

Laravel

Applicazioni Web

Architettura di una WebApp

- Caratteristiche di una **Applicazione Web**:
 - Nella vita di un'applicazione, l'**interfaccia** cambia più rapidamente delle **funzioni elaborative** o del **modello dei dati**;
 - **Stessi** dati possono essere visualizzati in modi (o attraverso canali) **differenti**;
 - Le **competenze** per la costruzione delle interfacce **sono diverse** da quelle necessarie per lo sviluppo delle funzioni elaborative o del modello dei dati;
 - Una **stessa interfaccia** può visualizzare il **risultato** dell'attivazione **di più funzioni elaborative**
- **Esigenza**: rendere modulare l'applicazione

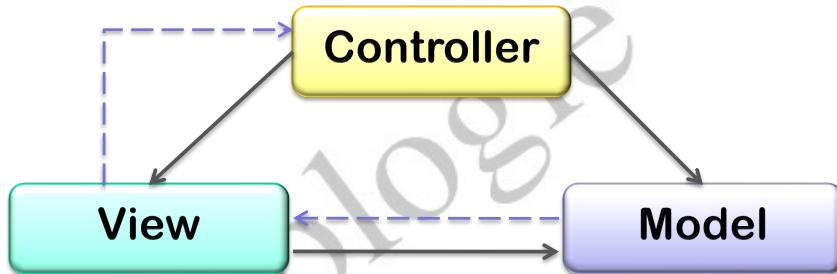
Architettura di una WebApp

- Soluzione: **separare** architetturalmente l'**interfaccia**, il **modello dei dati** (incluse le elaborazioni) e la **gestione dell'interazione** con l'utente
- Architettura **Model-View-Controller (MVC)**
 - **Model**: rappresenta e gestisce i **dati** dell'applicazione, modificandoli su indicazione del Controller o fornendoli per la visualizzazione alla View
 - **View**: implementa l'**interfaccia utente** dell'applicazione
 - **Controller**: cattura gli **eventi generati dall'utente** e **attiva** il **Model** e/o la **View** per la loro gestione.

Laravel

3

MVC Design Pattern

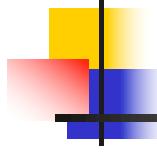


→ Associazione diretta

→ Associazione indiretta (ad esempio, attraverso il pattern "Observer")

Laravel

4

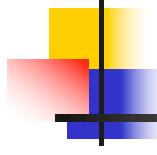


Caratteristiche MVC

- Benefici
 - Possibilità di implementare viste multiple sullo stesso modello
 - Possibilità di adattare la vista alle richieste/esigenze dell'utente
 - Confinamento di tutti i cambiamenti dell'interfaccia nella View, senza implicazioni per il Model o il Controller
- Costi
 - Architettura complessa (rispetto a quella 'classica')
 - Appesantimento dell'interazione tra i moduli dovuto al loro disaccoppiamento

Laravel

5



Implementazione MVC

- Usando un'architettura 'Classica'
 - Definizione: collezione di moduli software sviluppati con linguaggi dedicati (HTML, CSS, PHP, JavaScript, ...)
 - Caratteristiche:
 - Funzioni distribuite su molti file
 - Applicazione non strutturata
 - Differenti stili di codifica
 - Classi (modello OOP) non organizzate
 - Problemi:
 - Manutenzione difficile
 - Nessuna standardizzazione
 - Documentazione difficile
 - Difficoltà di sviluppo in team

Laravel

6

Implementazione MVC

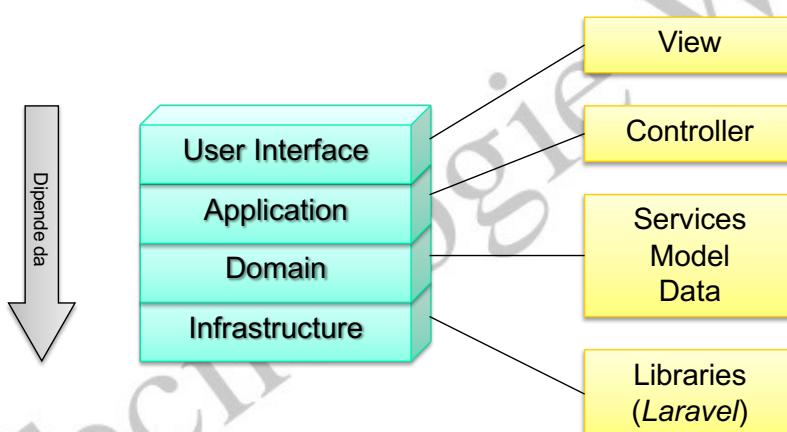
■ Usando un 'Framework'

- **Definizione:** insieme di programmi di supporto, librerie ed altri strumenti che consentono di sviluppare ed integrare le diverse componenti software di un'applicazione
- **Caratteristiche:**
 - Funzioni organizzate in librerie
 - Struttura dell'applicazione definita
 - Stile di codifica omogeneo
 - Classi (modello OOP) organizzate
- **Vantaggi:**
 - Manutenzione più facile
 - Standardizzazione spinta
 - Funzioni e Classi predefinite e facili da usare
 - Architettura dell'applicazione standard (basata su 'design patterns')

Laravel

7

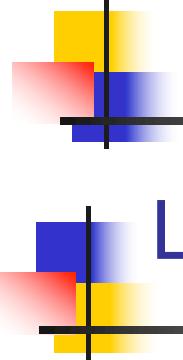
MVC Application Stack



- Attenzione a rispettare lo schema di dipendenza

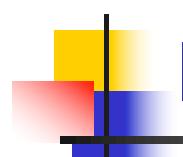
Laravel

8



Laravel

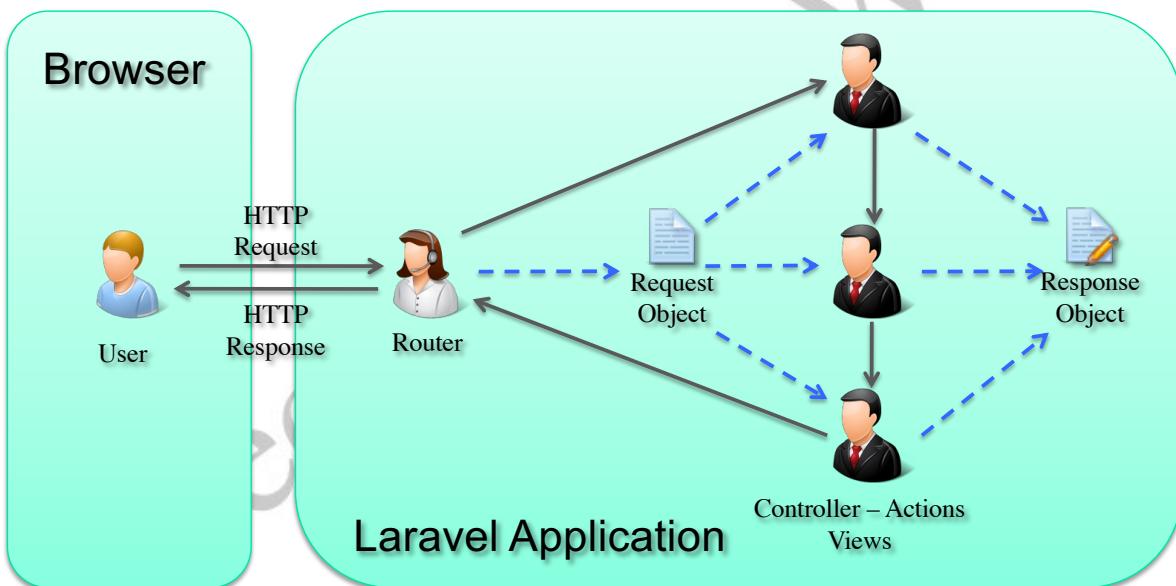
Fondamenti



Il Framework Laravel

- Framework open-source per lo sviluppo di applicazioni e servizi Web basato su PHP
 - Prima versione Luglio 2011
 - Versione ultima: Laravel 10 – Febbraio 2023
 - Versione usata nel corso: Laravel 9 – Febbraio 2022
- Basato su una libreria di componenti codificate come classi PHP
 - Modello di programmazione: Object Oriented
- Adotta l'MVC come pattern nativo per lo sviluppo
 - Ma le componenti possono essere usate anche singolarmente in applicazioni non MVC

