Programmation MVC

Cyril Rabat
cyril.rabat@univ-reims.fr

Licence 2 Informatique - Info0303 - Programmation Web 2

2020-2021

Cours n°6

Présentation du modèle MVC

Présentation de Laravel

Version 27 septembre 2020

Table des matières

Exemple : un site de vente en ligne

- Une boutique qui vend des articles
- Quels acteurs ?
  ↪ Visiteur
  ↪ Administrateur
  ↪ Acheteur
- Quelles actions ? Qui peut faire quoi ?

Table des matières

Cas d'utilisation

- Objectifs :
  - Définir les acteurs du site
  - Établir les interactions fonctionnelles entre acteurs et application
- Représentation des acteurs
- Représentation des actions
- Liens entre les acteurs et les actions qu'ils peuvent réaliser
- Possible de définir des héritages
  ↪ Lorsqu'un acteur hérite des interactions d'un autre acteur

Exemple avec le site de vente en ligne

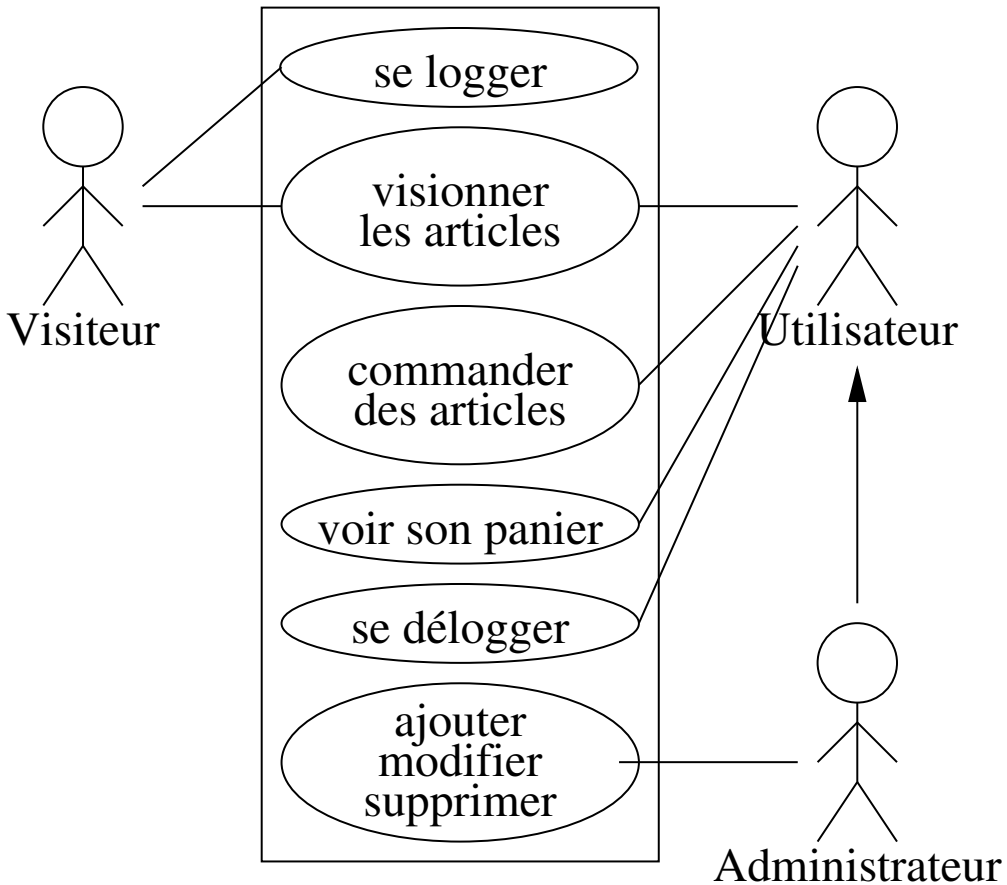
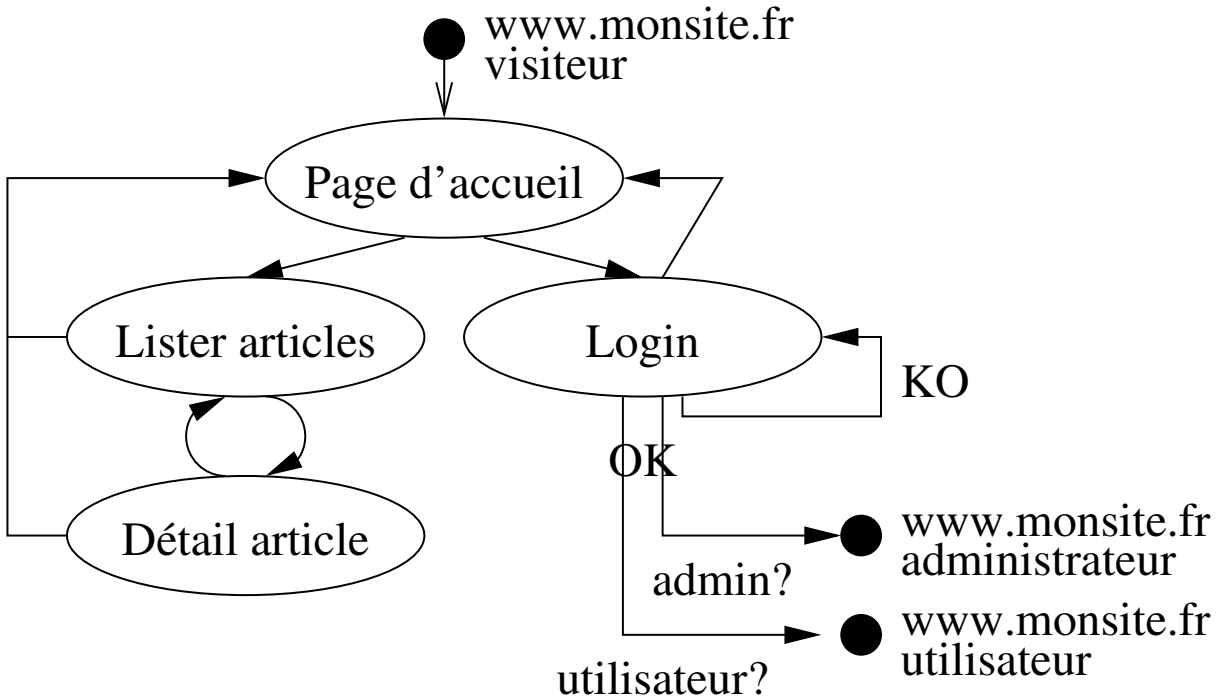


