

GUIDE COMPLET

DU TEST AUTOMATISE

Méthodes, stratégie, cas pratiques et outils pour débiter

Guide pédagogique enrichi • Version 1.1 • Octobre 2026

Objectif : acquérir une démarche de test automatisé, reproductible et orientée qualité.

Sommaire

Introduction.....	4
À qui s'adresse ce guide ?.....	4
Comment l'utiliser ?	4
Partie 1 : Fondamentaux du test automatisé.....	5
1.1 Définition et objectifs	5
1.2 Niveaux et types de tests automatisables.....	6
1.3 Techniques de sélection des données de test.....	6
1.4 La pyramide d'automatisation et la rentabilité.....	8
1.5 Anatomie d'un test automatisé.....	8
1.6 Vocabulaire essentiel.....	9
Schéma récapitulatif de la partie 1.....	10
Partie 2 : Stratégie et méthodologie d'automatisation.....	11
2.1 Le processus de test automatisé en un coup d'œil	11
2.2 Analyser les exigences et la testabilité.....	11
2.3 Élaborer une stratégie d'automatisation.....	12
2.4 Méthodologie de conception des cas de test automatisés.....	13
2.5 Architecture d'un framework de test.....	14
2.6 Le cahier de recette automatisé.....	15
2.7 Exécuter les tests et gérer les anomalies	15
2.8 Maîtriser les tests instables (flaky).....	16
2.9 Intégration continue et exécution planifiée	17
2.10 Mesurer et rendre compte.....	18
2.11 Non-régression, environnements et données de test.....	18
Schéma récapitulatif de la partie 2.....	19
Partie 3 : Outils open source pour le test automatisé.....	20
3.1 Outils par catégorie	20
3.2 Focus : Playwright, Cypress et Selenium	20
3.3 Automatisation mobile : Appium ou Maestro ?.....	21
3.4 Automatisation d'API.....	21
3.5 BDD avec Cucumber et Gherkin	21
3.6 Rapports et intégration continue	21
3.7 Quand automatiser, et quand ne pas le faire.....	22
Schéma récapitulatif de la partie 3.....	23
Partie 4 : Trois cas pratiques d'automatisation avec cahiers de recette.....	24

Cas 1 : test automatisé d'un logiciel web, formulaire d'inscription	24
Cas 2 : test automatisé d'une application mobile, écran de connexion (Android / iOS)	28
Cas 3 : test automatisé d'un produit, terminal de paiement	31
Schéma récapitulatif de la partie 4.....	34
Partie 5 : Conseils pour réussir en tant qu'automaticien junior.....	35
5.1 Les réflexes à acquérir	35
5.2 Erreurs fréquentes à éviter.....	35
5.3 Outils indispensables pour débiter	35
5.4 Les qualités d'un bon automaticien	36
5.5 Plan de progression sur 90 jours	36
5.6 Pour aller plus loin	36
Schéma récapitulatif de la partie 5.....	37
Annexes : modèles prêts à l'emploi.....	38
Annexe A : modèle de cas de test automatisé	38
Annexe B : modèle de rapport d'anomalie.....	38
Annexe C : checklist avant d'automatiser un test	38
Annexe D : checklist de relecture d'un script.....	39
Annexe E : aide-mémoire des commandes	39
Conclusion	40

Introduction

Automatiser les tests, c'est confier à un programme l'exécution des vérifications que le testeur réalisait à la main : ouvrir l'application, saisir des données, déclencher une action et comparer le résultat obtenu au résultat attendu. Bien menée, l'automatisation accélère les livraisons, sécurise la non-régression et libère du temps pour les tests où l'intelligence humaine est irremplaçable (exploration, ergonomie, cas inattendus). Mal menée, elle produit des scripts fragiles, coûteux à maintenir et auxquels plus personne ne fait confiance.

Ce guide reprend la démarche du test fonctionnel (analyse, stratégie, conception des cas, cahier de recette, gestion des anomalies) et l'adapte à l'automatisation : quoi automatiser, comment structurer le code, comment brancher les tests dans une chaîne d'intégration continue, avec quels outils open source. Trois cas pratiques complets (application web, application mobile, produit physique) servent de fil rouge, avec du code prêt à adapter.

À qui s'adresse ce guide ?

- Aux testeurs manuels qui veulent évoluer vers l'automatisation et ont besoin d'une méthode avant de choisir un outil.
- Aux développeurs et analystes qui souhaitent écrire des tests automatisés lisibles et durables.
- Aux testeurs juniors qui préparent leurs premiers projets d'automatisation ou un entretien.

Comment l'utiliser ?

Votre objectif	Lisez en priorité
Comprendre les bases de l'automatisation	Partie 1 (fondamentaux) et Partie 5 (réflexes et pièges)
Décider quoi automatiser	Parties 1.4 (rentabilité) et 2.3 (stratégie)
Structurer un projet de tests	Parties 2.4 à 2.6 (conception, framework, cahier de recette)
Brancher les tests dans la CI	Parties 2.9 et 3.6
Choisir ses outils	Partie 3
S'entraîner sur du concret	Partie 4 (trois cas pratiques) puis les annexes

À RETENIR

Un test automatisé n'est pas un test manuel « qui va plus vite » : c'est un **programme** qui doit être conçu, relu, versionné et maintenu comme tout autre code.

Règle d'or : **ne jamais automatiser un test dont on ne maîtrise pas la version manuelle**. L'automatisation reproduit une démarche de test, elle ne la remplace pas.

Partie 1 : Fondamentaux du test automatisé

1.1 Définition et objectifs

Un test automatisé est un script qui exécute des actions sur le produit (interface, API, service) puis **vérifie automatiquement** les résultats par des assertions, sans intervention humaine. Il suit la logique « boîte noire » du test fonctionnel : entrées, traitement ignoré, sorties comparées aux attentes.

- **Entrées** : données et actions pilotées par le script (saisies, clics, requêtes, gestes).
- **Traitement** : ignoré, on observe le produit de l'extérieur.
- **Sorties** : comparées par des assertions (URL, texte, code de réponse, donnée en base).

Objectifs principaux :

- Rejouer rapidement et à l'identique les tests de non-régression à chaque modification.
- Obtenir un retour en minutes (et non en jours) après chaque livraison.
- Couvrir davantage de combinaisons (navigateurs, jeux de données, appareils) que ne le permettrait le manuel.
- Fiabiliser la décision de mise en production grâce à des résultats reproductibles et tracés.
- Libérer les testeurs pour l'exploratoire et l'analyse de risque.

Bénéfices et limites

Bénéfices	Limites à connaître
Rapidité et répétabilité : même exécution à chaque fois	Coût initial de développement des scripts
Exécution nocturne ou à chaque commit, sans mobiliser de testeur	Maintenance continue : chaque évolution de l'interface peut casser des tests
Parallélisation sur plusieurs navigateurs ou appareils	Un script ne remarque que ce qu'on lui demande de vérifier
Traçabilité : rapports, captures, vidéos, traces	Ne détecte pas les défauts d'ergonomie ou de ressenti utilisateur
Données multiples avec un seul script (data-driven)	Mal conçu, il génère des échecs aléatoires (tests instables)

Vérification et validation

L'automatisation sert surtout la **vérification** (« le produit est-il conforme à la spécification ? »). La **validation** (« est-ce le bon produit pour l'utilisateur ? ») reste largement une affaire d'observation humaine : démonstrations, tests exploratoires, retours des utilisateurs.

1.2 Niveaux et types de tests automatisables

Les niveaux de test

Niveau	Cible	Qui	Outils typiques
Unitaire	Une fonction ou un composant isolé	Développeur	JUnit, pytest, Jest
Intégration / API	Interactions entre composants et services	Développeur / testeur	Postman, Rest-Assured, Karate, Playwright (API)
Système / interface (E2E)	Le produit complet via son interface	Testeur automatique	Playwright, Cypress, Selenium, Appium, Maestro
Acceptation (recette)	Conformité au besoin métier, souvent en BDD	Client / testeur / analyste	Cucumber, Behave, SpecFlow

Les principaux types de tests automatisés

Type	Objectif	Fréquence d'exécution
Smoke test	Vérifier en quelques minutes que les fonctions vitales marchent	À chaque build ou commit
Test fonctionnel	Vérifier les règles métier (cas passants, cas d'erreur, cas limites)	À chaque livraison de fonctionnalité
Non-régression (TNR)	S'assurer que l'existant fonctionne toujours	Quotidienne (nuit) et avant chaque version
Data-driven	Rejouer un même scénario avec de nombreux jeux de données	Intégré aux suites fonctionnelles
Compatibilité	Même scénario sur plusieurs navigateurs, appareils, versions d'OS	Avant chaque version majeure
Non fonctionnel	Performance, sécurité, accessibilité (contrôles automatisables en partie)	Selon les exigences

À RETENIR

Le test **exploratoire** et l'évaluation de l'ergonomie restent manuels. Une bonne stratégie combine les deux : l'automatisation protège l'existant, l'humain explore l'inconnu.

1.3 Techniques de sélection des données de test

Tester toutes les combinaisons est impossible, y compris en automatisation : un script rapide coûte tout de même du temps d'exécution et de maintenance. Les techniques de conception restent donc la base pour choisir **peu de cas, bien ciblés**. Leur grand avantage en automatisation : une fois la technique appliquée, le jeu de données se range dans un tableau que le script parcourt (test piloté par les données).

A. Partitionnement d'équivalence

On divise les entrées en classes censées produire le même comportement et on teste un représentant par classe.

EXEMPLE

Champ « âge » de 18 à 65 ans : classe invalide basse (10), classe valide (40), classe invalide haute (80).
On ajoute les classes techniques : vide, texte à la place d'un nombre, nombre négatif, décimal.

