

Features Flags

Backend Frontend



Feature Flags

Features Flags	1
I. Introduction Features Flags	3
A. Avantages	3
B. Inconvénients	3
II. Intégration au code	4
A. Backend (laravel)	4
B. Frontend (react)	5
1. Proxy Unleash (local Docker)	5
2. Configuration React	8
3. Utilisation	9
III. Gitlab	10

I. Introduction Features Flags

Les Features Flags, également connus comme fonctionnalités conditionnelles ou « feature toggles », sont des outils de développement logiciel qui permettent de séparer le déploiement d'une fonctionnalité d'une application de sa mise en production effective pour les utilisateurs finaux. Cela permet aux développeurs de contrôler le déploiement et l'activation des fonctionnalités de manière flexible et contrôlée.

Ce sont de simples conditions dans le code qui vérifient si une certaine fonctionnalité est activée ou non.

Des features flags plus avancés permettent de cibler des utilisateurs spécifiques afin de proposer progressivement une fonctionnalité à de plus en plus d'utilisateurs et de faire un déploiement continu.

Les features flags permettent notamment de pouvoir faire des test des nouvelles fonctionnalités directement en production sans craindre de devoir rollback puis redéployer l'ancienne version. Il suffira simplement de désactiver le feature flag.

A. Avantages

Les features flags offrent de nombreux avantages. Ils permettent un déploiement progressif et continu, notamment en rendant accessible les nouvelles fonctionnalités uniquement à un petit nombre d'utilisateurs finaux et en permettant aux développeurs de fusionner du code dans la branche principale sans avoir à attendre que toutes les fonctionnalités soient complètes.

Les features flags ont également un contrôle granulaire sur les fonctionnalités déployées, en activant et désactivant des fonctionnalités à des moments précis, mais aussi en réduisant les risques en désactivant rapidement une fonctionnalité sans avoir à déployer du nouveau code.

B. Inconvénients

Cependant, malgré les nombreux avantages, les features flags présentent également des inconvénients.

En effet, l'implémentation des features flags, si elle n'est pas effectuée correctement, va grandement complexifier le code, sa lisibilité, sa maintenance voir même provoquer des risques de bugs et de compromission de la sécurité si les informations sensibles sont mal sécurisées.

Il est donc important de planifier et de mettre en oeuvre une stratégie appropriée pour maximiser les avantages tout en minimisant les inconvénients.

II. Intégration au code

A. Backend (laravel)

Pour utiliser les feature flags dans un projet laravel plusieurs packages sont disponibles. Cependant, pour utiliser la fonctionnalité de gitlab, il est nécessaire d'utiliser un package se basant sur unleash. Le package j-webb/laravel-unleash est le plus adapté à cette utilisation.

Il faut dans un premier temps installer le package et le configurer . Une fois l'installation faite, il faut modifier le fichier .env pour configurer le package afin qu'il utilise l'api de gitlab pour la gestion des features.

```
# Your gitlab API link
UNLEASH_URL=https://gitlab.private-iconosquare.com/api/v4/feature_flags/unleash/63

# Enable or disable the Laravel Unleash client. If disabled, all feature checks will return false
UNLEASH_ENABLED=true

# Instance id for this application (typically hostname, podId or similar)
UNLEASH_INSTANCE_ID=9SHB2WnsxTogVNKWTsv_

# The Unleash environment name, which can be used to as a parameter for enabling/disabling features for local or development environments
UNLEASH_ENVIRONMENT=local
```

Ces quatre variables sont obligatoires pour que le package fonctionne. Il faut mettre les informations de l'api de gitlab, que l'on retrouve dans l'onglet Feature Flags de Deploy en allant sur le bouton « Configurer » :

- . UNLEASH_URL = API url
- . UNLEASH_INSTANCE_ID = Instance ID
- . UNLEASH_ENVIRONMENT = Application Environment (correspond au environnement gitlab : prod, local, dev ...)

Le package est maintenant fin prêt à être utilisé !

Pour ce qui est de l'utilisation des features flags, plusieurs méthodes sont possibles. Cependant, il faut toujours garder à l'esprit de faire le plus simple possible afin que ce soit le plus lisible et compréhensif possible.

