NUMÉRO 2 : AOUT 2025

QUALITÉ & TESTS

DOSSIER : OUTILS D'AUTOMATISATION DE TESTS WEB

CYPRESS, PLAYWRIGHT, ROBOT FRAMEWORK

INTERVIEW D'ISABELLE HOAREAU : DE LA QA AU RÔLE DE CHEF DE PROJET

LE MAG DU TESTEUR

LETTRE DE L'EDITEUR

Chers lecteurs,

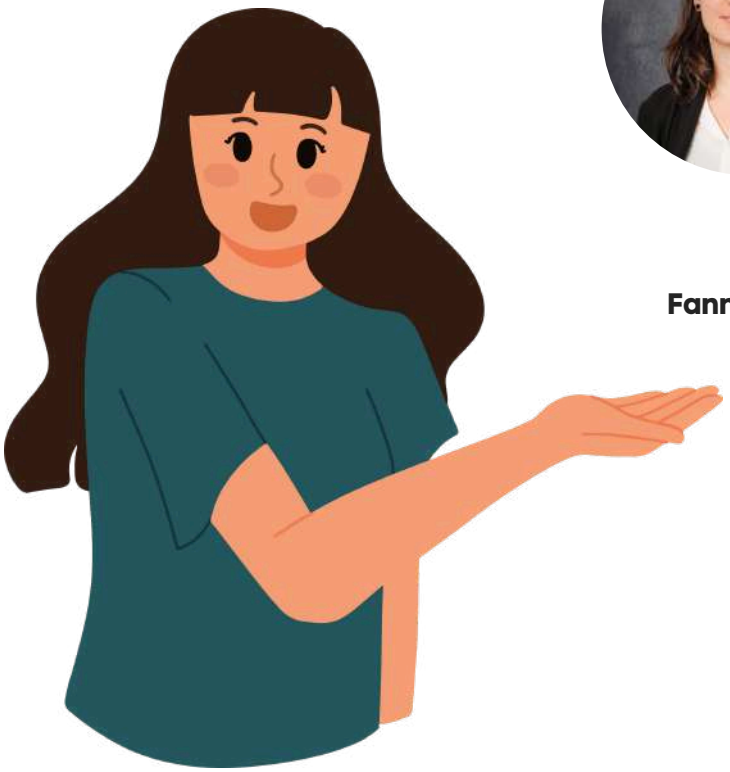
Bienvenue dans ce deuxième numéro de Le Mag du Testeur, un espace pensé pour vous, passionnés et professionnels de la qualité logicielle, qui cherchez à approfondir vos connaissances et découvrir les dernières pratiques en matière de test.

Ce deuxième numéro a été plus long a produire que prévu et je m'en excuse. Je vais faire preuve de plus d'assiduité pour les prochains numéros.

Bonne lecture, et n'hésitez pas à me faire part de vos retours et de vos expériences – car c'est ensemble que nous continuerons à élever les standards de qualité.

Avec enthousiasme,

Fanny Velsin



SOMMAIRE

5 Introduction aux tests fonctionnels

9 Le cycle de vie des tests

11 QA VS QC VS testeur

13 Tests manuels vs automatisés

15 Interview d'Isabelle HOAREAU

19 L'IA va nous remplacer ?

21 Le Shift-Left Testing

24 Gherkin

26 Comment récupérer les éléments ?

28 Dossier : Outils d'automatisation de tests web

30 Sauce Labs

36 Avis d'expert : Yassine Sidki

38 Cypress

SOMMAIRE

48 Playwright

58 Comment structurer une suite de tests automatisés efficace ?

60 Comment maintenir ses tests automatisés sur le long terme ?

63 Comment calculer le ROI d'une campagne d'automatisation ?

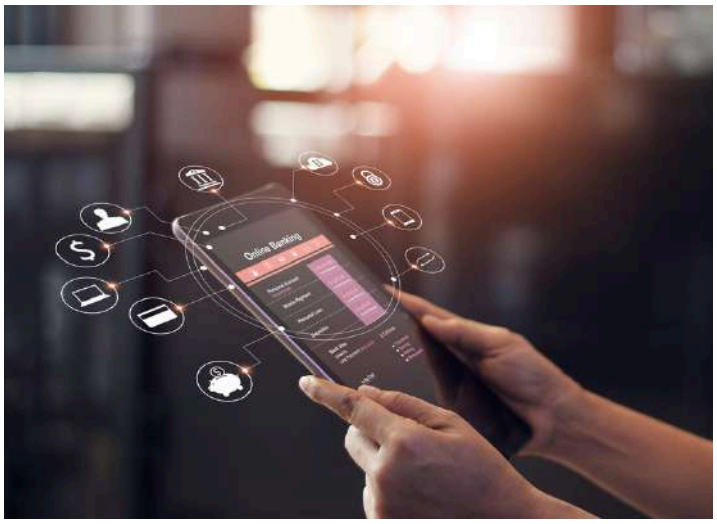
65 CI/CD

69 Jeu concours

70 Podcast de Nancaidah TOURE CHAUVIN

71 Teasing

INTRODUCTION AUX TESTS FONCTIONNELS

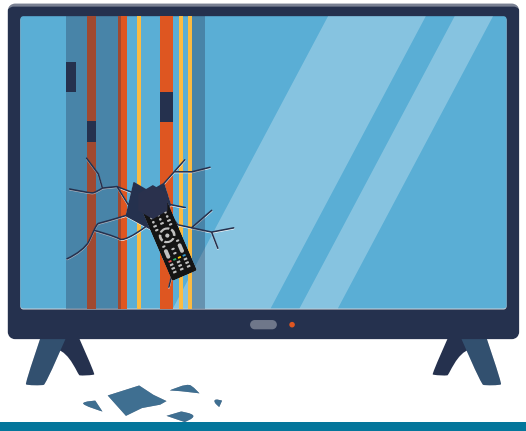


Les tests fonctionnels permettent de vérifier que les fonctionnalités d'un logiciel répondent aux exigences définies, en se basant sur des spécifications métier et des attentes utilisateur.

Contrairement aux tests non fonctionnels, que nous avons vu lors du numéro précédent, qui évaluent des aspects comme la performance ou la sécurité, les tests fonctionnels s'intéressent exclusivement au comportement du système.

🔹 Tests fonctionnels : Une télécommande avec des boutons testés un par un pour vérifier s'ils fonctionnent.

🔹 Tests non fonctionnels : Tester si la télécommande résiste aux chocs, si elle fonctionne sous différentes températures, si elle consomme peu d'énergie.



Pourquoi tester les fonctionnalités de votre logiciel ?

Lorsqu'une application est développée, elle est censée répondre à des besoins précis, comme par exemple :

🔹 Un site e-commerce doit permettre aux utilisateurs d'ajouter des articles à leur panier.

🔹 Une application bancaire doit permettre de consulter son solde et de réaliser des virements.

🔹 Un logiciel de gestion des congés doit s'assurer que seules les demandes valides sont acceptées.

Les objectifs sont donc :

🔹 Vérifier que chaque fonction du logiciel fonctionne conformément aux exigences.

🔹 Détecter des écarts entre les résultats attendus et obtenus.

🔹 Assurer une qualité fonctionnelle avant la mise en production.

🔹 Faciliter la validation par les utilisateurs finaux.

Pour mieux comprendre la place des tests fonctionnels dans la validation d'un logiciel, comparons-les aux tests non fonctionnels :

Critère	Tests Fonctionnels	Tests Non Fonctionnels
Objectif	Vérifier si le logiciel fonctionne selon les spécifications métier.	Évaluer la qualité du logiciel sur des aspects comme la performance, la sécurité et l'ergonomie.
Question principale	Le système fait-il ce qu'il est censé faire ?	Le système fonctionne-t-il correctement dans des conditions variées ?
Types de tests	Tests unitaires, tests d'intégration, tests système, tests d'acceptation (UAT).	Tests de charge, tests de sécurité, tests d'accessibilité, tests d'ergonomie.
Méthode	Basé sur des entrées et sorties attendues.	Basé sur des contraintes et des critères de qualité (temps de réponse, robustesse, UX, etc.).
Approche	Se concentre sur la logique métier et le comportement attendu du système.	Se concentre sur la robustesse, la vitesse, la fiabilité et la sécurité du système.
Techniques courantes	Tests boîte noire, tests basés sur les exigences.	Tests de montée en charge, tests de pénétration, tests d'accessibilité.
Outils	Selenium, Cypress, Playwright, Robot Framework.	JMeter, K6, OWASP ZAP, Lighthouse.
Moment d'exécution	Tout au long du développement, généralement avant la mise en production.	Souvent après les tests fonctionnels, pour évaluer la qualité globale du produit.
Livrables	Cas de test, rapports de test, résultats des exécutions.	Rapports de performance, logs d'analyse de sécurité, audits UX.

Comment réalise-t-on des tests fonctionnels ?

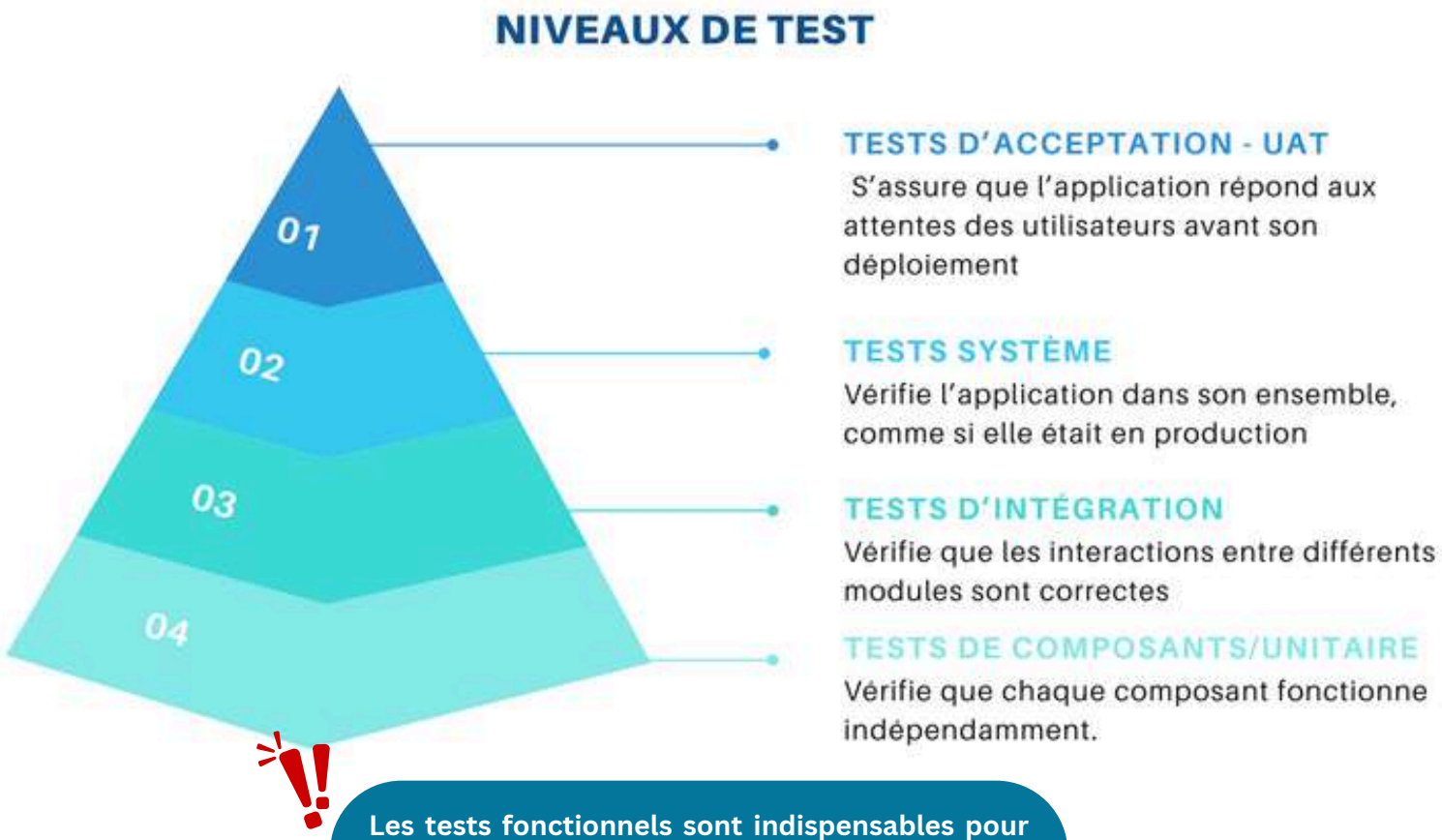
Les tests fonctionnels peuvent être manuels ou automatisés, selon le contexte et les besoins du projet.

🔹 Tests manuels : un testeur exécute les scénarios de test en simulant l'interaction avec l'application (ex. : cliquer sur un bouton, remplir un formulaire).

🔹 Tests automatisés : des scripts sont écrits avec des outils comme Selenium, Cypress, Robot Framework ou Playwright pour exécuter automatiquement les scénarios de test et détecter les anomalies plus rapidement.

Tous les tests ne sont pas réalisés au même niveau. L'ISTQB définit quatre niveaux de validation qui permettent d'assurer une couverture complète du logiciel, du plus petit composant jusqu'à l'application finale.

La pyramide ci-dessous illustre cette hiérarchie et montre comment chaque niveau de test s'intègre dans le processus global.

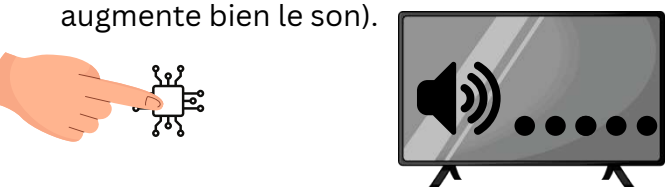


Les tests fonctionnels sont indispensables pour garantir que l'application réponde aux besoins métier et fonctionne correctement. Leur mise en place méthodique, qu'elle soit manuelle ou automatisée, permet de prévenir les erreurs, d'améliorer la satisfaction utilisateur et de limiter les coûts liés aux corrections tardives en production.

Si on reprend l'exemple de la télécommande avec les niveaux de test, nous avons :

Tests unitaires

Vérifier que chaque composant fonctionne isolément en testant individuellement chaque bouton (ex. : vérifier que le bouton "Volume +" augmente bien le son).



Tests d'intégration

Vérifier que plusieurs composants interagissent correctement entre eux en vérifiant que lorsque l'on appuie sur un bouton, un signal est bien envoyé au téléviseur et qu'il réagit comme prévu.

Tests système

Valider le fonctionnement global du produit en simulant des conditions réelles d'utilisation en testant la télécommande dans son ensemble : est-ce que tous les boutons fonctionnent bien ensemble ? Est-ce que la télécommande communique correctement avec différents modèles de téléviseurs compatibles ?

Tests d'acceptation

Vérifier que le produit répond aux attentes des utilisateurs et aux besoins métier avant sa mise sur le marché. L'idée est de faire tester la télécommande par des utilisateurs réels pour s'assurer qu'elle est intuitive, que les boutons sont bien placés et qu'elle fonctionne bien dans différentes situations (ex. : utilisation dans une pièce sombre, à distance, avec des piles faibles).

Pourquoi tester à ces différents niveaux ?

Tester à plusieurs niveaux permet d'identifier les défauts le plus tôt possible, ce qui réduit les coûts de correction et améliore la qualité du produit final.

Chaque niveau joue un rôle dans la validation du système :

- Les tests unitaires et d'intégration détectent les erreurs techniques et les incohérences entre les modules.
- Les tests système et d'acceptation s'assurent que l'ensemble fonctionne comme attendu et satisfait les utilisateurs.

Les tests fonctionnels sont indispensables pour garantir que l'application réponde aux besoins métier et fonctionne correctement. Leur mise en place méthodique, qu'elle soit manuelle ou automatisée, permet de prévenir les erreurs, d'améliorer la satisfaction utilisateur et de limiter les coûts liés aux corrections tardives en production.

LE CYCLE DE VIE DES TESTS

Les tests ne se limitent pas à l'exécution de scénarios pour détecter des anomalies. Ils suivent un cycle structuré qui permet de garantir une validation progressive du logiciel à chaque étape du développement :

Planification et contrôle des tests

Avant de commencer à tester, il faut définir une stratégie adaptée :

- Identification des exigences fonctionnelles à tester.
- Estimation des ressources nécessaires, des délais et des outils.
- Ajustements en fonction de l'avancement du projet.

Analyse et conception des tests

Une fois la stratégie définie, on passe à la conception des cas de test :

- Étude des spécifications pour identifier les scénarios à couvrir.
- Définition des conditions de test.
- Élaboration des cas de test avec entrées, actions et résultats attendus.

Implémentation et exécution des tests

Cette étape consiste à mettre en place et à exécuter les tests :

- Rédaction des scripts de test (manuels ou automatisés).
- Préparation de l'environnement de test.
- Exécution et enregistrement des résultats.

Évaluation des critères de sortie et rapport de test

À la fin des tests, il faut analyser les résultats pour prendre des décisions :

- Comparaison entre les résultats obtenus et les résultats attendus.
- Détection des anomalies et rédaction des rapports de bugs.
- Évaluation de l'atteinte des objectifs définis au départ.

Clôture des tests

Une fois la phase de test terminée, il est important de capitaliser sur l'expérience acquise :

- Archivage des cas de test et des résultats.
- Rétrospective sur le processus et identification des axes d'amélioration.

Planification et contrôle des tests

Définition de la stratégie de test.
Identification des exigences fonctionnelles à tester.
Estimation des ressources, délais et outils nécessaires.

Analyse et conception des tests

Étude des spécifications pour identifier les cas de test.
Définition des conditions de test
Conception des cas de test avec entrées, actions et résultats attendus.

Implémentation et exécution des tests

Rédaction des scripts de test manuels ou automatisés.
Préparation de l'environnement de test.
Exécution des tests et enregistrement des résultats.

Évaluation des critères de sortie et rapport de test

Comparaison des résultats obtenus avec les résultats attendus.
Identification des anomalies et création de rapports de bugs.
Évaluation si les objectifs de test ont été atteints.

Clôture des tests

Archivage des cas de test et des résultats.
Rétrospective et amélioration continue du processus de test.

QA VS QC VS TESTEUR

Démêlons les rôles

Dans l'univers de la qualité logicielle, les termes QA (Quality Assurance), QC (Quality Control) et testeur sont souvent mélangés, alors qu'ils recouvrent des réalités différentes.

Comprendre ces distinctions est un bon point de départ pour structurer votre organisation de test efficace et cohérente.

QA – Quality Assurance

La Quality Assurance se situe à un niveau stratégique et préventif. Elle ne se limite pas aux tests. Elle vise à mettre en place des processus, des normes et des bonnes pratiques qui garantiront que les produits livrés seront de qualité.

Elle s'intéresse à la manière dont on fabrique le logiciel, et non seulement à son résultat.

L'objectif est de réduire les défauts en amont, dès les premières phases du cycle de vie : spécifications, conception, développement.

Quelques exemples de missions en QA :

- Définir une stratégie qualité.
- Mettre en place une démarche de revue (ex : revue de code, revue des spécifications).
- Choisir un modèle de maturité (ex : TMMi, CMMI).
- Suivre des indicateurs qualité (KPIs, taux de couverture, taux de fuite...).
- Former et accompagner les équipes à la qualité logicielle.

QC – Quality Control

La Quality Control, elle, est plus réactive que proactive. Elle s'intéresse au produit déjà fabriqué et consiste à vérifier qu'il respecte les exigences. On parle ici de vérification (compliance) et validation (conformité fonctionnelle).

Les activités de QC sont donc plus opérationnelles. Elles incluent :

- L'exécution de cas de tests (manuels ou automatisés).
- La gestion des anomalies.
- La validation des livraisons.
- Les tests de régression.

Testeur

Le testeur (ou ingénieur QA/Test) est souvent l'acteur qui opère les activités de QC (et parfois une partie de la QA selon la maturité de l'organisation).

Il conçoit, exécute et automatise des tests. Mais son rôle ne se limite pas à « chercher des bugs » :

- Il participe aux ateliers 3 Amigos.
- Il challenge les spécifications.
- Il rédige des cas de test à valeur ajoutée.
- Il peut mettre en place un cadre d'automatisation, voire même contribuer à la stratégie de tests globale.

TESTS MANUELS VS AUTOMATISES

Le débat entre tests manuels et tests automatisés anime nos équipes depuis des années.

Certains prônent l'automatisation comme une solution miracle, tandis que d'autres insistent sur la nécessité de l'intervention humaine. Plutôt que de nous opposer, il est préférable d'explorer leur complémentarité pour optimiser les stratégies de test.

Le test manuel : la flexibilité et l'intuition humaine

Le test manuel repose sur l'exécution directe de scénarios par un testeur. Cette approche est particulièrement adaptée aux explorations et à la vérification d'aspects complexes difficilement automatisables.