B. Analyse des valeurs limites

Règle à tester	Bornes	Valeurs à tester	Résultat attendu
Âge de 18 à 65 ans	18 et 65	17 · 18 · 19 · 64 · 65 · 66	17 et 66 refusés ; les autres acceptés
Mot de passe de 8 à 64 caractères	8 et 64	7 · 8 · 9 · 63 · 64 · 65	7 et 65 refusés ; les autres acceptés
Quantité commandée de 1 à 99	1 et 99	0 · 1 · 2 · 98 · 99 · 100	0 et 100 refusés ; les autres acceptés

C. Table de décision

Pour une règle de connexion (compte existant, mot de passe correct, compte non verrouillé), chaque ligne de la table devient un cas automatisé :

Règle	Compte existe	Mot de passe correct	Compte verrouillé	Action attendue
R1	Oui	Oui	Non	Connexion réussie, accès à l'accueil
R2	Oui	Non	Non	Message « Identifiants incorrects », compteur + 1
R3	Oui	Oui	Oui	Message « Compte verrouillé », pas de connexion
R4	Non	—	—	Message générique « Identifiants incorrects »

D. Transitions d'état et E. Test combinatoire (Pairwise)

Pour un objet qui change d'état (commande : brouillon, validée, payée, expédiée), on automatise les transitions **valides** et on vérifie que les transitions **interdites** sont refusées. Pour plusieurs paramètres qui interagissent, le **pairwise** réduit 27 combinaisons (3 navigateurs × 3 systèmes × 3 langues) à 9 cas : un outil comme PICT les génère et le résultat alimente directement la matrice de projets du framework.

F. Les données de test dans le code : le test piloté par les données

```
// Les cas issus des valeurs limites sont rangés dans un tableau
const longueursMotDePasse = [
  { n: 7, ok: false }, { n: 8, ok: true }, { n: 9, ok: true },
  { n: 64, ok: true }, { n: 65, ok: false },
];

for (const { n, ok } of longueursMotDePasse) {
  test(`mot de passe de ${n} caractères : ${ok ? 'accepté' : 'refusé'}`, async ({ page }) => {
    // un seul script, cinq cas exécutés
  });
}
```

ASTUCE

Avant d'écrire une ligne de code, demandez-vous : « **Qu'est-ce qui pourrait mal se passer ici ?** ». Un script ne trouvera jamais un cas auquel personne n'a pensé.

1.4 La pyramide d'automatisation et la rentabilité

Dans une approche agile, la pyramide de Mike Cohn recommande beaucoup de tests unitaires, un nombre moyen de tests d'API et de services, et peu de tests d'interface de bout en bout. Les tests d'interface sont les plus lents, les plus fragiles et les plus chers à maintenir.

Niveau de la pyramide	Volume conseillé	Vitesse	Coût de maintenance
Tests d'interface de bout en bout (E2E)	Peu	Lents	Élevé
Tests d'API et de services	Moyennement	Moyens	Moyen
Tests unitaires	Beaucoup	Très rapides	Faible

Règle pratique : **tester le plus bas possible**. Une règle de gestion (par exemple « email déjà utilisé ») se vérifie par un test d'API en quelques millisecondes ; on garde un seul test d'interface pour vérifier que le message s'affiche.

Calculer la rentabilité d'un test automatisé

Un test automatisé est rentable lorsque le temps économisé dépasse son coût de création et de maintenance :

```
Gain = (nb d'exécutions × durée du test manuel)
      - (temps de développement + temps de maintenance
        + nb d'exécutions × durée de supervision de l'exécution automatique)
```

EXEMPLE

Un test manuel de 15 minutes est exécuté à chaque version. L'automatiser coûte 2 h de développement et 1 h de maintenance par an. Seuil de rentabilité : $180 \text{ min} \div 15 \text{ min} = 12$ **exécutions**. Avec une exécution par version et une version par semaine, le test est rentabilisé en trois mois.

1.5 Anatomie d'un test automatisé

Tout bon test automatisé suit la structure **Arrange, Act, Assert** (préparer, agir, vérifier) :

Étape	Rôle	Exemple (inscription)
Arrange (préparer)	Créer l'état initial : données, compte, page ouverte	Générer un email unique, ouvrir /register
Act (agir)	Effectuer l'action testée	Remplir le formulaire et cliquer sur « S'inscrire »
Assert (vérifier)	Comparer le résultat observé au résultat attendu	L'URL contient /dashboard et le message de bienvenue est visible
Cleanup (nettoyer)	Remettre l'environnement dans l'état initial	Supprimer le compte créé via l'API

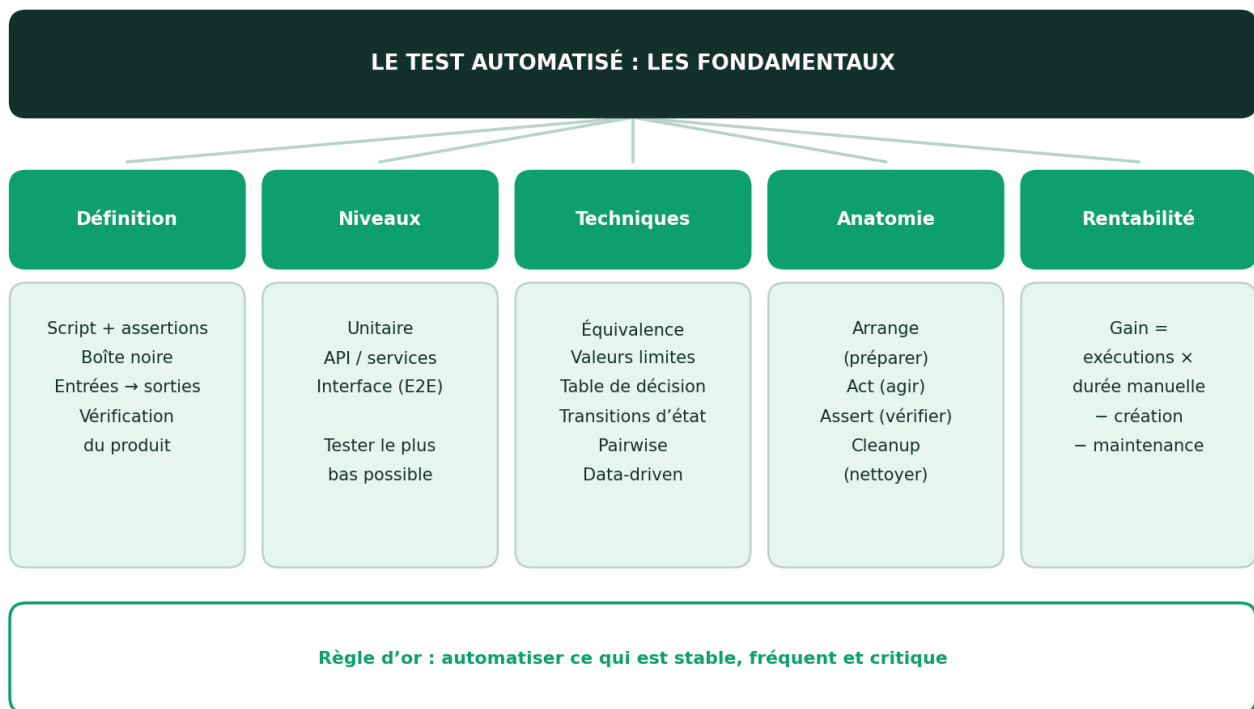
Les éléments clés d'un script

- **Localisateurs (locators)** : la manière de retrouver un élément. Préférer les rôles, libellés et attributs de test (`data-testid`) aux chemins CSS fragiles.
- **Assertions** : le cœur du test. Un script sans assertion n'est pas un test.
- **Attentes (waits)** : attendre qu'une condition soit vraie plutôt que d'attendre un temps fixe.
- **Fixtures** : préparation et nettoyage réutilisables (connexion, jeu de données, navigateur).
- **Isolation** : chaque test doit pouvoir s'exécuter seul et dans n'importe quel ordre.

1.6 Vocabulaire essentiel

Terme	Définition
Assertion	Vérification automatique qui fait échouer le test si la condition n'est pas remplie
Locator / sélecteur	Expression qui désigne un élément de l'interface à manipuler
Fixture	Mécanisme de préparation et de nettoyage partagé entre plusieurs tests
Page Object (POM)	Classe qui regroupe les éléments et actions d'une page pour éviter la duplication
Test instable (flaky)	Test qui réussit ou échoue de façon aléatoire sans changement du produit
Headless	Exécution d'un navigateur sans fenêtre visible, typique des environnements CI
Mock / stub	Substitut simulé d'un service externe pour rendre un test déterministe
Runner	Programme qui exécute les tests et produit le rapport (Playwright Test, pytest, Mocha...)
Intégration continue (CI)	Exécution automatique des tests à chaque modification du code
Pipeline	Suite d'étapes automatisées (build, tests, rapports, déploiement)
Trace	Enregistrement détaillé d'une exécution (actions, captures, réseau) pour analyser un échec
TNR	Tests de non-régression
BDD	Développement piloté par le comportement : scénarios en langage naturel (Gherkin) rendus exécutables

Schéma récapitulatif de la partie 1



Partie 2 : Stratégie et méthodologie d'automatisation

2.1 Le processus de test automatisé en un coup d'œil

1. **Planification** : définir périmètre, ressources, calendrier, risques et outils.
2. **Analyse** : étudier les exigences et repérer ce qui est automatisable.
3. **Conception** : écrire les cas de test, préparer les données et concevoir l'architecture du framework.
4. **Mise en place de l'environnement** : installer le framework, les navigateurs ou émulateurs, les comptes et jeux de données.
5. **Exécution** : lancer les scripts en local puis en intégration continue, collecter les rapports.
6. **Suivi et maintenance** : analyser les échecs, corriger les anomalies ou les scripts, retester.
7. **Clôture** : rapport de synthèse, indicateurs, retour d'expérience et plan d'amélioration.

En agile, ces activités se répètent à chaque itération. L'automaticien est intégré à l'équipe dès la rédaction des user stories, afin de prévoir en même temps la testabilité et les scripts.

2.2 Analyser les exigences et la testabilité

L'automatisation exige des exigences encore plus précises que le test manuel : un script ne sait pas interpréter une consigne floue.