Table des matières

Diagramme de navigation

- Représente les sections de l'application
- Affiche les liens entre ces sections
- Possible de spécifier les données échangées
- Un diagramme de navigation par type d'utilisateur
  ↪ Les pages accessibles sont spécifiques

Exemple avec le site de vente en ligne (1/2)



Exemple avec le site de vente en ligne (2/2)

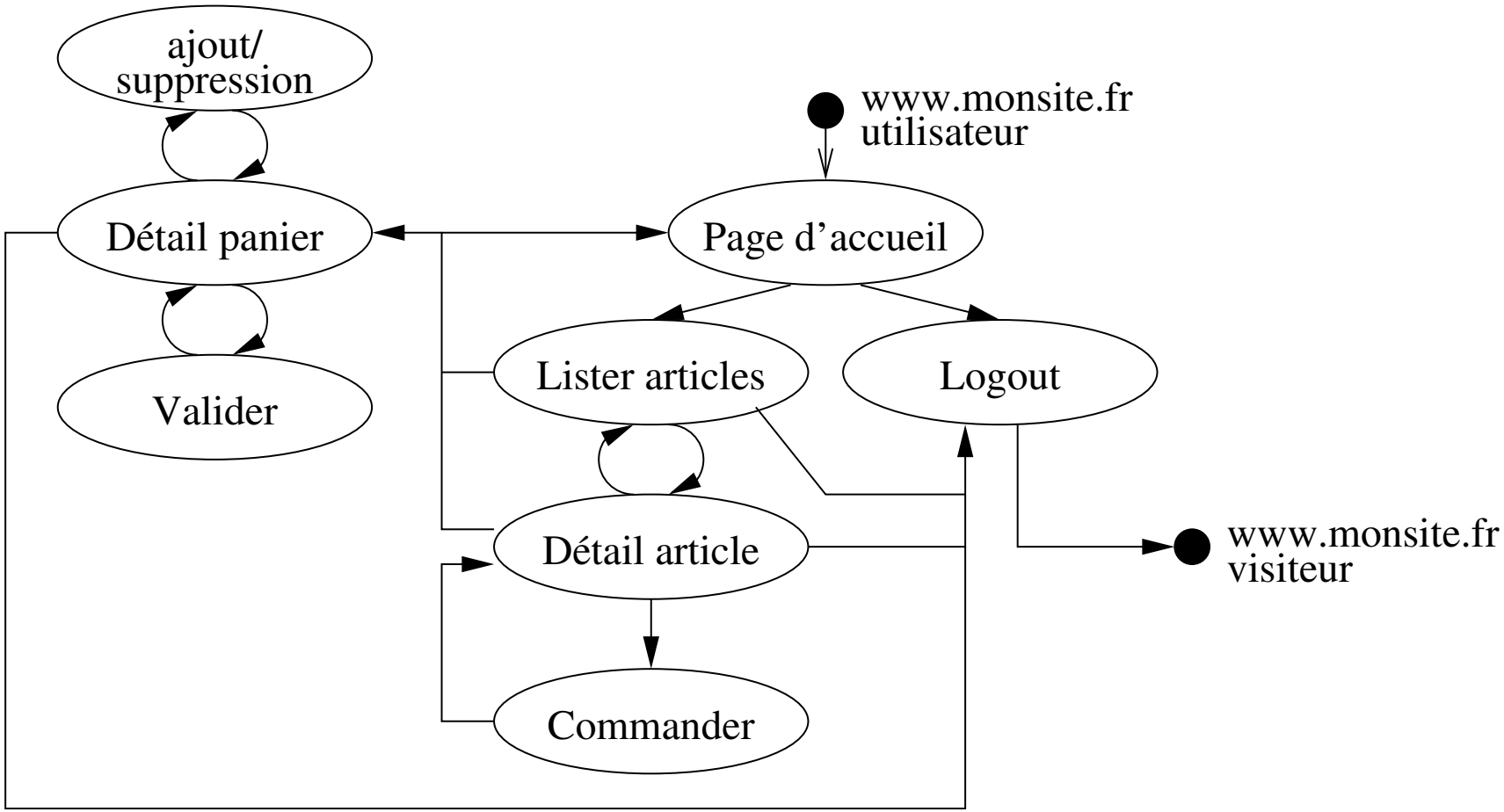


Table des matières

Flux de développement

- Expression des besoins
  ↪ Cahier des charges (sujet de projet)
- Partie fonctionnelle :
  - Définir les acteurs, les interactions avec l'application
    ↪ Cas d'utilisation
  - À partir du cas d'utilisation, génération du diagramme de navigation
- Partie *design* :
  - Création de la maquette