Avantages :

- Adapté aux tests exploratoires et aux validations UX/UI.
- Plus flexible pour les scénarios en constante évolution.
- Permet de détecter des problèmes inattendus.

Inconvénients :

- Temps d'exécution long.
- Risque d'erreurs humaines.
- Peu efficace pour les tests répétitifs.

Le test automatisé : rapidité et répétabilité

L'automatisation des tests repose sur des scripts et des outils (Selenium, Cypress, Playwright, etc.) permettant d'exécuter des scénarios en continu.

Avantages :

- Exécution rapide et répétitive.
- Idéal pour les tests de régression
- Réduction des erreurs humaines.

Inconvénients :

- Coût initial élevé.
- Maintenance nécessaire des scripts.
- Peu adapté aux tests UX et à l'exploration.

Critère	Test Manuel	Test Automatisé
Effort initial	Faible	Élevé
Temps d’exécution	Long	Rapide
Coût long terme	Élevé	Amorti sur la durée
Scénarios complexes	Oui	Non
Maintenance	Aucune	Nécessaire
Tests récurrents	Peu adapté	Très adapté
Tests exploratoires	Oui	Non

Une approche hybride : le meilleur des deux mondes

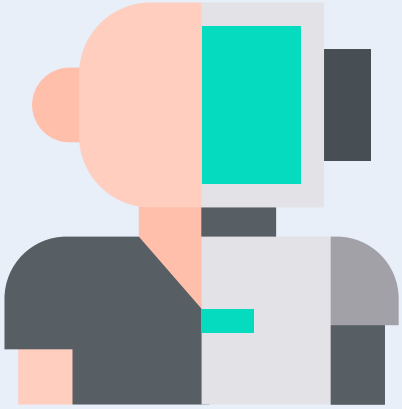
Une stratégie efficace combine intelligemment les deux approches. Voici comment les utiliser judicieusement :

Tests à automatiser :

- Tests de régression récurrents.
- Tests de performance et de charge.
- Tests cross-plateformes.

Tests à privilégier en manuel :

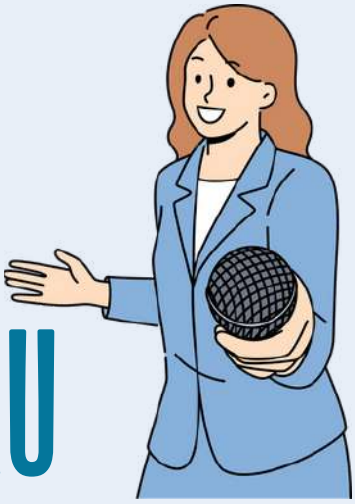
- Tests exploratoires et UX.
- Validation de nouvelles fonctionnalités.
- Scénarios complexes difficiles à coder.



L'automatisation des tests est un formidable levier pour améliorer l'efficacité des tests, mais elle ne remplace pas l'intuition et l'analyse humaine. La meilleure approche repose sur une stratégie hybride, combinant automatisation et test manuel en fonction des besoins du projet.

L'objectif n'est pas de choisir entre les deux, mais de les utiliser intelligemment pour garantir une qualité optimale.

INTERVIEW D'ISABELLE HOAREAU



De la QA au rôle de Chef de Projet, un parcours passionné par la Qualité

Pour commencer, pouvez-vous nous parler de votre parcours ?

Bien sûr ! J'ai plus de 25 ans d'expérience dans le domaine de l'informatique, avec un parcours plutôt atypique. J'ai commencé sans diplôme. Mon parcours s'est construit sur de la formations continue, de la certifications, de l'expérience et beaucoup de recherche personnel.

J'ai commencé avec une formation généraliste proposée par la chambre de commerce de l'Yonne et j'ai acquis mes compétences en société de service avec de grands comptes comme EDF.
N'étant pas développeur expérimenté, il m'était confié les tâches de tests et de validation techniques et métier.

En 2007 j'ai suivi une formation de Consultant qualité Logiciel et en 2023, une formation pour un titre RNCP de niveau bac+5 de chef de projet me permettant d'avoir un niveau de diplôme en cohérence avec mes 25 ans d'expérience.

En parallèle de votre expérience, vous avez aussi passé plusieurs certifications. Pouvez-vous nous en dire plus ?

Oui, j'ai passé plusieurs certifications qui m'ont énormément apporté. J'ai passé 2 certifications ISTQB (Fondation et tests manager), et l'**IREB**, qui est plus axée sur l'ingénierie des exigences. Ces formations ont structuré mes connaissances et m'ont permis de mieux comprendre les bonnes pratiques de mon métier. Elles m'ont aussi aidée à formaliser mes savoirs et à mieux transmettre mes compétences aux équipes avec lesquelles je travaille.

La qualité n'est vue que sous l'angle des tests en fin de cycle, alors qu'en réalité, elle commence dès les premières réflexions lors du lancement du projet. Les aspects qualitatifs, les conditions de validations, les KPI permettant de s'assurer d'un niveau de fiabilité adapté doivent être pensés dès de début.. Plus on détecte tôt les ambiguïtés et les incohérences, plus on gagne du temps et surtout de l'argent ! La qualité logicielle ce n'est pas que des tests .

C'est un investissement et non une charge.

Justement, pourquoi cet intérêt particulier pour l'expression des besoins ?

Parce que c'est un élément clé dans la réussite d'un projet. Une mauvaise expression des besoins entraîne des erreurs dès le début, qui se transforment ensuite en bugs coûteux à corriger. Identifier ces problèmes en amont permet d'éviter des coûts parfois astronomiques.

En tant que testeurs, nous nous devons de : comprendre, challenger et clarifier les attentes pour limiter les erreurs. Malheureusement, encore trop souvent, la qualité n'est vue que sous l'angle des tests en fin de cycle, alors qu'en réalité, elle commence dès la conception du produit. Plus on détecte tôt les ambiguïtés et les incohérences, plus on gagne du temps (et de l'argent !) par la suite.

Vous dites que la qualité peut-être difficile à mettre en place et à faire comprendre. Pourquoi ?

Oui, et c'est un vrai défi. Même lorsque l'on occupe des postes avec plus de responsabilités, il y a des blocages. Parfois, c'est une question de politique interne, d'autres fois, c'est une résistance au changement. Beaucoup d'équipes sont encore focalisées sur la rapidité de livraison et perçoivent la qualité comme un frein, alors qu'en réalité, elle est un accélérateur sur le long terme.

Il faut sans cesse expliquer, démontrer la valeur ajoutée de la qualité et s'adapter aux différentes personnes impliquées. Par exemple, un développeur va vouloir des retours rapides sur son code, alors qu'un décideur aura besoin de chiffres concrets pour évaluer les coûts des démarches de validation à venir.

Un aspect de notre métier moins connu et pourtant très prégnant, est l'aspect de la traduction : Nous sommes des traducteurs entre un langage technique et un langage métiers. En ça nous sommes un atout pour les projets.

Les testeurs doivent donc aussi avoir des compétences en communication ?

Exactement ! Nous sommes avant tout des messagers. Cependant, étant donné que la vision actuelle des tests est avant tournée principalement vers le bug, nos messages contiennent principalement des "problèmes", ce qui nous qualifie principalement de "mauvais messager". C'est un des problèmes du testeur, ce qui peut rendre l'exercice du métier difficile. Si à terme, nous pensons la qualité selon sa définition première, nos messages peuvent être remplis de bonnes nouvelles

Tester ne se résume pas à détecter des bugs, c'est aussi savoir communiquer efficacement avec les équipes. Les développeurs, les chefs de projet, les PO, les métiers... tout le monde a une perception différente du rôle du testeur.

Parfois, nous devons faire face à l'ego des personnes qui ont conçu le produit, et il faut savoir amener les choses intelligemment pour ne pas braquer les interlocuteurs. Un testeur doit être diplomate, pédagogue et capable d'argumenter ses retours avec des faits.
Être testeur, c'est porter plusieurs casquettes

- Technique : comprendre le produit et son architecture.
- Analytique : anticiper les scénarios critiques.
- Communicante : convaincre, expliquer et embarquer les équipes dans une démarche qualité.

En dehors de votre travail, vous continuez à vous former et à partager votre savoir. Comment faites-vous ?

Bien que je sois moins active dans les communautés de testeurs depuis quelque temps, en 2022 et 2023 j'ai eu l'opportunité d'effectuer des recherches pour un projet d'outil de qualification et de suivi de qualité.

Cette réflexion m'a amené sur des terrains dépassant l'informatique et à partir desquels il est possible de trouver des pistes de solutions et d'amélioration de nos méthodologies de qualifications des SI. (les notions de système, d'information, émergence, de chaos etc...)

Je suis passionnée par la qualité, donc je consacre une partie de mon temps personnel à me former, lire et approfondir mes connaissances. J'avais d'ailleurs écrit trois articles sur La Taverne du Testeur pour partager mes réflexions et mon expérience.

C'est un plaisir d'échanger avec la communauté, de confronter ses idées et de découvrir de nouvelles approches. La qualité logicielle est un domaine en constante évolution, et je trouve important de contribuer à sa diffusion.

Avez-vous une recommandation de lecture pour ceux qui veulent mieux comprendre l'expression des besoins ?

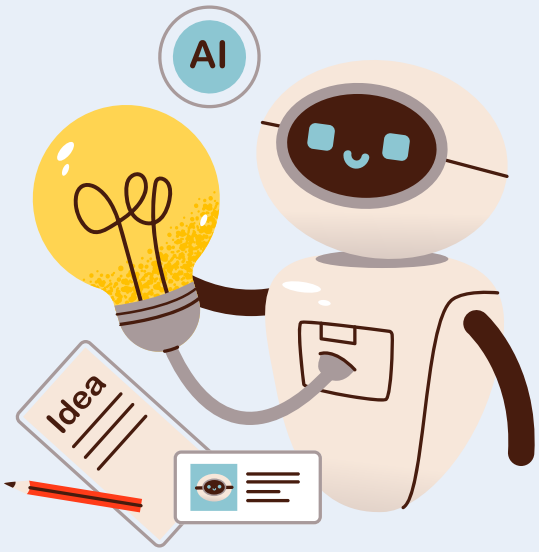
Oui, je recommande vivement le livre d'Yves Constantinidis sur l'expression des besoins pour le SI. Il est très bien structuré et donne des clés pour améliorer cette phase critique des projets.

L'avenir des métiers du test et de la qualité vous semble-t-il en évolution ?

Oui, et c'est passionnant ! Une simple sous-activité est devenu un véritable métier en 20 ans donc, malgré toutes les difficultés, il n'y a aucune raison pour que ce métier ne continue pas d'évoluer et ne trouve pas ses lettres de noblesse.

Les domaine de la sécurité des équipements et de cybersécurité peuvent être des sources d'inspiration intéressantes pour améliorer nos méthodes et nos processus de tests en informatique de gestion. Par exemple, la norme EBIOS Risk Manager permet de modéliser les menaces intentionnelles, de visualiser les scénarios d'attaque les plus plausibles, et de déterminer quelles fonctions ou composants sont les plus exposés. En croisant les résultats d'EBIOS RM avec les critères métiers, on obtient une vision claire des zones à tester en priorité — à la fois du point de vue de la sécurité et de la continuité de service.

Une autre tendance est également de gérer la complexité de nos systèmes d'information qui les rend "chaotiques" (dans le sens théorique du terme, c'est-à-dire dont le comportement n'est plus exactement prévisible), du fait de la multiplication des sous-systèmes qui les composent.



Une dernière tendance à citer, c'est l'intelligence artificielle. L'IA ne doit pas être vue que comme un outil d'aide au tests et à la qualification mais aussi et surtout comme un sujet à part entière à tester et valider. De nouvelles approches, méthodes et critères d'évaluation doivent être réfléchis autant pour les projets qui en contiennent. Il est à noter qu'il y a une explosion de l'utilisation de l'IA en développement.

Merci pour cet échange enrichissant ! Un dernier mot pour nos lecteurs ?

Si une bonne spécification vaut mille tests, le must est encore de trouver les bugs avant qu'ils ne soient codés !

Pour finir, selon vous, que faudrait-il améliorer pour réduire les bugs dans les projets ?

Des améliorations de l'expression des besoins est un premier volet mais avant tout, une prise de conscience par nos décideurs de l'intérêt d'une réflexion et d'investissement sur la qualité du SI en devenir et ceux dès le début des projets. Ces réflexions doivent être portées sur tous les aspects : Techniques, méthodologiques, métiers, organisationnels... Pour ce faire, l'appel de consultant expert me semble primordial.

Si les attentes sont bien définies, comprises et partagées dès le départ, on éviterait une grande partie des erreurs et des incompréhensions qui mènent aux bugs. Cela demande un effort collectif, mais c'est une approche qui bénéficie à tout le monde, y compris aux finances.

L'IA VA NOUS REMPLACER ?

L'intelligence artificielle s'impose progressivement dans tout les domaines, suscitant autant d'enthousiasme que de scepticisme. Les néophytes me demandent si nous risquons de perdre notre métier, remplacé par l'IA. Honnêtement, j'ai eu un doute et même de très gros doutes. Nous avons vu l'industrialisation et la mondialisation qui ont fait perdre de nombreux travails, alors pourquoi pas nous?

Entre promesses d'automatisation avancée et limites technologiques, où en est réellement l'IA dans le domaine du test ? Est-elle une véritable révolution ou un simple buzzword ?

L'IA dans le test logiciel : de quoi parle-t-on ?
L'IA appliquée aux tests logiciels repose sur plusieurs technologies :

- Le machine learning (ML) : apprentissage automatique à partir de données pour détecter des anomalies ou générer des cas de test intelligents.

- Le traitement du langage naturel (NLP) : analyse et compréhension de spécifications pour automatiser la création de tests.

- Les algorithmes prédictifs : identification des zones de risque les plus probables pour prioriser les tests.

L'objectif ? Réduire l'effort manuel, augmenter la couverture des tests et accélérer les cycles de développement.

Automatiser la génération des tests avec l'IA

L'un des principaux apports de l'IA est la capacité à générer automatiquement des cas de test. Certains outils exploitent l'analyse des spécifications, du code ou des interactions des utilisateurs pour créer des scénarios de test pertinents.

Exemples d'outils :

- Testim : Génère et optimise des tests automatisés en apprenant des interactions humaines.

- Functionize : Crée des tests intelligents grâce à l'IA et les adapte automatiquement aux évolutions de l'application.

- Applitools : Détecte les régressions visuelles en comparant des captures d'écran avec une précision avancée.

Les bénéfices :

- Réduction du temps de conception des tests

- Amélioration de la couverture fonctionnelle

- Adaptabilité aux évolutions du code.

Les limites :

- La qualité dépend des données d'apprentissage.

- Risque de faux positifs ou faux négatifs.

- Besoin d'une supervision humaine pour affiner les résultats.

Pour créer des tests autonomes?

L'IA permet aussi d'optimiser l'exécution des tests en fonction de leur pertinence et des modifications du code.

Cas d'usage :

- Test impact analysis : Identifier les tests à exécuter en priorité après une modification.
- Auto-réparation des tests : Ajustement dynamique des scripts de test face aux changements d'interface (ex. : un bouton renommé ou déplacé).
- Détection intelligente des anomalies : Analyse des logs et des résultats de tests pour repérer des comportements suspects.

Exemples d'outils :

- Mabl : Exécute des tests autonomes et auto-corrige les scénarios en fonction des évolutions.
- Selenium avec AI plugins : Utilisation de l'IA pour stabiliser les tests et réduire les échecs liés aux changements d'interface.

Les bénéfices :

- Réduction du nombre de tests inutiles.
- Meilleure stabilité des tests automatisés.
- Détection proactive des bugs.

Les limites :

- L'IA peut manquer de contexte métier.
- Les tests auto-réparés peuvent masquer des problèmes réels.

Pour détecter les anomalies ?

L'IA ne se contente pas d'exécuter des tests, elle peut aider aussi à analyser les résultats et à identifier les anomalies plus efficacement que des méthodes classiques.

Approches utilisées :

- Détection des tendances anormales : repérer des dérives de performance ou des schémas inhabituels.
- Comparaison visuelle avancée : détecter des changements graphiques subtils.
- Analyse des logs et des erreurs : corrélation entre les logs et les échecs de tests pour une analyse plus rapide.

Exemples d'outils :

- Dynatrace : Utilise l'IA pour surveiller les performances et détecter des anomalies en temps réel.
- New Relic AI : Analyse les logs et événements pour identifier automatiquement les causes racines des incidents.

Les bénéfices :

- Réduction du temps d'analyse des tests.
- Identification proactive des problèmes.
- Automatisation des rapports de test.

Les limites :

- Peut générer des alertes inutiles (bruit dans les résultats).
- Nécessite un bon calibrage pour éviter des analyses erronées.

L'IA ne remplace pas les testeurs, elle nous assiste !

LE SHIFT-LEFT TESTING

Avec l'accélération des cycles de développement et l'adoption des méthodes Agile et DevOps, tester tardivement n'est plus une option.

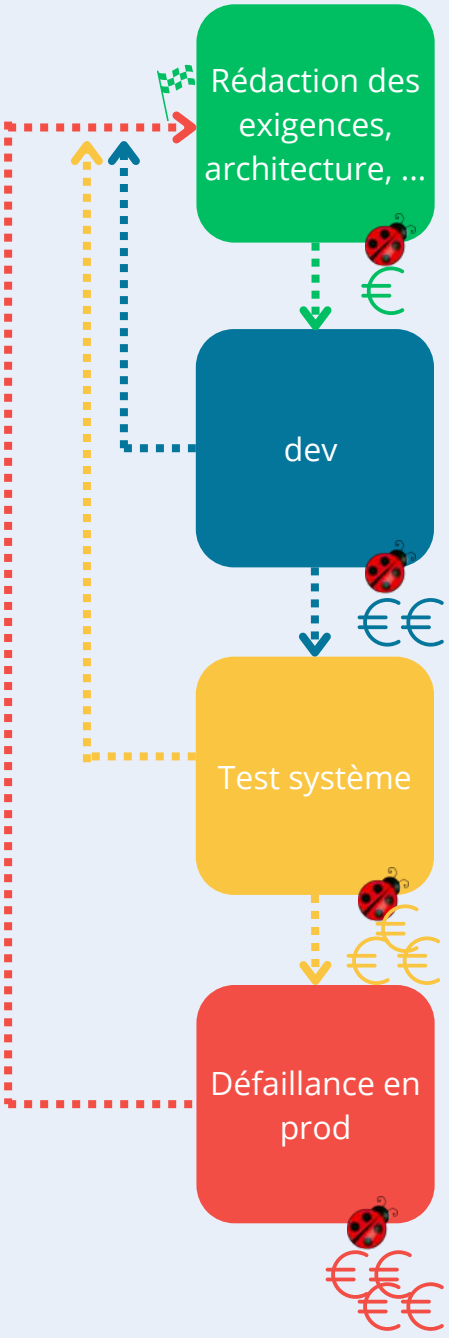
Le Shift-Left Testing, ou "déplacement à gauche", est une approche qui consiste à tester plus tôt dans le cycle de développement afin d'identifier et corriger les anomalies dès leur apparition. Cette stratégie permet d'améliorer la qualité logicielle tout en réduisant les coûts et les délais de correction.

Pourquoi adopter le Shift-Left Testing ?

Traditionnellement, les tests étaient effectués en fin de cycle de développement, ce qui augmentait les risques de découverte tardive de bugs et de retards dans les livraisons. Le Shift-Left Testing propose une approche différente :

- Détection précoce des défauts : plus un bug est identifié tôt, moins il coûte à corriger.
- Amélioration de la collaboration entre développeurs et testeurs dès les premières phases du projet.
- Accélération du cycle de livraison : moins d'interruptions en fin de projet, donc un déploiement plus fluide.
- Augmentation de la confiance dans les builds grâce à des feedbacks continus.

Ci-dessous un schéma des couts des “bugs” selon le moment où il est découvert : les “bugs” corrigés en phase de développement coûtent en moyenne 10 fois moins cher que ceux détectés en production.



Comment implémenter le Shift-Left Testing ?

Isabelle Hoareau se dit d'ailleurs qu'il est tout à fait envisageable de commencer les tests le plus tôt possible, dès l'étape des spécifications métier. On pourrait presque parler ici de Shift-Shift-Left Testing !

En effet, les spécifications contiennent souvent des incohérences, des oublis ou des erreurs. Leur analyse approfondie permet déjà d'identifier un grand nombre de défaillances potentielles avant même qu'elles ne se propagent dans les phases suivantes du projet.

Il est naturel que de telles imperfections existent à ce stade : les interlocuteurs métier ne sont pas des spécialistes du développement et ne peuvent pas toujours anticiper les implications techniques de leurs besoins.

