

Merci à Cédric Bertolini (tuteur de stage) de m'avoir accepté au sein de l'entreprise ainsi qu'à Hugo Lapierre qui a pu me mettre en contact avec mon tuteur.

RAPPORT DE STAGE

14/02/2022 -08/04/2022

RAPHAEL DUMAS

WIZAPLACE

2 Quai du Commerce, 69009 Lyon




Table des matières

1. Introduction & présentation	2
2. L'entreprise	2
3. Missions du stage & déroulé	3
a. Environnement de travail	3
b. Premier projet.....	Erreur ! Signet non défini.
c. Deuxième projet	4
d. Troisième projet	4
4. Annexes	6
a. Annexes du premier projet	6
b. Annexes du second projet	9
c. Annexes du troisième projet.....	12

1. Introduction & présentation

Je suis Raphaël DUMAS, étudiant en seconde année de BTS SIO (Service Informatique aux Organisations), faisant partie de l'option SLAM (Solution Matérielles & Application Métiers), à l'Institution des Chartreux à Lyon. Ce cursus nous demande au cours de ses 2 années de formation de réaliser un stage de minimum 4 semaines sur chaque année. Pour cette seconde année, j'ai donc réussi à décrocher un stage de 8 semaines, du 14 février au 8 avril 2022, au sein de la société Wizaplace en tant que développeur Back-End dans l'équipe dite « run ». Ce second stage me permettra donc d'acquérir encore plus de compétences, autant techniques qu'humaines, au sein du monde du travail, ainsi que de m'habituer à l'environnement de l'entreprise.

2. L'entreprise

La société [Wizaplace](#) est une start-up fondée en 2012 par Éric Alessandrini, ayant pour produit une solution SaaS (Software as a Service) de création de marketplace pour d'autres sociétés.

Elle compte actuellement plus de 60 employés, dont une bonne partie au développement. L'équipe run elle, sera celle en charge de l'administration, et donc de solutions internes à l'entreprise. Elle est composée de 8 personnes et est dirigée par mon tuteur de stage : Cédric Bertolini.

Généralement, les marketplaces vendues sont orientées vers un marché en B2B (Business to Business), mais elles peuvent également, en fonction de la demande convenir à des marchés B2C (Business to Consumer). Socle de chaque solution, le « Wizaplace Core » coordonne tous les processus et micro-services de la solution pour faire fonctionner chaque marketplace, notamment par l'utilisation de diverses APIs. Le « Core » peut également évoluer en fonction des besoins avec différents services supplémentaires pouvant être ajoutés. On y retrouve :

- Le PIM (Product Information Management) afin de pouvoir gérer le contenu de la marketplace, importer des données via API ou interface.
- La plateforme peut également permettre de gérer les ventes de produits & services sous forme d'abonnement incluant la fréquence de paiement, une période d'engagement, et le renouvellement)
- Les fournisseurs peuvent également définir pour les produits les zones dans lesquelles ils sont disponibles ou non.
- Le Workflow de validation, un outil permettant de gérer les utilisateurs et les commandes passées.
- Wizaplace permet à la marketplace d'inclure un module de paiement et de transaction via différents moyens, le tout au sein de l'Union Européenne.

3. Missions du stage & déroulé

a. Environnement de travail

Après mon installation et une visite des locaux, j'ai pu installer moi-même mon poste de travail en partant de 0.

Je dispose donc :

- D'une machine Linux (Ubuntu 20.04) configurée moi-même
- Les différents réseaux de l'entreprise pour communiquer (adresse email du domaine, Slack)
- Accès à l'organisation Wizaplace sur GitHub, me donnant accès (en lecture seulement) à différents projets, pour que je puisse comprendre l'architecture
- Un compte Atlassian (un éditeur de logiciels), me donnant accès à l'API d'administration et de gestion Jira.
- L'environnement de développement PHP Storm

b. Première mission

La première mission a été déterminée dès le premier jour : créer un nouveau projet sur l'IDE, y requêter l'API Jira, pour récupérer les différents groupes, et en fonction du groupe sélectionné, y afficher sur une nouvelle page avec une url type `/group/{groupid}`, les rôles dont fait partie ce groupe, et les projets auxquels ces rôles ont accès. Ce projet s'est décomposé de la manière suivante :

