

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Venus Framework Saint Ouen |  | PAQUET Judicaël |  | Judicael.paquet@gmail.com  https://github.com/las93 |

Venus Framework PHP

[Le nouveau framework MVC PHP]

Table des matières

1. [Présentation 2](#_Toc387998860)

[Installation](#_Toc387998861)

[Concept du routeur](#_Toc387998862)

[Routes avec des paramètres](#_Toc387998863)

[Routes complexes](#_Toc387998864)

1. [Nouvelle application 5](#_Toc387998865)

[Créer un nouveau projet](#_Toc387998866)

[Créer un contrôleur avancé](#_Toc387998867)

[Les modèles et entités de bases](#_Toc387998868)

[Les templates (vues) de bases](#_Toc387998869)

# Présentation

Venus Framework est une nouvelle alternative de qualité de développement open-source dans les environnements LAMP fonctionnant sous un système MVC complet.

## Installation

L’installation du Framework est relativement simple. Vous pouvez télécharger la version .zip sur le Github <https://github.com/las93>.

Un site web sera bientôt en ligne pour proposer des versions stables du framework.

Pour installer Venus Framework, il faut avoir au préalable avoir installé Linux (Windows), Apache (Nginx/IIS), Mysql et PHP 5.4.

De base, il existe un projet Demo pour vérifier votre installation. Il faut ajouter un vhost dans vos configurations Apache pour installer Venus Framework.

<VirtualHost \*:80>  
 ServerName localhost  
 DocumentRoot E:/venus/public/Demo/

<Directory E:/venus/public/Demo/>  
 DirectoryIndex index.php  
 AllowOverride All  
 Order allow,deny  
 Allow from all  
 </Directory>  
</VirtualHost>

Vous pouvez insérer les .htaccess dans vos configurations apache pour des raisons de performances.

## Concept du routeur

Le .htaccess est la base du routeur de notre framework. Il va router toutes les demandes vers le bootstrap (fichier de lancement) index.php du framework. Cependant, si un des fichiers demandés existent (comme une image par exemple), il retournera le contenu du fichier. Nous vous recommandons d’ailleurs de ne pas créer un autre fichier dynamique (type PHP) et de toujours travailler au sein du framework.

RewriteEngine on  
RewriteCond %{REQUEST\_FILENAME} !-f  
RewriteRule ^.\*$ /index.php [NC,L]

Le bootstrap lance le routeur du framework qui s’occupera d’appeler les méthodes nécessaires pour le bon fonctionnement de votre application.

Le routeur ne fait que respecter ce que vous lui définissez dans les fichiers de configuration. De base, le framework vous propose une configuration pour notre projet Demo :

{

"localhost" : {

"routes": {

"home": {

"route": "/",

"controller" : "\\Venus\\src\\Demo\\Controller\\Home",

"action": "show"

}

}

}

}

Les fichiers de configurations sont des fichiers Json. Vous pouvez d’ailleurs les faire valider sur ce site lors de vos modifications car l’erreur est très rapide :  
http://jsonlint.com/

« Localhost » est le domaine de votre application (ou la base). Vous pouvez également faire « localhost/demo ».

Dans notre fichier, nous définissons ensuite nos routes.

« home » est un simple alias qui sera cependant important car il permettra de créer des liens dynamiquement sur vos applications.   
« route » permet de définir l’URL a appeler pour lancer une action. Nous verrons plus tard, qu’il est possible de créer des URL dynamique.  
« controller » permet de définir la classe qui contiendra l’action à enclencher (le nommage respecte les namespace du framework. Même si cette méthode est assez ouverte, il est recommandé de suivre les normes de nommage comme vous le voyez dans l’exemple.  
« action » permet de définir l’action (en gros la méthode de la classe) à déclencher.

Vous pouvez aller voir le fichier /Venus/private/src/Demo/Controller/Home.php et la méthode show() pour voir comment le framework affiche un simple Hello World.