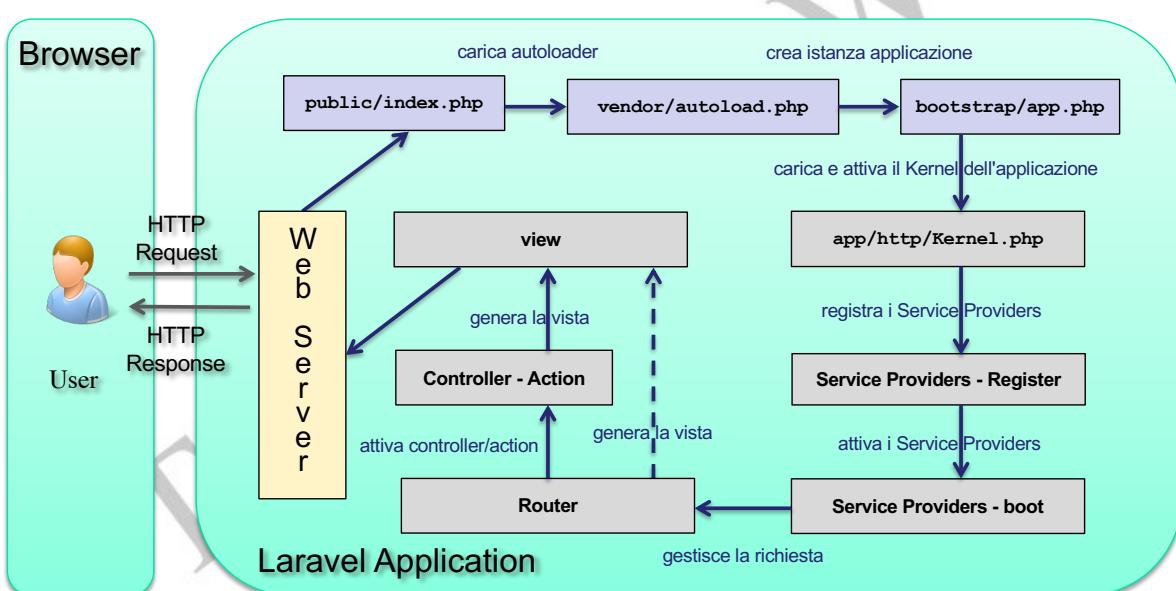
Laravel Workflow



Laravel

11

Laravel Workflow - Dettaglio



Laravel

12

Laravel - Concetti di Base

- **Routing** su base **URL**:
 - In una richiesta HTTP, la parte dell'URL che segue in nome di dominio (**rotta**) viene associata al processo di elaborazione che genera la risposta

http://www.mio-sito.it/prodotti
 - La **rotta** può contenere uno o più **parametri** ed i relativi **valori**

http://www.mio-sito.it/prodotti/id/7
 - Ogni **rotta** viene associata ad uno o più **verbi** HTTP (**get, post, put, patch, delete, options**)
- **Controller**: classe **derivata** dalla classe predefinita **Controller**
- **Action**: metodo pubblico del controller

Laravel

13

Laravel - Concetti di Base

- **View**
 - genera il **Response Object** a partire da un **Template** nel quale vengono **iniettati** i dati prodotti dalle action
 - usa il **Templating Engine BLADE**
- **Template**:
 - File **HTML** che definisce il contenuto del Response Object includendo direttive **BLADE** e/o codice **PHP**
 - File con estensione **.blade.php** memorizzato nella cartella **resources/views/** dell'applicazione (o in una sottocartella)
 - Il contenuto viene **tradotto in PHP** e **memorizzato nella cache dell'applicazione**, per poter essere utilizzato in modo più efficiente
 - Visualizzato dall'Helper **view(<nome_template>)**

Laravel

14

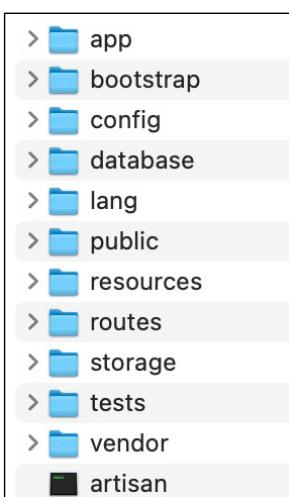
Gestione degli Errori

- Tutte le **eccezioni** (**errori**) generate in fase di esecuzione dell'applicazione **vengono automaticamente catturate e gestite** tramite i metodi della classe **App/Exceptions/Hanler**
- I **metodi** della classe che gestiscono le eccezioni sono due:
 - **report**: **registra l'eccezione** in un file locale o la invia ad un servizio esterno per la gestione dei log (**Flare**, ...)
 - **render**: **genera una pagina HTML** con le informazioni relative all'eccezione e la invia al browser
- Il **dettaglio dell'informazione generata** può essere definito tramite il **settaggio della variabile APP_DEBUG** nel file di configurazione dell'applicazione (**.env**)
- I **metodi** possono essere modificati per adattarli alle esigenze specifiche dell'applicazione

Laravel

15

Struttura Directory di Progetto



- Nuovo progetto creato con il comando
`laravel new <nome_cartella_progetto>`
- In **public** è presente anche il file nascosto **.htaccess**

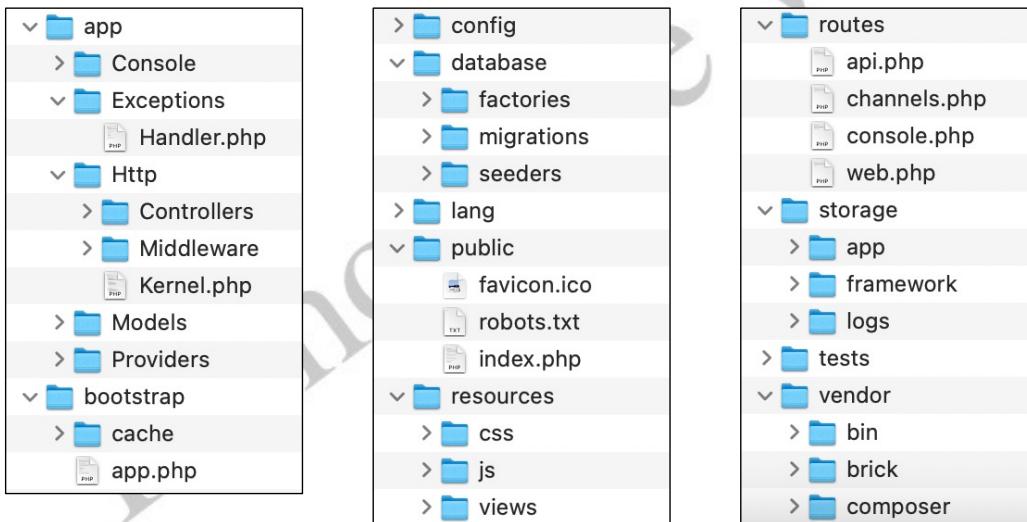
```
<IfModule mod_rewrite.c>
...
RewriteEngine On
...
# Send Requests To Front Controller...
RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^ index.php [L]

</IfModule>
```

