Giovanni Mavilla

SPIEGAZIONE DELL'ESERCITAZIONE

Premessa: la seguente esercitazione non sostituisce le slide e il materiale indicato dal prof, ma è un contenuto in più da poter consultare.

INFO UTILI:

- Progetto realizzato con Laravel 8
- Dati del DB (modificabili a piacere, ovviamente modificando anche il file .env)
    - DatabaseName: shop
    - User: admin
    - Pwd: admin

CREAZIONE PROGETTO

1. Installare il composer.exe
2. Installare l'installer di Laravel:

    ```
    composer global require "laravel/installer=~1.1"
    ```

3. Creare una nuova App
    a. **Tramite l'installer di Laravel**

        ```
        laravel new ${applicationName}
        ```

    b. **Tramite il generatore di Composer (consigliato)**

        ```
        composer create-project laravel/laravel ${applicationName} --prefer-dist
        ```

NOTA:

L'applicazione appena installata, oltre alle folder di struttura definite da Laravel, presenta la cartella `vendor` che raccoglie le dipendenze di terze parti. Al momento dell'inizializzazione del progetto essa conterrà esclusivamente Laravel ma successivamente potrà contenere ulteriori dipendenze.

In caso di utilizzo di un sistema per il versioning (**GIT** o SVN), tale cartella non dovrebbe essere condivisa, questo per non appesantire la gestione del progetto con file che possono essere scaricati dalla Rete. Una volta ricevuto un progetto già esistente basterà eseguire un

```
composer update
```

per ottenere tutte le dipendenze dalla Rete.

AVVIARE il server di Laravel

1. **Aprire un terminale**
2. **Andare dentro la cartella del progetto**
3. **Eseguire il comando:**

```
php artisan serve
```

CONTROLLER

Ad ogni rotta viene associata con un callback che è una “chiusura” (funzione anonima).

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

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

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

In questo esempio si ripete uno schema: ogni callback serve una Page(web)

La lista delle Route e dei relativi callback/closure potrebbe essere molto lunga.

Con il controller è possibile ricondurre tutti i callback a metodi di un gruppo di callback. Questo gruppo si chiamerà Page**s**Controller (NB:Page + s plurale)

Si può generare un controller tramite il tool artisan.

```
php artisan make:controller PagesController
```

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

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

   public function home(){

            $lista_azioni = ['azione1', 'azione2'];
            return view('home',
      [
         'azioni_pref'=> $lista_azioni,
         'msg'=> 'Benvenuti',
         'quando' =>'oggi',
         'data'=>request('data')
      ]
            );

   }

}
```

Web.php diventerà

```php
<?php
use Illuminate\Support\Facades\Route;
// va aggiunto questo use per il PagesController
use App\Http\Controllers\PagesController;


Route::get('/', [PagesController::class, 'home']);
Route::get('/contact', [PagesController::class, 'contact']);
```

COLLEGARE DATABASE ALL'APP

Creare un nuovo database e **modificare il file .env** sull'App

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=shop
DB_USERNAME=admin
DB_PASSWORD=admin
```

Effettuare una migrazione con il comando

```
php artisan migrate
```

Risultato di php artisan migrate

Migration table created successfully.

Migrating: 2014_10_12_000000_create_users_table

Migrated: 2014_10_12_000000_create_users_table (496.89ms)

Migrating: 2014_10_12_100000_create_password_resets_table

Migrated: 2014_10_12_100000_create_password_resets_table (334.41ms)

Migrating: 2019_08_19_000000_create_failed_jobs_table

Migrated: 2019_08_19_000000_create_failed_jobs_table (381.01ms)

PS C:\Users\giova\Desktop\Shop>

Per eliminare una migrazione (elimina l'effetto dell'ultimo comando migrate) – ogni chiamata migrate:rollback andrà indietro di una migrate

