

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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](#_Toc388274926)

[Installation](#_Toc388274927)

[Concept du routeur](#_Toc388274928)

[Routes avec des paramètres](#_Toc388274929)

[Routes complexes](#_Toc388274930)

1. [Nouvelle application 5](#_Toc388274931)

[Créer un nouveau projet](#_Toc388274932)

[Créer un contrôleur avancé](#_Toc388274933)

[Les modèles et entités de bases](#_Toc388274934)

[Les templates (vues) de bases](#_Toc388274935)

1. [Concepts avancés 9](#_Toc388274936)

[Utiliser un modèle différent du controller](#_Toc388274937)

[Des raccourcis dans les controller](#_Toc388274938)

[Manipulation des modèles](#_Toc388274939)

[Créer des entités/modèles en scaffolding](#_Toc388274940)

[Créer des vues](#_Toc388274941)

1. [Vos besoins 17](#_Toc388274942)

[Créer une page d’erreur 404/403](#_Toc388274943)

[Variables pré-définies et commentaires des templates](#_Toc388274944)

# 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.

# Concepts avancés

## Utiliser un modèle différent du controller

Nous avions vu pour faire une action qui appelait un modèle/entité du même nom mais vous aurez besoin d’appeler d’autres modèles/entités voire vous voudrez totalement dissocier le nom du controller aux noms des modèles/entités.

Pour cela nous allons créer une nouvelle table news dans notre base de données Mysql demo.

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

INSERT INTO `demo`.`news` (`id` ,`title`) VALUES (NULL , 'news1'), (NULL , 'news2');

Nous rajoutons notre table ensuite dans le DB.conf :

"news": {

"fields": {

"id": {

"type": "int",

"key": "primary",

"null": false,

"undefined": true,

"autoincrement": true

},

"title": {

"type": "varchar",

"null": false

}

}

}

Nous créons ensuite l’entité correspondante dans e:/venus/private/src/Demo/Entity/news.php

|  |
| --- |
| <?php  /\*\*  \* Model news  \*  \* @category Venus\src\  \* @package Venus\src\Demo\Entity  \* @author Judicaël Paquet <judicael.paquet@gmail.com>  \* @copyright Copyright (c) 2013-2014 PAQUET Judicaël FR Inc. (https://github.com/las93)  \* @license https://github.com/las93/venus/blob/master/LICENSE.md Tout droit réservé à PAQUET Judicaël  \* @version Release: 1.0.0  \* @filesource https://github.com/las93/venus  \* @link https://github.com/las93  \* @since 1.0  \*/  namespace Venus\src\Demo\Entity;  use \Venus\src\Demo\common\Entity as Entity;  /\*\*  \* Model news  \*  \* @category Venus\src\  \* @package Venus\src\Demo\Entity  \* @author Judicaël Paquet <judicael.paquet@gmail.com>  \* @copyright Copyright (c) 2013-2014 PAQUET Judicaël FR Inc. (https://github.com/las93)  \* @license https://github.com/las93/venus/blob/master/LICENSE.md Tout droit réservé à PAQUET Judicaël  \* @version Release: 1.0.0  \* @filesource https://github.com/las93/venus  \* @link https://github.com/las93  \* @since 1.0  \*/  class news extends Entity {  /\*\*  \* id  \*  \* @access private  \* @var int  \*  \* @primary\_key  \*/  private $id = null;  /\*\*  \* title  \*  \* @access private  \* @var string  \*  \*/  private $title = null;  /\*\*  \* get id of article  \*  \* @access public  \* @return int  \*/  public function get\_id() {  return $this->id;  }  /\*\*  \* set id of article  \*  \* @access public  \* @param int $id id of article  \* @return \src\FrontOffice\Entity\article  \*/  public function set\_id($id) {  $this->id = $id;  return $this;  }  /\*\*  \* get title of article  \*  \* @access public  \* @return string  \*/  public function get\_title() {  return $this->title;  }  /\*\*  \* set title of article  \*  \* @access public  \* @param string $title title of article  \* @return \src\FrontOffice\Entity\article  \*/  public function set\_title($title) {  $this->title = $title;  return $this;  }  } |