- Mise en place de l'environnement PHP Symfony (celui utilisé par l'entreprise)
- Lecture de documentation Symfony & Jira
- Mise en place de commandes (via des commandes internes à Symfony « bin/console make:command ») afin d'afficher dans la console les données récupérées par les différents liens de l'API.
- Création d'une première page avec la liste des groupes via un contrôleur (« bin/console make:controller GroupController »).
- Création d'un service dans Symfony pouvant requêter l'API
- Restructuration du projet : inversion de l'affichage des groupes et des rôles.
- Complications : mise en place d'un schéma relation de données pour éclaircir le projet (voir annexes)
- Création de la seconde page prenant l'ID d'un groupe en paramètre.
- Présentation d'un problème : le temps de requêtage de l'API est bien trop long (peux aller au-delà de 2 minutes avant d'afficher un page)
Ce problème mènera donc au second projet, apportant la solution.

c. Deuxième mission

Cette deuxième mission avait pour but de mettre en place un docker, une base de données locale (dans le docker) et pouvoir synchroniser l'API avec la base de données, afin de pouvoir requêter la base de données plutôt que l'API, et donc réduire le temps de requête et d'affichage lors de la navigation.

- Lecture de documentation puis installation de docker sur le poste
- Initialisation d'une base de données dans le docker grâce aux commandes internes de « Doctrine » (« bin/console dump:config doctrine » & « doctrine:database:create »)
- Création des différentes entités (Groups, Actors, Roles, Projects) avec les relations « Many to Many » entre chaque table (voir schéma relationnel en annexe)
- Exécuter les migrations sur la base de données pour générer la structure
- Créer des commandes Symfony pour synchroniser l'API à la base de données et ainsi y insérer les données nécessaires
- Modifier les contrôleurs afin d'utiliser la base de données plutôt que l'API

d. Troisième mission

Cette troisième mission s'inscrit dans la continuité des deux autres, mais en utilisant une méthodologie de programmation particulière : la méthode MoSCoW. Cette méthode peut être résumée par une hiérarchisation des différentes tâches du projet, des plus importantes aux moins importantes. Ce projet avait 4 tâches primordiales, et pour chacune, des améliorations, non-essentiels, mais néanmoins toujours pratiques qui pourraient être ajoutées si tout se déroule bien.

- Créer une page d'affichage des groupes (voir 1ère mission) ainsi qu'une page d'affichage des projets. Pour chacune, il fallait créer une route de détail qui s'affiche lorsque l'on clique sur un projet ou un groupe.
- Pour les groupes ajouter des paramètres de tri via l'URL
- Pour un groupe, afficher les utilisateurs en faisant partie
- Pour les projets, y afficher des paramètres de configuration, leur type ; leur style, ainsi qu'y ajouter des filtres via l'URL.
- Pour chaque projet ajouter un lien renvoyant vers la page du projet sur le domaine Jira de l'entreprise.

e. Quatrième mission

Après la mise en place des trois premières missions, il est arrivé le temps de finaliser le projet : la synchronisation entre la base de données locale et l'API.

Dans un premier temps, il a été question de mettre en place un système de messages avec RabbitMQ, mais en raison de problèmes de transmission, l'idée a été abandonnée. Nous nous sommes alors rabattus avec le module Symfony Messenger. Ce système permet notamment lors d'une action spécifique (par exemple appuyer sur un bouton), d'exécuter une certaine action, de manière asynchrone, évitant d'utiliser trop de ressource ou de provoquer un timeout par une requête trop longue. Pour mettre en place cela, il aura été nécessaire de :

- Mettre en place une entité message avec constructeur vide (simplement pour l'instancier)
- Mise en place d'une classe MessageHandler, afin d'indiquer ce que doit faire le message à chaque instance (grâce à la méthode `__invoke`) (voir annexes).
- Rédaction d'un script en JavaScript/Ajax pour créer un message à chaque fois qu'un clic sur
- Mettre en place supervisor pour consommer les messages en fond en faisant tourner une commande en boucle.

f. Cinquième mission

Cette mission est la dernière de toute, le projet a été finalisé, il ne reste alors plus qu'à le pousser en production, mission réalisée avec Christian, le DevOps de l'équipe.