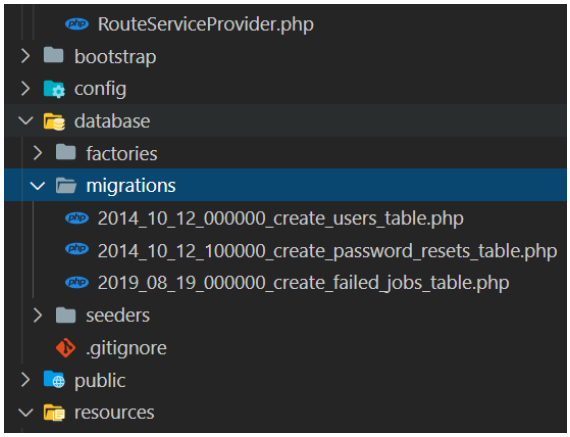
```
php artisan migrate:rollback
```

NB: un rollback si riferisce all'ultimo migrate effettuato

Per vedere lo stato di una migrazione

```
php artisan migrate:status
```

OGNI TABELLA corrisponde ad un file delle migrazioni dell'App. Questi sono i file che utilizza artisan migrate




```php
<?php

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

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

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

Ci sono due funzioni up() e down()

La funzione down()viene invocata con il migrate:rollback e cancella la tabella

La funzione up() viene invocate con migrate

Il metodo è create con:

- Primo argomento il nome della tabella
- Il secondo argomento è un callback con argomento un **Blueprint ($table)** che ha diversi metodi che saranno quelli che daranno luogo ai vari campi della tabella
  - **$table->id()**
  - **//crea il campo id (di default è autoincrement)**
  - **$table->string('name')**
  - **// crea un campo di tipo string (varchar (255)) di nome** name

Creare una nuova tabella (ad esempio order)

```
php artisan make:migration create_orders_table
```

NOTA: per creare le tabelle/entità usare nomi inglesi che al plurale aggiungono solo la "s"

(ad esempio: order->orders, book->books, …..)

MODELLI

Un modello è una classe (PHP) T, che dà accesso, con i suoi metodi, a una tabella t del DB relazionale che fa da backend dell'App

- Il programmatore non scrive query SQL, ma "vede" i record (righe) di t come istanze del modello
- Quindi usa i metodi dell'istanza sia per leggere lo stato che per modificarlo, quindi per leggere/modificare gli attributi del record della tabella t corrispondente all'istanza
- Quindi si può accedere al DB attraverso una classe-modello

ACTIVE RECORD in Laravel

L'implementazione di AR è affidata al componente eloquent

Il tool per generare le classi/tabelle per gli oggetti/record è sempre artisan

Creare una nuovo modello

```
php artisan make:model NameModello
```

- La convenzione dice che il NameModello 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 però è in minuscolo e plurale(-s)

NB: Per sperimentare con l'active record di Laravel, si usa il **tool tinker** che è una sorta di interprete dei comandi di Laravel (e PHP)

```
php artisan tinker
```

Da questa shell si posso fare delle query di select con i metodi all(), first(), latest()

Esempio:

```
>>> App\Models\Order::all();

>>> App\Models\Order::first();
```

Si può anche inserire un nuovo record nel database

```
>>> $order = new App\Models\Order();

>>> $order->num_order ='101';

>>> $order->date_order ='2021-01-16 15:30:00';

>>> $order->description ='ordine prova2';

>>> $order->amount =200.00;

>>> $order->save(); // con save() viene salvato il record nella tabella
```

CREARE APPLICAZIONE CRUD (create, read, update, delete)

1. Creare la tabella
2. Creare il modello Customer.php
3. Modificare il file migration della tabella customers
4. Effettuare una artisan migrate (crea la tabella nel db)
5. Creare un controller per il Customer: CustomersController.php
6. Creare una route (ad esempio /customers)
7. Creare una view per visualizzare i customers

Esempio Customer

1. Creare la tabella:

   php artisan make:migration create_customers_table

2. Creare il modello:

   php artisan make: model Customer

   (*c'è un metodo più avanzato per creare Model e Controller dell'entità/risorsa in un colpo solo)
   Verrà spiegato più avanti in questa guida

3. Nella cartella migration aprire e modificare il file….xxxx_create_customers_table.php
   (xxx sarà la data di creazione del file)
   Modificare il metodo up()

```
public function up()

  {
     Schema::create('customers', function (Blueprint $table) {
        $table->id();
        $table->string("name");
        $table->string("surname");
        $table->string("address");
        $table->string("email")->unique();
        $table->string("phone");
        $table->timestamps();
     });
  }
```

4. Creare la tabella nel DB:

   php artisan migrate

5. Creare un controller per Customer

(*c'è un metodo più avanzato per creare Model e Controller dell'entità/risorsa in un colpo solo)
Verrà spiegato più avanti in questa guida

php artisan make:controller CustomersController

a. ha un metodo index (richiamato dalla route)
b. effettua una select con \App\Models\Customer::all();
c. ritorno la view index passando come parametro il risultato della select