Laravel

16

Struttura: Dettaglio

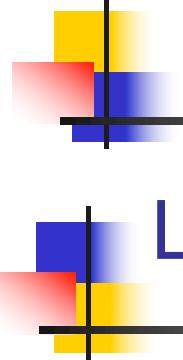


Laravel

17

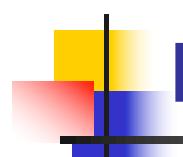
Esempio

- Progetto laraProj0 ([laravel v9](#))
 - **Installazione: alternativa 1**
 - Installare e configurare [Composer](#) (dependency manager per PHP)
 - Creare il progetto nella directory corrente con il comando
`composer create-project --prefer-dist laravel/laravel laraProj0 "9.0.*"`
 - **Installazione: alternativa 2**
 - Utilizzare la [cartella con il progetto già configurato](#) disponibile nella sezione Esempi Laravel del sito del corso
- Obiettivi didattici. Illustrare:
 - La [struttura](#) di base di un'applicazione Laravel
 - Le [rotte](#) e le [viste](#)
 - Il [workflow](#) dell'applicazione
 - Il [report](#) degli errori



Laravel

Componenti Principali

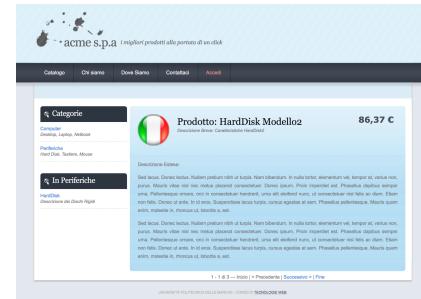


Blade: Templating Engine

- Permette di "sanificare" i valori delle variabili PHP passate al template applicando la funzione PHP `htmlspecialchars`
 - `{{ $username }}`
- Consente l'uso di direttive di controllo nel Template
 - Condizionale:
 - `@if`, `@elseif`, `@else`, `@endif`
 - `@switch`, `@case`, `@break`, `@default`
 - Iterativo:
 - `@for`, `@endfor`
 - `@foreach`, `@endforeach`
 - `@while`, `@endwhile`
 - Di autenticazione:
 - `@auth`, `@endauth`
 - `@guest`, `@endguest`

Blade: Templating Engine

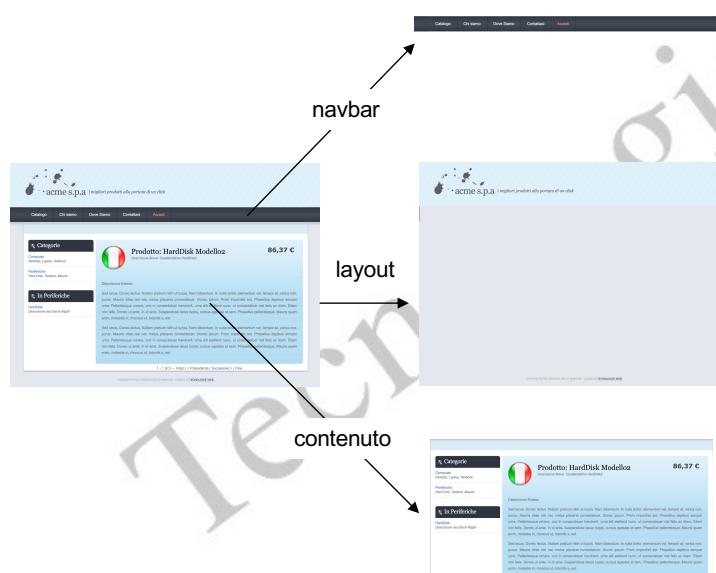
- Definisce **direttive** specifiche per la gestione delle **Form HTML**
 - `@csrf`
 - `@error`, `@enderror`
- Permette di definire **Layout** per i **Template** attraverso:
 - *Sub-View Inclusion*
 - `@include`
 - *Template Inheritance*
 - `@extends`
 - `@parent`
 - *Sections definition*
 - `@section`, `@endsection`
 - `@section`, `@show`
 - `@yield`



Laravel

21

Blade: Layouts



```
<ul>
...
</ul>
```

.../views/layouts/_navpublic.blade.php

```
<html>
...
@include('layouts/_navpublic')
...
@yield('content')
...
</html>
```

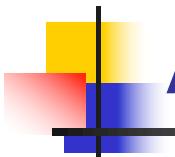
.../views/layouts/public.blade.php

```
@extends('layouts.public')
@section('content')
...
@endsection
```

.../views/catalog.blade.php

Laravel

22



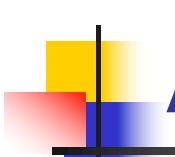
Artisan

E' una **shell** a linea di comando **parte integrante del framework**

- Consente di **attivare comandi** (codificati come script PHP) che supportano la fase di sviluppo dell'applicazione
- Si usa da **terminale** (Linux/OSX) o da **prompt dei comandi** (Windows), posizionandosi sulla **cartella radice del progetto**
- Sintassi:

```
php artisan <comando>
```

- Esempio di comandi (non esaustivo!).
 - **Lista** di tutti i comandi disponibili
 - **list**
 - **Help** dei singoli comandi
 - **help <comando>**



Artisan – Comandi Principali

- Creazione **componenti base** dell'applicazione
 - **make:controller <controller_name>**
 - **make:model <model_name>**
- Creazione **componenti** per la gestione delle **form**
 - **make:request <form_name>**
 - **make:rule <form_validation_rule_name>**
- Creazione del **database**
 - **make:migration <migration_name>**
 - **migrate**
- Visualizzazione di tutte le **rotte** definite per l'applicazione
 - **route:list**
- Cancellazione delle **cache**
 - **cache:clear**
 - **config:clear**
 - **route:clear**
 - **view:clear**
 - **optimize:clear** (Cancella **TUTTE** le chache)

Esempio

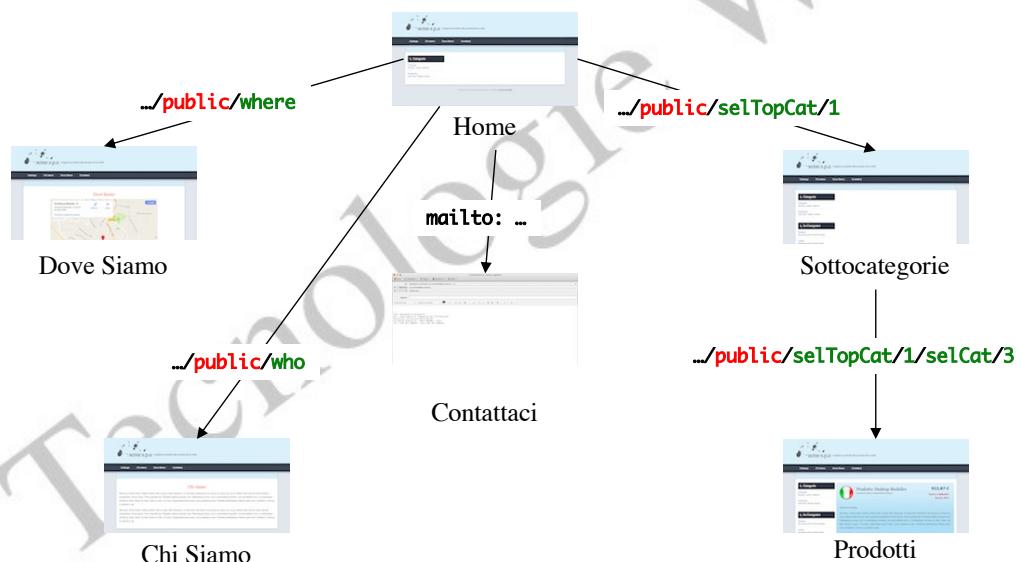
- Progetto laraProj1
 - Vetrina elettronica per catalogo prodotti consumer
 - Prodotti categorizzati su due livelli (categoria e sottocategoria)
 - Schede prodotto con informazioni a più livelli
- Obiettivi didattici: illustrare
 - L'uso dei controller
 - L'uso dei layout
 - L'uso del model (con dati non persistenti)
 - La definizione di un meccanismo di log dell'applicazione

laraProj1  **Laravel**

Laravel

25

laraProj1: Schema di link



Laravel

26

Dati Persistenti in Laravel

- Necessità di definire un **meccanismo di interfaccia** tra le **applicazioni Laravel** e le **sorgenti di dati persistenti (DB)**, per lo scambio di dati con il DBMS.