Nous créons ensuite le modèle correspondant dans e:/venus/private/src/Demo/Model/news.php

|  |
| --- |
| <?php  /\*\*  \* Entity news  \*  \* @category Venus\src\  \* @package Venus\src\Demo\Entity  \* @author Judicaël Paquet <judicael.paquet@gmail.com>  \* @copyright Copyright (c) 2013-2014 PAQUET Judicaël FR Inc. (https://github.com/las93)  \* @license https://github.com/las93/venus/blob/master/LICENSE.md Tout droit réservé à PAQUET Judicaël  \* @version Release: 1.0.0  \* @filesource https://github.com/las93/venus  \* @link https://github.com/las93  \* @since 1.0  \*/  namespace Venus\src\Demo\Model;  use \Venus\src\Demo\common\Model as Model;  /\*\*  \* Entity news  \*  \* @category Venus\src\  \* @package Venus\src\Demo\Entity  \* @author Judicaël Paquet <judicael.paquet@gmail.com>  \* @copyright Copyright (c) 2013-2014 PAQUET Judicaël FR Inc. (https://github.com/las93)  \* @license https://github.com/las93/venus/blob/master/LICENSE.md Tout droit réservé à PAQUET Judicaël  \* @version Release: 1.0.0  \* @filesource https://github.com/las93/venus  \* @link https://github.com/las93  \* @since 1.0  \*/  class news extends Model {  } |

A présent nous allons ajouter l’utilisation de ce modèle dans notre controller afin de bien comprendre la rapidité et l’efficacité du framework (c’est notre méthode recommandée mais ce n’est pas la seule méthode possible). Nous allons dans notre controller e:/venus/private/src/Demo/Controller/Exemple1.php

Nous commençons par rajouter l’alias « news » de la classe modèle en haut du fichier :

|  |
| --- |
| namespace Venus\src\Demo\Controller;  use \Venus\src\Demo\common\Controller as Controller;  use \Venus\src\Demo\Model\news as news; |

Ensuite nous créons notre raccourci d’accès au modèle dans le constructeur comme ceci :

|  |
| --- |
| public function \_\_construct() {  $this->modelNews = function() { return new news; };    parent::*\_\_construct*();  } |

Ensuite il ne nous reste plus qu’à modifier notre action show pour appeler ce nouveau modèle :

|  |
| --- |
| public function show() {  $aExemple = $this->modelNews->get();  $this->view  ->assign('Exemple', $aExemple)  ->display();  } |

Ensuite il suffit de vérifier le résultat dans notre navigateur :



## Des raccourcis dans les controller

Les controllers proposent des raccourcis intéressants qui pourraient bien vous aider à accélérer vos développements. Voici la liste de ceux-ci et les exemples associés.

Redirect

Vous pouvez créer une redirection http rapide (ou redirection interne en cli) comme ceci :

$oUrlManager = **new \Venus**\core\UrlManager;

$this->redirect($oUrlManager->getUrl('home', **array**()));

Forward

Vous pouvez créer une redirection vers une autre action comme ceci :

$oUrlManager = **new \Venus**\core\UrlManager;

$this->forward($oUrlManager->getUrl('home', **array**()));

Attention : La redirection n’est pas au niveau http, donc il n’y a pas de redirection 301 mais juste que le programme se dirige vers une autre action.

Cache

Vous pouvez cacher le développement d’un contrôleur en ajoutant un élément dans le PHPDoc de la méthode comme ceci :

/\*\*

\* The main menu manager

\*

\* **@access** public

\* **@return** void

\*

\* @Cache(maxage=12)

\*/

**public function** show() {

Raccourcis

Voici des raccourcis automatiquement propose à tous les contrôleurs que nous verrons plsu en détail par la suite :

$this->form = function() { return new Form(); };  
$this->security = function() { return new Security(); };  
$this->router = function() { return new Router; };  
$this->mail = function() { return new Mail; };  
$this->session = function() { return new Session; };  
$this->translator = function() { return new Translator; };

Vous pouvez automatiquement envoyer des erreurs HTTP dans les contrôleurs afin de gérer vous-mêmes des retours HTTP :

$this->NotFound();  
$this->Forbidden();

## Manipulation des modèles

Les modèles de Venus Framework sont simples à utiliser et complexes à la fois. Vous avez déjà vu précédement dans ce document un simple ->get() pour récupérer une liste de données sous forme de tableau d’objet.

|  |
| --- |
| public function show() {  $aExemple = $this->modelNews->get();  $this->view  ->assign('Exemple', $aExemple)  ->display();  } |

Le get() renvoi un tableau de ce type :

Array(new news(), new news());

Ceci permet de pouvoir gérer facilement l’ensemble des données :

foreach ($aExemple as $iKey => $oNews) {  
 echo $oNews->get\_title() . “<br/>”;  
}

Voici les fonctions natives des modèles :

|  |  |
| --- | --- |
| get() ou get($ocriteria) | Obtenir une liste |
| update(($ocriteria) | Modifier |
| insert(($ocriteria) | Insérer |
| delete(($ocriteria) | supprimer |

Vous pouvez également utiliser des méthodes très utiles comme décrites ci-dessous :

$oModel->findOneByid(12); (quand on a un champ id) => retourne un seul résultat

$oModel->findByid(12); (quand on a un champ id) => retourne tous les résultats correspondant

$oModel->findAll();=> retourne tout

$oModel->findOneBy(array('id' => 12); => retourne un résultat mais on passe les critères sous forme de tableau (permet de definir plusieurs critères)

$oModel->findBy(array('id' => 12); => retourne les résultats mais on passe les critères sous forme de tableau (permet de definir plusieurs critères)

Il est possible également de modifier ou de supprimer directement une entité pour gagner du temps :

$oEntity->save() ; ou $oEntity->remove() ;

## Créer des entités/modèles en scaffolding

Le scaffolding est un concept de création de classes à partir d’une base de données. Même si notre scaffolding se base sur le Db.conf, il a pour but de créer les entités et modèles automatiquement afin de représenter les tables sous forme d’objets manipulables.

Nous allons créer une nouvelle table avec notre moteur de scaffolding en ajoutant un schéma dans notre DB.conf de notre projet e:/venus/private/src/Demo/conf/Db.conf :

"article": {

"fields": {

"id": {

"type": "int",

"key": "primary",

"null": false,

"undefined": true,

"autoincrement": true

},

"title": {

"type": "varchar",

"null": false

}

}

}

Il faut ensuite lancer le script pour le scaffolding :

Sous Linux : /private/>php launch.php scaffolding –p Demo -c

Sous Windows : C:\Users\judicael.paquet>C:\xampp\php\php.exe -f "E:\venus\private\launch.php" -- scaffolding -p Demo –c

La table va se créer en base de données et il va également créer l’entité et le modèle correspondant dans notre projet.

## Créer des vues

Dans un framework MVC, il est indispensable de gérer des vues. Sous venus Framework, les vues sont des templates qui proposent de nombreuses possibilités dynamiques.

Voici un exemple de vue pour mieux comprendre déjà inclus dans le package que vous avez téléchargé :

|  |
| --- |
| {foreach from=$Exemple key=$ikey item=$oExemple}  {$oExemple->get\_title()} - {$oExemple->get\_id()}<br/><br/>  {/foreach} |

Les templates proposent comme vous le voyez ci-dessus, un foreach proche du php pour parcourir un tableau complet.

Les templates proposent deux types de méthodes : les modifiers et les fonctions.

Les modifiers vont permettre de modifier la variable proposée juste avant de l’afficher comme l’exemple ci-dessous :

{$title|capitalize} => va mettre le contenu de $title avec une majuscule sur le premier caractère.

Les fonctions qui vont demander des paramètres d’entrées, vont permettre d’excuter quelque chose (et pas appliquer une simple modification avant sortie). Voici un exemple :

{foreach from=$array key=$iKey item=$oArticle}{/foreach} => comme un foreach en PHP.

Une vue n’a accès aux variables que si le contrôleur lui en donne l’autorisation en assignant une variable au template. Nous l’avions déjà vu dans ce document sous cette forme :

|  |
| --- |
| $this->view  ->assign('Exemple', $aExemple)  ->display(); |

Dans le contrôleur, on assigne une variable au template (assign() ) avant delancer le template (display() ). La variable $Exemple sera donc utilisable dans le template appelé.

# Vos besoins

## Créer une page d’erreur 404/403

Il est bien pratique de pouvoir créer des pages d’erreurs personnalisées pour les applications de type web. Venus Framework propose de mettre en place ce type de page très rapidement.

Dans le package de base, vous avez déjà un exemple en place (e:/venus/private/conf/Route.conf) où l’on ajoute une route puis le template. Voici la route à créer :

|  |
| --- |
| "routes": {  "404": {  "template": "Error",  "vars":{  "error\_message": "Not Found",  "error\_code": "404"  },  "content\_type": "html",  "cache": {  "max\_age": 100  },  "methods": "GET"  }  } |

Les routes 404 et 403 sont automatiquement reconnues comme les routes de pages d’erreur. Les vars permettent de passer automatiquement des variables au template appelé dans la conf (cela permet d’appeler le même template avec des variables différentes).

Il vous suffira de modifier e:/venus/private/src/Demo/conf/Route.conf et e:/venus/private/src/Demo/View/Error.tpl pour personnaliser vos pages d’erreurs.

## Variables pré-définies et commentaires des templates

Dans les templates du venus Framework, il existe de nombreuses variables pré-définies afin d’accélérer la tache des développeurs/intégrateurs.

{counter} => permet d’afficher le numéro de boucles (pour les for par exemple)

{$SCRIPT\_NAME} => permet d’afficher la variable $\_SERVER[‘SCRIPT\_NAME’]

{\* et \*} => entourent des commentaires. Ceci ne s’affichera pas sur le site, c’est uniquement pour mettre des repères pour les développeurs/intégrateurs au sein du code source.

{literal} et {/literal} => permettent d’entourer du contenu statique. En gros le moteur de templating n’ira pas retraduire ce qu’il lit entre ces deux balises. C’est essentiel pour faire du javascript où les caractère { et } portent confusion avec le framework.

{$app.server} => représente la variable PHP $\_SERVER ;

{$app.get} => représente la variable PHP $\_GET ;

{$app.post} => représente la variable PHP $\_POST ;

{$app.cookies} => représente la variable PHP $\_COOKIES ;

{$app.env} => représente la variable PHP $\_ENV ;

{$app.session} => représente la variable PHP $\_SESSION ;

{$app.request} => représente un merge de toutes les variables tableaux dans ce sens strictement prioritaire : $\_GET, $\_POST, $\_COOKIES, $\_SERVER, $\_ENV.

{$app.now} => représente le timestamp du moment

{$app.conf} => représente toutes les constantes PHP définient.

{$app.template} => affiche le nom du template en cours

{$app.template\_object} => affiche l’object du template en cours.

{$app.current\_dir} => affiche le dossier du template en cours

{$app.version} => représente le numéro de version du système de template utilisé

{$app.ldelim} et {$app.rdelim} => représente les délimiteurs définissant {literal} et {/literal} que l’on peut reconfigurer à sa guise.

[en cours]