```
class CustomersController extends Controller
{
   public function index(){
      $clienti = \App\Models\Customer::all();
      return view('customers.index',
         ['clienti' => $clienti]
      );
   }
}
```

6. Creare una route

in web.php : aggiungere una ruote per customer e richiama un metodo del Controller

(*per il momento impostiamo le route singolarmente
per le route STANDARD che richiameranno i metodi standard CRUD più avanti in questa guida verrà spiegato come raggruppare le route in un gruppo di route
La cosa importante è mantenere le convenzioni di Laravel

```
Route::get('/customers', [CustomersController::class, 'index']);
```

7. Creare la view index.blade.php

Questa view visualizzerà sulla url ../customers l'elenco di tutti i customer

```
@section('body')
   <br>
   <button onclick="window.location.href='/customers/create'">Inserisci nuovo cliente</button>

   <br>
   @foreach ($clienti as $user)
      <p>User id: {{ $user->id }}</p>
      <li>Nome: {{ $user->name }}</li>
      <li>Cognome: {{ $user->surname }}</li>
      <li>Indirizzo: {{ $user->address }}</li>
      <li>Email: {{ $user->email }}</li>
      <li>Telefono: {{ $user->phone }}</li>
   @endforeach

@endsection
```

INSERIMENTO DI UN NUOVO CLIENTE

Intro: per inserire un nuovo cliente serve una pagina che abbia un form per l'inserimento dei dati e dei metodi che gestiscano questo inserimento

1. Inseriamo una nuova route che richiama un **metodo create** del Controller

```
Route::get('/customers/create', [CustomersController::class, 'create']);
```

2. Aggiungiamo il metodo create del Controller (questo metodo richiama una view)
   NB: customers.create è una sorta di path, infatti la view che andremo a creare (create.blend.php) la inseriremo all'interno di una sottocartella (customers) di Views.

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

3. Creiamo la view **create.blend.php** con il form
   (sarebbe la view parametro del metodo create) per l'inserimento dei dati
   a. **NOTA:**
      Va aggiunto **{{csrf_field}}**, Cross-Site Request Forgery, e serve ad inserire un campo invisibile che prenderà un valore random **(un token di sicurezza)**
      Su Laravel un POST inviato senza questo token genera un errore

```
<!DOCTYPE html>
<html lang="it">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <meta http-equiv="X-UA-Compatible" content="ie=edge">
   <title>Nuovo cliente</title>
</head>
<body>
   <h1>Inserisci nuovo utente</h1>

     {{ csrf_field() }}

   <form method="POST" action="/customers">
   <div><input type="text" name="name" placeholder="nome"></div>
   <div><input type="text" name="surname" placeholder="cognome"></div>
   <div><input type="text" name="address" placeholder="address"></div>
   <div><input type="text" name="email" placeholder="email"></div>
   <div><input type="text" name="phone" placeholder="phone"></div>
   <div><button type="submit">Inserisci cliente</button></div>
   </form>
</body>
</html>
```

4. Aggiungere una nuova Route di tipo post per /customers con metodo store

```
Route::post('/customers', [CustomersController::class, 'store']);
```

5. Aggiungere il metodo store nel controller

```
//salva i dati nel db
   public function store(){
      //return request('name');
      $customer = new \App\Models\Customer();
      $customer->name = request('name');
      $customer->surname = request('surname');
      $customer->name = request('address');
      $customer->email = request('email');
      $customer->phone = request('phone');
      $customer->save();
      return redirect('/customers');
   }
```

MODELLI E RISORSE

I Models in Laravel sono delle risorse (equivalenti delle entità in Spring Boot), ad esempio Customer è un modello/risorsa.
Nell'interezione con la wep app, il nome della risorsa (customers) è il primo componente di un gruppo di URL/route
Attraverso le ruote si può interagire con la risorsa.
Un possibile schema di rotte in web.php potrebbe essere:

QUESTE SONO LE 7 ROTTE STANDARD DI UN'APPLICAZIONE REST