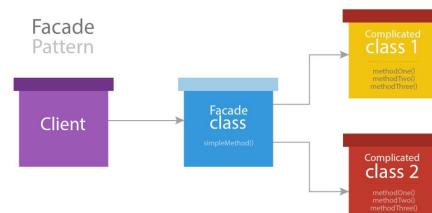
- Due livelli** di interfaccia:
 - Connessione dell'applicazione PHP al DBMS: **PDO**
 - Configurata in `config/database.php` e `.env`
 - Accesso alle funzioni del DBMS per la **manipolazione** dei dati del DB e **mappatura** degli stessi in strutture dati PHP:
 - Raw SQL
 - Query Builder
 - Eloquent ORM

Laravel

27

Laravel DB - Raw SQL

- Modalità di interfacciamento che consente di accedere al DB direttamente **tramite comandi SQL**
- Le operazioni di estrazione dati **producono i risultati** strutturati come **array** di **oggetti PHP** (istanze della classe `stdClass`)
- Metodi di accesso al DBMS **definiti** tramite la **FACADE DB**
- FACADE: design pattern** che definisce un oggetto che funge da **interfaccia semplice (proxy)** a oggetti/codice complessi
 - In Laravel le **FACADE implementano** un'**interfaccia statica** alle classi dei **service container** dell'applicazione



Laravel

28

Laravel DB - Raw SQL

Esempi d'uso della FACADE DB

- Estrazione

```
$users = DB::select('select * from users where active = ?', [1]);
```

- Inserimento

```
DB::insert('insert into users (id, name) values (?, ?)', [1, 'Dayle']);
```

- Modifica

```
$rows = DB::update('update users set votes = 1 where name = ?', ['John']);
```

- Cancellazione

```
$rows = DB::delete('delete from users');
```

- Istruzione SQL generica

```
DB::statement('drop table users');
```

- Gestione delle transazioni

```
DB::beginTransaction(), DB::rollback(), DB::commit()
```

Laravel

29

Laravel DB – Query Builder

- Modalità di interfacciamento che consente di accedere alle singole tabelle del DB mappandole in un oggetto (**Query Builder**) con metodi predefiniti per la costruzione la query
- Le **query** definite con questo metodo producono i risultati strutturati come **collection** di oggetti PHP
- Definizione dell'oggetto **Query Builder** tramite il metodo **table()** della **FACADE DB**

```
DB::table('<nome_tabella>')
```

Laravel

30

Laravel DB – Query Builder

Esempi d'uso del Query Builder

- Estrazione

```
$users = DB::table('users')->get();  
  
$user = DB::table('users')->where('name', 'John')->first();  
  
$email = DB::table('users')->where('id', 3)->value('email');  
  
$users = DB::table('users')->count();
```

- Selezione di attributi

```
$usersNames = DB::table('users')->select('name')->get();;
```

- Join

```
$users = DB::table('users')  
    ->join('contacts', 'users.id', '=', 'contacts.user_id')  
    ->join('orders', 'users.id', '=', 'orders.user_id')  
    ->select('users.*', 'contacts.phone', 'orders.price')  
    ->get();
```

Laravel

31

Laravel DB – Query Builder

- Operatori logici

```
$users = DB::table('users')  
    ->where('votes', '>', 100)  
    ->orWhere('name', 'John')  
    ->get();
```

- Query complesse

```
$users = DB::table('users')  
    ->where('name', '=', 'John')  
    ->where(function ($query) {  
        $query->where('votes', '>', 100)  
            ->orWhere('title', '=', 'Admin');  
    })  
    ->get();
```

- Corrisponde alla query SQL

```
select * from users where name = 'John' and (votes > 100 or title = 'Admin')
```

Laravel

32

Laravel DB – Eloquent ORM

- Definizione di un **ORM** per l'applicazione
 - “Object-relational mapping (ORM) in computer software is a programming technique for converting data between incompatible type systems in object-oriented programming languages. This creates, in effect, a ‘virtual object database’ that can be used from within the programming language”



- Caratteristiche:
 - Scalabilità
 - Flessibilità (posso cambiare DataSource in modo trasparente)

Laravel DB – Eloquent ORM

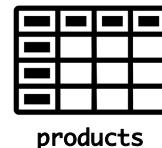
Eloquent è il componente di Laravel che implementa l'ORM utilizzando il design pattern **ActiveRecord**

- Ad ogni **tabella** del DB permette di associare un **modello specifico** (espresso come **classe PHP**) che consente di manipolare (**estrarre, inserire, modificare, cancellare, ...**) dati dalla tabella attivando i suoi **metodi predefiniti o specifici**.
- Tutti i **modelli derivano** dalla **classe predefinta Illuminate\Database\Eloquent\Model**
- È, di fatto, un **Query Builder** più potente ed **espressivo**:
 - Mantiene tutte le **funzionalità** del **Query Builder**
 - **Maschera la tabella** che il modello mappa **trasformandola** in un **oggetto PHP** che **perde i riferimenti formali alla sorgente**
 - Aggiunge **costrutti sintatticamente più 'chiari'** che consentono la codifica di istruzioni **più 'espressive'**

Laravel DB – Eloquent ORM

- Eloquent introduce una serie di convenzioni che rendono l'implementazione del modello più semplice:
 - Il nome della classe che implementa il modello viene associato automaticamente alla tabella del DB con lo stesso nome plurale in caratteri minuscoli
- Considera l'attributo id della tabella come chiave primaria a valori interi autoincrementanti
- Assume come presenti nella tabella gli attributi created_at e updated_at per la gestione del timestamp
- I valori convenzionali possono essere modificati:
 - \$table, \$primaryKey, \$incrementing, \$keyType, \$timestamp

```
<?php  
namespace App\Models\Products;  
use Illuminate\Database\Eloquent\Model;  
class Product extends Model { ... }
```



Laravel DB – Eloquent ORM

Esempi d'uso di Eloquent

- Estrazione di tuple

```
$product = Product::all();
```

- Inserimento di tuple

```
<?php  
use App\Models\Products\Product;  
class AdminController extends Controller {  
    public function storeProduct(Request $request) {  
        $product = new Product;  
        $product->fill($request->all());  
        $product->save();  
    }  
}
```

- Modifica di tuple

```
$product = Product::find(1);  
$product->name = 'NetBook Modello2';  
$product->save();
```

Laravel DB – Eloquent ORM

- Cancellazione di tuple

```
$product = Product::find(1);
$product->delete();
```

- Relazione One-To-One

```
<?php
namespace App\Models\Resources;
use Illuminate\Database\Eloquent\Model;
class Product extends Model {
    public function prodCat() {
        return $this->hasOne(Category::class, 'catId', 'catId');
    }
}
```

laraProj2_DB category

- catId : bigint(20) unsigned
- name : varchar(25)
- parId : int(11)
- desc : varchar(255)

laraProj2_DB product

- prodId : bigint(20) unsigned
- name : varchar(25)
- catId : bigint(20) unsigned
- descShort : varchar(30)
- descLong : varchar(2500)
- price : double(8,2)
- discountPerc : int(11)
- discounted : tinyint(4)
- Image : text

