Module IINF170 : Déploiement Continu/CD

Module IINF170 : Déploiement Continu/CD

Lien du cours en pdf

Description

Compétences visées

- Apprendre à déployer une application de façon automatisé dans une démarche DevOps
- Connaitre les outils qui servent dans le cadre du CI/CD

Objectifs généraux du module

- Appréhender l'outil Gitlab pour créer un pipeline.
- Utiliser Git pour créer un gestionnaire de versions
- Utilisation de Jenkins

Introduction / Définitions

Le Déploiement Continu (CD pour "Continuous Deployment") fait partie des pratiques DevOps.

Il vise à simplifier et accélérer le processus de déploiement ou de mise en production d'une application, en s'appuyant sur des outils d'automatisation.

Le terme a été introduit au début des années 2000, parallèlement aux méthodologies Agile. Le livre "Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation" a formalisé ses bases, bien que son titre mentionne la Livraison Continue (CD également, mais pour "Continuous Delivery").

Plus généralement, on parle de CI/CD. Ici, CD représente à la fois la "Livraison Continue" et le "Déploiement Continu".

C'est le bon moment pour définir plus précisément les trois termes d'intégration continue, livraison continue, déploiement continu, et leurs inter-relations.

- L'Intégration fait référence au processus qui vise à rassembler tous les composants d'un projet logiciel, et à vérifier qu'ils fonctionnent bien ensemble.
- La Livraison est la phase d'assemblage de tous les composants du projet en artefacts exécutables (ex. : un "bundle" d'application mobile, `.ipa` pour iOS, `.apk` ou `.aab` pour Android).
- Le Déploiement consiste à rendre votre application disponible pour ses utilisateurs finaux (ex. : mise en ligne sur un store et validation par Apple ou Google).

Le terme *continu* est un peu exagéré : un projet logiciel, même dans une très grande entreprise, n'est pas réellement intégré, livré ou déployé en continu. *Continue* signifie plutôt effectuer ces actions à chaque fois que du nouveau code est poussé vers le dépôt central de gestion de versions (Git, Mercurial, etc.).

Comment ces trois pratiques se situent-elles les unes par rapport aux autres ?

Il y a un chevauchement substantiel entre elles : le Déploiement Continu englobe la Livraison Continue, qui à son tour englobe l'Intégration Continue. Ce diagramme très simplifié représente leur étendue :

CI/CD overview

Vue d'ensemble de CI/CD

CI/CD vise à remplacer les procédures manuelles d'intégration, de livraison et de déploiement par des procédures automatisées. Cela est particulièrement bénéfique pour les grands projets, mais est intéressant même pour de petits projets personnels.

Afin d'automatiser les différentes étapes d'un CI/CD complet, nous allons utiliser des outils disponibles prêts à l'emploi. Il existe une très grande variété d'options. Beaucoup d'entre eux sont basés sur le cloud, mais d'autres peuvent être auto-hébergés.

Parmi eux, on peut citer :

- GitLab CI/CD
- GitHub Actions
- Bitbucket Pipelines
- Travis CI
- Jenkins

Ces outils implémentent le concept de *pipeline* : une série d'étapes à compléter pour effectuer l'intégration, la livraison ou le déploiement.

Si l'une des étapes échoue, le pipeline ne passera pas à la suivante.

Voici un exemple de pipeline relativement complexe, mais indépendant des technologies.

CI/CD overview

Quelques explications sur les "labels" de ce diagramme :

- Deps. Check → **Vérification des Dépendances** : Vérifie si les dépendances de l'application sont à jour (éviter les vulnérabilités au niveau des modules tiers).
- Lint → **Linting** : Analyse statique. Par exemple, utilisation de ESLint pour assurer la qualité du code.
- Code Quality → **Qualité du Code** : Analyse statique avancée. Utilisation d'outils comme SonarQube pour détecter des vulnérabilités, du code dupliqué, etc.
- **Tests** : Utilisation de frameworks de test comme Jest (JS/TS), JUnit (Java), PHPUnit, etc.
- Build → **Construction** : Compilation et préparation de l'application pour le déploiement.
- Vuln. Check → **Vérification des Vulnérabilités** : Analyse de sécurité et évaluation des vulnérabilités, cette fois au niveau des artefacts. Par ex. : détection de virus/trojan dans bundles d'app mobiles, de vulnérabilités dans des images Docker, etc.
- Deploy → **Déploiement** : Mise en production de l'application.