Exigence mal formulée	Problème	Exigence testable automatiquement
« L'application doit être rapide »	Non mesurable	« La page d'accueil s'affiche en moins de 2 secondes (95e centile) »
« Le mot de passe doit être sécurisé »	Flou	« 8 à 64 caractères, une majuscule, un chiffre, un caractère spécial »
« Un message d'erreur apparaît »	Texte et emplacement non définis	« Le message exact "Cet email est déjà utilisé" s'affiche dans un élément de rôle alert »

Rendre le produit testable

- Demander des **attributs de test stables** (`data-testid`, identifiants d'accessibilité mobile) sur les éléments clés.
- Disposer d'**API de préparation et de nettoyage** (créer un compte, supprimer un utilisateur, réinitialiser des données).
- Prévoir des **comptes et jeux de données dédiés** aux tests, distincts des vrais comptes.
- Obtenir un **environnement stable**, déployé de façon reproductible, avec des services externes simulables.

ASTUCE

Quelques minutes passées à négocier avec les développeurs un `data-testid` ou une route de nettoyage évitent des semaines de maintenance de scripts fragiles.

2.3 Élaborer une stratégie d'automatisation

La stratégie définit **quoi automatiser, comment, quand et avec quelles ressources**. Elle reprend les éléments de toute stratégie de test, avec des questions propres à l'automatisation :

Élément	Questions à se poser	Exemple
Périmètre	Quelles fonctionnalités automatiser ? Lesquelles rester en manuel ?	Inscription et connexion automatisées, exploration manuelle de l'ergonomie
Types de tests	Smoke, TNR, API, interface, compatibilité ?	Smoke à chaque commit, TNR chaque nuit
Environnements	Où s'exécutent les tests ? Avec quelles données ?	Intégration en CI, données fictives réinitialisées
Critères d'entrée	Quand peut-on lancer la campagne ?	Build déployé, smoke test vert, jeu de données prêt
Critères de sortie	Quand peut-on livrer ?	100 % des tests P1 verts, aucune anomalie bloquante ou majeure ouverte
Gestion des risques	Quelles zones sont les plus critiques ?	Paiement et authentification en priorité
Outils et ressources	Quelles compétences, quel outil, quel budget ?	Playwright, 1 automaticien à mi-temps, CI existante

Prioriser selon le risque

Impact \ Probabilité	Faible	Moyenne	Élevée
Impact élevé	P2	P1	P1
Impact moyen	P3	P2	P1
Impact faible	P3	P3	P2

- **P1 (critique)** : à automatiser en premier et à surveiller chaque jour.
- **P2 (important)** : à automatiser une fois le socle P1 stable.
- **P3 (secondaire)** : automatisation optionnelle, souvent laissée en manuel.

Choisir ce qui vaut la peine d'être automatisé

Donnez une note de 1 à 3 à chaque critère pour un test candidat. Plus le total est élevé, plus l'automatisation est justifiée :

Critère	1 point	3 points
Fréquence d'exécution	Une fois ou rarement	À chaque version ou chaque jour
Stabilité de la fonctionnalité	Change tout le temps	Stable depuis plusieurs versions
Criticité métier	Fonction secondaire	Fonction critique (connexion, paiement)
Déterminisme	Résultat dépend de facteurs incontrôlés	Résultat entièrement prévisible
Effort de scripting	Très élevé	Faible

- **12 à 15 points** : automatiser en priorité.

- **8 à 11 points** : automatiser si le contexte s'y prête.
- **moins de 8 points** : garder en manuel.

2.4 Méthodologie de conception des cas de test automatisés

La conception d'un cas automatisé commence comme celle d'un cas manuel, avec un champ de plus : le lien vers le script.

Élément	Description
ID	Identifiant unique (ex : TC-WEB-001), repris dans le nom du test pour la traçabilité
Scénario	Brève description de ce qui est testé
Exigence liée	Référence de l'exigence ou de la user story couverte
Préconditions	État initial requis, préparé par le script (fixture ou API)
Étapes	Actions automatisées, une action par étape
Données de test	Valeurs d'entrée, de préférence externalisées et générées de façon unique
Assertions attendues	Vérifications précises et observables
Script associé	Fichier et nom du test dans le dépôt
Statut	Pass / Fail / Flaky / Non exécuté
Priorité	P1, P2, P3

Bonnes pratiques :

- Un test = un objectif. Un test qui vérifie dix choses ne dit pas laquelle est cassée.
- Des tests **indépendants** : jamais de dépendance d'ordre entre scripts.
- Des **données maîtrisées** : générées à chaque exécution (email unique) ou réinitialisées.
- Des noms de tests lisibles comme des phrases : `TC-WEB-002 : inscription refusée si l'email est déjà utilisé`.
- Des assertions précises, sur ce que l'utilisateur voit réellement.
- Des **attentes conditionnelles**, jamais de pause fixe.
- Aucune logique complexe (boucles, conditions imbriquées) dans un test : si le test est difficile à lire, il est trop compliqué.

Exemple : mauvais test vs bon test

```
// MAUVAIS : pause fixe, sélecteur fragile, aucune assertion utile
test('login', async ({ page }) => {
  await page.goto('/login');
  await page.click('div > div:nth-child(3) > input');
  await page.waitForTimeout(5000);
  await page.click('.btn');
});

// BON : nom explicite, rôles et libellés, assertions précises
test('TC-WEB-010 : connexion refusée avec un mot de passe erroné', async ({ page }) => {
  await page.goto('/login');
```

```
await page.getByLabel('Email').fill('marie@example.com');
await page.getByLabel('Mot de passe').fill('Faux123!');
await page.getByRole('button', { name: 'Se connecter' }).click();
await expect(page.getByRole('alert')).toHaveText('Identifiants incorrects');
await expect(page).toHaveURL(/login/);
});
```

Les trois familles de cas à prévoir pour chaque fonction

- **Cas passants (nominal)** : l'utilisation normale attendue.
- **Cas non passants (erreur)** : données invalides, droits insuffisants, service indisponible.
- **Cas limites** : bornes, volumes, caractères particuliers.

2.5 Architecture d'un framework de test

Un projet d'automatisation qui grossit sans structure devient ingérable. On sépare les responsabilités : tests, pages, données, configuration.

```
projet-tests/
├── tests/                # les scénarios (un fichier par fonctionnalité)
│   ├── inscription.spec.ts
│   └── connexion.spec.ts
├── pages/               # Page Objects (éléments + actions)
│   └── RegisterPage.ts
├── fixtures/           # préparation et nettoyage réutilisables
├── data/                # jeux de données (JSON, CSV)
├── utils/               # générateurs (email unique), aides
├── playwright.config.ts # configuration (URL, navigateurs, rapports)
└── .github/workflows/   # pipeline d'intégration continue
```

Le modèle Page Object (POM)

Chaque page de l'application a sa classe, qui contient ses éléments et ses actions. Si l'interface change, on corrige un seul fichier au lieu de vingt tests.

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

export class RegisterPage {
  constructor(private page: Page) {}

  async open() { await this.page.goto('/register'); }

  async fillForm(d: { login: string; email: string; password: string; confirm: string }) {
    await this.page.getByLabel('Identifiant').fill(d.login);
    await this.page.getByLabel('Email').fill(d.email);
    await this.page.getByLabel('Mot de passe', { exact: true }).fill(d.password);
    await this.page.getByLabel('Confirmation').fill(d.confirm);
  }

