Architecture WEB

Faire du Web, mais pour de vrai

Organisation du module

- 6 heures de cours
- 6 heures de TD
- 18 heures de TP
- => un mode « projet »
- => un exam avec questions de cours + petits exercices

Contenu du cours

- Qu'est-ce que le Web / HTTP
- Principe génériques des Frameworks web
- Application à Symfony

Un Framework, c'est quoi ?

- Un framework est un ensemble d'outils et de composants logiciels organisés conformément à un plan d'architecture et des patterns, l'ensemble formant ou promouvant un « squelette » de programme. Il est souvent fourn sous la forme d'une bibliothèque logicielle, et accompagné du plan de l'architecture cible du framework.

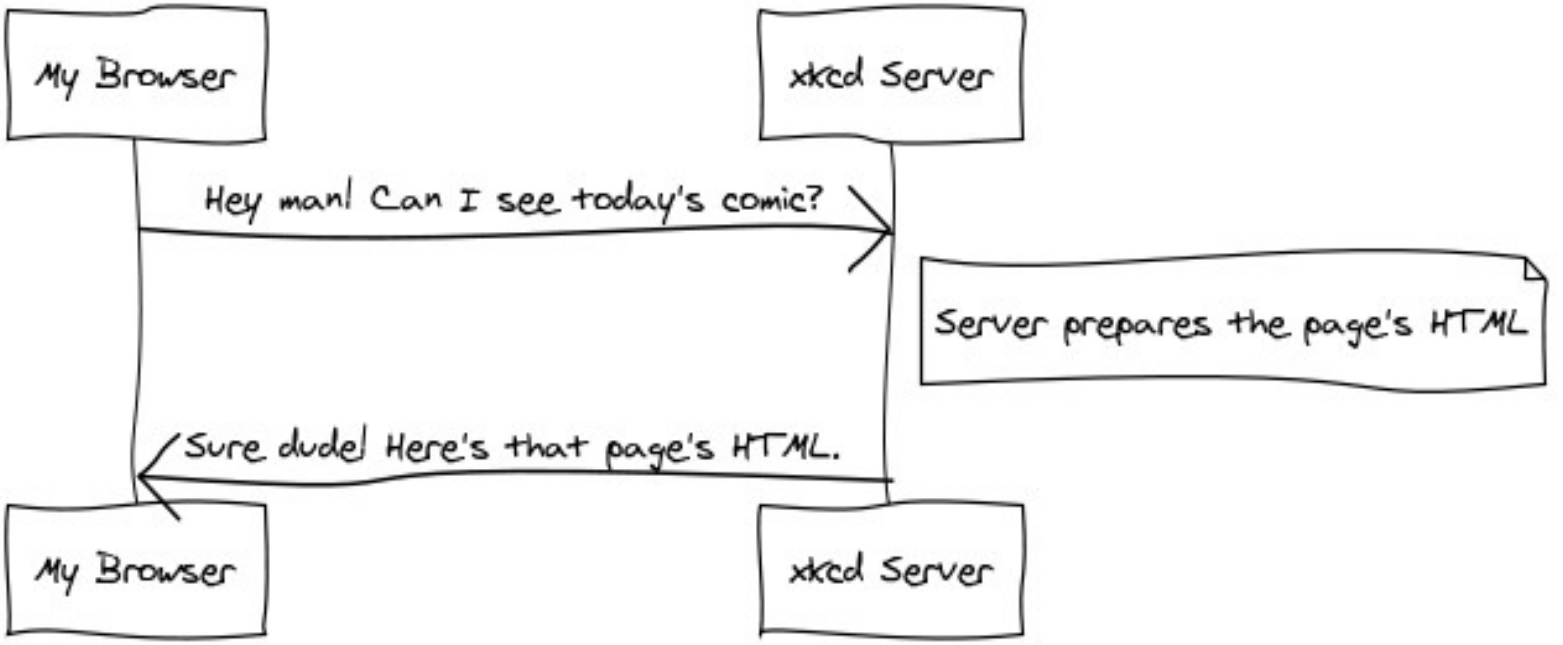
Et un Framework Web ?

