

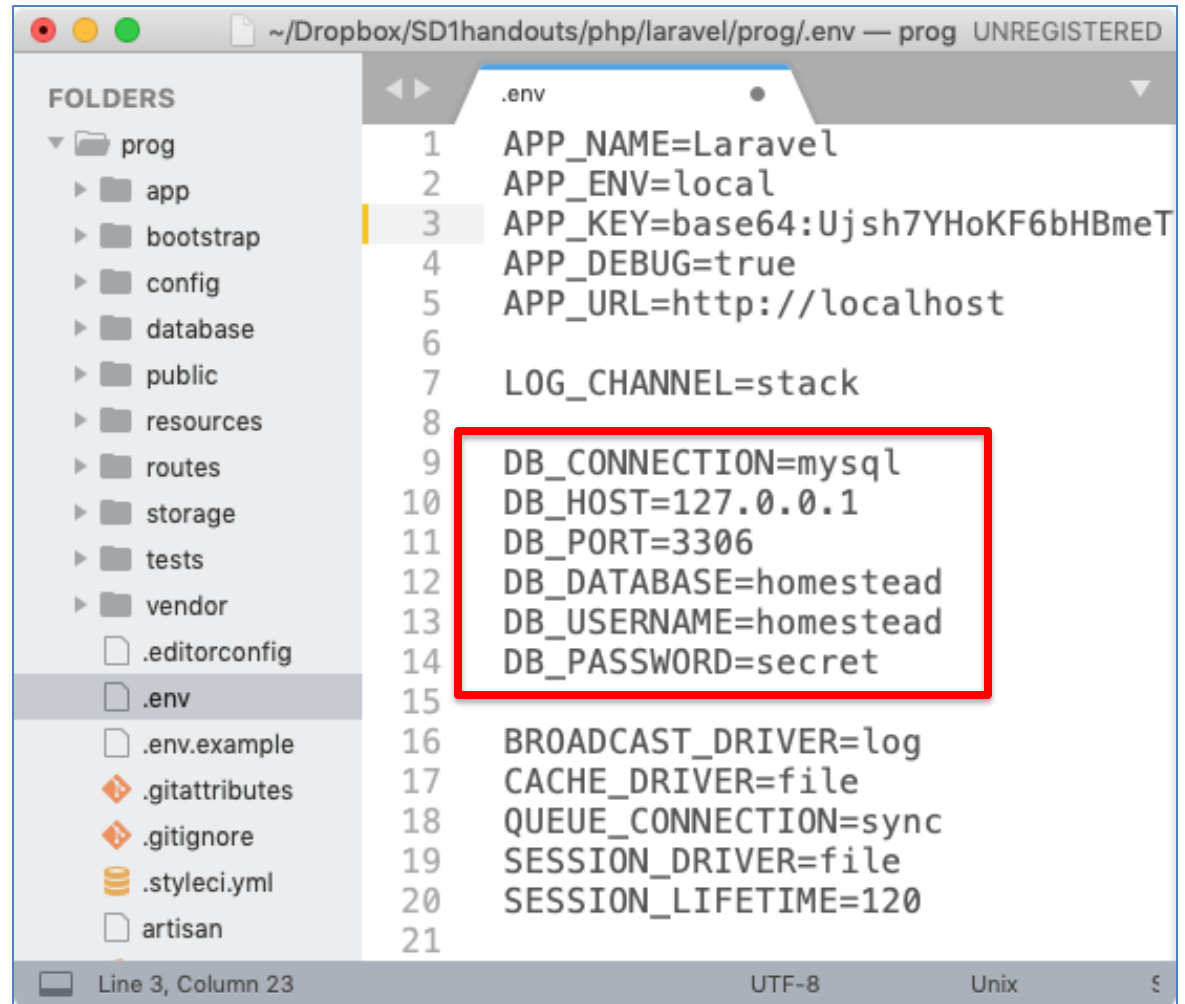
Modelli

- 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
```

```
...
```

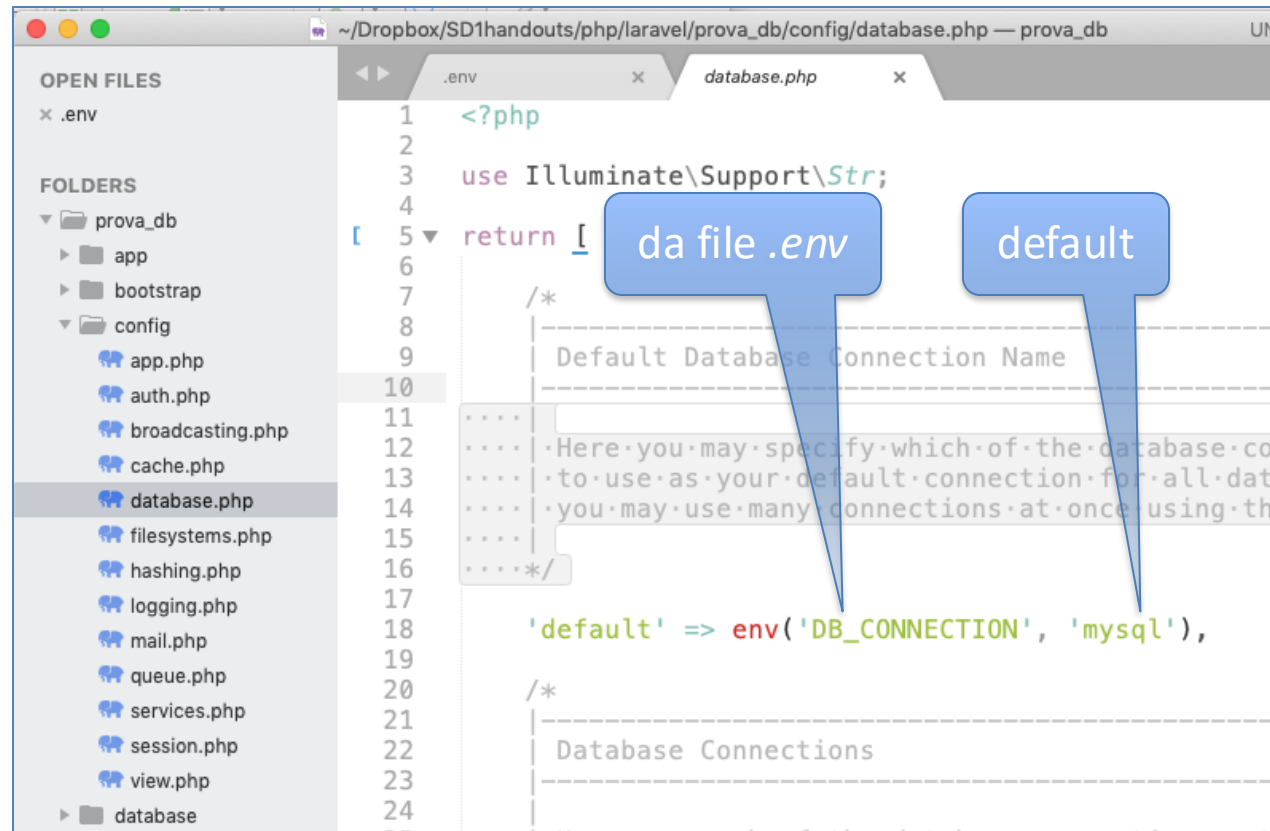
```
Application ready! Build something amazing.
```