L'une des méthodes les plus simples consiste à créer une nouvelle page avec les

modifications souhaitées, puis de faire la gestion du feature flag au niveau des routes. Pour cela dans le fichier web.php, on ajoute le use Unleash comme ceci :

```
use Jwebb\Unleash\Facades\Unleash;
```

Puis on fait la vérification du flag comme ceci :

```
if (Unleash::isEnabled('test')){
    Route::get('/', function(){
        return view('test', ['name' => 'Marius']);
    });
}else{
    Route::get('/', function () {
        return view('welcome');
    });
}
```

Cela veut dire que si le feature flag est actif, alors on affiche la page « test », sinon on affiche la page de base « welcome ».

Il est possible de passer par le middleware de laravel dont vous pourrez trouver un exemple d'utilisation ici : <https://github.com/j-webb/laravel-unleash?tab=readme-ov-file>.

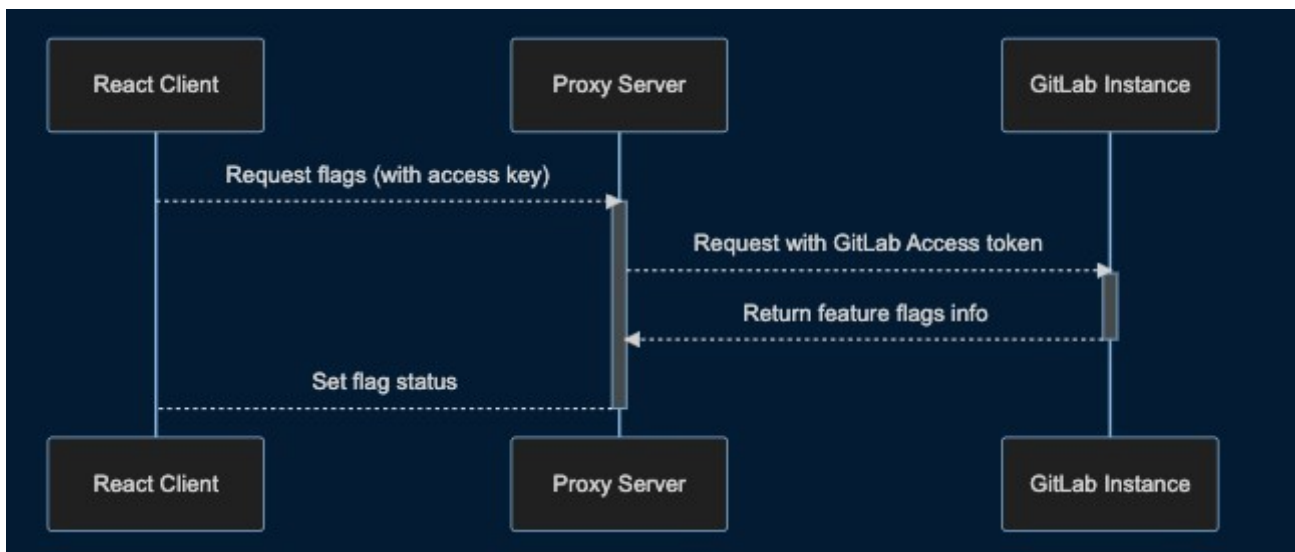
Il est également possible d'utiliser les feature flags directement dans du code html mais cette méthode n'est pas recommandée car peu lisible. Préférez toujours passer par un controller ou par les routes et évitez un maximum les feature flag imbriqués. Bien que possible, cela complexifiera grandement la lecture du code ainsi que sa maintenabilité. La structure reste la même qu'importe l'endroit de l'utilisation du feature flag. Le principe sera toujours :

```
if (Unleash::isEnabled('test')){
    // nouveau code
}else{
    // code de base
}
```

B. Frontend (react)

Pour utiliser les feature flags dans un projet React, plusieurs packages sont disponibles. Cependant, pour utiliser la fonctionnalité de gitlab, il est nécessaire d'utiliser un package se basant sur unleash.