C'est précisément là que le rôle des testeurs prend toute sa valeur. Grâce à notre double compétence – fonctionnelle et technique – nous jouons souvent un rôle de pont entre les équipes métier et les équipes de développement. Cette capacité à dialoguer avec les deux mondes nous permet d'identifier rapidement les points de friction et d'éviter des dérives coûteuses plus tard dans le cycle de vie du produit.

Plusieurs bonnes pratiques permettent de réussir cette transition vers un test plus tôt :

Automatiser les tests unitaires et d'intégration

L'un des piliers du Shift-Left Testing est l'automatisation. Il faut mettre en place des tests unitaires et d'intégration exécutés systématiquement à chaque commit.

Outils recommandés :

- JUnit, NUnit, TestNG : Pour les tests unitaires en Java et .NET.
- Cypress, Selenium, Playwright : Pour les tests d'intégration et fonctionnels automatisés
- Jest, Mocha : Pour les tests unitaires en JavaScript.

Mettre en place des tests statiques

Les tests statiques, tels que l'analyse de code et la revue de code automatisée, permettent d'identifier des problèmes sans exécuter le programme.

Outils recommandés :

- SonarQube : Analyse statique de code pour détecter les vulnérabilités et les erreurs.
- ESLint, Prettier : Pour l'analyse du code JavaScript.
- Pylint, Flake8 : Pour le code Python.

Adopter le Test-Driven Development (TDD)

Le TDD consiste à écrire les tests avant le code source, ce qui oblige à définir les exigences et les comportements attendus avant même l'implémentation.

Bénéfices du TDD :

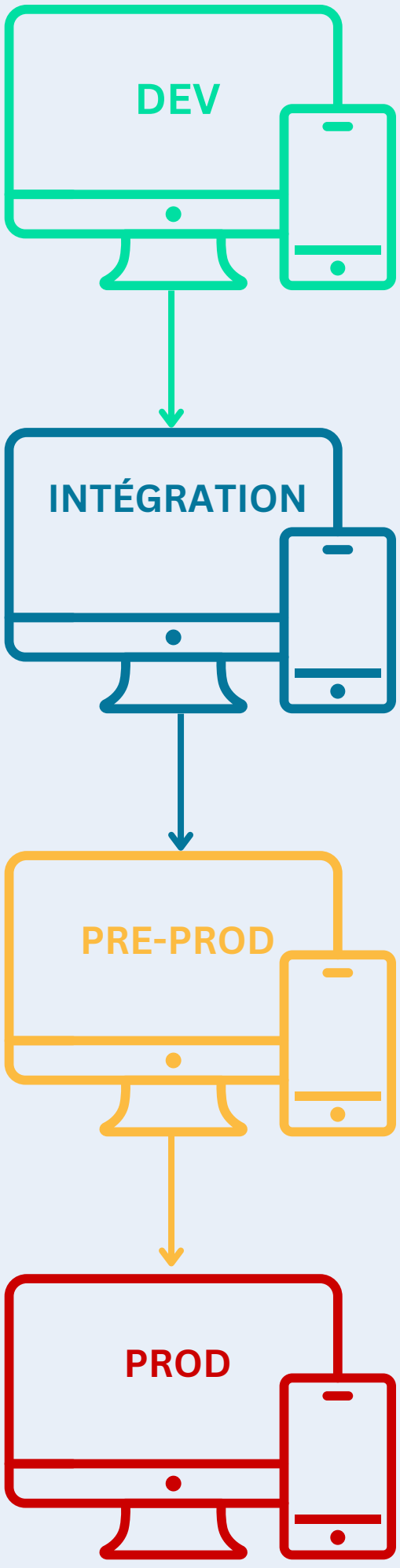
- Un code plus robuste et mieux conçu.
- Une meilleure compréhension des besoins fonctionnels.
- Moins de défauts en phase de validation.

Exécuter les tests tôt dès l'environnement de développement

Attendre la pré-production pour tester, par exemple, la performance et la sécurité expose à des décisions tardives et coûteuses. Tester ces aspects dès le développement ou 'intégration continue permet d'anticiper et d'atténuer les risques. Vous pouvez voir l'illustration ci-contre pour comprendre les environnements.

Outils recommandés :

- JMeter, k6 : Pour les tests de charge et de performance.
- OWASP ZAP, Burp Suite : Pour les tests de sécurité.



GHERKIN

Gherkin est un langage structuré en texte clair qui permet également d'être utilisé pour créer des user story.

Certaines équipes l'utilisent dans les approches BDD (Behavior-Driven Development) pour décrire les comportements attendus d'une application sous forme de scénarios compréhensibles.
A tord ou à raison? Le BDD est une notion plus complexe que l'écriture d'un langage.

Pourquoi utiliser Gherkin ?

L'utilisation de Gherkin présente plusieurs avantages :

- écrit en langage naturel, il est compréhensible par tous (développeurs, testeurs, chefs de projet, business analysts, etc.).

- intégré avec des outils comme :
  - Cucumber (Java, JavaScript, Ruby, Python...)
  - SpecFlow (C#)
  - Behave (Python)
  - Behat (PHP)

- encourage une meilleure communication entre les équipes métier et technique.

- les scénarios écrits en Gherkin servent de documentation

- et également de base pour vos tests automatisés car vous pouvez utiliser cucumber pour faire le lien entre votre code et les spécifications

Syntaxe de base de Gherkin

Gherkin repose sur une structure simple composée de mots-clés en anglais (ou dans d'autres langues supportées) :

- **Feature** : décrit la fonctionnalité testée

```
Scenario: Connexion réussie avec des identifiants valides

Given l'utilisateur est sur la page de connexion

When il saisit son identifiant "user123" et son mot de passe "password123"

And il clique sur le bouton de connexion

Then il est redirigé vers la page d'accueil

And un message de bienvenue s'affiche
```

Les termes utilisés sont :

- **Scenario** : décrit un cas de test spécifique

- **Given - When - Then** : structure du scénario
  - Given : Contexte initial (prérequis)
  - When : Action effectuée
  - Then : Résultat attendu

- **Background** : contexte commun à plusieurs scénarios

- **Outline et Examples** : paramétrisation des scénarios

Pour éviter la duplication, Scenario Outline permet de tester plusieurs cas en utilisant des valeurs différentes.
Vous pouvez voir le code ci-dessous :

```
Scenario Outline: Connexion avec différents utilisateurs
  Given l'utilisateur est sur la page de connexion
  When il saisit son identifiant "<identifiant>"
            et son mot de passe "<mot de passe>"
  And il clique sur le bouton de connexion
  Then il voit le message "<message>"

Examples:
  | identifiant | mot de passe  | message                 |
  | user123     | password123   | Connexion réussie       |
  | userXYZ     | wrongpass     | Identifiants incorrects |
```

Pour quel contexte Gherkin est pertinent ?

- Projets agiles, avec des équipes pluridisciplinaires.
- Produits où la valeur métier prime sur la technique.
- Projets de longue durée où la documentation vivante est un vrai enjeu.
- Environnements où la collaboration entre PO, testeurs et développeurs est déjà en place ou souhaitée.

Les bonnes pratiques pour écrire des scénarios Gherkin sont :

- éviter le jargon technique
- éviter les scénarios trop longs : un scénario **doit tester un seul** comportement.
- Gherkin décrit ce qui doit être testé, pas comment.
- Réutiliser les étapes

Le Gherkin représente une véritable valeur ajoutée lorsqu'il est utilisé à bon escient : il structure la communication, clarifie les attentes métier et facilite l'automatisation des tests.

Mais mal maîtrisé, il peut rapidement devenir contre-productif : scénarios trop nombreux, redondants, trop techniques ou détachés des besoins réels, ce qui alourdit inutilement le projet.

Lorsqu'on utilise un outil comme Jira, l'intégration de Gherkin peut apporter une vraie cohérence dans la gestion des exigences et des tests, notamment via des plugins comme Xray ou Zephyr.

Cependant, si les scénarios Gherkin sont rédigés sans collaboration, sans revue, ou en étant uniquement dictés par une logique technique, on court le risque de transformer ce langage métier en une usine à gaz illisible, où les cas de test perdent leur fonction première : servir la compréhension et la validation du besoin.

COMMENT RECUPERER LES ELEMENTS ?

Lorsque vous automatisez des tests web, il faut identifier correctement les éléments de la page.

Si vous utilisez un mauvais identifiant, vous allez devoir les mettre à jour à chaque mise à jour de la librairie que les développeurs utilisent ou à chaque changement de CSS ou alors...

Chaque outil utilise des sélecteurs (CSS, XPath, ID, etc.) pour interagir avec les éléments HTML. Voici comment procéder pour Cypress, Selenium et Robot Framework.

Utilisez les DevTools Chrome pour récupérer les sélecteurs :

- Clic droit sur l'élément > Inspecter

Vous pouvez voir les élements.

Privilégier les sélecteurs stables

- Évitez les sélecteurs basés sur des classes dynamiques (.random-xyz123) qui changent à chaque rafraîchissement de page.

- Utiliser les ID si disponibles : lLes IDs sont uniques et donc plus fiables.

- Éviter les XPath trop complexes comme les XPath très longs (/html/body/div[2]/div/...) sont fragiles aux modifications de l'UI.

Bonne pratique

La bonne pratique est de créer un nouveau attribut comme data-test avec une valeur unique.

Pour cela, vous devez travailler main dans la main avec le développeur afin de modifier le code et ajouter ces attributs.

Pourquoi data-test est la meilleure option ?

- Stable : Contrairement aux classes CSS ou aux IDs générés dynamiquement, un attribut comme data-test ne change pas lors des mises à jour du design.

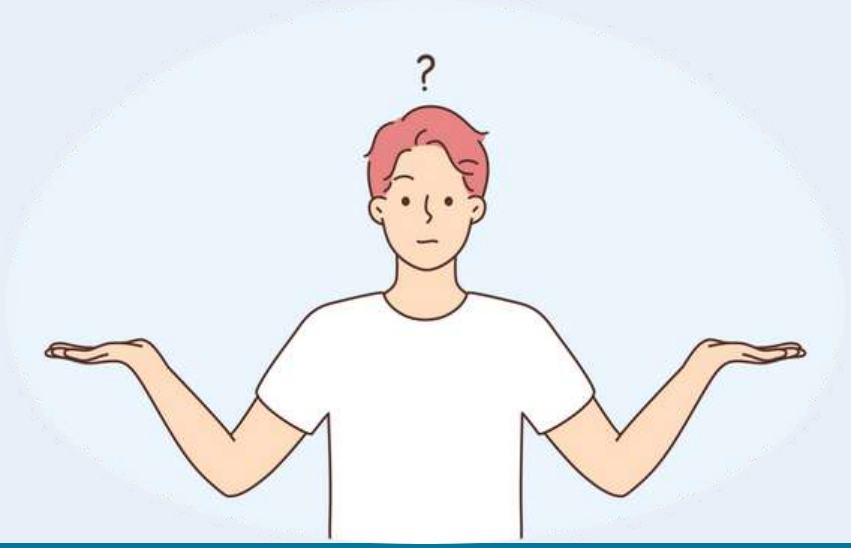
- Lisible : Un testeur comprend immédiatement que data-test="login-button" est fait pour les tests.

- Supporté partout : Compatible avec Cypress, Selenium et Robot Framework.

- Facile à intégrer : Les développeurs peuvent l'ajouter sans impacter le visuel ni la logique métier.

Voici un tableau récapitulatif des méthodes de sélections :

Type de sélecteur	Cypress	Selenium	Robot Framework
ID (id="myId")	✅ cy.get('#myId')	✅ driver.find_element(By.ID, "myId")	✅ Click Element id=myId
Class (class="myClass")	✅ cy.get('.myClass')	✅ driver.find_element(By.CLASS_NAME, "myClass")	✅ Click Element css=.myClass
Attribut data-test (data-test="login-button")	✅ cy.get('[data-test="login-button"]')	✅ driver.find_element(By.CSS_SELECTOR, '[data-test="login-button"]')	✅ Click Element css=[data-test="login-button"]
Attribut name (name="email")	✅ cy.get('[name="email"]')	✅ driver.find_element(By.NAME, "email")	✅ Click Element name=email
Sélection par texte (<button>Connexion</button>)	✅ cy.contains('Connexion')	❌ Non natif (nécessite XPath)	✅ Click Button Connexion
Sélection par lien (Cliquez ici)	✅ cy.get('a[href="page.html"]')	✅ driver.find_element(By.LINK_TEXT, "Cliquez ici")	✅ Click Link Cliquez ici



DOSSIER

OUTILS D'AUTOMATISATION DE TESTS WEB

Critère	Cypress	Selenium	Robot Framework	Playwright
Langages supportés	Javascript, Typescript	Java, Python, C#, Javascript, Ruby	Python	Javascript, Typescript, Python, C#
Facilité d'installation	Très simple	Complexe	Facile	Très simple
Exécution	Directement dans le navigateur	Utilise WebDriver	Utilise Selenium, Appium, ..	Directement dans le navigateur
Support multi-navigateurs	Chrome, Edge, Firefox,	Tous les navigateurs majeurs	Tous les navigateurs via Selenium	Chrome, Edge, Firefox, WebKit
Support mobile	Non supporté	Support via Appium	Support via Appium	Support via Appium
Attente automatique	Oui	Non	Partiellement	Oui
Test API	Oui	Non	Oui	Oui
Gestion des iframes	Limitée	Oui	Oui	Oui
Gestion des onglets	Non supporté	Oui	Oui	Oui
Débogage	très simple	Complexe	Log clair mais interface minimaliste	très simple
Mocking des requêtes HTTP	Oui	Non	Non	Oui
Exécution en parallèle	Expérimental	Oui	Oui	Oui
Idéal pour	Développeurs et testeurs	Projet nécessitant une large compatibilité	Testeurs fonctionnels cherchant un langage simple	Développeurs et testeur full-stack

SAUCE LABS

Sauce Labs est une plateforme cloud spécialisée dans l'exécution de tests automatisés et manuels sur une large gamme d'environnements. Elle permet de tester des applications web et mobiles sur différentes combinaisons de navigateurs, systèmes d'exploitation et appareils physiques, sans nécessiter de configuration locale complexe.

Fonctionnalités clés

- Tests multi-navigateurs et multi-plateformes : Sauce Labs prend en charge tous les navigateurs majeurs (Chrome, Firefox, Edge, Safari, Internet Explorer) et permet de tester sur plusieurs systèmes d'exploitation (Windows, macOS, Linux).
- Support pour plusieurs frameworks de tests : Sauce Labs est compatible avec Selenium, Cypress, Appium, Playwright, Espresso, XCUITest et bien d'autres frameworks.
- Exécution en parallèle des tests : grâce au cloud, il est possible d'exécuter plusieurs tests simultanément, réduisant ainsi le temps d'exécution des suites de tests.
- Rapports et analyses détaillés : Sauce Labs génère des logs, captures d'écran, vidéos et métriques de performance pour faciliter l'analyse des résultats et le débogage.
- Intégration avec les outils DevOps et CI/CD : la plateforme s'intègre avec Jenkins, GitHub Actions, Azure DevOps, GitLab CI/CD, etc., facilitant l'automatisation et le déploiement continu.
- Tests sécurisés : Sauce Labs propose des solutions sécurisées pour tester les applications en préproduction sans exposer les données sensibles.

Les inconvénients de Sauce Labs

- Coût élevé : Sauce Labs est un service premium, et son coût peut être un frein pour les petites entreprises ou startups.
- Dépendance au cloud : les tests étant exécutés sur des machines distantes, des latences peuvent apparaître par rapport à une exécution locale.
- Configuration initiale : bien que Sauce Labs soit puissant, sa configuration peut être plus complexe que des outils locaux comme Cypress.
- Limitations des émulateurs : les tests mobiles sur émulateurs ne sont pas toujours aussi précis que sur des appareils physiques.

Nous allons utiliser ce site de démonstration pour comparer les outils.

Comment utiliser SauceDemo pour l'automatisation ?

- Lancer un test et se connecter avec standard_user.
- Ajouter un produit au panier et valider l'affichage des éléments.
- Tester la déconnexion et vérifier la persistance des sessions.
- Utiliser un compte "buggué" (problem_user) pour voir comment les erreurs sont gérées.

SauceDemo est un site de démonstration fourni par Sauce Labs pour tester et valider les scripts de tests automatisés. Il simule une boutique en ligne avec une interface simple et des fonctionnalités classiques d'un e-commerce.

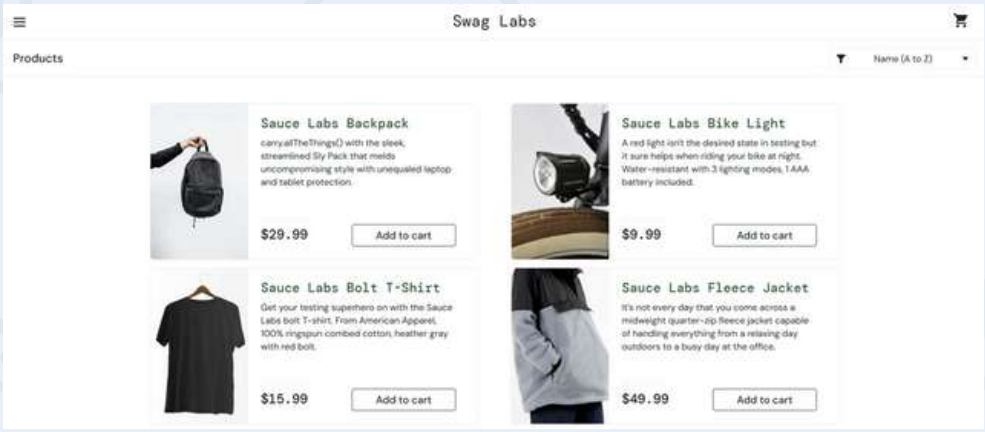
Pourquoi utiliser SauceDemo ?

🔹 Plateforme de test pour l'automatisation : il permet aux testeurs et développeurs d'écrire, exécuter et valider leurs scripts de tests Selenium, Cypress, ou Playwright.

🔹 Scénarios variés : L'application intègre différents types d'utilisateurs avec des permissions et des comportements spécifiques.

🔹 Données dynamiques : Les identifiants de connexion et l'état du panier varient selon l'utilisateur, offrant des scénarios réalistes.

🔹 Erreurs simulées : Certains comptes ont des bugs intégrés pour tester la gestion des erreurs dans les tests.



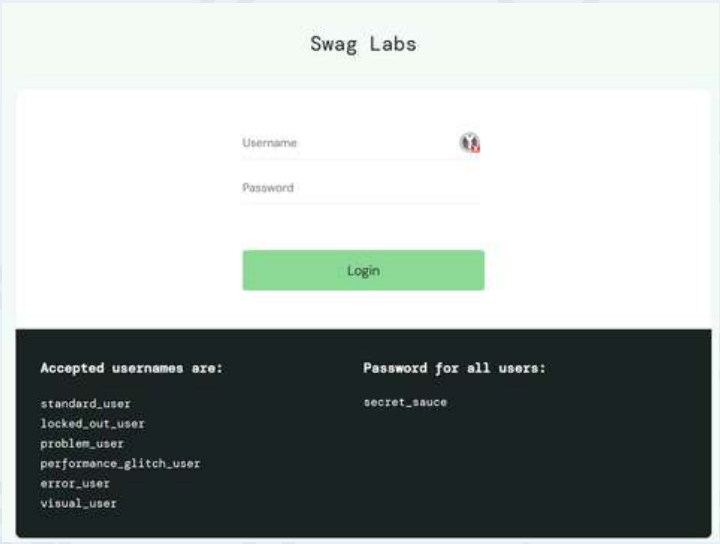
1

Le premier test “Lancer un test et se connecter avec standard_user”. **L’objectif est de vérifier qu'un utilisateur peut se connecter et ajouter un produit au panier.**

- Il faut déjà aller sur la page https://www.saucedemo.com/

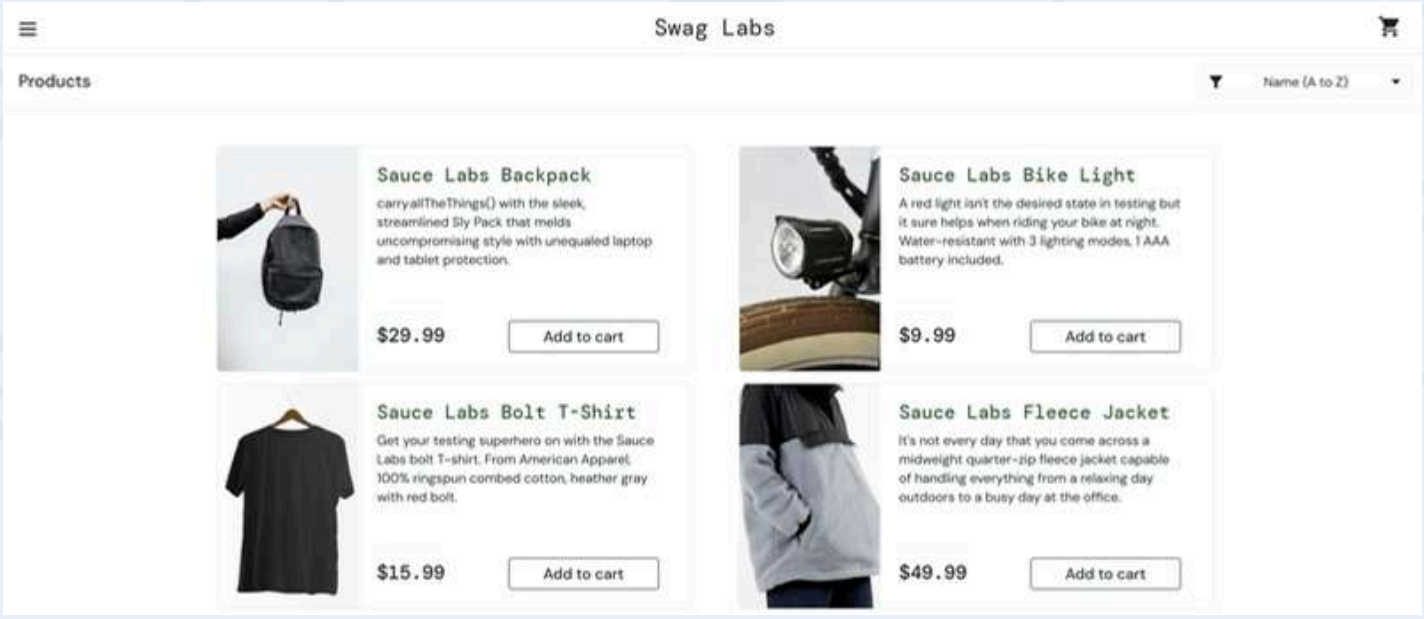
- Puis, il faut se connecter avec un utilisateur valide, choisissons standard_user avec le mot de passe secret_sauce.

