Corso Laravel

Lezione 1

Introduzione a Laravel, ambiente di lavoro e sviluppo di un’applicazione

Introduzione al corso

Questo corso ha come obbiettivo quello di dare tutti gli elementi per la realizzazione di applicazioni web con l’utilizzo di PHP e Laravel come framework applicativo.
Si parte percorrendo una panoramica su quelli che sono i componenti essenziali a definire l’architettura LAMP / LEMP Stack

Si analizzeranno gli aspetti fondamentali di Laravel, le cartelle, e la configurazione iniziale; verrà mostrata l’anatomia di un’applicazione web.

Si vedrà come utilizzare con profitto Blade come motore di template e di come combinarlo con Javascript e CSS; la gestione dei form e la validazione dei dati.

Si vedrà inoltre l’utilizzo di Eloquent come ORM predefinito di Laravel, la costruzione di un Web Service REST, l’integrazione con strumenti di sviluppo come Glup/Grunt.

Introduzione al corso

Ecco nel dettaglio come sarà strutturato il programma:

Lezione 1

- Introduzione a GIT
- Composer, Namespace e meccanismi di autoload
- Basi di MVC
- Installazione di Laravel e ambiente di sviluppo con Homestead
- Artisan e la riga di comando
- Introduzione al routing
- Controllers e Views
- Primo deploy su Cloud (RedHat OpenShift o Amazon AWS)

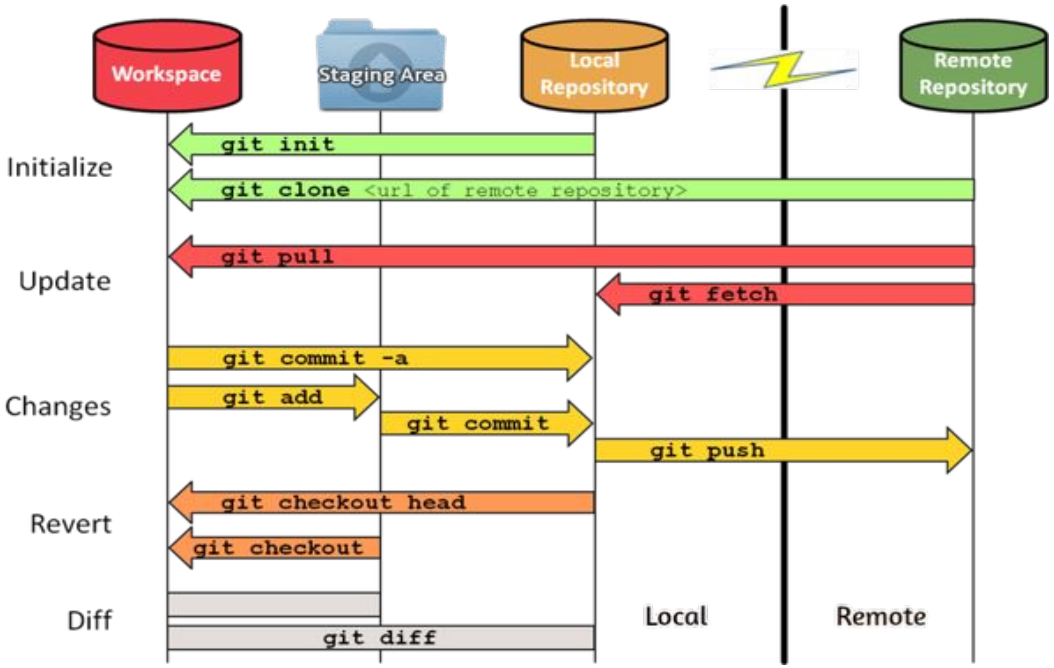
Introduzione a GIT

Partiamo da **git**, per impostare un corretto workflow di sviluppo e di versioning del codice.
Non che sia necessariamente legato a Laravel, ma sicuramente ci agevola nel gestire le evoluzioni dell'attività di codifica, nel poter lavorare in team o anche da soli, nel gestire le feature/branch che andremo a creare.

Comando	Descrizione
`git init`	inizializza un repository
`git add <nomefile>`	aggiunge uno o più file al repository
`git commit -m "descrizione"`	applica i file modificati sul repo locale
`git status`	mostra quali file sono stati modificati e quali sono da applicare al repo locale
`git push origin master`	carica/invia i file del commit al repo remoto
`git pull`	recupera il codice dal repo remoto, effettuando il merge con il codice locale

Introduzione a GIT

Quindi il tipico workflow con git è simile al seguente:



Introduzione a GIT

Quando creiamo un progetto o cloniamo un progetto già esistente, di fatto creiamo nella nostra macchina un repositori locale, che contiene il "workspace" del progetto.

Otteniamo anche un branch principale che è il "master".

Quando modifichiamo i file o ne creiamo di nuovi, per includerli nel repository dobbiamo usare "`git add <nomefile>`" o direttamente "`git add .`"

Successivamente quando vogliamo "salvare" le nostre modifiche sul repository locale, creando una version e andremmo ad eseguire una commit:

```
commit –m "descrizione della versione"
```

Quando siamo pronti per caricare le nostre mdifiche sul repository remoto andiamo ad eseguire `git push origin master` o `git push origin <nome del branch>`

Introduzione a GIT

Inoltre durante l'attività di sviluppo, possiamo creare dei branch, delle "ramificazioni" del nostro progetto, che vivono di vita propria e che successivamente possiamo unire "merge" con il branch principale "master".

Ad esempio dei branch tipici possono essere:

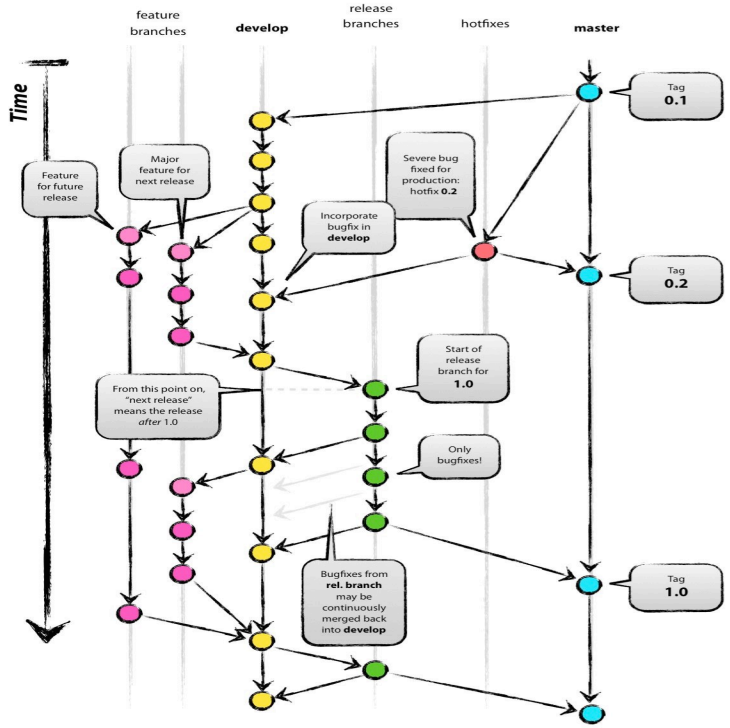
- **Branch per feature:** voglio introdurre/testare una nuova feature, creo un branch, effettuo la codifica e i test, quando pronto posso effettuare il merge nel branch master

