Agilité : test d'interfaces web et API

TP5 Agilité

Test d'interfaces web et API

Éric Ramat & Jérôme Buisine
eric.ramat@univ-littoral.fr
jerome.buisine@univ-littoral.fr

14 novembre 2022

Durée : 6 heures

L'objectif de ce TP est l'ajout d'une nouvelle étape de vérification au sein du pipeline. Nous allons pour cela utiliser « Cypress », un outil qui permet de tester une application web et des appels API.

1 Travail

Ce support propose d'intégrer un ensemble de tests d'interface afin de valider les besoins attendus du client sur l'application finale et la robustesse de celle-ci. De ce fait, vous allez dans ce TP développer ces différents tests en respectant des consignes spécifiques :

— 1. À partir d'une branche **`feature/InterfaceTestX`** spécifique à une histoire, écrire le test qui permet sa vérification.

— 2. Chaque test doit être associé à un commentaire ou être nommé en conséquence de l'histoire qu'il cherche à vérifier.

— 3. Pour chaque test développé, vérifiez que celui-ci fonctionne localement.

— 4. Soumettre en ligne la modification apportée et vérifier que le pipeline d'intégration continue fonctionne correctement.

Pour vous aider : 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. nodejs : environnement d'exécution JavaScript qui exécute le code JavaScript en dehors d'un navigateur Web.

— 4. npm : le gestionnaire de paquets associé à `node.js`.

— 5. Cypress : permet d'écrire des tests d'interfaces web en utilisant un navigateur embarqué.

— 6. Local Storage : niveau de persistance de données sous forme de dictionnaire côté client (navigateur web).

— 7. Cypress Local Storage plugin : documentation d'un plugin Cypress qui prend en compte la gestion du « *local storage* ».

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

Nous allons dans un premier temps vérifier le bon fonctionnement de l'application puis configurer l'environment permettant d'exécuter `Cypress`.

Important : Il faudra rajouter en tant que rapporteurs du projet : `eramat`.

2.1 Application Flask

L'application web proposée est développée avec le framework `Flask`. Il permet de développer rapidement un serveur composé de routes et de servir des contenus web / données.

Pour que l'application web puisse fonctionner correctement, il vous faut ajouter des dépendances Python au projet :

```
flask==2.2.2
gunicorn==20.1.0
numpy==1.23.2
```

Il faudra bien installer la dernière version du package `treevolution` pour que l'application web fonctionne correctement :

```
pip install -e .
```

Une fois les dépendances installées, il vous est possible de lancer l'application web :

```
gunicorn --bind 0.0.0.0:5000 wsgi:app
```

Important : Flask utilise une clé secrète qui sera utilisée pour signer de manière sécurisée le cookie de session et qui plus globalement est utilisée pour tout autre besoin lié à la sécurité. Il est possible d'en générer une spécifique en suivant cette documentation. Dans le fichier « `web/app.py` », remplacez la variable « `app.secret_key` » pour votre clé secrète nouvellement générée.

2.2 Installation des dépendances

Tout d'abord il vous sera nécessaire de posséder `nodejs`. Si besoin, vous pouvez télécharger la version LTS (*Long Term Support*) depuis la page de téléchargement. Vérifiez que votre version de `nodejs` est bien $>= 16.17.0$ avec la commande suivante :

```
node --version
```

Une fois `nodejs` installé, utilisez le package manager `npm` pour installer la dépendance `Cypress` :

```
npm install cypress cypress-localstorage-commands --save-dev
```

Remarque : un plugin est à installer également. Il permettra la prise en compte du local storage qui n'est pas géré par défaut par Cypress.

Puis, ajoutez dans votre fichier `.gitignore` les dossiers suivants :

```
# web application
web/static/tmp

# Dependency directories
node_modules/

# cypress directories
cypress/downloads
cypress/screenshots
cypress/videos
```

Note : certaines dépendances de votre système peuvent être nécessaires pour le bon fonctionnement de Cypress.

2.3 Configuration de Cypress

Lancez l'application Cypress graphiquement à l'aide de la commande suivante :

```
./node_modules/.bin/cypress open
```

Sélectionnez ensuite la mise en place d'une configuration E2E. Cela va vous créer un fichier de configuration cypress.config.js et un dossier cypress, tous deux à la racine du projet.