    ↪ Utilisation de logiciels spécifiques
    ↪ Généralement, un métier spécifique
  - Génération du/des templates
    ↪ Automatique suivant les logiciels

Illustration

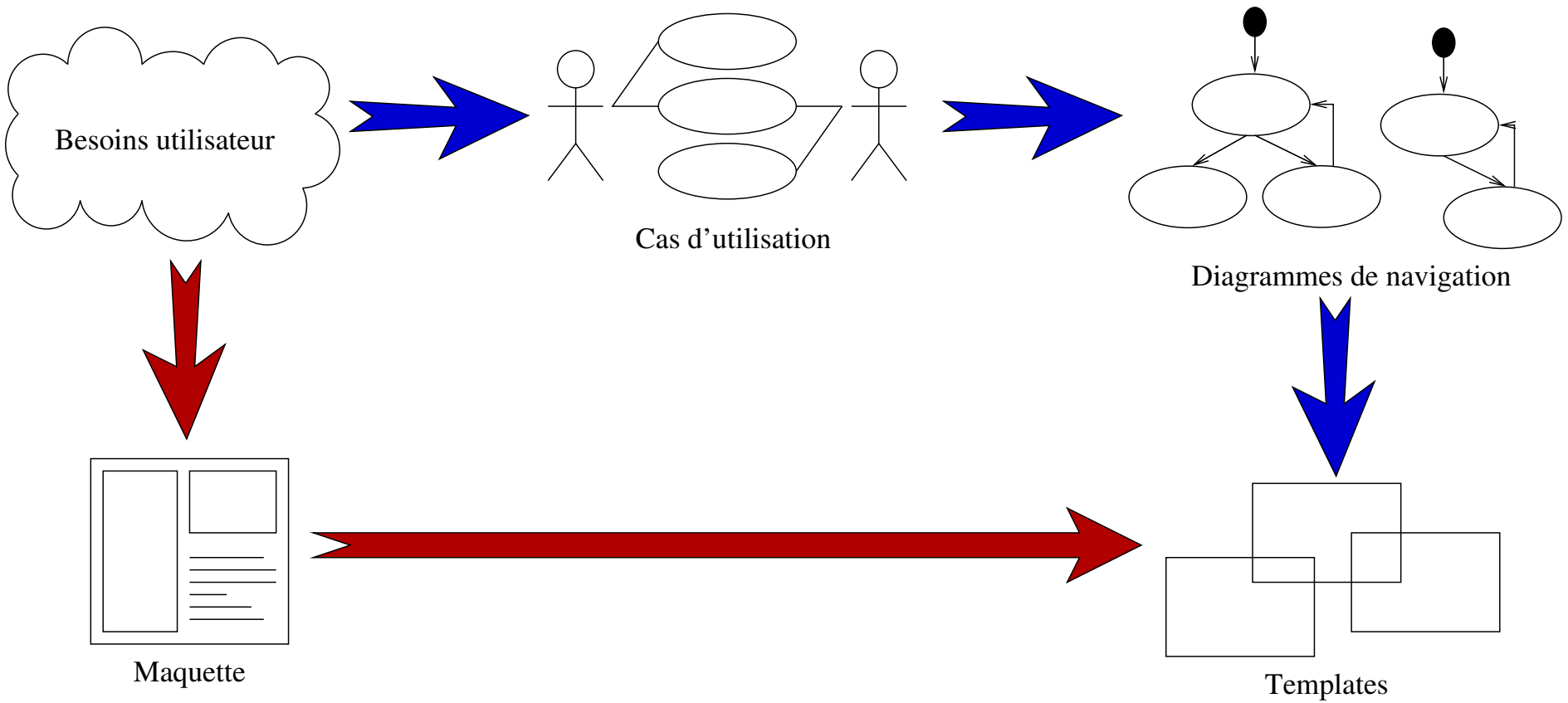


Table des matières

Motivations

- Particularité des applications Web :
  ↪ Mélange de technologies
  ↪ Codes exécutés côté client et serveur
- Script PHP : permet de travailler sur toutes les parties
  ↪ Front (génération de HTML), base de données, traitement, *etc.*
- Pour les grosses applications :
  ↪ Difficultés de développement
  ↪ Problèmes de maintenance
- Nécessité de structurer l'application
- Vers une décomposition efficace :
  - Séparation des fonctionnalités
  - Communication claire entres les différentes parties

