CI/CD avec GitHub Actions

m2iformation.fr

Sommaire

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Introduction

CI / CD

Dans l'univers du développement informatique, il est utile d'avoir des mécanismes de gestion de notre flux de travail de façon automatisée. Il est en effet long et répétitif de devoir, à chaque fois que l'on produit une nouvelle version de notre application, la tester, réaliser le packaging, le déploiement, contacter les intéressés du lien de téléchargement de cette version et/ou mettre à jour les environnements de production pour avoir la dernière version.

Pour éviter ceci, on a tendance à utiliser des outils de Continuous Integration/Continuous Delivery (**CI/CD**).

GitHub Actions

Pour réaliser ces mécanismes, il est possible d'utiliser plusieurs outils, dont GitHub Actions. Via ce service gratuitement offert par GitHub dans le cadre de repositories publiques, il est possible de réaliser à la fois du CI/CD, mais pas que.

Il est possible en effet de réaliser de la gestion de repositories de façon automatisée vu que le service est en lien direct avec GitHub.

TOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Récap

Git et GitHub

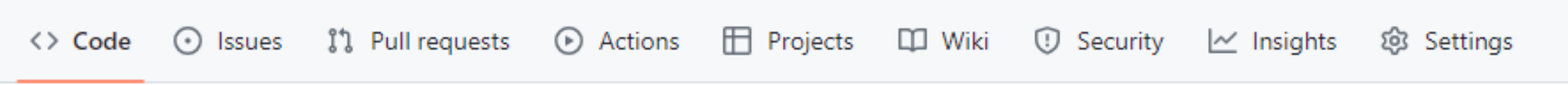
Premier Workflow

Concepts

- **Workflow**: Attachés à un repository, ils ajoutent l'automatisation via une liste de jobs. Il est possible de brancher leur exécution sur des évènements tels qu'un push sur une branche particulière.
- **Job**: Objectifs d'un workflow, séparés en un ou plusieurs steps. Chaque job possèdera son environnement d'exécution (un **runner**). Il est possible de les rendre conditionnels ou de les faire en parallèle.
- **Step**: Instructions séquentielles permettant l'atteinte d'un objectf donné. Bien souvent, il s'agit d'un script shell, mais il est possible d'utiliser des actions issues d'éléments externes.

Notre première action

Pour créer une action, il suffit d'avoir un repository et d'appuyer sur l'onglet **Actions** se trouvant dans la barre d'onglets:



Une fois cliqué, il nous est offert plusieurs choix d'actions pré-configurées. Pour notre prermier essai, nous allons choisir Simple Workflow et cliquer sur **Configure**.

Les fichiers de workflow

Un workflow se crée sous la forme d'un fichier à l'extension **.yml**. Ce fichier devra contenir plusieurs attributs, le premier étant le nom du workflow via la clé **name**. Viendra ensuite le déclencheur du workflow, qui peut être une série de déclencheurs via passage d'un texte ou d'un tableau de texte dans l'attribut **on**. Enfin, il faudra une liste de **jobs**:

- Les jobs seront des objets constitués d'un runner via **runs-on** et d'une série de **steps**.
- Les steps seront des objets contitués d'un **name** et d'une commande à effectuer via **run**.

Hello World Workflow

Un workflow basique provoquant des logs en console pourrait ressembler à ceci:

```yaml
name: First Workflow
on: workflow_dispatch
jobs:
  first-job:
    runs-on: ubuntu-latest
    steps:
      - name: Say Hello
        run: echo "Hello World"
      - name: Say Bye
        run: echo "Done - Cya!"
```

Créer des workflows dans le projet

Il est également possible de créer nos workflows directement dans notre projet. Pour cela, il suffit de créer des fichiers **.yml** via cette hiérarchie:

```
root_folder
└──.github
    └──workflows
        └──action_name.yml
```

Pour nous y aider, sur VS Code, il est possible d'installer l'extension **GitHub Actions** afin de bénéficier d'une auto-complétion et, si connecté à GitHub, de la capacité de déclencher les actions depuis VS Code.

Utiliser des actions

Imaginons que notre workflow va demander des instructions plus complexes qu'un simple log en console. Pour par exemple récupérer le code de notre application dans l'environnement du runner, il va falloir obtenir les credentials, exécuter plusieurs dizaines de lignes de commande...

Au lieu de cela, GitHub nous fournit des **actions** pré-fabriquées que l'on peut imbriquer dans nos propres workflows. Les actions sont pour la majorités proposées dans le Marketplace, mais il est possible, si on le désire, de créer nos propres actions (et si l'on le veut, de les rendre disponibles de la même façon).

Action: Récupérer le code

Ainsi, dans la majorité des projets, il va arriver une étape de récupération du code source. Pour ceci, nous pouvons utiliser l'action Checkout:

```
steps:
    - name: Get code
      uses: actions/checkout@v3
```

Si l'on le veut, on peut désormais créer toute une série d'actions en lien avec un projet particulier. Par exemple, dans le cas d'une application ReactJS, on pourrait réaliser les tests de notre application à chaque push via un workflow.