Vous pouvez utiliser d’autres utilisateurs, comme vous pouvez le voir dans le bandeau noir ci-dessous :



- Cliquer sur le bouton de connexion.

- Vérifier que l'utilisateur est bien redirigé vers la page des produits.



- Sélectionner le premier produit affiché.

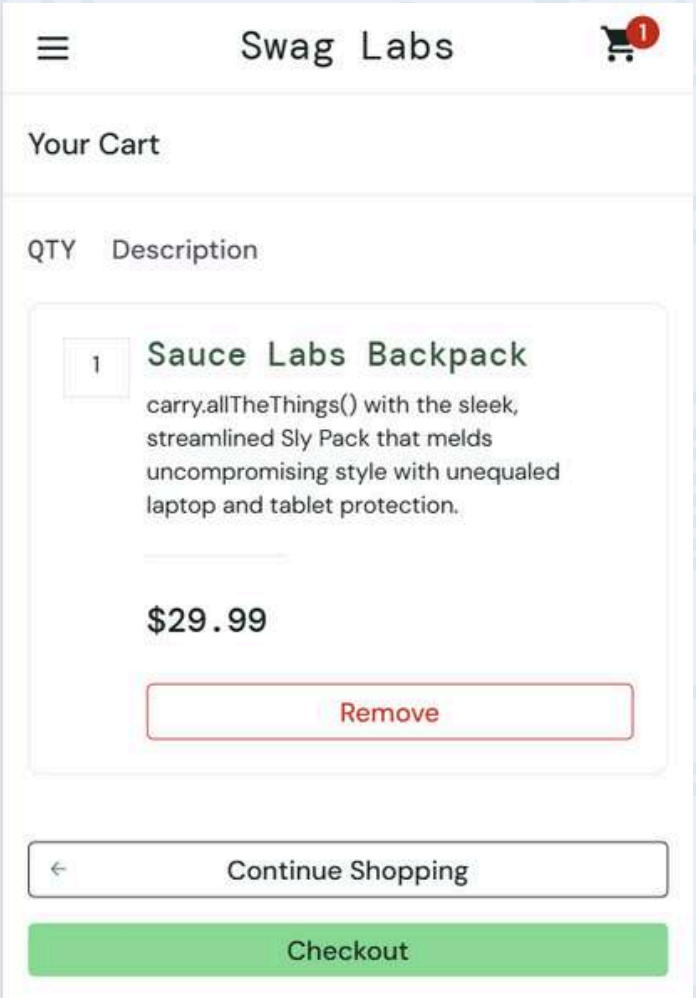
- Cliquer sur le bouton "Ajouter au panier".

- Vérifier que l’icône du panier affiche "1" (indiquant qu'un produit a été ajouté).

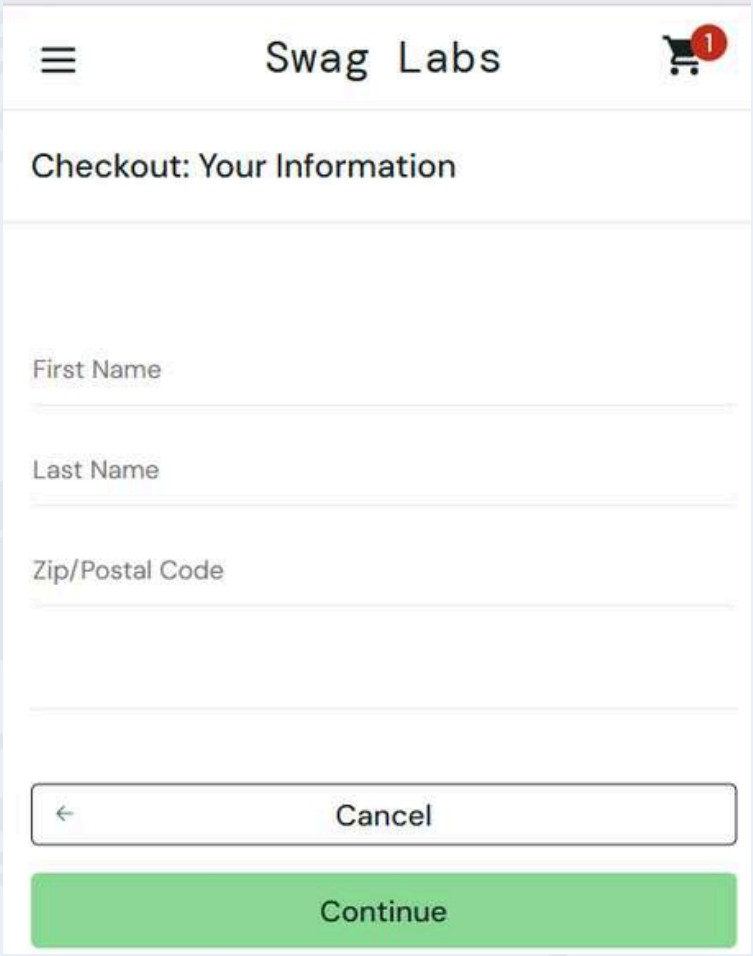
2

Le deuxième test “Ajouter un produit au panier et valider l'affichage des éléments.” **L’objectif est de s’assurer que le panier contient bien le produit ajouté et que l'on peut finaliser une commande.**

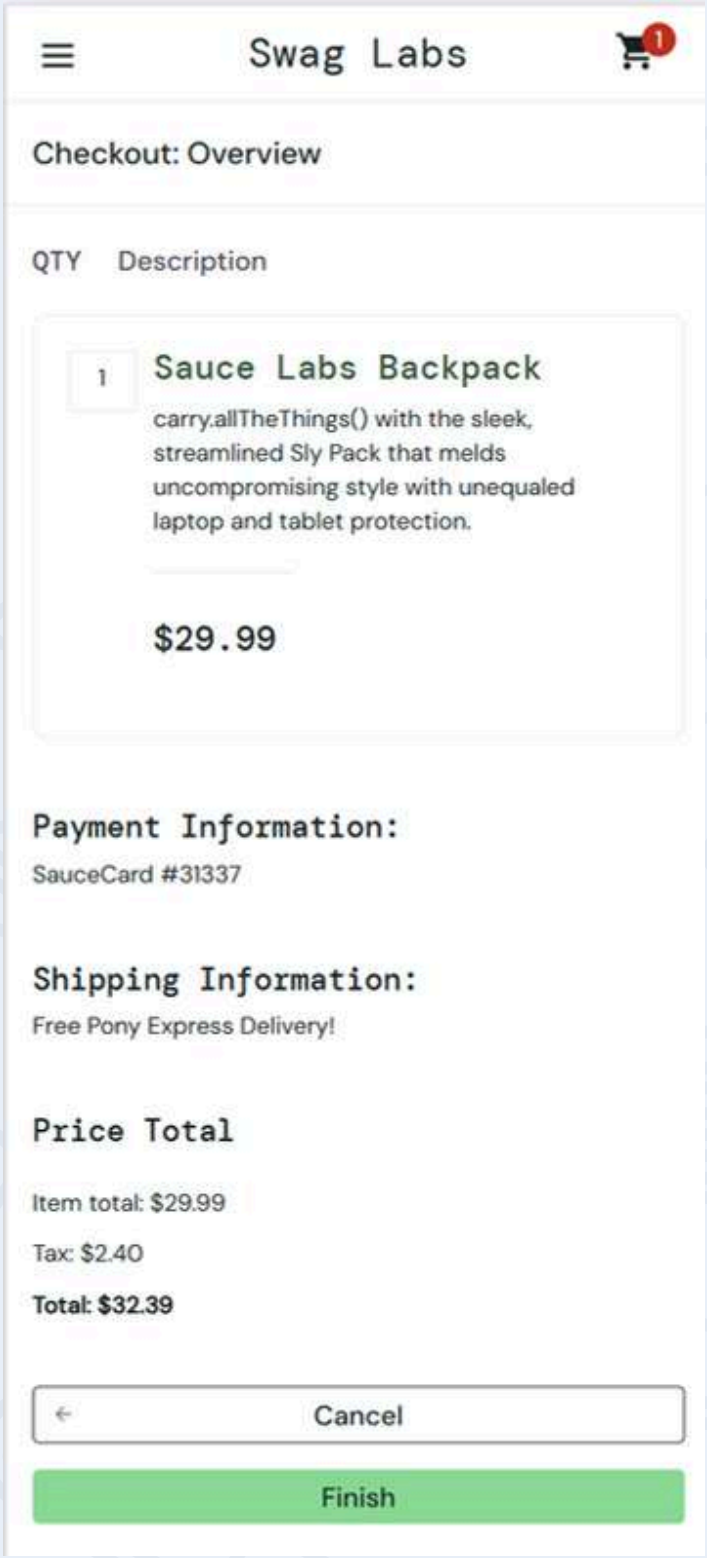
- Il faut déjà aller sur la page https://www.saucedemo.com/
- Se connecter avec un compte valide (comme dans le test précédent).
- Ajouter un produit au panier.
- Accéder à la page du panier en cliquant sur l’icône correspondante.
- Vérifier que le produit ajouté est bien présent dans le panier.

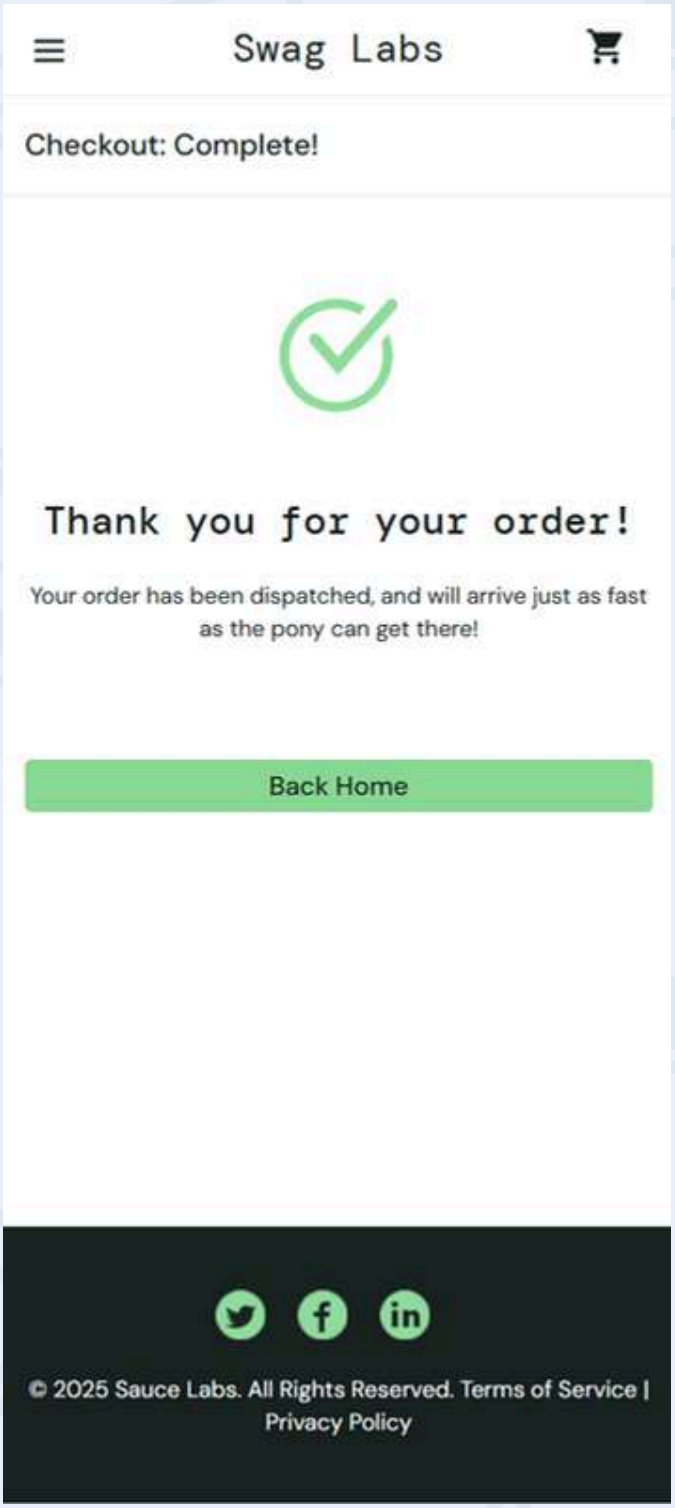


- Cliquer sur le bouton "Checkout" pour passer à l’étape suivante.
- Saisir les informations de facturation :
  - Prénom
  - Nom
  - Code postal



- Valider les informations et passer à l'étape de confirmation de commande.
- Vérifier que le récapitulatif de commande est bien affiché.
- Finaliser la commande en cliquant sur "Finish".
- Vérifier que le message "Thank you for your order!" apparaît.

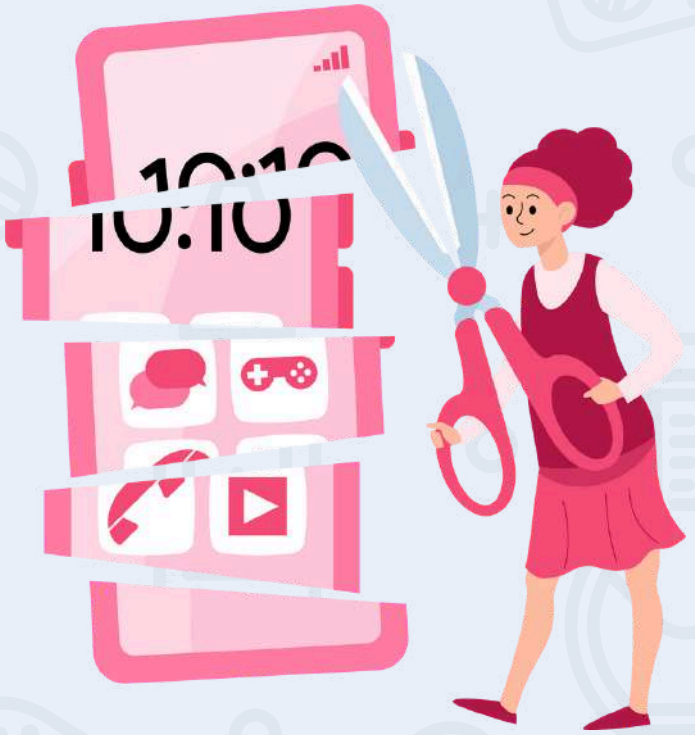




3

Le troisième test “Tester la déconnexion et vérifier la persistance des sessions” :

- Il faut déjà aller sur la page https://www.saucedemo.com/
- Se connecter avec un compte valide
- Ouvrir le menu latéral en cliquant sur le bouton hamburger (les trois barres).
- Cliquer sur "Logout" pour se déconnecter.
- Vérifier que l'utilisateur est bien redirigé vers la page de connexion.
- Vérifier que le bouton "Login" est de nouveau visible.



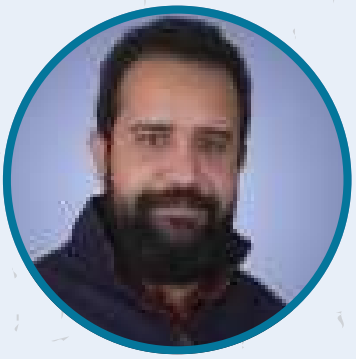
4

Le dernier test “Utiliser un compte "buggué" (problem_user) pour voir comment les erreurs sont gérées”.

- Aller sur la page https://www.saucedemo.com/
- Saisir les identifiants :
  - Nom d'utilisateur : problem_user
  - Mot de passe : secret_sauce
- Cliquer sur "Login" et vérifier que la connexion est réussie.
- Observer les éventuels bugs visuels ou fonctionnels, notamment :
  - Les images des produits s’affichent-elles correctement ?
  - Les boutons fonctionnent-ils normalement ?
  - Y a-t-il des incohérences dans la liste des produits ?
- Ajouter un produit au panier et vérifier si l'ajout fonctionne correctement.
- Aller sur la page du panier et vérifier si les produits ajoutés sont bien visibles.
- Tenter de passer la commande et observer s'il y a des comportements anormaux (bugs d'affichage, erreurs inattendues...).
- Se déconnecter en utilisant le menu latéral et s'assurer que l’action fonctionne bien.

AVIS D'EXPERT

Robot Framework



Un outil extensible et adaptable au contexte des projets

Robot Framework est un framework d'automatisation de tests open source qui se distingue par sa flexibilité et son extensibilité. Il est basé sur une syntaxe simple et lisible, ce qui permet aussi bien aux testeurs fonctionnels qu'aux développeurs de l'adopter facilement. Son principal atout réside dans sa nature modulaire : il est nativement extensible grâce à des bibliothèques développées en Python ou en Java, ce qui le rend adapté à une large variété de besoins en test logiciel.

Robot Framework est un package Python (installable via la commande pip) mais aussi une bibliothèque Python. Il offre plusieurs fonctionnalités réutilisables dans des projets Python, ce qui permet, par exemple, à une entreprise de construire des outils de test dans l'objectif d'offrir un cadre unique à toutes les équipes projet de l'entreprise.

Le choix d'un outil dépend du contexte

Le choix d'un framework de test ne doit pas être fait uniquement en fonction de sa popularité ou de ses fonctionnalités, mais surtout en fonction du contexte du projet et des besoins de l'entreprise.

Par exemple, une startup qui développe une application mobile sous Android pourrait privilégier un framework conçu par Google, comme Espresso, qui est nativement intégré à l'écosystème Android. De même, une entreprise qui développe principalement en JavaScript pourrait se tourner vers Cypress ou Playwright pour optimiser l'intégration avec son stack technique.

Yassine Sidki est un expert en automatisation des tests avec plus de dix ans d'expérience, ayant aidé de grandes entreprises à intégrer et industrialiser l'automatisation des tests dans divers environnements techniques. En tant que formateur, il a permis à de nombreux apprenants de maîtriser Robot Framework, facilitant ainsi leur insertion professionnelle.

Sa démarche pédagogique et pratique a aidé ses élèves à acquérir rapidement des compétences opérationnelles en automatisation des tests. De plus, il anime "La newsletter des testeurs" sur LinkedIn, où il partage des tendances, des astuces et des bonnes pratiques dans le domaine des tests, visant à créer une communauté de passionnés désireux d'échanger et d'approfondir leurs connaissances.

En revanche, pour une équipe transverse qui apporte une expertise en test au sein d'un grand groupe, Robot Framework constitue un choix stratégique. Grâce à sa flexibilité et à son extensibilité en Python, il permet de définir un cadre de test structuré et réutilisable, couvrant un large éventail de besoins. Il facilite ainsi l'uniformisation des pratiques de test à travers plusieurs projets, en évitant la fragmentation des outils.

Une Communauté Active et Innovante

Un autre facteur clé dans le choix d'un outil open source est l'activité de sa communauté. Robot Framework bénéficie d'une communauté dynamique, qui contribue régulièrement à l'amélioration du framework et au développement de nouvelles bibliothèques.
La conférence RoboCon 2025 en est une parfaite illustration. Cette année encore, elle a mis en avant des innovations impressionnantes, notamment une bibliothèque qui étend SeleniumLibrary en intégrant une IA de self-healing des locators. Ce type d'outil permet de détecter automatiquement les modifications mineures des éléments d'une page web et d'ajuster les tests en conséquence, sans intervention humaine. Cela répond à un problème récurrent dans l'automatisation des tests : la maintenance des scripts face aux évolutions des interfaces utilisateur.