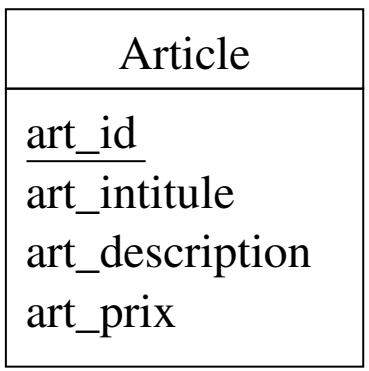
Parties de l'application

- Interface :
  - Affichage des données pour l'utilisateur
  - Récupération des données saisies
- Contrôle :
  - Déclenche des actions associées aux actions
- Logique applicative
  - Traitement associés à une application spécifique
- Logique métier/modèle
  - Représentation des données
  - Traitements associés
- Persistance
  - Sauvegarde/chargement des données
  - Utilisation d'une base de données

Table des matières

Description de l'application d'exemple

- Magasin en ligne proposant la vente d'articles
- Article caractérisé par un identifiant, un intitulé, une description et un prix
- Page d'accueil : affichage de tous les articles



MCD (pour l'instant)

Exemple 1 : affichage de la liste des articles

```
<!DOCTYPE html>
<html lang="fr">
  <head> ... </head>
  <body>
    <h1> Bienvenue dans mon magasin </h1>
<?php
$BD = new PDO("mysql:host=localhost;dbname=articles;charset=UTF8",
              "root", "");
if($requete = $BD->query("SELECT␣*␣FROM␣article")) {
    while($resultat = $requete->fetch(PDO::FETCH_ASSOC)) {
?>
    <div>
      <h2><?php echo $resultat['art_intitule']; ?></h2>
      <p><?php echo $resultat['art_description']; ?></p>
      <b><?php echo $resultat['art_prix']; ?> \euro </b>
    </div>
<?php
    }
}
?>
  </body>
</html>
```

Problèmes de cette solution

- Script mélangeant du code HTML et du code PHP
- Difficilement lisible !
- Solution : séparer le code
  - Partie PHP = récupération des données
    ↪ Contrôle : script `index.php`
  - Partie HTML = représentation des données
    ↪ Vue : script `vueArticles.php`
- Données récupérées par le contrôle puis passées à la vue
  ↪ Exemple : tableau `$articles`

Exemple 2 (1/2) : vueArticles.php

```
<!DOCTYPE html>
<html lang="fr">
  <head> ... </head>
  <body>
    <h1> Bienvenue dans mon magasin </h1>
<?php
foreach($articles as $article) {
  echo <<<HTML
    <div>
      <h2>{$article['art_intitule']}</h2>
      <p>{$article['art_description']}</p>
      <b>{$article['art_prix']} euros</b>
    </div>
HTML;
}
?>
  </body>
</html>
```

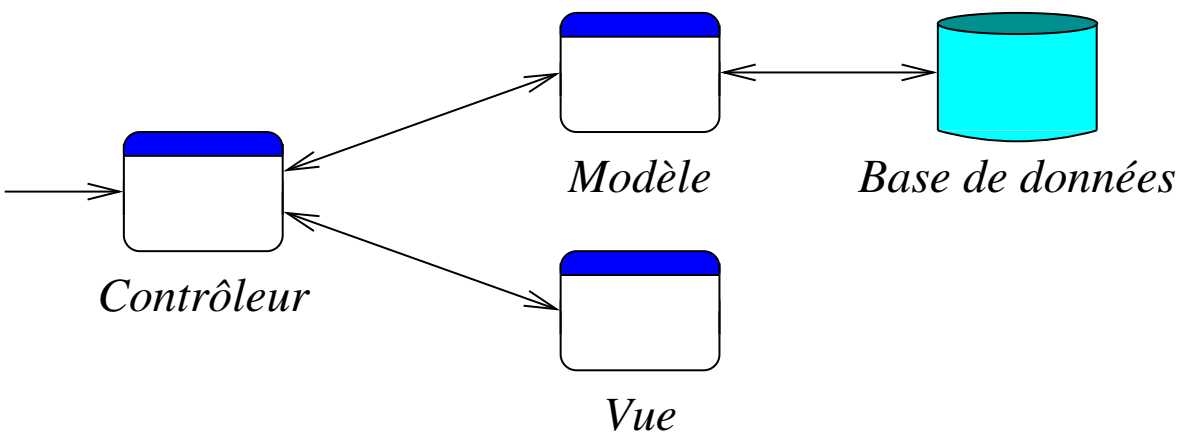
Exemple 2 (2/2) : index.php

```php
<?php
$BD = new PDO("mysql:host=localhost;dbname=articles;charset=UTF8",
              "root", "");

$articles = $BD->query("SELECT * FROM article");

require("vueArticles.php");
```