- Confronto Query Builder - Eloquent

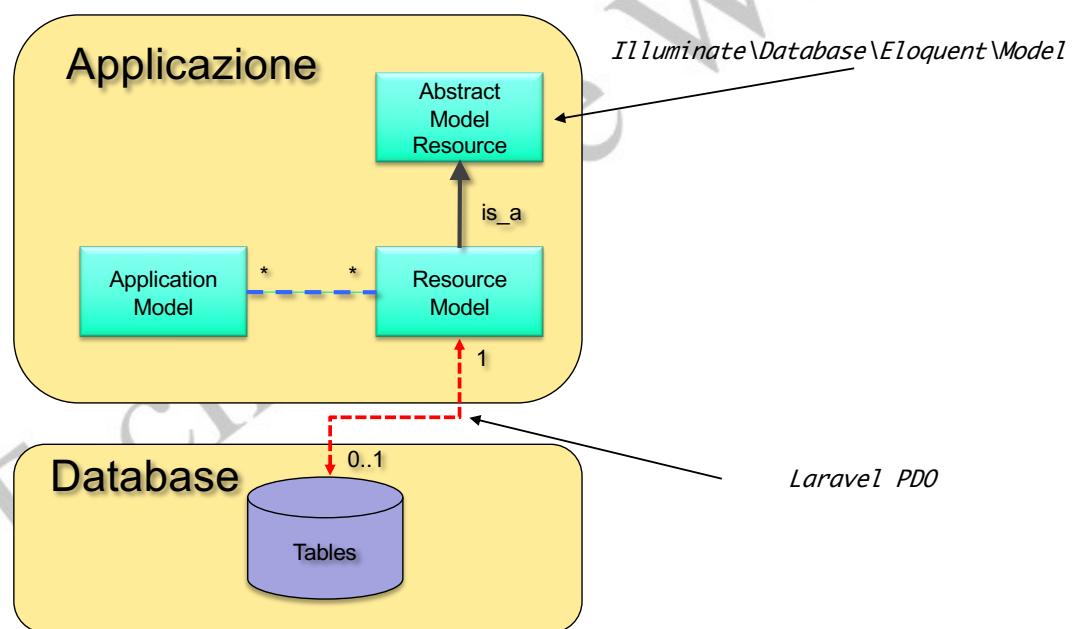
```
$product = DB::table('product')->get();
$product = DB::table('product')->where('id',$id)->first();
DB::table('product')->where('id',$id)->delete();
```

```
$product = Product::all();
$product = Product::find($id);
Product::delete($id);
```

Laravel

37

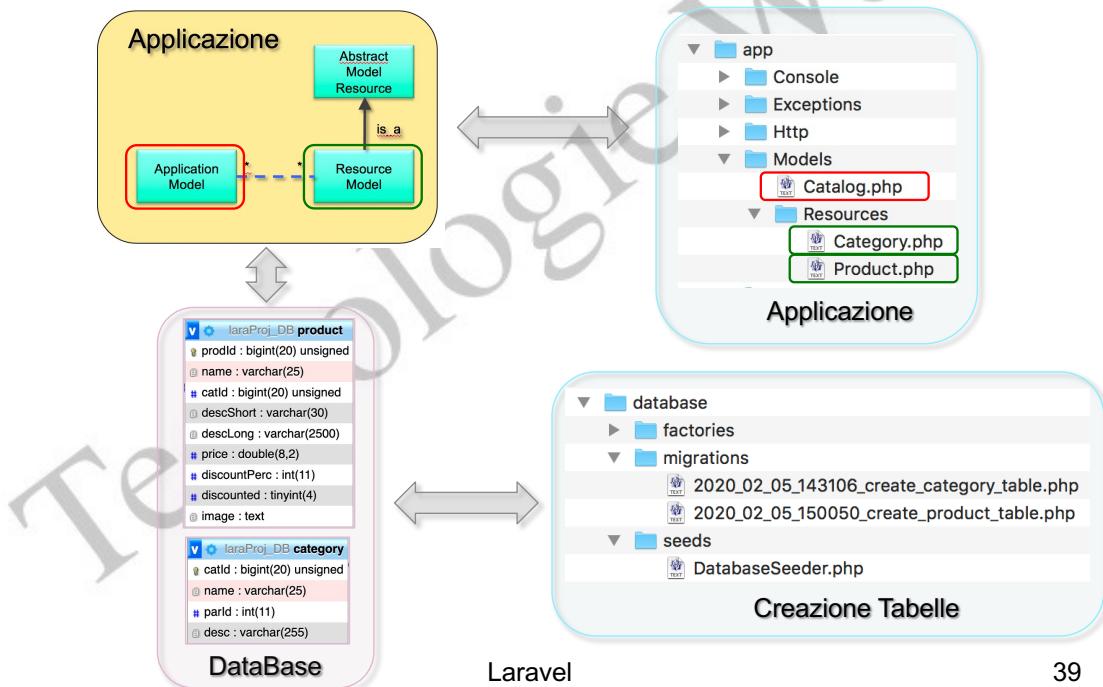
Model dell'Applicazione



Laravel

38

Implementazione



39

Creazione della Base di Dati

Laravel consente la **gestione dell'evoluzione** della **struttura della base di dati** e dei suoi **contenuti** durante lo sviluppo di un'applicazione attraverso **Migrations** e **Seeding**

- **Migrations:** meccanismo di definizione della **struttura** del DB (tabelle e relazioni tra di esse) **che consente il versioning**
 - Basato sulla definizione di **classi** che estendono la classe Laravel *Illuminate\Database\Migrations\Migration*
 - Le classi consentono di definire lo **schema delle tabelle** utilizzando la **FACADE Schema** nei due metodi pubblici: *up()* e *down()*
 - Le classi **possono essere create** con il comando *artisan*
`php artisan make:migration <nome_migrazione>`
 - I file che definiscono le classi sono memorizzati nella directory *database/migrations*
 - Comandi *artisan* per gestire l'aggiornamento del DB:
 - Esecuzione migrations: `php artisan migrate`
 - Cancellazione tabelle ed esecuzione migrations: `php artisan migrate:fresh`

Laravel

40

Creazione della Base di Dati

- **Seeding:** processo di popolamento delle **tabelle** del DB con **valori** utili per lo **sviluppo** ed il **test** dell'applicazione
 - Basato sulla definizione di **classi** che **estendono** la classe Laravel *Illuminate\Database\Seeder*
 - Ogni classe contiene un unico metodo pubblico, *run()*, che usa il **Query Builder** di Laravel per inserire valori nelle tabelle
 - Un Seeder **viene creato** con il comando *artisan*
`php artisan make:seeder <nome_seeder>`
 - I **file** che definiscono i seeder sono **memorizzati** nella directory *database/seeds*
 - Comandi *artisan* per **eseguire** i seeder:
 - Seeder **di default** (*DatabaseSeeder*): `php artisan db:seed`
 - Seeder **generico**: `php artisan db:seed --seeder=<nome_seeder>`

Laravel

41

Paginazione dei dati da DB

Laravel **consente di organizzare in pagine** il **contenuto** dei dati estratti tramite query dal DB attraverso il componente **paginator** integrato in **Query Builder** e in **Eloquent**

- La paginazione si attiva con l'uso del metodo *paginate()* applicato al risultato di una query

```
$products = DB::table('product')->paginate(15);  
$products = Product::paginate(15);
```

- *paginate()* **incapsula i dati** in un **oggetto** nel quale **vengono definiti** metodi utili per la gestione della paginazione **a livello di vista**
 - `lastPage()`, `currentPage()`, `firstItem()`, `lastItem()`, `total()`, `hasMorePages()`, `url($page)`, ...
- Il recupero della pagina richiesta dal client all'applicazione si basa sul parametro **page=<numero_pagina>** aggiunto in coda all'**url** della richiesta, che viene gestito **in modo trasparente** da Laravel
Request URL della seconda pagina: `.../public/selTopCat/1/selCat/4?page=2`