- Gère les spécificités du Web
- « Professionnalise » le développement
- Développer pour le Web sans Framework = Coder du Java sans les librairies de base
- => Avant de parler de Framework, nous devon parler du Web et du protocole HTTP

Le protocole HTTP

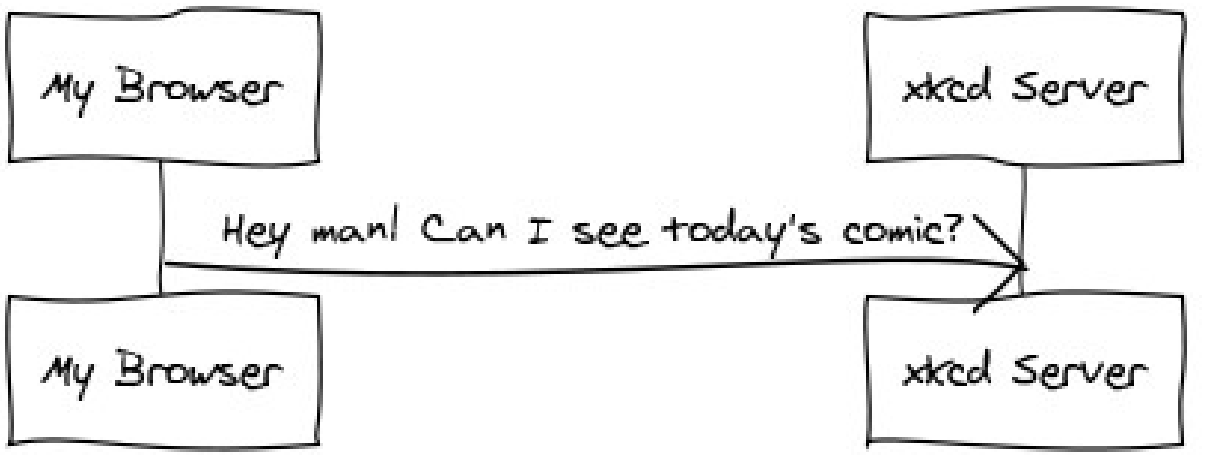
Un protocole simple

- Protocole texte
- On envoie une requête
- On reçoit une réponse



La requête

- Toute conversation sur le Web commence par une requête
- Une requête est un message texte créé par le client et envoyé au serveur, le tout, en langage HTTP



La requête en détails

```
GET / HTTP/1.1
Host: xkcd.com
Accept: text/html
User-Agent: Mozilla/5.0 (Macintosh)
```

- Première ligne est la plus importante :
  - La méthode HTTP
  - L'URI
  - La version du protocole

Les méthodes HTTP

- GET : Récupérer une ressource du serveur
- POST : Créer une ressource sur le serveur
- PUT : Mettre à jour une ressource sur le serveur
- DELETE : Effacer une ressource sur le serveu

