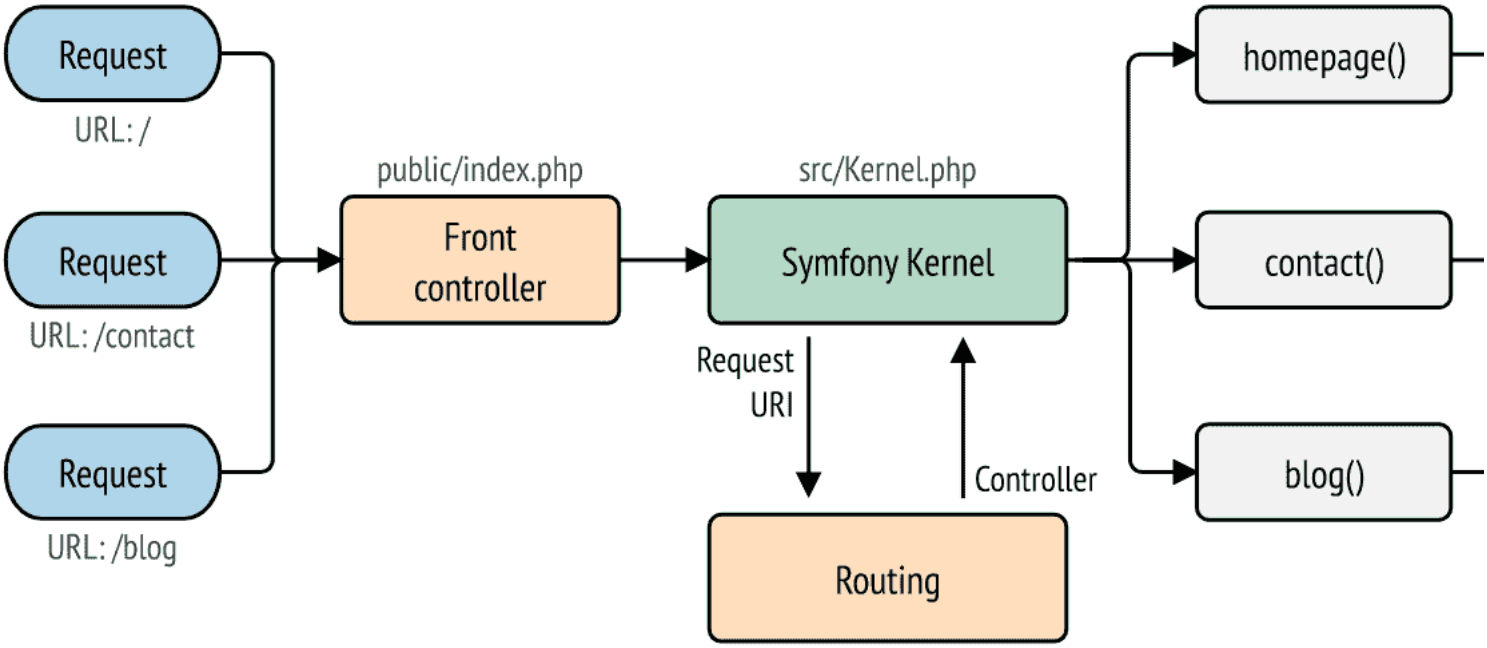
5 minutes

Fonctionnement d'une application Symfony

Nous allons brièvement voir dans l'ordre les éléments qui gèrent une requête `HTTP` entrante d'un client :



Le `script index.php`

Lorsqu'une requête `HTTP` est reçue par le serveur `Web` (par exemple `NGINX`) puis transmise à `PHP-FPM` (grâce au protocole `FastCGI`), le premier script `PHP` à être exécuté est `public/index.php`.

On l'appelle point d'entrée ou `Front controller`.

Le `front controller` est le contrôleur qui gère **l'ensemble des requêtes d'une application**. Autrement dit, toutes les requêtes `HTTP` entraînent obligatoirement l'exécution de ce `script`.

Si vous avez suivi le cours `PHP` vous noterez que c'est déjà une différence avec une application `PHP` basique qui exécute un script différent suivant la requête `HTTP` et qui n'a donc pas un seul point d'entrée pour l'ensemble des requêtes.

