Agilité : Intégration, déploiement, livraison

TP1 Agilité

Intégration continue, livraison, déploiement continue

Jérôme Buisine
jerome.buisine@univ-littoral.fr

18 septembre 2023

Durée : 6 heures

L'objectif de ce TP est la mise en place d'un pipeline d'automatisation de processus pour l'intégration, le déploiement et la livraison d'un projet.

1 Travail

Ce support est le premier d'une suite de travaux pratiques qui ont pour objectif la mise en place d'un simulateur d'évolution de forêt. Rassurez-vous, l'objectif principal étant les bonnes pratiques de code et l'automatisation d'un projet. Dans ce sens, les différentes règles relatives à la conception du simulateur seront détaillées pour faciliter leurs implémentations. L'ensemble du code sera réalisé en Python. Une correction du simulateur ainsi qu'une interface web de l'application seront fournit pour le TP5.

Voici un ensemble de ressources qui peuvent vous être utiles pour ce TP :

— 1. Documentation officielle de l'outil `Git`.
— 2. Gitlab : interface web pour la gestion de projets versionnés sous `Git`.
— 3. pylint : permet l'analyse de code relative à des conventions du langage.
— 4. pyenv : permet la gestion d'environnement Python.
— 5. Sonarqube : outil permettant de visualiser la qualité et la sécurité du code d'un projet.

Remarque importante : faites attention au copier/coller à partir du PDF qui peut prendre en compte des caractères inattendus et causer des erreurs.

2 Configuration de l'environnement

Tout d'abord, depuis l'interface Gitlab il vous faudra **cloner** le projet suivant : treevolution-etudiant (**ne réalisez pas un fork de celui-ci !**). Il s'agit du projet sur lequel vous allez travailler et contenant une structure de base.

Créer un projet **privé** Gitlab avec la convention de nommage suivante : « treevolution-YOUR_NAME » avec `YOUR_NAME` composé de la première de votre prénom puis votre nom. Exemple, pour Jean Dupont, on obtient : `jdupont`, soit « treevolution-jdupont ». Me rajouter ensuite en tant que rapporteur du projet : `jbuisine`.

Récupérez l'URL `git` de projet Gitlab créé et configurez l'URL `git` au sein du projet cloné depuis votre terminal :

```
git remote set-url origin "YOUR_URL.git"
```

Important : vous devez à ce stade configurer un environnement de travail propre de Python. Pour cela, référez-vous à la documentation suivante : environnement Python. Elle vous permet la mise en place d'un environnement virtuel Python. Nous utiliserons pour ce projet une version 3.8.0 de Python.

3 Description du projet

Maintenant que votre environnement est prêt, nous allons nous intéresser au contenu initial du projet.

3.1 Build du projet

Ce projet est un package Python, où la configuration de celui-ci est défini dans le fichier disponible à la racine du projet : `pyproject.toml`. Ce fichier précise certaines informations utiles à la description et au build du package. Il traque par défaut le dossier `src` du projet comme contenant les sources du package.

Remplacez les informations de contact du créateur de package par les vôtres. Faites de même pour les fichiers `LICENCE` et `README.md`.

Nous allons maintenant vérifier que le projet est correctement configuré en l'état. Installez les dépendances à l'aide du gestionnaire de packages de Python (`pip`) et compilez le projet :

```
# mettre à jour la version de pip
pip install --upgrade pip
# installation de toutes les dépendances spécifiées
pip install -r requirements.txt
# build du package du projet
python -m build
```

Note : si une erreur survient à cette étape, notifiez-moi, sinon passez à la suite.

3.2 Utilisation du package

Réaliser un build du projet ne signifie pas avoir installé le package, mais avoir créé son archive et son fichier prêt à être installer (`wheel`). Le dossier `dist` comprend l'ensemble des versions générées. Il est ainsi possible d'installer localement notre package :

```
# installation de treevolution
pip install dist/treevolution-0.0.1-py3-none-any.whl

# autre moyen : build + installation
pip install -e .
```

Installez la dépendance `ipython` depuis `pip` et lancez la commande depuis un terminal. Il s'agit d'une console interactive Python améliorée (utilisée notamment dans Jupyter notebook).

À partir de la console interactive, utilisez la librairie `treevolution` maintenant disponible en créant un Point aléatoire. Utilisez également les classes `Season` et `Weather` du module `treevolution.context` pour comprendre leur fonctionnement.

Indication : utilisez la librairie `datetime` disponible depuis Python.

