Support de Cours : Les Événements en Symfony 7

1. Introduction aux Événements en Symfony

Symfony utilise un système d'événements basé sur le **dispatcher d'événements** pour permettre une communication flexible entre différents composants sans couplage direct.

Le système repose sur trois éléments principaux :

- **Le Dispatcher** : Gère les événements et appelle les listeners/subscribers associés.
- **Les Événements** : Objets transportant des informations.
- **Les Listeners et Subscribers** : Fonctions ou classes qui réagissent à des événements.

Symfony fournit plusieurs événements natifs et permet la création d'événements personnalisés.

---

2. Événements Clés en Symfony

Symfony gère des événements tout au long du cycle de vie d'une requête HTTP. Voici quelques événements importants :

2.1. Cycle de vie de la requête

- `kernel.request` : Avant que le routeur ne détermine la route
- `kernel.controller` : Avant l'exécution du contrôleur
- `kernel.response` : Avant l'envoi de la réponse au client
- `kernel.terminate` : Après l'envoi de la réponse (utile pour des tâches lourdes en arrière-plan)
- `kernel.exception` : Lorsqu'une exception est levée

2.2. Événements de l'authentification (Security)

- `security.authentication.success` : Quand un utilisateur est authentifié avec succès
- `security.logout` : Lorsqu'un utilisateur se déconnecte

2.3. Doctrine (Base de données)

- `doctrine.orm.entity_manager` : Gestionnaire d'entités
- `doctrine.event_listener` : S'exécute avant ou après certaines actions (ex: `prePersist`, `postUpdate`)

---

3. Créer un Listener d'événement

Un **Listener** est une classe qui écoute un événement précis. Exemple : ajouter un log à chaque requête.

3.1. Création d'un Listener

Créons un listener pour `kernel.request` :

```
namespace App\EventListener;

use Symfony\Component\EventDispatcher\Attribute\AsEventListener;
use Symfony\Component\HttpKernel\Event\RequestEvent;
use Psr\Log\LoggerInterface;

class RequestListener
{
    private LoggerInterface $logger;

    public function __construct(LoggerInterface $logger)
    {
        $this->logger = $logger;
    }

    #[AsEventListener(event: 'kernel.request')]
    public function onKernelRequest(RequestEvent $event)
    {
        $request = $event->getRequest();
        $this->logger->info('Nouvelle requête : ' . $request->getPathInfo());
    }
}
```

Symfony détecte automatiquement la classe grâce à l'attribut `#[AsEventListener]`.

---

4. Créer un Subscriber d'événements

Un **Subscriber** permet d'écouter plusieurs événements en une seule classe.

4.1. Création d'un Subscriber

```
namespace App\EventSubscriber;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;
use Symfony\Component\HttpKernel\Event\ResponseEvent;
use Symfony\Component\HttpKernel\Event\ExceptionEvent;
use Symfony\Component\HttpKernel\KernelEvents;
use Psr\Log\LoggerInterface;

class KernelSubscriber implements EventSubscriberInterface
{
    private LoggerInterface $logger;

    public function __construct(LoggerInterface $logger)
    {
        $this->logger = $logger;
    }

    public static function getSubscribedEvents(): array
    {
        return [
            KernelEvents::RESPONSE => 'onKernelResponse',
            KernelEvents::EXCEPTION => 'onKernelException',
        ];
    }

    public function onKernelResponse(ResponseEvent $event)
```

```
    {
        $this->logger->info('Une réponse a été envoyée.');
    }

    public function onKernelException(ExceptionEvent $event)
    {
        $exception = $event->getThrowable();
        $this->logger->error('Une exception est survenue : ' .
$exception->getMessage());
    }
}
```

Ce subscriber gère les événements `kernel.response` et `kernel.exception`.

---

5. Créer un Événement Personnalisé

Parfois, il est utile de définir ses propres événements. Voici comment faire :

5.1. Définir un événement

Créons un événement qui se déclenche quand un utilisateur s'inscrit :

```
namespace App\Event;

use Symfony\Contracts\EventDispatcher\Event;
use App\Entity\User;

class UserRegisteredEvent extends Event
{
    public function __construct(private User $user) {}

    public function getUser(): User
    {
        return $this->user;
    }
}
```

5.2. Déclencher l'événement

Dans ton service (ex: `UserService`), déclenche l'événement après l'inscription :

```
use App\Event\UserRegisteredEvent;
use Symfony\Component\EventDispatcher\EventDispatcherInterface;

class UserService
{
    public function __construct(private EventDispatcherInterface $dispatcher) {}

    public function registerUser(User $user)
    {
        // Logique d'inscription
        $this->dispatcher->dispatch(new UserRegisteredEvent($user));
    }
}
```

5.3. Écouter cet événement

Crée un listener pour envoyer un email :

```
class SendWelcomeEmailListener
{
```

```
    public function __construct(private MailerInterface $mailer) {}

    #[AsEventListener(event: UserRegisteredEvent::class)]
    public function onUserRegistered(UserRegisteredEvent $event)
    {
        $user = $event->getUser();
        $this->mailer->sendWelcomeEmail($user);
    }
}
```

---

6. Conclusion

Les événements en Symfony 7 permettent de **découpler le code** et de réagir à des actions sans modifier le code source principal.