Voici son contenu :

```php
<?php

use App\Kernel;

require_once dirname(__DIR__).'/vendor/autoload_runtime.php';

return function (array $context) {
    return new Kernel($context['APP_ENV'], (bool) $context['APP_DEBUG']);
};
```

Ce script commence par initialiser la fonctionnalité du chargement automatique en fonction de l'environnement (`autoload`).

Ce script crée ensuite une instance de la classe `Kernel`.

La classe `Kernel`

La classe `Kernel` est chargée par le `script index.php`, elle se trouve dans `src/kernel.php`.

Voici son contenu :

```php
<?php

namespace App;
```

```
use SymfonyBundleFrameworkBundleKernelMicroKernelTrait;
use SymfonyComponentHttpKernelKernel as BaseKernel;

class Kernel extends BaseKernel
{
    use MicroKernelTrait;
}
```

`Symfony\Component\HttpKernel\Kernel` qui est ici renommée en `BaseKernel` est le cœur de `Symfony`. Elle va initialiser tous les `bundles Symfony` avec la configuration de votre application.

Vous pouvez aller regarder les fonctions d'initialisation des `bundles` dans `dymaproject/vendor/symfony/http-kernel/Kernel.php`

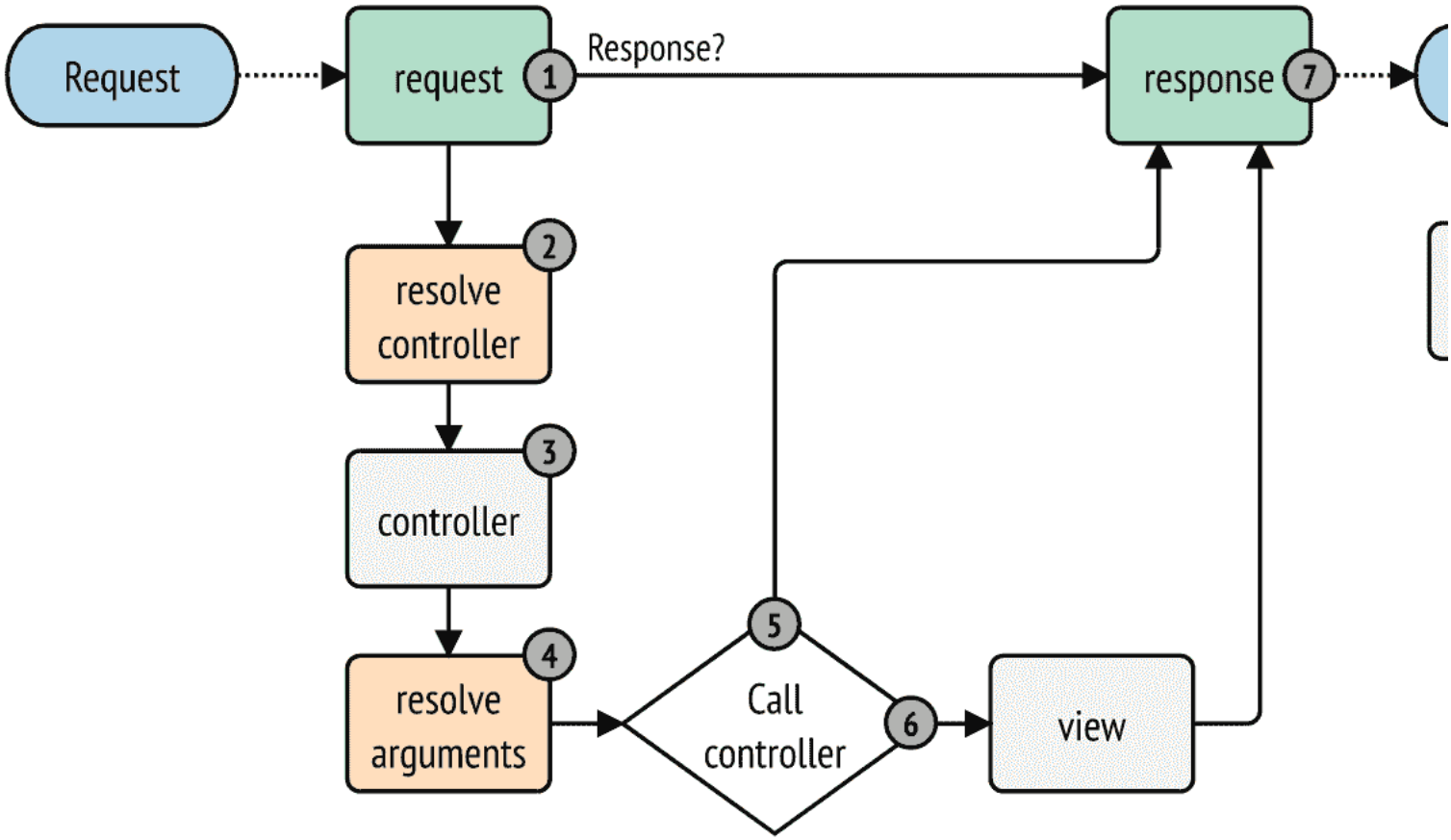
Vous y trouverez notamment une fonction `initializeBundles()` qui est chargée de cette initialisation.