```
.env
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
     *
     * Here you may specify which of the database connections
     * to use as your default connection for all database
     * operations. You may use many connections at once using the
     * database name, e.g. 'database'.
     */

    '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',
]
```

DB mysql di prova



- 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

```
.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

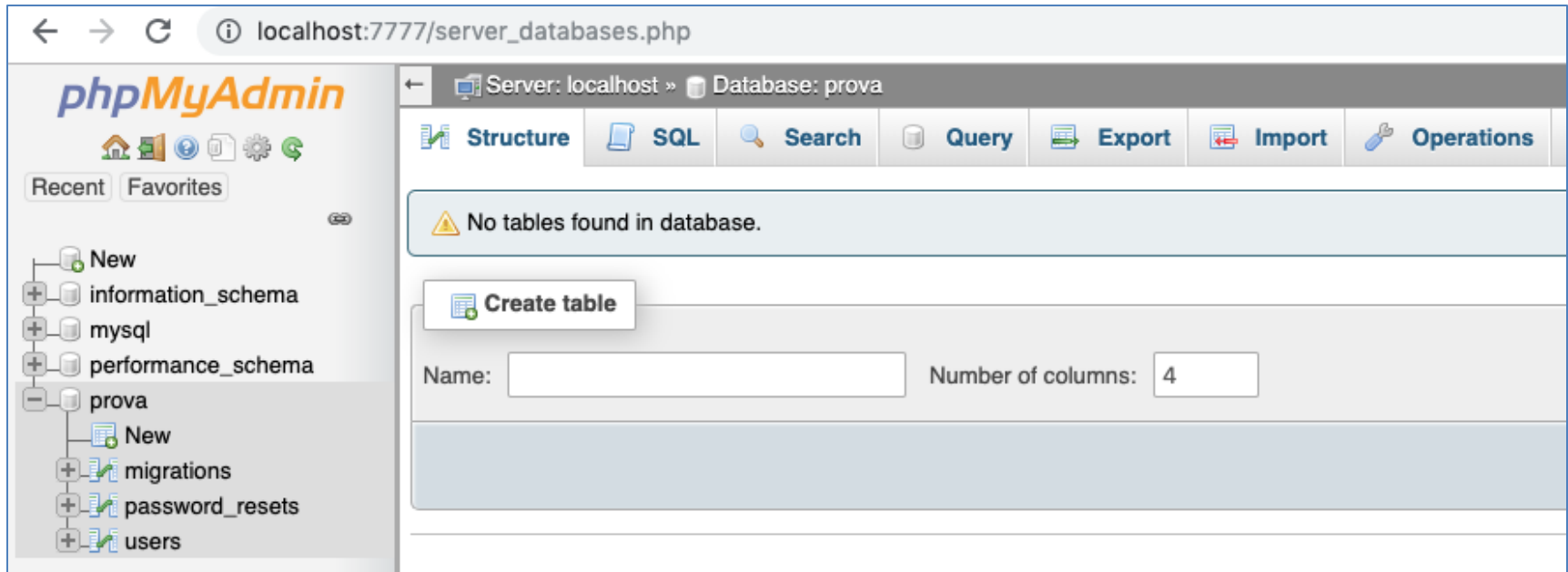
```
prova_db $ php artisan migrate
```

```
Nothing to migrate.
```

- Usiamo *phmyadmin* 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* lo si può osservare con *phpmyadmin*:



- 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 'prova' is selected, and the 'migrations' table is highlighted. The main panel shows the 'Table: migrations' view. A green message box states: 'MySQL returned an empty result set (i.e. zero rows). (Query took 0.0001 seconds.)'. Below this, the SQL query 'SELECT * FROM `migrations`' is displayed. At the bottom, the table structure is shown with columns 'id', 'migration', and 'batch'.

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

Redo di una migrazione

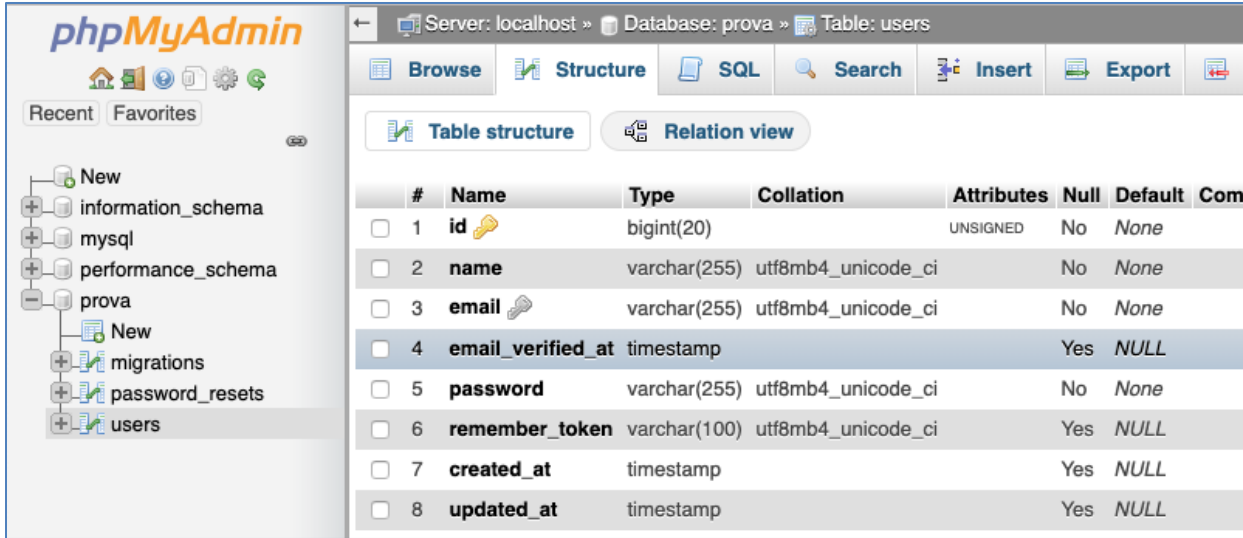
```
prova_db $ php artisan migrate # di fatto un redo
Migrating: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_000000_create_users_table
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated: 2014_10_12_100000_create_password_resets_table
prova2_db $ php artisan migrate:status
```

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. On the left is a sidebar with a tree view of databases and tables. The 'prova' database is selected, and the 'users' table is highlighted. The main panel shows the 'Table structure' view for the 'users' table. It displays a list of columns with their respective data types, collations, attributes, nullability, and default values.