Précisez la prise en compte du plugin dans la configuration des tests *end to end* dans le fichier « cypress/support/e2e.js » en y ajoutant l'import :

```
import 'cypress-localstorage-commands'
```

Dans le cadre du projet, nous allons tester les modifications apportées soit localement, soit depuis un environnement distant. Pour cela, nous allons configurer le fichier « cypress.config.js » comme suit :

```
const { defineConfig } = require("cypress");

module.exports = defineConfig({
  e2e: {
    setupNodeEvents(on, config) {
      // load the localstorage plugin
      require("cypress-localstorage-commands/plugin")(on, config)

      // manage environment server
      const envServer = config.env.server || 'development'

      if (envServer === 'local')
        config.baseUrl = `http://127.0.0.1:5000`
      else
        config.baseUrl = `https://treevolution-XXXXXX-${envServer}.fly.dev`
```

```
        return config
      },
    },
});
```

Ainsi, il sera possible de spécifier l'environment à tester à l'exécution de la commande `cypress` :

- `./node_modules/.bin/cypress run -env server=local` : environnement local.
- `./node_modules/.bin/cypress run -env server=development` : environnement de développement `fly.io` (par défaut).
- `./node_modules/.bin/cypress run -env server=production` : environnement de production `fly.io`.

Remarque : la commande `run` de `Cypress` permet d'exécuter l'ensemble des tests d'interface sans retour graphique. Ce qui réduit la lenteur de vérification.

Important : si vous utilisez `Heroku`, il faudra modifier la variable `baseUrl` en conséquence.

2.4 Configuration CI/CD pour Cypress

Dans le dossier `cypress/E2E`, créer le fichier « `treevolution-config.cy.js` » qui sera le premier fichier de test. Ajoutez-y votre premier test d'interface :

```
describe('Visit Treevolution', () => {

  it('treevolution website', () => {
    cy.visit('/')
  })
})
```

Vérifiez que celui-ci fonctionne bien localement ou lorsque l'on spécifie un environment distant (l'application web devra être lancée pour que `Cypress` puisse y accéder).

Ajoutez maintenant un « `stage` » nommé `interface` en fin de pipeline CI/CD :

- On y retrouvera deux jobs, un qui teste l'interface de l'environment de développement et un l'environnement de production. Ces tests sont uniquement exécutés lorsque l'environment ciblé est déployé.
- Chaque job se base sur une image spécifique proposée par `Cypress` : `cypress/base:16.17.0`
- Le job en question requiert la commande « `npm ci` » pour installer les dépendances et fonctionner correctement.
- Au sein du job, cypress est exécuté par la commande : « `npx cypress run` » (voir la documentation).

Réalisez un premier commit depuis la branche `develop` et vérifiez que le nouveau job s'exécute correctement. Une fois la configuration terminée, vous pouvez continuer le TP.

3 Développements

Il vous faudra bien identifier ce qu'il vous sera utile dans la documentation de `Cypress` pour chaque vérification d'histoire. L'ensemble des tests d'interface et d'API sont à coder dans le dossier `cypress/E2E`. Pour chaque grande fonctionnalité de l'application, un fichier de test créé.

Le « *local storage* » est utilisé au sein de l'application pour son bon fonctionnement. Pour le prendre compte correctement pour chaque histoire, il faudra que chacun des fichiers de tests soient composés de :

```
beforeEach(() => {
  cy.restoreLocalStorage() // entre chaque `it` restaure le local storage
  cy.visit('/') // requiert également un rafraichissement de page
  cy.wait(500)
})

afterEach(() => {
  cy.saveLocalStorage() // sauvegarde le local storage après chaque `it`
})

describe('StoryX', () => {

  it('Check local storage expected access', () => {

    cy.getLocalStorage("expected-key").then(element => {
      expect(element).to.not.equal(null);
    });
  })
})
```

Important : l'instruction `cy.wait` permet de réaliser un temps d'attente avant de poursuivre le test. Ce qui peut être nécessaire lors de l'appel à une validation côté serveur qui est elle asynchrone.

3.1 Vérification de la configuration

L'ensemble des tests d'interface liés à la vérification du bon fonctionnement de la configuration devront être écrit dans le fichier « `treevolution-config.cy.js` » préalablement créé.

Histoire 1 : La page principale de l'application doit avoir un titre composé du texte « *Welcome to the Treevolution simulator* ».

Histoire 2 : En accédant à l'onglet `/config` de l'application. Vérifier que les champs nombres de jours et nombres d'arbres indiquent correctement leurs valeurs associées si modifiées.

Indication : regarder la documentation des méthodes `invoke` et `trigger` de `Cypress`.

Histoire 3 : En accédant à l'onglet `/config` de l'application. Vérifiez qu'il n'est pas possible de valider une configuration ayant une date de fin de simulation inférieure à celle de début. Un message d'erreur peut-être vérifié.

