmyadmin* (prossima slide)

# Nuova entità *Task* nel DB



The screenshot shows the phpMyAdmin interface for a database named 'prova'. The 'tasks' table is selected. The 'Relation view' tab is highlighted with a red box. The table structure is as follows:

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
1	<b>id</b>	bigint(20)		UNSIGNED	No	None		AUTO_INCREMENT	
2	<b>project_id</b>	bigint(20)		UNSIGNED	No	None			
3	<b>description</b>	varchar(255)	utf8mb4_unicode_ci		No	None			
4	<b>completed</b>	tinyint(1)			No	0			
5	<b>created_at</b>	timestamp			Yes	NULL			
6	<b>updated_at</b>	timestamp			Yes	NULL			

Below the table, there are buttons for 'Check all', 'With selected:', 'Browse', 'Change', 'Drop', 'Primary', 'Unique', 'Index', 'Fulltext', 'Add to central columns', and 'Remove from central columns'. At the bottom, there are buttons for 'Print', 'Propose table structure', 'Track table', 'Move columns', 'Normalize', 'Add 1 column(s)', 'Console', and 'Go'.

- Ma un clic su *Relation view* ci conferma che la relazione è ancora solo nelle bostre intenzioni!
- Eliminiamo i *projects* con *id* 1 e 2 (se li abbiamo ancora)...

# Relazioni tra entità: *hasMany()*

- Per definire in Laravel la relazione *1 Project* → *Molti Task*, si definisce, dentro la classe *Project* una funzione *tasks()*
- Così, dato un *Project \$proj*, si potrà riferire *\$proj->tasks* !

**NB:** come campo, non come funzione *tasks()*

```
prova_db $ php artisan tinker
>>> App\Project::first();
=> App\Project {#3197
    id: 3,
    title: "My new project",
    description: "Aggiustare il PC",
}
>>> App\Project::first()->tasks();      # NB: non è la funzione tasks() che serve, ma l'attributo tasks
=> Illuminate\Database\Eloquent\Relations\HasMany {#3187}
>>> App\Project::first()->tasks;
=> Illuminate\Database\Eloquent\Collection {#3191
    all: [],
}
```

```
// file App/Project.php
class Project extends Model
{
    protected $guarded = [ ];
    public function tasks()
    {
        return $this->hasMany(Task::class);
    }
}
```

- Ora inseriamo (a mano, in phpmyadmin) due task con *project\_id* pari a 3

# Due nuovi task per un Progetto (3)

- *Insert* poi clic su *Go*

The screenshot shows the phpMyAdmin interface for the 'prova' database. The 'tasks' table is selected. The 'Insert' tab is active. A new row is being inserted with the following values:

Column	Type	Value
id	bigint(20) unsigned	1
project_id	bigint(20) unsigned	3
description	varchar(255)	Comprare RAM e disco
completed	tinyint(1)	0
created_at	timestamp	NOW
updated_at	timestamp	NOW

The 'Go' button is visible at the bottom right.

- Qui a destra l'effetto, dopo avere inserito un secondo task, cioè ("aprire il case del PC")

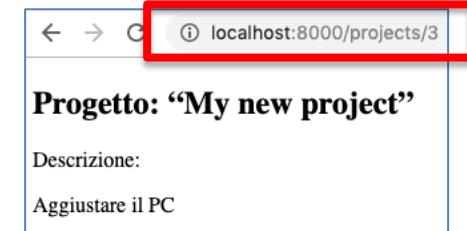
The screenshot shows the phpMyAdmin interface for the 'prova' database. The 'tasks' table is selected. The 'Browse' tab is active. The table now contains two rows:

	id	project_id	description	completed	created_at	updated_at
<input type="checkbox"/>	1	3	Comprare RAM e disco	0	2019-06-20 13:23:09	2019-06-20 13:23:09
<input type="checkbox"/>	2	3	aprire il case del PC	0	2019-06-20 13:24:36	2019-06-20 13:24:36

# Il Project (3) ha ora la collezione dei suoi tasks!

```
>>> App\Project::first()->tasks;
=> Illuminate\Database\Eloquent\Collection {#3190
    all: [
        App\Task {#3204
            id: 1,
            project_id: 3,
            description: "Comprare RAM e disco",
            completed: 0,
            created_at: "2019-06-20 13:23:09",
            updated_at: "2019-06-20 13:23:09",
        },
        App\Task {#3205
            id: 2,
            project_id: 3,
            description: "aprire il case del PC",
            completed: 0,
            created_at: "2019-06-20 13:24:36",
            updated_at: "2019-06-20 13:24:36",
        },
    ],
}
>>>
```

- Ovviamente non sono visibili nella attuale view *show*
- La si modifichi: come si vede è nella view che va il codice di presentazione
  - mentre nel controller (in cui non cambia nulla) ci si limita a invocare la view, passandole il progetto corrente (qui **3**)



# La view *project/3/show* con i task

```
@extends('layout')

@section('content')

<h2>Progetto: &ldquo;{{$project->title}}&rdquo;</h2>

<div>Descrizione:<p>
    {{$project->description}}</p></div>

<div>
    @foreach($project->tasks as $task)
        <li>{{ $task->description }}</li>
    @endforeach
</div>
<p>
<a href="/projects/{{ $project->id }}/edit">
    Modifica questo progetto
</a>
@endsection

{{-- file projects/show.blade.php --}}
```

- O, meglio, si possono elencare i task solo se ci sono effettivamente per quel dato progetto:

```
@if ($project->tasks->count())
<div>
    @foreach($project->tasks as $task)
        <li>{{ $task->description }}</li>
    @endforeach
</div>
@endif
```

# Relazioni tra entità: *belongsTo*

Per i *Task* occorre una possibilità duale:

- una funzione che consenta di simulare un attributo *Project*
- che assuma per valore l'unico *Project* cui il *Task* appartiene ("belongs to")

Notare che dietro le quinte vengono eseguite delle query SQL

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;

class Task extends Model
{
    public function project() {
        return $this->belongsTo(Project::class);
    }
}
```

```
>>> App\Task::Find(2);
=> App\Task {#3203
    id: 2,
    project_id: 3,
    description: "aprire il case del PC",
    completed: 0,
    created_at: "2019-06-20 13:24:36",
    updated_at: "2019-06-20 13:24:36",
}
>>> App\Task::Find(2)->project;
=> App\Project {#3191
    id: 3,
    title: "My new project",
    description: "Aggiustare il PC",
    created_at: "2019-06-16 22:29:24",
    updated_at: "2019-06-16 22:29:24",
}
>>>
```

# Le query dietro le quinte

```
>>> DB::listen(function ($query) { dump($query->sql); dump($query->bindings); dump($query->time); });
```

O, più semplicemente:

```
>>> DB::listen(function ($query) { dump($query->sql); dump($query->bindings); }));
```

```
>>> App\Project::all();
"select * from `projects`"
[]
...
>>> $proj = App\Project::first();
"select * from `projects` limit 1"
[]
...
>>> $proj->tasks;
"select * from `tasks` where `tasks`.`project_id` = ? and
`tasks`.`project_id` is not null"
array:1 [
  0 => 3
]
...
...
```