3.3 Autres composantes du projet

Au sein du projet, vous pouvez également y retrouver les dossiers suivants :

— `docs` : dossier où l'on y retrouve une documentation de la librairie.

— `tests` : dossier où l'on ira ajoutera l'ensemble des tests unitaires pour couvrir au mieux le code.

Nous reviendrons sur l'utilisation de ces dossiers lors de la mise en place des tâches qui leurs sont associées au sein du pipeline d'automatisation.

4 Vers l'automatisation des processus

Nous allons utiliser l'outil CI/CD (Continuous Integration / Continuous Delivery) de Gitlab. Cet outil se repose sur une configuration d'étapes nommées « `stages` » d'automatisation à effectuer, elles-même composées de plusieurs sous-étapes nommées « `jobs` ».

Cette configuration sera détaillée dans le fichier `.gitlab-ci.yml` dont une version minimale est actuellement disponible dans le projet. Elle se base sur un seul « `stage` » et un « `job` » associé. Le déploiement côté Gitlab est effectué au sein d'une instance Docker dont l'image est spécifié dans cette configuration.

Remarque : nous verrons qu'il est possible de spécifier une image spécifique à un « `job` » si besoin.

Cette configuration a été intégrée lors du commit initial du projet. Vérifiez depuis l'interface Gitlab (depuis le raccourci « `CI/CD` » de votre projet) si le pipeline s'est effectué correctement et familiarisez-vous avec l'interface.

4.1 Automatisation du build

Créer une branche **develop** à partir de laquelle vous allez principalement développer. Modifiez la configuration CI/CD pour y remplacer le « `job` » actuel par :

```
build:
  stage: build
  script:
    - python -m build
```

Réalisez un commit et soumettez-le sur le serveur Gitlab. Vérifiez l'état du pipeline avec cette nouvelle configuration. Une **erreur** devrait être normalement présente. Analysez les erreurs obtenues disponibles au sein de l'instance Docker depuis Gitlab. Corrigez le script CI/CD pour corriger le problème.

Une fois l'erreur corrigée, vous pouvez passer à la suite.

4.2 Vérification automatisée des tests unitaires

Comme précisé, un dossier `tests` est présent dans le projet afin de proposer une couverture du code du package `treevolution`.

Testez localement depuis votre terminal les commandes suivantes :

```
# lecture par défaut du dossier tests
pytest

# spécification du dossier à couvrir
# permet de donner un pourcentage de couverture du code
pytest --cov=src/treevolution
```

Nous allons maintenant ajouter à notre CI/CD, un nouveau « stage »nommé test. Associez à ce « stage » un « job » permettant de lancer les tests unitaires. Inspirez-vous de la syntaxe du job précédent, réalisez un commit et visualisez l'état du pipeline.

Si une **erreur** survient, analysez-la, et proposez une correction (indication, se référer à l'installation locale du package). Une fois l'erreur corrigée, vous pouvez passer à la suite.

À niveau là, les deux « jobs » doivent être fonctionnels. Toutefois, on remarque qu'une commande reste commune à ceux-ci impliquant une perte de temps d'exécution :

```
pip install -r requirements.txt
```

Il est possible de proposer des commandes communes à tous les « jobs » par une instruction nommée before_script :

```
1  stages:
2    - build
3    - test
4
5  image: python:3.8-slim-buster
6
7  before_script:
8    - echo "Cette commande sera executée pour tous les jobs si souhaité"
9
10 ...
```

Modifiez votre configuration en conséquence, réalisez un commit et vérifiez l'état de votre pipeline.

4.3 Génération et publication de la documentation

Proposer un outil de développement à des développeurs tel qu'un package nécessite une documentation. Il permet à toute personne souhaitant utiliser l'outil de se familiariser avec celui-ci et d'en comprendre son fonctionnement.

4.3.1 Description du formalisme de la documentation

Appelée docstring, ce formalisme utilisé pour générer la documentation est présent à plusieurs niveaux :

— Description du module

— Description d'une classe ou d'une fonction d'un module

— Description des méthodes d'une classe

Une génération automatisée de la documentation via la `docstring` présente est alors possible. Il sera important pour la suite de vos développements d'utiliser cette convention pour une bonne génération de la documentation. Voici un exemple de `docstring` d'une classe présente dans un module :

```
"""
Array and List module
"""

class MyList():
        """
        List class structure
        ...
        """


        ...

        def is_inside(self, value):
                """
                Find if value exists inside current list instance

                Attributes:
                        value: {float} -- value to check

                Returns:
                        {bool}: value found or not

                """
                return value in self._values
```

On remarque bien dans cet exemple que chaque argument et retour d'une fonction sont précisés à partir des mots clés `Attributes` et `Args`. Le package contient actuellement une `docstring` à jour. Vous pourrez également vous en inspirer.

