**Implémentation des pratiques DevOps pour l'optimisation des systèmes de gestion aéronautiques à l'ASECNA Cameroun.**

Mémoire en vue d’obtention du diplôme de :

Bachelor en Génie Logiciel

Rédigé et présenté p ar :

**MOUKOURI VICTOR ERIC**

Encadreur académique : Encadreur académique :

M. MENDJEMEN M. NGUEA Aristide

Année académique 2023/2024

Dédicace à ma famille.

Remerciements

Ces remerciements sont adressés à toutes les personnes qui de près ou de loin m’ont apporté leur aide pour la réalisation de ce travail, qu’elle ait été matérielle, morale, financière, ou encore en rapport direct avec mon étude. Ainsi je dis sincèrement merci à :

● M. MENDJEMEN pour son encadrement durant ce projet de fin d’études, pour l’aide, le support et les différentes directives concernant les concepts évoqués dans ce projet qu’il m’a apporté lors du suivi de sa conception.

● M. NGUEA Aristide pour la confiance, le savoir et la passion pour le métier accordé tout au long de la période de stage.

● Tous les membres de l’administration de KEYCE INFORMATIQUE & IA pour leur soutien.

● Tout le corps enseignant de KEYCE INFORMATIQUE & IA pour leur savoir, leur savoir-faire.

● A mes sœurs SOPPO MOUKOURI VIVIANE THERESE et KOLLE MOUKOURI REGINE ANGELE et également à MME TEKLA POUT pour le soutien moral, spirituel, financier et pour les nombreux conseils.

● Mes oncles et tantes pour le soutien spirituel.

● Mes camarades de promotion pour leur soutien académique et moral.

● Également à vous, membres du jury et tous ceux ici présents.

RÉSUMÉ

L’Agence pour la Sécurité de la Navigation Aérienne en Afrique et à Madagascar (ASECNA) joue un rôle crucial dans la gestion de l’espace aérien de plusieurs pays africains, y compris le Cameroun. L’intégration des pratiques DevOps au sein de l’ASECNA au Cameroun pourrait permettre d’optimiser les systèmes critiques de gestion aéronautique, en améliorant la collaboration entre les équipes de développement et d’exploitation.

Les DevOps sont un ensemble de pratiques qui visent à automatiser et à intégrer les processus entre les équipes de développement logiciel (Dev) et celles des opérations informatiques (Ops). L’objectif principal de DevOps est d’améliorer la collaboration entre ces deux équipes pour accélérer le cycle de vie du développement logiciel, tout en maintenant une haute qualité et une grande fiabilité des applications livrées.

Mots clés : DevOps, optimiser les systèmes, améliorer la collaboration, automatiser, intégrer les processus,

ASTRACT

The Agency for the Safety of Air Navigation in Africa and Madagascar (ASECNA) plays a crucial role in the management of the airspace of several African countries, including Cameroon. The integration of DevOps practices within ASECNA in Cameroon could help optimize critical aviation management systems, improving collaboration between development and operations teams.

DevOps is a set of practices that aim to automate and integrate processes between software development (Dev) and IT operations (Ops) teams. The main goal of DevOps is to improve collaboration between these two teams to accelerate the software development lifecycle, while maintaining high quality and reliability of the applications delivered.

Keywords: DevOps, optimize systems, improve collaboration, automate, integrate processes,

LISTE DES TABLEAUX

Tableau 1 : Les tâches principales de l’ASECNA, origine et objectifs .........................

Tableau 2 : Les concurrents de l’ASECNA sur le plan international ..........................

Tableau 3 : Les concurrents de l’ASECNA en Afrique et sur le plan national ..............

Tableau 4 : Les spécificités de l’ASECNA face à ses concurrents ..............................

Tableau 5 : Analyse comparative des systèmes de gestion aéronautiques existants ..

Tableau 6 : Aspect méthodologique par les méthodes de modélisation ....................

Tableau 7 : Dictionnaire de données .......................................................................

Tableau 8 : Tableaux comparatifs des solutions ......................................................

Tableau 9 : Analyse approximatif des coûts de solutions ..........................................

Tableau 10 : Cout approximatif du matériel utilisé ...................................................

Tableau 11 : Récapitulatifs des couts liés au projet ..................................................

LISTES DES FIGURES

Figure 1 : Organigramme général de l’ASECNA ..........................................................

SIGLES ET ABREVATIONS

SOMMAIRE

DÉDICACES ..............................................................................................................

REMERCIEMENTS .....................................................................................................

RÉSUMÉ ..................................................................................................................

ASTRACT .................................................................................................................