Vue d'ensemble des activités du premier jour

Nous allons nous concentrer sur le déploiement continu pour commencer, bien que nous montrions au moins un exemple de processus qui s'inscrit dans la partie intégration continue.

À la fin de la journée, vous aurez — si tout va bien 🤞 — déployé une simple application web, via un processus entièrement automatisé.

Nous passerons par :

- la mise en place d'une application Node.js d'exemple,
- son hébergement dans un dépôt GitLab,
- la création d'un pipeline GitLab CI/CD,
- la construction d'images Docker comme étape du pipeline,
- le déploiement de l'application sur Heroku en utilisant l'image Docker

Selon le temps, nous pourrons également aborder beaucoup d'autres points, tels que :

- des outils permettant d'éviter de *committer* et de pousser du code défectueux vers le dépôt, en mettant en place des "hooks Git de pré-commit".
- l'utilisation de bonnes pratiques

ℹ️ ***Note** : je suis ouvert à vos **propositions**, si vous voulez travailler sur une autre app, même si je préférerais (largement) qu'elle soit :*

- *en JavaScript ou **plutôt TypeScript** (intérêt : avoir une étape de build / compilation !)*
- *plutôt du backend (pour explorer le déploiement d'un conteneur pour l'app, et d'un autre pour la BDD, ainsi que la question des migrations de la BDD)*
- *d'un scope relativement restreint (pour ne pas passer tout votre temps à débugger du code complexe)*

Avant de commencer

GitLab CI vs `gitlab-ci-local`

Comme alternative, ou plutôt en complément pour celles et ceux qui auraient le temps, il est possible de faire fonctionner le pipeline GitLab localement, avec un outil que vous avez a priori déjà utilisé : `gitlab-ci-local`.

J'ai personnellement testé les deux options. Si vous avez déjà installé `gitlab-ci-local`, il peut servir :

- pour faire ses premiers pas (ou des révisions) dans l'écriture de pipelines,
- on peut faire a priori tout ce qu'on peut faire dans GitLab, y compris pousser des images.
- c'est a priori une très bonne option, même j'ai eu à résoudre quelques problèmes pour que mon pipeline fonctionne aussi bien que sur GitLab.
- cela peut vous emmener jusqu'au build de votre image Docker. Il serait possible de la pousser vers un registre auto-hébergé, mais pour ce qui est de la "mise en production", les options sont nombreuses, et variables suivant les OS.

Écriture des pipelines

Un pipeline est composé d'étapes (*stages*) qui peuvent s'enchaîner de différentes façons. On rencontre aussi le terme *jobs* : dans la terminologie de GitLab, chaque *stage* du pipeline donne lieu à l'exécution d'un *job*.

GitLab CI propose deux méthodes d'écriture des fichiers `.gitlab-ci.yml`. La différence réside dans la gestion des dépendances et l'exécution parallèle des jobs.

Attention, les exemples ci-dessous utilisent un certain ordre entre "test" et "build", **qui n'est pas** celui que vous aurez à suivre.

1. Exécution séquentielle :

```yaml
stages:
  - build
  - test

build_job:
  stage: build
  script:
    - echo "Building the project"

test_job:
  stage: test
  script:
    - echo "Running tests"
```

2. Exécution parallèle :

```yaml
build_job:
  script:
    - echo "Construction du projet"

test_job:
  needs: [build_job]
  script:
    - echo "Exécution des tests"

other_job:
  needs: [build_job]
  script:
    - echo "Autre tâche"
```

Le mot-clé `needs` crée une dépendance. `test_job` et `other_job` ne s'exécuteront qu'après la fin de `build_job`. En revanche, ils s'exécuteront en parallèle.

Plus généralement, les jobs sans dépendances spécifiées ou appartenant au même stage peuvent s'exécuter en parallèle.

> ⚠️ *Gardez bien à l'esprit ces deux approches. Pour ce qu'on souhaite faire aujourd'hui, l'une semble peut-être plus évidente. Il est probablement un peu tôt pour y réfléchir* 😅*. Et le passage de l'une à l'autre se fait rapidement !*

Gestion des Variables d'Environnement dans GitLab CI/CD

> *Vous pourrez revenir à cette section plus tard, quand il vous semblera nécessaire de faire appel à des variables, plutôt que de "hardcoder" des informations dans vos fichiers* `.gitlab-ci.yml`.

