**CHAPITRE 1 : ETAT DE ART.**

1.1 Les Tests Automatisés dans le Développement Logiciel

## Introduction

Fondée en 2017***,*** ADWA SARL est un organisme de prestataire de services dans le domaine du digital, fournissant ainsi des solutions numériques, des services d’audit et conseil des Systèmes d’information (SI), de pilotage de projet, d’intégration de progiciels ...

Dans le cadre de mon stage académique au sein de l’entreprise, j’ai eu à travailler dans le département de QAHD assignée à la qualité logicielle (QA). Les tâches d'un Testeur QA consistent principalement à : tester les fonctionnalités, trouver des bugs, vérifier les performances sans toutefois omettre de documenter les bugs.

Durant ma période de stage je remarque que les processus de tests sont généralement performés manuellement en suivant les instructions du cahier de charges des solutions. Ces derniers peuvent varier en fonctions des besoins des clients et justifier l’existence de plusieurs types de tests que nous découvrirons dans le paragraphe suivant.

* Présentation des types de tests (tests unitaires, tests d’intégration, tests end-to-end)

Il existe plusieurs types de tests logiciels, chacun ayant des objectifs spécifiques pour vérifier les différents aspects de la qualité d'un produit. Voici un aperçu des principaux types de tests logiciels :

# Tests Unitaires

* Objectif

Tester les plus petites unités de code (souvent des fonctions ou méthodes) pour s'assurer qu'elles fonctionnent individuellement comme prévu.

* Importance

Ils sont majoritairement utiles dans la détection de bugs, la correction et l’amélioration de code mais également peuvent servir pour une documentation générative de ce dernier.

# Tests d'Intégration

* Objectif

Tester l'interaction entre plusieurs composants ou modules pour vérifier qu'ils fonctionnent bien ensemble.

* Importance

Les modules peuvent fonctionner correctement seuls, mais des erreurs peuvent survenir lorsqu'ils interagissent.

# Tests de Système

* Objectif

Tester le système complet dans son environnement global pour s'assurer qu'il répond aux exigences initiales.

* Importance

Ce type de test vérifie l'application du point de vue de l'utilisateur final et est un étape cruciale avant la mise en production.

# Tests de Régression

* Objectif

S'assurer que les nouvelles modifications ou corrections de bugs n'ont pas introduit de nouveaux problèmes dans les parties déjà testées de l'application.

* Importance

Cette étape est cruciale pour une bonne continuité après chaque mise à jour ou ajout de nouvelles fonctionnalités selon les retours de tests.

# Tests de Validation

* Objectif

Vérifier si le produit répond bien aux besoins du client et est conforme aux exigences spécifiées.

* Importance

Sert de passeport pour la validation effective du produit ou de la solution vis-à-vis des attentes du client.

# Tests de Performance

* Objectif

Évaluer la rapidité, la réactivité, la stabilité et l'efficacité de l'application sous différentes charges de travail.

* Importance

Tests de charge (pour voir comment l'application se comporte sous une charge normale) et tests de stress (pour voir comment elle réagit en cas de charge extrême).

# Tests de Sécurité

* Objectif

Identifier les vulnérabilités et failles de sécurité pour protéger l'application contre les attaques.

* Importance

Vérification et assurance de la gestion des données sensibles, protection contre les injections SQL, XSS, etc.

# Tests d'Acceptation Utilisateur (User Acceptance Testing ou UAT)

* Objectif

Valider que le produit final est conforme aux attentes de l'utilisateur ou du client.

* Importance

Vérification pré-lancement pour s'assurer que le produit est réellement utilisable par l'utilisateur final.

# Tests Fonctionnels

* Objectif

Vérifier que chaque fonctionnalité de l'application fonctionne conformément aux spécifications.

* Importance

Concerne surtout le "quoi" que le logiciel est censé faire, indépendamment de "comment" il le fait.

# Tests de Compatibilité

* Objectif

Tester l'application sur différents navigateurs, appareils, systèmes d'exploitation, et configurations réseau pour vérifier la compatibilité.

* Importance

Essentiel pour les applications web et mobiles, qui doivent fonctionner sur divers environnements.

# Tests Exploratoires

* Objectif

Permettre aux testeurs de découvrir des bugs en explorant l'application sans plan de test strict, souvent de manière intuitive.

* Importance

Utile pour détecter des bugs inattendus que des tests automatisés pourraient manquer.

# Tests d'Interface Utilisateur (UI Testing)

* Objectif

S'assurer que l'interface utilisateur est fonctionnelle et agréable, et que tous les éléments visuels et interactifs fonctionnent comme prévu.