LISTE DES TABLEAUX ...............................................................................................

LISTES DES FIGURES ...............................................................................................

SIGLES ET ABRÉVIATIONS .......................................................................................

SOMMAIRE ..............................................................................................................

INTRODUCTION GENERALE .....................................................................................

CHAPITRE 1 : PRESENTATION DE L’AGENCE POUR LA SECURITE DE LA

NAVIGATION AERIENNE EN AFRIQUE ET A MADAGASCAR .........................................

I. Genèse de l’ASECNA ......................................................................

II. Environnement de l’ASECNA ................................................................................

III. Les concurrents de l’ASECNA ...........................................................................

IV. Déroulement du stage .......................................................................................

Conclusion ............................................................................................................

CHAPITRE 2 : .....................................................................................................

I. .................................................

II. .................................

III. ................................

Conclusion ............................................................................................................

CHAPITRE 3 : ANALYSE FONCTIONNELLE ...............................................................

I. Analyse des besoins et exigences ........................................................................

II. Cahier de charges ..............................................................................................

Conclusion ............................................................................................................

CHAPITRE 4 : IMPLEMENTATION DE LA SOLUTION ..................................................

I. Préparation de l’environnement ...........................................................................

II. Installation et configuration des différents services .............................................

III. Test et validation ...............................................................................................

CONCLUSION GENERALE ......................................................................................

TABLES DES MATIÈRES ..........................................................................................

RÉFÉRENCES BIBLIOGRAPHIQUES ........................................................................

RÉFÉRENCÉS WEBOGRAPHIQUES .........................................................................

ANNEXES ..............................................................................................................

**Introduction Générale**

La conception et la réalisation de solutions numériques doit son succès non pas seulement aux développeurs et aux marketeurs, mais plus crucialement aux testeurs logiciels. Les **tests logiciels** sont des processus systématiques visant à évaluer la qualité, la fiabilité et la conformité d’un logiciel par rapport à ses exigences. Ils permettent d'identifier les éventuelles erreurs ou failles avant le déploiement, garantissant une meilleure expérience utilisateur et réduisant les coûts liés aux corrections avant son lancement en production.

Dans le développement de solutions numériques, l’implémentation d’un système de tests automatisés, tel que celui fourni par Cypress, joue un rôle essentiel pour relever les défis de qualité, de fiabilité et de rapidité dans un contexte où les cycles de développement et de déploiement sont de plus en plus courts. Cependant, l’intégration de tests automatisés apporte des contraintes spécifiques liées à la conception, au développement et au déploiement de solutions numériques, qui nécessitent des approches bien planifiées pour garantir un déploiement fluide et efficace.

Dans le cadre de l’amélioration continue de la qualité logicielle au sein de l’entreprise ADWA Sarl, la mise en place d’un système de tests automatisés est devenue une priorité stratégique. Les tests automatisés permettent non seulement de détecter rapidement les erreurs et dysfonctionnements, mais aussi de renforcer la robustesse des solutions fournies aux clients. Parmi les divers outils disponibles, Cypress s’avère particulièrement adapté pour les tests de bout en bout des applications web, offrant une solution complète et flexible par rapport aux autres outils comme Selenium, Jest, Postman, etc...

L’objectif principal de cette étude est de concevoir et d'implémenter un système de tests automatisés pour automatiser les cas de test de l'infrastructure logicielle d'ADWA Sarl. Cela inclut les tests unitaires, fonctionnels, d’interface utilisateur, et de régression pour garantir que les applications répondent aux exigences fonctionnelles et qualitatives définies par l’entreprise. Notre mission finale sera alors d'optimiser de manière significative les coûts et délais défiant les précédents dans les projets à venir.

Nous verrons donc exhaustivement dans la suite de cette robuste introduction les différentes étapes parcourues ou à parcourir pour parvenir à nos fins les plus professionnelles. Il s'agira en outre : des recherches et études sur les tests logiciels et leurs types, de l'installation, configuration et mise en place d'un environnement de tests automatisés avec l'outil Cypress ainsi que la réalisation effective sur une solution de l’entreprise.

**CHAPITRE 1 : ETAT DE ART.**

Dans ce chapitre nous nous soumettrons à la présentation et à l’étude la prestigieuse agence qu’est ADWA en tant que service IT.