  => Il en existe 9 au total

```
DELETE /blog/15 HTTP/1.1
```

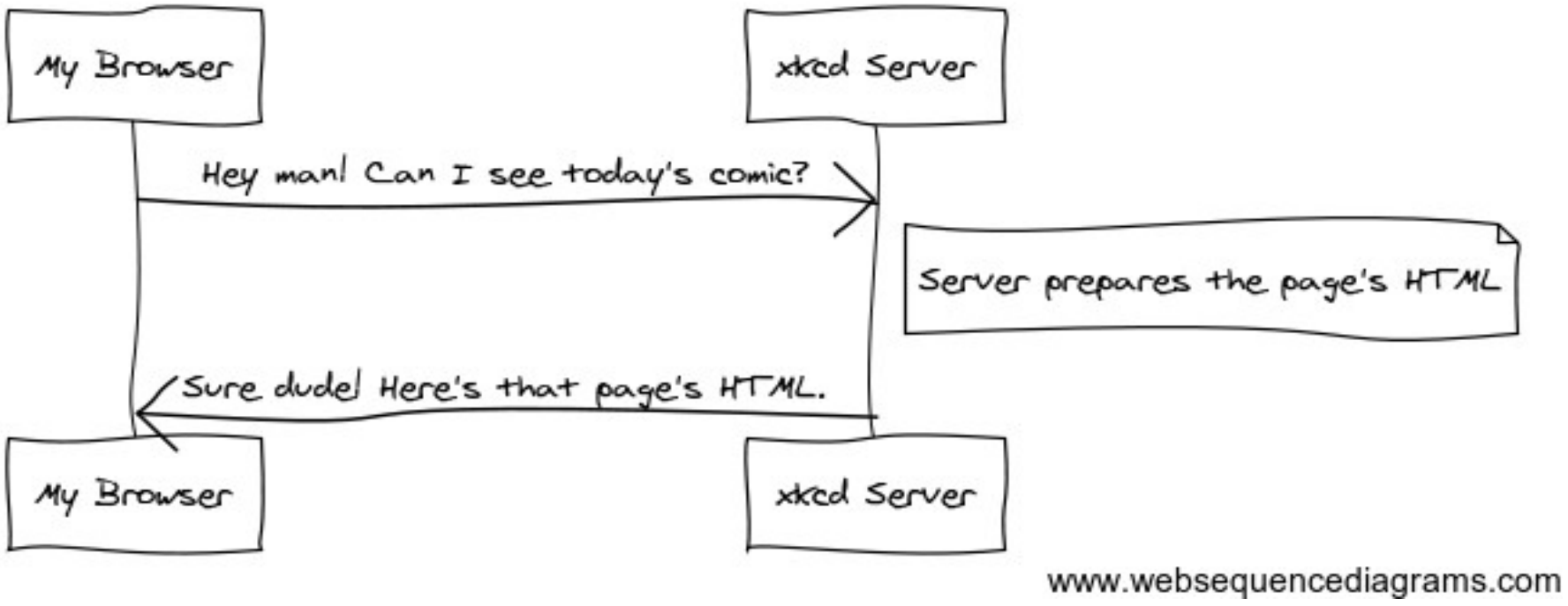
Les headers

- Contient le reste de la requête
  - L'hôte demandé (Host)
  - Le format de réponse accepté (Accept)
  - L'application utilisée pour réaliser la requête (User Agent)

```
GET / HTTP/1.1
Host: xkcd.com
Accept: text/html
User-Agent: Mozilla/5.0 (Macintosh)
```

La réponse

- Grâce à l'URI et à la méthode, le serveur sait ce qu'il doit faire avec la requête



Contenu de la réponse

```
HTTP/1.1 200 OK
Date: Sat, 02 Apr 2011 21:05:05 GMT
Server: lighttpd/1.4.19
Content-Type: text/html

<html>
  <!-- ... HTML for the xkcd comic -->
</html>
```

- La première ligne est très importante : elle contient le « status code HTTP »
  - 200 : ok
  - 404 : non trouvé
  - 500 : erreur serveur
  - ...

Les headers

- Comme pour la requête, la réponse contient des headers
  - Content-type : HTML, JSON, XML, …
  - Server
  - Date
  - …
- Peuvent être utilisés pour des systèmes de cache

Recap

- Peu importe le langage / framework que vous utilisez côté serveur, le but d'une telle application est toujours de :
  - Comprendre une requête
  - Retourner la réponse appropriée

Exemple en PHP

```php
<?php
$uri = $_SERVER['REQUEST_URI'];
$foo = $_GET['foo'];

header('Content-Type: text/html');
echo 'The URI requested is: '.$uri;
echo 'The value of the "foo" parameter is: '.$foo;
```

- PHP utilise des super-globales pour lire le contenu de la requête
- Header + echo sont utilisées pour créer la réponse

Réponse générée

```
HTTP/1.1 200 OK
Date: Sat, 03 Apr 2011 02:14:33 GMT
Server: Apache/2.2.17 (Unix)
Content-Type: text/html

The URI requested is: /testing.php?foo=symfony
The value of the "foo" parameter is: symfony
```

Abstraction

- PHP est une exception : il a été « conçu » pour le web
- Mais d'autres langages « non web » sont aussi utilisés sur le Web (python, ruby, …)
- Tous les Framework Web proposent une abstraction des requêtes et réponses HTTP