4.3.2 Génération locale de la documentation

Pour générer cette documentation nous allons utiliser la librairie `Sphinx`. Elle se base sur un formalisme de fichiers d'extension `.rst` (`reStructuredText` markup language, tout comme l'est le `Markdown`) permettant de générer du contenu `html` rapidement. L'outil `sphinx` va scanner les fichiers `.rst` dont un lien est présent dans le fichier initial `index.rst`. De plus, `sphinx` va traiter la `docstring` attendue des fichiers sources `Python` ciblés par un fichier `.rst`. Il pourra ainsi générer un site web contenant la description complète du package (voir dossier `docs` et la documentation sphinx).

Le package Python `sphinx` est déjà présent dans vos dépendances de projet et normalement déjà installé. Vous pouvez donc générer localement la documentation du dossier `docs` par la commande suivante :

```
# Génération html de la documentation
# - dossier source sphinx : docs/source
```

```
# - dossier de sortie : docs/build
sphinx-build -b html docs/source/ docs/build
```

À partir de votre navigateur, vous pouvez ouvrir le fichier `index.html` généré au sein du dossier `docs/build`.

Ouvrez le fichier `docs/source/conf.py` de configuration `sphinx`. Modifiez-le pour y ajouter vos informations personnelles (copyright et auteur).

4.3.3 Intégration de la documentation dans la CI/CD

Tout comme pour les étapes précédentes, nous souhaitons que l'intégration continue traite la génération automatique de la documentation. L'objectif étant d'avoir toujours la dernière version de la documentation en ligne (oublier de générer celle-ci à chaque commit devient très probable).

Pour cela, créez un nouveau « `stage` » nommé `deploy`. Vous pouvez y associer un « `job` » nommé `pages` qui permettra de générer la documentation et proposer à Gitlab de l'exploiter. Pour que Gitlab puisse servir la documentation générée, il faudra qu'elle soit accessible. Pour cela, la plateforme propose dans la notion `artefacts`, éléments sauvegardés d'un « `job` » rendus accessibles et qui peuvent être téléchargés.

Faites alors en sorte que votre `job` génère la documentation dans un dossier `public` à la racine du projet et proposer ce dossier en tant qu'`artefact` :

```yaml
pages:
  ...
  artifacts:
    paths:
    - public
```

Il est également possible d'ajouter une règle précise d'exécution du job permettant qu'elle soit générée uniquement pour la branche par défaut (master ou main par exemple) :

```yaml

pages:
  ...
  rules:
    - if: $CI_COMMIT_REF_NAME == $CI_DEFAULT_BRANCH
```

Remarque : cette règle est importante car si l'on commence à générer la documentation depuis n'importe quelles branches, cela impliquera qu'elle comportera des informations potentiellement non complètes ou non validées.

Réalisez un commit des modifications apportées, et soumettez-les au serveur. Intégrer également les modifications dans la branche principale et soumettez-les au serveur. Vérifiez le pipeline et que l'accès au site web est maintenant disponible (`Settings > Pages`).

Une fois la documentation valide et accessible en ligne, vous pouvez passer à l'étape suivante.

4.4 Proposer une livraison

Comme vous le savez, il est possible de créer un `tag` pour une certaine version du projet. L'objectif est d'acter d'une version stable du projet. Il est possible au sein du processus d'intégration de Gitlab de proposer une `release` du projet `treevolution`. Pour cela, depuis votre branche `develop` ajoutez un « `stage` » nommé `release` et y associer le « `job` » suivant :

```yaml
release_job:
  stage: release
  image: registry.gitlab.com/gitlab-org/release-cli:latest
  rules:
    - if: $CI_COMMIT_TAG  # Run this job when a tag is created
  before_script:
    - echo "Prepare new release"
  script:
    - echo "Running release_job"
  release: # See https://docs.gitlab.com/ee/ci/yaml/#release
    tag_name: '$CI_COMMIT_TAG'
    description: '$CI_COMMIT_TAG'
```

Remarque : pour ce « `job` », on utilise une image Docker spécifique (proposée par Gitlab), une règle particulière pour n'exécuter ce « `job` » que lors d'ajout de tag et une instruction `before_script` localement pour ne pas exécuter l'installation des packages Python (inutile et impossible étant donnée l'image Docker).