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

```
1 <?php
2
3 use Illuminate\Support\Facades\Schema;
4 use Illuminate\Database\Schema\Blueprint;
5 use Illuminate\Database\Migrations\Migration;
6
7 class CreateUsersTable extends Migration
8 {
9     /**
10      * Run the migrations.
11      *
12      * @return void
13      */
14     public function up()
15     {
16         Schema::create('users',
17             function (Blueprint $table) {
18                 $table->bigIncrements('id');
19                 $table->string('name');
20                 $table->string('email')->unique();
21                 $table->timestamp('email_verified_at')->nullable();
22                 $table->string('password');
23                 $table->rememberToken();
24                 $table->timestamps();
25             });
26     }
27 }
```

Server: localhost » Database: prova

Browse

Structure

SQL

Table structure

Relation view


#	Name	Type	Co
1	id	bigint(20)	
2	name	varchar(255) utf	
3	email	varchar(255) utf	
4	email_verified_at	timestamp	
5	password	varchar(255) utf	
6	remember_token	varchar(100) utf	
7	created_at	timestamp	
8	updated_at	timestamp	

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	Co
1	id	bigint(20)	
2	name	varchar(255) utf	
3	email	varchar(255) utf	
4	email_verified_at	timestamp	
5	password	varchar(255) utf	
6	remember_token	varchar(100) utf	
7	created_at	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>

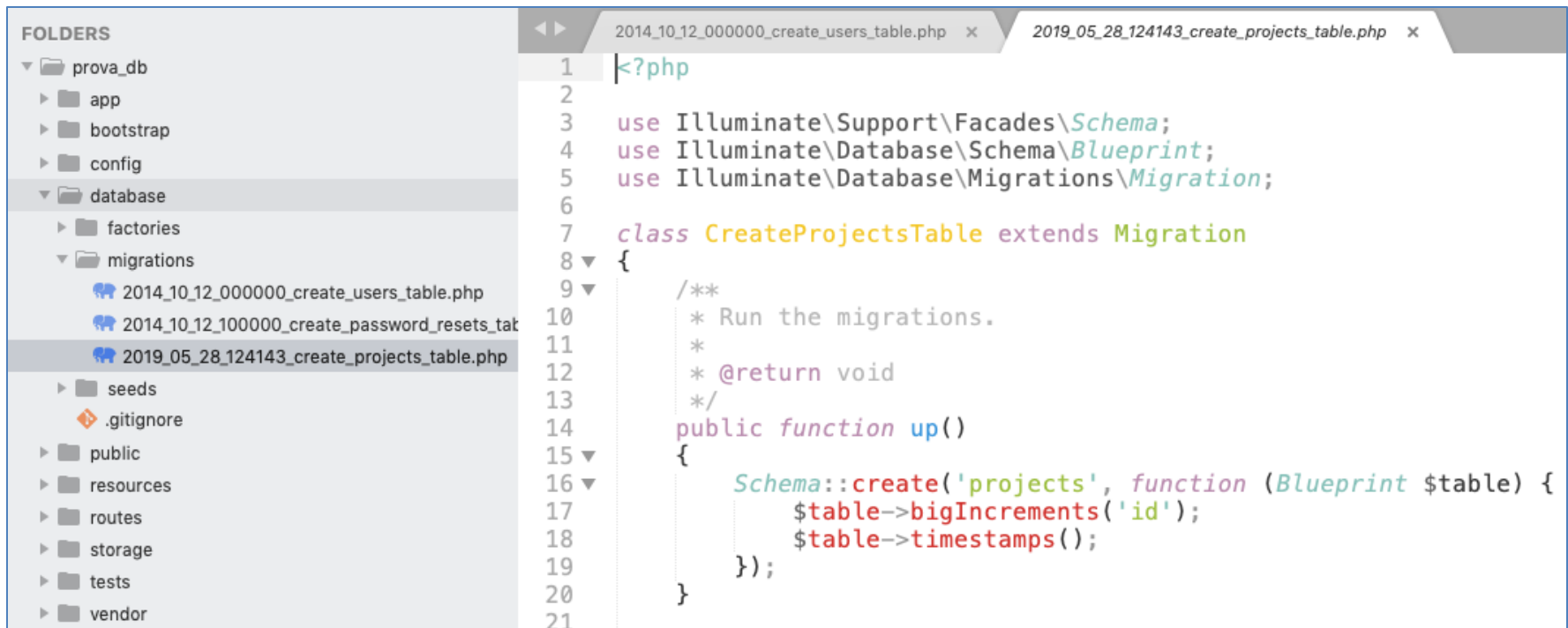
prova_db $ php artisan make:migration create_projects_table
Created Migration: 2019_05_28_124143_create_projects_table
```

- 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 ...))*



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

# ***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 tabelle 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

# 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",
}
```

**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, 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');
});
```

- La funzione controller si chiamerà *index*

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

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

# Il ProjectsController / 1 (statico)

- Partiamo con una view statica, nella subdirectory  
*.../resources/views/projects/index.php*  
— ciò perché si presume altri modelli potranno avere una view chiamata *index*

```
<!-- 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)

- Potremmo rendere la risposta dinamica facendo restituire la 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, *\App!*

```
<!-- 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');
 }
}
```

Symfony \ Component \ Debug \  
Exception \ **FatalThrowableError**  
(E\_ERROR)  
**Class**  
'App\Http\Controllers\Project'  
' not found

```
<!-- 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');
 }
}
```

localhost:8001/projects/

Raw Parsed

```
<!-- 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"}]
```

# View 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), si torna alla view rendendola dinamica grazie a un parametro *\$progetti* passatole dal controller

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

```
<?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]
);
 }
}
```

# 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 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 $prog)
 {{$prog->title}}
 @endforeach
</body>
</html>
```

# Creazione di un oggetto/record *Project*

- Si introduce una route */projects/create*, mappata su una funzione controller *create()*
- (Si introduce un link sulla URL base)
- Si introduce la view *projects.create* che corrisponde al file *projects/create.blade.php* (cioè *create.blade.php* nella directory *projects*)
  - l'idea è di raggruppare le view relative al modello *Project* nella directory *.../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"></div><p>
 <div><button type="submit">Crea progetto</button></div>
 </form>
</body>
</html>
```

## Crea un progetto

Progetto 1

Bellissimo!

Crea progetto

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, 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');
 }
 x
}
// ProjectsController.php
```

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



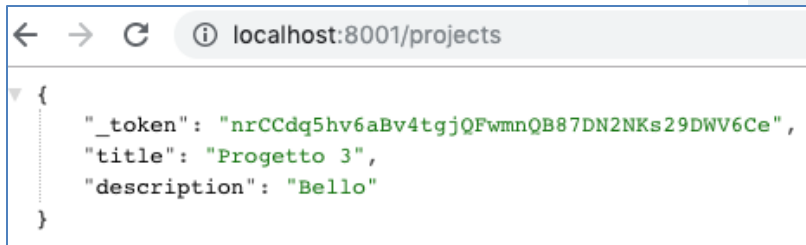
# Il metodo *store()* del controller

- La richiesta pervenuta dal cliente con POST è considerata "falsa" se non contiene un token unico, che 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!