Exemple avec ReactJS

```yaml
name: Test on push
on: push
jobs:
    test:
        runs-on: ubuntu-latest
        steps:
            - name: Get code
              uses: actions/checkout@v3
            - name: Install dependencies
              run: npm ci
            - name: Run tests
              run: npm test
```

Plusieurs Jobs

Il est aussi très commun, dans un workflow, d'avoir besoin de plusieurs jobs, tels que le déploiement une fois les tests effectués. Dans le cas où l'on veut faire plusieurs jobs se déroulant en parallèle, c'est très simple, il suffit d'utiliser une syntaxe telle que:

```yaml
name: Multi-Job Workflow
on: push
jobs:
    first-job:
        runs-on: ubuntu-latest
        steps:
            ...
    second-job:
        runs-on: ubuntu-latest
        steps:
            ...
```

Jobs multiples séquentiels

Dans le cas où l'on aimerait éviter le lancement de nos différents jobs de façon parallèle, il est possible d'ajouter un simple attribut (**needs**) de sorte à demander la fin d'un job avant le lancement d'un autre :

```yaml
name: Multi-Job Workflow
on: push
jobs:
    first-job:
        ...

    second-job:
        needs: first-job
        ...
```

Provoquer les actions de plusieurs façon

Imaginons que l'on veuille pouvoir avoir plusieurs triggers de notre action. Dans ce cas, on peut, à la place d'une chaine de caractères, avoir un tableau de chaînes de caractères en valeur de **on** :

```yaml
name: Multi-Trigger Workflow
on: [push, workflow_dispatch]
```

ou

```yaml
name: Multi-Trigger Workflow
on:
  - push
  - workflow_dispatch
```

Les expressions et contextes

Dans un workflow, il est possible d'utiliser des expressions en lieu et place des données statiques. De la sorte, il est possible d'obtenir le résultat de fonctions disponibles et si l'on veut de modifier des variables ou de parcourir des objets complexes. Pour ce faire, il faut utiliser la syntaxe:

```
${{ expression }}
```

Certaines variables sont disponibles par défaut dans les workflows GitHub, ces variables peuvent être par exemple le contexte GitHub du repository où se trouve le workflow, accessible via l'objet `github`.

Action: Affichage du contexte Github

Ainsi, dans le cas où l'on veut avoir un affichage au format JSON de notre contexte GitHub, on peut créer une action de ce genre:

```yaml
name: Output Information
on: workflow_dispatch
jobs:
    info:
        runs-on: ubuntu-latest
        steps:
            - name: Output GitHub Context
              run: echo "${{ toJson(github) }}"
```

TOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Les évènements

Déclenchement de nos workflows

Aller plus loin avec les évènements

Dans le cas où l'on souhaite avoir plus de contrôle quant au déclenchement de nos évènements, il va falloir, à la place d'une simple chaîne de caractères ou d'une liste de chaînes de caractères, envoyer à l'attribut **on** des objets plus ou moins complexes. Pour définir la spécificité, il existe plusieurs façon de le faire:

- **Activity Types**: Par exemple, en cas de pull request validée, en cours de traitement, refusée, etc...
- **Filters**: Par exemple, en cas de push sur une branche particulière

Activity Types

Certains évènements peuvent être rendus spécifiques via l'utilisation de **types d'activités**. Ces types sont disponibles dans la documentation officielle de l'évènement concerné. Ainsi, si l'on regarde la documentation de pull request, on peut voir plein de différents types d'actions en lien avec une Pull Request. Par exemple, si l'on veut déclencher **à l'ouverture** d'une pull request:

```yaml
on:
  pull_request:
    types:
      - opened
  workflow_dispatch:
```

Event Filters

Une autre façon de rendre nos workflows plus spécifiques est l'utilisation de **filtres**. Par exemple, via les filtres, il est possible de ne déclencher un workflow qu'en cas de push sur une branche spécifique, ou pour toutes les branches sauf une série de branches.
Encore une fois, on peut savoir le nom des filtres via la documentation des évènements de la même façon:

```yaml
on:
  push:
    branches:
      - prod # prod
      - 'dev-*' # dev- | dev-01 | dev-blabla
```

Précisions (1/2)

Par défaut, les workflows déclenchés par ouverture de pull requests ne seront pas déclenchés en cas de **fork**.

Cette différence est nécessaire dans le but d'éviter des comportements malicieux tels que l'appel d'un workflow en boucle pouvant causer un surcoût important pour l'utilisateur de GitHub Actions. Dans ce genre de cas, l'exécution du workflow devra être approuvée manuellement avant qu'elle puisse avoir lieu.