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 :

* Tableau descriptif des différents types de tests.

|  |  |  |
| --- | --- | --- |
| **Type de test** | **Objectif** | **Importance** |
| **Tests unitaires** | Tester les plus petites unités de code (souvent des fonctions ou méthodes) pour s'assurer qu'elles fonctionnent individuellement comme prévu. | 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** | Tester l'interaction entre plusieurs composants ou modules pour vérifier qu'ils fonctionnent bien ensemble. | Les modules peuvent fonctionner correctement seuls, mais des erreurs peuvent survenir lorsqu'ils interagissent. De plus ils valident les flux de données entre les systèmes ou services. |
| **Tests de bout en bout (E2E)** | Tester le système complet dans son environnement global pour s'assurer qu'il répond aux exigences initiales.  . | Ce type de test vérifie l'application du point de vue de l'utilisateur final et est une étape cruciale avant la mise en production. |

Tableau présentant les différents types de tests.

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).

**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 (nécessite des 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 et la popularité 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.

**CHAPITRE 2 : METHODOLOGIE ET RESOLUTION DU PROBLEME**

# Analyse des Besoins et Cahier des Charges

**CAHIER DE CHARGES : OUTIL DE TESTS AUTOMATISES CYPRESS**

**Identification des besoins :**

L’outil Cypress servira pour les tests de bout en bout automatisés pour des applications web modernes. Il doit permettre aux développeurs d'écrire, exécuter et maintenir des tests automatisés pour améliorer la qualité du code.

* Cypress doit pourvoir accéder à un projet tiers pour jouer son rôle.
* Assurer la stabilité et la fiabilité des applications web via des tests automatisés.
* Réduire les erreurs humaines et les coûts de maintenance en identifiant rapidement les bugs et en facilitant leur reproduction.
* Offrir une interface simple pour l’écriture et la gestion des tests.

**Exigences du projet :**

* **Tests de bout en bout (E2E)** : Cypress doit permettre de simuler l’expérience utilisateur complète, de la navigation aux interactions complexes sur l’interface.
* **Tests en temps réel** : Cypress exécute les tests en temps réel, montrant instantanément les changements et erreurs détectées dans l’interface.
* **Rechargement automatique** : Recharger les tests en direct dès qu’une modification est faite dans le code source.
* **Support des assertions et des sélecteurs** : Cypress doit offrir une variété d’assertions (vérifications) et la possibilité de sélectionner des éléments par divers attributs (ex. id, class).
* **Support des mocks et des stubs** : Simuler des réponses de serveur pour tester les cas où le backend ne fonctionne pas.
* **Rapports détaillés** : Générer des rapports des tests pour faciliter l’analyse des erreurs.
* **Intégration avec CI/CD** : Faciliter l’intégration avec des plateformes de CI/CD (intégration et déploiement continus) comme Jenkins, GitLab CI ou BitBucket dans notre cas d’utilisation.

**Contraintes :**

• **Contraintes de temps** :

* Cypress doit être compatible avec les navigateurs courants (Chrome, Edge, Firefox) et leur dernière version stable.
* Il doit fonctionner avec les applications écrites en JavaScript, TypeScript ou d’autres frameworks populaires (React, Angular, Vue).

• **Contraintes de temps** :

* Livrer une version stable dans les six mois suivant le début du développement, avec des fonctionnalités de base.

• **Contraintes de budget** :

* Limiter les coûts d’infrastructure pour l’intégration dans des pipelines de tests continus.  
  • **Contraintes juridiques** :
* Respecter les lois de protection des données pour les tests qui manipulent des données sensibles.

**Livrables et qualité attendue :**

• **Livrables attendus** :

* Documentation complète sur l’installation, la configuration et l’utilisation de Cypress.
* Des exemples de tests de démonstration pour aider les utilisateurs à se familiariser avec l’outil.
* Un support de communauté et des guides pour intégrer Cypress avec des outils populaires de CI/CD.

• **Critères de qualité** :

* Cypress doit fournir une interface utilisateur ergonomique et facile à prendre en main pour les développeurs et testeurs.
* Le taux d’erreurs et de bugs dans l’outil Cypress doit être inférieur à 1 % pour garantir la fiabilité des tests.
* Les tests doivent s’exécuter rapidement (en moins de 3 secondes par action sur l’interface).

**Organisation et suivi du projet :**

• **Rôles et responsabilités** :

• **Développeur** : Créer et maintenir les fonctionnalités de Cypress.

• **Testeur** : Tester Cypress pour vérifier qu’il répond aux besoins en simulation de flux utilisateur. En ressortir le journal de tests effectif.

• **Chef de projet** : Gérer le développement, organiser les mises à jour et vérifier la conformité du produit avec les exigences.

• **Gestion du projet** : Utiliser une méthodologie agile pour une flexibilité optimale.

• **Modalités de suivi** :

* Mise en place de points de suivi bi-hebdomadaires pour évaluer les progrès.
* Tests de validation à chaque phase de développement.