Dans GitLab CI/CD, les variables d'environnement sont essentielles pour gérer des informations sensibles ou spécifiques à l'environnement. Par exemple :

- nom de l'application et token d'accès Heroku,
- nom d'utilisateur et token d'accès Docker Hub.

Vous pouvez les définir dans l'interface GitLab sous "CI / CD Settings".

Il est d'usage d'activer l'option "Protect variable" (active par défaut) pour que ces variables soient disponibles uniquement dans les branches protégées (comme `main` ou `dev`).

Cela assure la sécurité, car les branches non protégées n'auront pas accès à ces variables, prévenant ainsi les déploiements accidentels ou non autorisés.

Vous pouvez les déprotéger *temporairement* pour des tests sur d'autres branches (notamment pendant que vous expérimentez avec les pipelines).

Dans ce cas, assurez-vous de les protéger à nouveau après vos tests pour la sécurité des branches principales (`main` ou `dev`).

Détail des étapes du pipeline CI/CD

Pour chacune de ces étapes, on vous propose un **objectif** et des **pistes d'exploration**.

⚠️ ***TRÈS important*** *: ne restez pas bloqué·e·s pendant des heures, surtout si un point explicité ci-dessous n'est pas clair ! L'idée est de ne pas (trop) vous tenir la main, et de vous laisser en autonomie. Mais il y a probablement des choses dont j'ai sous-estimé la complexité. Donc, si vos recherches ne vous mènent nulle part - ou simplement pour discuter d'un des points listés,* ***demandez*** *(à moi ou à vos camarades !).*

1. Configuration de base du projet

ℹ️ *Cette étape pourrait être dispensable… Mais si vous n'avez jamais mis en place de projet Node.js avec TypeScript, elle peut constituer un bon exercice. Mais je ne veux pas qu'on y passe trop de temps ! J'avais prévu, à la base, de vous fournir la base de l'application Node.js déjà configurée.*

- **Objectif** : Initialiser un projet Node.js avec Git et installer les dépendances.
- **Pistes d'exploration** :
  - Rechercher la configuration de base d'un projet Node.js, l'utilisation de `.gitignore`, et la gestion des dépendances avec `yarn`.
  - Commencer à écrire votre fichier pipeline GitLab CI, avec comme première étape, l'installation des dépendances.

2. *Build* du projet

⚠️ *C'est une étape critique et un peu ardue.*

Le *build*, dans le pipeline, peut être divisé en deux sous-étapes :

- **compilation** des sources avec TypeScript (cette étape),
- puis **construction** (*build*) de l'image Docker (l'étape 3).

Les deux pourraient être rassemblées en une seule, mais cela peut mener à la construction d'une image Docker spécifique, ce qui est une tout autre affaire !

Pour ce qui s'agit de **cette** étape :

- **Objectif** : Configurer le projet pour qu'il *build* avec succès.
- **Pistes d'exploration** :
  - Comprendre la compilation TypeScript, et certaines options du fichier `tsconfig.json`, notamment l'option `rootDir` (emplacement des fichiers source), et l'option `outDir` (pour éviter que les fichiers `.js` générés se retrouvent dans votre répertoire `src`).
  - Configurer un script de build dans `package.json`.
- **Difficultés potentielles** : les étapes (*stages*) d'un pipeline GitLab CI sont indépendantes ; l'étape de build TypeScript va avoir besoin des dépendances listées dans votre `package.json` ; mais les `node_modules` ayant été installés à l'étape précédente, comment y accéder dans celle-ci ? La réponse à cette question servira pour plus tard.
- 👉 **Une fois cette étape accomplie** (compilation réussie), bravo, vous avez jeté les première bases 👏 ! Vous pouvez commencer à pousser des mises à jour du code depuis votre machine locale, et à vérifier qu'il compile toujours dans le pipeline.

3. Dockerisation, *build* de l'image et *push* sur un registre

- **Objectif** : Créer un Dockerfile pour conteneuriser l'application.