  => Une fois la partie HTTP abstraite, on code dans le langage de façon classique

Exemple d'abstraction d'une Request

```php
<?php
use Symfony\Component\HttpFoundation\Request;

$request = Request::createFromGlobals();

// the URI being requested (e.g. /about) minus any query parameters
$request->getPathInfo();

// retrieve GET and POST variables respectively
$request->query->get('foo');
$request->request->get('bar', 'default value if bar does not exist');

// retrieve SERVER variables
$request->server->get('HTTP_HOST');

// retrieves an instance of UploadedFile identified by foo
$request->files->get('foo');

// retrieve a COOKIE value
$request->cookies->get('PHPSESSID');

// retrieve an HTTP request header, with normalized, lowercase keys
$request->headers->get('host');
$request->headers->get('content_type');

$request->getMethod();    // GET, POST, PUT, DELETE, HEAD
$request->getLanguages(); // an array of languages the client accepts
```

Exemple d'abstraction d'une Response

```php
<?php
use Symfony\Component\HttpFoundation\Response;

$response = new Response();

$response->setContent('<html><body><h1>Hello world!</h1></body></html>');
$response->setStatusCode(Response::HTTP_OK);
$response->headers->set('Content-Type', 'text/html');

// prints the HTTP headers followed by the content
$response->send();
```

Organiser son code : comment écrire du code maintenable ?

Organisation traditionnelle

- Traditionnellement, une page = un fichier
  - index.php
  - contact.php
  - blog.php
- Obligé de faire plein d'include
- Impossible de renommer les fichiers
- Très peu modulable
- La réponse : utiliser un « Front Controller »

Le Front Controller

- Fichier unique où arrivent toutes les requêtes
  - /index.php => index.php
  - /index.php?rub=contact => index.php
  - /index.php?rub=blog => index.php
- Ou avec mod_rewrite
  - /index.php => index.php
  - /index.php/contact => index.php
  - /index.php/blog => index.php

Le Front Controller

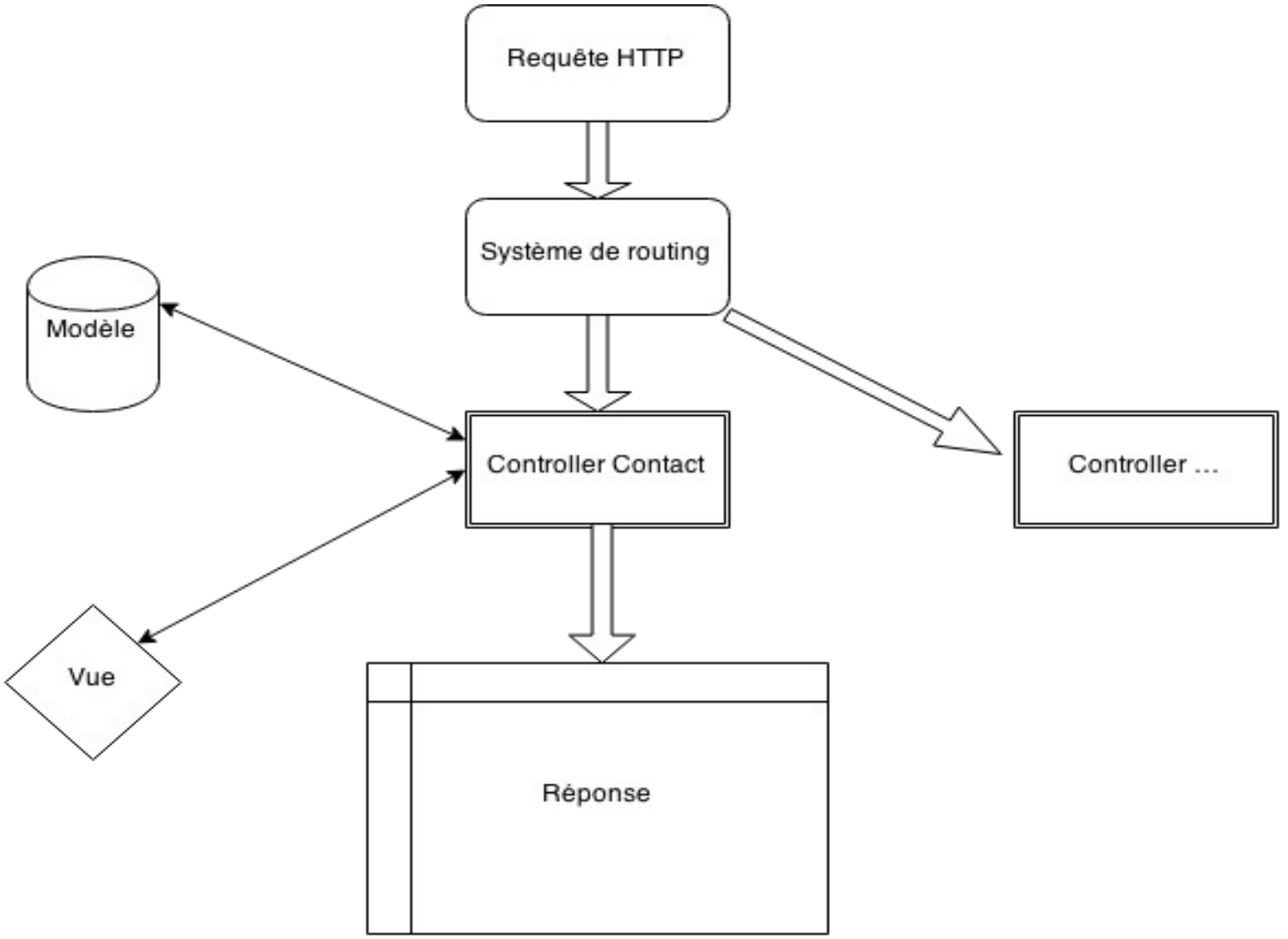
- /index.php est toujours exécuté
- Le routing est géré en interne : plus facile à maintenir
- Toutes les applications modernes utilisent ce principe