```
<?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>
...

```



```
{
 "_token": "nrCCdq5hv6aBv4tgjQFwmnQB87DN2NKs29DWV6Ce",
 "title": "Progetto 3",
 "description": "Bello"
}
```

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

- Quindi *request()* (con o senza *->all()*) restituisce effettivamente la richiesta (in formato JSON)

```
localhost:8001/projects
{
 "_token": "nrCCdq5hv6aBv4tgjQFwmnQB87DN2NKs29DWV6Ce",
 "title": "Progetto 3",
 "description": "Bello"
}
```

- 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

```
view-source:localhost:8001/projects/create
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <title></title>
5 </head>
6 <body>
7 <h1>Crea un progetto</h1>
8 <form method="POST" action="/projects">
9 <input type="hidden" name="_token" value="nrCCdq5hv6aBv4tgjQFwmnQB87DN2NKs29DWV6Ce">
10 <div><input type="text" name="title"
11 placeholder="Titolo progetto"></div><p>
12 <div>
13 <textarea name="description"
14 placeholder="Descrizione progetto"></textarea>
15 </div><p>
16 <div><button type="submit">Crea progetto</button></div>
17 </form>
18 </body>
19 </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, con il metodo *helper redirect()* mostriamo il nuovo elenco progetti

```
<?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*



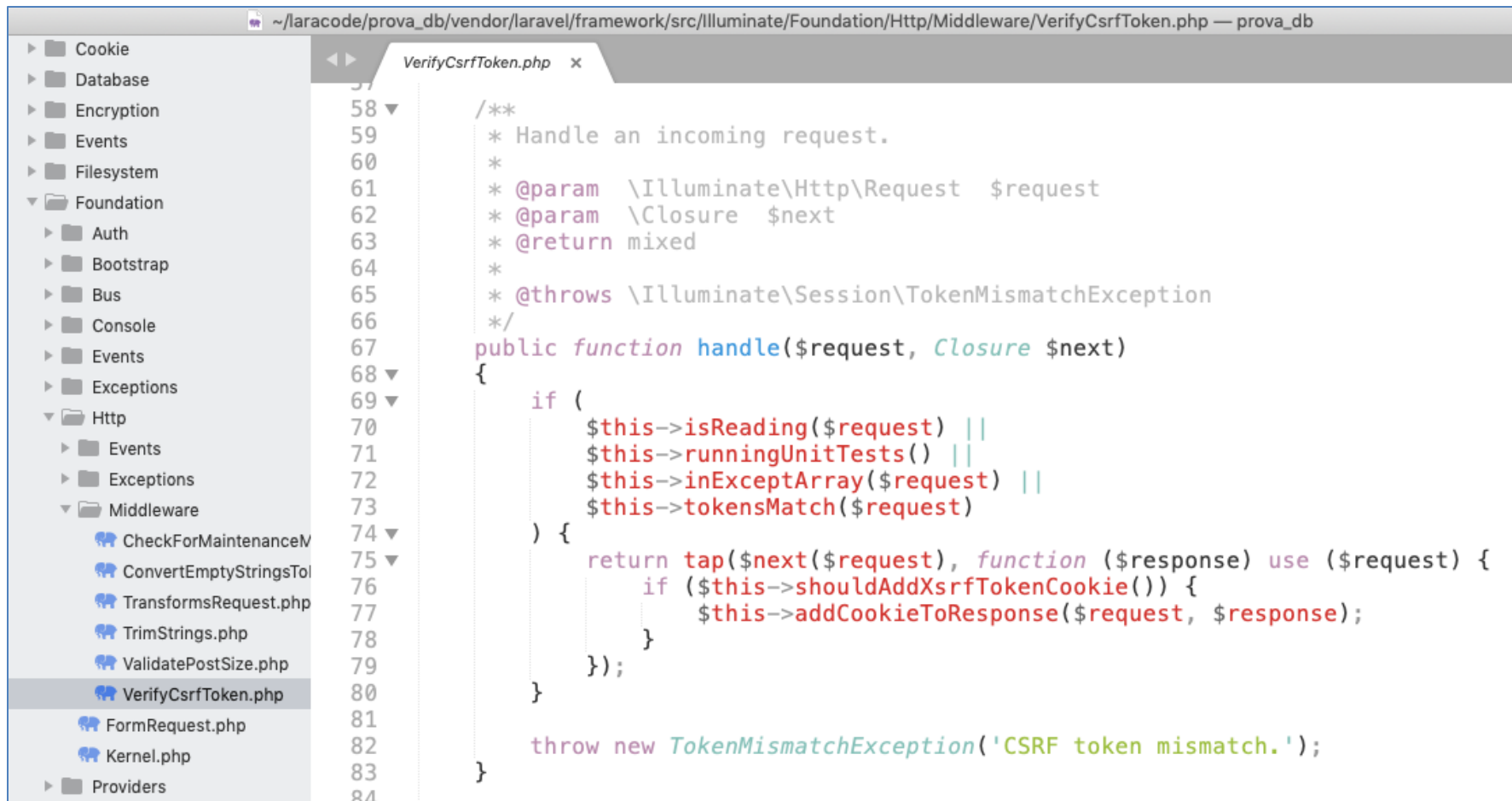
```
28 //
29 protected $middlewareGroups = [
30 'web' => [
31 \App\Http\Middleware\EncryptCookies::class,
32 \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
33 \Illuminate\Session\Middleware\StartSession::class,
34 // \Illuminate\Session\Middleware\AuthenticateSession::class,
35 \Illuminate\View\Middleware\ShareErrorsFromSession::class,
36 \App\Http\Middleware\VerifyCsrfToken::class,
37 \Illuminate\Routing\Middleware\SubstituteBindings::class,
38],
39
40 'api' => [
41 'throttle:60,1',
42 'bindings',
43],
44];
45
46 /**
47 * The application's route middleware.
48 *
49 * These middleware may be assigned to groups or used individually.
50 *
51 * @var array
52 */
53 protected $routeMiddleware = [
```



```
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



```
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 throw new TokenMismatchException('CSRF token mismatch.');
```