```php
<?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
```

- 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)*

Nel controller standard ci sono questi 7 metodi e bisogna implementare:

- **Le 4 operazioni CRUD (Create, Read - su Laravel Read corrisponde a "show", Update e Destroy)**
- **Store che fa da supporto a Create (Create mostra un form, store memorizza i nuovi dati sul DB)**
- **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
/*
GET /customers (index)
POST /customers (store)  //nuovo cliente nel DB (cf. create)

GET /customers/create (create) //page con form per inserire i dati
GET /customers/1 (show in base all'id)
PATCH /customers/1 (update) //update del record n. 1
DELETE /customers/1 (destroy) //delete dal DB del recordo n. 1

GET /customers/1/edit (edit) //get form con i dati del record 1 per eventuali modifiche

*/


Route::get('/customers', [CustomersController::class, 'index']);

Route::get('/customers/create', [CustomersController::class, 'create']);
Route::post('/customers', [CustomersController::class, 'store']);

Route::get('/customers/{customer}', [CustomersController::class, 'show']);

Route::get('/customers/{customer}/edit', [CustomersController::class, 'edit']);

Route::patch('/customers/{customer}', [CustomersController::class, 'update']);

Route::delete('/customers/{customer}', [CustomersController::class, 'destroy']);
```

Create le route vanno creati anche tutti metodi corrispondenti nel Controller

NOTA: usare le convenzioni per le route e per i metodi può semplificare la programmazione.

Tutte le route sopra si posso raggruppare in una route “collettiva” relativa alla singola risorsa

```
//route collettiva per Customer
Route::resource('/customers', CustomersController::class);
```

Visualizzare la liste delle route

```
php artisan route:list
```

```
PS C:\Users\giova\Desktop\Shop> php artisan route:list
+--------+-----------+--------------------------+-------------------+-----------------------------------------------+------------+
| Domain | Method    | URI                      | Name              | Action                                        | Middleware |
+--------+-----------+--------------------------+-------------------+-----------------------------------------------+------------+
|        | GET|HEAD  | /                        |                   | App\Http\Controllers\PagesController@home     | web        |
|        | GET|HEAD  | api/user                 |                   | Closure                                       | api        |
|        |           |                          |                   |                                               | auth:api   |
|        | GET|HEAD  | contact                  |                   | App\Http\Controllers\PagesController@contact  | web        |
|        | GET|HEAD  | customers                | customers.index   | App\Http\Controllers\CustomersController@index | web        |
|        | POST      | customers                | customers.store   | App\Http\Controllers\CustomersController@store | web        |
|        | GET|HEAD  | customers/create         | customers.create  | App\Http\Controllers\CustomersController@create | web        |
|        | GET|HEAD  | customers/{customer}     | customers.show    | App\Http\Controllers\CustomersController@show  | web        |
|        | PUT|PATCH | customers/{customer}     | customers.update  | App\Http\Controllers\CustomersController@update | web        |
|        | DELETE    | customers/{customer}     | customers.destroy | App\Http\Controllers\CustomersController@destroy | web        |
|        | GET|HEAD  | customers/{customer}/edit | customers.edit    | App\Http\Controllers\CustomersController@edit  | web        |
+--------+-----------+--------------------------+-------------------+-----------------------------------------------+------------+
PS C:\Users\giova\Desktop\Shop>
```

Con una ruote collettiva si può **utilizzare il Wizard per boilerplate nel controller**

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

```
PS C:\Users\giova\Desktop\Shop> php artisan help make:controller
Description:
  Create a new controller class

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

Arguments:
  name                  The name of the class

Options:
      --api             Exclude the create and edit methods from the controller.
      --force           Create the class even if the controller already exists
  -i, --invokable       Generate a single method, invokable controller class.
  -m, --model[=MODEL]   Generate a resource controller for the given model.
  -p, --parent[=PARENT] Generate a nested resource controller class.
  -r, --resource        Generate a resource controller class.
  -h, --help            Display help for the given command. When no command is given display help for the list command
  -q, --quiet           Do not output any message
  -V, --version         Display this application version
      --ansi            Force ANSI output
      --no-ansi         Disable ANSI output
  -n, --no-interaction  Do not ask any interactive question
      --env[=ENV]       The environment the command should run under
  -v|vv|vvv, --verbose  Increase the verbosity of messages: 1 for normal output, 2 for more verbose output and 3 for debug
PS C:\Users\giova\Desktop\Shop>
```