- **Pistes d'exploration** :
    - Un récapitulatif / aide-mémoire Docker est fourni ici.
    - Se plonger dans les bases de Docker, écrire un Dockerfile, et comprendre les concepts de conteneurisation.
    - Il existe diverses manières de conteneuriser une application Node.js. Est-ce un conteneur orienté développement ou production ? Sachant qu'il s'agit ici de Déploiement Continu, vous connaissez la réponse 😉 .
    - Cette question est d'ailleurs liée au fait qu'on ait divisé le *build* en deux étapes. Que va-t-on faire du résultat de la précédente ?
    - Le *push* interviendra à cette étape, mais **pas vers le Docker Hub !** On utilisera le registre (*registry*) Heroku.
    - Faut-il pousser l'image vers un registre Docker quand on travaille sur une branche transitoire ?
    - Utiliser les variables GitLab CI/CD dans les pipelines.

4. Déploiement sur Heroku

- **Objectif** : Déployer le conteneur Docker sur Heroku.
- **Pistes d'exploration** :
    - Apprendre sur Heroku, créer un compte.
    - Installer la CLI Heroku (https://devcenter.heroku.com/articles/heroku-cli) .
    - Créer une app Heroku, soit via la CLI (nécesite d'être authentifié), soit depuis le dashboard web. Je vous encourage à utiliser la première méthode.
    - Explorer le déploiement sur Heroku via des images Docker — qui tranche avec une méthode plus "évidente" (que j'ai personnellement beaucoup utilisée), consistant à lier directement un dépôt Git à Heroku.
    - Documentation sur cette méthode de déploiement (https://devcenter.heroku.com/articles/container-registry-and-runtime)
    - Utiliser l'API de Heroku pour la mise en production du conteneur. Vous aurez ici besoin d'un *token* d'authentification (ce qui est différent de l'authentification avec `heroku login` !).

Étapes bonus

Deux Environnements : Staging et Production

Dans un workflow CI/CD, il est courant d'avoir deux environnements distincts : staging et production. Le staging est utilisé pour les tests et la validation avant le déploiement en production. Cela implique deux applications Heroku distinctes, chacune avec sa propre variable d'environnement pour le nom d'application.

- **Objectif** : déployer vers *staging* ou *production* en fonction de la branche poussée.
- **Pistes d'exploration** :
    - créer une deuxième application Heroku qui servira de *staging*
    - puis modifier son pipeline pour déployer vers `staging` si on est sur la branche `develop`, et vers `production` si on est sur `main`
    - cet article (https://blog.frankel.ch/conditional-build-gitlab/) pointe vers une solution à ce problème

Linting

> ℹ️ *Normalement, suivant l'ordre du pipeline d'exemple décrit plus haut, cette étape devrait intervenir bien avant le build ! Mais comme le focus du jour est sur le déploiement plutôt que sur l'intégration, je préfère qu'on garde cette étape pour celles et ceux qui auront de l'avance.*

Ceci étant dit, si le compilateur TypeScript détecte beaucoup des erreurs qu'ESLint détecte, chacun a son utilité propre.

Il est essentiel de connaître ESLint, et de savoir l'installer… surtout si vous récupérez un jour une vieille *codebase* JavaScript. ESLint peut alors détecter des erreurs qui pourraient mener à des crashs de votre app en production (c'est du vécu !).

- **Objectif** : Configurer ESLint pour les contrôles de qualité du code.
- **Pistes d'exploration** : Apprendre sur ESLint, son intégration dans les projets Node.js, et configurer les règles de linting. Découvrir où cela devrait s'insérer dans la pipeline.

Hooks de pré-commit

Ce n'est pas une partie du pipeline à proprement parler. C'est plutôt quelque chose destiné à être utilisé sur votre station de travail du développeur : les hooks de pré-commit peuvent nous empêcher de commettre et pousser du code cassé. cela évite de déclencher le pipeline GitLab CI "pour rien".

- **Objectif** : mettre en place des hooks de pré-commit pour exécuter eslint à chaque fois que nous sommes sur le point de commettre notre code.
- **Pistes d'exploration** : rechercher la configuration des hooks de pré-commit avec husky. Trouver comment utiliser cet outil pour exécuter des scripts définis dans la section "scripts" du `package.json` de l'application.

Vérification des paquets obsolètes

- **Objectif** : Comprendre la gestion des paquets et maintenir les dépendances à jour.
- **Pistes d'exploration** : petit indice, vous pouvez explorer les exemples de l'outil gitlab-ci-local (https://github.com/firecow/gitlab-ci-local) .