* Importance

Vérifie les boutons, les formulaires, les menus, l'expérience utilisateur générale.

# Tests de Localisation et d'Internationalisation

* Objectif

S'assurer que l'application est adaptée à différentes langues, régions et cultures.

* Importance

Essentiels pour les produits destinés à un public mondial.

# Tests de Maintenance

* Objectif

Valider que le produit reste fonctionnel et stable après des modifications de code (correctifs, mises à jour de sécurité, ou ajouts de nouvelles fonctionnalités).

* Importance

Assurent l’amélioration continue du produit même après son lancement et fait sensiblement référence aux tests de régression.

Chaque type de test joue un rôle dans le processus d'assurance qualité pour garantir que l'application est fiable, performante, et prête pour les utilisateurs finaux.

* Importance des tests automatisés dans l’assurance qualité des logiciels

On se doute bien qu’avec un plan de test général en partant des tests unitaires jusqu’aux tests de maintenance, le travail des testeurs sera qualifié d’ardu. Alors nous partons sur un principe qui visera déjà dans un premier temps à regrouper un maximum de types de tests selon leur compatibilité, à les automatiser dans un deuxième temps nous permettant ainsi d’optimiser les coûts en matière de temps, d’énergie, d’ergonomie de travail mais également de ressources.

Ces tests automatisés ont pour mission particulière d’accélérer, d’alléger et de promouvoir les processus de conception-production au sein du système collaboratif d’ADWA et pour cela nous nous saisirons de l’outil Cypress.

* 1. Présentation de Cypress
* Aperçu des fonctionnalités de Cypress : écriture de tests, exécution rapide, interface utilisateur

**Cypress c'est quoi ?**

L’outil Cypress est un framework moderne et open-source mis à disposition de ses utilisateurs dans le but d’automatiser des tests, principalement conçu pour tester les applications web selon son service premier de tests de bout-en-bout (E2E).

Voici une liste informative des produits proposés par Cypress :

- **Cypress App**, une application gratuite et open source, Application installée localement pour l’écriture et l’exécution de tests.

- **Cypress Cloud**, un service payant pour l’enregistrement des tests, la mise en évidence des résultats des tests et la fourniture d’analyses de tests.

- **UI Coverage**, un add-on payant offrant une vue d’ensemble visuelle de la couverture des tests sur chaque page et composant de votre application, offrant des informations claires sur les zones non couvertes que tout le monde peut comprendre. (Perhaps it's why i only see first page then nun testing)

- **Cypress Accessibility**, un add-on payant fournissant des vérifications d’accessibilité, qui aide à détecter les obstacles pour les personnes handicapées utilisant votre application. (ayo, not sure opening this point)

Cypress est une solution robuste qui peut améliorer la qualité de votre application et elle est assez légère à appréhender. On s'intéressera donc ici dans un premier temps au produit **Cypress App** que l'on étudiera

**Comment Cypress permet-il d'écrire les tests ?**

Cypress utilise JavaScript pour écrire des tests, ce qui le rend très accessible pour les développeurs frontend habitués à travailler avec ce langage. Cypress offre une syntaxe intuitive et des fonctions puissantes pour interagir avec les éléments d'une page web. Pour écrire des tests, vous vous habituerez à des commandes telles que :

* **describe ( )** pour structurer les tests, en les regroupant sous un même ensemble logique pour une meilleure organisation et lisibilité.
* **it ( )** pour définir un test individuel, représentant une validation ou une assertion unique de comportement dans une application.
* **cy.visit ( )** pour accéder à une page spécifique.
* **cy.get ( )** pour sélectionner des éléments (comme les boutons, champs de formulaire, etc.).
* **cy.click ( )**, **cy.type ( )**, etc., pour interagir avec les éléments.

Ces commandes permettent de définir les actions que l’utilisateur effectuerait sur la page, comme remplir un formulaire, cliquer sur des boutons, ou vérifier l'affichage d'éléments spécifiques. On peut aussi ajouter des assertions (avec **cy.should ( )** par exemple) pour valider que les éléments de la page répondent aux critères attendus.

**Sur quelle base écrivons-nous des tests pour une application web donnée ?**

Les tests sont généralement basés sur les cas d'utilisation et les spécifications de l’application. On peut diviser les tests en plusieurs catégories :

**Pourquoi Cypress est-il rapide à l'exécution ?**

La rapidité et la fluidité de l’exécution des tests automatisés de Cypress est due à plusieurs facteurs :

* **Exécution directement dans le navigateur** : Cypress exécute les tests dans le même environnement de navigateur que l’application, ce qui réduit les temps de communication et rend les tests plus rapides et plus proches de la réalité d’utilisation de l’utilisateur.
* **Architecture axée sur le DOM et l’API** : Cypress interagit directement avec le Document Object Model de la page et n’a pas besoin de driver externe comme Selenium, ce qui évite la latence de communication entre le client et le serveur.
* **Tests asynchrones gérés automatiquement** : Cypress gère automatiquement l'attente pour des éléments ou des actions, éliminant le besoin de timers ou de délais manuels dans les tests, ce qui permet un flux d'exécution optimisé.