Table des matières

Séparer le contrôle du modèle

- Pour le moment, le script principal `index.php` :
  ↪ Réalise les accès à la base de données
  ↪ ET dirige vers la vue
- Solution : séparer encore le code
  - Modèle : récupération des données (accès à la base)
  - Vue : représentation des données
  - Contrôleur : lien entre la vue et le modèle
- Le modèle possède plusieurs fonctionnalités :
  ↪ Utilisation de fonctions différentes
  ↪ Possible d'utiliser une classe



Exemple 3 (1/2) : `ArticleModel.php` (le modèle)

```php
<?php
class ArticleModel {
  public static function getArticles() : PDOStatement {
    $DB = MyPDO::getInstance();

    return $DB->query("SELECT * FROM article");
  }
  public static function create(Article $a) : bool { ... }
  public static function read(int $id) : Article { ... }
  public static function update(Article $a) : bool { ... }
  public static function delete(int $a) : bool { ... }
}
```

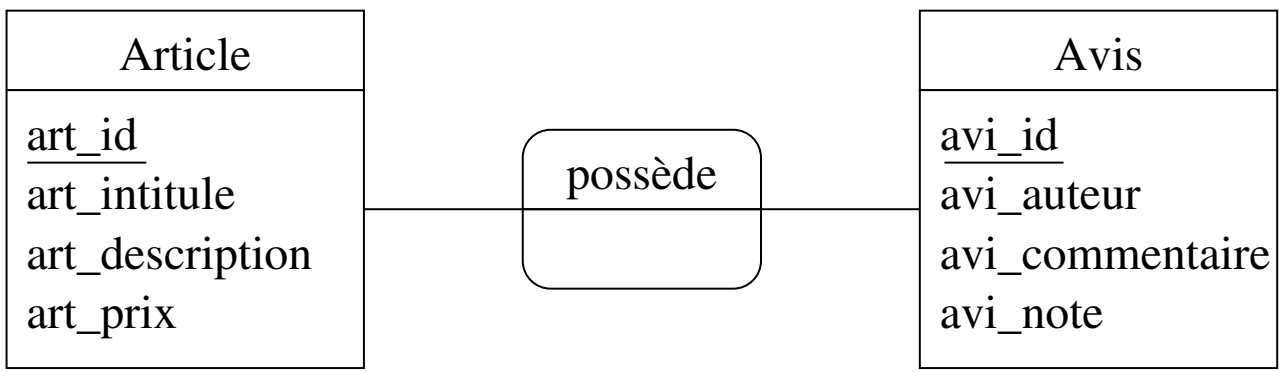
- Classe CRUD (pour *Create*, *Read*, *Update* et *Delete*)

Exemple 3 (2/2) : index.php (le contrôleur)

```php
<?php
require("ArticleModel.php");

$articles = ArticleModel::getArticles();

require("vueArticles.php");
```

Ajouts de fonctionnalités

- Nous désirons maintenant récupérer les avis des utilisateurs
- Modification du MCD (ajout d'une entité = nouvelle table)
- Nouveau contrôleur :
  ↪ Récupère l'identifiant de l'article
  ↪ Appel du modèle/vue correspondant
- Modèle : ajout de nouvelles méthodes
  ↪ Récupération d'un article et des avis d'un article



MCD modifié

Exemple 4 (1/5) : `ArticleModel.php` (un modèle)

```php
<?php
public class ArticleModel {
  public static function getArticles() : PDOStatement {
    $DB = MyPDO::getInstance();
    return $DB->query("SELECT * FROM article");
  }
  public static function getArticle(int $idArticle) : array {
    $DB = MyPDO::getInstance();
    $requete = $DB->prepare("SELECT * FROM article WHERE art_id=:
        article");
    $requete->execute([":article" => $idArticle]);
    return $requete->fetch();
  }
  ...
}
```

Exemple 4 (2/5) : AvisModel.php (un modèle)

```
public class AvisModel {
  public static function getAvis(int $idArticle) : PDOStatement {
    $DB = MyPDO::getInstance();
    $requete = $DB->prepare("SELECT␣*␣FROM␣avis␣WHERE␣avi_article=:
        article");
    $requete->execute([":article" => $idArticle]);
    return $requete;
  }
  ...
}
```

Exemple 4 (3/5) : `controller.php` (le contrôleur)

```php
<?php
require("ArticleModele.php");

if(isset($_GET['article'])) {
    $article = ArticleModel::getArticle(intval($_GET['article']));
    $listeAvis = AvisModel::getAvis(intval($_GET['article']));
    require("vueArticle.php");
}
else {
    echo "Erreur ! Identifiant de l'article manquant.";
}
```

Exemple 4 (4/5) : vueArticles.php (première vue)

```
<html lang="fr">
  <head> ... </head>
  <body>
    <h1> Bienvenue dans mon magasin </h1>
<?php
foreach($articles as $article) {
  echo <<<HTML
    <div>
      <h2>{$article['art_intitule']}</h2>
      <p>{$article['art_description']}</p>
      <b>{$article['art_prix']} euros</b>
      <p>
        <a href="article.php?article={$article['art_id']}">
          Voir cet article
        </a>
      </p>
    </div>
HTML;
}
?>
  </body>
</html>
```