Précisions (2/2)

Les workflows vont également être stoppés automatiquement en cas d'échec d'un job faisant partie du workflow.
D'ailleurs, en cas de problème de déclenchement d'un workflow non désiré, il est bon de savoir qu'il est possible d'annuler son exécution manuellement afin d'éviter de consommer GitHub Actions sans raison. On peut également faire en sorte de passer l'exécution d'un workflow, via des exceptions que l'on peut spécifier par exemple dans le commit entre crochets:

```
git commit -m "texte commentaire [skip ci]"
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Les Données

Job Artifacts

Dans le cas où l'on aurait un job dont l'objectif est le build d'une application, il va nous falloir idéalement pouvoir la mettre en ligne, la conteneuriser, etc... Ceci serait possible avec l'ajout d'autres étapes dans notre job, mais ce n'est pas la meilleure chose à faire.
Dans la pratique, on va plutôt extérioriser le résultat d'un job sous la forme d'**assets/artifacts** (par exemple le site web, le binaire de l'application). Leur destinée peut alors être:

- Un téléchargement et usage immédiat en tant que tel
- Un téléchargement et usage dans un autre job

Stockage de la sortie

Dans le cas d'une application React, la sortie de l'application suite à un build se trouvant dans un dossier **dist**. Si l'on souhaite pouvoir exporter la sortie de ce build, il est possible d'utiliser une action disponible dans le marketplace pour cela:

```yaml
- name: Upload artifacts
  uses: actions/upload-artifact@v3
  with:
    name: dist-files
    path: |
      fileOrFolderName
      folder/fileName.extension
      !NotIncludedFileName.extension
```

Récupération dans un job ultérieur

En cas de nécessité d'utiliser les fichiers produits par un premier job dans un second job, il va nous falloir, en début de l'autre job, utiliser une autre étape et une autre action prévue à cet effet:

```
- name: Get build artifact
  uses: actions/download-artifact@v3
  with:
    name: dist-files
- name: Show files
  run: ls
```

Les Outputs

Dans le cas où l'on aimerait avoir des sorties autre que des fichiers, par exemple des valeurs récupérées par des commandes Linux telles qu'un nom de fichier, il va nous falloir utiliser le mecanisme des **outputs** ainsi que les variables de contexte **steps** et **needs**:

```yaml
on: push
outputs:
  workflow-output-name: ${{ steps.stepId.outputs.stepOutputName }}
steps:
  - name: Publish filename
    id: stepId
    run: find path/to/*.txt -type f -execdir echo 'stepOutputName={}' >> $GITHUB_OUTPUT ';'
```

```yaml
- name: Output filename
  run: echo ${{ needs.build.outputs.script-file }}
```

Le Caching

Si certains de nos jobs prennent un certain temps à s'exécuter, il peut être intéressant de se servir d'un mécanisme de **caching** afin d'éviter de refaire les mêmes instructions et commandes à chaque déclenchement de notre workflow. Via le caching on peut, en cas de non modification des instructions préalables, réutiliser les valeurs du workflow précédent jusqu'à l'instruction causant une modification. Ainsi, on pourrait faire en sorte de mettre en cache l'**installation des dépendances** d'une application si celles-ci n'ont pas changé. En plus de réduire le temps d'exécution de notre workflow, on va ainsi potentiellement économiser des ressources / de l'argent.

Mettre en cache les dépendances

Pour faire ce mécanisme de caching ultra-fréquent, GitHub propose encore une fois une action pré-fabriquée pour nous :

```yaml
- name: Cache dependencies
  uses: actions/cache@v3
  with:
    path: ~/.npm
    key: deps-node-modules-${{ hashFiles('**/package-lock.json') }}
```

Si l'on utilise cette étape en amont de n'importe quelle installation des dépendances Node.JS, peu importe le job ou le workflow, alors le cache sera utilisé tant que notre **package-lock.json** n'a pas changé (d'où le branchement de la clé du cache sur son hash).

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Les Variables Globales

Variables d'Environnement

Les variables d'environnement sont généralement utilisées dans les projets de sorte à extérioriser des variables pour éviter qu'elles ne se retrouvent dans le code source ou pour rendre l'application configurable. Pour pouvoir nous en servir dans les workflows il suffit d'utiliser l'attribut `env` (au niveau du workflow ou du job) et de lui passer les clés-valeurs désirées:

```yaml
env:
  MONGODB_DB_NAME: workflow_db
  MONGODB_DB_USER: admin
```

Utiliser les variables dans notre code

En fonction du type de runner que l'on utilise, la méthodologie d'utilisation des variables d'environnement peut différer.

- Ainsi, dans le cas d'utilisation d'un runner de type `ubuntu-latest`, il est possible d'avoir recours à cette syntaxe:

```
run: |
  echo "MONGODB_USERNAME: $MONGODB_USERNAME"
  echo "MONGODB_USERNAME: ${{ env.MONGODB_USERNAME }}"
```

- Dans le cas où l'on veut avoir une récupération de la variables indépendante du runner utilisé, on peut utiliser les expressions de workflow, soit la seconde syntaxe ci-dessus.

Les Secrets

Dans le cas où l'on aurait besoin d'utiliser des données de type mot de passe ou clés API, il vaudrait mieux éviter de pouvoir les rendre accessibles. Pour éviter ceci, il est possible d'utiliser des **secrets**.
Pour pouvoir les stocker, il est possible d'aller du côté du compte d'organisation ou du repository. Dans les settings du repository se trouve l'onglet **Secrets**. Pour en créer un, il suffit d'utiliser l'interface graphique. Pour les utiliser, on peut utiliser l'expression ci-dessous:

```
run: |
  echo "Secret: ${{ secrets.SECRET_KEY }}" # Secret: ***
```

L'affichage sera masqué automatiquement par l'action GitHub.

Environnement de Repository

Dans le cas où l'on utilise des repositories publics ou si l'on a souscrit aux offres relatives à la gestion de GitHub, il est aussi possible d'utiliser des **environnements GitHub**. Ces environnements regroupent par exemple des secrets ou des variables. Pour créer un environnement, il suffit d'aller dans les onglets de configuration de nos repositories.

Via les environnement, il est possible par exemple d'avoir un ensemble de variables de connexion à une base de données en version production ou en test, et d'en changer en modifiant simplement l'environnement utilisé par le workflow.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

La Logique

Les Conditions

Si nos workflows demandent l'exécution d'un job ou d'une action particuliers en fonction de la réussite ou de l'échec d'un job précédent, alors il nous faut avoir recours à des conditions. Pour réaliser ceci, GitHub nous offre plusieurs solutions:

- Le champ `if` pour les jobs et les étapes
- Le champ `continue-on-error` pour les étapes
- L'utilisation d'expressions personnalisées pour les jobs et les étapes

if

Par défault, les jobs dépendant d'un autre job via la clause `needs` (de même que les étapes successives) ne sont exécutés qu'en cas de complétion du job de dépendance. Cependant, il est possible de faire la même chose au niveau des étapes via l'utilisation de la clause `if` tel que:

```
- name: Test code
  id: test-code
  run: npm test
- name: Publish coverage
  if: failure() && steps.test-code.outcome == 'failure'
  uses: actions/upload-artifact@v3
  with:
    name: test-report
    path: test.json
```

Les fonction conditionnelles

Dans le code précédent, nous avons eu besoin, en plus de la condition d'échec, de l'ajout de l'appel d'une fonction. Sans cela, notre workflow aurait provoqué le fonctionnement par défaut de GitHub et notre étape aurait été ignorée. Il existe d'autres fonctions, dont voici le récapitulatif:

- `failure()`: Vrai si échec
- `success()`: Vrai si réussite
- `always()`: Vrai tout le temps
- `cancelled()`: Vrai si job ajourné

Jobs conditionnels

Pour créer l'exécution d'un job uniquement en cas d'échec d'un autre job, alors il est possible de réaliser cela avec la syntaxe ci-dessous. L'utilisation de `needs` est bien entendu présente afin d'éviter le déroulement parallèle du job et un comportement indésirable:

```yaml
job-name:
  needs: [job-a, job-b]
  if: failure()
  runs-on: ubuntu-latest
  steps:
    - name: Something went wrong
      run: echo "${{ toJSON(github) }}"
```

Améliorer le caching

Dans l'action de mise en cache, un output est généré. Cet **output** va nous informer du succès de récupération d'un cache correspondant aux dépendances. Il nous suffit alors de l'utiliser comme condition:

```
- name: Cache dependencies
  id: cache
  uses: actions/cache@v3
  with:
    path: node_modules
    key: deps-node-modules-${{ hashFiles('**/package-lock.json') }}
- name: Install dependencies
  if: steps.cache.outputs.cache-hit != 'true'
  run: npm ci
```

Continue-on-error

Dans le cas où l'on souhaiterait éviter l'arrêt de notre job en cas d'erreur dans l'une de ses étapes, il est possible d'ajouter, dans cette étape, la clause `continue-on-error` avec comme valeur un booléen ou une expression retournant un booléen dans le but de provoquer la continuation du job malgré l'échec de cette étape.
Vis à vis de GitHub, une étape possédant cette clause, même échouée, va produire un **Success**. En cas de non échec de la suite des instruction, l'ensemble du workflow s'en verra produire un statut **Success**.

Matrix Jobs

Dans le cas où l'on veut pouvoir provoquer l'exécution de plusieurs Workflow avec des variantes facilement, il n'est pas nécessaire de créer une vingtaine de fichiers de workflow, il suffit d'utiliser le principe de la **matrice**. En nous servant de cette clause, il est ainsi possible de provoquer X variantes de jobs basées sur X paramètres, comme dans l'exemple ci-après le système d'exploitation et la version de NodeJS.

Par défaut, les jobs possédant des échecs vont causer le non lancement d'autres jobs parallèles, mais il est possible de changer cela via `continue-on-error` au niveau du **job**.

Matrix Job: Exemple