  async submit() { await this.page.getByRole('button', { name: "S'inscrire" }).click(); }
  alert() { return this.page.getByRole('alert'); }
}
```

ATTENTION

Ne surchargez pas l'architecture dès le premier jour. Commencez par quelques tests simples, puis introduisez Page Objects et fixtures quand la duplication apparaît.

2.6 Le cahier de recette automatisé

Même automatisée, la recette reste documentée : c'est elle qui prouve au client que le produit est conforme. Le cahier de recette comprend :

1. **Page de garde** : projet, version testée, date, auteur, statut du document.
2. **Contexte** : objectifs, périmètre, documents de référence.
3. **Environnement de test** : URL, versions, navigateurs ou appareils, outils et leurs versions, branche du dépôt.
4. **Synthèse des tests** : tableau récapitulatif généré depuis le rapport d'exécution.
5. **Cas de test détaillés** : regroupés par fonctionnalité, avec le script associé à chacun.
6. **Anomalies connues** : bugs non bloquants, tests volontairement ignorés avec justification.
7. **Conclusion** : avis sur la conformité, réserves, recommandation de livraison.

Exemple de tableau de synthèse

Fonctionnalité	Cas automatisés	Passés	Échoués	Instables	Taux de réussite
Inscription	8	7	1	0	87,5 %
Connexion	6	6	0	0	100 %
Mot de passe oublié	5	3	1	1	60 %
Total	19	16	2	1	84,2 %

ASTUCE

Le rapport HTML généré par l'outil (Playwright, Allure) remplace la saisie manuelle des résultats : il sert de preuve d'exécution, à joindre au cahier de recette.

2.7 Exécuter les tests et gérer les anomalies

Bonnes pratiques d'exécution

- Exécuter d'abord en **local** pour écrire et déboguer, puis en **CI** pour la fiabilité.
- Lancer les tests dans l'ordre de priorité : smoke, puis P1, puis le reste.
- Conserver les **preuves** : rapport, captures, vidéos et traces des échecs.
- Noter la version du produit, de l'outil et de l'environnement à chaque campagne.

Un test est rouge : bug, test cassé ou instabilité ?

Diagnostic	Indices	Action
Vrai bug du produit	Échec reproductible à la main, trace cohérente avec le défaut	Déclarer une anomalie avec la trace
Test cassé (maintenance)	Interface ou règle modifiée volontairement, sélecteur introuvable	Mettre à jour le script, éventuellement le Page Object
Test instable (flaky)	Réussit au second essai, dépend du timing ou des données	Corriger la cause, mettre en quarantaine en attendant
Problème d'environnement	Plusieurs tests échouent en même temps, service indisponible	Corriger l'environnement, relancer

Cycle de vie et sévérité d'une anomalie

Nouvelle → Assignée → En cours de correction → Corrigée → À retester (le test automatisé joue ce rôle) → Fermée, ou Rouverte si le test échoue encore.

Sévérité	Définition	Exemple
Bloquante	Empêche d'utiliser une fonction essentielle	Impossible de se connecter
Majeure	Fonction importante dégradée	Le paiement échoue pour une carte sur deux
Mineure	Gêne limitée, contournement simple	Un filtre ne se réinitialise pas
Cosmétique	Défaut d'affichage sans impact fonctionnel	Faute d'orthographe, décalage visuel

Rédiger un bon rapport d'anomalie issu d'un test automatisé

- **Titre** précis et **identifiant du test** en échec (TC-WEB-002).
- **Environnement** : version, navigateur ou appareil, commit, pipeline.
- **Étapes de reproduction** manuelles et minimales.
- **Résultat attendu** versus **résultat obtenu** (message d'assertion).
- **Preuves** : lien vers la trace, capture et vidéo de l'exécution.
- **Fréquence** : systématique ou intermittente (x échecs sur y exécutions).

ERREUR FRÉQUENTE

Déclarer une anomalie sans avoir rejoué le scénario à la main. Si le défaut n'est pas reproductible hors du script, c'est peut-être le test qui est en cause.

2.8 Maîtriser les tests instables (flaky)

Un test instable détruit la confiance dans toute la suite : au bout de quelques faux positifs, plus personne ne regarde les résultats rouges. Le traitement des tests instables est une priorité de maintenance.

Cause fréquente	Symptôme	Remède
Attente fixe ou insuffisante	Échoue quand le serveur est lent	Attentes conditionnelles (auto-waiting, attente d'un élément ou d'une réponse réseau)
Sélecteur fragile	Échoue après un changement d'affichage	Rôles, libellés, data-testid
Données partagées ou polluées	Échoue si un autre test a modifié la donnée	Données uniques par test, nettoyage en fin de test
Dépendance entre tests	Échoue quand l'ordre change	Isolation : chaque test prépare son état
Service externe instable	Échecs aléatoires sur paiement, email, API tierce	Simulation (mock) ou environnement de test dédié
Animations et transitions	Clic sur un élément en mouvement	Attendre la stabilité de l'élément, désactiver les animations en test

ATTENTION

Relancer automatiquement un test échoué (retries) peut masquer une instabilité. Utilisez-le avec parcimonie (1 relance en CI), surveillez les tests qui passent au second essai et corrigez-les.

2.9 Intégration continue et exécution planifiée

Un test automatisé qu'on oublie de lancer ne protège de rien. L'intégration continue exécute les scripts automatiquement, à chaque modification ou chaque nuit, et publie les rapports.

Étape du pipeline	Contenu	Déclencheur
1. Vérification rapide	Analyse statique, tests unitaires	Chaque commit
2. Smoke test	10 à 20 tests critiques, moins de 10 minutes	Chaque commit ou demande de fusion
3. Tests fonctionnels	Suite P1 et P2 en parallèle	Chaque demande de fusion
4. TNR complet	Toutes les suites, plusieurs navigateurs	Chaque nuit
5. Publication	Rapport HTML, traces, notifications à l'équipe	En fin de pipeline, même en cas d'échec

```
name: tests-automatisees
on:
  pull_request:
  schedule:
    - cron: '0 3 * * *' # TNR chaque nuit à 3 h (heure UTC)
jobs:
  e2e:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with: { node-version: 20 }
      - run: npm ci
      - run: npx playwright install --with-deps
      - run: npx playwright test
      - uses: actions/upload-artifact@v4
```

```
if: always()
with: { name: rapport-playwright, path: playwright-report/ }
```

2.10 Mesurer et rendre compte

Indicateur	Formule	Intérêt
Taux d'exécution	Tests exécutés ÷ tests prévus × 100	Avancement de la campagne
Taux de réussite	Tests passés ÷ tests exécutés × 100	Qualité perçue de la version
Taux d'instabilité	Tests instables ÷ tests exécutés × 100	Fiabilité de la suite (cible : moins de 2 %)
Durée de la suite	Temps total d'exécution du pipeline	Rapidité du retour d'information
Couverture des exigences	Exigences couvertes par un test automatisé ÷ total × 100	Complétude de la recette
Taux d'automatisation	Cas automatisés ÷ cas de test total × 100	Maturité de l'automatisation
Temps de correction d'un test cassé	Délai moyen entre échec et réparation	Coût réel de maintenance

ATTENTION

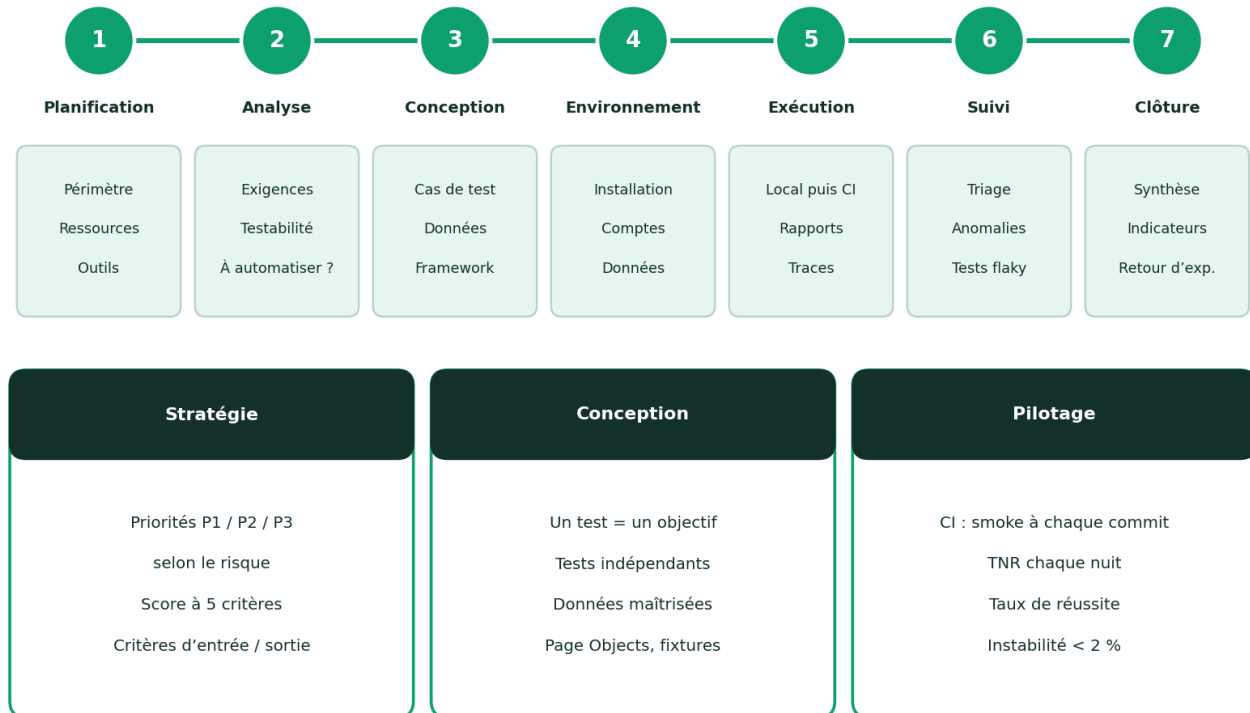
Un taux d'automatisation de 90 % ne garantit pas la qualité : il compte des scripts, pas des risques couverts. Pilotez par la **valeur** des tests, pas par leur nombre.

2.11 Non-régression, environnements et données de test

- **Smoke test** automatisé : 10 à 20 tests ultra-critiques, joués à chaque version en moins de 10 minutes. S'il échoue, la version est refusée.
- **TNR automatisé** : cas P1 et P2 rejoués chaque nuit, avec rapport envoyé à l'équipe.
- **Environnements** : développement, intégration, qualification. Les tests automatisés ne doivent jamais détruire de données en production.
- **Données** : jeux fictifs ou anonymisés (RGPD), jamais de vraies données personnelles.
- **Réinitialisation** : script de remise à zéro ou API de nettoyage pour repartir d'un état connu avant chaque campagne.
- **Services externes** : simuler (mock) ce qui est instable ou payant (SMS, paiement, e-mail).

Schéma récapitulatif de la partie 2

STRATÉGIE ET MÉTHODE : LE PROCESSUS EN 7 ÉTAPES



Partie 3 : Outils open source pour le test automatisé

3.1 Outils par catégorie

Catégorie	Outils recommandés	Usage principal
Web E2E	Playwright, Cypress, Selenium	Automatisation de navigateur
Mobile	Appium, Maestro, Detox	Applications iOS et Android
API	Postman + Newman, Bruno, Rest-Assured, Karate	Services REST et GraphQL
BDD	Cucumber, Behave, SpecFlow	Spécifications exécutables (Gherkin)
Rapports	Rapport HTML Playwright, Allure Report	Résultats, traces, tendances
Intégration continue	GitHub Actions, GitLab CI, Jenkins	Exécution automatique des tests
Données de test	Faker, Mockaroo	Jeux de données réalistes et fictifs
Gestion des tests	TestLink, Kiwi TCMS, tableur	Référentiel de cas et campagnes

3.2 Focus : Playwright, Cypress et Selenium

D'après les enquêtes de la communauté de test citées en 2026, **Playwright** domine le marché avec un taux d'adoption d'environ 45 % et un taux de rétention d'environ 94 % (ordres de grandeur, ils varient selon les sondages). Ses principaux avantages :

- **Auto-waiting** intégré : attend automatiquement que les éléments soient prêts, ce qui réduit fortement les tests instables.
- **Parallélisation native** et gestion des projets multi-navigateurs.
- **Multi-navigateurs** : Chromium, Firefox, WebKit.
- **Multi-langages** : JavaScript/TypeScript, Python, Java, C#.
- **Outils intégrés** : générateur de code (codegen), visionneuse de traces, rapports HTML, tests d'API.

Critère	Playwright	Cypress	Selenium
Langages	JS/TS, Python, Java, C#	JavaScript / TypeScript	Java, Python, C#, JS, Ruby...
Navigateurs	Chromium, Firefox, WebKit	Chromium, Firefox (WebKit expérimental)	Tous via WebDriver
Attentes automatiques	Oui, natives	Oui	Non, à gérer explicitement
Débogage	Trace viewer, mode UI	Interface visuelle très appréciée	Dépend de l'environnement
Prise en main	Facile	Très facile	Plus longue
Cas idéal	Nouveaux projets, multi-navigateurs	Équipes JavaScript, débogage visuel	Suites legacy, large écosystème

Recommandation pour débutant : commencez par Playwright. Lancez `npx playwright codegen`` pour enregistrer vos premières actions, puis nettoyez le code généré (rôles et libellés, assertions) avant de le versionner.

3.3 Automatisation mobile : Appium ou Maestro ?

Critère	Maestro	Appium
Écriture des tests	Fichiers YAML lisibles, sans code	Code (JavaScript, Java, Python...)
Démarrage	Très rapide	Configuration plus lourde (serveur, drivers, capacités)
Puissance	Parcours utilisateur simples à moyens	Très flexible, scénarios complexes
Stabilité	Attentes intégrées	À gérer soigneusement
Pour qui ?	Débutants, parcours standards	Équipes avec compétences de développement

3.4 Automatisation d'API

Beaucoup de règles métier se vérifient plus vite et plus fiablement au niveau API qu'à travers l'interface. Playwright, Postman (avec Newman en ligne de commande), Bruno ou Karate permettent d'envoyer des requêtes et de contrôler la réponse.

Vérification	Exemple
Code HTTP	POST /api/register valide → 201 ; email déjà utilisé → 409
Contenu	Le JSON renvoyé contient l'identifiant et jamais le mot de passe
Contrôle d'accès	Un utilisateur standard appelle une route administrateur → 403
Temps de réponse	Moins de 500 ms pour une connexion

3.5 BDD avec Cucumber et Gherkin

Le BDD décrit les scénarios en langage naturel (Gherkin), compréhensible par le métier, puis les relie à du code. Il facilite le dialogue avec les non-techniciens, mais ajoute une couche à maintenir : à réserver aux équipes qui exploitent réellement ces scénarios comme spécification vivante.