Il a alors fallu rendre le projet propre et utilisable, cela est donc passé par un nettoyage global du code du projet, l'optimisation de certaines méthodes, ainsi que la rédaction d'un README utile et pratique.

Après tout cela, le projet était donc terminé en temps et en heure, le dernier jour.

4. Annexes

a. Annexes du premier projet

Routes définies :

```
project_roles :  
  path : /project-roles/{groupId}  
  controller : App\Controller\ProjectRolesController::index
```

```
groups :  
  path: /groups  
  controller: App\Controller\GroupController::index
```

Méthode du contrôleur utilisée pour afficher les projets et les rôles en fonction d'un groupe, qu'il prend en paramètre

```
class ProjectRolesController extends AbstractController  
{  
  
    /**  
     * @Route("/project/roles", name="project_roles")  
     */  
    public function index(ManagerRegistry $doctrine, string $groupId): Response  
    {  
  
        $em = $doctrine->getManager();  
  
        $group = $em->getRepository(Groups::class)->find($groupId);  
        $actors = $group->getActors();  
  
        $projects = [];  
  
        foreach($actors as $a) {  
            $roles = $a->getRoles();  
            foreach($roles as $r) {  
                $projects[] = $r->getProjects();  
            }  
        }  
  
        return $this->render('project_roles/index.html.twig', [  
            'projects' => $projects,  
            'groupName' => $group->getName()  
        ]);  
    }  
}
```

Ce contrôleur à été mis à jour depuis, mais utilisait alors une méthode définie dans un service que voici :

```

public function main(string $groupId)
{
    $groupName = $this->getGroupNameFromId($groupId);
    $projectsIDs = $this->getProjectsIds();

    dump($groupName);
    $response = array();

    foreach($projectsIDs as $pID) {
        $rolesIds = $this->getRolesIdsFromProjectId($pID);
        foreach($rolesIds as $rID) {
            $request = $this->jiraClient->request('GET', 'project/' . $pID . '/role/' . $rID)->toArray();
            $actors = $request['actors'];
            foreach($actors as $a) {
                if(array_key_exists('name', $a)) {
                    if($a['type'] == 'atlassian-group-role-actor' && $a['name'] == $groupName) {
                        $response[$pID][] = $rID;
                    }
                }
            }
        }
    }
    return $response;
}

```

```

public function getGroupNameFromId(string $groupId) : string
{
    $query = $this->jiraClient->request('GET', 'group/bulk/?groupId=' . $groupId)->toArray();
    $values = $query['values'];
    $array = $values[0];
    return $array['name'];
}

```

```

public function getProjectsIds() : array
{
    $pIds = array();
    $projects = $this->jiraClient->request('GET', 'project')->toArray();

    foreach($projects as $p) {
        $pIds[] = $p['id'];
    }

    return $pIds;
}

```

```

public function getRolesIdsFromProjectId(string $projectId) : array
{
    $response = $this->jiraClient->request('GET', 'project/' . $projectId)->toArray();

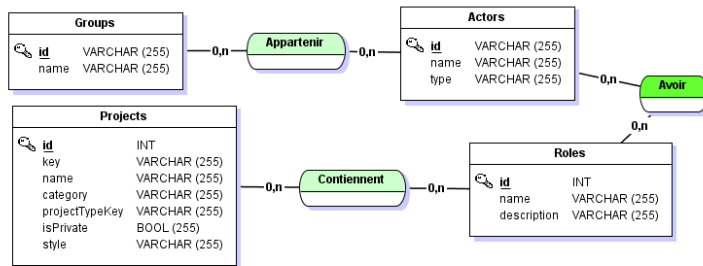
    $rolesIds = array();
    $roles = $response['roles'];
    foreach($roles as $r) {
        $rolesIds[] = intval(substr($r, -5));
    }

    return $rolesIds;
}

```

Le chemin peut paraître compliqué pour récupérer les données, cependant c'est le seul moyen d'y parvenir.