Organiser le controller

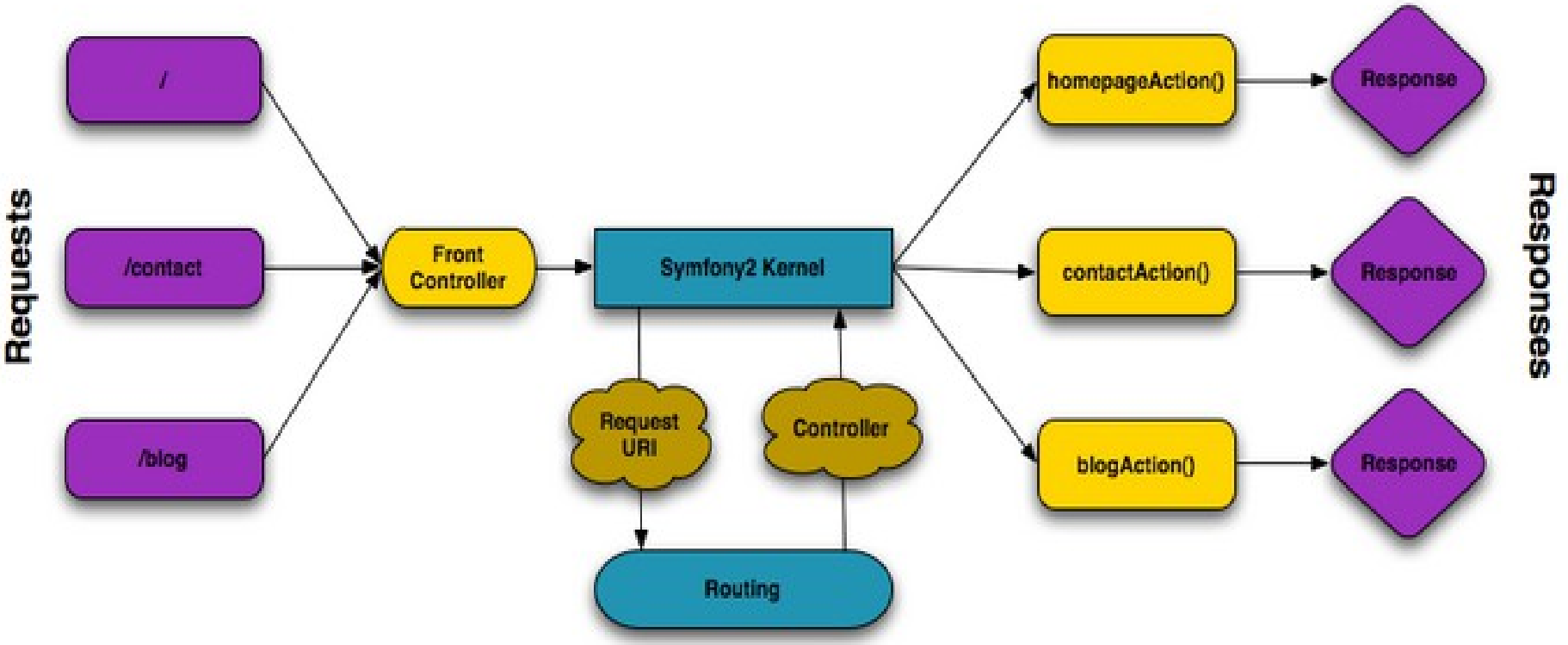
```php
<?php
// index.php
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

$request = Request::createFromGlobals();
$path = $request->getPathInfo(); // the URI path being requested

if (in_array($path, array('', '/'))) {
    $response = new Response('Welcome to the homepage.');
} elseif ('/contact' === $path) {
    $response = new Response('Contact us');
} else {
    $response = new Response('Page not found.', Response::HTTP_NOT_FOUND);
}
$response->send();
```