Laravel

42

Esempio

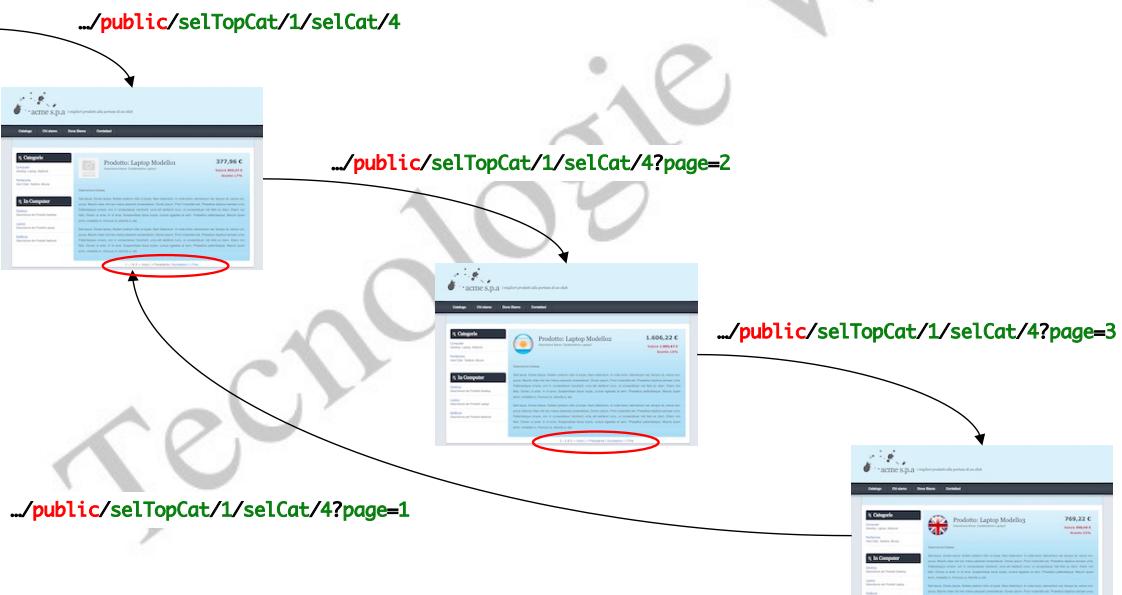
- Progetto laraProj2
 - Estensione di laraProj1
 - Implementazione del Model con dati persistenti
 - Paginazione delle schede prodotto
- Obiettivi didattici: illustrare
 - La costruzione del Model
 - *Illuminate\Database\Eloquent\Model*
 - Migrations e seeding
 - Il componente per la paginazione del framework

laraProj2  Laravel

Laravel

43

LaraProj2: paginazione

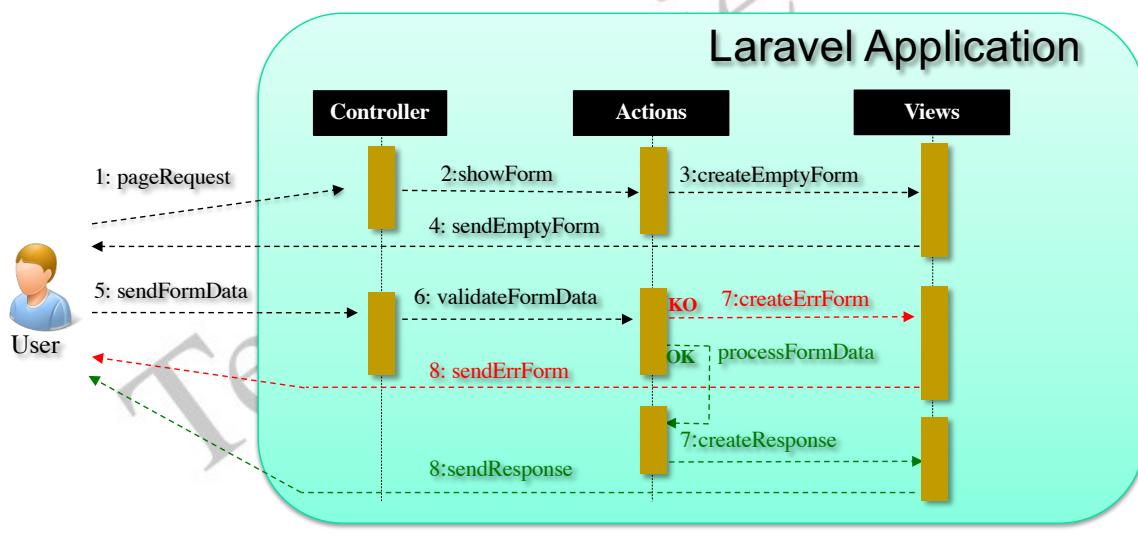


Laravel

44

Gestione Form HTML

Schema generale di **interazione** tra l'utente e l'applicazione Web basata sul **FORM HTML**



45

Gestione Form HTML

- **Premessa:** dati contenuti in una form **vanno sempre validati prima di essere usati** dall'applicazione
 - In caso di **errore**, la validazione (lato server), deve indicare **all'utente** la natura dello stesso e **consentire la sua correzione**, riproponendo la **FORM** con i dati precedentemente inseriti
 - In questo caso, l'**interazione C/S prosegue** con le stesse modalità **sino a che non vengono trasmessi dati corretti** nella FORM
 - In presenza di dati **corretti**, la logica dell'applicazione **attiva la procedura** per la loro utilizzazione
- **Conseguenza:** implementazione di un processo **complesso** lato server **per la gestione dell'interazione (ripetuta) C/S** sino alla definizione di dati corretti

Laravel fornisce **servizi a supporto** di questo processo, che rendono **semplice** la sua **implementazione**

Laravel

46

Gestione Form HTML in Laravel

Il processo di **validazione** del contenuto di una FORM può essere gestito in Laravel utilizzando la classe predefinita *FormRequest*

- Ad ogni **FORM** dell'applicazione viene associata una **sottoclasse specializzata di *FormRequest***, che esplicita i **vincoli di accesso** alla FORM (quali utenti la possono utilizzare) e le **regole di validazione** dei suoi elementi
- Se nell'**action** del controller attivata dal Router Laravel per gestire la richiesta HTTP generata dal **submit** della FORM il **parametro in input** (in cui vengono passati i dati della form) è **tipizzato (Type Hinted)** con la **sottoclasse** di *FormRequest* definita per la FORM:
 - prima di eseguire il codice dell' action, **si attiva** automaticamente **la validazione dei dati** rispetto ai vincoli definiti
 - in presenza di **errori**, il client viene rediretto all' action di visualizzazione della FORM, **alla cui vista vengono resi disponibili** i **dati inseriti dall'utente** ed i **messaggi d' errore**

Gestione Form HTML in Laravel

- Nella classe specializzata di *FormRequest* possono essere specificate **regole di validazione**:
 - **predefinite**
 - `required`, `numeric`, `integer`, `min`, `max`, `image`, `url`, `email`, `regex`, ...
 - **definite ad hoc**, estendendo la classe predefinta *Rule*
- Le regole di validazione **vengono applicate** utilizzando il **trait *ValidateRequest***, associato nel framework ad ogni controller
 - **trait**: nel linguaggio PHP, **contenitori di metodi e proprietà**, simili a classi, che **possono essere associati a qualunque classe** senza utilizzare il legame di **ereditarietà**.
- Ogni FORM **deve contenere** un campo 'hidden' con un **token** generato automaticamente per **proteggere l'applicazione** da attacchi **CSRF** (Cross-Site Request Forgery)
 - La **presenza** del valore corretto del token CSRF all'interno dei dati della FORM inviati da client viene **verificata** dal middleware *VerifyCsrfToken*