Conclusion

Il n'existe pas de meilleur outil universel. Le choix d'un framework de test dépend avant tout des technologies utilisées et du contexte du projet. Robot Framework se distingue par sa capacité à s'adapter à une grande diversité de besoins, ce qui en fait un excellent choix pour les équipes cherchant à standardiser et industrialiser leurs tests à l'échelle d'une organisation. Sa communauté active et ses évolutions constantes en font également un outil d'avenir dans le monde du test logiciel.

A lire !

En février 2025, Yassine Sidki a publié le livre "Automatisez vos tests avec Robot Framework" aux Éditions ENI. Cet ouvrage, destiné aux débutants comme aux experts, est conçu pour aider les lecteurs à maîtriser l'automatisation des tests avec Robot Framework. Il offre des conseils pratiques et des exemples concrets pour réussir des projets d'automatisation.

Ce livre est disponible en version imprimée et en ligne sur le site des Éditions ENI, ainsi que chez d'autres libraires en ligne tels que la FNAC.

CYPRESS

Il se distingue des solutions traditionnelles comme Selenium par son architecture unique et son approche orientée développeurs. Contrairement aux outils basés sur WebDriver, Cypress fonctionne directement dans le navigateur et interagit avec l'application comme le ferait un utilisateur réel.

Pourquoi utiliser Cypress ?

Cypress est particulièrement apprécié pour :

- Sa simplicité d'installation et de configuration : Une seule commande npm install cypress suffit pour démarrer.
- Sa rapidité d'exécution : Grâce à son exécution directe dans le navigateur, les tests sont plus rapides que ceux basés sur WebDriver.
- Son interactivité : L'interface graphique permet de voir en temps réel l'exécution des tests, avec une capture des erreurs et un débogage simplifié.
- Son écosystème intégré : Cypress fournit un tableau de bord interactif, des fonctionnalités de mocking (interception des requêtes réseau), et un enregistrement vidéo des tests.
- Cypress gère les attentes implicites et n'a pas besoin de wait ou sleep explicites pour attendre un élément, évitant ainsi des tests fragiles.

Quand utiliser Cypress ?

- Tests End-to-End (E2E) : Vérifier le fonctionnement global d'une application.
- Tests d'intégration : Valider le comportement d'un composant isolé avec ses dépendances.
- Tests d'interface utilisateur : Vérifier que les interactions avec l'UI fonctionnent correctement.

A lire !

En octobre 2023, j'ai publié le livre « Automatisez vos tests avec Cypress » aux Éditions ENI. Cet ouvrage, accessible aussi bien aux débutants qu'aux profils plus expérimentés, a pour objectif d'accompagner les lecteurs dans la découverte et la mise en œuvre de Cypress. À travers des explications que j'espère claires, des cas pratiques et de nombreux exemples de scripts, il vous guide pas à pas dans vos projets de tests automatisés.
Le livre est disponible en version imprimée et numérique sur le site des Éditions ENI, ainsi que chez des libraires en ligne comme la FNAC.

Limites de Cypress

- Ne supporte pas le multi-onglet et les tests natifs mobiles.
- Cypress fonctionne avec une politique de même origine (same-origin policy), ce qui complique les tests impliquant plusieurs domaines.
- Pour des suites de tests très lourdes, Cypress peut être plus lent que Selenium, surtout lorsqu'il s'agit de tests sur des applications complexes.
- Bien que Cypress soit puissant, il est limité à l'écosystème JavaScript. Ceux qui utilisent d'autres langages comme Java ou Python pourraient préférer Selenium.

Installation de Cypress

L'installation de Cypress est rapide, il suffit de taper :

```
npm install cypress --save-dev
```

```
PS C:\Users\fanny\Code\Le-mag-testeur-cypress> npm install cypress -save-dev

added 175 packages, and audited 176 packages in 1m

40 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
npm notice
npm notice New major version of npm available! 8.19.3 -> 11.2.0
npm notice Changelog: https://github.com/npm/cli/releases/tag/v11.2.0
npm notice Run npm install -g npm@11.2.0 to update!
npm notice
PS C:\Users\fanny\Code\Le-mag-testeur-cypress>
```

Fonctionnalités principales de Cypress

- Exécution dans le navigateur : Contrairement à Selenium, Cypress n'utilise pas WebDriver mais exécute les tests directement dans le navigateur, ce qui réduit la latence et les problèmes liés à la synchronisation.
- Attente automatique : Cypress détecte automatiquement la fin des requêtes réseau et attend que les éléments soient disponibles avant d'interagir avec eux.
- Débogage simplifié : Grâce à son interface, il est possible de voir l'état du DOM et de rejouer des étapes de test.
- Mocking et Stubbing des requêtes HTTP : Cypress permet de simuler des réponses d'API pour tester des scénarios spécifiques sans dépendre du backend.
- Prise en charge des tests end-to-end, d'intégration et d'UI : Cypress est principalement utilisé pour les tests E2E, mais il permet aussi d'écrire des tests d'intégration et d'interface utilisateur.

Deux fichiers sont apparus : package.json et package-lock.json et le dossier node_modules

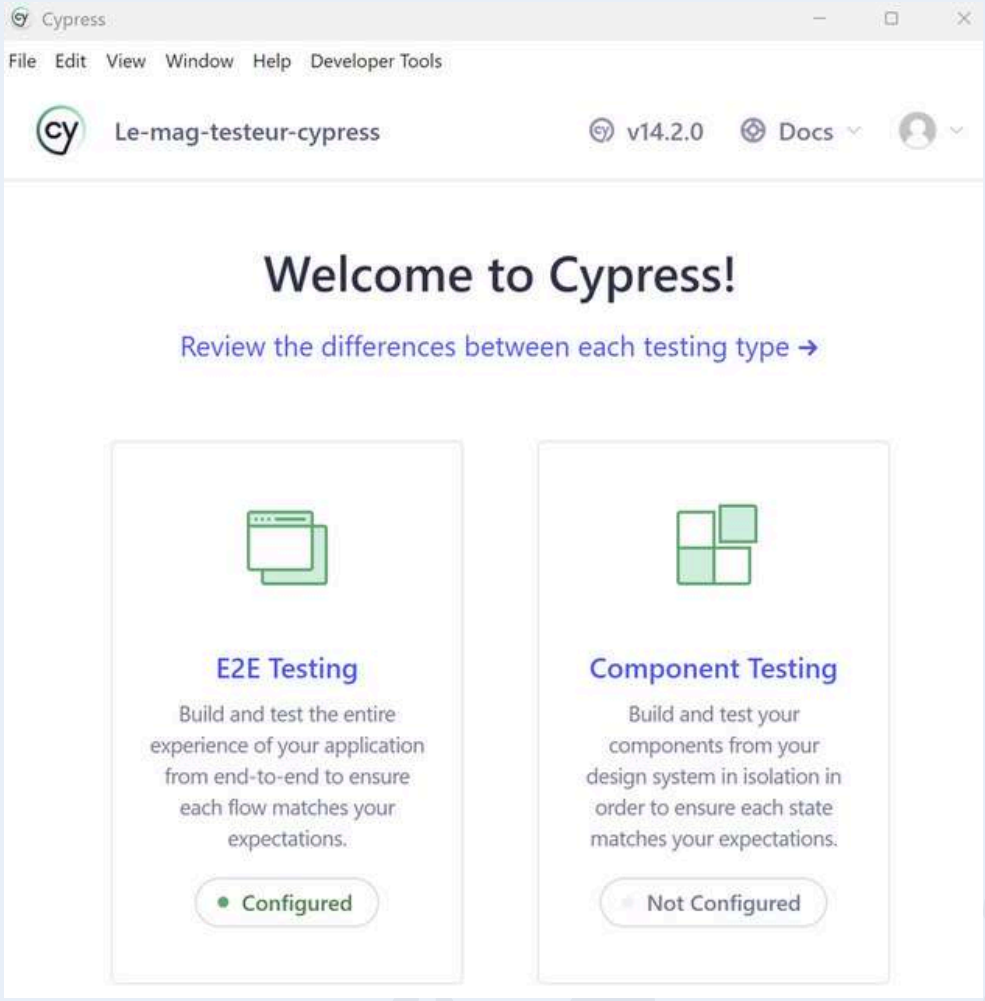
Configuration

Lors de la première exécution de Cypress, il faut le configurer. Pour cela, il suffit d'exécuter une commande :

```
npx cypress open
```

```
PS C:\Users\fanny\Code\Le-mag-testeur-cypress> npx cypress open
It looks like this is your first time using Cypress: 14.2.0

✓ Verified Cypress! C:\Users\fanny\AppData\Local\Cypress\Cache\14.2.0\Cypress

Opening Cypress...

DevTools listening on ws://127.0.0.1:53555/devtools/browser/6caad226-0d3e-46a3-812d-
```

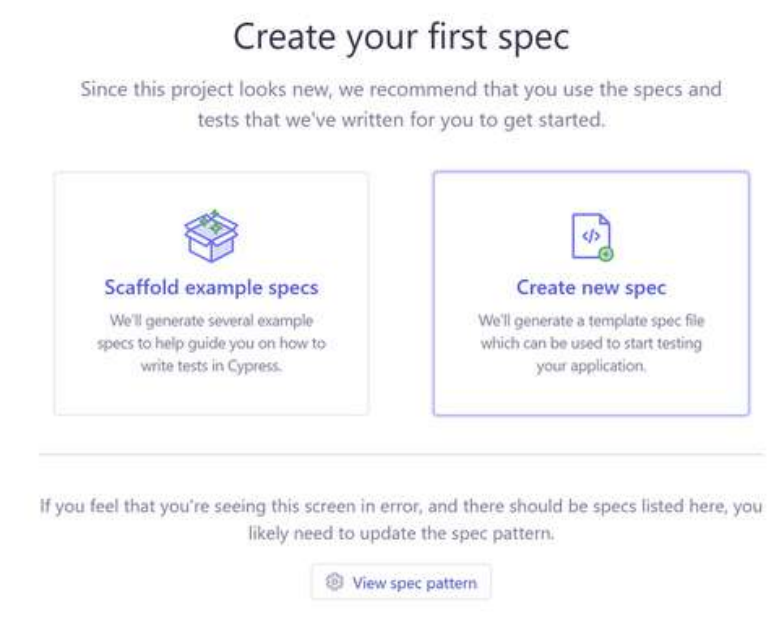
Une fenêtre s'ouvre



Vous pouvez configurer E2E testing ou Component Testing. On parle de E2E testing pour les tests de bout en bout alors que Component Testing sont pour tester individuellement les tests comme un bouton.

Cliquez sur E2E Testing et suivez les étapes. L'architecture est créée :

Ensuite, vous pourrez créer vos tests et les exécuter. Cliquer sur "Create new spec" :

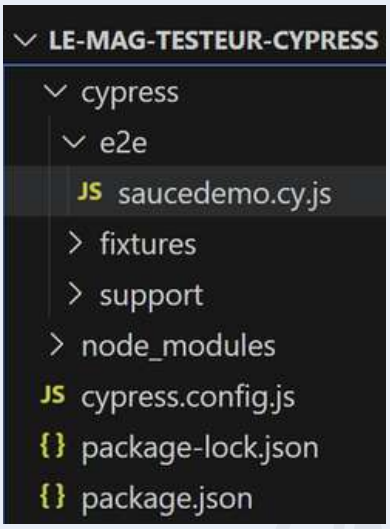


Vous pouvez le nommer par exemple : cypress\e2e\saucedemo.cy.js. Puis cliquez sur "create another spec"

Reprenons les étapes que nous avions vu dans la partie Sauce Demo.

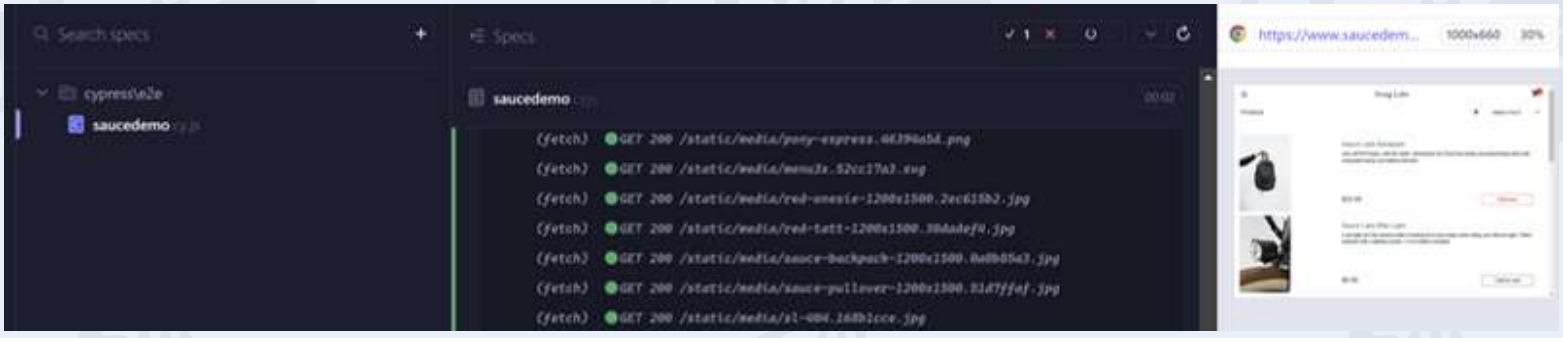
Connexion à l'application avec un utilisateur valide

Vous pouvez ajouter ce test dans le fichier qui a été créé dans votre repository. Vous pouvez trouver ci-contre la nouvelle architecture du projet et le code ci-dessous :




```
it('Se connecter et ajouter un produit au panier', () => {
    // Aller sur la page
    cy.visit('https://www.saucedemo.com/');
    // Connexion avec un utilisateur valide
    cy.get('[data-test="username"]').type('standard_user');
    cy.get('[data-test="password"]').type('secret_sauce');
    cy.get('[data-test="login-button"]').click();

    // Vérifier que la connexion est réussie
    cy.url().should('include', '/inventory.html');
    cy.contains('Products').should('be.visible');

    // Ajouter le premier produit au panier
    cy.get('.inventory_item').first().within(() => {
      cy.get('.btn_inventory').click();
    });

    // Vérifier que l'icône du panier indique 1 produit
    cy.get('.shopping_cart_badge').should('contain', '1');
  });
```

Vous pouvez exécuter le test avec l'interface graphique comme ci-dessous avec la commande npx cypress open ou en ligne de commande headless (sans ouverture de navigateur) avec la ligne de commande npx run cypress. Cette dernière est très pratique pour l'exécution dans une CI.



Ajouter un produit au panier et valider l'affichage des éléments.

Vous pouvez ajouter ce test dans le fichier votre fichier.

```
it('Vérifier le panier et finaliser la commande', () => {
  // Connexion
  cy.get('[data-test="username"]').type('standard_user');
  cy.get('[data-test="password"]').type('secret_sauce');
  cy.get('[data-test="login-button"]').click();

  // Ajouter un produit
  cy.get('.inventory_item').first().within(() => {
    cy.get('.btn_inventory').click();
  });

  // Aller au panier
  cy.get('.shopping_cart_link').click();
  cy.url().should('include', '/cart.html');

  // Vérifier la présence du produit dans le panier
  cy.get('.cart_item').should('have.length', 1);

  // Procéder au checkout
  cy.get('[data-test="checkout"]').click();
  cy.url().should('include', '/checkout-step-one.html');

  // Remplir les informations du client
  cy.get('[data-test="firstName"]').type('John');
  cy.get('[data-test="lastName"]').type('Doe');
  cy.get('[data-test="postalCode"]').type('75001');
  cy.get('[data-test="continue"]').click();

  // Vérifier l'étape finale et terminer l'achat
  cy.get('[data-test="finish"]').click();
  cy.contains('Thank you for your order!').should('be.visible');
});
```

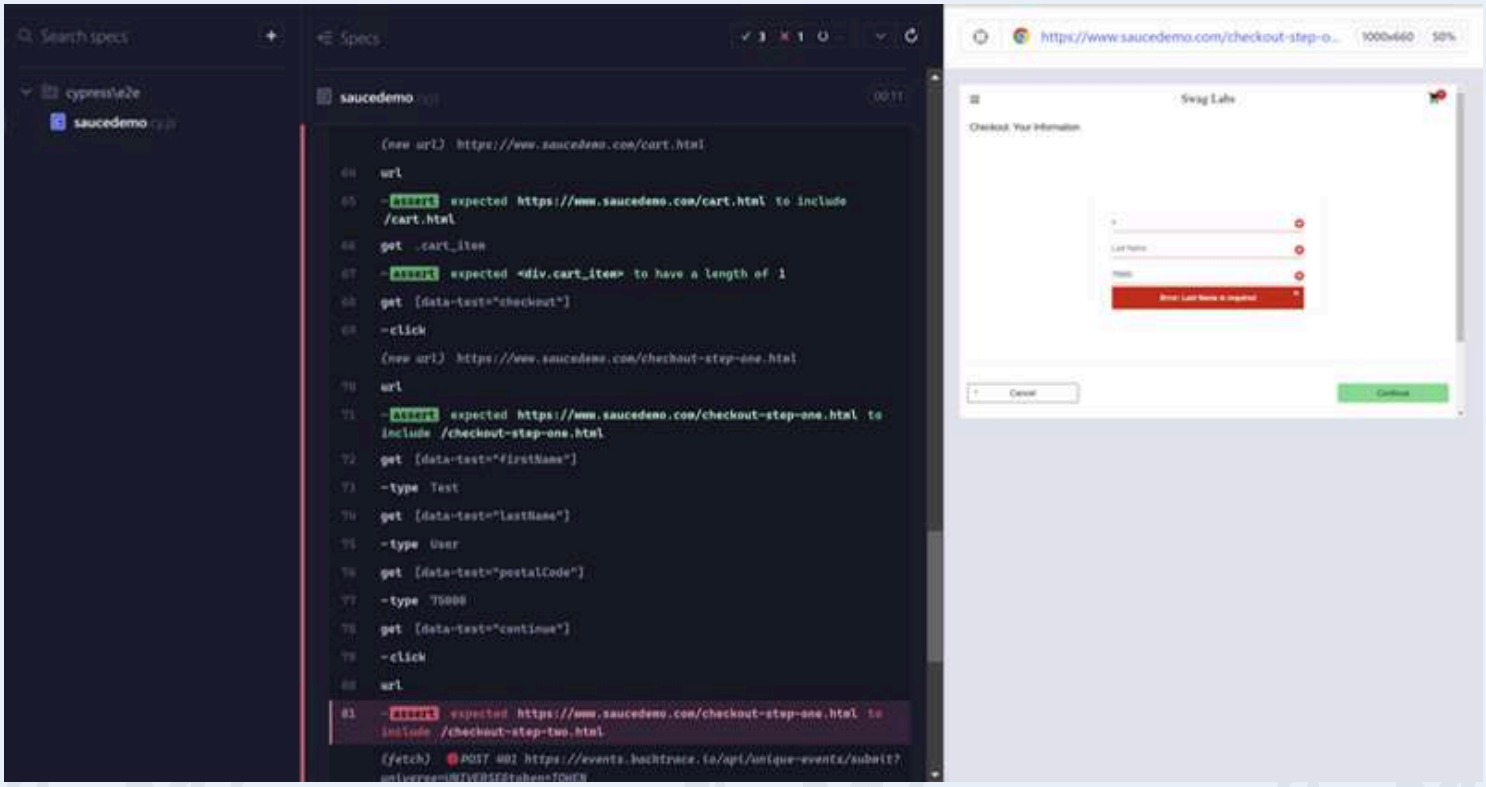
Tester la déconnexion et vérifier la persistance des sessions

Vous pouvez ajouter ce test dans le fichier qui a été créé dans votre repository. Vous pouvez trouver ci-contre la nouvelle architecture du projet et le code ci-dessous :

```
it('Se déconnecter de l application', () => {
  // Connexion
  cy.get('[data-test="username"]').type('standard_user');
  cy.get('[data-test="password"]').type('secret_sauce');
  cy.get('[data-test="login-button"]').click();

  // Ouvrir le menu et cliquer sur Logout
  cy.get('#react-burger-menu-btn').click();
  cy.get('#logout_sidebar_link').click();

  // Vérifier que l'on est bien déconnecté
  cy.url().should('include', '/');
  cy.get('[data-test="login-button"]').should('be.visible');
});
```

Utiliser un compte "buggué" (problem_user) pour voir comment les erreurs sont gérées


```
it('Se connecter avec problem_user et vérifier les bugs potentiels', () => {
    // Connexion
    cy.get('[data-test="username"]').type('problem_user');
    cy.get('[data-test="password"]').type('secret_sauce');
    cy.get('[data-test="login-button"]').click();

    // Vérifier la connexion réussie en contrôlant l'URL
    cy.url().should('include', '/inventory.html');

    // Vérifier si les images des produits sont bien affichées
    cy.get('.inventory_item_img').each(($img) => {
      cy.wrap($img).should('be.visible');
    });

    // Vérifier que les boutons d'ajout au panier sont bien cliquables
    cy.get('.btn_inventory').each(($btn) => {
      cy.wrap($btn).should('be.visible').and('be.enabled');
    });

    // Vérifier la cohérence des produits affichés
    cy.get('.inventory_item').should('have.length', 6);

    // Ajouter un produit au panier
    cy.get('.inventory_item').first().find('.btn_inventory').click();

    // Vérifier que l'icône du panier est mise à jour
    cy.get('.shopping_cart_badge').should('be.visible').and('contain', '1');

    // Aller sur la page du panier
    cy.get('.shopping_cart_link').click();
    cy.url().should('include', '/cart.html');

    // Vérifier que le produit ajouté est bien dans le panier
    cy.get('.cart_item').should('have.length', 1);

    // Tenter de passer la commande
    cy.get('[data-test="checkout"]').click();
    cy.url().should('include', '/checkout-step-one.html');

    // Remplir les informations de commande
    cy.get('[data-test="firstName"]').type('Test');
    cy.get('[data-test="lastName"]').type('User');
    cy.get('[data-test="postalCode"]').type('75000');
    cy.get('[data-test="continue"]').click();

    // Vérifier la présence des éléments de la commande
    cy.url().should('include', '/checkout-step-two.html');
    cy.get('.cart_item').should('have.length', 1);
  });
```



N'utilisez pas Cypress si :

- Vous devez tester une application nécessitant plusieurs onglets.
- Vous devez tester du mobile.
- Vous devez exécuter des tests sur différents domaines (cross-origin).
- Vous avez une équipe qui n'est pas JS-oriented.