```
# language: fr
Fonctionnalité: Inscription
  Scénario: Inscription refusée si l'email est déjà utilisé
    Étant donné qu'un compte existe avec l'email "marie@example.com"
    Quand je m'inscris avec l'email "marie@example.com"
    Alors je vois le message "Cet email est déjà utilisé"
```

3.6 Rapports et intégration continue

- **Rapport HTML Playwright :** résultats par test, traces et captures jointes aux échecs.

- **Allure Report** : tableau de bord riche (historique, catégories d'échecs, tendances) pour plusieurs outils.
- **GitHub Actions, GitLab CI, Jenkins** : exécution automatique et publication des rapports comme artefacts.
- **Notifications** : message dans l'outil de discussion de l'équipe quand un pipeline passe au rouge.

3.7 Quand automatiser, et quand ne pas le faire

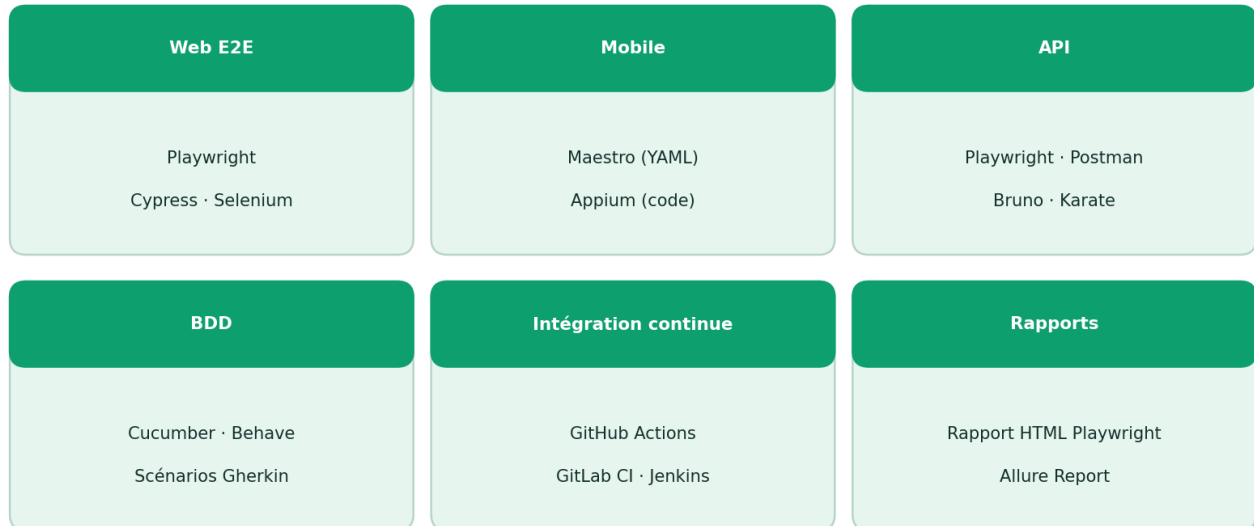
Automatiser	Garder en manuel
Tests de non-régression stables et répétés	Tests exploratoires
Smoke tests exécutés à chaque build	Évaluation de l'ergonomie, du ressenti utilisateur
Parcours critiques (connexion, paiement)	Fonctions qui changent tous les jours
Tests à données multiples (data-driven)	Tests qu'on ne jouera qu'une seule fois
Contrôles de compatibilité répétitifs	Interactions physiques impossibles à simuler (capteurs, manipulation d'un terminal réel)

ATTENTION

Automatiser un test mal conçu donne un test mal conçu qui s'exécute plus vite. **Écrivez et validez d'abord le test manuellement**, puis automatisez-le si le retour sur investissement est clair.

Schéma récapitulatif de la partie 3

LA BOÎTE À OUTILS OPEN SOURCE



Pour débiter : Playwright + Maestro + GitHub Actions

Automatiser : tests stables, fréquents, critiques | Manuel : exploratoire, ergonomie, physique

Partie 4 : Trois cas pratiques d'automatisation avec cahiers de recette

Chaque cas pratique suit la même démarche : contexte, analyse, plan d'automatisation, cahier de recette, mise en œuvre avec du code, points de vigilance. Les exemples de code sont des **modèles à adapter** : sélecteurs, routes d'API et libellés doivent être alignés sur votre application réelle.

Cas 1 : test automatisé d'un logiciel web, formulaire d'inscription

Contexte

Application web avec formulaire d'enregistrement : identifiant (login), email, mot de passe et confirmation. Un bouton « S'inscrire » valide la saisie.

Règles de gestion supposées (à confirmer avec la spécification)

- Identifiant : 3 à 20 caractères, lettres, chiffres, point et tiret bas.
- Email : format valide, unique dans la base.
- Mot de passe : 8 à 64 caractères, avec au moins une majuscule, un chiffre et un caractère spécial.
- Confirmation : identique au mot de passe.
- Succès : création du compte, ouverture de session, redirection vers /dashboard.

Analyse : application des techniques

Champ	Classes valides	Classes invalides	Valeurs limites
Identifiant	3 à 20 caractères autorisés	Vide, 2 ou 21 caractères, caractères interdits	2 · 3 · 4 · 19 · 20 · 21
Email	Format standard valide	Sans @, sans domaine, avec espaces, déjà utilisé	Longueur maximale de la spécification
Mot de passe	8 à 64 caractères avec règles respectées	7 caractères, sans majuscule, sans chiffre, sans spécial	7 · 8 · 9 · 63 · 64 · 65
Confirmation	Identique	Différente, vide	Différence d'un caractère ou de casse

Plan d'automatisation : quoi automatiser, à quel niveau ?

Contrôle	Niveau	Pourquoi
Parcours nominal d'inscription (TC-WEB-001)	Interface (E2E)	Un seul test de bout en bout valide l'intégration complète
Règles de validation des champs (TC-WEB-003 à 008)	Interface avec données multiples	Les messages affichés doivent être vérifiés côté utilisateur
Unicité de l'email (TC-WEB-002)	API (et un test E2E d'affichage)	La règle se contrôle en millisecondes sans interface

Contrôle	Niveau	Pourquoi
Double soumission, injection (TC-WEB-009, 010)	Interface	Comportement de l'interface et sécurité de base
Accessibilité clavier et lecteur d'écran (TC-WEB-012)	Manuel + contrôle automatisé partiel	L'expérience réelle doit être jugée par un humain

Cahier de recette

TC-WEB-001	Inscription avec données valides — Priorité P1
Préconditions	Aucun utilisateur avec cet email n'existe (email généré de façon unique)
Données de test	Login : marie.dupont · Email : marie.<horodatage>@example.com · Mot de passe : MotDePasse123!
Étapes automatisées	1. Ouvrir la page /register 2. Remplir login, email, mot de passe et confirmation 3. Cliquer sur « S'inscrire »
Assertions attendues	URL contient /dashboard, message de bienvenue visible, compte créé (vérifié via l'API).
Script associé	tests/inscription.spec.ts · « TC-WEB-001 »
Résultat (rapport CI)	
Statut	☐ Pass ☐ Fail ☐ Flaky ☐ Non exécuté

TC-WEB-002	Inscription avec email déjà utilisé — Priorité P1
Préconditions	Un compte existe avec l'email de test (créé par l'API avant le test)
Données de test	Même email, mot de passe différent, autre login
Étapes automatisées	1. Créer le compte de référence par l'API 2. Ouvrir /register et saisir les données 3. Cliquer sur « S'inscrire »
Assertions attendues	Message « Cet email est déjà utilisé » visible, URL toujours /register, aucun second compte créé.
Script associé	tests/inscription.spec.ts · « TC-WEB-002 » + tests/api/register.spec.ts
Résultat (rapport CI)	
Statut	☐ Pass ☐ Fail ☐ Flaky ☐ Non exécuté

Autres cas à concevoir et automatiser :

ID	Scénario	Priorité	Automatisation
TC-WEB-003	Mot de passe trop court (7 caractères)	P1	Interface, données multiples
TC-WEB-004	Mot de passe de 8 caractères exactement	P2	Interface, données multiples
TC-WEB-005	Confirmation différente du mot de passe	P1	Interface, données multiples
TC-WEB-006	Tous les champs vides	P2	Interface
TC-WEB-007	Email sans @	P2	Interface, données multiples

ID	Scénario	Priorité	Automatisation
TC-WEB-008	Login avec caractères interdits	P2	Interface, données multiples
TC-WEB-009	Double clic rapide sur « S'inscrire »	P2	Interface (un seul compte créé)
TC-WEB-010	Injection de script dans le login	P2	Interface + contrôle de réponse
TC-WEB-011	Mot de passe jamais exposé en clair (URL, réponse réseau)	P1	Interception réseau dans le test
TC-WEB-012	Navigation au clavier et libellés accessibles	P3	Manuel, contrôle d'accessibilité automatisé en appoint

Mise en œuvre avec Playwright

Étape 1 : installer Playwright.