```yaml
name: Matrix Demo
on: push
jobs:
  build:
    strategy:
      matrix:
        node-version: [12, 14, 16, 18]
        os: [ubuntu-latest, windows-latest]
    runs-on: ${{ matrix.os }}
    steps:
      - name: Get Code
        uses: actions/checkout@v3
      - name: Install NodeJS
        uses: actions/setup-node@v3
        with:
          node-version: ${{ matrix.node-version }}
      - name: Install Dependencies
        run: npm ci
      - name: Build project
        run: npm run build
```

Include Exclude

Relatif aux jobs matrices, il est possible d'utiliser les clauses `include` et `exclude` dans le but de provoquer des variantes groupées de jobs ou d'éviter des ensembles de variantes lors de la matrice.

```yaml
strategy:
      matrix:
        node-version: [12, 14, 16, 18]
        os: [ubuntu-latest, windows-latest]
        include:
          - node-version: 18
            os: ubuntu-latest
        exclude:
          - node-version: 12
            os: windows-latest
```

Workflows réutilisables

Dans le cas où l'on aimerait pouvoir utiliser un workflow dans un autre workflow, il peut être intéressant de le rendre réutilisable. Pour permettre à un workflow de se faire appeller dans d'autres workflows, il faut utiliser l'évènement **workflow-call**:

```yaml
name: Reusable Workflow
on: workflow_call
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Output Info
        run: echo "Deploying and Uploading..."
```

Les Inputs

Il peut également être intéressant, dans le cadre de workflows, d'avoir la possibilité de donner des **variables d'entrée** afin d'altérer le fonctionnement du workflow pour chaque exécution sans modifier le fichier **.yml** en soi ou les **variables d'environnement**. Pour cela, on peut utiliser la clause `inputs` dans l'appellé et `with` dans l'appellant:

```
workflow_call:
    inputs:
      artifact-name:
        type: string
        description: Name of the deployable artifact file
        required: true
        default: 'dist-file'
```

Secrets de workflow

Il est possible d'avoir des secrets au niveau du workflow via la clause `secrets` de la même façon qu'`inputs`:

```yaml
secrets:
  my-secret:
    type: string
    required: true
```

Il faudra ensuite dans les workflows utiliser des secrets en tant que valeur dans la clause `with`:

```yaml
uses: ./.github/workflows/my-workflow.yml
with:
  my-secret: ${{ secrets.MY_SECRET }}
```

Les Outputs de workflow

Il est aussi possible de définir des sorties pour notre workflow via la clause `outputs`:

```yaml
outputs:
  result:
    description: Result of the deploy action
    value: ${{ jobs.deploy.outputs.outcome }}
jobs:
  deploy:
    outputs:
      outcome: ${{ steps.set-result.outputs.step-result }}
    steps:
      - name: Set result
        id: set-result
        run: echo "step-result=success" >> $GITHUB_OUTPUTS
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Docker

Jobs dans des conteneurs

Praticité

Dans un conteneur Docker, il est fréquent qu'un ensemble de **dépendances** ainsi qu'un **environnement pré-fait** soit disponible. De par leur utilisation dans GitHub Actions, nous ne sommes ainsi plus limités à seulement X runners et X environnements de repositories, mais nous pouvons avoir autant d'environnement que l'on le souhaite.

Ainsi, de par l'utilisation de Docker en plus de GitHub Actions, nous n'avons plus besoin d'utiliser plusieurs instructions pour mettre en place notre runner avant de réaliser les actions qui nous intéressent, on peut simplement récupérer une **image Docker** !

Executer les jobs dans un conteneur

Pour exécuter les jobs dans un conteneur, on va simplement utiliser la clause `container`. Il est possible de spécifier l'image directement, ou d'utiliser des attributs en cas de nécéssité de variables d'environnement:

```
container: node:16-alpine
```

ou

```
container:
      image: node:16-alpine
      env:
        CONTAINER_ENV_VARIABLE_A: value
        CONTAINER_ENV_VARIABLE_B: value
```

Service Containers

Il est possible d'avoir besoin de dépendances externes dans un workflow, comme par exemple d'une base de données pour faire des tests unitaires.

Dans le cas où l'on aimerait éviter l'utilisation de la base de données de production, il est possible de brancher le projet, durant les tests, sur un conteneur possédant une base de données similaire. Ce genre d'utilisation d'image crée ce que l'on appelle des **conteneurs de services**. De la sorte, on peut disposer d'une dépendance à un moment précis et ne pas avoir à la désinstaller plus tard.