La fonction `handle()`

Une fois toutes les configurations, le routing et les `bundles` chargés, la requête est gérée par la fonction `handle()` de la classe `HttpKernel`.

Cette fonction a pour objectif de partir d'une requête `HTTP` et de la transformer en la réponse `HTTP` attendue.

C'est donc une fonction très importante dont le schéma suivant résume son fonctionnement :



Cette fonction va envoyer des événements qui sont ensuite gérés par des gestionnaires d'événements.

Nous allons voir les grandes étapes du processus :

1) Un événement `kernel.request` est émis. Plusieurs gestionnaires sont appelés à ce niveau. Par exemple, le gestionnaire du `bundle Security` qui va déterminer si la requête est autorisée ou non. La requête peut être transformée en réponse `403` (pour `forbidden`) et retournée à ce moment. Un autre gestionnaire important pour cet événement est le `Router`, il va déterminer quel contrôleur doit être appelé en fonction de la requête.

2) Le Contrôleur est résolu. La fonction `handle()` va appeler une fonction `getController()` (située dans `dymaproject/vendor/symfony/http-kernel/Controller/ControllerResolver.php`) qui va être chargé de récupérer et d'exécuter le bon contrôleur en fonction de la propriété `_controller` qui a été placé par le `Router` sur le tableau associatif `Request`.

Vous pouvez voir cette mécanique au début de la fonction `getController()` :

```
<?php
public function getController(Request $request) {
  if (!$controller = $request->attributes->get('_controller')) {
```

```
    if (null !== $this->logger) {
      $this->logger->warning('Unable to look for the controller as the "_controller" parameter is missing.');
    }
    return false;
  }
}
```

3) Un événement `kernel.controller` est émis. Il permet d'exécuter des gestionnaires d'événement juste avant que le contrôleur récupéré ne soit exécuté. C'est un `design pattern` commun appelé `hooks` en programmation. Cela permet d'exécuter du code lors d'événements clés.

4) Récupération des arguments pour le contrôleur. Avant l'exécution du contrôleur, une fonction `getArguments()` va récupérer les arguments à passer au contrôleur en fonction de l'environnement et de la configuration.

5) Exécution du contrôleur. Le contrôleur, qui est chargé de construire la réponse à renvoyer au client, est exécuté. Il va créer une réponse qui peut être une page `HTML`, une réponse au format `JSON` ou toute autre réponse `HTTP` valide. C'est à cette étape que sera exécuté votre code : votre contrôleur, qui va éventuellement appeler vos modèles et vos vues.

6) Un événement `kernel.view` est émis. Permet de gérer les cas où aucune réponse n'est retournée par le contrôleur. Par défaut, `Symfony` n'a aucune gestionnaire pour cet événement et vos contrôleurs doivent obligatoirement retourner une réponse. Certains `bundles` utilisent cet événement, comme par exemple `FOSRestBundle`.

7) Un événement `kernel.response` est émis. Permet d'exécuter des gestionnaires juste avant que la réponse ne soit envoyée au client. Des exemples d'utilisation sont la modification des `headers` de la réponse, l'ajout de `cookies` etc.

8) Un événement `kernel.terminate` est émis. La réponse a été envoyée au client. Cet événement permet de procéder aux nettoyages ou à des tâches pouvant être réalisées après la réponse `HTTP` (par exemple, enregistrement de données d'analyse, envoi d'emails etc).