# Modelli e risorse

- Modello per la Web app = Risorsa per l'utente
- Nell'esempio, *Project* è una risorsa
  - è il primo componente di un gruppo di route
- Progettiamo l'interazione con la risorsa a partire dalle rotte
- Inseriamo uno schema di rotte come commento nel router *web.php*
  - altri componenti della rotta specificano l'operando (se c'è) e l'operazione (se non è già definita dal messaggio)
  - in parentesi l'operazione "logica"
  - i *GET* richiedono dati o form
  - gli altri messaggi cambiano lo stato
  - *PUT* (qui non c'è) e *PATCH* sono quasi equivalenti

```
<?php

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

/*
GET /projects (index)

GET /projects/create (create)

GET /projects/1/edit (edit)

POST /projects (store)

PATCH /projects/1 (update)

DELETE /project/1 (destroy)
*/
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 il file delle rotte per introdurre "a mano" le rotte e i corrispondenti callback
- I callback saranno altrettanti metodi del *ProjectsController*
- NB: anche se qui non lo prevediamo, si sappia che un messaggio *HEAD* avrà la risposta con header come per *GET* e *body* vuoto

```
<?php
Route::get('/', function () { return view('welcome'); });
/*
GET /projects (index)
GET /projects/create (create form)
GET /projects/1 (show #)

POST /projects (store form)
GET /projects/1/edit (edit form #)

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');

// web.php
```

# artisan:route

```
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()*

# 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

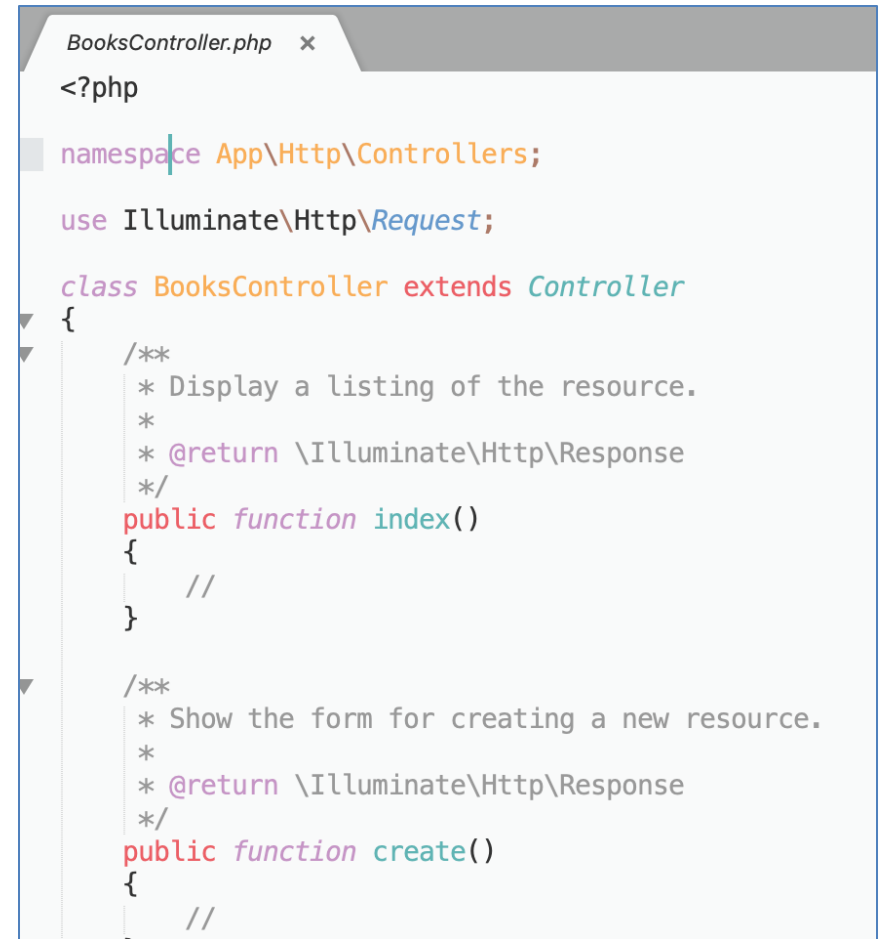
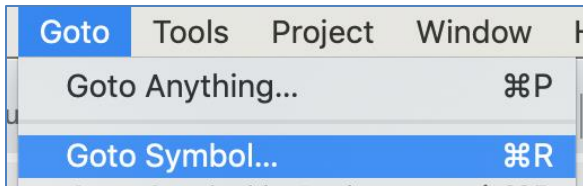
## Options:

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

...

# Boilerplate nel controller: esempio (risorsa *Book*)

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



# 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

# Wizard per controller e 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 conviene 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 *Post.php* (e use App\Post nel *PostsController*)
- nel controller, il boilerplate per il callback di *SHOW* riceve un argomento *active record* *Post \$post* (cf. qui a destra)
  - ciò consente interazioni "automatiche" 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 invoca il callback *update()* che modifica il record 1
- Similmente per HTTP *DELETE*, URL *HOST/projects/1*, callback *delete()*