En revanche contrairement à Laravel, il est impossible de connecter directement une application React directement à gitlab. Il est donc obligatoire de passer par un proxy Unleash. La démarche sera donc la suivante : l'application React se connecte au proxy, puis le proxy se connecte à Gitlab, puis la réponse fait le chemin inverse comme sur le schéma suivant.



1. Proxy Unleash (local Docker)

Avant de pouvoir commencer à utiliser les feature flags dans votre code, il faut au préalable configurer un proxy Unleash, y compris en local afin de pouvoir faire les tests.

Contrairement au package de laravel, le proxy, lui, a besoin d'avoir un token d'accès utilisateur ou à un projet. Il est fortement recommandé pour cet usage d'utiliser un token d'accès au projet car cela restreindra le proxy uniquement au projet.

Pour ce qui est de la configuration, il est possible d'utiliser la commande suivante :

```
docker run \
  -e UNLEASH_PROXY_SECRETS=9vZBzCd2jeqE7JY \
  -e UNLEASH_URL=https://gitlab.private-iconosquare.com/api/v4/feature_flags/
unleash/65 \
  -e UNLEASH_API_TOKEN=glpat-hpSbyTqraS3d2uyLGq83 \
  -e UNLEASH_INSTANCE_ID="TeTFLJNEjdnjzsWgrr6s" \
  -p 3030:3030 \
  unleashorg/unleash-proxy
```

Mais il est préférable de passer par un docker-compose.yml.

```
proxy:
  container_name: proxy
  image: unleashorg/unleash-proxy
  ports:
    - "3000:3000"
  environment:
    - UNLEASH_PROXY_SECRETS=9vZBzCd2jeqE7JY
    - UNLEASH_URL=https://gitlab.private-iconosquare.com/api/
v4/feature_flags/unleash/65
    - UNLEASH_API_TOKEN=glpat-hpSbyTqraS3d2uyLGq83
    - UNLEASH_INSTANCE_ID=TeTFLJNEjdnjzswgrr6s
```


Petit point sur les variables du proxy :

de UNLEASH_PROXY_SECRETS : c'est une chaîne aléatoire qui sera utilisée pour le client react afin de contacter le proxy

UNLEASH_URL : c'est l'api url de gitlab que l'on trouve dans l'onglet Feature Flags de Deploy sur le bouton « configure »

UNLEASH_API_TOKEN : c'est le token d'accès au projet créé dans gitlab. Pour le créer, il faut se rendre dans l'onglet « access token » des paramètres et créer un token d'accès pour le projet.

!/\ Pensez à garder la clé du token précieusement quelque part car une fois créée, elle ne vous sera affichée qu'une seule fois. Si vous la perdez, il faudra en recréer une nouvelle !!!

UNLEASH_INSTANCE_ID : c'est l'id de l'api gitlab que l'on trouve dans l'onglet Feature Flags de Deploy sur le bouton « configure ».

Avant de pouvoir commencer, il reste un point important auquel il faut faire attention. En effet, l'image docker du proxy unleash est configuré pour écouter sur le port 3000 et l'on ne peut pas le modifier. Il faut donc vérifier si l'application React en local écoute sur le port 3000 et, le cas échéant, changer le port. Cela peut être effectué en passant par le fichier package.json comme ceci :

```
"scripts": {  
  "start": "PORT=65000 react-scripts start",  
  "build": "react-scripts build",  
  "test": "react-scripts test",  
  "eject": "react-scripts eject"
```

2. Configuration React

Maintenant, nous pouvons commencer à configurer l'application react. Dans un premier temps, il faut installer le package @unleash/proxy-client-react puis dans le fichier index.js faire la configuration pour le proxy unleash :

```
import { FlagProvider } from '@unleash/proxy-client-react';

const {
  REACT_APP_PROXY_URL,
  REACT_APP_PROXY_CLIENT_KEY,
  REACT_APP_PROXY_APP_NAME,
  REACT_APP_PROXY_ENVIRONMENT
} = process.env;

const config = {
  url: REACT_APP_PROXY_URL,
  clientKey: REACT_APP_PROXY_CLIENT_KEY,
  appName: REACT_APP_PROXY_APP_NAME,
  environment: REACT_APP_PROXY_ENVIRONMENT,
  refreshInterval: 10, // Fréquence (en secondes) à laquelle le client
  doit interroger le proxy pour obtenir des mises à jour.
};

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <FlagProvider config={config}>
      <App />
    </FlagProvider>
  </React.StrictMode>
);
```