Generando il controller con **le [options] –r –m** verranno generati in automatico da artisan il **Modello e il Controller** con tutti i metodi base sopra descritti

```
php artisan make:controller CustomersController -r -m Customer
```

Questo comando genera i file CustomersController.php (con lo scheletro dei metodi base) e il model Customer.php

I metodi ovviamente vanno implementati

IMPLEMENTAZIONE DEL METODO UPDATE e EDIT

IMPLEMENTAZIONE DEL METODO edit()

a. Bisogna creare la pagina di edit (con un form che carica i dati del record da modificare)
b. Passiamo come parametro un oggetto customer (selezionato in base all'id)

NOTA: successivamente sfruttando il Model bindig passeremo come parametro l'oggetto Customer anziché l'id

```php
public function edit($id)
  {
    $customer = Customer::find($id);
    return view('customers.edit', compact('customer'));
  }
```

Customers.edit.blade.php

```
<!DOCTYPE html>
<html lang="it">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Nuovo cliente</title>
</head>
<body>
  <h1>Modifica cliente</h1>
  {{--PROBLEMA con il metodo di tipo PATCH :  il browser non lo invia
    SOLUZIONE: invare un post e aggiungere un nuovo campo input nascosto
    con name="_method" e value="PATCH"
  --}}
  <form method="POST" action="/customers/{{$customer->id}}">
    {{ csrf_field() }}
    <div><input type="hidden" name="_method" value="PATCH"></div>

    <div><input type="text" name="name" placeholder="nome" value="{{$customer->name}}"></div>
    <div><input type="text" name="surname" placeholder="cognome" value="{{$customer->surname}}"></div>
    <div><input type="text" name="address" placeholder="address" value="{{$customer->address}}"></div>
    <div><input type="text" name="email" placeholder="email" value="{{$customer->email}}"></div>
    <div><input type="text" name="phone" placeholder="phone" value="{{$customer->phone}}"></div>
    <div><button type="submit">Aggiorna cliente</button></div>
  </form>
</body>
</html>
```

PROBLEMA : I browser non inviano form di tipo PATCH, quindi come visto sopra con **l'aggiunta dell'input nascosto** con name="_method" e value="PATCH", Laravel riesce a capire che si tratta di un update

In alternativa si possono usare anche degli helper blade più concisi come:

```
{{method_field('PATCH')}}
```

Oppure

```
@method('PATCH')
```

L'action del form edit.blade.php, sarà una PATCH che richiamerà il metodo update() che dovrà fare l'update sul DB.

IMPLEMENTAZIONE DEL METODO update()

```
public function update($id)
  {

    //dump-and-die è una funzione del php
    // che restituisce il contenuto del suo argomento
    //dd(request()->all());
    $customer = Customer::find($id);
    $customer->name = request('name');
    $customer->surname = request('surname');
    $customer->address = request('address');
    $customer->email = request('email');
    $customer->phone = request('phone');
    $customer->save();
    return redirect('/customers');
  }
```

IMPLEMENTAZIONE DEL METODO destroy()

- Per richiamare questo metodo che corrisponde alla Route
  `Route::delete('/customers/{customer}', [CustomersController::class, 'destroy']);`
- Usiamo la stessa view edit.blade.php, aggiungendo un secondo form POST con @method('DELETE')

```
  <div><input type="text" name="phone" placeholder="phone" value="{{$customer->phone}}"></div>

    <br>
    <div><button type="submit">Aggiorna cliente</button></div>
  </form>
  <br>
  <form method="POST" action="/customers/{{$customer->id}}">
    @csrf
    @method('DELETE')
    <div><button type="submit">Elimina cliente</button></div>
  </form>
```

Metodo destroy()

```
public function destroy($id)
  {
    $customer = Customer::find($id)->delete();
    return redirect('/customers');
  }
```

IMPLEMENTAZIONE DEL METODO show()

```
public function show($id)
   {
      $customer = Customer::findOrFail($id);
      return view('customers.show', compact('customer'));
   }
```

Relativa view richiamata (show.blade.php)

```
@extends('template')

@section('title','Cliente')

@section('intestazioneh1')
   Customer id: {{ $customer->id }}
@endsection

@section('body')

   <br>
   <li>{{ $customer->name }}</li>
   <li>{{ $customer->surname }}</li>
   <li>{{ $customer->address }}</li>
   <li>{{ $customer->email }}</li>
   <li>{{ $customer->phone }}</li>
   <br>
   <a href="/customers/{{ $customer->id}}/edit"> Modifica Cliente</a>
@endsection
```