PLAYWRIGHT

Pourquoi utiliser Playwright ?

Playwright est un framework moderne d'automatisation de tests end-to-end développé par Microsoft. Il se distingue par :

- Son support natif multi-navigateurs (Chromium, Firefox, WebKit).
- Sa capacité à gérer les scénarios complexes (multi-onglets, iframes, authentification, etc.).
- Une API moderne, stable et complète permettant des tests robustes.
- Une exécution rapide et fiable, grâce à des mécanismes d'attente automatique.

Quand utiliser Playwright?

- Tests End-to-End multi-plateformes : il exécute les tests sur Chromium, Firefox, webkit (donc sur windows, mac et linux).
- Gère nativement ces cas complexes : tests avec des iframes, popups et authentifications
- Tests sur des apps modernes en JS comme React, Angular, Vue...
- Automatisations dans les pipelines CI/CD : grâce à son mode headless rapide, il est idéal dans ces workflows CI
- Simulation réaliste d'un utilisateur : contrôle clavier, souris, API, réseau, géolocalisation, fichiers, ...

Limites de Playwright

Même s'il est très complet, Playwright présente aussi quelques limitations :

- Pas de support direct pour les applications mobiles natives (contrairement à Appium). Cependant, il y a Playwright for Android (expérimental) pour piloter du mobile web via WebKit iOS/Chromium Android sur émulateur
- La communauté est plus jeune que celle de Selenium donc il y a moins de plugins ou d'intégrations
- Les tests avec un rendu visuel pixel-perfect sont plus adaptés à des outils spécialisés comme Applitools. Il y a une solution qui existe cependant : prendre un screen shot et le comparer à l'attendu.

Fonctionnalités principales

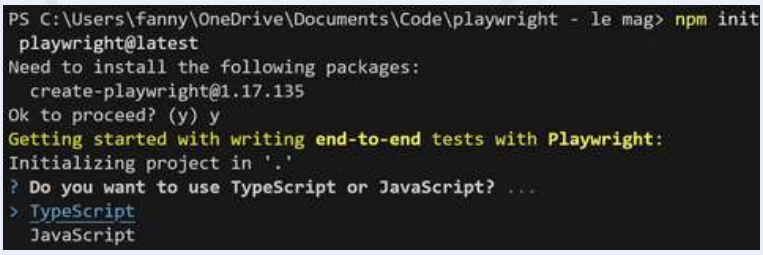
- Comme nous l'avons vu, il est multi-navigateur
- Multi-onglets et iframes
- Attente automatique
- Générateur de code intégré : npx playwright codegen génére des tests automatiquement à partir de clics dans le navigateur
- Test réseau et mocking : possibilité d'intercepter les requêtes HTTP/GraphQL et simuler des réponses
- Tests parallèles : optimisé pour l'exécution concurrente des tests
- Vidéos et screenshots
- Exécution sur CI/CD : compatible avec GitHub Actions, Gitlab CI, Jenkins, ...

Installation de Playwright

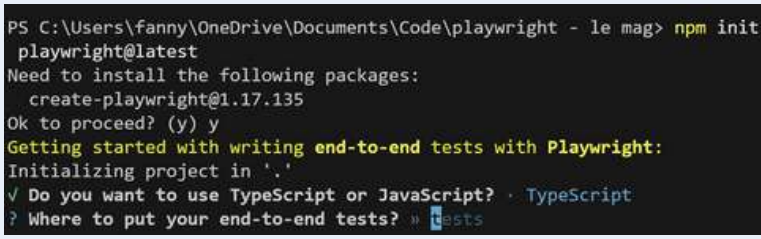
L'installation de Playwright est rapide, il suffit d'exécuter :

```
npm init playwright@latest
```

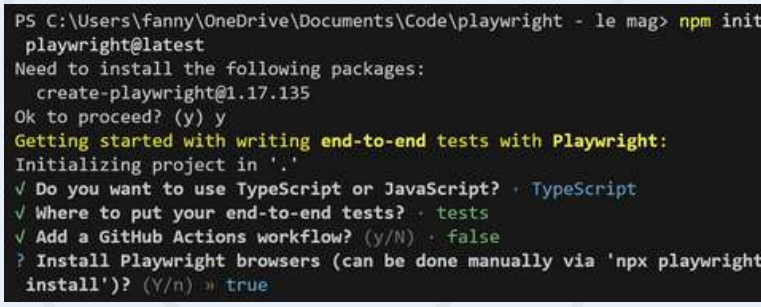
Ensuite, vous allez devoir choisir entre Typescript ou Javascript. La valeur sélectionnée par défaut est Typescript.



La prochaine question vous demande d'entrer le nom de votre dossier Tests (la valeur par défaut est tests ou e2e si vous avez déjà un dossier tests dans votre projet)



Ajoutez un workflow GitHub Actions pour exécuter facilement des tests sur CI. Nous verrons plus en détails cette solution dans la partie CI.




```
Initializing NPM project (npm init -y)…
Wrote to C:\Users\fanny\OneDrive\Documents\Code\playwright - le mag\package.json:

{
  "name": "playwright---le-mag",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}


Installing Playwright Test (npm install --save-dev @playwright/test)…

added 3 packages, and audited 4 packages in 3s

found 0 vulnerabilities
Installing Types (npm install --save-dev @types/node)…

added 2 packages, and audited 6 packages in 1s

found 0 vulnerabilities
Writing playwright.config.ts.
Writing tests\example.spec.ts.
Writing tests-examples\demo-todo-app.spec.ts.
Writing package.json.
Downloading browsers (npx playwright install)…
Downloading Chromium 136.0.7103.25 (playwright build v1169) from https://cdn.playwright.dev/dbazure/download/playwright/builds/chromium/1169/chromium-win64.zip
```

Playwright commence par créer un projet Node "comme squelette". Le -y équivaut à répondre "yes" à toutes les questions de npm init, d'où la génération immédiate d'un package.json minimal.

Le wizard installe la dépendance principale : @playwright/test, c'est-à-dire le runner + l'API assertion + les fixtures. L'option --save-dev déclare la lib dans devDependencies, car elle sert uniquement durant le développement/CI et pas en production.
npm informe : 3 paquets téléchargés (le runner + deux dépendances), 4 vérifiés au total (les 3 nouveaux + @types/node déjà en cache), aucun problème de sécurité détecté par la base npm-audit.

Téléchargement du navigateur complet pour l'exécution « headed » (avec interface).

```
144.4 MiB [====================] 100% 0.0s
Chromium 136.0.7103.25 (playwright build v1169) downloaded to C:\Users\
fanny\AppData\Local\ms-playwright\chromium-1169
Downloading Chromium Headless Shell 136.0.7103.25 (playwright build v11
69) from https://cdn.playwright.dev/dbazure/download/playwright/builds/
chromium/1169/chromium-headless-shell-win64.zip
89.1 MiB [====================] 100% 0.0s
Chromium Headless Shell 136.0.7103.25 (playwright build v1169) download
ed to C:\Users\fanny\AppData\Local\ms-playwright\chromium_headless_shel
l-1169
Downloading Firefox 137.0 (playwright build v1482) from https://cdn.pla
ywright.dev/dbazure/download/playwright/builds/firefox/1482/firefox-win
64.zip
92.1 MiB [====================] 100% 0.0s
Firefox 137.0 (playwright build v1482) downloaded to C:\Users\fanny\App
Data\Local\ms-playwright\firefox-1482
Downloading Webkit 18.4 (playwright build v2158) from https://cdn.playw
right.dev/dbazure/download/playwright/builds/webkit/2158/webkit-win64.z
ip
57.1 MiB [====================] 100% 0.0s
Webkit 18.4 (playwright build v2158) downloaded to C:\Users\fanny\AppDa
ta\Local\ms-playwright\webkit-2158
Downloading FFMPEG playwright build v1011 from https://cdn.playwright.d
ev/dbazure/download/playwright/builds/ffmpeg/1011/ffmpeg-win64.zip
1.3 MiB [====================] 100% 0.0s
FFMPEG playwright build v1011 downloaded to C:\Users\fanny\AppData\Loca
l\ms-playwright\ffmpeg-1011
Downloading Winldd playwright build v1007 from https://cdn.playwright.d
ev/dbazure/download/playwright/builds/winldd/1007/winldd-win64.zip
0.1 MiB [====================] 100% 0.0s
Winldd playwright build v1007 downloaded to C:\Users\fanny\AppData\Loca
l\ms-playwright\winldd-1007
✓ Success! Created a Playwright Test project at C:\Users\fanny\OneDrive
\Documents\Code\playwright - le mag

Inside that directory, you can run several commands:

  npx playwright test
    Runs the end-to-end tests.

  npx playwright test --ui
    Starts the interactive UI mode.

  npx playwright test --project=chromium
    Runs the tests only on Desktop Chrome.

  npx playwright test example
    Runs the tests in a specific file.

  npx playwright test --debug
    Runs the tests in debug mode.

  npx playwright codegen
    Auto generate tests with Codegen.

We suggest that you begin by typing:

    npx playwright test

And check out the following files:
  - .\tests\example.spec.ts - Example end-to-end test
  - .\tests-examples\demo-todo-app.spec.ts - Demo Todo App end-to-end t
ests
  - .\playwright.config.ts - Playwright Test configuration

Visit https://playwright.dev/docs/intro for more information. ✨

Happy hacking! 🎭
npm notice
npm notice New major version of npm available! 8.19.3 -> 11.3.0
npm notice Changelog: https://github.com/npm/cli/releases/tag/v11.3.0
npm notice Run npm install -g npm@11.3.0 to update!
npm notice
```

Variante ultra-allégée destinée au mode headless. Elle évite de télécharger tout Chromium (CI sans affichage).

Build officiel Firefox, figé pour garantir la reproductibilité entre machines.

Version Apple WebKit patchée par l'équipe Playwright (utile pour tester Safari sous Windows/Linux).

Bibliothèque qui assure l'enregistrement vidéo et la capture audio pendant les tests

Playwright l'utilise pour lister les DLL requises et vérifier qu'elles sont présentes sur la machine cible

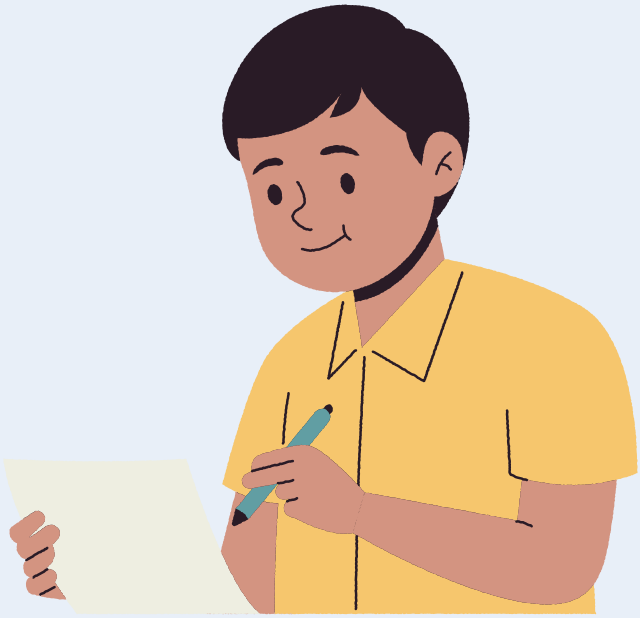
Message de réussite

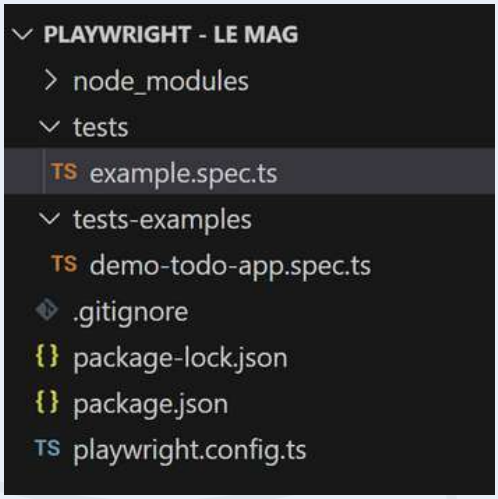
Alerte possible d'une nouvelle version npm disponible. Ce n'est pas une obligation d'upgrader.

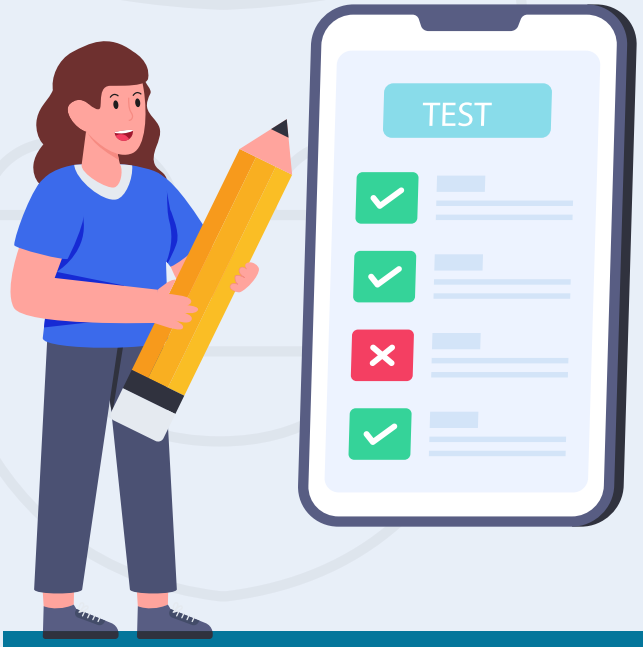
Ensuite, il faut y noter l'ajout de playwright.config.ts et deux fichiers de test dans deux nouveaux dossier : tests/example.spect.ts & tests-examples/demo-todo-app.spec.ts.

Le fichier playwright.config.ts est un fichier de configuration, vous allez le modifier par exemple si vous voulez l'exécution de tests en parallèle, la personnalisation du rapport de tests, etc...

Les deux autres fichiers tests/example.spec.ts & tests-examples/demo-todo-app.spec.ts. sont des fichiers d'exemples de Playwright. Nous vous conseillons de les lire et de les comprendre car ce sont des notions à prendre en main.







A lire !

En décembre 2024, YKailash Pathak a publié le livre "Web automation Testing Using Playwright" aux Éditions bpb. Cet ouvrage, couvre les tests E2E, API, accessibilité, ... avec Playwright

Ce livre est disponible en version imprimée et format kindle sur amazon. Le livre est rédigé en anglais.

Reprenons les étapes que nous avions vu dans la partie Sauce Demo.

Connexion à l'application avec un utilisateur valide

Vous pouvez créer un nouveau fichier connexion.spec.ts. Et dans ce nouveau fichier :

```
// tests/login-add-to-cart.spec.ts
import { test, expect } from '@playwright/test';

test('Connexion standard_user puis ajout au panier', async ({ page }) => {
  /* 1. Page de login */
  await page.goto('https://www.saucedemo.com/');

  // Sélection des champs par id (les id sont stables sur ce site-démo)
  await page.fill('#user-name', 'standard_user');
  await page.fill('#password', 'secret_sauce');

  // Bouton « Login »
  await page.click('#login-button');

  /* 2. Vérifier la redirection */
  // L'URL doit contenir /inventory.html ; on vérifie aussi que le titre « Products » est
visible
  await expect(page).toHaveURL(/inventory\.html/);
  await expect(page.locator('.title')).toHaveText('Products');

  /* 3. Sélection du premier produit puis clic sur « Add to cart » */
  // .inventory_item représente chaque carte produit ; on prend le premier
  const firstItem = page.locator('.inventory_item').first();

  // Le bouton porte l'attribut data-test="add-to-cart-..." : plus fiable que le texte
  await firstItem.locator('[data-test^="add-to-cart"]').click();

  /* 4. Vérifier le badge du panier */
  const cartBadge = page.locator('.shopping_cart_badge');
  await expect(cartBadge).toHaveText('1');
});
```

Avant d'executer le test, vous pouvez lister les tests qui vont être exécuter afin de voir les tests mais aussi les navigateurs qui vont être utilisés :

```
npx playwright test --list
```

Avant d'executer le test, vous pouvez lister les tests qui vont être exécuter afin de voir les tests mais aussi les navigateurs qui vont être utilisés :

```
PS C:\Users\fanny\OneDrive\Documents\Code\playwright - le mag> npx playwright test --list
Listing tests:
  [chromium] › connexion.spec.ts:4:5 › Connexion standard_user puis ajout au panier
  [chromium] › example.spec.ts:3:5 › has title
  [chromium] › example.spec.ts:10:5 › get started link
  [firefox] › connexion.spec.ts:4:5 › Connexion standard_user puis ajout au panier
  [firefox] › example.spec.ts:3:5 › has title
  [firefox] › example.spec.ts:10:5 › get started link
  [webkit] › connexion.spec.ts:4:5 › Connexion standard_user puis ajout au panier
  [webkit] › example.spec.ts:3:5 › has title
  [webkit] › example.spec.ts:10:5 › get started link
Total: 9 tests in 2 files
```

Supprimez le fichier example.spect.ts et exécuter à nouveau la commande :

```
PS C:\Users\fanny\OneDrive\Documents\Code\playwright - le mag> npx playwright test --list
Listing tests:
  [chromium] › connexion.spec.ts:4:5 › Connexion standard_user puis ajout au panier
  [firefox] › connexion.spec.ts:4:5 › Connexion standard_user puis ajout au panier
  [webkit] › connexion.spec.ts:4:5 › Connexion standard_user puis ajout au panier
Total: 3 tests in 1 file
```

Vous pouvez y voir 3 exécutions, une sur chaque navigateur : Chrome, firefox et webkit.
Exécuter le test avec :

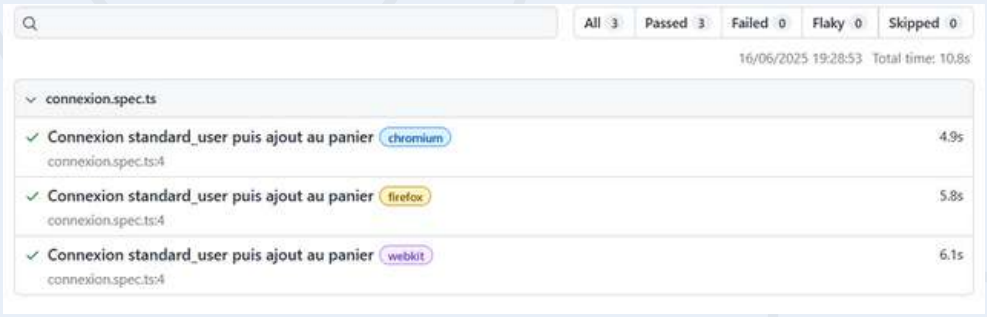
```
npx playwright test
```

```
PS C:\Users\fanny\OneDrive\Documents\Code\playwright - le mag> npx playwright test

Running 3 tests using 3 workers
  3 passed (10.8s)

To open last HTML report run:

  npx playwright show-report
```

Pour voir le rapport, exécuter :

```
npx playwright show-report
```

Dans un navigateur, naviguez jusque :
http://localhost:9323



Ajouter un produit au panier et valider l'affichage des éléments.