Il faut également mettre toutes les informations dans des variables d'environnement :

```
REACT_APP_PROXY_URL=http://localhost:3000/proxy
REACT_APP_PROXY_CLIENT_KEY=9vZBzCd2jeqE7JY
REACT_APP_PROXY_APP_NAME=reactFlag
REACT_APP_PROXY_ENVIRONMENT=local
```

En ce qui concerne la variable REACT_APP_PROXY_APP_NAME elle n'est pas utilisée pour gitlab donc nous pouvons mettre ce que nous voulons. La variable REACT_APP_PROXY_CLIENT_KEY correspond à la chaîne de caractères renseignée lors de la configuration du proxy, de même que pour REACT_APP_PROXY_ENVIRONMENT.

3. Utilisations

Plusieurs utilisations sont possibles, cela dépend de ce que vous souhaitez faire. Cependant la structure reste toujours la même. Il faut faire l'import de useFlag depuis le module unleash :

```
import { useFlag } from "@unleash/proxy-client-react";
```

Puis, dans le component, nous devons faire une variable qui est = à useFlag('nom_du_flag'). Le nom du flag sera le même que dans gitlab. Puis un return avec ce que vous souhaitez en fonction de si le flag est actif ou non. Cela peut être juste un string simple que l'on transmet comme ceci :

```
export const ExampleComponent = () => {  
  const exampleFlag = useFlag(`example_flag_key`);  
  return <p>{ exampleFlag ? 'Enabled' : 'Disabled' }</p>  
}
```

Mais cela peut aussi être directement un changement de vu comme ceci :

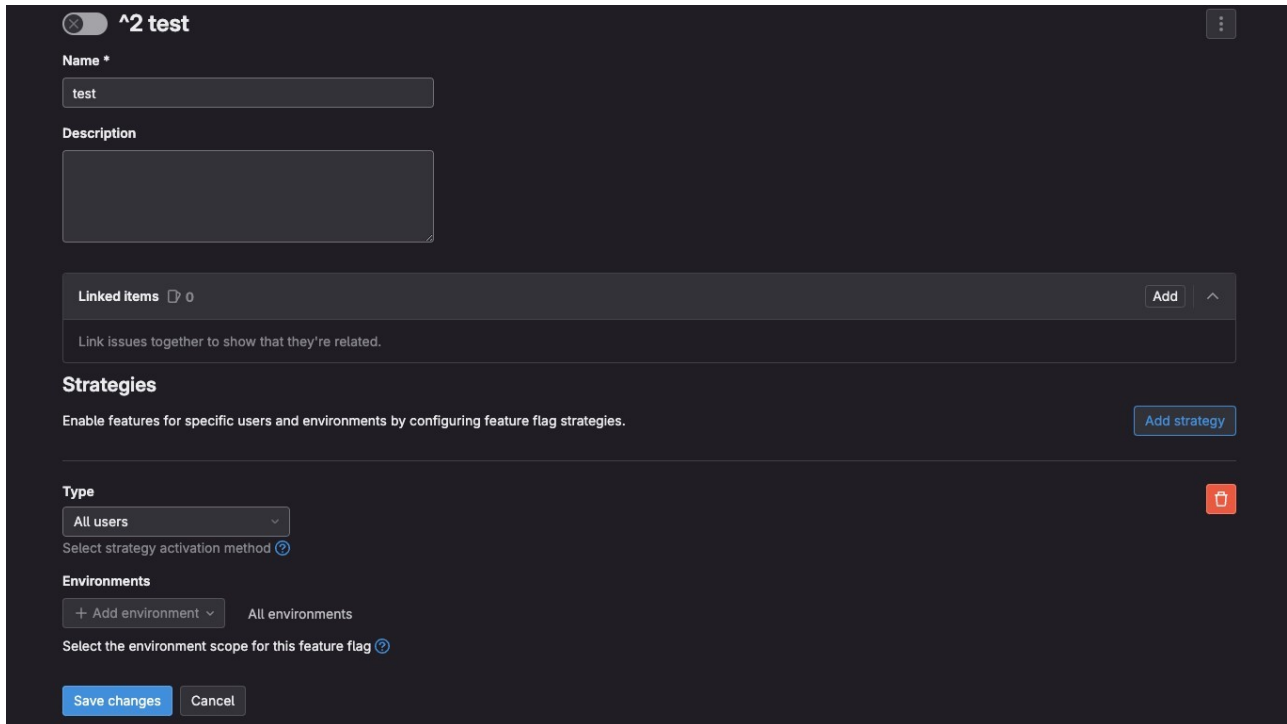
```
import { useFlag } from "@unleash/proxy-client-react";  
import logo from "../logo.svg";  
import UserActivation from "../UserIdActivationStrategy";  
import MyFirstComponent from "../MyFirstComponent";  
import maintenanceIcon from "../setting-line-icon.svg";  
  
const MainComponent= () => {  
  const Flag = useFlag('main');  
  const result = Flag ? (  
    <div className="App">  
      <header className="App-header">  
        <img src={logo} className="App-logo" alt="logo" />  
        <p>  
          Edit <code>src/App.js</code> and save to reload.  
        </p>  
        <a className="App-link" href="https://reactjs.org" target="_blank" rel="noopener noreferrer">Learn React</a>  
        <UserActivation/>  
        <MyFirstComponent name="Robert" lastname="Dupont"><p>La dedans je mets ce que je veux</p></MyFirstComponent>  
      </header>  
    </div>  
  ) : (  
    <div className="App">  
      <header className="App-header">  
        <img src={maintenanceIcon} className="App-logo" alt="logo" />  
        <h1>En Maintenance</h1>  
      </header>  
    </div>  
  );  
  return (result);  
}  
  
export default MainComponent;
```