La structure de l'API (du moins les entités utilisées et requêtées) peuvent être modélisées de la sorte :



Ici un exemple de commande « app:get » récupérant toutes les données du lien passé en paramètre ainsi qu'une liste de commandes créées

```

class GetCommand extends Command
{
    protected static $defaultName = 'app:get';
    protected static $defaultDescription = 'Gets whatever get link is passed';
    private HttpClientInterface $jiraClient;

    public function __construct(HttpClientInterface $jiraClient)
    {
        parent::__construct();
        $this->jiraClient = $jiraClient;
    }

    protected function configure(): void
    {
        $this->addArgument('getLink', InputArgument::REQUIRED, 'Link to get');
    }

    protected function execute(InputInterface $input, OutputInterface $output): int
    {
        $io = new SymfonyStyle($input, $output);
        $link = $input->getArgument('getLink');

        $response = $this->jiraClient->request('GET', $link)->toArray();

        dump($response);

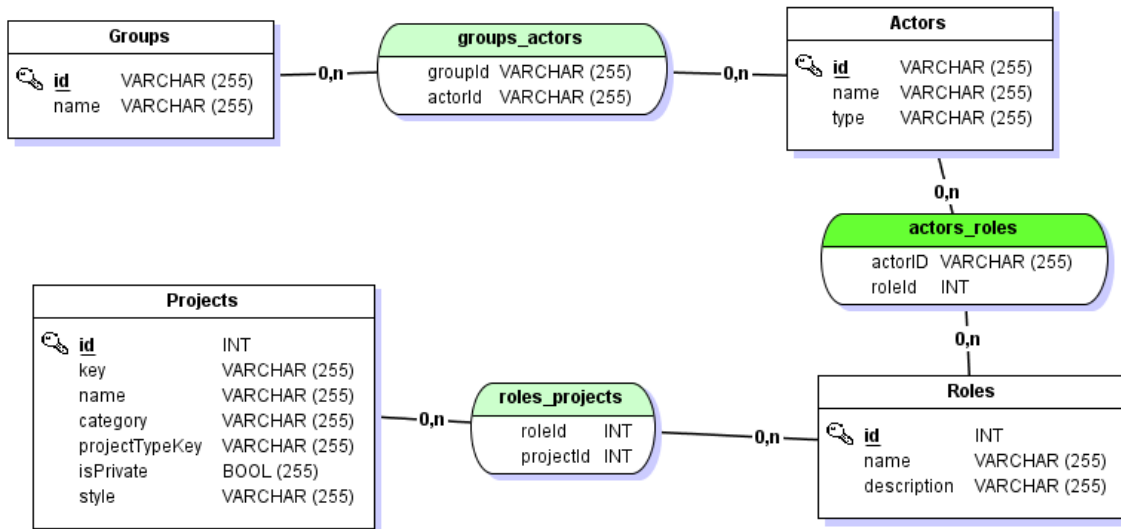
        $io->success('Executed');

        return Command::SUCCESS;
    }
}
  
```

- GetCommand.php
- JiraGroupIdCommand.php
- JiraGroupsCommand.php
- JiraProjectRolesCommand.php
- JiraProjectsCommand.php
- JiraRolesCommand.php

b. Annexes du second projet

Structure de la base de données : (toutes les relations sont de type « plusieurs- plusieurs »)



Structure des entités :

```
class Actors
{
    /**
     * @ORM\Id
     * @ORM\Column(type="string", length=255)
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=255)
     */
    private $name;

    /**
     * @ORM\Column(type="string", length=255)
     */
    private $type;

    /**
     * @ORM\Column(type="string", length=255, nullable=true)
     */
    private $accountId;

    /**
     * @ORM\ManyToOne(targetEntity=Groups::class, inversedBy="actors")
     */
    private $groupId;

    /**
     * @ORM\ManyToOne(targetEntity=Roles::class, mappedBy="actors")
     */
    private $roles;
}
```

```

class Groups
{
    /**
     * @ORM\Id
     * @ORM\Column(type="string", length=255)
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=255)
     */
    private $name;

    /**
     * @ORM\ManyToMany(targetEntity=Actors::class, mappedBy="groupId")
     */
    private $actors;
}

```

```

class Projects
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=20)
     */
    private $projectKey;

    /**
     * @ORM\Column(type="string", length=255)
     */
    private $name;

    /**
     * @ORM\Column(type="string", length=255, nullable=true)
     */
    private $projectTypeKey;
}