Esempio

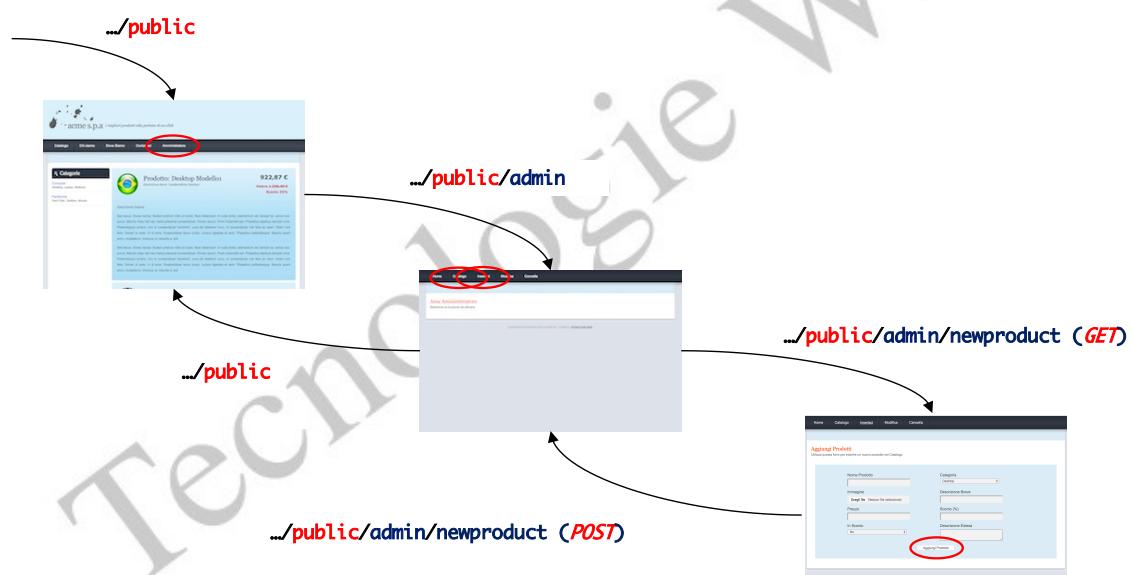
- Progetto laraProj3
 - Estensione di laraProj2
 - Implementazione del Controller Amministratore
 - Funzionalità di inserimento nuovi prodotti nel catalogo
- Obiettivi didattici: illustrare
 - L'uso di layout diversi nella stessa applicazione
 - La gestione delle Form HTML in Laravel
 - La definizione di Form e l'acquisizione dei dati
 - Il processo di validazione tramite trait PHP

laraProj3  Laravel

Laravel

49

laraProj3: routing



Laravel

50

Package Laravel

I **package** (o **Plugin**) sono lo strumento primario per **estendere le funzionalità** di Laravel

- Sono **componenti software integrabili** nel framework e sviluppate per fornire **nuove funzionalità** o **estendere le esistenti**.
- Possono essere di **due tipi**:
 - **stand alone**: progettati per l'uso con **qualsiasi package PHP**
 - **specifici**: nati per **integrarsi** con il workflow e le componenti di un'applicazione **Laravel** (routing, controller, viste,...)
- Vengono integrati in Laravel con tutte le loro dipendenze utilizzando **Composer**
- I più diffusi forniscono supporto per l'**autenticazione**, l'**auditing**, la **gestione delle e-mail**, la **gestione delle notifiche**, l'implementazione di **funzionalità di e-commerce**, ...
 - Un **sito** di riferimento: <https://packalyst.com>

Laravel Collective

Laravel Collective è un package free specifico per Laravel che integra nel framework le seguenti **componenti** principali:

- **HTML**: **generatore**, a livello di vista, di FORM e componenti **HTML complesse tramite costrutti sintattici semplici**.
 - Apertura di una FORM

```
{!! Form::open(['url' => 'foo/bar']) !!}
```
 - Generazione di un'ancora

```
echo link_to('foo/bar', $title = null, $attributes = []);
```
- **Remote**: componente per la **gestione delle connessioni SSH** all'interno dell'applicazione
 - Esecuzione di un comando SSH sul server locale

```
SSH::run(['cd /var/www']);
```
- **Annotation**: componente per **l'inserimento** di **annotazioni** nel codice
 - Annotazione per registrare una rotta GET associata ad un'azione

```
@Get("/")
```

Laravel Collective

Principali costrutti sintattici di **Laravel Collective** per l'inserimento nei **Template** di elementi di una FORM:

- Tag di apertura e chiusura di **FORM**

```
 {{ Form::open(array('route' => 'newproduct.store', 'id' => 'addproduct')) }}  
 {{ Form::close() }}
```

- Elemento di input di tipo **TEXT** e relativa **LABEL**

```
 {{ Form::label('name', 'Nome Prodotto', ['class' => 'label-input']) }}  
 {{ Form::text('name', '', ['class' => 'input', 'id' => 'name']) }}
```

- Elemento di input di tipo **SELECT**

```
 {{ Form::select('catId', $cats, '', ['class' => 'input', 'id' => 'catId']) }}
```

- Elemento di input di tipo **TEXTAREA**

```
 {{ Form::textarea('descLong', '', ['class' => 'input', 'id' => 'descLong',  
 'rows' => 2]) }}
```

- Bottone **SUBMIT**

```
 {{ Form::submit('Aggiungi Prodotto', ['class' => 'form-btn1']) }}
```

Laravel

53

Esempio

- Progetto **laraProj4**
 - **Estensione** di **laraProj3**
 - **Definizione** FORM HTML usando **Laravel Collective**
 - **Installazione**: dalla cartella di progetto
composer require laravelcollective/html
- Obiettivi didattici: illustrare
 - Come **estendere** le funzionalità del framework Laravel attraverso laggiunta di **package**
 - Come **utilizzate** il package **Laravel Collective** per gestire le **FORM HTML**
 - Come **localizzare** i messaggi d'errore delle FORM in **italiano**

Autenticazione

- È il **processo** che verifica l'identità di un utente, che si qualifica attraverso le sue **credenziali** (di solito **username** e **password**)
- Un'applicazione **Laravel** può implementare il processo di autenticazione attraverso l'aggiunta del package **Breeze**, uno starter kit che definisce una **serie di componenti già configurate** (**authentication scaffolding**) per la gestione:
 - dei **driver di autenticazione**, cioè le **sorgenti** di informazioni per l'autenticazione
 - nella configurazione base, il **Model** dell'utente definito dalla tabella **users**
 - del **processo di autenticazione**, attraverso una serie di **controller** associati alle diverse fasi
 - **RegisteredUserController**, **AuthenticatedSessionController**, **NewPasswordController**, **PasswordResetLinkController**, ...
 - della **persistenza dell'identità** dell'utente autorizzato utilizzando, di default, le **sессии PHP**

Laravel

55

Autenticazione

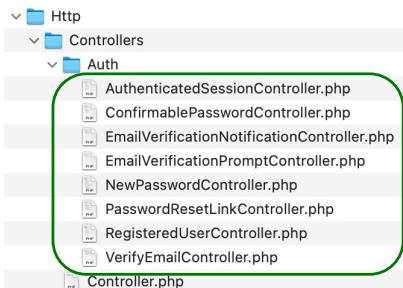
- **Breeze** definisce anche le **viste** e le **rotte** associate ai vari processi che gestisce
 - Le **viste** sono definite nella cartella **resources/views/auth**
 - **register.blade.php**, **rlogin.blade.php**, **reset-password.blade.php**, ...
 - Le **rotte** sono definite nel file **routes/auth.php** e richiamate in **routes/web.php**
- Una volta implementato il processo di autenticazione, attraverso la **FACADE AUTH** è possibile accedere alle **informazioni** legate all'utente attualmente autenticato memorizzate nella **sessione PHP** e attivare il **processo di logout**
 - **Auth::user()**, **Auth::id()**, **Auth::check()**, **Auth::logout()**