```
npm init playwright@latest
```

Étape 2 : configurer le projet (adresse de l'application, navigateurs, rapports, preuves en cas d'échec).

```
// playwright.config.ts
import { defineConfig, devices } from '@playwright/test';

export default defineConfig({
  testDir: './tests',
  retries: process.env.CI ? 1 : 0,
  workers: process.env.CI ? 2 : undefined,
  reporter: [['list'], ['html']],
  use: {
    baseURL: process.env.BASE_URL ?? 'http://localhost:3000',
    trace: 'on-first-retry',
    screenshot: 'only-on-failure',
    video: 'retain-on-failure',
  },
  projects: [
    { name: 'chromium', use: { ...devices['Desktop Chrome'] } },
    { name: 'firefox', use: { ...devices['Desktop Firefox'] } },
    { name: 'webkit', use: { ...devices['Desktop Safari'] } },
  ],
});
```

Étape 3 : écrire le test nominal avec le Page Object et un email unique.

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

test('TC-WEB-001 : inscription valide', async ({ page }) => {
  const register = new RegisterPage(page);
  const email = `marie.${Date.now()}@example.com`; // donnée unique

  await register.open();
  await register.fillForm({ login: 'marie.dupont', email,
    password: 'MotDePasse123!', confirm: 'MotDePasse123!' });
  await register.submit();

  await expect(page).toHaveURL(/dashboard/);
  await expect(page.getByText('Bienvenue')).toBeVisible();
});
```

```
});
```

Étape 4 : automatiser les cas d'erreur avec des données multiples.

```
const casInvalides = [
  { id: 'TC-WEB-003', mdp: 'Court1!', conf: 'Court1!', msg: '/8 caractères/ },
  { id: 'TC-WEB-005', mdp: 'MotDePasse123!', conf: 'Autre123!', msg: '/ne correspondent pas/ },
];

for (const c of casInvalides) {
  test(`${c.id} : inscription refusée`, async ({ page }) => {
    const register = new RegisterPage(page);
    await register.open();
    await register.fillForm({ login: 'marie.dupont',
      email: `marie.${Date.now()}@example.com`, password: c.mdp, confirm: c.conf });
    await register.submit();
    await expect(register.alert()).toContainText(c.msg);
    await expect(page).toHaveURL(/register/);
  });
}
```

Étape 5 : contrôler la règle d'unicité au niveau API (rapide et stable).

```
test('TC-WEB-002 (API) : email déjà utilisé', async ({ request }) => {
  const payload = { login: 'marie.dupont', email: `marie.${Date.now()}@example.com`,
    password: 'MotDePasse123!' };
  const premier = await request.post('/api/register', { data: payload });
  expect(premier.status()).toBe(201);

  const second = await request.post('/api/register', { data: payload });
  expect(second.status()).toBe(409); // route et codes à adapter
});
```

Étape 6 : lancer et analyser.

```
npx playwright test           # tous les tests
npx playwright test --ui      # mode interactif pour déboguer
npx playwright show-report    # rapport HTML avec traces
```

Points de vigilance spécifiques au web

- **Données uniques** : un email généré à chaque exécution évite les collisions entre tests ou campagnes.
- **Nettoyage** : supprimer les comptes créés (API de nettoyage) pour ne pas polluer l'environnement.
- **Pas de pause fixe** : s'appuyer sur les attentes automatiques et les assertions web.
- **Compatibilité** : exécuter au moins un test P1 sur chaque navigateur du périmètre.
- **Mot de passe de test** : stocké dans des variables d'environnement ou le coffre de secrets de la CI, pas en clair dans le dépôt.

Cas 2 : test automatisé d'une application mobile, écran de connexion (Android / iOS)

Contexte

Application mobile native ou hybride avec écran de connexion (email et mot de passe), éventuellement avec option « Se souvenir de moi » et déverrouillage biométrique.

Spécificités de l'automatisation mobile

Domaine	Ce qui change pour l'automatisation
Plateformes	Un parc d'émulateurs, de simulateurs et d'appareils réels ; matrice de versions d'OS à choisir avec soin
Identifiants d'éléments	Les identifiants d'accessibilité sont la clé d'un test stable : à demander à l'équipe de développement
Durée d'exécution	Démarrage d'émulateur et installation de l'application : plus lent que le web, d'où une suite ciblée
Interruptions	Appel entrant, notification, mode avion : simulables en partie par commandes de l'émulateur
Réseau	Coupure et lenteur simulables par des commandes ou outils réseau
Appareils réels	Indispensables pour la biométrie, le comportement du clavier et de la batterie
Intégration continue	Émulateur Android lancé dans le pipeline ; iOS exige un exécuteur macOS

Plan d'automatisation

Contrôle	Automatisable ?	Remarque
Connexion nominale et mot de passe erroné (TC-MOB-001, 002)	Oui	Cœur du smoke test mobile
Rotation de l'écran (TC-MOB-003)	Oui	Commande de rotation de l'émulateur
Mode avion (TC-MOB-005)	Oui	Commande réseau de l'émulateur
Mise en arrière-plan puis retour (TC-MOB-007)	Oui	Action d'application en arrière-plan du driver
Appel entrant (TC-MOB-004)	Partiellement	Simulable sur émulateur, à confirmer sur appareil réel
Biométrie (TC-MOB-008)	Partiellement	Simulation d'empreinte sur émulateur ; appareil réel pour la confiance
Taille de police système, confort visuel (TC-MOB-011)	Non (manuel)	Jugement humain sur la lisibilité

Cahier de recette

TC-MOB-001	Connexion avec identifiants valides — Priorité P1
Préconditions	Compte utilisateur actif, application installée sur l'émulateur, réseau disponible

Données de test	Email : user@test.com · Mot de passe : Valid123! · Android 14 / iOS 17
Étapes automatisées	1. Lancer l'application 2. Saisir l'email et le mot de passe 3. Appuyer sur « Se connecter »
Assertions attendues	Écran d'accueil affiché, message « Bienvenue » visible, jeton de session stocké de façon sécurisée.
Script associé	flows/login_ok.yaml (Maestro) · specs/login.spec.ts (Appium)
Résultat (rapport CI)	
Statut	☐ Pass ☐ Fail ☐ Flaky ☐ Non exécuté

TC-MOB-002	Connexion avec mot de passe incorrect — Priorité P1
Préconditions	Compte actif
Données de test	Email valide, mot de passe erroné
Étapes automatisées	1. Lancer l'application 2. Saisir l'email valide et un mot de passe erroné 3. Appuyer sur « Se connecter »
Assertions attendues	Message « Identifiants incorrects » visible, écran de connexion toujours affiché, compteur de tentatives incrémenté.
Script associé	flows/login_ko.yaml (Maestro) · specs/login.spec.ts (Appium)
Résultat (rapport CI)	
Statut	☐ Pass ☐ Fail ☐ Flaky ☐ Non exécuté

Autres cas mobiles :

ID	Scénario	Priorité	Automatisation
TC-MOB-003	Rotation de l'écran pendant la saisie	P2	Oui : saisie conservée après rotation
TC-MOB-004	Interruption par appel entrant	P2	Partielle : émulateur puis appareil réel
TC-MOB-005	Connexion en mode avion	P1	Oui : message d'erreur réseau
TC-MOB-006	Perte du réseau pendant l'envoi	P2	Oui : coupure réseau déclenchée par le test
TC-MOB-007	Arrière-plan 5 minutes puis retour	P2	Oui : session conservée ou reconnexion
TC-MOB-008	Biométrie : succès, échec, annulation	P2	Partielle : simulation sur émulateur
TC-MOB-009	Clavier : ouverture, masquage, boutons accessibles	P3	Oui : visibilité des éléments
TC-MOB-010	Mot de passe affiché ou masqué (icône œil)	P3	Oui : attribut de masquage
TC-MOB-011	Taille de police système agrandie	P3	Non : manuel

Mise en œuvre avec Maestro (sans code)

```
# flows/login_ok.yaml
```

```
appId: com.exemple.monapp
---
- launchApp:
  clearState: true
- tapOn: "Email"
- inputText: ${EMAIL}
- tapOn: "Mot de passe"
- inputText: ${PASSWORD}
- tapOn: "Se connecter"
- assertVisible: "Bienvenue"
```

```
# flows/login_ko.yaml (même structure, assertion différente)
- assertVisible: "Identifiants incorrects"

# Lancement avec variables, sans mot de passe en dur dans le fichier
maestro test -e EMAIL=user@test.com -e PASSWORD=Valid123! flows/login_ok.yaml
```

Mise en œuvre avec Appium (avec code)

```
npm install -g appium
appium driver install uiautomator2
appium # démarre le serveur sur le port 4723
```

```
// capacités de connexion à l'émulateur Android
const capabilities = {
  platformName: 'Android',
  'appium:automationName': 'UiAutomator2',
  'appium:deviceName': 'Pixel_7_API_34',
  'appium:app': process.env.APK_PATH,
};

// TC-MOB-001 (WebdriverIO + Appium)
it('TC-MOB-001 : connexion valide', async () => {
  await $('~login_email').setValue(process.env.EMAIL);
  await $('~login_password').setValue(process.env.PASSWORD);
  await $('~login_submit').click();
  await expect($('~welcome_message')).toBeDisplayed();
});
```

ASTUCE

Demandez aux développeurs d'ajouter des **identifiants d'accessibilité** ('loginemail', 'loginpassword'...) sur les éléments clés. Ils rendent les scripts stables et améliorent l'accessibilité de l'application.