2.2 Choix de l'Architecture et des Scénarios de Test

• Architecture du Système de Tests : Schéma et organisation des répertoires, modules et configurations nécessaires pour Cypress

Structure du projet Cypress

Cypress crée les dossiers suivants dans votre projet :

cypress/e2e/ : Pour vos tests End-to-End.

cypress/fixtures/ : Pour les données de test.

cypress/support/ : Pour les fichiers utilitaires ou de configuration.

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

const fs = require('fs');

const path = require('path');

module.exports = defineConfig({

  projectId: '6nf6ro',

  projectID : 'cypadw ',

  e2e: {

baseUrl: 'http://127.0.0.1:3000/Bootstrap/startbootstrap-landing-page/dist/index.html',

}

})

• Définition des Scénarios de Test : Choix des scénarios en fonction des besoins de l'entreprise et des objectifs du projet

2.3 Outils et Processus de Développement

• Installation et Configuration de Cypress : Procédure d'installation, paramétrage, intégration avec le pipeline CI/CD

\* Mise en place d'un environnement de test de logiciels avec Cypress

L’exécution de tests automatisés via Cypress nécessite un environnement sollicitant 3 principaux outils qui sont :

Node.js

Node.js est un environnement d'exécution JavaScript côté serveur qui permet aux développeurs d'utiliser JavaScript pour créer des applications backend. Notamment réputé pour être rapide et efficace dans la gestion des applications en temps réel, des requêtes réseau et des systèmes d'entrées/sorties intensives.

### Principales caractéristiques de Node.js

1. \*\*Architecture non bloquante et asynchrone\*\* : Node.js utilise un modèle d'entrées/sorties non bloquant, ce qui signifie que les opérations peuvent s'exécuter sans attendre la fin des précédentes. Cela le rend particulièrement performant pour les applications nécessitant un grand nombre de requêtes simultanées, comme les applications de chat en temps réel ou les systèmes de streaming.

2. \*\*Évolutivité\*\* : Grâce à son modèle basé sur des événements, Node.js est capable de gérer des milliers de connexions simultanées, ce qui le rend particulièrement adapté pour les applications nécessitant une grande scalabilité.

3. \*\*Écosystème riche de modules\*\* : Node.js dispose d'un gestionnaire de paquets appelé \*\*npm\*\* (Node Package Manager), qui offre des milliers de bibliothèques et de modules permettant d’étendre ses fonctionnalités sans avoir à écrire du code depuis zéro.

### Pourquoi choisir Node.js ?

Node.js est souvent préféré pour ses performances élevées, sa capacité à gérer de grandes charges de trafic, et pour sa flexibilité dans les applications nécessitant une communication en temps réel. Grâce à son architecture légère et rapide, il est devenu un choix populaire pour les startups comme pour les grandes entreprises.

## Comment installer Node.js ?

Il existe plusieurs méthodes pour installer Node.js en fonction de votre système d'exploitation, de vos besoins (environnement de production ou de développement) et de votre préférence pour la gestion des versions. Voici les méthodes d'installation les plus courantes :

1. Installer Node.js avec le programme d'installation officiel

Allez sur le site officiel de Node.js nodejs.org et téléchargez l'installateur pour votre système d'exploitation (Windows, macOS ou Linux). Un fois sur le site officiel, il faudra cliquer sur le bouton "Télécharger Node.js (LTS)", ce qui nous permettra d'obtenir la version v22.11.0 étant la plus adaptée aujourd'hui car plus stable.

Une fois le téléchargement terminé, exécutez le programme d’installation et suivez les instructions à l’écran.

Cette méthode est simple et rapide, mais elle ne permet pas de gérer plusieurs versions de Node.js facilement.

2. Utiliser un gestionnaire de versions de Node (nvm)

nvm (Node Version Manager) est un gestionnaire de versions Node.js qui permet d’installer et de basculer entre différentes versions de Node.js sur le même système.

Pour l'installer, téléchargez nvm depuis le dépôt officiel sur GitHub et suivez les instructions.

Une fois nvm installé, vous pouvez utiliser des commandes comme nvm install <version> pour installer une version spécifique de Node.js, et nvm use <version> pour basculer entre les versions.

Cette méthode est idéale pour les développeurs qui ont besoin de tester des applications sur différentes versions de Node.js.

. Installer avec Docker

Si vous travaillez avec des conteneurs, vous pouvez utiliser Docker pour créer un environnement Node.js sans l'installer directement sur votre système.

Créez un fichier Dockerfile avec l’image Node.js, par exemple :

dockerfile

FROM node:18

Cette méthode est utile pour isoler les environnements et garantir que votre application tourne de manière cohérente sur différents systèmes.