2

Vous pouvez ajouter ce test dans un nouveau fichier : test/checkout-order.spec.ts

```
// tests/checkout-order.spec.ts
import { test, expect } from '@playwright/test';

test('Ajouter un produit puis finaliser la commande', async ({ page }) => {
  /* Page de login */
  await page.goto('https://www.saucedemo.com/');

  await page.fill('#user-name', 'standard_user');
  await page.fill('#password', 'secret_sauce');
  await page.click('#login-button');

  /* Vérifier la redirection vers l'inventaire */
  await expect(page).toHaveURL(/inventory\.html/);
  await expect(page.locator('.title')).toHaveText('Products');

  /* Sélection du premier produit et ajout au panier */
  const firstItem = page.locator('.inventory_item').first();
  const productName = (await
firstItem.locator('.inventory_item_name').textContent())?.trim();

  await firstItem.locator('[data-test^="add-to-cart"]').click();

  /* Accéder au panier et vérifier le contenu */
  await page.click('.shopping_cart_link');
  await expect(page).toHaveURL(/cart\.html/);

  const cartItem = page.locator('.cart_item .inventory_item_name');
  await expect(cartItem).toHaveText(productName!);

  /* Démarrer le checkout */
  await page.click('[data-test="checkout"]');

  /* Formulaire d'informations client */
  await expect(page).toHaveURL(/checkout-step-one\.html/);
  await page.fill('#first-name', 'Fanny');
  await page.fill('#last-name', 'Tester');
  await page.fill('#postal-code', '75000');
  await page.click('[data-test="continue"]');

  /* Vérifier le récapitulatif */
  await expect(page).toHaveURL(/checkout-step-two\.html/);
  await expect(page.locator('.summary_info')).toBeVisible();

  /* Finaliser la commande */
  await page.click('[data-test="finish"]');

  /* Vérification finale */
  await expect(page).toHaveURL(/checkout-complete\.html/);
  await expect(page.locator('.complete-header')).toHaveText('Thank you for your order!');
});
```

Tester la déconnexion et vérifier la persistance des sessions

Vous pouvez ajouter ce test dans un nouveau fichier logout-session-persistence.spect.ts.

```
import { test, expect } from '@playwright/test';

test('Déconnexion et vérification post-logout', async ({ page }) => {
 /* Connexion */
 await page.goto('https://www.saucedemo.com/');

 await page.fill('#user-name', 'standard_user');
 await page.fill('#password', 'secret_sauce');
 await page.click('#login-button');

 // Contrôle rapide : nous sommes bien sur la page Produits
 await expect(page).toHaveURL(/inventory\.html/);

 /* Ouverture du menu latéral */
 await page.click('#react-burger-menu-btn');
 // Le menu glisse depuis la gauche ; attendons qu'il soit réellement visible
 await expect(page.locator('.bm-menu')).toBeVisible();

 /* Déconnexion */
 await page.click('#logout_sidebar_link');

 /* Vérifications post-logout  */
  await expect(page).toHaveURL('https://www.saucedemo.com/');
 await expect(page.locator('#login-button')).toBeVisible();

 /* Persistance de session : accès direct à /inventory  */
 await page.goto('https://www.saucedemo.com/inventory.html');
 await expect(page).toHaveURL('https://www.saucedemo.com/');
 await expect(page.locator('#user-name')).toBeEmpty();
});
```

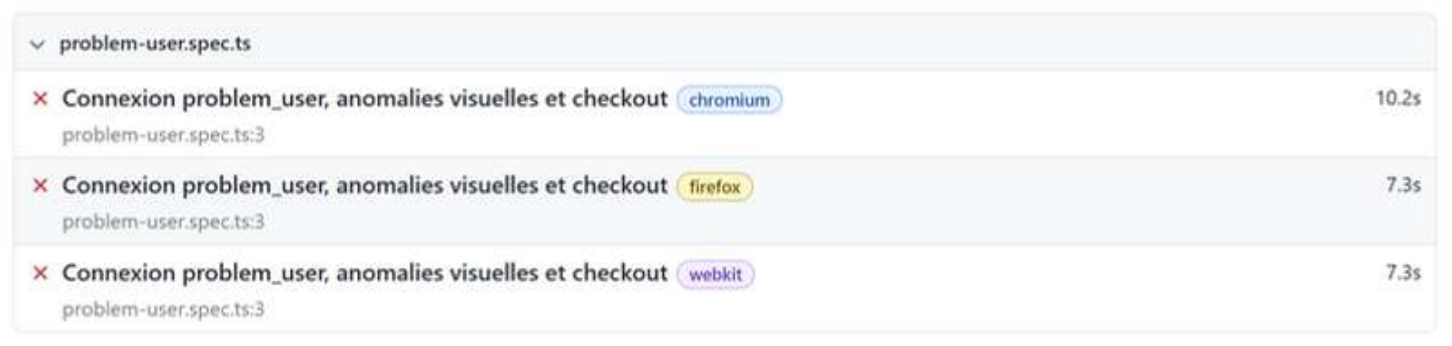
Utiliser un compte "buggué" (problem_user) pour voir comment les erreurs sont gérées

4

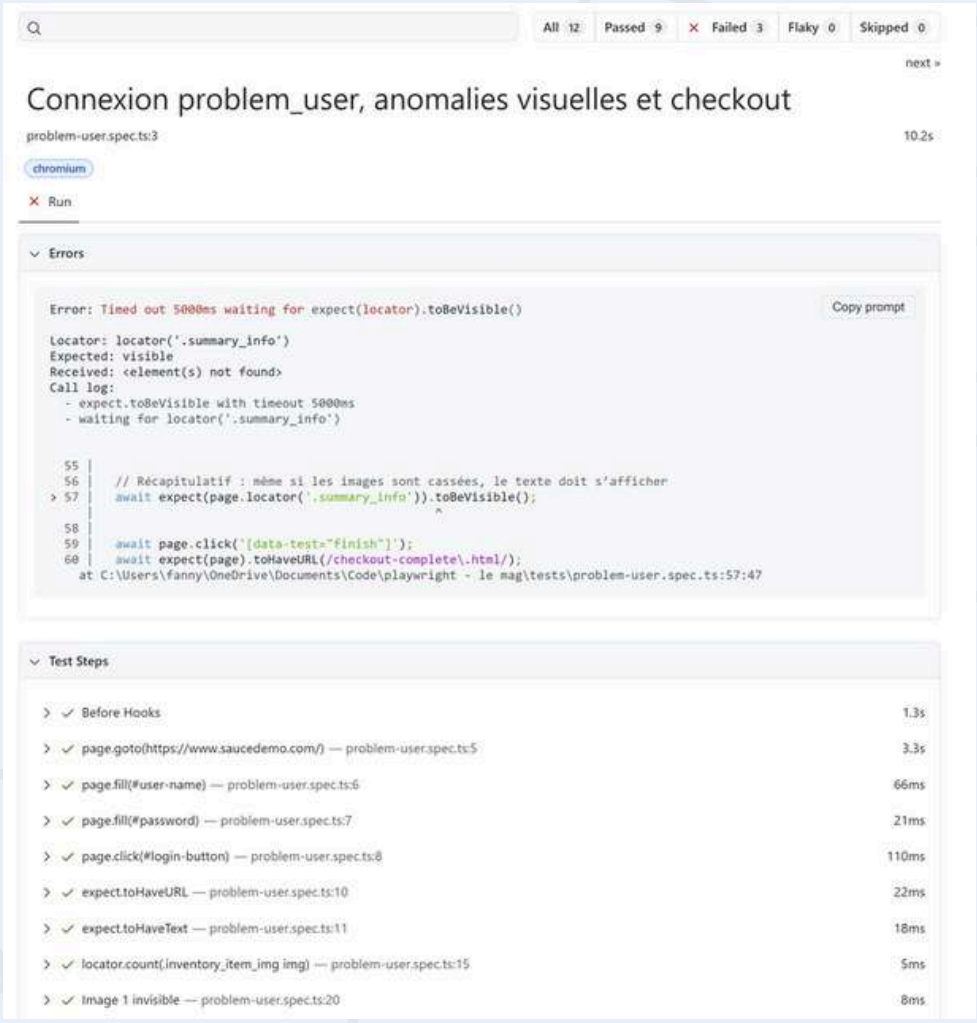
```
import { test, expect } from '@playwright/test';

test('Connexion problem_user, anomalies visuelles et checkout', async ({ page }) => {
 /*Connexion */
 await page.goto('https://www.saucedemo.com/');
 await page.fill('#user-name', 'problem_user');
 await page.fill('#password', 'secret_sauce');
 await page.click('#login-button');

 await expect(page).toHaveURL(/inventory\.html/);
 await expect(page.locator('.title')).toHaveText('Products');

 /* Inspection des images produits */
 const images = page.locator('.inventory_item_img img');
 const imgCount = await images.count();

 for (let i = 0; i < imgCount; i++) {
  const img = images.nth(i);
  // Vérification visibilité
  await expect.soft(img, `Image ${i + 1} invisible`).toBeVisible();

  const width = await img.evaluate(el => (el as HTMLImageElement).naturalWidth);
  await expect.soft(width, `Image ${i + 1} cassée`).toBeGreaterThan(0);
 }

 /* Test des boutons « Add / Remove » */
 const firstItem = page.locator('.inventory_item').first();
 const addBtn = firstItem.locator('[data-test^="add-to-cart"]');

 await expect.soft(addBtn).toBeVisible();
 await expect.soft(addBtn).toBeEnabled();

 await addBtn.click();

 // Le badge panier doit afficher « 1 »
 const badge = page.locator('.shopping_cart_badge');
 await expect(badge).toHaveText('1');

 /* Vérification du contenu du panier */
 await page.click('.shopping_cart_link');
 await expect(page).toHaveURL(/cart\.html/);
```

```
const cartItems = page.locator('.cart_item');
await expect(cartItems).toHaveCount(1);


await page.click('[data-test="checkout"]');
await expect(page).toHaveURL(/checkout-step-one\.html/);

await page.fill('#first-name', 'Bug');
await page.fill('#last-name', 'Hunter');
await page.fill('#postal-code', '99999');
await page.click('[data-test="continue"]');

await expect(page.locator('.summary_info')).toBeVisible();

await page.click('[data-test="finish"]');
await expect(page).toHaveURL(/checkout-complete\.html/);

// Message de confirmation
await expect(page.locator('.complete-header'))
.toHaveText('Thank You for Your Order!');

/* Déconnexion et contrôle de fin de session */
await page.click('#react-burger-menu-btn');
await expect(page.locator('.bm-menu')).toBeVisible();
await page.click('#logout_sidebar_link');

await expect(page).toHaveURL('https://www.saucedemo.com/');
await expect(page.locator('#login-button')).toBeVisible();
});
```

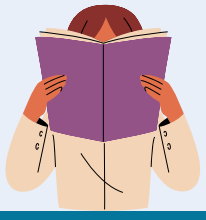
Le dernier test est en erreur, vous pouvez allé sur le rapport comme nous l'avons tout à l'heure en vous rendant sur localhost:9323 :



Cliquez sur l'un des tests en erreur :



Vous pouvez visualiser l'erreur en détail et également voir les étapes avec le temps que chacune a pris pour s'exécuter.



COMMENT STRUCTURER UNE SUITE DE TESTS AUTOMATISES EFFICACE ?

Structurer une suite de tests automatisés de manière efficace est un vrai travail **d'architecte qualité**. Cela demande une approche réfléchie, anticipative et évolutive.

Avant même de commencer le codage, posez-vous ces quelques questions :

- Que voulez-vous valider avec cette suite ?
- À quelle fréquence ces tests seront-ils exécutés (CI/CD, nightly, à la demande) ?
- À qui s'adressent les résultats ? (développeurs, PO, QA, ops...)

Cela vous permettra de distinguer :

- Tests de régression : validité fonctionnelle à chaque release.
- Tests de non-régression rapide (Smoke ou Sanity) : vérifier si l'application "tient debout".
- Tests d'intégration et d'API : pour valider les interactions entre composants.
- Tests de bout en bout : pour simuler le parcours utilisateur.
- Tests de performance : pour mesurer les temps de réponse.

Quand vous concevez une suite de tests, ne cherchez pas à tout mettre dans la même catégorie. votre rôle est de bâtir un mix adapté à votre projet :

- Des tests rapides et fiables pour le feedback immédiat.
- Des tests profonds pour les parcours critiques.
- Des tests spécifiques pour la technique (API, perf, sécurité, etc.).

Au niveau technique, des designs patterns peuvent vous aider à structurer votre projet. En voici quelques uns :

- **Modèle de séparation des tests par type**

Idéal quand vous gérez plusieurs niveaux de tests dans un même repo :

```
tests/
├── unit/
│   └── utils.test.js
├── integration/
│   └── api-auth.spec.js
└── e2e/
    └── user-journey.spec.js
```

Cela vous permet :

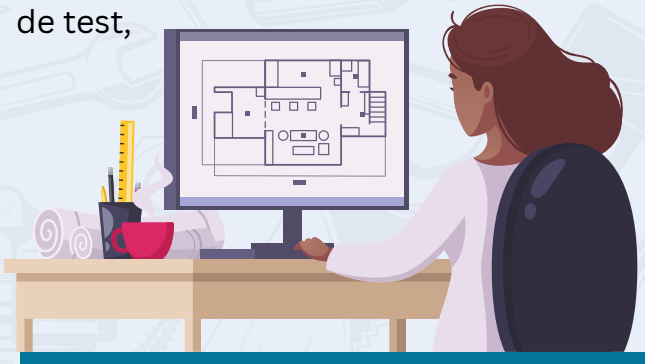
- d'exécuter les tests par niveau
- de mieux prioriser dans la CI/CD
- de filtrer facilement les campagnes

- **Pattern Page Object Model (POM)**

Très utile pour les tests UI. Chaque page de votre application est représentée par une classe/fichier.

Les bénéfices sont de :

- séparer la logique de navigation/interaction de la logique métier de test,

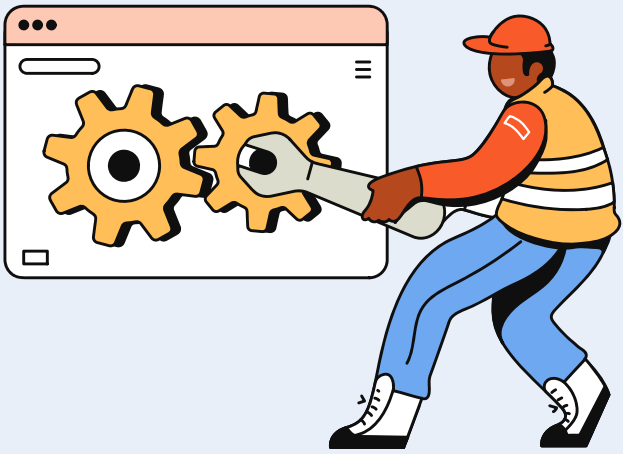


- centraliser les sélecteurs et actions d'une page,
- réduire la duplication de code,
- et faciliter la maintenance en cas de changement dans l'UI.

Prenons par exemple, un projet Cypress, l'architecture de votre projet sera :

```
cypress/
├── e2e/
│   ├── login.spec.js
│   └── panier.spec.js
├── pages/
│   ├── LoginPage.js
│   └── ProductPage.js
├── support/
│   ├── commands.js
│   └── e2e.js
```

COMMENT MAINTENIR SES TESTS AUTOMATISES SUR LE LONG TERME ?



Maintenir ses tests automatisés sur le long terme est un véritable défi. C'est souvent là que les projets d'automatisation échouent ou s'enlisent, non pas au moment du proof of concept, mais lorsque le **volume augmente,** que l'application évolue, et que le test automatisé devient une dette plutôt qu'un atout. Voici une réponse approfondie pour bien comprendre les leviers de pérennisation d'une stratégie de tests automatisés.

Penser « maintenance » dès la conception

Un test automatisé est une forme de code. Il doit être écrit pour être maintenable, pas juste pour "fonctionner une fois".

- **Nommer clairement** les cas de test : évitez les noms vagues comme test_1() ou testLogin. Préférez test_authentification_valide_redirige_vers_homepage().
- **Factoriser** les actions répétées : si l'on répète dix fois la même connexion ou recherche dans le code, alors il faut abstraire cela dans une fonction, un helper, ou une page object.
- Utiliser un langage compréhensible : même si vous utilisez Cypress, Selenium ou Playwright, pensez à vos collègues qui reliront ce test. Le code doit être **lisible**.

Construire une architecture de test modulaire

La modularité est la clé pour éviter l'effet domino lors des évolutions applicatives :

- Modèle Page Object / Screenplay : ces modèles permettent de séparer le « quoi tester » du « comment interagir avec l'interface ». Si un bouton change de nom, vous n'avez qu'un seul endroit à modifier.

- Fichiers de configuration centralisés : stocker les URLs, identifiants, environnements, dans un fichier unique facilite l'adaptation de vos tests.

- Découpler les données : évitez de « coder en dur » vos jeux de données dans les tests. Utilisez des fichiers JSON, des fixtures, ou un générateur dynamique.

Avoir une stratégie claire de tests automatisés

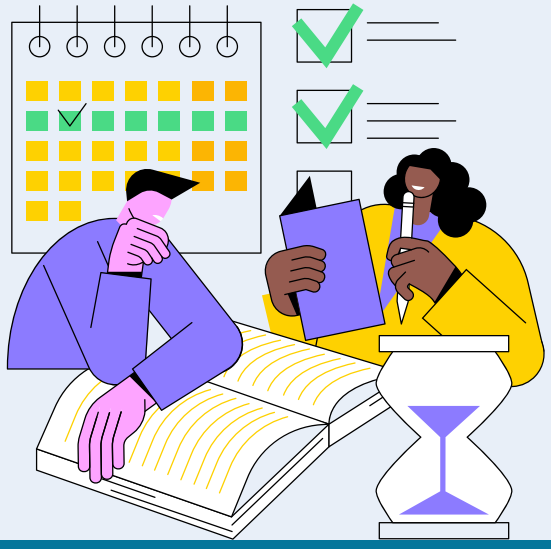
Automatiser n'est pas « tout tester ». Il faut cibler :

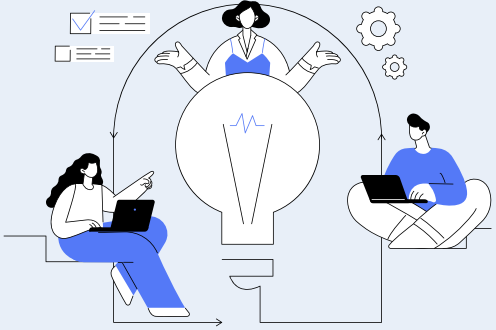
- Les chemins critiques (parcours client, flux de commande, authentification)

- Les routines à forte régression (champs calculés, règles métier)

- Les tests stables uniquement : un test qui échoue une fois sur deux érode la confiance et surcharge la maintenance.

Et surtout, réviser régulièrement le périmètre de test. Ce qui était pertinent l'an dernier ne l'est pas forcément aujourd'hui.





Intégrer les tests dans le cycle de vie du projet

- CI/CD obligatoire : les tests doivent être déclenchés automatiquement à chaque push ou merge, via GitLab CI, GitHub Actions, Jenkins ou autre.

- Feedback rapide : priorisez les tests courts en début de pipeline, réservez les tests plus longs (end-to-end, non-fonctionnels) pour la nuit ou la préprod.

- Monitoring des tests : une notification sur un test qui échoue, c'est bien. Une analyse automatique des tendances d'échec, c'est mieux.

Refactoriser régulièrement les tests

Comme le code de production, les tests doivent être nettoyés, réorganisés, supprimés si besoin :

- Supprimez les tests obsolètes (sur des fonctionnalités supprimées)

- Refactorez les tests flous, flakys ou mal nommés

- Faites des revues de tests, comme on fait des revues de code.

Un bon rythme est de prévoir un temps de maintenance par sprint ou par release.

Collaborer avec les équipes de dev, produit et infra

- La QA ne peut pas maintenir seule tous les tests. Les développeurs doivent être formés à l'automatisation et écrire certains tests.
- Le Product Owner (ou Business Analyst) peut participer via des scénarios en Gherkin si une telle approche est en place.
- L'équipe infra peut aider à fiabiliser les environnements d'exécution.

Sans cette synergie, l'automatisation devient un silo fragile.

Outils et pratiques pour faciliter la maintenance

- Tests parallélisés : réduire le temps d'exécution pour maintenir la vitesse.
- Snapshots intelligents : utiles mais à condition de ne pas snapshotter toute la page.
- Test Data Management (TDM) : éviter les jeux de données figés, injecter ou générer les données à la volée.