Laravel

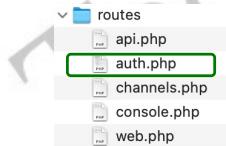
56

Autenticazione

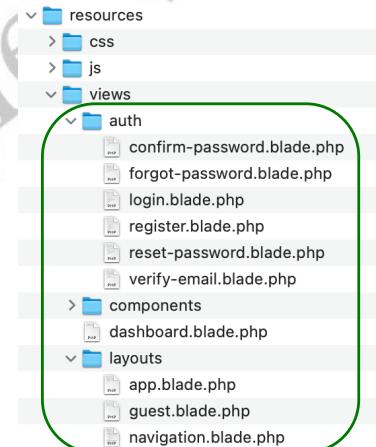
- ## ■ Aggiunge controllers



- ### ■ Aggiunge rotte



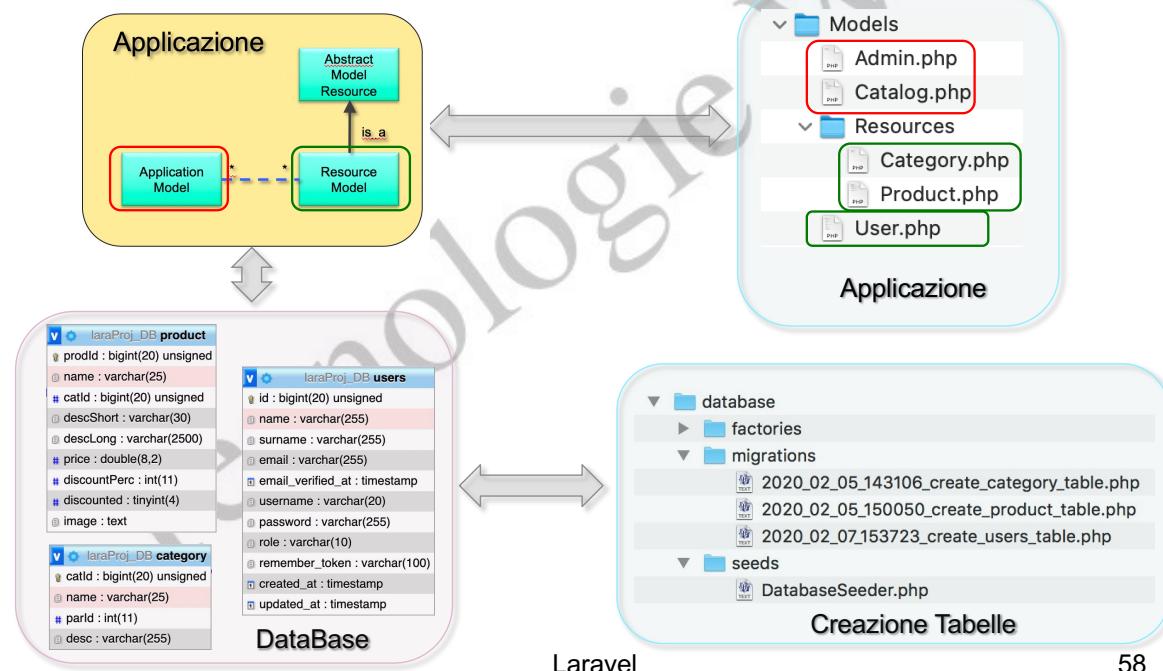
- ## ■ Aggiunge viste/layouts



Laravel

57

Implementazione Model



Laravel

58

Autorizzazione

- È l'attività che verifica se **qualcuno** (un utente, un processo, ...) ha il permesso di fare **qualcosa** (attivare, accedere, ...) su una **risorsa** (un'azione, un dato, ...).
- In Laravel ci sono **due meccanismi** per gestire l'autorizzazione: i **Gates** e le **Policies**
 - **Gates**: funzioni che **definiscono le condizioni** per cui l'utente autenticato può eseguire una certa azione
 - **Policies**: classi che **organizzano le logiche** di autorizzazione relativamente ad un **modello** o una **risorsa** (**non le approfondiamo**)
 - **Non sono alternativi**: possono essere usati **entrambi** nella stessa App
- Come e dove implementare un'autorizzazione nel pattern **MVC**?
 - Su quali **risorse**? I Controllers, le Actions, i Models?
 - A che **livello**? Sulle rotte o nelle singole risorse?

Laravel

59

Gates

- Definizione di una **regola** (in *App\Providers\AuthServiceProvider*)
 - `Gate::define('isAdmin', function ($user) { return $user->hasRole('admin');});`
 - `Gate::define('can-edit', function ($user, $product) { return $user->hasRole(['admin', 'staff']);});`
- Uso di una **regola** di autorizzazione
 - `if (Gate::allows('isAdmin')) { // l'utente ha ruolo 'admin' }`
 - `if (Gate::denies('can-edit', $product)) { // l'utente non può editare il prodotto }`

Laravel

60

Middleware Authorize

È possibile utilizzare le regole definite nei Gates o nei Providers utilizzando il middleware `Illuminate\Auth\Middleware\Authorize`

- In questo modo il meccanismo di autorizzazione può entrare in azione prima dell'attivazione di una **rotta** o di un'**azione di un controller**
- L'attivazione del middleware Authorize avviene attraverso l'uso della chiave `can` definita nella classe `App\Http\Kernel`
 - Uso nella **rotta**
 - `Route::get('/user', [UserController::class, 'index'])->middleware('can:isUser');`
 - Uso nel **costruttore del controller**
 - `public function __construct() { $this->middleware('can:isAdmin'); }`

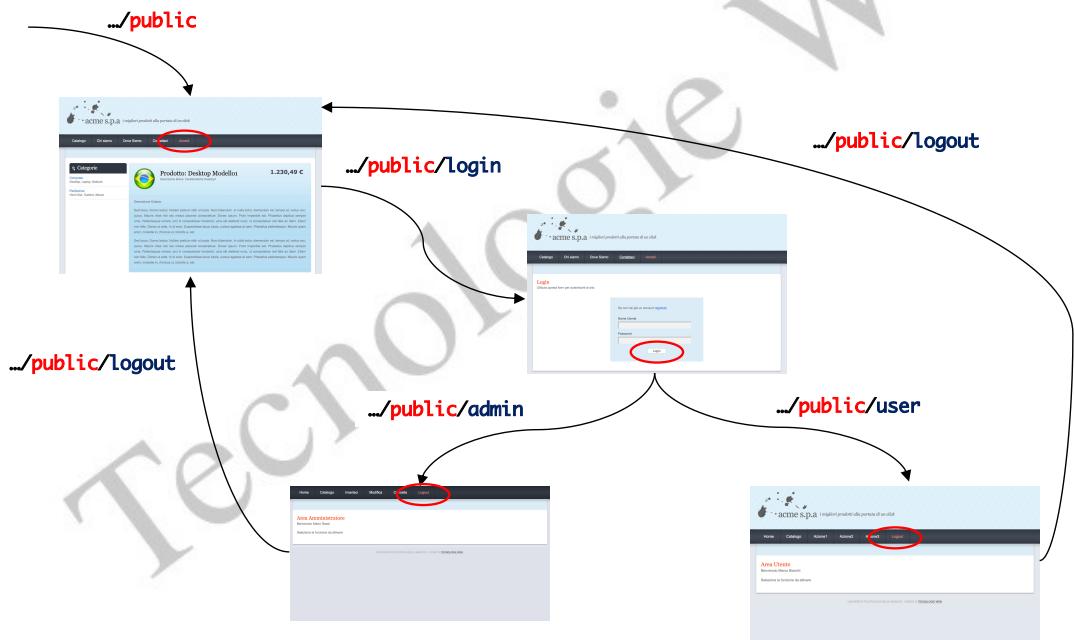
Laravel

61

Esempio

- Progetto `laraProj5`
 - Estensione di `laraProj4`
 - Implementazione del meccanismo di autenticazione
 - Implementazione delle autorizzazioni all'accesso
 - Aggiunta di due sezioni del sito: Admin e User
- Obiettivi didattici: illustrare
 - Il package `breeze`
 - L'`authentication scaffolding`
 - L'uso dei `Gates`
 - L'uso del middleware `Authorize`

Autenticazione: routing



63

Registrazione: routing



64