Ensuite pour vérifier que vous avez éffectivement installé Node.js, dans le cas d'un OS Windows :

- nous ouvrirons l'invite de commande encore plus communément appelé cmd pour taper ensuite les commandes :

- node -v qui nous affichera la version de Node.js présente dans notre système.

- npm -v qui nous affichera la version de Node Package Manager présente dans notre système.

II / Visual Studio Code

Visual Studio et Visual Studio Code sont deux outils de développement créés par Microsoft, mais ils ont des objectifs et des fonctionnalités différentes.

Visual Studio Code, souvent abrégé en VS Code, est un éditeur de code source léger et flexible, conçu pour être rapide et personnalisable. Contrairement à Visual Studio, VS Code est plutôt un éditeur de texte avancé qu'un IDE complet.

Fonctionnalités et utilisations principales de Visual Studio Code :

- Édition de code source : Il prend en charge une large gamme de langages de programmation, comme JavaScript, Python, TypeScript, Go, et bien d’autres, avec la possibilité d’ajouter des extensions pour d’autres langages.

- Personnalisation avec les extensions : VS Code a un écosystème d’extensions très riche, permettant aux développeurs d’ajouter des fonctionnalités supplémentaires, comme le débogage, le contrôle de version, le formatage de code, et même l’intégration avec des services cloud.

- Développement web et JavaScript : Il est très populaire pour le développement web grâce à des fonctionnalités intégrées comme l’auto-complétion, le débogage JavaScript, et l’intégration avec des outils de développement frontend.

- Interface de ligne de commande intégrée : Il offre un terminal intégré qui permet aux développeurs de lancer des commandes directement depuis l'éditeur, facilitant l'exécution de scripts, de commandes Git, ou encore de commandes de gestion de packages comme npm.

- Débogage : Il propose un environnement de débogage basique, idéal pour les projets JavaScript, Python, et Node.js, entre autres, avec des points d’arrêt et l’inspection des variables.

- Contrôle de version : Il intègre des outils de contrôle de version comme Git, ce qui permet aux développeurs de gérer leur code source directement dans l’éditeur.

Pourquoi utiliser Visual Studio Code ?

VS Code est largement utilisé pour son léger, sa vitesse et son extensibilité. Il est idéal pour :

- Les développeurs front-end et back-end.

- Les projets légers ou les prototypes.

-Ceux qui veulent un éditeur personnalisable avec des outils de développement modernes.

- Ceux qui recherchent un éditeur multiplateforme, car VS Code est disponible sur Windows, macOS et Linux.

En résumé, Visual Studio est un IDE complet pour des projets complexes, tandis que Visual Studio Code est un éditeur léger et modulaire, particulièrement adapté aux développeurs de web et de projets nécessitant de la flexibilité.

# Comment installer Visual Studio Code ?

Visual Studio code est en principe très facile d'acquisition et d'accès et nous avons la possibilité de l'obtenir en suivant les étapes qui suivent ce magnifique paragraphe introductif qui sera brutalement interrompu par :

**1. Accédez au site officiel**

* Rendez-vous sur le site officiel de Visual Studio Code en utilisant ce lien : [Visual Studio Code](https://code.visualstudio.com/).

**2. Sélectionnez votre système d'exploitation**

* Sur la page d'accueil, cliquez sur le bouton correspondant à votre système d'exploitation :
  + **Windows**
  + **macOS**
  + **Linux**

**3. Téléchargez l'installateur**

* Une fois que vous avez sélectionné votre système d'exploitation, le téléchargement du fichier d'installation commencera automatiquement.
* Si ce n’est pas le cas, cliquez sur le lien fourni pour démarrer le téléchargement.

**4. Installez Visual Studio Code**

**Pour Windows :**

1. Double-cliquez sur le fichier téléchargé (ex. VSCodeSetup.exe).
2. Suivez les instructions de l’assistant d’installation :
   * Acceptez les termes et conditions.
   * Sélectionnez les options comme l’ajout de VS Code au menu contextuel.
3. Cliquez sur **Installer**, puis sur **Terminer** une fois l’installation achevée.

**5. Lancez Visual Studio Code**

* Une fois installé, ouvrez l'application et explorez son interface conviviale.

CYPRESS

**Initialisation du projet**

1. **Créez un répertoire de projet** (si ce n'est pas déjà fait) :

mkdir mon-projet-cypress

cd mon-projet-cypress

1. **Initialisez un fichier package.json** dans le projet :

npm init -y

Cela génère un fichier package.json avec les configurations par défaut.