MODEL BINDING nei callback → al metodo viene passato direttamente l'oggeto Customer

Utilizzando il model binding **si può eliminare l'istruzione che recupera il dato con find o findOrFail**

Esempio di modifica del metodo show()

```
public function show(Customer $customer)
  {
    //$customer = Customer::findOrFail($id);
    return view('customers.show', compact('customer'));
  }
```

- Con il model binding si posso gestire rotte che contengono un id numerico.
- Si passa un parametro che 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 avrà un 404 – quindi l'equivalente della funzione findOrFail()
- NOTA: come personalizzare la colonna nella definizione del parametro del percorso
    - **Esempio:**
      In questo esempio il MODELLO è post
      e COLONNA personalizzata è slug anziché id

```
use App\Models\Post;
use App\Models\User;

Route::get('/users/{user}/posts/{post:slug}', function (User $user, Post $post) {
    return $post;
});
```

Facility dei modelli per il METODO store()

Il metodo store() (metodo che salva i dati nel DB) può essere modificato con un'altra facility dei modelli, utilizzando il metodo create()

```
public function store()
  {
     /*
     $customer = new \App\Models\Customer();
     $customer->name = request('name');
     $customer->surname = request('surname');
     $customer->address = request('address');
     $customer->email = request('email');
     $customer->phone = request('phone');
     $customer->save();
     return redirect('/customers');
     */

     //function store con la facility create()
     Customer::create([
        'name' => request('name'),
        'surname' => request('surname'),
        'address' => request('address'),
        'email' => request('email'),
        'phone' => request('phone')
     ]);
     return redirect('/customers');
  }
```

Create prende come argomento un array hash che contiene le coppie chiave-valore che corrispondono agli attributi della risorsa/modello.

Per una questione di sicurezza l'assegnazione in massa di attributi causa un errore. Per evitare questo errore e consentire questa assegnazione di massa bisogna dichiarare nel file modello quali attributi del modello si desidera rendere assegnabili in massa.

Un modo è utilizzando la **$fillable** proprietà sul modello.

```
class Customer extends Model
{
   use HasFactory;
   protected $fillable = ['name','surname','email','address','phone'];
}
```

$fillable sarà un array che contiene gli attributi per l'inserimento di massa.

ATTENZIONE:
se l'attributo **non viene inserito** in questo array, la query SQL non inserirà il valore nel DB.

Altra cosa bisogna modificare anche la struttura della tabella nel file migration opportunamente specificando eventuali valori che possono assumere valore null.

Esempio: Modifica al file migration di customer

```
public function up()
  {
    Schema::create('customers', function (Blueprint $table) {
      $table->id();
      $table->string("name");
      $table->string("surname");
      $table->string("address");
      $table->string("email")->unique();
      $table->string("phone")->nullable(true)->default('000-000');
      $table->timestamps();
    });
  }
```

- I campi name, surname e address definiti in questo modo NON posso avere un valore null.
- Email non può avere un valore null.
- Phone può essere null, quindi se nel form di inserimento il campo viene lasciato vuoto, non ci sarà un errore.
- Per assegnare però un valore di default va modificato anche il **metodo store()** in questo modo. (ad esempio vogliamo che phone se viene lasciato vuoto allora assume di default il valore "000-000"

```
public function store()
    {

        //function store con la facility create()
        Customer::create([
            'name' => request('name'),
            'surname' => request('surname'),
            'address' => request('address'),
            'email' => request('email'),
            'phone' => request('phone') ?? '000-000'
        ]);
        return redirect('/customers');
    }
```

NOTA:

dopo ogni modifica di un file migration bisogna effettuare un rollback e una nuova migration per vedere le modifiche. Una nuova migration cancellerà i dati nel DB.

Come evitare di cancellare i dati?