Maintenir ses tests automatisés à long terme, c'est :

- Penser lisibilité, modularité, robustesse
- Accepter de faire du ménage régulièrement
- Travailler en collaboration avec toutes les parties prenantes
- Et surtout, mesurer ce que les tests apportent réellement : gain de temps, couverture, réduction des anomalies en production.

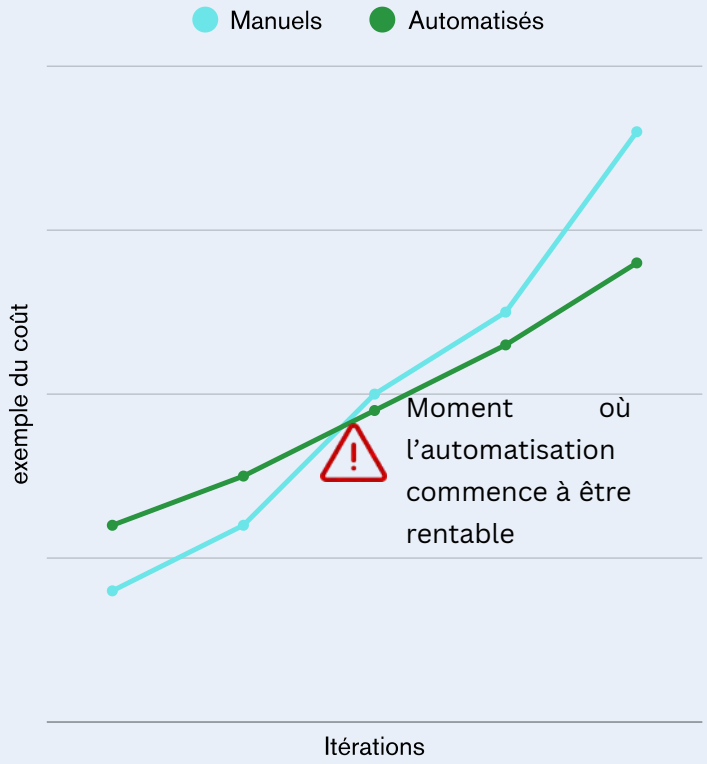
COMMENT CALCULER LE ROI D'UNE CAMPAGNE D'AUTOMATISATION ?

Votre premier réflexe est de sortir votre calculette pour donner des chiffres à vos responsables? C'était mon reflexe également mais il y a tellement plus d'avantage que juste un chiffre. Posez vous quelques questions qui s'appliquent à votre projet pour appuyer sur l'avantage de l'automatisation :

- Quelles suites veut-on automatiser ? (ex. régression, fumée, performances) : le volume et la fréquence d'exécution conditionnent l'économie potentielle.

- Fréquence d'exécution ? (par build, par jour, par sprint) : plus la cadence est élevée, plus les gains se cumulent vite.

- Horizon temporel ? (6 mois, 12 mois, 3 ans) : l'automatisation coûte surtout au départ et rapporte surtout sur la durée : choisissez un pas de temps réaliste.



Parlons quand même des chiffres

La formule de base pour calculer un ROI est :

$$ROI(\%) = \frac{\text{Benefices totaux} - \text{Couts totaux}}{\text{Couts totaux}} \times 100$$

Prenons un exemple concret

Poste	Montant (€)
Coûts	
Licences outil UI + mobile	8 000
Infrastructure CI/CD (cloud)	4 000
Dév. initial (45 j-h à 500 €)	22 500
Maintenance (15 j-h)	7 500
Total coûts	**42 000**
Bénéfices	
Régression manuelle : 31h (35h - 4h nécessaire pour la supervision sur la pipeline CI/CD)	
Builds/an	30
Économie main-d'œuvre (31 h × 30 × 70 €/h)	65100
Déploiement plus rapide (1 jour gagné × 10 releases × 3 000 €/jour)	30 000
Défauts prod évités (3 bugs bloquants × 5 000 €)	15 000
Total bénéfices	**110100**

Le résultat est :

$$ROI(\%) = \frac{110100 - 42000}{42000} \times 100$$

Soit : 162%

Calculons le seuil de rentabilité

$$ROI(\%) = \frac{\text{Benefices totaux} - \text{Couts totaux}}{\text{Couts totaux}} \times 100$$

Dans notre cas :

$$ROI(\%) = \frac{\text{Benefices totaux} - \text{Couts totaux}}{\text{Couts totaux}} \times 100$$

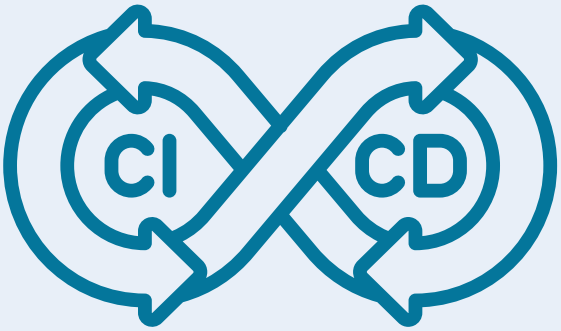
Soit : 4,6 mois

Dans ce cas précis, l'automatisation sera rentable au bout de 4,6 mois passé.

> Un testeur manuel effectue des tests 8 heures par jour avant de rentrer chez lui. À ce moment-là, les tests s'arrêtent. [...]
>
> Si nous automatisons les tests [...], nous exécuterons cinq fois plus de tests par heure. [...]
>
> Les défauts de code détectés après la mise en production coûtent cinq fois plus cher à corriger que ceux détectés lors des tests unitaires.

Sofia Palamarchuk, « The True ROI of Test Automation », blog d'Abstracta, 31 août 2015

CI/CD



Comme nous l'avons vu, les tests doivent être déclenchés automatiquement à chaque push ou merge, via GitLab CI, GitHub Actions, Jenkins ou autre pour maintenir correctement ses tests automatisés sur le long terme.

Concevoir une pipeline

Une chaîne CI/CD efficace suit un découpage par étapes indépendantes :

- Build & static scan : compilation, linters, SCA.
- Tests unitaires parallélisés : exécution en multi-processus afin d'obtenir un premier retour en moins de deux minutes.
- Packaging / Artefacts : images OCI, paquets NPM, libraries Maven.
- Tests d'intégration & e2e : Cypress ou Playwright sur un Selenium Grid ou un provider cloud (BrowserStack, Sauce Labs).
- Analyse de couverture & mutation.
- Vérifications de sécurité (SAST, secrets scan, IaC).

Raccourcir le feedback grâce au Test Impact Analysis (TIA)

L'exécution exhaustive de milliers de tests à chaque commit devient vite prohibitive. Le TIA identifie, via la couverture et la dépendance de code, le sous-ensemble significatif de scénarios à relancer. Les gains constatés peuvent allé jusqu'à presque 50 % de temps économisé sur la fenêtre build-test. Azure Pipelines propose la case "Run only impacted tests" depuis la version 2 du Visual Studio Test task, sans configuration supplémentaire. Vous pouvez le voir la page blog de https://devblogs.microsoft.com/devops/accelerated-continuous-testing-with-test-impact-analysis-part-2/

Nous allons créer un fichier GitHub Actions qui se déclenche :

- lors d'un push sur main, ou,
- à chaque pull-request, tous les jours à 21 h (heure de Paris) ou,
- via le bouton Run workflow

Veillez cependant, à ne jamais lancer deux exécutions simultanées sur la même branche.

Une fois démarré, le script installe les dépendances Node 20, exécute nos tests Cypress dans le navigateur demandé (sans passer par Cypress Cloud) puis archive automatiquement les vidéos, captures d'écran et rapports générés.

Secrets ou variables :

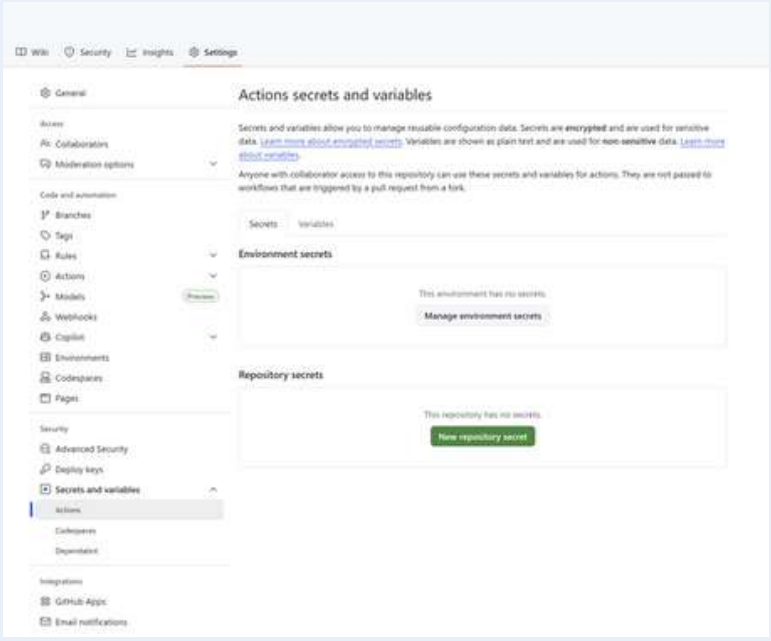
Vous pouvez utiliser des variables et des secrets. Ce n'est pas le cas dans l'exemple que nous avons crée.

Cependant, se peut être votre besoin si vous avez Cypress Cloud pour exécuter vos tests en parallèle et y mettre votre clé Cypress Cloud.

Le code sera donc :

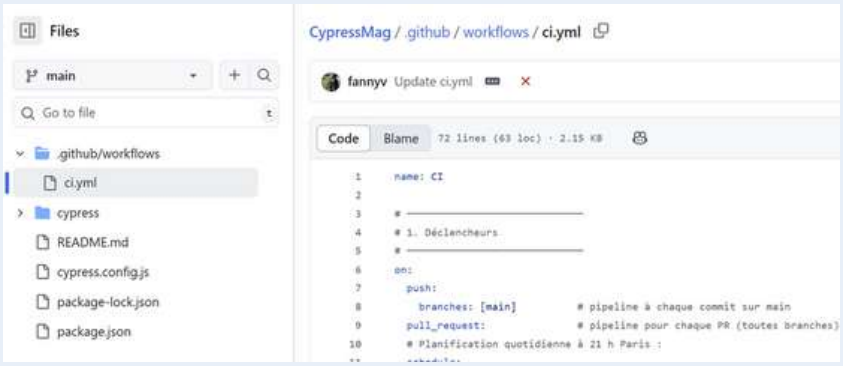
```
- name: Run Cypress
    uses: cypress-io/github-action@v6.8.0
    with:
      record: true
      parallel: true
      browser: ${{ inputs.browser || 'chrome' }}
    env:
      CYPRESS_RECORD_KEY: ${{ secrets.CYPRESS_RECORD_KEY }}
      SAUCE_USERNAME:     ${{ secrets.SAUCE_USERNAME }}
      SAUCE_ACCESS_KEY:   ${{ secrets.SAUCE_ACCESS_KEY }}
```

Vous pouvez définir vos variables et secrets dans github directement. Allez dans les settings de votre projet puis dans Secrets and variables :



Cela permettra à github de remplacer les variables dans vos scripts.

Créez un fichier ci.yml dans le dossier .github/workflows :



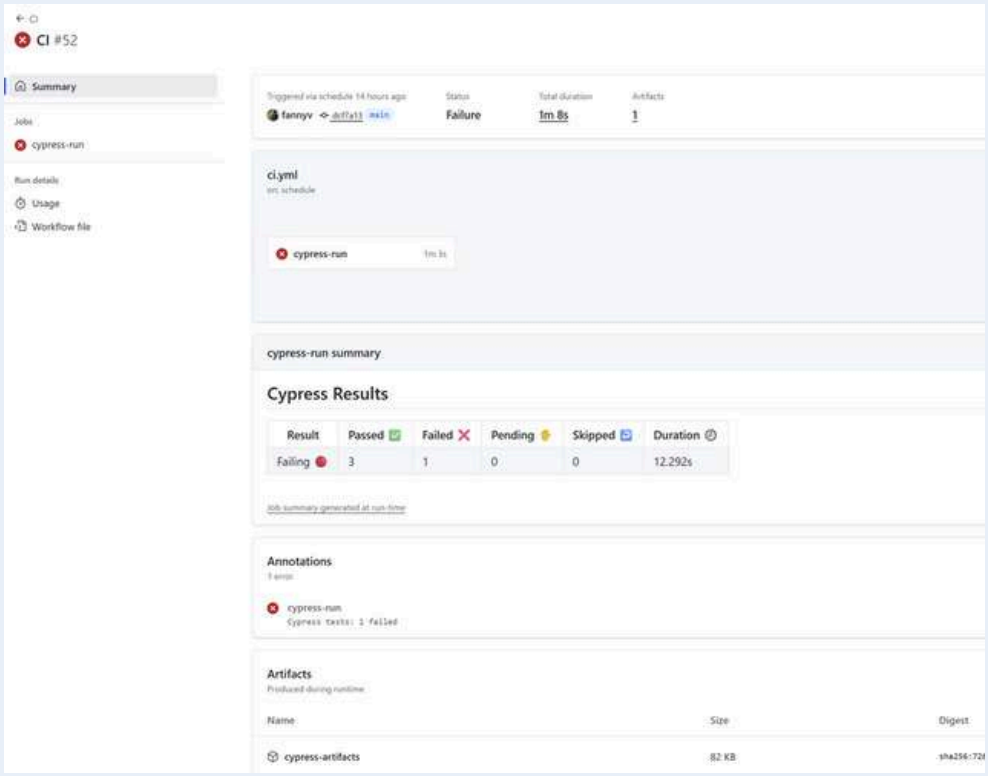
Voyons à présent le code à écrire, vous le retrouverez également sur github avec le QR Code :

```yaml
name: CI

on:
  push:
    branches: [main]        # pipeline à chaque commit sur main
  pull_request:             # pipeline pour chaque PR (toutes branches)

  schedule:
    - cron:  '0 19 * * *'     # 21 h CEST

  workflow_dispatch:
    inputs:
      browser:
        description: "Navigateur : chrome, firefox, edge…"
        required: false
        default: "chrome"

concurrency:
  group: ci-${{ github.ref }}
  cancel-in-progress: true
```

```yaml
jobs:
  cypress-run:
    runs-on: ubuntu-latest
    defaults:
      run:
        shell: bash

    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Node 20
        uses: actions/setup-node@v4
        with:
          node-version: 20
          cache: 'npm'

      - name: Install dependencies
        run: npm ci

      - name: Run Cypress
        uses: cypress-io/github-action@v6.8.0
        with:
          browser: ${{ inputs.browser || 'chrome' }}

      - name: Upload artefacts (vidéos, screenshots)
        if: always()
        uses: actions/upload-artifact@v4
        with:
          name: cypress-artifacts
          path: |
            cypress/videos
            cypress/screenshots
            cypress/results
```

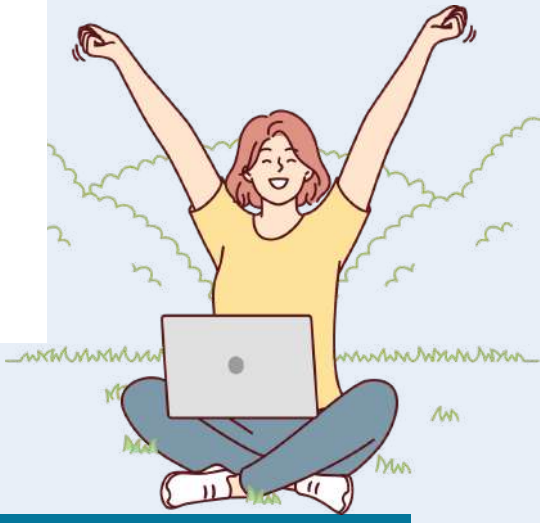
Après un premier run (ici 52...), le résultat est :

- 3 tests sont ok
- 1 est en échec (c'est attendu, c'est l'anomalie que nous attendions)
- les tests ont duré 12,292s



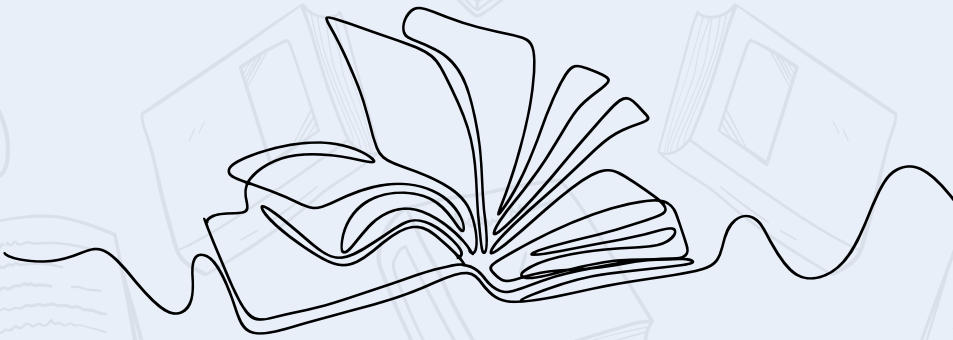
Si vous cliquez sur les artifacts, vous allez télécharger un zip qui contient toutes les copies d'écrans des tests en échec. Ici :

Et les deux emails envoient alors :



JEU CONCOURS

DEFI DU TESTEUR



Tentez de remporter un exemplaire du livre “Automatisez vos tests avec Cypress” !

5 dessins sont cachés dans le magazine (en plus de celle-ci), à vous de les trouver et d’indiquer les numéros des pages où ils se trouvent

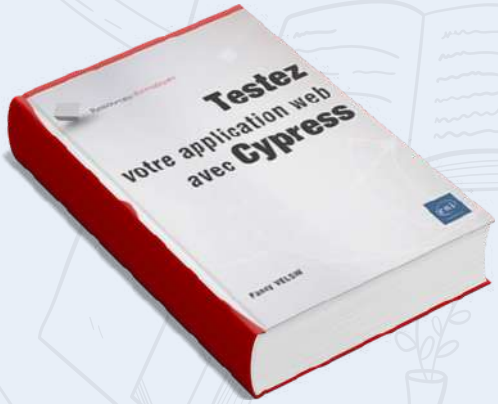
Image à trouver !

Comment participer ?

Envoyez votre réponse par email avant le 31 aout 2025 à 23h59 à : feedback@lemagdutesteur.fr

Lot à gagner :

2 exemplaires du livre “Testez votre application web avec Cypress” (valeur : 32 €) d’édition ENI.



- Le règlement complet est consultable ici ou à la demande par email:

- Concours gratuit – Une seule participation par personne – Données supprimées à l’issue du jeu

- Les gagnants seront contactés par mail

PODCAST DE NANCAIDAH TOURE CHAUVIN

Qalisty est un podcast francophone dédié aux métiers du test et de la qualité logicielle, créé et animé par Nancaidah Touré-Chauvin, actuellement Test Lead dans une entreprise de services numériques.

Le podcast vise à vulgariser les questions liées aux métiers du test dans le secteur numérique, en proposant des échanges avec des invités variés pour offrir une vision complète et actuelle du domaine.

Objectifs du podcast :

- Démystifier le métier de testeur logiciel et les différentes facettes de la qualité logicielle.
- Partager des expériences et des parcours professionnels inspirants.
- Informer sur les tendances, les outils et les bonnes pratiques en matière de test et de qualité logicielle.

Quelques épisodes notables :

- S3 - #5 - Qualité et mentorat : l'impact d'OpenClassrooms sur les testeurs de demain avec David Rochelet. Il évoque l'importance du mentorat et de la manière dont OpenClassroom contribue à préparer ces professionnels aux défis actuels de notre secteur.
- S3 - #3 - Paris Test Conf 2025 : quand un évènement incontournable de la Qualité devient une association. Nancaidah reçoit quatre invités : Anne-Laure Gaillard, Benjamin Butel, Simon Garny et Thomas Facquet pour une discussion sur l'évolution de la Paris Test Conf en association, avec les implications pour la communauté du test logiciel.
- S3 - #1 - IA et qualité logicielle : les impacts réels sur nos métiers avec Bruno Legeard. Cet épisode offre une perspective approfondie sur la manière dont l'IA redéfinit les compétences attendues des testeurs et l'évolution des méthodologies de test.

Disponibilité : Qalisty est disponible sur plusieurs plateformes de podcast, notamment :

- Spotify :
- Apple Podcasts :

TEASING

Partagez-nous vos retours !

Ce magazine a été conçu pour vous accompagner dans l'amélioration de vos tests. Vos impressions, idées et questions sont précieuses pour que Le Mag du Testeur continue de grandir et de s'adapter à vos besoins.

N'hésitez pas à m'écrire à **feedback@lemagdutesteur.fr** pour échanger autour de vos expériences et de vos suggestions !

Un avant-goût du prochain numéro...
Qu'en est-il de l'automatisation des tests mobiles : Appium vs Detox

Merci pour votre fidélité et à très bientôt dans les pages de Le Mag du Testeur !