Écriture de tests

> ℹ️ *Les tests feront l'objet d'un autre module que nous ferons ensemble à partir de février. Ce n'est donc pas le "focus" du jour, mais cela peut constituer une bonne préparation !*

- **Objectif** : Apprendre les bases des pratiques de test
- **Pistes d'exploration** : effectuer des recherches sur les frameworks de test JavaScript/TypeScript, se concentrer dans un premier temps sur l'écriture de *tests unitaires* (sur des fonctions par ex.) ; si le temps le permet, explorer les tests d'intégration pour Express.

Autres points

Ils ne sont pas des choses à faire dès aujourd'hui, rassurez-vous ! Mais des pistes si vous vous voulez creuser plus loin.

Ces points ne concernent pas nécessairement l'écriture de pipelines, mais d'autres aspect de la gestion d'une mise en production, et des bonnes pratiques.

Nous aborderons ces problématiques une prochaine fois.

Rollback Release sur Heroku :

- **Objectif** : Comprendre comment annuler un déploiement sur Heroku.
- **Pistes d'exploration** : Utilisation de la CLI Heroku pour les rollbacks.

DB sur Heroku (Postgres) :

- **Objectif** : Configurer et utiliser une base de données Postgres sur Heroku.
- **Pistes d'exploration** : Création et gestion de la base de données dans l'environnement Heroku.

Migrations DB :

- **Objectif** : Gérer les migrations de base de données.

- **Pistes d'exploration** : Utilisation d'outils de migration comme Sequelize ou Knex.

Git Flow / Branching / Naming Issues :

- **Objectif** : Comprendre et appliquer les bonnes pratiques de Git.
- **Pistes d'exploration** : Exploration de Git Flow, conventions de nommage des branches et des issues.

Tagging des Images Selon les Branches :

- **Objectif** : Taguer les images Docker en fonction des branches de développement.
- **Pistes d'exploration** : Configuration des pipelines CI pour appliquer des tags dynamiques aux images Docker.

Obtention et installation du projet exemple

ℹ️ *À nouveau, il est possible d'explorer un autre projet.*

Le projet exemple sur lequel nous allons travailler est une application e-commerce *très* simplifiée. C'est une application web de style traditionnel, c'est-à-dire qu'elle ne repose pas sur une architecture moderne telle qu'une API REST ou GraphQL et une SPA (Single Page Application) comme on pourrait en construire avec Angular, React ou Vue.js.

C'est une application Node.js construite sur le framework Express (https://expressjs.com) , qui servira des pages HTML.

Prérequis

- **Version de Node.js** : puisque, au moment de la rédaction, Node.js 20 est en phase de support à long terme actif, vous devriez l'avoir installé (ou au moins la version 16). Vous pouvez vérifier quelle version est installée sur votre ordinateur en ouvrant un terminal et en exécutant `node -v`. Si vous ne l'avez pas ou si votre version est trop ancienne, vous pouvez l'installer en utilisant nvm (https://github.com/nvm-sh/nvm) (Linux, macOS) ou nvm-windows (https://github.com/coreybutler/nvm-windows) .
- **Yarn** : Yarn (https://yarnpkg.com) est une alternative populaire à NPM, souvent plus performante. Nous l'utiliserons pour la gestion des dépendances. Vous pouvez vérifier s'il est installé en exécutant `yarn -v`. S'il ne l'est pas, exécutez simplement `npm i -g yarn`.

Installation

:warning: [TODO] remplacer l'origine du repo

- Allez dans votre répertoire habituel de projets, clonez le projet : `git clone https://gitlab.com/bhubert/ipi-cicd-example.git`
- `cd ipi-cicd-example`
- Exécutez `yarn`

Travailler sur le projet

Le projet n'a été que très peu configuré. C'est à vous d'implémenter les fonctionnalités de base. L'objectif ici n'est pas de (ré)apprendre Node.js, mais d'ajouter des fonctionnalités très simples, et de vérifier si notre application se construit, localement et dans une pipeline CI/CD.

Exemples de fonctionnalités à implémenter

- Afficher la liste des produits sur la page d'accueil
- Ajouter un produit à la liste
- Ajouter un produit au panier
- Afficher le contenu du panier sur une page "Panier"

Données et persistance

Pour commencer, les produits seront stockés dans un simple tableau d'objets, peuplé à partir d'un fichier JSON. Afin de garder les choses simples, nous ne sauvegarderons pas la liste mise à jour sur le disque lorsque nous ajoutons un nouveau produit, ce qui a une conséquence évidente : la liste des produits sera réinitialisée à chaque fois que nous redéployons l'application.