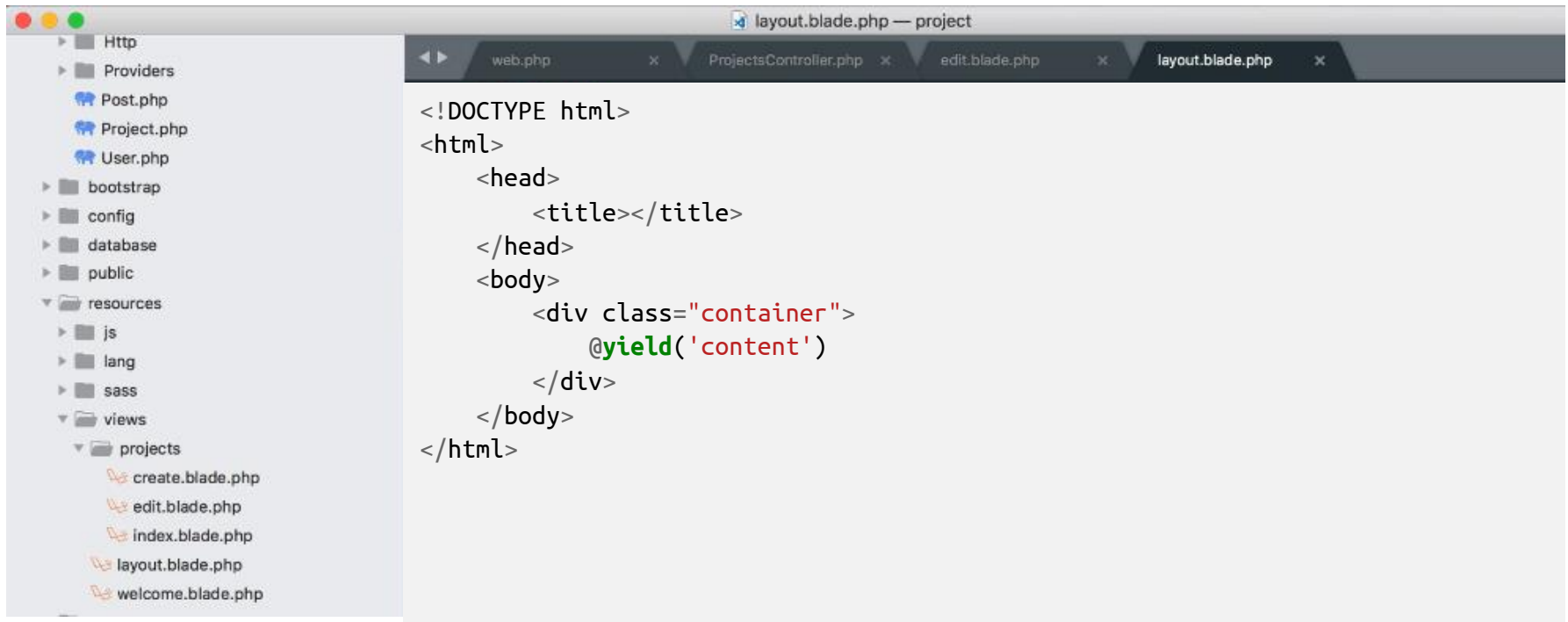
```
...
public function edit() // risponde a: HOST/projects/1/edit
{
 return view('projects.edit');
}

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*:



- 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*

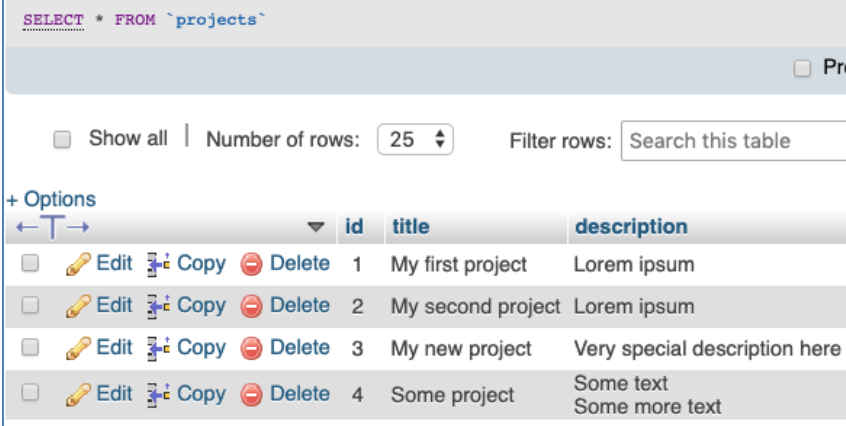
```
public function edit() // risponde a: HOST/project/1/edit
{ // e dovrebbe mostrare record 1
 $project = Project::find(1); // 1 va parametrizzato
 return view('projects.edit'); // solo un abbozzo: la view
} // dovrà dipendere da $project

// 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 verifica via *phpmyadmin*



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

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



# Callback *edit()* con parametro

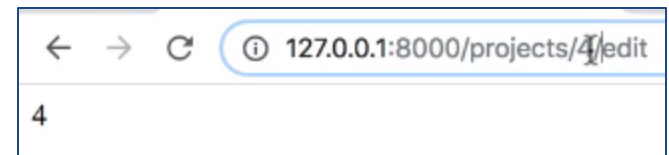
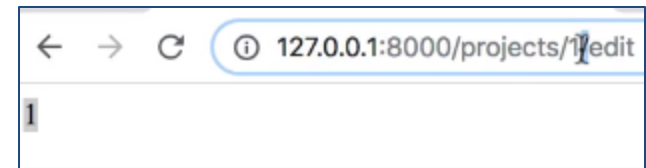
Torniamo al controller, callback *edit()*, bypassiamo la view e il *find()*, aggiungiamo un parametro *\$id* e proviamo a restituirlo...

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* 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');
}
```



```
→ ↻ ⓘ 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 blade**)

Nella view *edit* introduciamo 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
- cioè i valori correnti da modificare