- Deviens vite difficile à gérer
- Les Frameworks résolvent ce souci

Le Modèle MVC



Par exemple avec Symfony2



Symfony2

Frameworks Web

- Rails => Ruby
- Django => Python
- Symfony => PHP

=> Nous allons étudier Symfony2, mais les principes généraux restent les mêmes

Les composants Symfony2

- HttpFoundation
- Routing
- Form
- Validator
- Templating
- Security
- Translation

De PHP à Symfony2 : Qu'est-ce qu'apporte l'utilisation d'un Framework ?

Un blog en PHP

```php
<?php
// index.php
$link = mysql_connect('localhost', 'myuser', 'mypassword');
mysql_select_db('blog_db', $link);
$result = mysql_query('SELECT id, title FROM post', $link);
?>

<!DOCTYPE html>
<html>
    <head>
        <title>List of Posts</title>
    </head>
    <body>
        <h1>List of Posts</h1>
        <ul>
            <?php while ($row = mysql_fetch_assoc($result)): ?>
            <li>
                <a href="/show.php?id=<?php echo $row['id'] ?>">
                    <?php echo $row['title'] ?>
                </a>
            </li>
            <?php endwhile ?>
        </ul>
    </body>
</html>

<?php
mysql_close($link);
?>
```

Séparons la « vue »

```php
<?php
// index.php
$link = mysql_connect('localhost', 'myuser', 'mypassword');
mysql_select_db('blog_db', $link);

$result = mysql_query('SELECT id, title FROM post', $link);

$posts = array();
while ($row = mysql_fetch_assoc($result)) {
    $posts[] = $row;
}

mysql_close($link);

// include the HTML presentation code
require 'templates/list.php';
```

Le fichier « vue »

```
<!DOCTYPE html>
<html>
    <head>
        <title>List of Posts</title>
    </head>
    <body>
        <h1>List of Posts</h1>
        <ul>
            <?php foreach ($posts as $post): ?>
            <li>
                <a href="/read?id=<?php echo $post['id'] ?>">
                    <?php echo $post['title'] ?>
                </a>
            </li>
            <?php endforeach ?>
        </ul>
    </body>
</html>
```

Séparons le « modèle »

```php
<?php
// model.php
function open_database_connection()
{
    $link = mysql_connect('localhost', 'myuser', 'mypassword');
    mysql_select_db('blog_db', $link);

    return $link;
}

function close_database_connection($link)
{
    mysql_close($link);
}

function get_all_posts()
{
    $link = open_database_connection();

    $result = mysql_query('SELECT id, title FROM post', $link);
    $posts = array();
    while ($row = mysql_fetch_assoc($result)) {
        $posts[] = $row;
    }
    close_database_connection($link);

    return $posts;
}
```