Points de vigilance spécifiques au mobile

- Partir d'un **état propre** (application réinstallée ou données effacées) pour chaque test.
- Prévoir au moins **un appareil réel par plateforme** pour les parcours P1 : un émulateur ne reproduit pas tout.
- Maintenir la **matrice d'appareils** raisonnable (pairwise sur version d'OS × taille d'écran).
- Conserver **vidéos et journaux** d'exécution, très utiles pour comprendre un échec sur appareil.
- Ne jamais mettre d'identifiants réels dans les scripts : variables d'environnement ou secrets de la CI.

Cas 3 : test automatisé d'un produit, terminal de paiement

Contexte

Terminal de paiement électronique (TPE) acceptant la carte à puce et sans contact. Le produit est un **objet physique** doté d'un logiciel embarqué : l'automatisation exige des dispositifs spécifiques et ne couvre qu'une partie des tests.

Que peut-on automatiser sur un produit physique ?

Niveau	Principe	Exemples
Logiciel embarqué sur simulateur	Le simulateur ou le SDK du fabricant exécute le même logiciel sans le matériel	Scénarios de paiement, de refus, d'annulation, de remboursement
API et serveur	Contrôler ce que le serveur d'acquisition enregistre	Statut et montant de la transaction, clôture de journée
Terminal réel piloté	Banc de test : bras robotisé, carte de test simulée, interface de commande du terminal	Smoke test sur matériel, tests d'endurance
Manuel	Observation humaine du matériel	Distance de lecture sans contact, qualité du ticket imprimé, sons, retrait de carte

- **Cartes de test uniquement** : fournies par le constructeur, la banque ou l'organisme certificateur, sur un environnement de qualification. Jamais de vraies cartes.
- **Certifications** (EMV, normes de sécurité PCI) : la recette automatisée s'y ajoute, elle ne les remplace pas.
- **Traçabilité** : noter la version du logiciel embarqué et le numéro de série du terminal à chaque campagne.

Cahier de recette

TC-TPE-001	Paieement par carte sans contact valide — Priorité P1
Préconditions	Simulateur du terminal démarré et connecté au serveur de test
Données de test	Montant : 25,00 € · Carte de test sans contact valide
Étapes automatisées	1. Lancer la transaction de 25,00 € sur le simulateur 2. Présenter la carte de test sans contact 3. Récupérer le résultat, le ticket et l'enregistrement serveur
Assertions attendues	Statut « Accepté », ticket avec le bon montant, transaction enregistrée côté serveur avec le même montant et le même statut.
Script associé	teststpe/testpaiement.py::testtctpe_001
Résultat (rapport CI)	
Statut	☐ Pass ☐ Fail ☐ Flaky ☐ Non exécuté
TC-TPE-002	Paieement refusé pour provision insuffisante — Priorité P1
Préconditions	Simulateur démarré

Données de test	Carte de test configurée avec solde insuffisant
Étapes automatisées	1. Lancer la transaction 2. Présenter la carte de test à solde insuffisant 3. Récupérer le résultat
Assertions attendues	Statut « Refusé », message adapté, aucun débit enregistré côté serveur.
Script associé	teststpe/testpaiement.py::testtctpe_002
Résultat (rapport CI)	
Statut	☐ Pass ☐ Fail ☐ Flaky ☐ Non exécuté

Tests additionnels et niveau d'automatisation :

ID	Scénario	Priorité	Automatisation
TC-TPE-003	Carte expirée	P1	Simulateur
TC-TPE-004	Montant supérieur au plafond sans contact : demande du code PIN	P1	Simulateur
TC-TPE-005	Perte de connexion réseau pendant le paiement	P1	Simulateur avec coupure réseau simulée
TC-TPE-006	Code PIN erroné (1ère, 2ème, 3ème tentative)	P1	Simulateur
TC-TPE-007	Retrait prématuré de la carte à puce	P1	Banc matériel ou manuel
TC-TPE-008	Annulation d'un paiement	P2	Simulateur + contrôle serveur
TC-TPE-009	Remboursement partiel et total	P2	Simulateur + contrôle serveur
TC-TPE-010	Montants limites (0,01 € et maximum)	P2	Simulateur, données multiples
TC-TPE-011	Coupure d'alimentation en cours de transaction	P2	Banc matériel (alimentation pilotable) ou manuel
TC-TPE-012	Batterie faible	P3	Manuel
TC-TPE-013	Clôture de journée : totaux terminal = journal = serveur	P1	Simulateur + API serveur

Mise en œuvre (exemple illustratif avec pytest)

L'exemple suivant suppose un simulateur exposant une API de commande. **Les adresses et champs sont fictifs** : à remplacer par ceux du SDK de votre fabricant.

```
import pytest, requests

SIMULATEUR = 'http://simulateur-tpe.local/api' # fictif
SERVEUR = 'http://serveur-test.local/api' # fictif

def payer(montant_cts, carte):
    r = requests.post(f'{SIMULATEUR}/transactions',
                      json={'montant': montant_cts, 'carte': carte})
    assert r.status_code == 200
    return r.json()
```

```
def test_tc_tpe_001_sans_contact_valide():
    t = payer(2500, 'VISA_SANS_CONTACT_VALIDE')
    assert t['statut'] == 'ACCEPTE'
    assert t['ticket']['montant'] == 2500 # ce que le ticket imprime
    srv = requests.get(f'{SERVEUR}/transactions/{t['id']}').json()
    assert srv['statut'] == 'ACCEPTE' and srv['montant'] == 2500 # côté serveur

@pytest.mark.parametrize('carte, statut', [
    ('CARTE_SOLDE_INSUFFISANT', 'REFUSE'),
    ('CARTE_EXPIREE', 'REFUSE'),
])
def test_tc_tpe_002_003_refus(carte, statut):
    t = payer(2500, carte)
    assert t['statut'] == statut
    srv = requests.get(f'{SERVEUR}/transactions/{t['id']}').json()
    assert srv['debit'] is False # aucun débit
```

ATTENTION

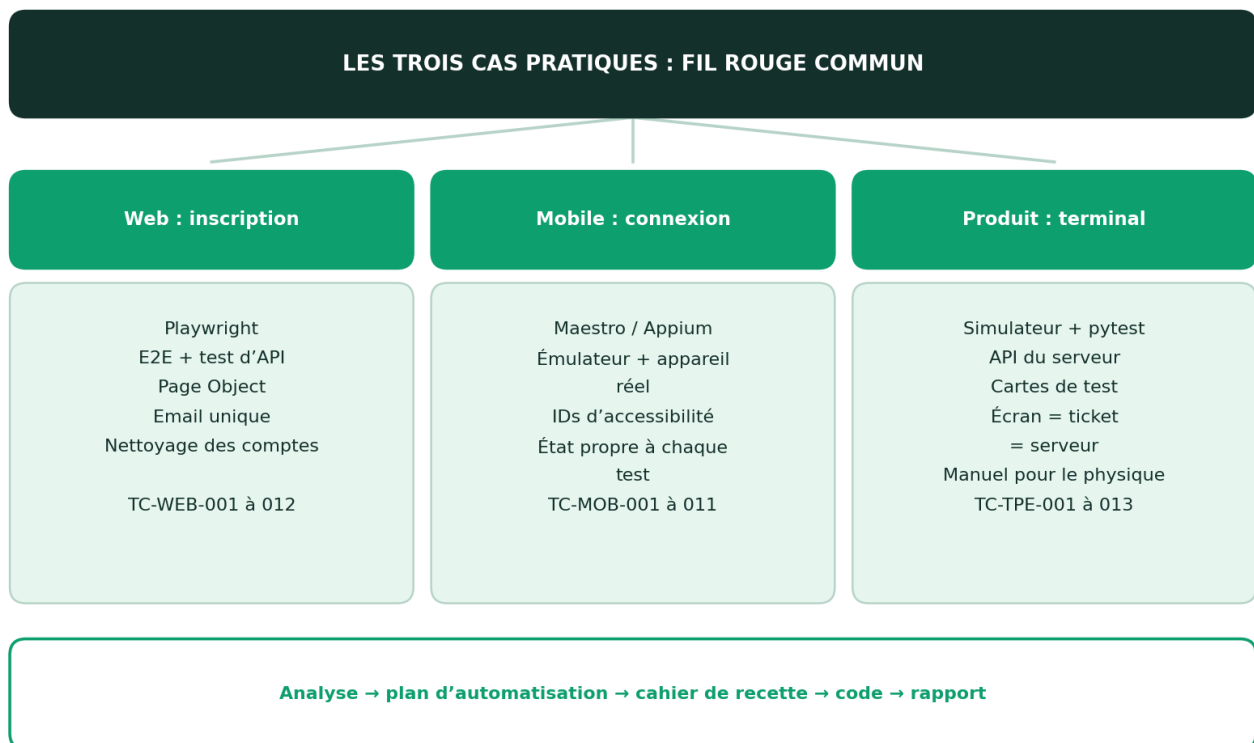
Pour chaque test, **comparez trois sources** : ce que le terminal affiche, ce que le ticket imprime et ce que le serveur enregistre. Une incohérence est un défaut, même si l'écran affiche « Paiement accepté ».

Vérifiez aussi ce que le terminal **ne doit pas faire** : ne jamais afficher ni imprimer le numéro complet de la carte.

Points de vigilance spécifiques au produit physique

- Le simulateur doit être **fidèle** au matériel : un écart de comportement fausse tous les résultats. Validez-le par quelques tests sur le terminal réel.
- Les tests sur matériel sont **lents et coûteux** : les réserver au smoke test et aux risques que le simulateur ne couvre pas.
- Prévoir des **jeux de cartes de test** documentés (valide, expirée, solde insuffisant, PIN bloqué) et leur durée de validité.
- Garder des **tests manuels** pour tout ce qui relève du physique : lecture à distance, ticket, sons, robustesse.

Schéma récapitulatif de la partie 4



Partie 5 : Conseils pour réussir en tant qu'automaticien junior

5.1 Les réflexes à acquérir

1. **Lire la spécification en premier** : le test automatisé se base sur les exigences, pas sur l'interface actuelle.
2. **Penser aux cas limites** : les bugs se cachent aux bornes, pas au milieu.
3. **Écrire d'abord le test manuel**, puis l'automatiser.
4. **Tester le plus bas possible** dans la pyramide (API plutôt qu'interface quand c'est possible).
5. **Traiter le code de test comme du code de production** : versionné, relu, lisible, refactoré.
6. **Traquer les tests instables** dès qu'ils apparaissent : un test peu fiable est pire que pas de test.
7. **Un bug = un rapport clair** : étapes, attendu versus obtenu, trace ou capture.

5.2 Erreurs fréquentes à éviter

Anti-pattern	Conséquence	Bonne pratique
Pauses fixes (sleep)	Tests lents et instables	Attentes conditionnelles
Sélecteurs CSS ou XPath longs et fragiles	Casse à chaque changement de mise en page	Rôles, libellés, data-testid
Tests dépendants les uns des autres	Échecs en cascade, ordre imposé	Tests isolés avec préparation propre
Tout automatiser par l'interface	Suite lente et chère à maintenir	Pyramide : plus d'API, peu d'E2E
Données en dur et partagées	Collisions, résultats aléatoires	Données générées, nettoyage systématique
Identifiants dans le code	Fuite de secrets	Variables d'environnement, coffre de secrets
Ignorer les tests rouges	Perte de confiance dans la suite	Triage quotidien, quarantaine documentée
Automatiser trop tôt une fonction instable	Maintenance permanente	Attendre la stabilité de la fonctionnalité

5.3 Outils indispensables pour débiter

Besoin	Outil	Justification
Automatisation web	Playwright	Attentes automatiques, rapports et traces intégrés, gratuit
Automatisation mobile	Maestro ou Appium	Maestro pour débiter, Appium pour la puissance
Tests d'API	Playwright ou Postman / Bruno	Rapides à écrire et très stables
Gestion du code	Git et un dépôt hébergé	Versionner, relire, déclencher la CI

Besoin	Outil	Justification
Intégration continue	GitHub Actions ou GitLab CI	Exécution automatique et rapports partagés
Suivi des bugs	Jira (gratuit pour petites équipes)	Standard du marché

5.4 Les qualités d'un bon automaticien

- **Rigueur de développeur** : lisibilité, nommage, factorisation, relecture de code.
- **Esprit de testeur** : curiosité, scepticisme, goût des cas limites.
- **Communication** : signaler un défaut de façon factuelle ; expliquer simplement un échec de test à un non-technicien.
- **Compréhension du métier** : on automatise des risques, pas des clics.
- **Apprentissage continu** : outils et pratiques évoluent vite.

5.5 Plan de progression sur 90 jours

Période	Objectifs	Livrables
Jours 1 à 30	Acquérir les bases de JavaScript/TypeScript ou Python et de Git ; écrire des cas de test manuels solides sur une application publique	10 à 15 cas rédigés et un premier script Playwright qui passe
Jours 31 à 60	Structurer un framework (Page Objects, fixtures, données) ; automatiser des tests d'API ; produire des rapports	Dépôt avec 15 tests structurés et rapport HTML
Jours 61 à 90	Brancher la suite sur la CI, traiter les tests instables, découvrir Maestro ou Appium	Pipeline fonctionnel, README, premier smoke test mobile

5.6 Pour aller plus loin

- **Certification ISTQB** : niveau Foundation puis spécialisation en ingénierie de l'automatisation des tests.
- **Documentation officielle** de Playwright, Appium et Maestro, régulièrement mise à jour.
- **Entraînement** : applications de démonstration publiques conçues pour s'exercer à l'automatisation.
- **Communautés** de testeurs (meetups, conférences, groupes en ligne).

Schéma récapitulatif de la partie 5

DEVENIR AUTOMATICIEN : L'ESSENTIEL

7 réflexes

- 1 Lire la spécification
- 2 Penser aux limites
- 3 Test manuel d'abord
- 4 Tester le plus bas
- 5 Code de test = code de production
- 6 Traquer les flaky
- 7 Bug = rapport clair

Anti-patterns → bonnes pratiques

Pauses fixes	→	Attentes conditionnelles
Sélecteurs CSS fragiles	→	Rôles, libellés, data-testid
Tests dépendants	→	Tests isolés
Tout en E2E	→	Pyramide : plus d'API
Secrets dans le code	→	Variables, coffre CI

Plan de progression sur 90 jours

Jours 1 à 30

Bases : JS/TS ou Python,
Git, cas manuels solides

Jours 31 à 60

Framework : Page Objects,
données, API, rapports

Jours 61 à 90

CI, tests flaky,
Maestro ou Appium

Annexes : modèles prêts à l'emploi

Annexe A : modèle de cas de test automatisé

TC-XXX-000	Titre précis du scénario — Priorité P1 / P2 / P3
Préconditions	État initial requis, préparé par le script (compte, donnée, droit)
Données de test	Valeurs d'entrée exactes ou référence au fichier de données
Étapes automatisées	1. Première action 2. Deuxième action 3. Troisième action
Assertions attendues	Vérifications observables : texte, URL, code de réponse, donnée enregistrée
Script associé	chemin/du/fichier.spec.ts · « TC-XXX-000 »
Résultat (rapport CI)	
Statut	☐ Pass ☐ Fail ☐ Flaky ☐ Non exécuté

Annexe B : modèle de rapport d'anomalie

Champ	À renseigner
Identifiant	BUG-000
Titre	Verbe + contexte + symptôme, en une phrase
Test en échec	Identifiant du cas (TC-XXX-000) et lien vers l'exécution CI
Version et environnement	Version testée, commit, navigateur / appareil, système, URL
Étapes de reproduction manuelle	1. ... 2. ... 3. ...
Résultat attendu	Selon la spécification (référence)
Résultat obtenu	Message d'assertion et comportement réel
Fréquence	Systématique / intermittente (x échecs sur y exécutions)
Sévérité	Bloquante / Majeure / Mineure / Cosmétique
Preuves	Trace, capture, vidéo, journaux, requête réseau

Annexe C : checklist avant d'automatiser un test

	Point de contrôle
☐	Le test manuel existe, il est validé et son résultat attendu est précis
☐	La fonctionnalité est stable et le test sera exécuté souvent
☐	Le test est indépendant des autres et prépare ses propres données

	Point de contrôle
☐	Le niveau de test est le plus bas possible (API plutôt qu'interface si c'est suffisant)
☐	Les éléments à manipuler ont des identifiants stables (data-testid, identifiants d'accessibilité)
☐	Le nettoyage de l'environnement est prévu
☐	Aucun secret ni donnée personnelle réelle dans le script
☐	Le test est rentable : coût de développement et de maintenance inférieur au gain

Annexe D : checklist de relecture d'un script

	Point de contrôle
☐	Le nom du test est lisible et contient l'identifiant du cas (TC-XXX-000)
☐	Le test suit la structure préparer, agir, vérifier
☐	Il contient au moins une assertion qui porte sur ce que l'utilisateur voit
☐	Aucune pause fixe, aucun sélecteur fragile
☐	Les données sont générées ou externalisées, pas dupliquées dans le code
☐	Pas de logique complexe (boucles, conditions) à l'intérieur du test
☐	Le test a été exécuté plusieurs fois de suite, seul et en parallèle, sans échec aléatoire
☐	Les preuves en cas d'échec (trace, capture) sont activées

Annexe E : aide-mémoire des commandes

Besoin	Commande
Installer Playwright	<code>npm init playwright@latest</code>
Enregistrer des actions	<code>npx playwright codegen http://localhost:3000</code>
Lancer tous les tests	<code>npx playwright test</code>
Lancer un fichier ou un test	<code>npx playwright test tests/inscription.spec.ts -g "TC-WEB-001"</code>
Mode interactif de débogage	<code>npx playwright test --ui</code>
Ouvrir le rapport HTML	<code>npx playwright show-report</code>
Lancer un flux Maestro	<code>maestro test flows/login_ok.yaml</code>
Démarrer le serveur Appium	<code>appium</code>
Lancer les tests Python (TPE)	<code>pytest -v tests_tpe/</code>

Conclusion

Le test automatisé n'est pas une fin en soi : c'est un moyen d'obtenir, vite et de façon répétable, une information fiable sur la qualité du produit. Il repose sur les mêmes principes que le test fonctionnel : comprendre les spécifications, concevoir des cas pertinents avec les bonnes techniques, définir des résultats attendus précis et documenter clairement.

En choisissant avec soin ce que l'on automatise, en structurant le code (Page Objects, fixtures, données maîtrisées), en traquant les tests instables et en intégrant la suite dans la chaîne d'intégration continue, un testeur junior peut bâtir une suite d'automatisation durable, avec des outils open source comme Playwright, Maestro ou Appium et Postman ou Bruno.

À RETENIR

Les trois idées à retenir : (1) automatisez ce qui est stable, fréquent et critique, au niveau le plus bas possible ; (2) un test automatisé est du code : il doit être lisible, isolé et maintenu ; (3) un test peu fiable détruit la confiance : corrigez les instabilités sans attendre.