Il y a tout de même une petite subtilité entre les deux utilisations. En effet, pour la première utilisation, on a directement mis dans le return la valeur que l'on

souhaite transmettre en fonction de l'état du flag. Cependant, pour renvoyer une vue complète, il faut d'abord passer par une variable (ici results) dans laquelle on met les différentes vues en fonction de l'état du flag. Dans l'exemple précédent, si le flag est actif, on affiche la page de base sinon on affiche une page de maintenance.

III. Gitlab

Afin de pouvoir gérer les feature flags créés dans votre code, il faut également les créer dans Gitlab. Cependant il faut impérativement que les noms des feature flag correspondent au nom des feature flags dans votre code sinon cela ne fonctionnera pas.



The screenshot shows the GitLab interface for configuring a feature flag named '^2 test'. At the top left, there is a toggle switch and the flag name. Below this, the 'Name' field is filled with 'test'. The 'Description' field is empty. A 'Linked Items' section shows 0 items with an 'Add' button. The 'Strategies' section has a description and an 'Add strategy' button. The 'Type' section is set to 'All users' with a 'Select strategy activation method' link. The 'Environments' section has a '+ Add environment' button and 'All environments' selected. At the bottom, there are 'Save changes' and 'Cancel' buttons.

☐ ^2 test

Name *

test

Description

Linked Items 0 Add ^

Link issues together to show that they're related.

Strategies

Enable features for specific users and environments by configuring feature flag strategies. Add strategy