**Commentaires sur l'interface utilisateur de Cypress**

L'interface utilisateur de Cypress est simple et interactive, ce qui la rend très agréable à utiliser pour les développeurs. Elle offre un "Test Runner" qui affiche l’application web en direct à côté des logs de tests, montrant chaque étape en temps réel. On peut voir l’état actuel de chaque commande exécutée et même "remonter dans le temps" pour inspecter l'état de la page à différents points du test.

Cypress fournit également des outils de debug visuel et des captures d’écran automatiques en cas d’échec, ce qui rend le débogage plus rapide et efficace. Grâce à cette interface utilisateur, les développeurs peuvent rapidement identifier les erreurs et comprendre le comportement de l’application pendant les tests.

* Tableau comparatif de Cypress avec d’autres outils de test (Selenium, Puppeteer, etc.)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Cypress | Selenium | Jest | Postman | Puppeteer |
| Langages supportés | JavaScript | Multi-langage (Java, Python, C#, etc.) | JavaScript | Pas de code, collections d'API (et JS avec Postman Scripts) | JavaScript |
| Installation | Simple (npm install) | Complexe (drivers pour chaque navigateur) | Simple (npm install) | Simple (app standalone) | Simple (npm install) |
| Configuration | Facile à configurer pour les apps web modernes | Configurations complexes pour divers navigateurs | Simple, souvent sans config initiale | Configuration intuitive pour les appels API | Configuration simple |
| Performance | Rapide, conçu pour le frontend | Plus lent en fonction des drivers | Rapide (tests unitaires) | Rapide pour les tests d'API | Rapide, mais limité à Chrome/Chromium |
| Communauté et support | Forte et en croissance | Très grande communauté et nombreux plugins | Très grande dans l'écosystème JS | Forte communauté autour des API | Forte communauté, souvent utilisée avec DevTools |
| Cas d'utilisation idéal | Tests E2E pour des applications frontend modernes | Automatisation sur plusieurs navigateurs et environnements | Tests unitaires en JavaScript (Node.js) | Tests d'API (REST et SOAP) | Automatisation pour Chrome avec DevTools |

* 1. Avantages et Limites de Cypress
* Les points forts de Cypress pour les tests end-to-end

Les points sensibles qui font notamment la force de l’outil Cypress sont les suivants :

* ****Installation et configuration simples** :** Cypress est très facile à installer et à configurer. Contrairement à d'autres frameworks de tests qui nécessitent des configurations spécifiques du navigateur ou des installations complexes, Cypress peut être installé en une seule commande et fonctionne directement sans configurations supplémentaires.
* **Rapidité et la précision dans les tests :** Cypress exécute les tests très rapidement, car il est conçu pour fonctionner dans le même contexte que l'application. Il interagit directement avec le DOM et utilise un modèle d'exécution asynchrone optimisé, ce qui rend les tests plus rapides et moins susceptibles d’échouer en raison de la latence.
* **Rechargement automatique :** Cypress propose le rechargement automatique de l’application dès qu’il y a des changements dans le code, ce qui accélère le cycle de développement. Les développeurs peuvent voir les effets de leurs modifications de code presque instantanément dans le navigateur.
* **Exécution des tests en temps réel :** Cypress exécute les tests en temps réel et affiche les résultats dans le navigateur, ce qui permet aux développeurs de voir chaque action et chaque assertion directement. Cela améliore considérablement la compréhension de l'exécution des tests et permet de corriger plus facilement les erreurs.
* ****API intuitive** :** Cypress propose une API simple et lisible qui facilite l'écriture de tests. Elle est bien documentée et les commandes sont conçues pour être enchaînées, ce qui rend les tests plus lisibles et maintenables.
* ****Gestion de l’asynchronisme** :** Cypress gère automatiquement les opérations asynchrones en attendant la fin des actions avant de passer à la suite. Cela signifie que les développeurs n'ont pas besoin d'écrire des waits explicites, comme c'est souvent le cas dans d'autres frameworks de tests.
* ****Prise en charge des tests de bout en bout et d'intégration** :** En plus des tests E2E, Cypress prend en charge les tests d'intégration en permettant des tests spécifiques de composants. Cela peut être particulièrement utile pour les applications React ou Vue.js, où chaque composant peut être testé de manière isolée.
* ****Communauté et écosystème** :** Cypress a une grande communauté active et une documentation très complète. Il existe également de nombreux plugins et intégrations avec des services externes (comme GitHub, Jenkins, etc.), ce qui facilite son intégration dans les pipelines CI/CD.
* Limites de Cypress, notamment pour certains types de tests ou intégrations spécifiques