Nous introduirons la persistance des données plus tard si le temps le permet, mais comme Heroku est un moyen facile et pratique de déployer des applications, nous pourrions y arriver.

Récapitulatif / Aide-mémoire sur Docker

Docker est un outil de conteneurisation qui permet de packager une application et ses dépendances dans un conteneur isolé. Cela facilite le déploiement et la portabilité des applications. Docker s'applique à divers types d'applications, pas seulement aux applications web, mais aussi aux bases de données, aux applications en ligne de commande, etc.

Exemple

> *Nécessite d'avoir installé Node.js*

Commencez par vous placer dans un nouveau répertoire.

Lancez alors : `npm i express`.

Créez un fichier `index.js` avec ce contenu :

```
const express = require("express");
const app = express();
const port = 3000;

app.get("/", (req, res) => {
  res.send("Hello World!");
});

app.listen(port, () => {
  console.log(`Example app listening on port ${port}`);
});
```

Puis créez un fichier `Dockerfile` :

```
# Image de base
FROM node:20-alpine

# Répertoire où seront copiés les fichiers,
# et d'où sera lancée l'application.
WORKDIR /app

# Copie du package.json et package-lock.json.
# Ces fichiers changeant a priori moins souvent que les fichiers de l'app,
# on les copie dès maintenant, afin de bénéficier de la mise en cache des
# "layers" constituant l'image (s'ils ne sont pas modifiés, seul le dernier COPY
# devra être ré-effectué au prochain build).
COPY package.json package-lock.json .

# Installation des dépendances
RUN npm install

# Copie de l'app (ici un seul fichier).
COPY index.js .
```

```
# Commande lancée au démarrage du conteneur
CMD node index.js
# Syntaxe alternative
# CMD ["node", "index.js"]
```

Construisez l'image Docker :

```
docker build -t express-hello .
```

Exécutez le conteneur :

```
docker run -d -p 3000:3000 express-hello
```

Commandes Docker Essentielles

Créer une image Docker :

- `docker build -t mon-app .` : Construit une image Docker à partir d'un `Dockerfile`.

Taguer une image :

- `docker tag mon-app:latest mon-app:v1.0` : Tag l'image avec une nouvelle version.

Exécuter un conteneur :

- `docker run -d -p 3000:3000 mon-app` : Lance le conteneur.

Lister les conteneurs/images :

- `docker ps` : Liste les conteneurs actifs.
- `docker ps -a` : Liste tous les conteneurs.
- `docker images` : Liste toutes les images.

Supprimer conteneurs/images :

- `docker rm [CONTAINER_ID]` : Supprime un conteneur spécifique.
- `docker rmi [IMAGE_ID]` : Supprime une image spécifique.

Push une image sur Docker Hub :

- `docker push mon-app` : Envoie l'image sur Docker Hub (nécessite un login préalable).

Configuration du projet exemple

> *Cette section sert de référence : elle décrit la configuration initiale du projet. Vous n'avez pas besoin de la suivre pour l'instant, puisque vous êtes déjà fourni avec les modules de base du projet.*

Dans cette section, nous allons décrire comment configurer une application node js et express à partir de zéro, en utilisant TypeScript.

Le résultat des étapes suivantes vous sera fourni, mais il peut être utile comme référence.

Paquets requis

Nous allons utiliser Yarn pour gérer les dépendances du projet, en partie parce qu'il est plus rapide que npm (surtout sur Windows).

Vous devez l'installer en utilisant `npm i -g yarn`.

Tout d'abord, évaluons ce dont nous aurons besoin :

- dépendances de développement :
  - `typescript` (compilateur TypeScript),
  - `eslint` (Linter),
  - ~~mocha ou jest (Outil de test)~~ (plus tard),
- dépendances de l'application :
  - `express`,
  - `morgan`,
  - `dotenv`

Initialisation du projet

:warning: Avant de commencer la configuration, nous allons installer yarn s'il n'est pas déjà installé : `npm i -g yarn`.