## Routes avec des paramètres

Les routes peuvent être plus dynamiques et prendre en compte par exemple des paramètres :

{

"localhost" : {

"routes": {

"home": {

"route": "/[:id]",

"controller" : "\\Venus\\src\\Demo\\Controller\\Home",

"action": "show",  
 "constraints": {

"id": "[0-9]+"

}

}

}

}

}

On a rajouté ici un paramètre id dans la « route » que l’on définit plus précisément dans les « constraints » sous forme d’expression régulière. Il faudra également rajouter $id dans la méthode show(). Les paramètres sont également automatiquement rajoutés dans le $\_GET.

Pour gérer une configuration alternative (pour un environnement de développement par exemple), vous pouvez créer un fichier Route.conf-local qui d’ailleurs est prioritaire sur le fichier Route.conf.

## Routes complexes

Venus Framework est un framework MVC donc il propose un système de templating complexe. Cependant pour des pages statiques, il permet de diriger une route directement vers un template pour omettre les phases MC.

Voici comment écrire par exemple sa route :

"template": {

"route": "/template/",

"template": "Demo",

"layout": false,

"content\_type": "html"

}

Nous allons créer également un fichier e:\venus\private\src\Demo\View\Demo.tpl avec du texte quelconque dedans. Le « template » comprend automatiquement le lieu où chercher le template ainsi que l’extension .tpl de celui-ci.

Le « layout » permet d’automatiser la présence d’un layout. Pour ajouter ce Layout, il suffit de créer un fichier e:\venus\private\src\Demo\View\Layout.tpl et d’ajouter cela à l’intérieur pour appeler le modèle définit : {include file=$model}

Vous pouvez ainsi tester de mettre true à « layout ».

Dans une route il est possible également de préciser la méthode http utilisée : GET, POST, HEAD, PUT, DELETE, TRACE, CONNECT pour créer des services full-rest par exemple.

"test": {

"route": "/template/",

"template": "MenuManager",

"layout": true,

"content\_type": "html",

"methods": "GET"

},

Vous pouvez également mettre du cache sur la page qui permettra d’indiquer le temps de vie de celle-ci.

"test": {

"route": "/template/",

"template": "MenuManager",

"layout": true,

"content\_type": "html",

"cache\_max\_age": 100

},

Pour mettre en place des pages en https, vous pouvez également indiquer que la page sera en HTTPS :

"test": {

"route": "/template/",

"template": "MenuManager",

"layout": true,

"content\_type": "html",

"schemes": "https"

},

# Nouvelle application

## Créer un nouveau projet

Pour se lancer dans l’aventure, le mieux c’est de se créer un nouveau projet. Pour cela, il faut directement travailler en ligne de commande afin de lancer le script de création de projet.

Sous Linux : /private/>php launch.php create\_project –p Front

Sous Windows : C:\Users\judicael.paquet>C:\xampp\php\php.exe -f "E:\venus\private\launch.php" – create\_project -p Front

Votre projet est à présent créé avec public/Front et private/src/Front.

## Créer un contrôleur avancé

Pour le moment nous avons vu qu’un contrôleur simple qui affichait un Hello World.

Comme dans de nombreux framework, le routeur va gérer la vue associée et le modèle (sous forme d’ORM complexe et d’entité). Nous allons commencer par regarder notre fichier e:/venus/private/src/Demo/Controller/Exemple1.php qui contient un exemple d’appel de template.

|  |
| --- |
| class Exemple1 extends Controller {  /\*\*  \* the main page  \*  \* @access public  \* @return void  \*/  public function show() {  $aExemple = $this->model->get();    $this->view  ->assign('Exemple', $aExemple)  ->display();  } ) |

Pour faire simple, la fonction appelle un modèle Exemple1 ($this->model est raccourcis vers le modèle du même nom que le contrôleur) et ensuite on appelle la vue qui affiche la liste des Exemple1 qu’on a récupéré. Comme pour le modèle, si on ne définit pas la vue à appeler, le framework va automatiquement appeler la vue du même nom soit ici Exemple1.tpl.