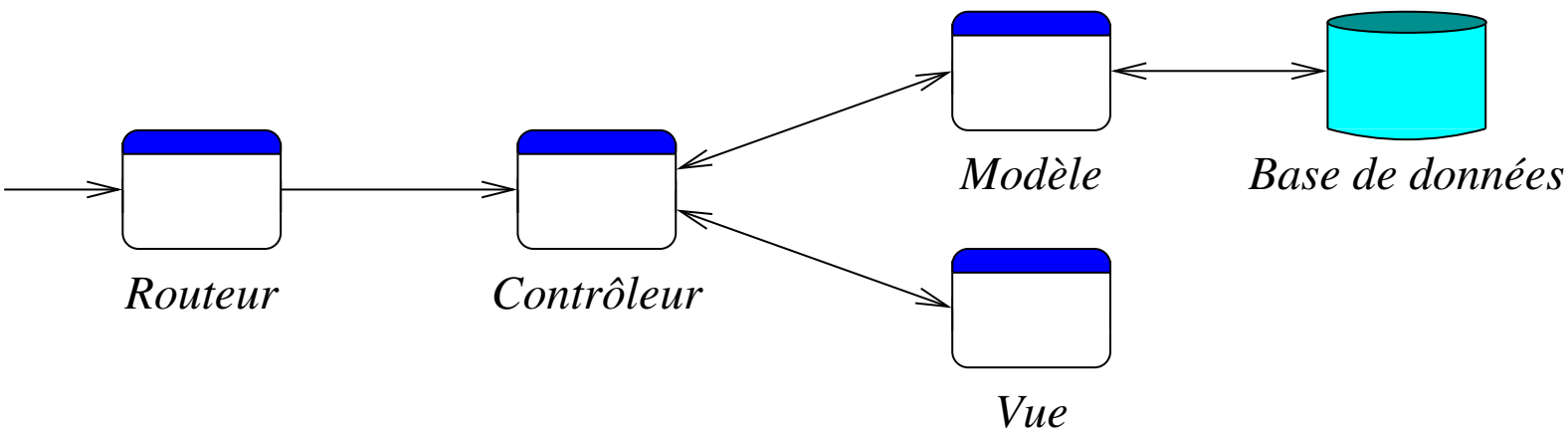
Exemple 4 (5/5) : vueArticle.php (deuxième vue)

```
...
  <body>
    <h1> Article sélectionné </h1>
    <div>
      <h2><?php echo $article['art_intitule']; ?></h2>
      <p><?php echo $article['art_description']; ?></p>
      <b><?php echo $article['art_prix']; ?> ? </b>
    </div>
    <h2> Avis des utilisateurs </h2>
<?php
while($avis = $listeAvis->fetch()) {
  echo <<<HTML
    <div>
      <b>{$avis['avi_auteur']}</b>
      <i>{$avis['avi_commentaire']}</i>
      <b>{$avis['avi_note']} / 5 </b>
    </div>
HTML;
} ?>
    <p> <a href="index.php"> Retour à l'accueil </a> </p>
  </body>
</html>
```

Table des matières

Routage dans l'application

- Pour le moment, un contrôleur par action :
  ↪ Multiplication des contrôleurs dans l'application finale !
- Solution : centraliser sur un seul point d'accès
- Le routeur ou le *front controller*



Mise en place du routeur

- Contrôleur :
  - Contient les différentes parties (dans des fonctions)
  - Peut être découpé en sous-parties
- Routeur :
  - Vérifications générales (exemple : conditions d'accès, paramètres)
  - Appelle les méthodes du contrôleur
  - Possible d'utiliser un attribut spécifique (paramètre `action`)
    $\hookrightarrow$ Vérifier la validité pour la sécurité !

Exemple 5 (1/2) : index.php (le routeur)

```php
<?php
require("controller.php");
define("ACTION_DEFAUT", 1);
define("ACTION_ARTICLE", 2);

if(isset($_GET['action']))
    $action = intval($_GET['action']);
else
    $action = ACTION_DEFAUT;

switch($action) {
    case ACTION_ARTICLE:
        if(isset($_GET['article']))
            ArticleController::afficherArticle();
        else
            require("vueErreur.php");
        break;
    default:
        ArticleController::listerArticles();
        break;
}
```

Exemple 5 (2/2) : ArticleController.php

```php
<?php
require("modele.php");

class ArticleController {
  public static function listerArticles() : void {
    $articles = getArticles();
    require("vueArticles.php");
  }

  public static function afficherArticle() : void {
    $article = getArticle(intval($_GET['article']));
    $listeAvis = getAvis(intval($_GET['article']));
    require("vueArticle.php");
  }
}
```

Table des matières

Problématique

- Modèle MVC : séparation des fonctions
  ↪ Maintenance plus simple
  ↪ Réutilisation du code
- Problème : multiplication des fichiers !
- Séparation des fichiers publiques des fichiers privés
- Ajout de bibliothèques externes
- Solution : proposer une arborescence
  ↪ Séparation en fonction des parties de l'application