Configuration

Pour utiliser un conteneur de service, il est nécessaire d'alimenter la clause `services` au sein d'un job. Cette clause va contenir des sous objects devant au moins avoir en attribut leur `image`.
Il est ensuite possible de spécifier si nécessaire les variables d'environnement (`env`):

```yaml
services:
  service-name:
    image: image-name
    env:
      VAR_A: value
```

Communiquer avec les conteneurs

Pour ensuite accéder aux conteneurs, il est possible, en cas d'exécution de notre job dans un conteneur également d'utiliser simplement le nom du conteneur de service. En effet, GitHub Actions configure automatiquement un **réseau** permettant aux conteneurs de communiquer durant le job.

Dans le cas contraire, on peut utiliser l'attribut `ports` de la même façon que l'on le ferait via un `docker-compose.yml` de sorte à exposer les ports du conteneur au runner exécutant le job :

```yaml
ports:
  - "1234:1234"
```

Exemple avec MongoDB

```yaml
test:
    environment: testing
    runs-on: ubuntu-latest
    env:
      MONGODB_CONNECTION_PROTOCOL: mongodb
      MONGODB_CLUSTER_ADDRESS: localhost:27017
      MONGODB_USERNAME: root
      MONGODB_PASSWORD: password
      PORT: 8080
    services:
      mongodb:
        image: mongo
        env:
          MONGO_INITDB_ROOT_USERNAME: root
          MONGO_INITDB_ROOT_PASSWORD: password
        ports:
          - "27017:27017"
```

Les Actions

Actions personnalisées

La raison d'utilisation d'actions personnalisées est généralement en lien avec ces deux réflexions :

- On chercher à **simplifier** nos workflows avec une action regroupant plusieurs étapes, pour potentiellement l'utiliser dans d'autres workflows.
- Il **n'existe pas** (encore) d'action résolvant le problème que l'on rencontre dans notre workflow et nous n'avons alors pas le choix que de faire la nôtre.

Types d'actions

Il existe plusieurs types d'actions que l'on peut réaliser, peu importe le résultat voulu:

- **Javascript Actions**: On écrit la logique de l'action via du code Javascript (Node.js).
- **Docker Actions**: Via l'utilisation d'un conteneur Docker, on utilise le langage que l'on veut.
- **Composite Actions**: On combine des workflows et des étapes GitHub Action dans une action personnalisée.

Composite Actions

Dans un premier temps nous allons voir les actions les plus simples à appréhender, les **Composite Actions**. Pour créer une action, il faut créer un fichier **.github/actions/action-name/action.yml**:

```yaml
name: Action name
description: Description
runs:
  using: composite
  steps:
```

Pour pouvoir les utiliser, il faudra utiliser encore une fois un lien relatif du style `uses: .github/actions/action-name`

Inputs et Outputs (1/2)

Pour utiliser ces deux composantes, il nous suffit, tout comme sur les workflows réutilisables, d'avoir les clauses `inputs` et `outputs` et de nous en servir de la même façon:

```yaml
name: Action name
description: Description
inputs:
  variable-name:
    description: Variable description
    required: false
    default: 'dist'
runs:
  using: composite
  steps:
    - name: Print input
      run: echo ${{ inputs.variable-name }}
      shell: bash
```

Inputs et Outputs (1/2)

Pour la section `outputs`, on peut imaginer avoir:

```yaml
name: Action name
description: Description
outputs:
  variable-name:
    description: Output variable description
    value: ${{ steps.step-id.outputs.variable-name }}
runs:
  using: composite
  steps:
    - name: Set output
      id: step-id
      run: echo 'variable-name=value' >> $GITHUB_OUTPUTS
      shell: bash
```

Javascript Actions

Une autre façon de faire des actions personnalisées est de les rédiger en Javascript. Ces actions sont, de par leur utilisation de Node.js, compatibles avec tous les **packages npm**:

```yaml
name: Action name
description: Action description
runs:
  using: node20
  pre: setup.js # Code à lancer en amont de l'action
  main: main.js # Code à lancer pour l'action
  post: cleanup.js # Code à lancer en aval de l'action
```

Pour les dépendances **Node.js**, il nous faut **@actions/core**, **@actions/exec** et **@actions/github**.

Code Javascript

Il ne nous reste ensuite plus qu'à écrire la partie Javascript de notre première action, celle-ci sera provoquée par l'action comme décrit dans le fichier **action.yml**:

```
// main.js
const core = require('@actions/core');
const github = require('@actions/github');
const exec = require('@actions/exec');

function run() {
    core.notice('Hello from my custom Javascript Action!');
}

run();
```

Les Inputs

Dans le cas où notre action aurait besoin d'inputs, il est possible, dans le fichier **action.yml**, de les récupérer comme dans le cas d'actions composites. Pour cependant y avoir également accès au sein de notre code Javascript, il va falloir utiliser, depuis le package **core**, la fonction `.getInput()`:

```
const { notice, getInput } = require('@actions/core');

function run() {
    const variableName = getInput('variable-name', {required: true});

    notice(`Hello ${variableName}!`);
}

run();
```