Créez un commit avec la configuration de la CI et soumettez les modifications au serveur. N'oubliez pas d'ajouter également ces modifications dans la branche principale.

Créez ensuite un nouveau tag v0.0.1 avec le message suivant « Treevolution initial version ». Proposez la nouvelle version au serveur Gitlab avec la commande : `git push -tags`. Vérifiez que la `release` a bien été générée.

Note : il est possible d'ajouter un fichier de description complet de la version. Cela peut-être intéressant pour préciser davantage son contenu.

5 Outil de mesure de qualité

Pour évaluer la qualité du code du projet, nous allons utiliser un outil externe : Sonarqube. Un serveur est mis à disponibilité dans le cadre de ce module et accessible depuis l'adresse suivante : https://diran.univ-littoral.fr.

Vous pouvez vous connecter avec la même convention de pseudo que proposé en début de TP pour le nom de votre projet (`jdupont` pour Jean Dupont). Le mot de passe par défaut de votre compte est `agility2022` (modifiez-le si besoin). Si un problème de connexion survient, merci de m'en faire part.

5.1 Configuration de l'outil

Depuis votre compte Sonarqube, vous allez créer un nouveau projet. Pour cela, depuis la page d'accueil de votre compte, cliquez sur « Ajouter un projet », puis sur « Manuellement ». Vous allez pouvoir indiquer le nom de votre projet à l'identique de celui de Gitlab : « treevolution-XXXXX ». Vous pouvez conserver le même clé de projet et indiquer la branche principale comme étant **master**.

Il est maintenant nécessaire de générer le token Sonarqube, choisissez l'option qui implique aucune expiration de celui-ci. **Conservez ensuite le token Sonarqube fourni.**

Sonarqube propose une commande permettant de lancer une analyse du code localement. Nous allons plutôt procéder par l'utilisation d'un fichier de configuration spécifique à Sonarqube contenant l'ensemble des paramètres de cette commande. Pour cela, créez le fichier `sonar-project.properties` à la racine du projet avec le contenu suivant :

```
sonar.projectKey=treevolution-XXXXXX
sonar.qualitygate.wait=true
sonar.sources=src/treevolution
sonar.python.version=3.8
```

La valeur de la clé `projectKey`, correspond à celle qui était fournie dans la commande Sonarqube. Vous pouvez également remarquer que des paramètres liés aux sources du projet ont été ajoutées : le dossier à cibler et la version Python ciblée. Nous reviendrons sur le paramètre `qualitygate`.

5.2 Intégration de Sonarqube au sein de la CI/CD

Pour l'intégration de Sonarqube au sein du pipeline, nous allons suivre la documentation officielle CI/CD proposée par Sonarqube (SonarQube CLI)

Il est possible d'ajouter des variables d'environnement à la pipeline CI/CD de votre projet Gitlab : « Settings > CI/CD > Variables ».

Suivez ensuite les instructions de la section "Setting environment variables" en précisant que les variables d'environnements Gitlab sont masquées. C'est ici que votre token Sonarqube sera utilisé par la variable d'environnement `SONAR_TOKEN` de Gitlab. La valeur de la variable `SONAR_HOST_URL` quant à elle, correspond simplement à l'adresse d'accès du serveur Sonarqube que nous utilisons, soit l'adresse suivante : https://diran.univ-littoral.fr.

Important : il vous faudra utiliser une image particulière pour prendre en compte le langage Python pour ce job `sonar`, soit l'image suivante « `rawshark/python-sonar-scanner:3.8` ».

Remarque : l'utilisation de variables d'environnements Gitlab permet d'éviter l'ajout de contenu sensible au sein du projet.

Proposez maintenant un nouveau « `stage` » nommé `sonarqube` auquel on associe le « `job` » proposé dans l'onglet « `SonarScanner CLI` » de la section instructions CI/CD de cette documentation.

Au sein de la configuration CI, indiquez l'ordre suivant des « *stages* » :

```
stages:
  - build
  - test
  - deploy
  - sonarqube
  - release
```

Depuis la branche `develop`, réalisez un commit et soumettez les modifications au serveur. Vérifiez que le pipeline s'exécute intégralement, bien que le « `job` » de Sonarqube ait peut-être échoué. Le paramètre `allow_failure` permet en effet de continuer l'exécution de « `jobs` » suivants un « `job` » ayant échoué.