PROCEDIMENTO di esempio PER MODIFICARE il campo di una tabella (effettuare un alter table tramite artisan)

Premessa: va installato doctrine/dbal

```
composer require doctrine/dbal
```

Esempio di modifica del campo name

DA:

```
DA:
    $table->string("name");
```

A:

```
    $table->string("name",50);
```

1. **Creare un nuovo file di migration con artisan**

```
php artisan make:migration change_customer_name_column_type
```

2. **Modificare il metodo up() del file migration appena creato**
   Con questo nuovo Shcema::talbe e con il metodo change() stiamo specificando che vogliamo cambiare le proprietà dell'attributo name

```
public function up()
  {
    Schema::table('customers', function (Blueprint $table) {
      $table->string("name",50)->change();
    });
  }
```

3. **Effettuare una nuova migrate con artisan**

```
php artisan make:migration change_customer_name_column_type
```

TABELLA ordini e RELAZIONE con customers

L'idea sarebbe di realizzare una web app con due entità/risorse (customers, orders) con una relazione 1 a molti, ovvero un customer può fare più ordini ma un ordine corrisponde ad un solo customer.

- **Creiamo la nuova tabella nel DB**

```
php artisan make:migration create_orders_table
```

- **Modificare il metodo up() del file migratrion appena creato (inserire i campi della tabella)**

```php
public function up()
  {
    Schema::create('orders', function (Blueprint $table) {
      $table->id();
      $table->bigInteger('customer_id')->unsigned()->nullable(false); //sarà la chiave esterna
      $table->string('description');
      $table->float('amount');
      $table->date('date_order');
      $table->boolean('completed')->default(false);
      $table->timestamps();
      $table->foreign('customer_id')
          ->references('id')
          ->on('customers')
          ->onCascade('delete');
    });
  }
```

- **Effettuare la migrate**

```
php artisan migrate
```

- **Creare il modello e il controller per ORDER**

```
php artisan make:controller OrdersController -r -m Order
```

- **Modificare i model di Order e Customer**

Sul model Order add function customer() => sarà una proprietà di un oggetto Order che permetterà di estrarre il customer associato

```
class Order extends Model
{
   use HasFactory;

   public function customer(){
      return $this->belongsTo(Customer::class);
   }

}
```

Sul model Customer add function orders() => sarà una proprietà di un oggetto Customer che permetterà di estrarre tutti gli ordini effettuati

```
class Customer extends Model
{
   use HasFactory;
   protected $fillable = ['name','surname','email','address','phone'];

   public function orders()
   {
      return $this->hasMany(Order::class);
   }

}
```

Dato un Customer $customer, ci si potrà riferire a

$customer->orders

NOTA: ci si deve riferire a orders come proprietà dell'oggetto $customer, e non come metodo orders().

Inserimento delle route STANDARD nel file web.php

```
use App\Http\Controllers\OrdersController;
```

```
//route collettiva per Order
Route::resource('/orders', OrdersController::class);
```

IMPLEMENTAZIONE DEI METODI DI OrdersController:

index(), create(), store(), show(), edit(), update(), destroy()

METODO: index()

```
public function index()
   {
      $ordini = \App\Models\Order::all();
      return view('orders.index',
         ['ordini' => $ordini]
      );
   }
```

View: index.blade.php

```
@extends('template')

@section('title','Orders')

@section('intestazioneh1')
   Lista Ordini
@endsection

@section('body')

   <br>
   @foreach ($ordini as $ordine)

      <p>Order id: {{ $ordine->id }}</p>
      <li>Customer id: {{ $ordine->customer_id }}</li>
      <li>Description: {{ $ordine->description }}</li>
      <li>Amount: {{ $ordine->amount }}</li>
      <li>Date: {{ $ordine->date_order }}</li>
      @if ($ordine->completed)
      <li>Status: completed</li>
      @else
      <li>Status: NOT completed</li>
      @endif

   @endforeach

@endsection
```

METODO: create()

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

View: create.blade.php

```
<!DOCTYPE html>
<html lang="it">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <meta http-equiv="X-UA-Compatible" content="ie=edge">
   <title>Nuovo ordine</title>
</head>
<body>
   <h1>Inserisci nuovo ordine</h1>
   <form method="POST" action="/orders">
      {{ csrf_field() }}
   <div><input type="number" name="customer_id" placeholder="customer_id"></div>
   <div><input type="text" name="description" placeholder="description"></div>
   <div><input type="number" step="0.01" name="amount" placeholder="amount"></div>
   <div><input type="date" name="date_order" placeholder="date_order"></div>
   <div>
      <select name="completed">
         <option value="0">Not Completed</option>
         <option value="1">Completed</option>
      </select>
   </div>
   <br>
   <br>
   <div><button type="submit">Inserisci ordine</button></div>
   </form>

</body>
</html>
```