```

```

/**
 * @ORM\Column(type="string", length=255, nullable=true)
 */
private $style;

/**
 * @ORM\Column(type="boolean")
 */
private $isPrivate;

/**
 * @ORM\ManyToMany(targetEntity=Roles::class, inversedBy="projects")
 */
private $roles;

/**
 * @ORM\Column(type="string", length=255, nullable=true)
 */
private $category;
}

```

```

class Roles
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=255)
     */
    private $name;

    /**
     * @ORM\Column(type="string", length=255, nullable=true)
     */
    private $description;

    /**
     * @ORM\ManyToMany(targetEntity=Actors::class, inversedBy="roles")
     */
    private $actors;

    /**
     * @ORM\ManyToMany(targetEntity=Projects::class, mappedBy="roles")
     */
    private $projects;
}

```

Ici le contrôleur permettant l'affichage de la page des groupes (voir projet 1 pour les projets et rôles relatifs aux groupes)

```

class GroupController extends AbstractController
{
    /**
     * @Route("/group", name="group")
     */
    public function index(ManagerRegistry $doctrine): ?Response
    {
        $em = $doctrine->getManager();
        $groups = $em->getRepository(Groups::class)->findAll();
        return $this->render('group/index.html.twig', [
            'groups' => $groups
        ]);
    }
}

```

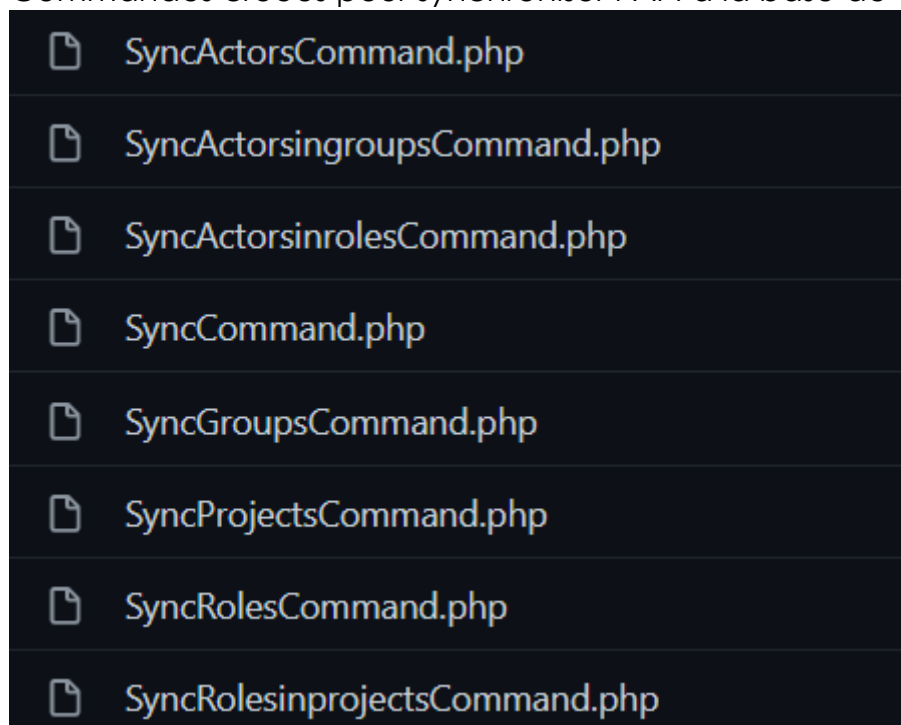
Exemple d'un fichier « template » (ici project_roles) :

```
{% block body %}

<div class="container">
  <table class="table table-striped table-hover">
    <thead>
      <tr>
        <td>Group Id</td>
        <td>Group Name</td>
      </tr>
    </thead>
    <tbody>
      {% for group in groups %}
        <tr>
          <td><a href="{{ path('project_roles', {groupId: group.getId() }}" target="_blank">{{ group.getId() }}</a></td>
          <td>{{ group.getName() }}</td>
        </tr>
      {% endfor %}
    </tbody>
  </table>
</div>
{% endblock %}
```

c. Annexes du troisième projet

Commandes créées pour synchroniser l'API à la base de données



On y retrouve une commande pour chaque entité, ainsi qu'une commande qui devra exécuter toutes les autres (selon une priorité définie). Cette dernière n'est pas encore implémentée.