Type

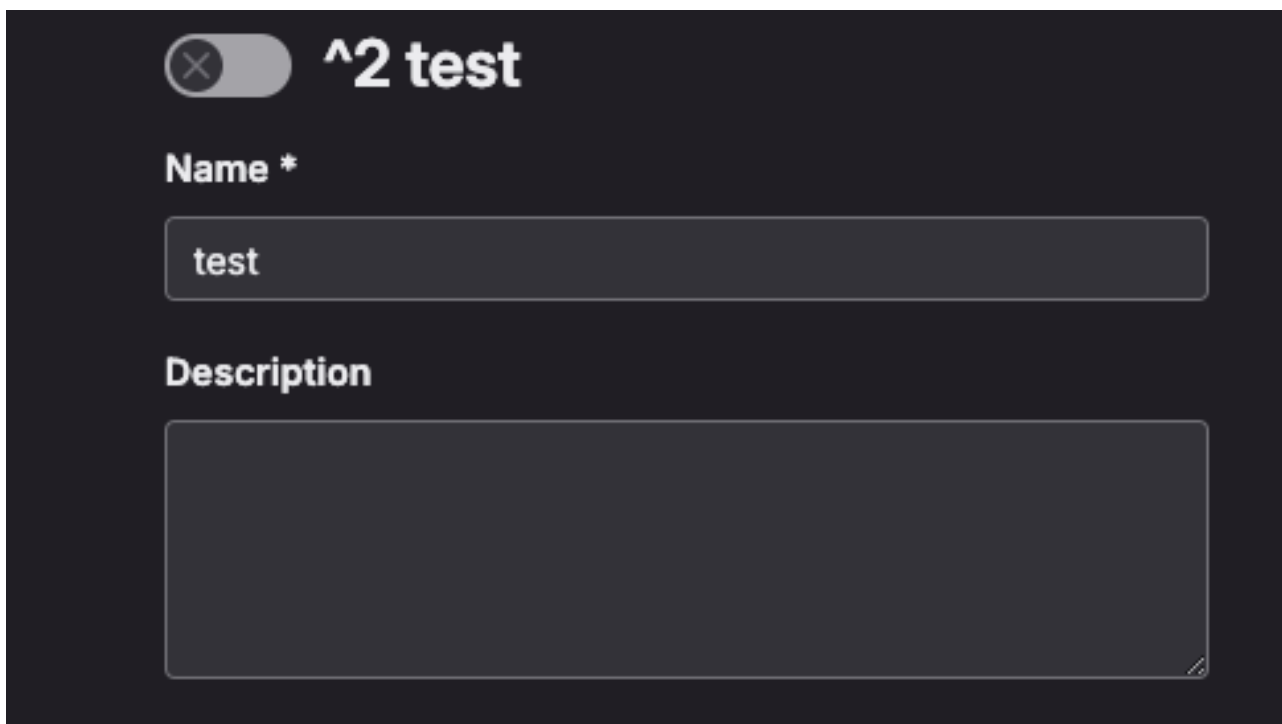
All users Select strategy activation method ?

Environments

+ Add environment All environments

Select the environment scope for this feature flag ?

Save changes Cancel



This is a close-up view of the top part of the configuration page. It shows the toggle switch, the flag name '^2 test', the 'Name' field with 'test', and the empty 'Description' field.

☐ ^2 test

Name *

test

Description

Il faut également définir une stratégie pour le flag. En effet plusieurs stratégies sont possible pour un seul et même flag et pour différents environnements.

Strategies

Enable features for specific users and environments by configuring feature flag strategies.

Type

All users ▾

Select strategy activation method ?

Environments

+ Add environment ▾ local ✕

Select the environment scope for this feature flag ?

Type	Percentage	Based on
Percent rollout ▾	50 %	Session ID ▾
Select strategy activation method ?	Enter an integer number between 0 and 100	Consistency guarantee method

Environments

+ Add environment ▾ prod ✕

Select the environment scope for this feature flag ?

Save changes **Cancel**

Sur cet exemple, une première stratégie est en place : tous les utilisateurs de l'environnement local verront le feature flag activé. La deuxième stratégie, quant-à-elle, agira uniquement en prod et va rediriger 50% des utilisateurs vers la nouvelle version et elle se basera sur le session ID de l'utilisateur.

Pour plus d'informations sur les stratégies possibles, veuillez vous référer à la doc de Gitlab : [https:// docs.gitlab.com/ee/operations/feature_flags.html#feature-flag-strategies](https://docs.gitlab.com/ee/operations/feature_flags.html#feature-flag-strategies).

Il est donc possible de mettre des stratégies différentes pour un même flag en fonction de l'environnement.

Une fois le flag créé, on peut l'activer et le désactiver en un simple clic. Il peut tout de même y avoir un délai de l'ordre de quelques minutes dû à l'appel de l'api.