La « *Quality Gate* » détermine si le code est en l'état acceptable ou non. Elle suit des règles bien précises spécifiées dans le cadre d'un projet. En l'état nous utilisons une « *Quality Gate* » par défaut de Sonar qu'il est possible de visualiser depuis l'onglet « *Quality Gates* » de Sonarqube. On y retrouve des métriques importantes à respecter comme :

— La couverture du code, fixée à 80% minimum.

— Pourcentage de lignes dupliquées, fixé ici à 3% maximum.

— Les points sensibles de sécurité, qui ici doivent tous être traités.

Dans l'aperçu de votre projet, vérifiez l'état actuel de votre « *Quality Gate* » et comprendre les éléments bloquants la validation de la qualité du code du projet (si existants).

Dans tous les cas, vous pouvez traiter les « *Security Hotspots* » liés au package `random` de Python et les assigner comme étant « *Safe* ». En effet, nous verrons comment gérer correctement l'effet aléatoire lors de la couverture du code par nos tests.

Si votre « *Quality Gate* » est valide vous pouvez passer à la suite.

5.3 Mesures de qualité du projet

En l'état actuel Sonarqube n'offre aucune métrique de couverture et de qualité de code.

Pour mesurer la qualité d'un code, nous allons utiliser `pylint`. Il permet de mettre en avant des erreurs potentielles et les bonnes pratiques de code après avoir scanner les sources du projet.

Utilisez la commande suivante pour mesurer la qualité de votre code :

```
pylint src
```

On peut remarquer rapidement que des erreurs sont remontées. Cela est du fait que les programmes principaux `src/main.py` et `src/main_visu.py` exploitent une version finale du package et donc des méthodes de classes encore non présentes.

En l'état, nous allons désactiver le message `C0303` qui affecte énormément les commentaires de la `docstring` et non souhaités. Pour cela, lancer la commande pylint avec le paramètre suivant :

```
pylint --disable=C0303 src
```

Remarque : on y notifie rapidement une amélioration de la qualité du code en terme de score.

5.4 Intégration des métriques au sein de Sonarqube

Il est possible de générer des rapports dans un format spécifique à la fois des tests de couverture et des mesures de qualité de code. Modifiez le « `job` » associé à Sonarqube pour y ajouter la génération des rapports avant exécution de la commande `sonar-scanner` :

```
...
script:
  - pip install -e .
  - pytest -v -o junit_family=xunit1 --cov=src
    --cov-report xml:tests/coverage.xml
    --junitxml=tests/nosetests.xml
```

```
  - pylint --exit-zero --disable=C0303,R0902
    --output pylint-report.txt src
  - sonar-scanner
...
```

Pour que Sonarqube puisse prendre en considération ces rapports, il est nécessaire de lui spécifier leurs chemins au sein du fichier configuration `sonar-project.properties` :

```
sonar.python.xunit.reportPath=tests/nosetests.xml
sonar.python.coverage.reportPaths=tests/coverage.xml
sonar.python.pylint.reportPath=pylint-report.txt
```

Réalisez un commit et soumettez-le au serveur depuis la branche develop. Vérifiez maintenant que Sonarqube prend en compte la couverture du code et l'analyse de code `pylint`.

6 Ajout d'indicateurs au projet

Posséder des indicateurs visibles et régulièrement mis à jour est un avantage pour un projet. Il permet de mettre en avant les problèmes potentiels et dans le meilleur des cas, sa stabilité (inspirant généralement la confiance d'un point de vue extérieur).

6.1 Rapport tests unitaires Gitlab

En vous inspirant de la documentation Gitlab, ajoutez au sein de votre « `job` » `test`, la possibilité à Gitlab de récupérer les rapports de tests.

6.2 Les badges Gitlab

Les badges sont des indicateurs intéressants associés à la description du projet. En vous inspirant de la documentation Gitlab à cet effet, ajoutez les badges suivants :

— Succès de l'exécution du pipeline.

— Pourcentage de couverture des tests unitaires.

— Version de la livraison du projet.

6.3 Bonus : badges spécifiques

Ajoutez le badge Sonarqube disponible depuis l'onglet « *Project Information* » au sein de votre fichier `README.md`

Puis, proposez également un indicateur de qualité de code `pylint` généré depuis la CI/CD de Gitlab.

7 Modalités de rendu du TP

Réalisez une version de votre projet en l'état (v0.0.2) qui sera traitée comme rendu de votre TP.