Exemple d'un fichier commande de synchronisation (ici pour les projets) :

```
class SyncProjectsCommand extends Command
{
    protected static $defaultName = 'app:sync:projects';
    protected static $defaultDescription = 'Add a short description for your command';
    private HttpClientInterface $jiraClient;
    private ManagerRegistry $doctrine;

    public function __construct(HttpClientInterface $jiraClient, ManagerRegistry $doctrine)
    {
        parent::__construct();
        $this->jiraClient = $jiraClient;
        $this->doctrine = $doctrine;
    }
}
```

```
protected function execute(InputInterface $input, OutputInterface $output): int
{
    $io = new SymfonyStyle($input, $output);

    $em = $this->doctrine->getManager();
    $data = $this->jiraClient->request('GET', 'project')->toArray();
    foreach($data as $d) {
        $project = $em->getRepository(Projects::class)->find($d['id']);
        if($project == NULL) {
            var_dump('here');
            $project = new Projects();
            $project->setId(intval($d['id']));
        }
        $project->setProjectKey($d['key'])
            ->setName($d['name'])
            ->setProjectTypeKey($d['projectTypeKey'])
            ->setStyle($d['style'])
            ->setIsPrivate($d['isPrivate']);
        if(array_key_exists('projectCategory', $d)) {
            $project->setCategory($d['projectCategory']['name']);
        }
        $em->persist($project);
        $em->flush();
    }

    $io->success('Projects synced!');

    return Command::SUCCESS;
}
```

Ici, la commande va requêter l'API sur la page des projets, pour chaque projet récupéré, la commande va tenter de récupérer une entité avec un ID

correspondant, si elle n'existe pas, on instancie un nouvel objet avec l'identifiant, sinon, on édite ses attributs.

Code d'un contrôleur pour afficher les détails d'un projet, en fonction de l'ID passé en paramètre.

```
class ProjectByIdController extends AbstractController
{
    /**
     * @Route("/project", name="project_by_id")
     */
    public function index(ManagerRegistry $doctrine, int $projectId): Response
    {
        $em = $doctrine->getManager();

        $project = $em->getRepository(Projects::class)->find($projectId);

        return $this->render('project_by_id/index.html.twig', [
            'controller_name' => 'ProjectByIdController',
            'project' => $project
        ]);
    }
}
```

d. Annexes du quatrième projet

Structure de la classe MessageHandler, ayant pour but de reconnaître les services passés en argument, et de demander une synchronisation à chaque instantiation de l'entité Message.

```
#[AsMessageHandler]
class MessageHandler implements MessageHandlerInterface, LoggerAwareInterface
{
    use LoggerAwareTrait;

    /** @var SynchronizerInterface[] $services */
    private array $services;

    public function __construct(iterable $services)
    {
        foreach($services as $s) {
            if($s instanceof SynchronizerInterface) {
                $this->services[] = $s;
            } else {
                throw new LogicException();
            }
        }
    }

    public function __invoke(Message $message)
    {
        $this->logger->debug('invoked');
        foreach($this->services as $s) {
            $s->synchronize();
        }
    }
}
```

Script Ajax, ouvrant en fond une page dont le contrôleur sert à créer un message

```
{% block javascripts %}
<script type="text/javascript">
    function sync() {
        let httpRequest = new XMLHttpRequest();

        if (!httpRequest) {
            alert('Abandon :( Impossible de créer une instance de XMLHttpRequest');
            return false;
        }

        httpRequest.onreadystatechange = alertContents;
        httpRequest.open('GET', '{{ path('app_sync') }}');
        httpRequest.send();
    }

    function alertContents(httpRequest) {
        if (httpRequest.readyState === XMLHttpRequest.DONE) {
            if (httpRequest.status === 200) {
                alert(httpRequest.responseText);
            } else {
                alert('Il y a eu un problème avec la requête.');
```



```
class SyncController extends AbstractController
{
    private MessageBusInterface $bus;

    public function __construct(MessageBusInterface $bus)
    {
        $this->bus = $bus;
    }

    #[Route('/sync', name: 'app_sync')]
    public function index(): Response
    {
        $this->bus->dispatch(new Message());
        return new Response('', 204);
    }
}
```