Arborescence classique

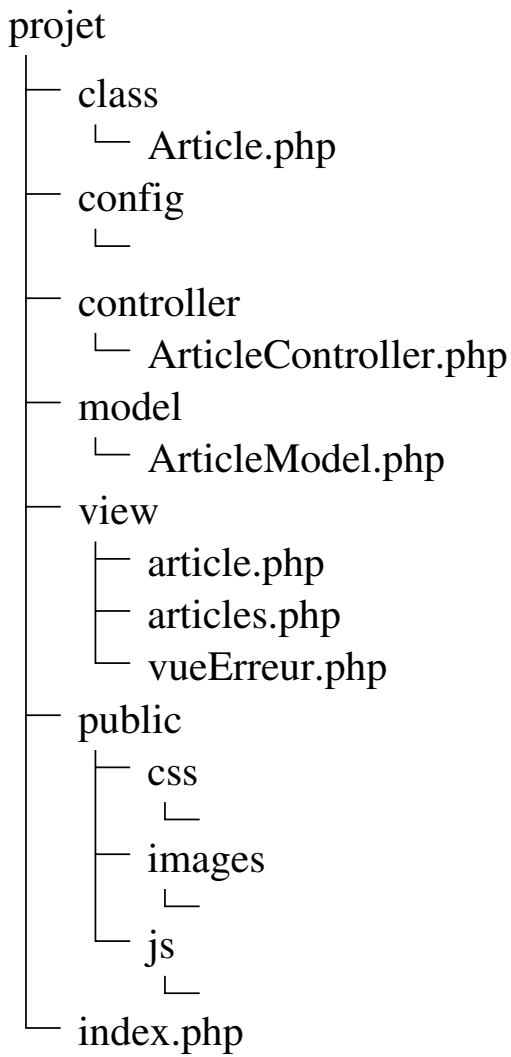


Table des matières

Introduction

- Développement d'une application Web «*from scratch*» déconseillé
- Quelques *frameworks* PHP :
  - *Symfony*, *CakePHP*, *Laravel*, *CodeIgniter*, ...
- L'utilisation d'un *framework* apporte :
  - Les éléments classiques
  - Des facilités d'écriture
  - La gestion de la BD simplifiée
  - Des outils de débogage en ligne
  - Une sécurité renforcée...
- Nécessite une connaissance du *framework*
  $\hookrightarrow$ Il n'empêche pas les mauvaises pratiques

Ce que propose *Laravel*

- Système de routage complet (RESTFul et ressources)
- Créateur de requêtes SQL, ORM (*Object-Relational Mapping*)
- Moteur de template
- Système d'authentification, de validation, de pagination
- Gestion des migrations pour les bases de données
- Gestion des sessions, des évènements, des autorisations
- Système de cache...

Installation de *Laravel*

- Nécessite :
  - PHP en ligne de commandes
    ↪ Attention à la configuration et au fichier `php.ini`
  - Activation de certaines bibliothèques (BD, SSL, *etc.*)
  - *composer*, un gestionnaire de dépendances
  - Un accès à Internet
- Pour créer un projet en ligne de commandes :
  `composer create-project -prefer-dist laravel/laravel exemple`
- Les modifications sont réalisées à l'aide de `artisan`
  ↪ `php artisan`

Arborescence et fichiers utiles

- `app` : les données de l'application
  ↪ Modèles, contrôleurs
- `bootstrap` : scripts d'initialisation
- `config` : configuration de l'application
- `database` : migrations et peuplement
  ↪ Définition des classes, du peuplement de la base
- `public` : dossiers publiques de l'application
  ↪ Le routeur, l'icône, `.htaccess`, *etc.*
- `resources` : vues, fichiers de langue...
- `routes` : routes de l'application
- `vendor` : bibliothèques, API, *etc.*
- `.env` (à la racine) : fichier de configuration
  ↪ Application, base de données, mail, *etc.*

Table des matières

Point d'entrée

- Le routeur est le point d'entrée de l'application
  ↪ `public/index.php`
- Il charge le fichier `web.php` qui décrit les routes
  ↪ Ce fichier doit être personnalisé

Contenu par défaut de `web.php`

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

- Le '/' indique l'URL (ici, uniquement le nom de domaine)
- La fonction anonyme retourne la vue (ici «welcome»)
  ↪ Vue située dans le répertoire `resources/views/`
- Accès : `localhost/exemple/`

Vue paramétrée

- Possible de récupérer des données (ici en GET)
- Passées à la vue lors de l'appel
- Accès : `localhost/exemple/article/2`

Contenu de `web.php`

```
Route::get('article/{n}', function($n) {
        return view('article')->with('numero', $n);
    });
```

Vue `article.php`

```
<!DOCTYPE html>
<html lang="fr">
  <head> ... </head>
  <body>
    <p>Ceci est l'article n°<?php echo $numero; ?></p>
  </body>
</html>
```

Utilisation de *blade*

- Moteur de template permettant de simplifier l'écriture
- La vue doit être nommée `XXX.blade.php`
  ↪ Dans le cas contraire, les directives *blade* sont ignorées

Sans *blade*

```
<p>Ceci est l'article n°<?php echo $numero; ?></p>
```

Avec *blade*

```
<p>Ceci est l'article n°{{$numero}}</p>
```

Les templates avec *blade*

- Définition du template et description des champs attendus
  ↪ `@yield('contenu')`, par exemple
- La vue se contente de donner le contenu des champs du template
  - `@extends('template')` : pour l'héritage du template
    ↪ Permet de spécifier le template choisi
  - `@section('contenu')` et `@endsection('contenu')` : définition du contenu du champ 'contenu'