Cypress, bien qu'excellent pour les tests end-to-end, a certaines limitations. Voici quelques-uns des défis auxquels les développeurs peuvent être confrontés lorsqu’ils utilisent Cypress, en particulier pour des types de tests ou des intégrations spécifiques :

* ****Incompatibilité avec certains autres navigateurs**** **:** Bien que Cypress fonctionne parfaitement avec Chrome, Edge, Firefox et Electron, sa compatibilité avec d'autres navigateurs, comme Safari par exemple, est limitée.
* ****Difficulté avec les tests multi-onglets et multifenêtres** :** Cypress ne prend pas en charge les tests dans plusieurs onglets ou fenêtres simultanés. C'est un inconvénient pour les applications nécessitant des interactions dans plusieurs fenêtres, comme les applications de messagerie, ou les workflows d'authentification avec des redirections OAuth qui ouvrent des fenêtres pop-up.
* ****Limitations pour les tests de mobile natif** :** Bien que Cypress puisse simuler différentes tailles d'écran pour des tests responsive, il ne prend pas en charge les tests d'applications mobiles natives. Il est limité aux applications web, ce qui exclut les applications Android et iOS.
* ****Restrictions sur l’accès direct au backend et aux bases de données** :** Cypress fonctionne côté client et ne peut pas directement interagir avec le backend ou la base de données de manière native. Pour les tests qui nécessitent une configuration préalable ou la vérification des données en backend, il faut utiliser des solutions alternatives, comme des scripts Node.js pour préparer les données, ce qui peut complexifier le workflow.
* ****Absence de tests parallèles natifs pour une meilleure scalabilité**** : Cypress ne prend pas nativement en charge l'exécution de tests parallèles, bien que cela puisse être configuré dans un environnement CI/CD payant avec Cypress Cloud. Sans cette option, les tests sur de grands ensembles peuvent être plus lents et moins scalables.
* ****Limites avec l'authentification et les cookies sur plusieurs domaines** :** Cypress n’autorise pas les interactions avec plusieurs domaines dans un même test. Cela peut être contraignant pour les applications nécessitant des redirections vers différents sous-domaines (comme des portails d’authentification). Les tests nécessitant des interactions entre plusieurs domaines nécessitent souvent des solutions de contournement.
* ****Problèmes avec les tests visuels et les animations** :** Cypress n'est pas spécifiquement conçu pour les tests visuels (tests d'interface utilisateur qui comparent les rendus de pages). De plus, les animations peuvent rendre certains éléments instables dans les tests, obligeant les développeurs à gérer ces effets en ajustant les délais ou les animations.
* ****Difficulté avec les actions réseau complexes (ex. Web Sockets)** :** Cypress prend en charge les requêtes HTTP, mais il est moins performant pour les tests basés sur des connexions en temps réel, comme les Web Sockets, car il ne les intercepte pas de la même manière que les requêtes HTTP. Les tests qui impliquent des applications en temps réel ou de streaming nécessitent parfois des configurations et des tests spécifiques difficiles à intégrer avec Cypress.
* ****Limites dans la gestion des éléments d’interface complexe (ex. Canva, SVG)** :** Cypress a parfois des difficultés à interagir avec des éléments d'interface non-standards comme les éléments canvas et les SVG complexes, car ceux-ci ne se comportent pas comme des éléments HTML traditionnels. Les tests de précision sur des éléments graphiques, comme des graphiques ou des outils de dessin en ligne, peuvent être complexes à automatiser.
* ****Complexité de maintenance des tests lourds en interaction**** : Bien que Cypress soit simple pour des tests standards, les tests lourds qui requièrent des interactions complexes (comme des formulaires longs, des interactions en drag-and-drop, ou des tests multi tâches très détaillés) peuvent parfois être plus instables. Cela peut nécessiter des ajustements fréquents des tests en raison de la manière dont Cypress gère le DOM et les événements.

En somme, même si Cypress est très performant pour les tests end-to-end sur des applications web modernes, il peut rencontrer des limitations pour des tests de haute complexité ou des fonctionnalités spécifiques. Les alternatives comme Selenium ou Playwright peuvent offrir plus de flexibilité pour les tests impliquant des multifenêtres, des tests mobiles natifs, ou des besoins de tests multi-domaines.