**3. Installation de Cypress**

1. Installez Cypress en tant que dépendance de développement avec npm :

npm install cypress --save-dev

1. Vérifiez que Cypress a été installé correctement en exécutant :

npx cypress verify

**4. Ouvrir Cypress**

1. Pour la première fois, ouvrez Cypress pour initialiser les fichiers de configuration :

npx cypress open

1. Une interface graphique s'ouvrira. Cypress crée automatiquement un répertoire cypress/ et un fichier de configuration cypress.config.js dans votre projet.

**6. Ajouter un script pour Cypress dans package.json**

Ajoutez une commande pour exécuter Cypress depuis le terminal :

"scripts": {

"test": "cypress run",

"cypress:open": "cypress open"

}

Cela vous permet de démarrer Cypress avec :

npm run cypress:open

**3. Intégration avec le pipeline CI/CD**

Cypress peut être intégré dans des outils de CI/CD comme Jenkins, GitHub Actions, GitLab CI, etc., pour exécuter vos tests automatiquement à chaque commit.

1. **Ajoutez Cypress aux scripts npm** : Modifiez votre package.json pour inclure les commandes suivantes :

"scripts": {

"test": "cypress run",

"cypress:open": "cypress open"

}

1. **Configurer Cypress pour CI/CD** :
   * Ajoutez un fichier de configuration spécifique à votre outil CI/CD.

name: Cypress Tests

on:

push:

branches:

- main

jobs:

cypress-run:

runs-on: ubuntu-latest

steps:

- name: Checkout code

uses: actions/checkout@v3

- name: Setup Node.js

uses: actions/setup-node@v3

with:

node-version: 16

- name: Install dependencies

run: npm install

- name: Run Cypress tests

run: npm run test

**Résumé des bénéfices**

* **Installation simple** : Une commande suffit pour démarrer.
* **Configuration flexible** : Adaptez les paramètres à votre projet.
* **Intégration fluide avec CI/CD** : Automatisez vos tests pour garantir une livraison continue fiable.

**• Workflow de Tests : Description du processus de création, validation, et exécution des tests**

**Workflow de Tests avec Cypress**

Le workflow de tests avec Cypress est un processus structuré qui inclut la création, la validation, et l’exécution des tests.

**1.3 Rédaction des Tests**

Utilisez des commandes Cypress pour écrire des tests. Exemple d’un test de connexion :

describe('Test de connexion', () => {

it('Vérifie qu’un utilisateur peut se connecter avec des identifiants valides', () => {

cy.visit('/login'); // Accès à la page de connexion

cy.get('#email').type('utilisateur@example.com'); // Remplir le champ email

cy.get('#password').type('motdepasse123'); // Remplir le champ mot de passe

cy.get('button[type="submit"]').click(); // Soumettre le formulaire

cy.url().should('include', '/dashboard'); // Vérifier la redirection

});

});

**2. Validation des Tests**

**2.1 Exécution locale pour validation**

* Lancez les tests en local pour vérifier leur comportement.
* Commande pour exécuter les tests :

npx cypress open

* Choisissez le test dans l’interface graphique de Cypress.

**2.2 Débogage**

* Utilisez les outils intégrés de Cypress pour inspecter les tests.
  + **Console d’exécution** : Affiche les commandes exécutées et leurs résultats.
  + **Time Travel** : Permet de revoir chaque étape d’un test.
* Exemple de validation d’éléments :

cy.get('#message').should('contain.text', 'Connexion réussie');

**2.3 Refactorisation**

* Optimisez les tests pour qu’ils soient :
  + **Lisibles** : Utilisez des fonctions utilitaires.
  + **Réutilisables** : Centralisez les étapes répétées dans support/commands.js.

**3. Exécution des Tests**

**3.1 Exécution locale**

* Exécutez les tests via l’interface graphique ou en ligne de commande :

npx cypress run

**3.3 Génération de rapports**

* Utilisez des outils comme **Mochawesome** pour créer des rapports détaillés :

npm install mochawesome --save-dev

**3.4 Surveillance continue**

* Enregistrez les résultats des tests (logs, vidéos, captures d’écran) pour identifier les régressions dans les versions ultérieures.

**Importance du Workflow**

1. **Création structurée** :
   * Réduit les erreurs dans la rédaction des tests.
   * Assure une couverture complète des fonctionnalités critiques.
2. **Validation efficace** :
   * Facilite la détection et la correction des erreurs avant l’intégration.
3. **Exécution automatisée** :
   * Permet un feedback rapide sur la qualité du code.
   * Garantit une livraison continue fiable.