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

```
@extends('layout')

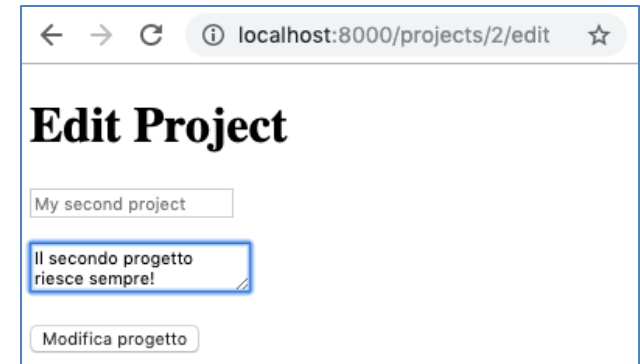
@section('content')
 <h1>Edit Project</h1>
 <form method="POST" action="/projects">
 {{ csrf_field() }}
 <div><input type="text" name="title"
 value="{{ $project->title }}">
 </div><p>
 <div>
 <textarea name="description"
 >{{ $project->description }}</textarea>
 </div><p>
 <div><button type="submit">
 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 modifichi il campo *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 web browser at `localhost:8000/projects/2/edit`. The page title is "Edit Project". It contains a form with the following fields:

- Text input: "My second project"
- Text area: "Il secondo progetto riesce sempre!"
- Button: "Modifica progetto"

The browser's developer tools are open, showing the HTML structure. The form has a `PATCH` method and an `action="/projects/2"`. The URL in the address bar is `localhost:8000/projects/2?title=My+second+project&description=Il+secondo+progetto%0D%0Ariesce+sempre%21`. The console shows a `BadMethodCallException` error: "Method App\Http\Controllers\ProjectsController::show does not exist."

Due problemi

- *PATCH* ignorato
- callback **show()** assente nel controller

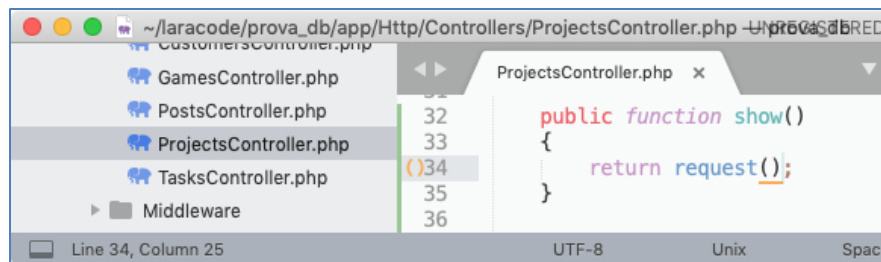
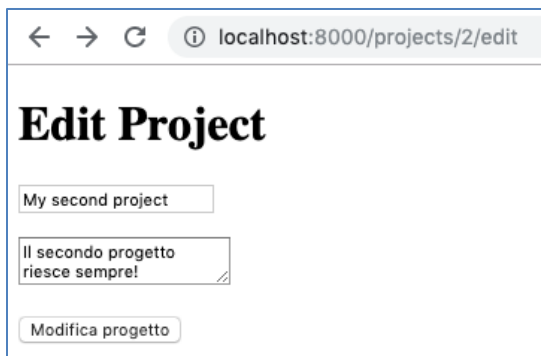
Ricordiamo ancora una volta le route, e precisamente:

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

| Domain | Method   | URI                     | Name | Action                                         | Middleware |
|--------|----------|-------------------------|------|------------------------------------------------|------------|
|        | 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        |

Il browser ignora *PATCH* e, per default, invia *GET*; la app attiva *show()*...

# Una rudimentale view per *show()*

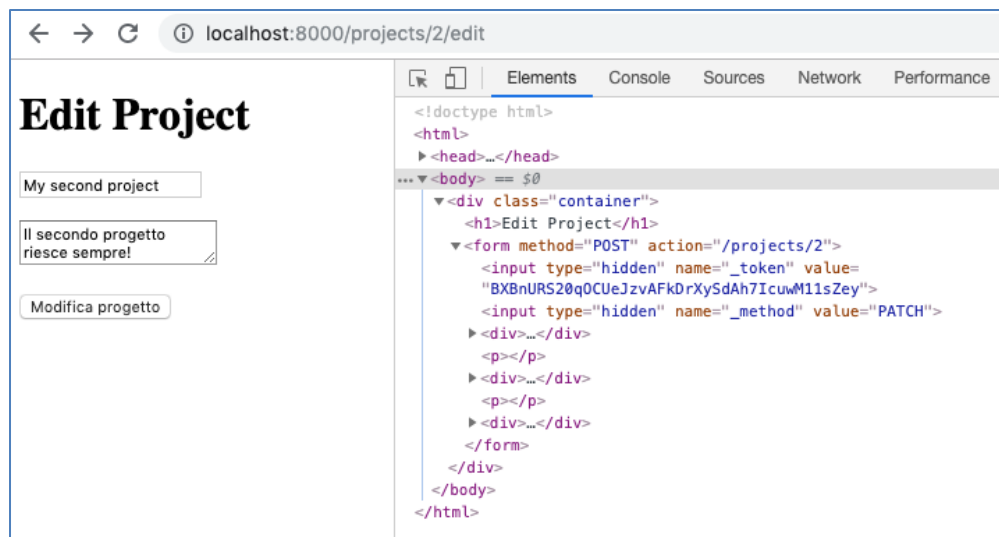


Ecco l'effetto, prodotto dal nuovo callback *show()*:



Ma occorre inviare una *PATCH*:

- si invia una comune *POST* con campo *hidden* e parametro di nome `"_method"` e valore `"PATCH"`
- all'uopo, si può anche usare un metodo PHP
- l'app Laravel sa di dover reagire come per *PATCH*, cioè con *update()*





# Il callback *update()*

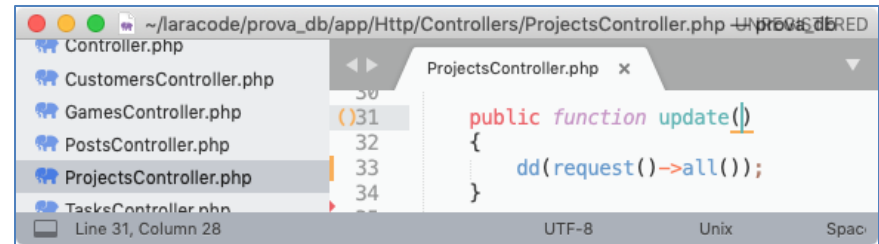
Iniziamo con un semplice *dump-and-die*, cioè *dd()*, della *request()* che arriva al *ProjectsController*

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

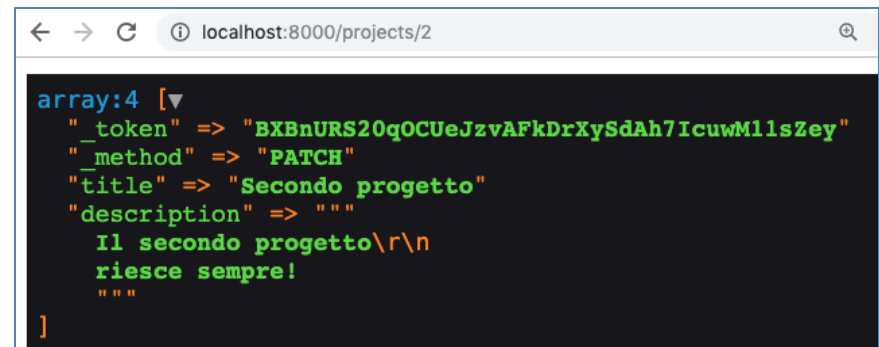
Ora un *update()* che faccia il lavoro, attraverso il pattern *active record*:

```
// 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*:



```
public function update()
{
 dd(request()->all());
}
```



```
array:4 [▼
 "_token" => "BxBnURS20qOCUeJzvAFkDrXySdAh7IcuwM1lsZey"
 "_method" => "PATCH"
 "title" => "Secondo progetto"
 "description" => ""
 Il secondo progetto\r\n riesce sempre!
 ""
]
```



## Progetti

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

|                          |                  | id | title            | description                        |
|--------------------------|------------------|----|------------------|------------------------------------|
| <input type="checkbox"/> | Edit Copy Delete | 1  | My first project | Lorem ipsum                        |
| <input type="checkbox"/> | Edit Copy Delete | 2  | Secondo progetto | Il secondo progetto riesce sempre! |

# Cancellazione: view e HTML

Ricordando ancora una volta le route:

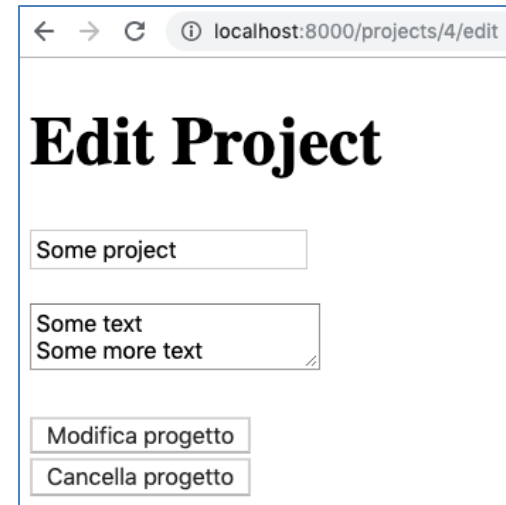
```
prova_db $ php artisan route:list
```

| 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        |

Si può gestire anche dalla view col form attivato da *edit*, introducendo un bottone che invii *POST* con un *DELETE* nascosto;