Le nouveau controller

```php
<?php
require_once 'model.php';

$posts = get_all_posts();

require 'templates/list.php';
```

Réutiliser le layout

- Il reste encore un souci : le design général doi
  être répété dans chaque fichier « vue »
- Essayons de réorganiser cela

Réécriture du layout

```
<!-- templates/layout.php -->
<!DOCTYPE html>
<html>
    <head>
        <title><?php echo $title ?></title>
    </head>
    <body>
        <?php echo $content ?>
    </body>
</html>
```

Réécriture du fichier list.php

```php
<?php $title = 'List of Posts' ?>

<?php ob_start() ?>
    <h1>List of Posts</h1>
    <ul>
        <?php foreach ($posts as $post): ?>
        <li>
            <a href="/read?id=<?php echo $post['id'] ?>">
                <?php echo $post['title'] ?>
            </a>
        </li>
        <?php endforeach ?>
    </ul>
<?php $content = ob_get_clean() ?>

<?php include 'layout.php' ?>
```

Résultat

- Pas de duplication de code (DRY)
- Code organisé
- Facile de créer d'autres pages

  => Créons une nouvelle page pour afficher un article grâce à son id

Modifions le modèle

```php
<?php
// model.php
function get_post_by_id($id)
{
    $link = open_database_connection();

    $id = intval($id);
    $query = 'SELECT date, title, body FROM post WHERE id = '.$id;
    $result = mysql_query($query);
    $row = mysql_fetch_assoc($result);

    close_database_connection($link);

    return $row;
}
```

Créons le nouveau contrôleur

```php
<?php
require_once 'model.php';

$post = get_post_by_id($_GET['id']);

require 'templates/show.php';
```

Créons le template

```php
<?php $title = $post['title'] ?>

<?php ob_start() ?>
    <h1><?php echo $post['title'] ?></h1>

    <div class="date"><?php echo $post['date'] ?></div>
    <div class="body">
        <?php echo $post['body'] ?>
    </div>
<?php $content = ob_get_clean() ?>

<?php include 'layout.php' ?>
```

Résultat

- Pas de code dupliqué
- Mais de nouveaux problèmes :
  - S'il n'y a pas de paramètre **id** ?
  - Et si on oublie le **intval** ?
  - On doit inclure le **model.php** dans chaque contrôleur
  - Et si on devait inclure d'autres fichiers ?
  - …

Utilisation d'un contrôleur principal

Sans contrôleur principal
/index.php => Page de liste des articles (index.php est exécuté)
/show.php => Page d'affichage d'un article (show.php est exécuté)

Avec index.php comme contrôleur principal
/index.php => Page de liste des articles (index.php est exécuté)
/index.php/show => Page d'affichage d'un article (index.php est exécuté)

Création d'un contrôleur principal

```php
<?php
// index.php

// load and initialize any global libraries
require_once 'model.php';
require_once 'controllers.php';

// route the request internally
$uri = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
if ('/index.php' == $uri) {
    list_action();
} elseif ('/index.php/show' == $uri && isset($_GET['id'])) {
    show_action($_GET['id']);
} else {
    header('Status: 404 Not Found');
    echo '<html><body><h1>Page Not Found</h1></body></html>';
}
```

Le fichier controllers

```php
<?php
function list_action()
{
    $posts = get_all_posts();
    require 'templates/list.php';
}

function show_action($id)
{
    $post = get_post_by_id($id);
    require 'templates/show.php';
}
```

Résumé

- On a une meilleure structure, mais qui pourrait être amélioré
- Index.php a maintenant un nouveau rôle => il gère les requêtes mais pas parfaitement
- Et si on devait gérer les formulaires ?
- La sécurité ?
- L'authentification

Ajoutons du Symfony2

Le contrôleur principal

```php
<?php
// index.php
require_once 'vendor/autoload.php';

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

$request = Request::createFromGlobals();

$uri = $request->getPathInfo();
if ('/' == $uri) {
    $response = list_action();
} elseif ('/show' == $uri && $request->query->has('id')) {
    $response = show_action($request->query->get('id'));
} else {
    $html = '<html><body><h1>Page Not Found</h1></body></html>';
    $response = new Response($html, Response::HTTP_NOT_FOUND);
}

// echo the headers and send the response
$response->send();
```

Les contrôleurs

```php
<?php
// controllers.php
use Symfony\Component\HttpFoundation\Response;

function list_action()
{
    $posts = get_all_posts();
    $html = render_template('templates/list.php', array('posts' => $posts));

    return new Response($html);
}

function show_action($id)
{
    $post = get_post_by_id($id);
    $html = render_template('templates/show.php', array('post' => $post));

    return new Response($html);
}

// helper function to render templates
function render_template($path, array $args)
{
    extract($args);
    ob_start();
    require $path;
    $html = ob_get_clean();

    return $html;
}
```

Le même code en Symfony

```php
<?php
// src/AppBundle/Controller/BlogController.php
namespace AppBundle\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class BlogController extends Controller
{
    public function listAction()
    {
        $posts = $this->get('doctrine')
            ->getManager()
            ->createQuery('SELECT p FROM AcmeBlogBundle:Post p')
            ->execute();

        return $this->render('Blog/list.html.php', array('posts' => $posts));
    }

    public function showAction($id)
    {
        $post = $this->get('doctrine')
            ->getManager()
            ->getRepository('AppBundle:Post')
            ->find($id);

        if (!$post) {
            // cause the 404 page not found to be displayed
            throw $this->createNotFoundException();
        }

        return $this->render('Blog/show.html.php', array('post' => $post));
    }
}
```

Le Templating de base

```php
<!-- app/Resources/views/Blog/list.html.php -->
<?php $view->extend('layout.html.php') ?>

<?php $view['slots']->set('title', 'List of Posts') ?>

<h1>List of Posts</h1>
<ul>
    <?php foreach ($posts as $post): ?>
    <li>
        <a href="<?php echo $view['router']->generate(
            'blog_show',
            array('id' => $post->getId())
        ) ?>">
            <?php echo $post->getTitle() ?>
        </a>
    </li>
    <?php endforeach ?>
</ul>
```

```
<!-- app/Resources/views/layout.html.php -->
<!DOCTYPE html>
<html>
    <head>
        <title><?php echo $view['slots']->output(
            'title',
            'Default title'
        ) ?></title>
    </head>
    <body>
        <?php echo $view['slots']->output('_content') ?>
    </body>
</html>
```

Le Templating amélioré avec Twig

```
{# app/Resources/views/Blog/list.html.twig #}
{% extends "layout.html.twig" %}

{% block title %}List of Posts{% endblock %}

{% block body %}
    <h1>List of Posts</h1>
    <ul>
        {% for post in posts %}
        <li>
            <a href="{{ path('blog_show', {'id': post.id}) }}">
                {{ post.title }}
            </a>
        </li>
        {% endfor %}
    </ul>
{% endblock %}
```

```
{# app/Resources/views/layout.html.twig #}
<!DOCTYPE html>
<html>
    <head>
        <title>{% block title %}Default title{% endblock %}</title>
    </head>
    <body>
        {% block body %}{% endblock %}
    </body>
</html>
```

Le routing

```
# app/config/routing.yml
blog_list:
    path:     /blog
    defaults: { _controller: AppBundle:Blog:list }

blog_show:
    path:     /blog/show/{id}
    defaults: { _controller: AppBundle:Blog:show }
```

Le noyau

```php
<?php
// web/app.php
require_once __DIR__.'/../app/bootstrap.php';
require_once __DIR__.'/../app/AppKernel.php';

use Symfony\Component\HttpFoundation\Request;

$kernel = new AppKernel('prod', false);
$kernel->handle(Request::createFromGlobals())->send();
```

Les points forts de Symfony

- Code clair, organisé et réutilisable
- Pas besoin de coder les tâches de bas niveau
- Inclusion d'outils Open Source : Doctrine, SwiftMailer, Templating, ...
- URLs totalement flexibles
- Système de cache puissant