**CHAPITRE 3 : RÉALISATION**

**3.1 Introduction**

Dans ce chapitre, nous présentons la conception et l’implémentation d’un système de tests automatisés avec Cypress. L’objectif principal est d’assurer la qualité des applications en validant les fonctionnalités critiques et en identifiant les anomalies potentielles. Le projet portera sur les fonctionnalités de la **Landing Page Bootstrap** en vue d’adapter ultérieurement cette solution à d’autres projets.

**Objectifs du projet**

* Valider les fonctionnalités critiques de l’application.
* Identifier les anomalies présentes et garantir la qualité logicielle.
* Mettre en place une architecture flexible pour les tests automatisés adaptable à divers projets.

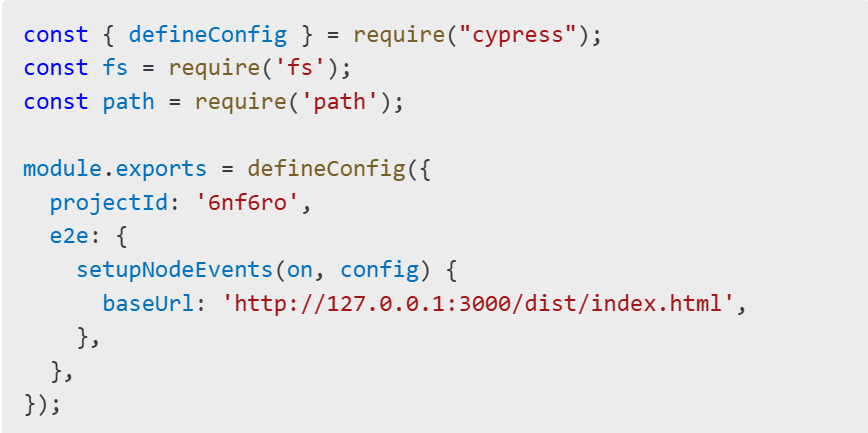
**Outil utilisé : Cypress (version 13.15.0)**

**Cypress** est un outil puissant pour les tests end-to-end (E2E) qui simplifie la validation des applications web.

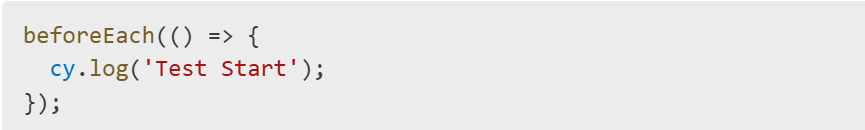
**3.2 Mise en place du système de tests automatisés**

**3.2.1 Configuration initiale**

La configuration du fichier **cypress.config.js** est une étape essentielle. Ce fichier contient les paramètres globaux, tels que l’URL de base de l’application et les configurations liées aux différents environnements.



**Ajout d’un message de démarrage des tests :** Dans le fichier cypress/support/e2e.js :



Ce message apparaît au début de chaque exécution de test.

**3.2.2 Rédaction des cas de test**

Les cas de test à implémenter ont été définis dans le tableau ci-dessous :

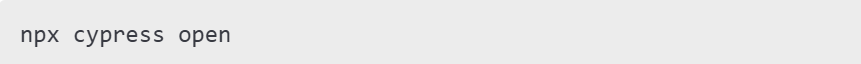
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **ID** | **Nom du Test** | **Description** | **Type de Test** | **Priorité** | | TC1 | Accès et chargement de Landing Page | Accéder à l’URL et vérifier que tous les éléments se chargent correctement | Interface utilisateur | Haute | | TC2 | Vérification de la barre de navigation | Vérifier l’affichage et l’état de la barre de navigation | Interface utilisateur | Moyenne | | TC3 | Validation de l’email | Vérifier les contraintes du champ d’entrée de l’email et la soumission | Unitaire, interface utilisateur | Haute | | TC4 | Affichage des fonctionnalités | Vérifier l’affichage correct des icones et leur contenu | Interface utilisateur | Faible | | TC5 | Vérification de la section de témoignages | Vérifier l’authenticité et l’affichage des témoignages | Interface utilisateur | Faible | | TC6 | Vérifications du footer | Vérifier les éléments et liens du footer | Interface utilisateur, fonctionnel | Moyenne | |

**3.2.3 Premier test : Accès à la Landing Page**

Voici un exemple de test automatisé :

 **3.2.4 Exécution des tests**

Pour exécuter les tests, utilisez la commande suivante :



L’interface graphique de Cypress permet de sélectionner le navigateur et le fichier de test à exécuter.