rispetto al codice 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');
}
```



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

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

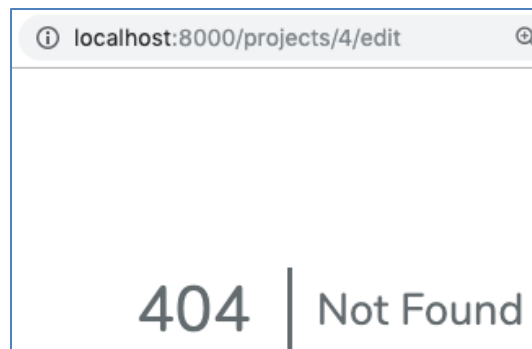
# Accesso a record inesistente: *findOrFail()*

- 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        |

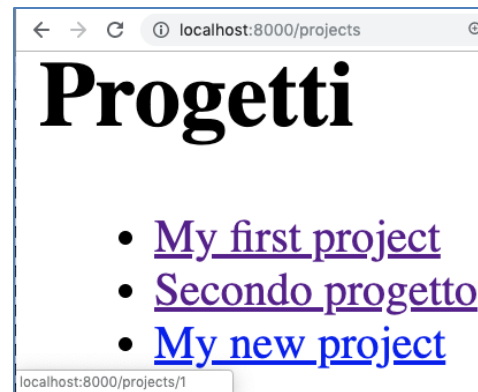
Per comodità, rendiamo l'elenco di progetti restituito da */projects – index()* cliccabile come link verso */projects/id – show(id)* (con il codice evidenziato):

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

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

 @foreach ($prog as $progetti)
 id }}">
 {{ $prog->id }}
 @endforeach

</body>
</html>
```



# Callback *show()* e view

Il codice di *show(\$id)* presuppone una view omologa alla quale si passa l'active record di chiave *\$id*

La view (a dx) visualizza il record e ha un link per passare alla view *edit*



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

```
@extends('layout')

@section('content')

<h2>Progetto: “{{$project->title}}”</h2>
<div>Descrizione:<p>
 {{$project->description}}</p></div>
id}}/edit">
 Modifica questo progetto
@endsection
{{-- file projects/show.blade.php --}}
```

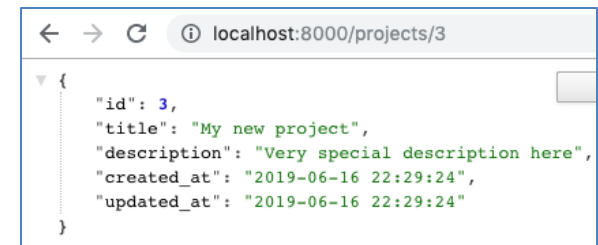
Un esperimento interessante:

- callback *show(\$id)* che, anziché passare alla view, restituisce l'active record
- *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;
}
```



# 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 ha come tipo la classe del 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($id)
{
 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 *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)



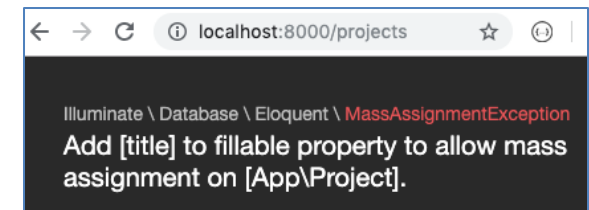
localhost:8000/projects/create

## Crea un progetto

Progetto 7

Andare in vacanza

Crea progetto



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



# Modelli: proprietà *fillable* e *guarded*

```
localhost:8000/projects

Illuminate \ Database \ QueryException (HY000)
SQLSTATE[HY000]: General error: 1364 Field
'description' doesn't have a default value
(SQL: insert into `projects` (`title`, `updated_at`,
`created_at`) values (Progetto 7, 2019-06-18
11:02:46, 2019-06-18 11:02:46))
```

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

```
localhost:8000/projects

Progetti

• My first project
• Secondo progetto
• My new project
• Progetto 7
```

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

```
// file Project.php
class Project extends Model
{
 protected $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*)

```
localhost:8000/projects/create

Crea un progetto

Progetto estate

Andare in vacanza ora!

Crea progetto
```

```
localhost:8000/projects

Progetti

• My first project
• Secondo progetto
• My new project
• Progetto 7
• Progetto estate
```

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

```
localhost:8000/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.

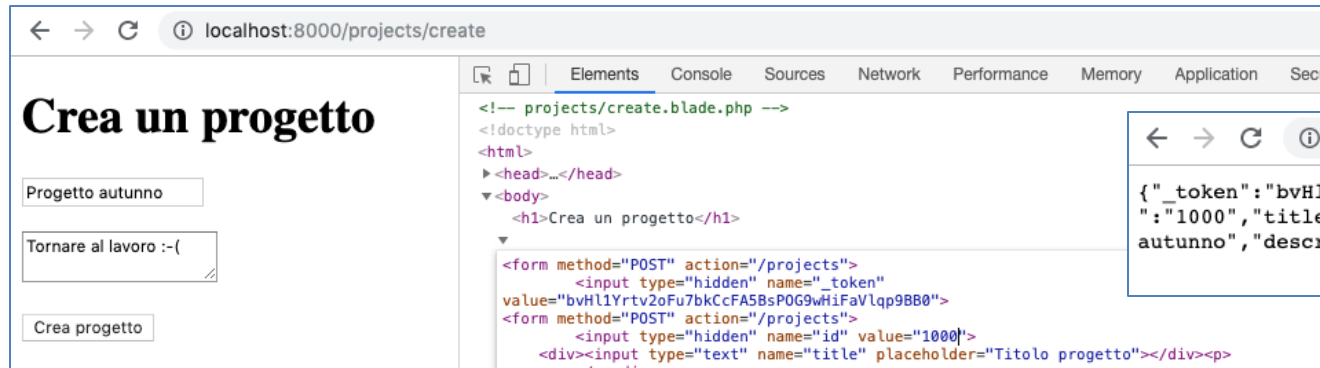
Come si ricorderà, il token era nascosto!

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

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

```
localhost:8000/projects

{
 "_token": "BxBnURS20qOCUeJzvAFkDrXySdAh7IcuwMlsZey",
 "title": "Progetto autunno",
 "description": "Tornare al lavoro :-("
}
```



localhost:8000/projects/create

**Crea un progetto**

Progetto autunno

Tornare al lavoro :-("

Crea progetto

Elements Console Sources Network Performance Memory Application Security

```
<!-- 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="bvH1Yrtv2oFu7bkCcFA5BsPOG9wHiFaVlqp98B0">
> <form method="POST" action="/projects">
> <input type="hidden" name="id" value="1000">
> <div><input type="text" name="title" placeholder="Titolo progetto"></div><p>
```

```
localhost:8000/projects

{
 "_token": "bvH1Yrtv2oFu7bkCcFA5BsPOG9wHiFaVlqp98B0", "id"
 ": "1000", "title": "Progetto
 autunno", "description": "Tornare al lavoro :-(" }
}
```

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!

# Callback ancora più concisi

Utilizziamo ovunque il model binding, il metodo *Project::create()* nel callback *store()* e il metodo (di istanza di *Project*) *update()*:

```
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
```