Histoire 4 : En accédant à l'onglet `/config` de l'application. Vérifiez qu'il n'est pas possible de valider une configuration n'ayant aucun type d'arbre sélectionné. Un message d'erreur peut-être vérifié.

Histoire 5 : En accédant à l'onglet `/config` de l'application. Il est possible de valider une configuration. Vérifiez que :

- Un message de validation est proposé.
- L'ensemble des champs saisis sont bien retrouvés après validation.
- La page de simulation propose les mêmes paramètres de configuration.

Histoire 6 : Lorsqu'un utilisateur se connecte, un identifiant unique lui est associé depuis le « *local storage* ». Cet identifiant permet de sauvegarder la configuration de l'utilisateur. Vérifiez que la configuration

est correctement réinitialisée en supprimant la clé « `treevolution_uuid` » présente dans le « *local storage* ».

Indication : regarder la documentation du plugin associé à `Cypress` permettant la prise en compte du « *local storage* ».

3.2 Vérification de la simulation

L'ensemble des tests d'interface liés à la vérification du bon fonctionnement du simulateur devront être écrit dans un fichier « `treevolution-simulation.cy.js` »

Histoire 7 : À partir d'une configuration valide, vérifiez qu'il est possible de créer une nouvelle simulation depuis l'onglet `/simulation`. Vérifiez que :

- La date de début correspond bien à celle attendue.
- Le nombre d'arbres vivants également.
- Le nombre d'arbres en humus doit être de 0.
- Le nombre de seeds doit être de 0.

Histoire 8 : À partir d'une configuration valide et d'une nouvelle simulation créée. Vérifiez que le nombre d'espèces d'arbres sélectionné est correct.

Histoire 9 : Lorsqu'une simulation est créée depuis l'onglet `/simulation` à partir d'une configuration. Celle-ci peut-être supprimée si la configuration vient à changer.

Histoire 10 : Lorsqu'une simulation est exécutée, vérifiez que la date et le nombre d'arbres ont évolué après un certain délai.

Histoire 11 : Lorsqu'une simulation est exécutée, puis mise en pause, vérifiez que la date et le nombre d'arbres sont restés fixes après un certain délai.

Histoire 12 : Lorsqu'une simulation est exécutée, puis mise en pause, vérifiez qu'il est possible de la réinitialiser. La barre de progression doit alors être de 0%, tout comme les informations de simulation réinitialisées.

Histoire 13 : Vérifiez qu'il est possible d'atteindre la fin d'une simulation. La date et le nombre d'arbres sont restés fixes après la fin de cette simulation. Il n'est plus possible de continuer la simulation, mais on peut en créer une nouvelle à partir de la même configuration.

Histoire 14 : Vérifiez que la simulation n'est plus fonctionnelle lorsque la clé « `treevolution_uuid` » est supprimée depuis le « *local storage* ».

3.3 Vérification avancée de la simulation

Généralement, lorsque l'on vérifie une interface web, on souhaite également vérifier que les données obtenues depuis un serveur correspondent. Nous allons dans notre cas vérifier les données obtenues depuis l'API.

Indication : il est possible d'effectuer une requête depuis le serveur et de traiter la réponse.

```
cy.request({
  url: '/state',
  failOnStatusCode: false
}).as('state')

cy.get('@state').should((response) => {
  expect(response.status).to.equal(200)
})
```

Note : le paramètre `failOnStatusCode`, permet d'exécuter la requête et de la traiter qu'importe le code d'erreur.

L'ensemble des tests d'interface couplés à la vérification de l'API devront être écrit dans le fichier « `treevolution-api.cy.js` ».

Histoire 15 : Lorsqu'il n'existe pas de configuration de simulation, la route `/state` de l'API ne permet pas de récupérer un état de simulation.

Histoire 16 : Lorsqu'une configuration est établie et une simulation ajoutée (mais non lancée), vérifiez que les types d'arbres présents dans l'état de simulation correspondent bien à ceux de la configuration.

Histoire 17 : Lorsqu'une simulation est lancée puis mise en pause. L'affichage proposé par l'application doit concorder avec le retour de l'API (`/state`). Vérifiez notamment la date et le contexte météo.

Histoire 18 : Lorsqu'une simulation est terminée, vérifiez que cette indication est bien présente dans les informations transmises par l'API (`/state`).

4 Remise des travaux

N'oubliez pas de réaliser un dernier commit de vos travaux. Taguez ensuite votre projet avec le tag « `tp5` » et soumettez-le sur le serveur Gitlab. Vérifiez que le pipeline reste fonctionnel. Cette version de projet fera office de rendu.