METODO: store()

```
public function store(Request $request)
   {

      $order = new \App\Models\Order();
      $order->customer_id = request('customer_id');
      $order->description = request('description');
      $order->amount = request('amount');
      $order->date_order = request('date_order');
      $order->completed = request('completed');
      $order->save();
      return redirect('/orders');

   }
```

METODO: show()

```
public function show(Order $order)
   {
      return view('orders.show', compact('order'));
   }
```

View: show.blade.php

```
@extends('template')

@section('title','Ordine')

@section('intestazioneh1')
   Order id: {{ $order->id }}
@endsection

@section('body')

   <br>
   <li>Customer id: {{ $order->customer_id }}</li>
   <li>Description: {{ $order->description }}</li>
   <li>Amount {{ $order->amount }}</li>
   <li>Date {{ $order->date_order }}</li>
   <li>Status {{ $order->completed }}</li>
   <br>

   <a href="/orders/{{ $order->id}}/edit"> Modifica Ordine</a>

@endsection
```

METODO: edit()

```
public function edit(Order $order)
    {
        return view('orders.edit', compact('order'));
    }
```

View: edit.blade.php

```
<!DOCTYPE html>
<html lang="it">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Edit ordine</title>
</head>
<body>
    <h1>Modifica ordine</h1>


    <form method="POST" action="/orders/{{$order->id}}">
        @csrf
        @method('PATCH')

        <div><input type="number" name="customer_id" placeholder="customer_id"value="{{$order->customer_id}}"></div>
        <div><input type="text" name="description" placeholder="description" value="{{$order->description}}"></div>
        <div><input type="number" step="0.01" name="amount" placeholder="amount" value="{{$order->amount}}"></div>
        <div><input type="date" name="date_order" placeholder="date_order" value="{{$order->date_order}}"></div>
        <div>
            <select name="completed">
                <option
                    @if($order->completed == 0)
                    selected="selected"
                    @endif
                    value="0">Not Completed</option>

                    <option
                    @if($order->completed == 1)
                    selected="selected"
                    @endif
                    value="1">Completed</option>
            </select>
        </div>
        <div><button type="submit">Aggiorna ordine</button></div>
    </form>
    <br>


    <form method="POST" action="/orders/{{$order->id}}">
        @csrf
        @method('DELETE')
        <div><button type="submit">Elimina ordine</button></div>
    </form>

</body>
</html>
```

METODO: update()

```
public function update(Request $request, Order $order)
   {
      $order->customer_id = request('customer_id');
      $order->description = request('description');
      $order->amount = request('amount');
      $order->date_order = request('date_order');
      $order->completed = request('completed');
      $order->save();
      return redirect('/orders');
   }
```

METODO: destroy()

```
public function destroy(Order $order)
   {
      $order->delete();
      return redirect('/orders');
   }
```

MODIFICA del metodo show() di Customer per elencare gli ordini di un cliente

Aggiungiamo il seguente codice

```
{{--
   lista ordini di questo cliente: sfruttando la proprietà del Model orders
   facciamo un if con il metodo count() per verificare che esistano ordini
--}}
<h3>Lista ordini di {{$customer->name}} </h3>
@if ($customer->orders->count())
<div>
   @foreach ($customer->orders as $order)
      <li>ID ordine: {{$order->id}}</li>
      <li>Description: {{$order->description}}</li>
      <li>Date: {{$order->date_order}}</li>
      <li>Amount: {{$order->amount}}</li>
      <li>Status: {{$order->completed}}</li>
   @endforeach

</div>
@endif
```

MODIFICA del metodo show() di Order per elencare il cliente dell'ordine

Aggiungiamo il seguente codice che usa la proprietà customer del Model Order per recuperare i dati del cliente

```
 {{--
   inserimento del cliente che ha effettuato l'ordine
   sfruttando la proprieta customer del Model Order
   $order->customer
--}}
<div>
   <h3>Cliente</h3>

   <li>ID cliente: {{$order->customer->id}}</li>
   <li>name: {{$order->customer->name}}</li>
   <li>surname: {{$order->customer->surname}}</li>
   <li>address: {{$order->customer->address}}</li>
   <li>email: {{$order->customer->email}}</li>
   <li>phone: {{$order->customer->phone}}</li>
</div>
```