**3.3 Résultats des tests**

|  |  |  |  |
| --- | --- | --- | --- |
| ID | Nom du Test | Statut | Observation |
| TC1 | Accès et chargement de Landing Page | OK | RAS |
| TC2 | Vérification de la barre de navigation | OK | RAS |
| TC3 | Validation de l’email | OK | RAS |
| TC4 | Affichage des fonctionnalités | OK | RAS |
| TC5 | Vérification de la section de témoignages | OK | RAS |
| TC6 | Vérifications du footer | OK | RAS |

**Rapport technique :**

* Total de tests : 6
* Tests réussis : 6
* Durée totale : 9,56 secondes
* Rapport généré : cypress/results/test-results.json

**3.4 Configuration avancée : Gestion des environnements**

**3.4.1 Importance des environnements multiples**

* **Flexibilité :** Adapte les scripts aux différents environnements (staging, production).
* **Fiabilité :** Identifie les problèmes avant leur impact en production.
* **Efficacité :** Centralise les paramètres pour réduire les modifications manuelles.

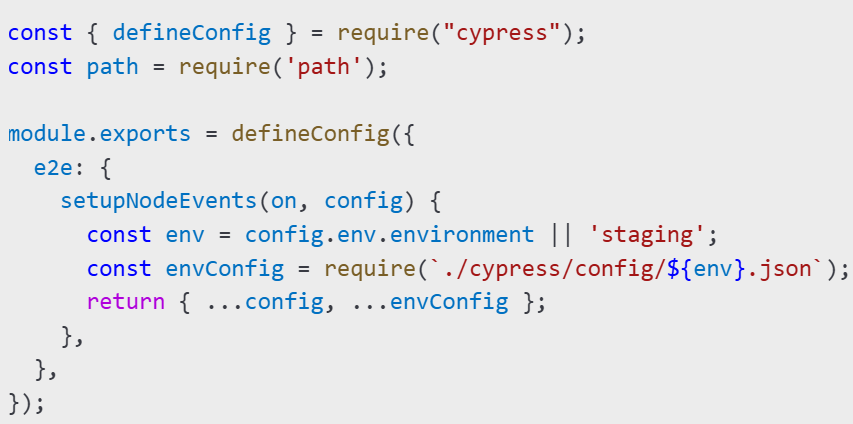
**3.4.2 Configuration des environnements dans Cypress**

Créer des fichiers JSON pour chaque environnement dans le dossier cypress/config :

**Exemple : staging.json**



**Charger la configuration dynamiquement dans cypress.config.js :**



**Exécution pour un environnement spécifique :**



**3.5 Stratégies de maintenance et d’évolution**

**3.5.1 Maintenance des tests**

* **Mises à jour régulières :** Les tests doivent être ajustés en fonction des modifications du code ou des nouvelles fonctionnalités.
* **Révision périodique des cas de tests :** Identifier et retirer les tests obsolètes.
* **Utilisation des hooks :** Simplifier la maintenance avec des fonctions globales (before, after, etc.).

Exemple d’utilisation de hooks dans Cypress :



**3.5.2 Évolutivité des tests**

* **Adoption d’un modèle modulaire :** Diviser les tests en composants réutilisables pour simplifier l’ajout de nouvelles fonctionnalités.
* **Automatisation dans le pipeline CI/CD :** Intégrer les tests dans des outils tels que Jenkins, GitLab CI/CD, ou Bitbucket Pipelines.

**3.6 Intégration avec le pipeline CI/CD**

**3.6.1 Rôle de l’automatisation**

L’intégration des tests dans le pipeline CI/CD garantit :

* Une détection rapide des régressions.
* Une validation continue des builds.

**3.6.2 Configuration avec Bitbucket Pipelines**

Voici un exemple de fichier bitbucket-pipelines.yml pour l’exécution des tests Cypress :



**Observation des résultats :** Les résultats des tests sont automatiquement remontés dans Bitbucket pour consultation.

**3.7 Limites et perspectives**

**3.7.1 Limites actuelles**

* **Adaptabilité :** Les tests sont spécifiquement conçus pour une application Bootstrap, ce qui limite leur réutilisation immédiate.
* **Environnement d’exécution :** La configuration des environnements multiples nécessite une gestion rigoureuse pour éviter les erreurs.
* **Temps d’exécution :** Bien que rapide, l’exécution peut être optimisée pour des suites de tests plus volumineuses.

**3.7.2 Perspectives**

* Étendre la solution à d’autres frameworks (React, Angular).
* Incorporer des outils de reporting avancés comme Allure ou Mochawesome.
* Augmenter la couverture des tests pour inclure des scénarios complexes (stress tests, tests de performance).