Configuration du dépôt Git et du projet NPM

Nous commençons par créer le dépôt git avec `git init`.

```
mkdir ipi-cicd-example
cd ipi-cicd-example
git init
git branch -m main
```

Nous téléchargeons ensuite le fichier `.gitignore` de GitHub pour les projets Node.js (https://github.com/github/gitignore/blob/main/Node.gitignore) (partie d'une large collection de fichiers `.gitignore` (https://github.com/github/gitignore))

```
wget -O .gitignore
https://raw.githubusercontent.com/github/gitignore/main/Node.gitignore
```

Puis nous initialisons le projet Node.js :

```
npm init -y
git add package.json .gitignore
git commit -m"Initialisation du projet Node.js avec package.json et .gitignore"
```

Installation des dépendances de développement de base

Nous procédons ensuite à l'installation des dépendances de développement essentielles. Les packages préfixés par `@types/` fournissent les types TypeScript pour les API standard de Node.js, Express.js et la bibliothèque de logging Morgan respectivement.

```
yarn add --dev typescript ts-node nodemon @types/node @types/express
@types/morgan
git add package.json yarn.lock
git commit -m"Installation des dépendances de développement de base"
```

Configuration de TypeScript

Nous configurons ensuite le fichier `tsconfig.json` de TypeScript. Nous avons deux options pour cela :

- Exécuter `npx tsc --init`, qui le crée avec les paramètres par défaut habituels.

- Hériter d'une configuration spécifique à Node.js fournie par un tiers, comme tsconfig/bases (https://github.com/tsconfig/bases/) .

Bien que la seconde semble être une bonne idée, nous avons eu des problèmes avec elle pour Node.js 20. Nous allons donc initialiser `tsconfig.json` avec `tsc --init`, avec quelques arguments :

- Les fichiers sources du projet seront situés sous `src`,
- Les fichiers JavaScript produits par `tsc` iront sous `dist`,
- Nous autorisons l'importation de fichiers `.json` depuis notre code TypeScript,
- Nous autorisons l'utilisation des fonctionnalités récentes d'EcmaScript.

```
npx tsc --init --rootDir src --outDir dist --resolveJsonModule --lib es2022
git add tsconfig.json
git commit -m"Configuration de TypeScript"
```

Dépendances de l'application

Nous installons ensuite les packages qui seront utilisés par l'application elle-même (nous n'avons pas installé `@types/dotenv` car les types sont fournis avec `dotenv` lui-même).

```
yarn add express morgan dotenv
git add package.json yarn.lock
git commit -m"Installation des dépendances de base de l'application"
```

Serveur Express de base

Nous créons une application serveur Express simple dans `src/index.ts`, avec le contenu suivant :

```
import express from "express";
import morgan from "morgan";
import dotenv from "dotenv";

// Charger les variables d'environnement depuis le fichier .env
dotenv.config();

const app = express();

// Middlewares
app.use(morgan("dev"));
app.use(express.urlencoded({ extended: true }));

// Routes
app.get("/", (req, res) => {
  res.send("Hello, world!");
});

// Démarrer le serveur
const port = process.env.PORT || 3000;
app.listen(port, () => {
  console.log(`Le serveur fonctionne sur le port ${port}`);
});
```

Nous devons également configurer nos scripts `package.json`, pour faciliter le développement de notre app. Nous avons précédemment installé `nodemon` et `ts-node`.

nodemon surveille les fichiers sources de l'application et la redémarre lorsqu'un changement se produit. ts-node nous permet d'exécuter des programmes TypeScript sans avoir à les compiler manuellement avec tsc. Nous ajoutons cette ligne dans la section "scripts" de notre package.json (au-dessus de "test") :

```
    "dev": "nodemon --exec ts-node src/index.ts",
```

Nodemon utilisera `ts-node` comme environnement d'exécution, au lieu d'utiliser le `node` "vanilla".

Grâce à cela, nous pouvons démarrer notre application en mode "watch" en lançant `yarn dev`.

Nous traçons ces changements :

```
git add src/index.ts package.json
git commit -m"Écriture d'une app Express basique, ajout d'un script de
démarrage"
```

À partir de là, nous sommes prêts à implémenter des fonctionnalités basiques !

Il reste cependant un peu de mise en place à faire :

- scripts pour compiler l'app,
- scripts pour démarrer l'app compilée,
- configuration d'ESLint,
- configuration de hooks Git,
- installation et configuration d'un framework de test,
- etc.