Exemple : un template

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <title>@yield('titre')</title>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1
        ">
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/
        bootstrap/4.1.3/css/bootstrap.min.css">
  </head>
  <body>
    <div class="container">
      <div class="card">
        <div class="card-header bg-primary text-white text-center">
            @yield('titre')</div>
        <div class="card-body bg-light">@yield('contenu')</div>
      </div>
    </div>
  </body>
 </html>
```

Exemple : la vue qui exploite le template

```
@extends('template')

@section('titre')
Les articles
@endsection

@section('contenu')
<p>C'est l'article n°{{$numero}}</p>
@endsection
```

Définition de contrôleurs

- Construction automatique avec `artisan` :
  `php artisan make:controller ArticleController`
- Création d'une classe vide dans le répertoire
  `app/http/controllers`

Exemple avec la création d'une méthode `show`

```php
<?php
namespace App\Http\Controllers;

use Illuminate\Http\Request;

class ArticleController extends Controller
{
    public function show($n) {
        return view('article')->with('numero', $n);
    }
}
```

Liaison d'un contrôleur avec le routeur

- Même principe qu'avec une vue quelconque...
- ...sans la définition d'une fonction anonyme
  ↪ Le nom du contrôleur suivi du nom de la fonction
- Accès : `localhost/exemple/article/2`

```
Route::get('article/{n}',
           'ArticleController@show')->where('n', '[0-9]+');
```

Table des matières

Fonctionnalités autour de la base de données

- *Laravel* permet de :
  - Créer les tables de la base de données (migration)
  - Les remplir (*seeder*)
  - Construire des requêtes (*query builder*)
- Outil *Eloquent* (ORM) :
  - Une classe par table
  - Lire et enregistrer les données

Les migrations

- Pour construire une nouvelle table :
  `php artisan make:migration actualites`
- Plusieurs paramètres :
  - `--create=nom_table` : pré-remplit la migration avec le code nécessaire
  - `--table=nom_table` : pré-remplit une migration à partir d'une table existante

Contenu du fichier de la migration

```php
<?php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class Users extends Migration
{
    /**
     * Run the migrations.
     * @return void
     */
    public function up()
    {
    }

    /**
     * Reverse the migrations.
     * @return void
     */
    public function down()
    {
    }
}
```

Exemple de migration

```
public function up()
{
  Schema::create('actualites', function (Blueprint $table) {
      $table->increments('id');
      $table->string('titre', 100);
      $table->text('message');
      $table->datetime('date');
      $table->timestamps();
    });
}
public function down()
{
    Schema::dropIfExists('actualites');
}
```

Spécification des champs

- Entiers :
  ↪ `integer`, `tinyInteger`, etc.
- Chaînes de caractères : `string(colName, length)`
  ↪ `VARCHAR(length)`
- Date : `datetime`, `timestamp`, `time`
- Texte : `text`, `longText`
- Propriétés supplémentaires :
  - `nullable`
  - `default(valeur)`
  - `unsigned`
  - `first` ou `after` pour placer les colonnes (*MySQL*)
  - `unique`, `primary` et `index`

Exécuter les migrations

- `php artisan migrate`
  ↪ Crée les tables
- Possible de revenir en arrière en cas d'erreur
  ↪ `php artisan migrate:rollback`

Peupler la base

- Création du modèle associé aux actualités :
  ↪ `php artisan make:model Actualite`
  ↪ Création d'une classe `Actualite.php` dans le répertoire `app`
- Création d'un *seeder* (pour peupler la base) :
  ↪ `php artisan make:seed ActualiteTableSeeder`
  ↪ Création d'une classe dans le répertoire `database/seeds`
- Pour exécuter le peuplement : `php artisan db:seed`
  ↪ Appelle par défaut le *seeder* `DatabaseSeeder.php`
- Possible d'utiliser le *faker*
  ↪ Génération de valeurs aléatoires
  ↪ Par exemple : nom/prénom aléatoires, adresses, dates, *etc.*

Exemple de *seeder*

Méthode `run` de `ActualiteTableSeeder`

```
DB::table('actualites')->truncate();

App\Actualite::create([
    'titre' => 'Actualité␣1',
    'message' => "Ceci␣est␣l'actualité␣numéro␣1.␣C'est␣cool␣!!!",
    'date' => '2020-09-26'
  ]);
```

Méthode `run` de `DatabaseSeeder`

```
$this->call(ActualiteTableSeeder::class);
```

Table des matières

Conclusion

- *Laravel* est un *framework* PHP très puissant
- Simplifie la vie des développeurs mais également la vie de l'application
  ↪ Conception, maintenance...
- Nécessite de comprendre ce que l'on fait !
  ↪ Aspects administration et développeur
- Beaucoup d'autres fonctionnalités :
  - Nous en découvrirons en TP
  - Consultez la documentation :
    ↪ `https://laravel.com/docs/8.x`