Exécuter des commandes

Si l'on a besoin, au sein de notre action, d'utiliser des commandes bash, il est possible, via le package **exec**, d'utiliser la fonction `exec()` de la façon suivante :

```
const {  getInput } = require('@actions/core');
const { exec } = require('@actions/exec');

function run() {
    const searchPath = getInput('search-path', {required: true});

    exec(`ls ${searchPath}`);
}

run();
```

Générer des Outputs

Si l'on a désormais envie d'avoir des valeurs en sortie de notre action, il est possible de générer des outputs dans le code Javascript via `setOutput()`:

```
const { getInput, setOutput } = require('@actions/core');

function run() {
    const firstName = getInput('firstname', {required: true});
    const lastName = getInput('lastname', {required: true});
    const fullName = firstName + " " + lastName;

    setOutput('full-name', fullName) // ::set-output
}

run();
```

Docker Actions

Enfin, il est possible de faire des actions via Docker. Dans cette façon de fonctionner, les **inputs** sont accessible via l'utilisation de variables d'environnement commençant par `INPUT_` et possédant un nom correspondant au nom de l'input dans le fichier **action.yml** mis en **UPPERCASE**.

Pour les **outputs**, il va falloir parcourir le fichier `GITHUB_OUTPUT`, disponible via les variables d'environnement, et y ajouter des ensembles clés-valeurs.

Partager nos actions

Dans le but de réutiliser nos actions à plusieurs endroits, il nous faut les stocker dans un **repository public**. Pour ce faire, cela n'est pas bien compliqué, il suffit de placer tout le contenu d'un dossier du type **.github/actions** directement à la racine du repository. Une fois fait, il va falloir ajouter un tag à notre repository, par exemple:

```
git tag -a -m "My action release v1.0" v1

git push --follow tags
```

Pour utiliser l'action, il suffit désormais d'utiliser:

```
uses: account-name/action-name@v1
```

TOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Sécurité

Soucis de sécurité

Il est important d'être conscient lorsque l'on réalise du CI/CD avec **GitHub Actions** de quelques notions de sécurité:

- **Script Injection**: Des valeurs / commandes peuvent être injectées dans nos workflows par autrui pour altérer leur fonctionnement et les détourner
- **Malicious Third-Party Actions**: Une action peut réaliser n'importe quelle logique, dont des modifications malicieuses (Attention à vos sources !)
- **Permission Issues**: Attention à ne pas donner trop de droits aux mauvaises personnes !

Script Injection

Pour par exemple nous protéger des injections de script lors des workflow provoqués par des **issues**, il est préférable d'utiliser des variables d'environnement et de les brancher sur le titre de l'issue:

```
run: |
  echo "title=${{ github.event.issue.title }}"
  if [[$title == 'bug']]; then
```

devient:

```
env:
  TITLE: ${{ github.event.issue.title }}
run: |
  if [[$TITLE == 'bug']]; then
```

Third-Parties Actions

Lorsque l'on utilise des actions provenant du marketplace, il est important de privilégier les actions faites par des créateurs vérifiés. De la sorte, on a beaucoup moins de chances que ses actions soient malicieuses.

Si l'on souhaite quand même utiliser des actions ne provenant pas de telles sources, alors il est intéressant d'aller voir dans le code source disponible sur le repository de l'action (après tout, chaque action partagée se trouve dans un repository publique).

Permissions de Job

Dans le cas où l'on veut, au sein d'un job, limiter les permissions de ses commandes relatives à la manipulation GitHub, il est possible de le faire assez facilement via la clause `permissions` sur les jobs ou sur le workflow:

```yaml
jobs:
  my-job:
    runs-on: ubuntu-latest
    permissions:
      issues: write
    steps:
      ...
```

Par défaut, les workflows / jobs ont accès à toutes les permissions.

La variable $GITHUB_TOKEN

Cette variable d'environnement est générée automatiquement par GitHub pour faciliter l'utilisation de l'**API GitHub** au sein de nos workflows (via les headers). Ce token est automatiquement alimenté via les permissions configurées préalablement au moyen de la clause `permissions`. Rappellons que par défaut, ce token aurait toutes les permissions. Il faut donc faire attention, en cas de son utilisation, de ne pas avoir trop de permissions afin d'éviter les manipulations impromptues de nos repositories par un workflow où aurait lieu, par exemple, une injection de script.

OpenID

Dans le cadre de workflows faisant usage de plateformes externes à l'environnement GitHub, il est fréquent d'avoir besoin de **credentials** tel qu'un ID et une clé privée.
Afin de gérer les permissions de notre workflow relativement à cet utilisation, il faut idéalement mettre les clés en tant que **variables d'environnement**.

Pour encore plus de sécurité, il faudrait idéalement utiliser **OpenID Connect** afin créer des permissions temporaires et limitées à un objectif donné.