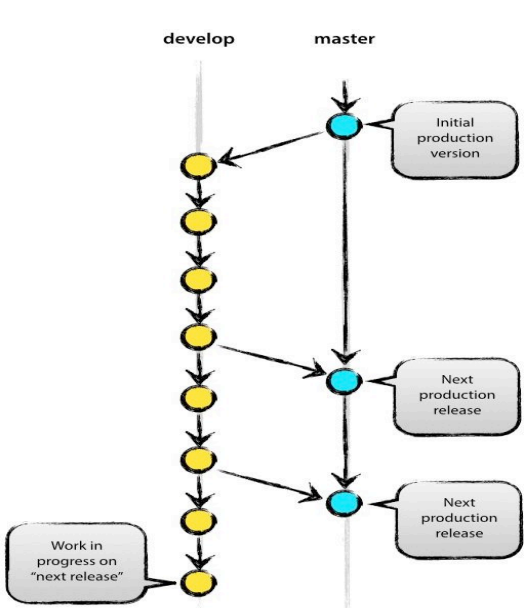
- **Branch per release**: creo dei branch in base alle release che ho in essere nel progetto, tipicamente develop, stage, production

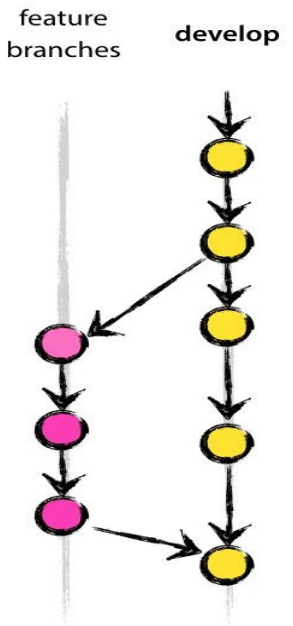
- **Branch per fix**: nella fase del ciclo di vita del software, in base ai fix che devo effettuare, creo delle branch specifiche

Introduzione a GIT

Ecco qualche rappresentazione grafica:







Introduzione a GIT

Durante il corso, svilupperemo un progetto completo, con area riservata e upload di file; verrà pubblicato in un ambiente cloud (Open Shift di RedHat)

Qui dei link di riferimento per una rapida lettura:

- http://www.html.it/articoli/git-in-pochi-passi-2/ [consigliato]
- http://stackoverflow.com/questions/292357/what-is-the-difference-between-git-pull-and-git-fetch
- http://www.yourinspirationweb.com/2013/10/16/introduzione-a-git-parte-2-qual i-sono-le-funzioni-principali-di-git/
- http://get-git.readthedocs.io/it/latest/dailygit.html
- http://nvie.com/posts/a-successful-git-branching-model/
- https://git-scm.com/book/it/v2/Git-Branching-Branching-Workflows

Composer, Namespace e meccanismi di autoload

Nella loro definizione più generale, i namespace rappresentano un modo per aggregare elementi che sono legati tra loro.

I namespace in PHP permettono di creare insiemi di classi, interfacce, funzioni, costanti legate tra loro dal comune obbiettivo di risolvere un problema.

Prima dell'introduzione dei namespace, questo meccanismo di aggregazione veniva realizzato tramite folder sul filesystem, cui i namespace stessi sono un'analogia.

Grazie ai namespace e alla possibilità di funzioni di autoloading, il nostro codice PHP è indipendente da dove sono definiti i suoi elementi all'interno del filesystem.

Composer, Namespace e meccanismi di autoload

Innanzitutto, diamo la definizione dei termini che useremo in seguito:

• Nomi non qualificati: sono identificativi senza un \ (namespace separator), come per esempio Myclass;

• Nomi qualificati: sono identificativi che contengono dei namespace separator, come per esempio MyProject\Myclass;

• Nomi completamente qualificati: sono identificativi che iniziano con un namespace separator, come per esempio \MyProject\MysubModule\Myclass;

Il concetto di namespace è ben rappresentato da un filesystem: in esso le directory vengono utilizzate per creare insiemi di file e ne definiscono il namespace.

Per esempio, se avessimo due file con lo stesso nome testo.txt in due directory **/tmp/prima** e **/tmp/seconda**, essi non potrebbero esistere all'interno della stessa directory e, per accedere a ognuno di essi al di fuori delle loro directory, dovremmo usare i loro path assoluti **/tmp/prima/testo.txt** e **/tmp/seconda/testo.txt**.

Composer, Namespace e meccanismi di autoload

In PHP i namespace sono stati progettati per risolvere due problemi nella creazione di classi e funzioni riutilizzabili:

• collisioni tra nomi di classi e funzioni all'interno del nostro codice e classi/funzioni/costanti interne a PHP o definite da librerie di terze parti;

• possibilità di creare alias o ridurre nomi molto lunghi, introdotti per evitare il problema precedente, rendendo il codice più leggibile.

I namespace possono essere dichiarati usando la parola chiave “namespace”. Un file contenente un namespace deve dichiarare il namespace prima di qualsiasi altro codice.

Nessun tipo di codice può precedere la dichiarazione di un namespace. In particolare non devono essere presenti stringhe al di sopra dell’apertura dello script PHP, altrimenti si riceverà un fatal error. Inoltre, nessun namespace deve contenere parole chiave in PHP, altrimenti sarà generato un parse error.

Composer, Namespace e meccanismi di autoload

Ecco qualche tutorial di riferimento:

http://www.sitepoint.com/how-to-use-php-namespaces-part-3-keywords-and-autoloading/
https://mattstauffer.co/blog/a-brief-introduction-to-php-namespacing

Documentazione su autoload
http://php.net/manual/en/language.oop5.autoload.php
https://getcomposer.org/

Composer, Namespace e meccanismi di autoload

Composer è uno strumento per la gestione dei pachetti e delle dipendenze in un progetto con PHP; sfrutta i meccanismi di autoloading nativi di PHP, in modo tale da caricare una classe SOLO quando viene istanziata e gestirne le eventuali dipendenze.

Composer si basa su uno strumento a riga di comando, che è possibile scaricare da https://getcomposer.org/download/ e averlo disponibile nel PATH di sistema; anche l'eseguibile "php" deve essere disponibile nel path di sistema.

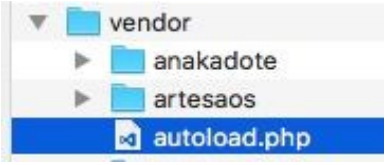
Composer, Namespace e meccanismi di autoload

Il funzionamento di composer, avviene da riga di comando èd è basato, oltre al suo eseguibile, da un file composer.json che contiene l'elenco delle librerie, il loro percorso, la loro versione, sulla quale poi generare l'autoload.

Per inserire quindi le librerie/package nel nostro progetto, abbiamo due strade:

- Inserire a mano le librerie nel file composer.json modificandolo manualmente e lanciando il comando `composer install`

- Utilizzare il comando stesso di composer che è `composer require <nome-package>`



In entrambi i casi, al termine installazione verrà creata e aggiornata la cartella "**`vendor`**" con dentro il file "**`autoload.php`**" che si occuperà quindi di caricare le classi in memoria e ottimizzandone l'uso.

Composer, Namespace e meccanismi di autoload

I package verranno scaricati dal proprio repository recuperato via remoto da https://packagist.org che a sua volta rimanderà al repo del pacchetto.

Inoltre sempre su composer.json possiamo definire delle classi “custom” (quindi al di fuori di packasgist) mediante il construtto (https://getcomposer.org/doc/articles/custom-installers.md#composer-json)

```
"autoload": {
    "psr-4": {
        ”Namespace\\Classes\": ”folder/classes/nome-file.php”
    }
}
```

Qui un buon tutorial di riferimento:
http://www.html.it/guide/composer-e-packagist-la-guida/

Introduzione a MVC

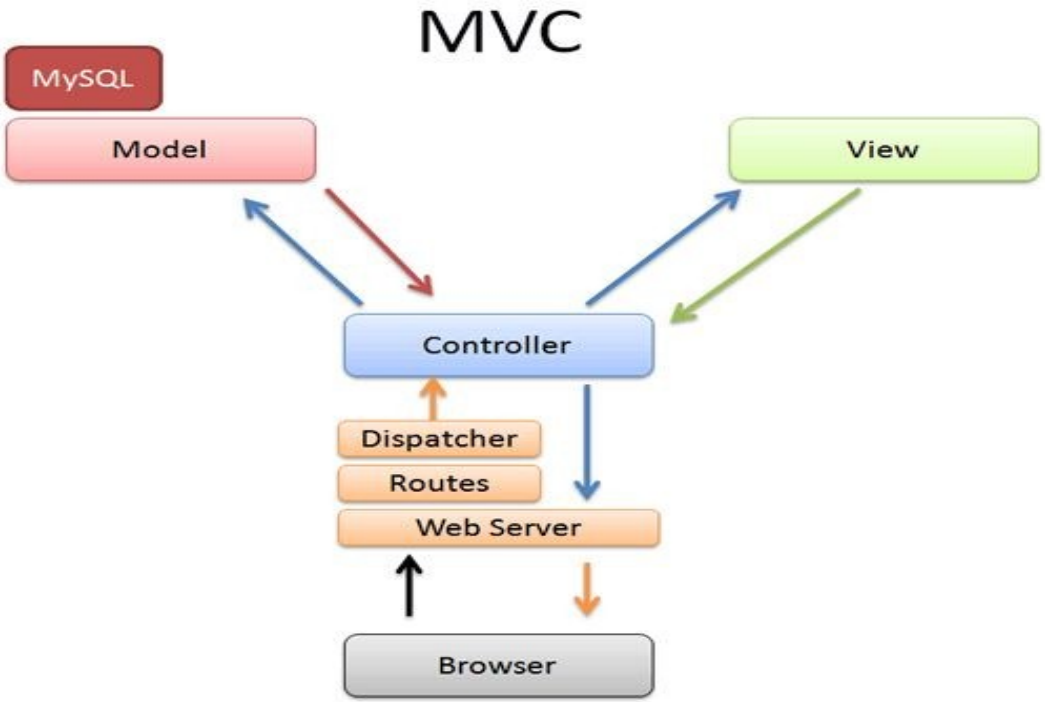
Model-View-Controller (MVC, talvolta tradotto in italiano Modello-Vista-Controllore) è un pattern architetturale molto diffuso nello sviluppo di interfacce grafiche di sistemi software object-oriented.

Originariamente impiegato dal linguaggio Smalltalk, il pattern è stato esplicitamente o implicitamente sposato da numerose tecnologie moderne, come framework basati su

- PHP (Symfony, Zend Framework, CakePHP),
- su Ruby (Ruby on Rails),
- su Python (Django, TurboGears, Pylons, Web2py),
- su Java (Swing, JSF e Struts),
- su Objective C
- o su .NET.

Introduzione a MVC

Il funzionamento di tale pattern può essere schematizzato nel seguente modo:



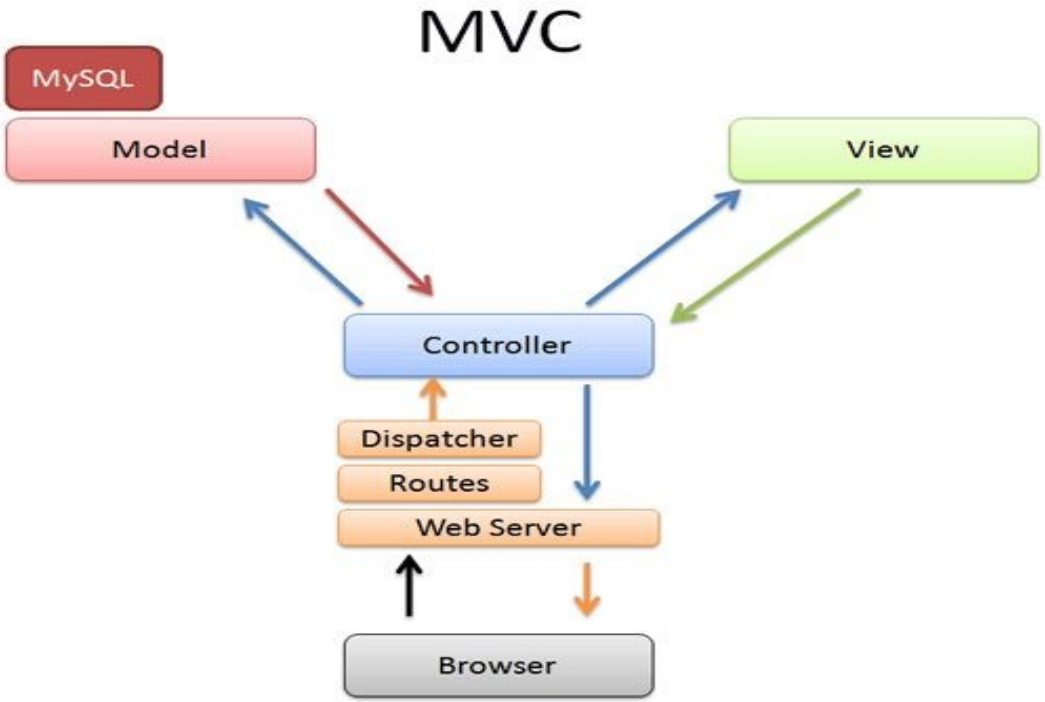
Questo modello si sposa perfettamente con il funzionamento di una web application moderna, dove vengono intercettare le azioni dell'utente.

Il nostro applicativo che interagisce con l'utente mediante il browser, si deve quindi comportare e modificare in base alle azioni dell'utente.

Ci si riferisce agli Use Case

Introduzione a MVC

Il funzionamento di tale pattern può essere schematizzato nel seguente modo:



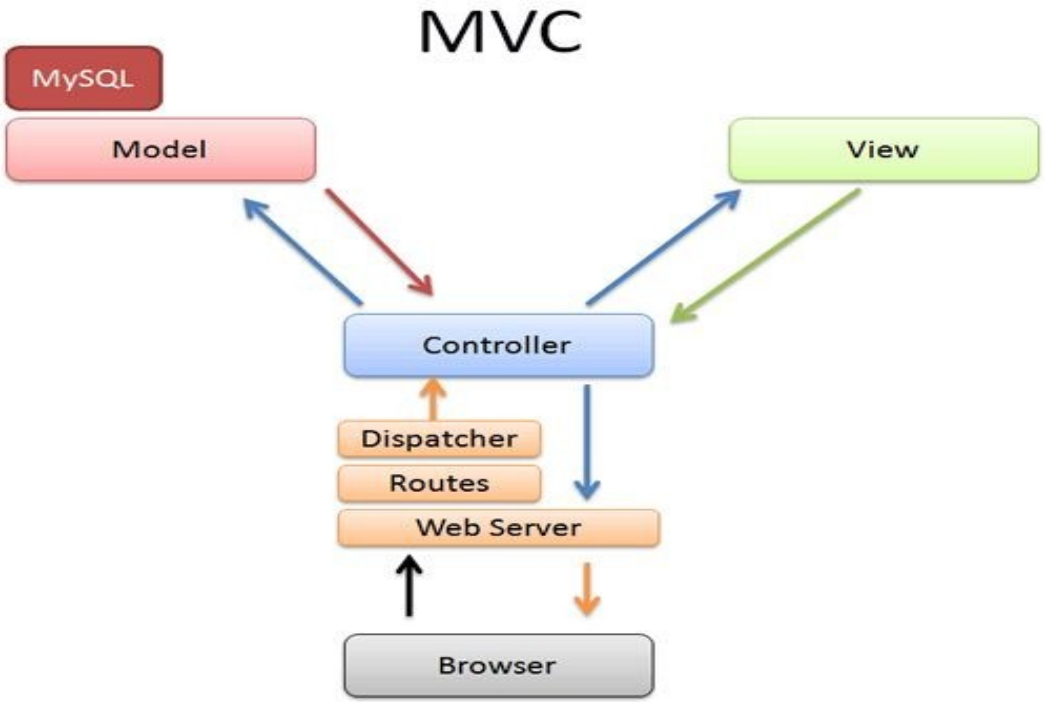
Il browser invia al WebServer l'azione che ha compiuto l'utente (GET, POST).

Questo è il punto di ingresso nel nostro applicativo, che deve intercettare la richiesta dell'utente (Routes) e valutarla (Dispatcher) in modo da richiamare il Controller adeguato, con i giusti parametri.

Tra gli elementi che entrano in gioco c'è lato WebServer il modulo di ReWrite delle URL.

Introduzione a MVC

Il funzionamento di tale pattern può essere schematizzato nel seguente modo:



Il Controller gestisce quindi il flusso di funzionamento del nostro applicativo.

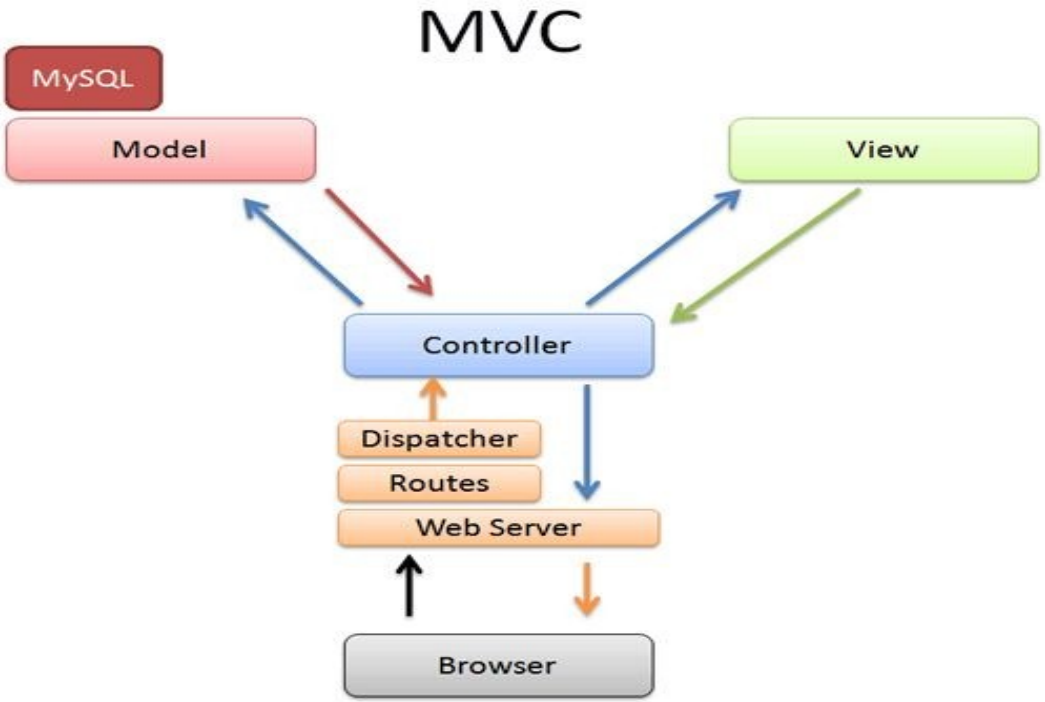
Se necessario dialoga con il Model, ne cattura il risultato, o gli invia dei dati.

Il Controller dialoga con le View per "disegnare" l'output da inviare al WebSever.

Il WebServer invierà l'output al Browser che ne interpreterà il contenuto.

Introduzione a MVC

Il funzionamento di tale pattern può essere schematizzato nel seguente modo:

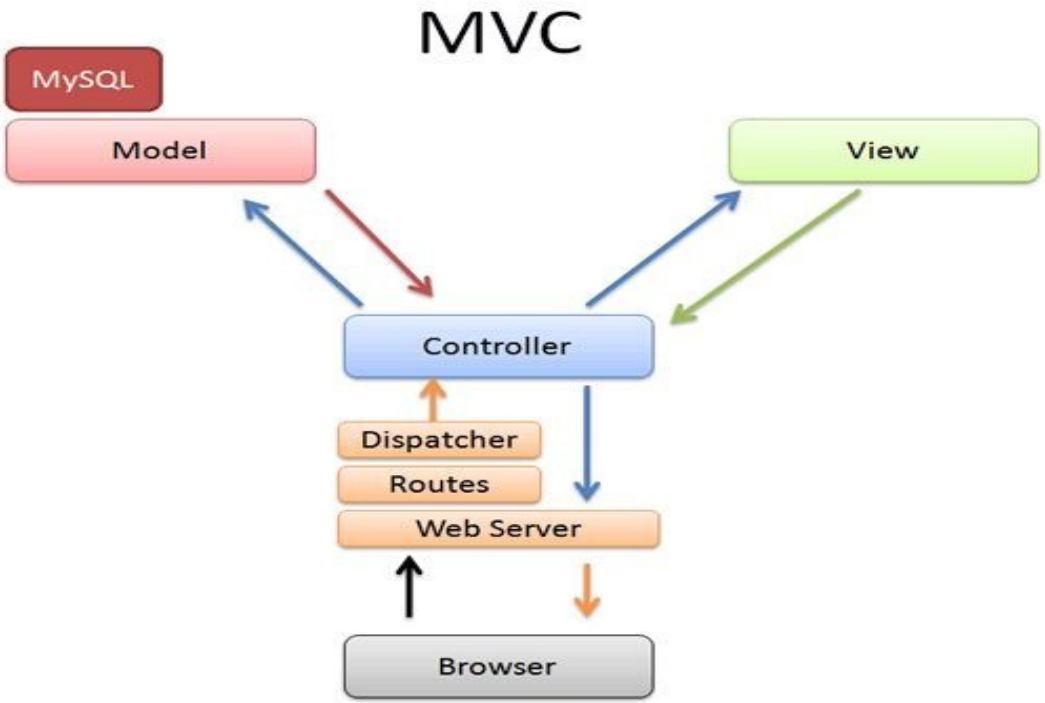


Il Model è il componente designato a comunicare con i dati, riceve le richieste da parte del Controller e restituisce dei dati.

Il Model quindi non si deve occupare di disegnare qualcosa e non ha nessun OUTPUT ma ha un dialogo con il controller basato su dati di ingresso e di uscita.

Introduzione a MVC

Il funzionamento di tale pattern può essere schematizzato nel seguente modo:



Il Model quindi parlerà con i DB, ed userà le sue classi di "business" per i dati.

In questo settore si collocano gli ORM, che consentono di effettuare tale dialogo con i Database mediante una logica "ad oggetti", in modo da rendere trasparente ai Controller l'accesso ai dati, sia in scrittura che in lettura.

Introduzione a Laravel

Laravel è un framework full-stack per la creazione di applicazioni Web basate sull'architettura Apache/Nginx, DB (MySQL, Oracle, PostgreSQL) e PHP.

Ad oggi per gli sviluppatori PHP c'è a disposizione un vasto panorama di framework, dalle applicazioni più semplici a quelle di classe enterprise e tra questi cito:

- Symfony (http://symfony.com)
- Zend Framework (http://framework.zend.com)
- Yii (http://www.yiiframework.com)
- Code Igniter (http://www.codeigniter.com)

Inoltre vi sono a disposizione die microframework che gestiscono il routing delle URL, alla quale corrisponde un output HTML/XML/JSON, tra questi cito:

- FlightPHP (http://flightphp.com)
- Silex (http://silex.sensiolabs.org)
- Slim (http://www.slimframework.com)
- Lumen (http://lumen.laravel.com) derivato da Laravel

Introduzione a Laravel

Laravel è quindi un framwork MVC-Based scritto in PHP progettato per migliorare la qualità del software riducendo sia i costi di sviluppo e costi di manutenzione, migliorare l'esperienza di sviluppo delle applicazioni fornendo una sintassi chiara ed espressiva e un insieme di funzionalità e integrazioni che vi farà risparmiare ore di tempi di implementazione.

Laravel è stato progettato con la filosofia di utilizzo di "convention over configuration" (https://it.wikipedia.org/wiki/Convention_Over_Configuration).
"Convention Over Configuration è un paradigma di programmazione che prevede configurazione minima (o addirittura assente) per il programmatore che utilizza un framework che lo rispetti, obbligandolo a configurare solo gli aspetti che si differenziano dalle implementazioni standard o che non rispettano particolari convenzioni di denominazione o simili.

Questo tipo di approccio semplifica notevolmente la programmazione soprattutto agli stadi iniziali dello studio di un nuovo framework, senza necessariamente perdere flessibilità o possibilità di discostarsi dai modelli standard.”
Il risultato è un codice semplice, facile da comprendere, modulare e lineare.

Introduzione a Laravel

Laravel è uno dei pochi framework PHP che offre vera modularità del codice. Questo risultato è ottenuto attraverso una combinazione di driver e sistema di caricamento dinamico delle classi.

I driver consentono facilmente di utilizzare la cache, le sessioni, il database e funzionalità di autenticazione.
Utilizzando pacchetti (composer), saremo in grado di realizzare codice sia per il proprio riutilizzo oppure per fornire funzionalità al resto della comunità Laravel. Questo è vantaggio perché tutto ciò che può essere scritto in Laravel può essere assemblato come bundle e distribuito.

Introduzione a Laravel

Inoltre si un buon uso delle feature di PHP 5.4 tra cui:

- Utilizzo di Namespace (PSR-4 convetion)
- Interfacce
- Anonymous Function (proprio intese come nel caso di javascript)
- Overloading (http://php.net/manual/en/language.oop5.overloading.php#language.oop5.overloading.methods)
- Shorter Array Syntax

Introduzione a Laravel

Riassumendo ecco le principali features di Laravel (e mi riferirò sempre alla versione 5.3):

- Configurazione semplificata
- Modularità del codice
- Gestione avanzata del routing (GET, POST, PUT, DELETE)
- Eloquent ORM & Query Builder
- DB Schema Builder, Migrations, Seeding
- Blade Template Engine (derivazione di Twig)
- Email System & integration
- Autenticazione (ACL & Rules)
- Integrazione con Redis per il content caching
- Integrazione con sistemi di gestione di code (Beanstalkd, Amazon SQS)
- Astrazione del file system (Amazon S3)
- Event e command-bus

Creazione dell'ambiente di lavoro

Nella creazione dell'ambiente di lavoro farò distinzione tra Windows e Liunx/MacOSX, ma in comune c'è l'esigenza di avere installati:

- PHP 5.4 o superiore, disponibile sul PATH di sistema
- Composer (https://getcomposer.org) disponibile sul PATH di sistema
- nodejs e npm (https://nodejs.org)
- Un buon terminale (cmder)
- Grunt (http://gruntjs.com)
- Gulp (http://gulpjs.com)
- Git (https://git-scm.com) disponibile sul PATH di sistema

Creazione dell'ambiente di lavoro [Windows]

Nel caso di Windows (7/8/10) vi consiglio di utilizzare lo storico **XAMPP** (https://www.apachefriends.org/it/index.html) da installare nella cartella "c:\xampp" in modo da avere dei percorsi assoluti per "php.exe" e "mysql.exe"

Vi suggerisco di usare **Rapid EE** (http://www.rapidee.com/en/about) per configurare facilmente i PATH di sistema. Questo ci consentirà di utilizzare PHP e MySQL da command-line da qualsiasi cartella.
In questo caso dobbiamo (all'occorrenza) i nostri virtualhost manualmente: potete seguire quanto indicato in questa guida
http://www.gianfranco-castro.it/tutorial/2014/04/21/creare-virtuahost-xampp-per-windows.html

In alternativa possiamo usare anche **Laragon** (http://laragon.org) come validissima alternativa e sfruttare i VirtualHost automatici
(http://laragon.org/auto-create-virtual-hosts).

Il motivo della creazione dei virtual host (nonstante "artisan serve" che vedremo più avanti) è quello di testare un ambiente "simile" a quello di produzione.

Creazione dell'ambiente di lavoro [Windows]

Come terminale, vi consiglio di utilizzare **cmder** (http://cmder.net) come validissimo emultatore di terminale Unix, che ha un'installer che contiene anche *msygit* (https://git-for-windows.github.io) quindi con un solo installer abbiamo anche git; in alternativa ci sono anche **Console2** (http://sourceforge.net/projects/console/) e **ConEmu** (http://conemu.github.io) ma in questo caso dobbiamo installarci *git* manualmente e renderlo sempre dispnibile nel PATH.

Creazione dell'ambiente di lavoro [MacOSX]

Nel caso di Mac OSX, vi consiglio di utilizzare MAMP PRO (https://www.mamp.info/en/) ma occorrono due accorgimenti per rendere php e mysql disponibili nel PATH di sistema.

Consiglio di creare i seguenti alias nel file .bash_profile

alias **php**="/Applications/MAMP/bin/php/php5.6.10/bin/php"
alias **mysql**="/Applications/MAMP/Library/bin/mysql"
alias **composer**="~/bin/composer.phar"

Si preferisce utilizzare php-cli di MAMP e non quello "built-in" di Mac OSX in modo tale che php-cli utilizzi le stesse impostazioni di php "web", quindi può connettersi a MySQL e quindi far funzionare le migrations di artisan.

Creazione dell'ambiente di lavoro [Vagrant e Homestead]

Laravel mette a disposizione una macchina virtuale per lo sviluppo basata su Vagrant e Virtual Box già configurata e ottimizzata per lo sviluppo con PHP, MySQL, Apache. Inoltre è una soluzione multipiattaforma, quindi in un team di sviluppatori è indifferente se si utilizza Windows, Mac OSX o Linux: tutti hanno lo stesso sistema, ugualmente configurato.

Per l'utilizzo d Homestead occorre avere installato:

- Virtualbox (https://www.virtualbox.org)
- Vagrant (https://www.vagrantup.com)

Per installare Homestead occorre lanciare da riga di comando:

```
$ vagrant box add laravel/homestead
==> box: Loading metadata for box 'laravel/homestead'
       box: URL: https://vagrantcloud.com/laravel/homestead
==> box: Adding box 'laravel/homestead' (v0.2.2) for provider: virtualbox
       box: Downloading:
https://vagrantcloud.com/laravel/boxes/homestead/versions/0.2.2/providers/virtualbox.box
==> box: Successfully added box 'laravel/homestead' (v0.2.2) for 'virtualbox'!
```

Creazione dell’ambiente di lavoro [Vagrant e Homestead]

Una "box" è un termine che si riferisce ad un package di Vagrant; i package sono delle immagini di macchine virtuali con dentro già i vari sfotware installati e configurati; la comunità di sviluppatori di Vagrant mantiene un elenco ben documentato sulle varia macchine disponibili https://atlas.hashicorp.com/boxes/search tra cui appunto Homestead https://atlas.hashicorp.com/laravel/boxes/homestead

Qui la documentazione ufficiale per l’installazione
http://laravel.com/docs/5.3/homestead

Riassumo brevemente i vantaggi di utilizzo:
- Piaffatorma unica di sviluppo con tutto già installato
- Nel caso di sviluppo in team, tutti hanno lo stesso ambiente, ugualmente configurato

Qui qualche tutorial in merito
http://www.sitepoint.com/6-reasons-move-laravel-homestead/

http://www.sitepoint.com/quick-tip-get-homestead-vagrant-vm-running/

Creazione dell'ambiente di lavoro [Laravel Installer]

Quindi ottenuto il nostro ambiente di sviluppo, ci occorre installare Laravel Installer, che si peroccuperà di creare l'intera struttura di cartelle del nostro progetto; per installarlo occorre composer:

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

A fine installazione avremo un file eseguibile che anche questo va impostato nel PATH di sistema, nel mio caso:

```
$ ll /Users/gcastro/.composer/vendor/bin/
total 8
drwxr-xr-x 3 gcastro staff 102 6 Ago 12:38 .
drwxr-xr-x 9 gcastro staff 306 6 Ago 12:38..
lrwxr-xr-x 1 gcastro staff  28 6 Ago 12:38 laravel -> ../laravel/installer/laravel
```

Creazione dell'ambiente di lavoro [Laravel Installer]

Quindi lanciando il comando "laravel" abbiamo a disposizione il nostro installer per il framework:

```
gcastro: /Users/gcastro/laravelapps
$ laravel
Laravel Installer version 1.2.1

Usage:
  command [options] [arguments]

Options:
  -h, --help            Display this help message
  -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
  -v|vv|vvv, --verbose  Increase the verbosity of messages: 1 for normal output, 2 for more verbose out

Available commands:
  help  Displays help for a command
  list  Lists commands
  new   Create a new Laravel application.

gcastro: /Users/gcastro/laravelapps
$ laravel new gestionale
Crafting application...
sh: composer: command not found
Application ready! Build something amazing.
```

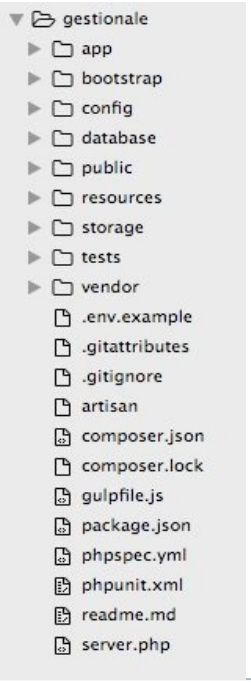
Creazione dell'ambiente di lavoro [Laravel Installer]

Proviamo quindi a creare un'applicazione che chiamiamo "gestionale" dentro la cartella "$HOME/**laravelapps**": ci basterà lanciare "`laravel new gestionale`"

```
gcastro: /Users/gcastro/laravelapps
$ laravel
Laravel Installer version 1.2.1

Usage:
  command [options] [arguments]

Options:
  -h, --help            Display this help message
  -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
  -v|vv|vvv, --verbose  Increase the verbosity of messages: 1 for normal output, 2 for more verbose out

Available commands:
  help  Displays help for a command
  list  Lists commands
  new   Create a new Laravel application.

gcastro: /Users/gcastro/laravelapps
$ laravel new gestionale
Crafting application...
sh: composer: command not found
Application ready! Build something amazing.
```

Struttura di un'applicazione

Il comando "`laravel new gestionale`" andrà quindi a creare la cartella "gestionale" con la seguente struttura:



Ecco nel dettaglio i file principali sulla quale concentrare la nostra attenzione.

- **.env:** Laravel5 utilizza la classe dotenv (https://github.com/vlucas/phpdotenv) per gestire la configurazione degli enrivonment, quindi configurazione del DB, eventuali chiavi di accesso agli ambienti cloud e tutte quelle variabili di configurazione.

- **app:** questa cartella contiene il codice effettivo della nostra applicazione, quindi i controller, i model, il middleware

Struttura di un’applicazione

- **artisan:** è lo strumento command-line che viene utilizzato durante lo sviluppo del nostro applicativo; ci consente di avviare le migrations, creare controller, gestire la cache applicativa, ottimizzare il codice, avviare il webserver di sviluppo.

- **bootstrap:** questa cartella contiene i file di avvio del nostro applicativo, la lettura della configurazione, la definizione dei path delle cartelle delle classi (in determinati casi), ma solitamente non è necessario modificare il contenuto dei file di questa cartella.

- **config:** questa cartella contiene i file di configurazione di Laravel come le credenziali di accesso al database, la gestione delle sessioni, etc. Impostazioni che possono essere indipendenti da .env

- **database:** questa cartelle contiene i file delle migrations e dei seed

- **gulpfile.js:** Laravel5 introduce una nuova feature chiamata Elixir per la gestione degli asset e per l'automatizzazione di task relativi ai file CSS, Javascript, test sulla UX, etc. Durante lo sviluppo è necessario avere nodejs e gulp installati

Struttura di un’applicazione

- **package.json:** è un file di configurazione utilizzato da Elixir per installare lo stesso Elixir e le relative dipendenze

- **phpspec.yml:** file di configurazione se sviluppiamo secondo il behavior driven development utilizzando il tool phpspec (http://www.phpspec.net/en/latest/).

- **public:** è la document root del nostro applicativo che contiene la parte pubblica esposta dal webserver, contiene i file .htaccess, favicon.ico, robots.txt e index.php; quest'ultimo file (il front controller) è il responsabile del caricamento e dell'esecuzione del nostro applicativo.

- **resources:** questa cartella contiene le views del progetto, i file di localizzazione e tutti i file sorgente relativi agli assett (CoffeeScript, SCSS, LESS, etc)

- **vendor:** questa cartella contiene tutte le classi e le librerie di terze parti necessarie e utilizzate dal nostro applicativo, Laravel compreso; generalmente non abbiamo la necessità di modificare i file di questa cartella ma utilizziamo il file composer.json per gestire quali librerie utilizzare nel nostro applicativo.

Struttura di un’applicazione

Quindi, dopo aver creato il nostro progetto (laravel new gestionale) possiamo utilizzare già artisan: lanciando il comando

```
$ php artisan serve
```

Verrà avviato il webserver di sviluppo che è in ascolto su http://localhost:8000 dove possiamo vedere una schermata iniziale di Laravel.

Dato che Laravel utilizza lo standard PSR-4 (http://www.php-fig.org/psr/psr-4/) per l’autoloading delle classi, dobbiamo impostare il namespace principale del nostro applicativo; per impostarlo, sempre da artisan, possiamo lanciare il comando

```
$ php artisan app:name gestionale
```

Configurazione dell’applicazione

La configurazione di Laravel risiede nella cartella “config” e i file da considerare sono

app.php: in questo file vi sono le impostazioni che riguardano il funzionamento di tutto l'applicativo tra cui: il debug mode, la URL (il dominio), il timezone e il locale e due array fondamentali: "provides" e "aliases".

L'array "providers" contiene la lista dei service providers che verranno caricati automaticamente all'avvio dell'applicazione e istanziati all'occorrenza; questo elenco si va a modificare man mano che aggiungiamo "moduli" (o bundle)

L'array "aliases" contiene l'elenco degli alias per le classi che vengono istanziate al lancio dell'applicazione.

auth.php: questo file contiene le impostazioni specifiche di autenticazione, inclusa la descrizione del model utilizzato per mappare gli utenti, il DB che contiene le informazioni.

Configurazione dell’applicazione

broadcasting.php: questo file viene utilizzato nella nuova feature di Laravel 5.1 per il broadcasting degli eventi.

cache.php: Laravel supporta diversi driver per la cache, compreso il file system, database, memcached, redis e altri; questo file viene utilizzato per specificare quale cache utilizzare con i relativi parametri.

compile.php: Laravel implementa un meccanismo di compilazione per migliorare le performance applicative, generando una serie di file per ottimizzare il package autoloading. Questo file consente di definire quali classi "custom" e addizionali devono essere incluse in questo processo di ottimizzazione.

database.php: questo file contiene le imposazioni per il database, estrapolate da .env, ma indipendenti da esso

filesystems.php: questo file contiene la definizione del filesystem del progetto, considerando anche la gestione degli asset, degli upload; attualmente sono supportati il filesystem locale, Amazon S3 e Raskspace.

Configurazione dell’applicazione

mail.php: qui la configurazione per il servizio di spedizione delle email: SMTP, Sendmail, PHP mail() function, Mailgun e le API di Mandrill.

queue.php: le code (queues) possono incrementare le performance di un'applicazione; Laravel supporta nativamente la comunicazione verso Beanstalkd e Amazon Simple Queue Service e in questo file vi è la configurazione.

services.php: se il nostro applicativo fa uso di servizi di terze parti (come Stripe per i pagamenti o Mandrill per le newsletter) in questo file vengono configurati gli accessi a tali servizi con i relativi provider.

session.php: molto probabilmente il nostro applicativo userà le sessioni per gestire l'autenticazione degli utenti e altri impostazioni di sessione a "run-time"; Laravel supporta diversi driver di sessione tra cui il filesystem, cookies, DB, APC, Memcached e Redis; in questo file va quindi configurato il driver desiderato, e gestire altri aspetti della funzionalità di gestione della sessione di Laravel.

view.php: in questo file viene configurata la cartella che conterrà tutte le viste del nostro applicativo, che di default è "resources/views"

Configurazione dell'applicazione

Sempre relativamente alla configurazione, Laravel suppone sempre di trovarsi nell'environment di produzione e questo si traduce che le impostazioni dei file della cartella "config" siano già quelle di produzione; in fase di sviluppo invece le impostazioni sono ovviamente diverse.

Lo scenario è quindi il seguente:

- In sviluppo utilizzo il file ".env" inserendo le impostazioni relative a questo ambiente
- In produzione **ELIMINO** il file ".env" e pongo attenzione, che sia sotto "**git ingore**" in modo da non inserirlo nei commit e nei push

Per verificare quindi quale environment è in uso basterà lanciare il comando
$ php artisan env

Che in presenza del file ".env" darà come output

```
Current application environment: local
```

Che in **ASSENZA** del file ".env" darà come output

```
Current application environment: production
```

Creazione di una prima applicazione

Nella creazione della nostra prima applicazione abbiamo bisogno di Laravel Installer: possiamo lanciare il comando

```
$ laravel new {nomeApp}
```

Verrà creata una cartella con dentro tutti i file del progetto; durante il corso creeremo un applicazione gestionale per un asset manager, quindi:

- Gestione anagrafica utenti
- Gestione Album e immagini per album
- Servizio di esposizione (Web Service REST) degli album e delle immagini
- C'è anche una parte pubblica di consultazione delle immagini caricate

Quindi andiamo a creare la nostra applicazione "AssetManager"

```
$ laravel new AssetManager
```

Creazione di una prima applicazione

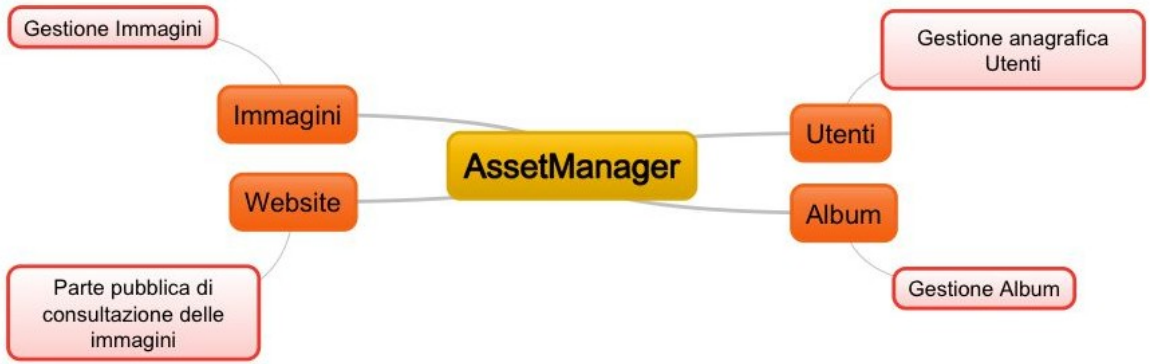
```
gcastro: /Users/gcastro/laravelapps
$ laravel new AssetManager
Crafting application...
> php -r "copy('.env.example', '.env');"
> php artisan clear-compiled
> php artisan optimize
Generating optimized class loader
> php artisan key:generate
Application key [0rkTFmPEKAl84coaOsJYhM5Om1lFAAEn] set successfully.
Application ready! Build something amazing.
```

Verrànno creati tutti i file necessari della nostra app; passo successivo è quello di definire il namespace globale della nostra app

```
$ php artisan app:name AssetManager
Application namespace set!
```

Creazione di una prima applicazione

La struttura del nostro applicativo è la seguente:



Creazione di una prima applicazione

Avremo quini i seguenti controller:

- Utenti
- Album
- Immagini
- Website

Andiamo quindi a creare il controller per la gestione degli Utenti; useremo il comando `artisan make:controller`

```
$ php artisan make:controller --help
Usage:
  make:controller [options] [--] <name>

Arguments:
  name                The name of the class

Options:
      --plain         Generate an empty controller class.
  -h, --help          Display this help message
```

Creazione di una prima applicazione

gcastro: **/Users/gcastro/laravelapps/AssetManager**
$ php artisan make:controller Utenti
Controller created successfully.

Verrà creato il file AssetManager/app/Http/Controllers/Utenti.php sulla quale andremo a creare la prima funzione public e che andremo successivamente a definire in /routes/web.php

Adesso andiamo a creare i routing e le relative viste.

Creazione di una prima applicazione

gcastro: **/Users/gcastro/laravelapps/AssetManager**
$ php artisan make:controller Utenti
Controller created successfully.

Apriamo il file /routes/web.php e andremo ad inserire:

```
Route::get('/utenti', 'Utenti@index');
```

Questo significa che per la URL http://applicativo/utenti verrà utilizzato il controller "Utenti.php" e la action "index".

Nel file app/Http/Controllers/Utenti.php

```
/**
 * Display a listing of the resource.
 *
 * @return Response
 */
public function index()
 {
  return view('utenti.index');
 }
```

Creazione di una prima applicazione

```
/**
 * Display a listing of the resource.
 *
 * @return Response
 */
public function index()
{
    return view('utenti.index');
```

Questo significa che verrà restituita una risposta HTTP 200 FOUND e verrà renderizzata la vista "index" contenuta nella cartella "utenti"; come abbiamo visto nelle slide precedenti la cartella "resources" contiene le "views" ; quindi creermo:

1) manualmente la cartella "utenti" dentro "resources/views"
2) manualmente il file chiamato index.blade.php
3) Inseriremo il seguente contenuto:

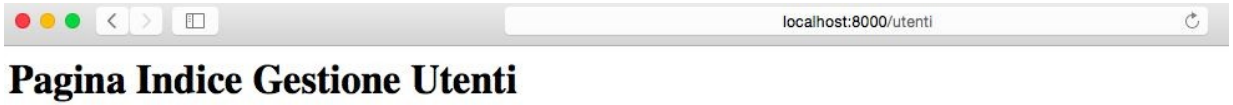
```
<h1>Pagina Indice Gestione Utenti</h1>
```

Creazione di una prima applicazione

Adesso da command-line lanciamo il Web Server di sviluppo:

```
$ php artisan serve
Laravel development server started on http://localhost:8000/
```

Andando sulla URL http://localhost:8000/utenti avremo



Nella gestione dell'HTML delle viste useremo Blade come template engine e in particolare del suo meccanismo di gestione del layout.

Debug di un’applicazione

Laravel mette a disposizione diversi strumenti per effettuare il debug e il logging di un’applicativo:

- La funzione“dd()”
- Il logging con Monolog e i plugin per i Browser
- La Debug Bar nativa

La funzione “dd()” è da considerare come un’evoluzione della funzione “var_dump()” [http://php.net/manual/en/function.var-dump.php] nativa di PHP e ho scopo di generare un output“human readable” di ciò che gli viene passato in ingresso.

Debug di un’applicazione[dd]

Se nella action Utenti@index andiamo ad inserire il seguente codice

```
public function index()
{
   $arUtente = [
      'nome' => 'Gianfranco',
      'cognome' => 'Castro'
   ];

   //===DumpHTMLdellavariabile
   dd($arUtente);

   return view('utenti.index');
}
```

Avremo il seguente output HTML alla URL /utenti

```
array:2 [▼
   "nome" => "Gianfranco"
   "cognome" => "Castro"
]
```

Debug di un'applicazione [Laravel Logger]

Laravel utilizza il diffuso framework Monolog (https://github.com/Seldaek/monolog) per la gestione del logging e come configurazione di default avremo il log concretizzato nel file che risiede su

AssetManager/storage/logs/**laravel.log**

Vediamo come usarlo: sempre su Utenti@index andremo a scrivere:

```
 // === Registro un messaggio di log
\Log::debug($arUtente);
```

Gli altri metodi per il logging sono:

```
\Log::info('message...');
\Log::warning('message...');
\Log::error('message...');
\Log::critical('message...');
```

Debug di un'applicazione [Laravel Debug Bar]

Laravel mette a disposizione una sua debug bar (https://github.com/barryvdh/laravel-debugbar), utilissima in fase di sviluppo

Per l'installazione occorrono questi semplici passaggi:

- Lanciare $ composer **require** barryvdh/laravel-debugbar
- Modificare il file config/app.php inserendo:
  - Il service provider
  - L'alias
- Lanciare "php artisan vendor:publish" per pubblicare la debugbar

Debug di un'applicazione [Laravel Debug Bar]

- Lanciare $ composer **require** barryvdh/laravel-debugbar

```
gcastro: /Users/gcastro/laravelapps/AssetManager
$ composer require barryvdh/laravel-debugbar
Using version ^2.0 for barryvdh/laravel-debugbar
./composer.json has been updated
> php artisan clear-compiled
Loading composer repositories with package information
Updating dependencies (including require-dev)
  - Installing maximebf/debugbar (v1.10.4)
    Downloading: 100%

  - Installing barryvdh/laravel-debugbar (v2.0.5)
    Downloading: 100%

maximebf/debugbar suggests installing kriswallsmith/assetic (The best way to manage assets)
maximebf/debugbar suggests installing predis/predis (Redis storage)
Writing lock file
Generating autoload files
> php artisan optimize
Generating optimized class loader
```

Debug di un’applicazione [Laravel Debug Bar]

- Modificare il file config/app.php inserendo sull’array “providers”:

```
'providers' => [

    /*
     * Laravel Framework Service Providers...
     */
    Illuminate\Foundation\Providers\ArtisanServiceProvider::class,
    Illuminate\Auth\AuthServiceProvider::class,
    Illuminate\Broadcasting\BroadcastServiceProvider::class,
            ...

    /*
     * Application Service Providers...
     */
    AssetManager\Providers\AppServiceProvider::class,
    AssetManager\Providers\AuthServiceProvider::class,
            ...
    /*
     * Provider custom
     */
    Barryvdh\Debugbar\ServiceProvider::class,
```

Debug di un'applicazione [Laravel Debug Bar]

- Modificare il file config/app.php inserendo sull'array "aliases":

```
'aliases' => [

    'App'      => Illuminate\Support\Facades\App::class,
    'Artisan' => Illuminate\Support\Facades\Artisan::class,
    'Auth'     => Illuminate\Support\Facades\Auth::class,
...
    'Debugbar' => Barryvdh\Debugbar\Facade::class,
```

- Lanciare "**php artisan vendor:publish**" per pubblicare la debugbar

```
$ php artisan vendor:publish
Copied File [/vendor/barryvdh/laravel-debugbar/config/debugbar.php] To
[/config/debugbar.php]
Publishing complete for tag []!
```

Debug di un'applicazione [Laravel Debug Bar]

Quindi basta andare su una URL e vedere la Debugbar in azione; inoltre (come da documentazione) può essere utilizzata come log/message bar utilizzando i seguenti metodi:

```
Debugbar::info($object);
Debugbar::error('Error!');
Debugbar::warning('Watch out...');
Debugbar::addMessage('Another message', 'mylabel');
```

Utilizzo di Tinker Console

Tinker Console è una console applicativa sul nostro progetto Laravel; ci consente di eseguire piccoli snippet di codice, fare esperimenti e piccoli test. In alcuni casi si rivela molto utile sia un fase di sviluppo che in produzione facilitando l'individuazione di bug.

Per lanciare questa console ci basterà eseguire

```
$ php artisan tinker
Psy Shell v0.5.2 (PHP 5.6.10 — cli) by Justin Hileman
>>> $ar = ['uno','due','tre']
=> [
     "uno",
     "due",
     "tre",
   ]
>>> \Log::debug($ar);
=> null
>>>
```

Utilizzo di Tinker Console

Ad esempio con i comandi $ar ... e \Log::debug($ar) abbiamo inserito delle informazioni di debug sul file storage/logs/laravel.log

Come vedremo più avanti, possiamo utilizzare Tinker per provare anche i nostri model.