Pour tester en réel, ce code, il nous faut se connecter sur la base de donnée et de créer la table Exemple1 comme ceci :

CREATE DATABASE demo

Puis :

CREATE TABLE `Exemple1` (  
  `id` int(10) unsigned NOT NULL AUTO\_INCREMENT,  
  `title` varchar(150) NOT NULL,  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=latin1

INSERT INTO `demo`.`exemple1` (`id` ,`title`) VALUES (NULL , 'test1'), (NULL , 'test2');

Le système propose du scaffolding mais nous verrons cela plus tard.

Dès que cela est fait, il faut aller configurer la connexion de sa base de données dans le fichier e:/venus/private/src/Demo/conf/Db.conf pour mettre l’utilisateur et le passeport de la base de données.

{

"configuration": {

"demo": {

"type": "mysql",

"host": "localhost",

"db": "demo",

"user": "root",

"password": "",

"tables": {

"Exemple1": {

"fields": {

"id": {

"type": "int",

"key": "primary",

"null": false,

"undefined": true,

"autoincrement": true

},

"title": {

"type": "varchar",

"null": false

}

}

}

}

}

}

}

\* Si vous changez de base, il faut impérativement modifier le private/conf/Const.php et le champ « db\_conf ».

Comme vous pouvez d’ailleurs le voir, ce fichier vous permettra de créer l’ensemble des outils automatiquement à l’avenir pour utiliser vos bases de données.

Ajouter ensuite dans votre fichier de route, la nouvelle route pour tester ce contrôleur :  
e:/venus/private/src/Demo/conf/Route.conf

"model": {

"route": "/model",

"controller" : "\\Venus\\src\\Demo\\Controller\\Exemple1",

"action": "show",

}

Si vous avez réussi vous pourrez avoir dans votre navigateur :



## Les modèles et entités de bases

Afin de machin l’ensemble du travail du développeur, les bases de données sont gérées de façon pseudo-automatique.

Notre table Exemple1 a un modèle et une entité au sein du framework. Une **entité** est un conteneur représentatif de la table (avec des possibilités d’extensions) et le **modèle** est le gestionnaire de cette entité avec de nombreuses fonctions de manipulations de bases (plus de 90% des besoins).

Le modèle est une classe vide dans laquelle on peut réaliser des demandes ORM plsu complexes afin de répondre à l’ensemble des besoins de manipulations de Base de données. Voici un exemple :

|  |
| --- |
| /\*\*  \* Get Lasts folders  \*  \* @access public  \* @param integer $iLimit limit  \* @return array  \*/  public function getLastFolders($iLimit = 3) {  $aJoin = [  [  'type' => 'right',  'table' => 'article\_type',  'as' => 'at',  'left\_field' => 'at.id',  'right\_field' => 'a.id\_article\_type'  ]  ];  $result = $this->orm  ->select(array('SQL\_CALC\_FOUND\_ROWS', '\*'))  ->from($this->\_sTableName, 'a')  ->join($aJoin)  ->where(array('a.id\_article\_type' => 3))  ->orderBy(array('a.'.LibEntity::*getPrimaryKeyName*($this->entity).' DESC'))  ->limit($iLimit)  ->load();  $result[0]->count = $this->orm  ->select(array('FOUND\_ROWS()'))  ->load();  $result[0]->pages = *floor*($result[0]->count / 10);  if (*isset*($result)) { return $result; }  else { return array(); }  } |

Entité : e:/venus/private/src/Demo/Entity/Exemple1.php  
Modèle : e:/venus/private/src/Demo/Model/Exemple1.php

## Les templates (vues) de bases

Un template, outil efficace pour tout framework, a pour but de dissocier l’affichage et la gestion de donnée. Il propose cependant des options dynamiques afin de proposer un maximum de flexibilité tout en gardant une séparation propre.

Voici le template de notre exemple précédent : e:/venus/private/src/Demo/View/Exemple1.tpl

{foreach from=$Exemple key=$iKey item=$oExemple}

{$oExemple->get\_title()} - {$oExemple->get\_id()}<br/><br/>

{/foreach}

Notre template contient de l’HTML et un balisage particulier permettant de boucler sur les deux exemples que l’on a insérés en base.

Nous verrons plus tard toutes les options proposées par le système de templating de venus Framework.

[en cours]