Exemple avec AWS (1/2)

Dans le cas d'utilisation d'AWS, pour obtenir une permission via OpenID, il va falloir passer par un **IAM**. Dans l'onglet concerné, il va falloir ajouter un provider (ici **OpenID**). en valeur provider URL, il va falloir ajouter celle mentionnée dans GitHub:

```
https://token.actions.githubusercontent.com
```

Il faudra faire de même avec l'Audience:

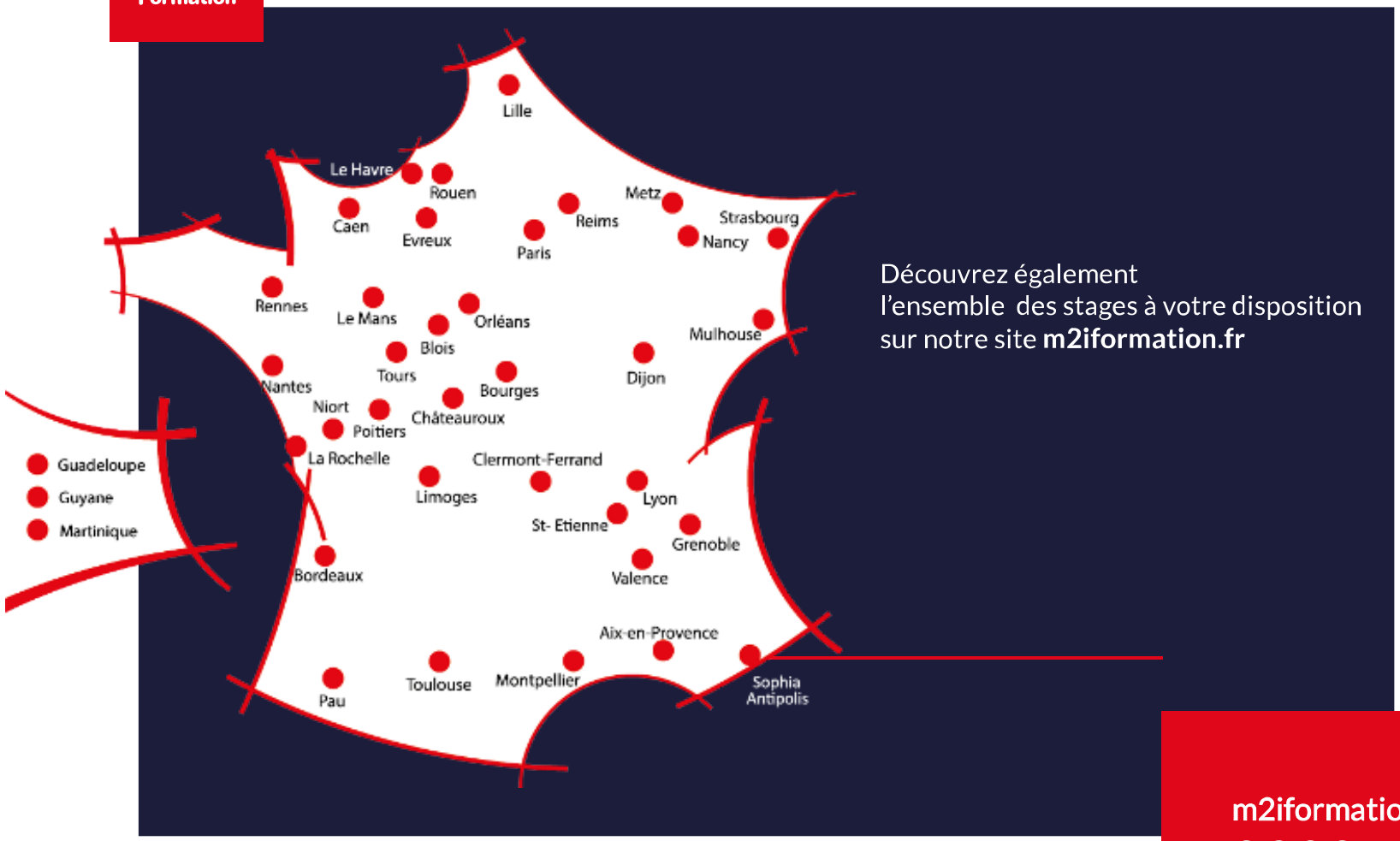
```
sts.amazonaws.com
```

Il faut ensuite simplement ajouter des rôles à notre entité dans l'interface IAM.

Exemple avec AWS (2/2)

Côté workflow, il va nous falloir ajouter une étape pour obtenir les permissions via OpenID. Pour cela, il existe une action prévue à cet effet:

```yaml
permissions:
  id-token: write
  contents: read
steps:
- name: Get AWS permissions
  uses: aws-actions/configure-aws-credentials@v1
  with:
    role-to-assume: <AWS Role ARN>
    aws-region: eu-west-3
```



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr