DOCUMENT TECHNIQUE

DE FORMATION SUR LES TESTS AUTOMATIQUES

V 2.0

Cas de l'enseigne

SOMMAIRE

CHAPITRE I : GENERALITES SUR LES TESTS AUTOMATIQUES

Qu'est-ce qu'un test automatisé ?

Un test automatisé est une méthode de test dans laquelle des outils automatisés exécutent des cas de test prédéfinis pour comparer les résultats attendus du produit développé avec les résultats obtenus. Si le scénario de test s'exécute avec succès sans erreur, le test est considéré comme réussi.

Contrairement aux tests manuels, les tests automatisés sont effectués sans intervention humaine. Cette méthode nécessite l'utilisation de solutions informatiques pour effectuer des actions prédéterminées dans des scripts et analyser des produits sur des itinéraires spécifiques.

Pourquoi faire des tests automatiques ?

L'intérêt d'automatiser pour les entreprises :

L'objectif des tests automatisés est de rendre les tests aussi simples que possible grâce à des scripts. Ensuite, exécutez des tests basés sur celui-ci, rapportez les résultats et comparez-les avec les résultats des tests précédents. Son principal avantage est de gagner du temps et de l'argent.

Pour l'entreprise, les bienfaits sont également palpables, qui raviront l'équipe dirigeante.

La Satisfaction des clients : Conséquence d'un produit fiable, qui s'enrichit tout en maintenant un niveau de qualité optimal.

La Compétitivité : Libérer les forces de production grâce à une stratégie qualité efficace et pérenne basée sur l'automatisation des tests est un levier de performance évident qui aide à enrichir en continu un produit.

Apporter de l'Innovation : cela laisse aussi le temps pour explorer, s'occuper d'autres tâches.

L'intérêt d'automatiser pour les Equipes :

L'équipe de développement, qu'elle soit en méthode Agile ou en méthode traditionnelle de Cycle en V, a également des bénéfices à retirer de l'automatisation de la qualité fonctionnelles.

Favoriser l'exécution en partie de tests exploratoires : Libérer les testeurs manuels des tests répétitifs, fastidieux et chronophages leur permet de se concentrer sur des tests pour lesquels ils ont

une meilleure valeur ajoutée : les tests exploratoires, de sérendipité. Ce qui permet de réintroduire une part de hasard dans la phase de recette fonctionnelle.

Améliorer la réactivité face à un bug : Pour un développeur, pouvoir lancer des campagnes de tests automatisées aussi souvent qu'il le souhaite lui permet d'être alerté très rapidement en cas de bug et de prendre les actions nécessaires.

Réduire la maintenance corrective : Augmenter la fréquence des tests permet également de détecter plus de bugs avant livraison en production. Et de fait, ce sont autant de bugs qui ne seront pas détectés en production. Par exemple si la résolution d'un bug coûte environ 100 € lors de la conception du logiciel, alors sa correction, une fois l'application diffusée, coûtera plus de 10 000€ à l'éditeur.

La phase de débogage occupe une place critique dans le cycle de vie du développement applicatif. Méthodes itératives et chasse à la régression sont de mise pour la mener à bien.

Libérer les forces de production : Si la charge de la maintenance corrective diminue, cela laisse inévitablement plus de temps aux développeurs pour se concentrer sur ce qu'ils aiment faire et sur leur cœur de métier : la production de nouvelles fonctionnalités.

Épanouissement des équipes : Qui ne serait pas épanoui dans ce genre de situation où tout est fait avec méthode pour optimiser la production et la qualité ?

L'intérêt d'automatiser les Tests Fonctionnels :

Automatiser les tests fonctionnels présente les avantages suivants :

Accroître la rapidité des tests : Une exécution automatique est aisément plus rapide que l'exécution manuelle. Cela est dû au fait que le test automatique est programmé pour enchaîner les actions dès qu'elles sont possibles, ce qui rend plus efficace l'automate que le testeur humain dans le déroulement du scénario.

Assurer la reproductibilité : Un test automatisé est programmé de telle manière qu'il peut être exécuté autant de fois que souhaité. Il est donc programmé une fois par un humain puis il peut être exécuté à volonté. Objectif efficacité !

Multiplier les contextes de tests : Une fois un test automatisé programmé, il est possible de l'exécuter avec différents jeux de données. Un même test peut donc se décliner en autant d'exécutions qu'il y a de contextes d'utilisation.

Paralléliser les tests : Alors qu'un testeur manuel ne peut réaliser qu'un seul test à la fois, plusieurs automates peuvent donc être lancés en parallèle, exécutant chacun un test. Ceci rend d'autant plus efficace la campagne de tests.

Étendre la couverture des tests : Avec le temps, si la base de tests automatisés est maintenue à jour, elle ne fait que croître puisqu'on vient l'enrichir de nouveaux scénarios couvrant les nouvelles fonctionnalités et les évolutions.

Maintenir la fiabilité : Grâce à la couverture des tests qui s'étend en continu, c'est l'ensemble du produit qui conserve son niveau de fiabilité.

Augmenter la fréquence des tests : Les campagnes de tests automatisés prennent moins de temps tout en couvrant davantage de fonctionnalités que des campagnes de tests manuels. Cela permet de lancer facilement et quand on le souhaite des tests.

Développer la capacité de tests : Facile quand on teste plus, plus vite et potentiellement plus souvent.

Permettre la planification des tests : Les testeurs manuels sont des humains soumis à des horaires de travail. Les automates n'ont pas ces contraintes. Ainsi il n'y a pas d'obstacle pour exécuter pendant la nuit les tests des développements effectués le jour.

Faciliter le contrôle visuel : Pour un automate, prendre une copie d'écran fait partie de son B.A.BA et cette fonctionnalité peut permettre à un humain de contrôler très facilement le rendu visuel dans le navigateur web s'il souhaite garder la maîtrise de la qualité finale. Sans être ralenti par les temps d'exécution et de chargement des pages, il a juste à faire défiler des captures d'écran dans une visionneuse.

Les Différents Types de Test Logiciel :

1.3.1. Tests fonctionnels :

Les tests fonctionnels sont utilisés pour vérifier les fonctions d'une application logicielle conformément aux spécifications des exigences. Ils impliquent principalement des tests de boîte noire et ne dépendent pas du code source de l'application.

Les tests fonctionnels consistent à vérifier l'interface utilisateur, la base de données, les API, les applications client/serveur ainsi que la sécurité et la fonctionnalité du logiciel testé. Ils peuvent être effectués manuellement ou en utilisant l'automatisation.

Les différents types de tests fonctionnels sont les suivants :

Tests unitaires (ou test de composant) : Les tests unitaires consistent à isoler une partie du code et à vérifier qu'il fonctionne parfaitement unitairement. Il s'agit de petits tests qui valident l'attitude d'un objet et la logique du code. Les tests unitaires sont généralement développés et lancés par les développeurs pendant la phase de développement des applications mobiles ou logicielles.

Tests d'intégration : Sachez qu'il existe deux types de tests d'intégration : les tests d'intégration composants et les tests d'intégration système :

Les tests d'intégration composants : ils permettent de vérifier si plusieurs unités de code fonctionnent bien ensemble, dans un environnement de test assez proche du test unitaire, c'est-à-dire de manière isolée, sans lien avec des composants extérieurs et ne permettant pas le démarrage d'une vraie application ;

Les tests d'intégration système : ils permettent de vérifier le fonctionnement de plusieurs unités de code au sein d'une configuration d'application, avec éventuellement des liens avec des composants extérieurs comme une base de données, des fichiers, ou des API en réseau.

Tests de système : Ils consistent à exécuter plusieurs scénarios complets qui constituent les cas d'utilisation du logiciel. Dans le jargon informatique, on le qualifie de test de type boîte noire et ils permettent de s'assurer de la fonctionnalité globale du logiciel et de son comportement sur les terminaux d'utilisation. Pour aboutir à des reportings objectifs pour ce type de test, l'équipe qui en a la charge doit se distinguer et assurer une totale indépendance vis à vis des équipes de développement.

Tests d'intégrité : Test des méthodes et processus utilisés pour accéder et gérer les données ou base de données, pour s'assurer que les méthodes d'accès, processus et règles de données fonctionnent comme attendu et que lors des accès à la base de données, les données ne sont pas corrompues ou inopinément effacées, mises à jour ou créées.

Tests de fumée : Un test de fumée consiste en des tests fonctionnels ou unitaires de fonctions logicielles critiques. Ces tests viennent avant d'autres tests approfondis.

Un test de fumée répond aux questions comme :

- Est-ce que le programme démarre correctement ?"
- Est-ce que les boutons de contrôle principaux fonctionnent ?"
- Pouvez-vous enregistrer un nouveau compte d'utilisateur de test vierge ?"

Si cette fonctionnalité de base échoue, il est inutile d'investir du temps dans un travail plus détaillé à ce stade.

Tests d'interface : À travers ce test, l'équipe s'assure que la présentation graphique est suffisamment attrayante pour être accepté d'abord par le Product Owner puis par les cibles.

À cet effet, on pourra recourir aux outils d'automatisation des tests.

Tests de régression (appelé par un abus de langage « test de non-régression ») : Les tests de régression, appelés TNR, sont des tests qui permettent de vérifier que des modifications n'ont pas entraîné d'effets de bord non prévus de nature à dégrader la qualité d'une version antérieurement validée.

1.3.2. Tests non fonctionnels :

Les tests non fonctionnels sont effectués pour vérifier les aspects non fonctionnels tels que les performances, l'utilisabilité, la fiabilité, etc. de l'application testée.

Les différents types de tests non fonctionnels sont les suivants :

Tests de performances : les tests de performances se fondent sur des indicateurs comme la consommation CPU, la progression du codage sur l'exploitation de la mémoire vive, le volume de commandes lancées à la seconde ou encore le mouvement d'entrée et de sortie des utilisateurs sur le logiciel. Ce sont ces paramètres qui définissent la position du logiciel sur un classement comparatif avec ses concurrents.

Tests de charge : Un test de charge consiste à effectuer un test permettant de mesurer la performance d'un système en fonction de la charge d'utilisateurs simultanées.

Test d'acceptation : Il s'agit ici de valider l'adéquation du logiciel aux spécifications du client. En fait, toute solution informatique est motivée par un besoin au niveau des utilisateurs. Alors, sur la base d'un cahier des charges arrêté et établi en amont avec le client, ce test rassure sur la conformité du logiciel aux critères d'acceptation et aux besoins des cibles. Ils sont donc généralement réalisés par le client final ou les utilisateurs, on peut aussi appeler cela la « recette » du logiciel.

Tests de volume : Le test de volume, comme son nom l'indique, est un test effectué sur de gros volumes de données. Il appartient à un groupe de tests non fonctionnels qui sont effectués dans le cadre de tests de performances où un produit logiciel ou une application avec un volume élevé de données est testé, comme un grand nombre de fichiers d'entrée, d'enregistrements de données ou une grande taille de table de base de données dans le système. Il est également connu sous le nom de test d'inondation.

Tests de sécurité : La sécurité devient un enjeu important pour les entreprises, il faut donc intégrer la sécurité comme une exigence non fonctionnelle dans les spécifications, définir les niveaux de sécurité et prévoir les tests de sécurité associés.

Tests de compatibilité : Ces tests interviennent généralement pour vérifier le bon fonctionnement du logiciel sur des terminaux ciblés notamment les systèmes d'exploitation après leur installation. Vous devez le réserver aux testeurs avancés afin que ceux-ci valident l'éligibilité de votre logiciel par rapport à des plateformes de destination.

Ainsi, par exemple il sera possible ici de définir les critères pour savoir si la solution peut être exécutée sans droits d'administration ou encore s'il est possible de la désinstaller en toute sécurité.

Tests d'installation : Les tests d'installation peuvent rechercher des erreurs qui se produisent dans le processus d'installation et qui affectent la perception et la capacité de l'utilisateur à utiliser le logiciel installé. De nombreux événements peuvent affecter l'installation du logiciel et les tests d'installation peuvent tester la bonne installation tout en vérifiant un certain nombre d'activités et d'événements associés.

Tests de fiabilité : Les tests de fiabilité du produit peuvent aider à prévoir le comportement futur pendant le cycle de vie complet du produit, du composant ou du matériau testé. En combinaison avec la mesure des paramètres pertinents et l'utilisation de méthodes analytiques appropriées, il est également possible d'identifier les modes de défaillance. Les tests de fiabilité permettent également de contrôler périodiquement les signes de fatigue et d'usure et d'évaluer la fonctionnalité sur la durée d'utilisation prévue du produit.

Tests d'utilisabilité : Test qui détermine la facilité avec lesquels les utilisateurs avec handicaps peuvent utiliser un composant ou un système.

Tests de conformité : c'est un processus de test pour déterminer la conformité d'un composant ou système (à ses exigences).

Tests de localisation : Les tests de localisation ont pour objectifs de vérifier qu'aucune erreur ne s'est glissée dans le processus de traduction, que le contenu traduit s'affiche correctement et que le produit localisé fonctionne comme prévu pour le marché cible

Quelles sont les activités de test à automatiser ?

La vie d'un testeur ne se résume fort heureusement pas à l'exécution de tests scriptés. Un testeur a de nombreuses activités variées qu'il serait trop long d'énumérer de manière exhaustive. On peut néanmoins citer des activités très fréquentes comme :

- La conception des tests
- La préparation des campagnes (données, environnements, tests …)
- La maintenance des tests
- L'exécution de sessions de tests exploratoires
- L'analyse des résultats des campagnes
- Implémenter et améliorer les processus de test
- Initier et contribuer au plan de test
- Créer et suivre les fiches d'anomalie
- En Agile : participer aux activités de l'équipe et faire des tâches non spécifiques au testeur …

D'autres activités ont moins de valeur ajoutée et peuvent donc être des candidats à l'automatisation (au moins en partie) très intéressantes. Notamment :

1.4.1. La maintenance des tests de régression

Cette maintenance peut vite être chronophage et fastidieuse. Elle est d'ailleurs une cause majeure des échecs des projets d'automatisation des tests. Fort heureusement, elle peut être automatisée en partie ! Une manière d'automatiser partiellement la maintenance est de concevoir un automate de test avec une architecture modulaire qui adopte les bonnes pratiques du code (notamment avec une bonne factorisation). Cela permet de mettre à jour en fois chaque étape commune à plusieurs tests !

1.4.2. L'Edition des bilans des campagnes

L'édition des bilans de campagnes est une tâche particulièrement importante. Le bilan est la vitrine des tests, c'est le document qui sera consulté par les non-testeurs, c'est le fruit de tout le travail du testeur sur une campagne. De manière générale, il semble donc un peu intéressant de l'automatiser. Néanmoins, on remarque qu'avec la multiplication des campagnes (avec l'automatisation des tests scripts) et la standardisation du contenu du bilan de ces dernières, cette tâche peut vite devenir fastidieuse. Dans ce cas, il devient nécessaire d'automatiser au moins en partie ces bilans en fonction du résultat de tests.

1.4.3. La gestion des environnements et des données de test

La création d'environnement de Test est primordiale pour pouvoir tester efficacement, c'est d'ailleurs un élément obligatoire. Il est conseillé d'avoir un environnement crée spécialement pour chaque campagne (cela assure la présence des données) pour être impacté le moins possible par des éléments extérieurs.

Il est important de rappeler qu'il n'existe pas d'environnement de test et de test sans données de test. Il est donc nécessaire de créer les environnements avec les données nécessaires ou encore de générer ces données après la création des environnements, ce qui peut permettre, dans certains contextes, de vérifier que la création d'éléments proposés par le logiciel fonctionne correctement.

1.4.4. L'écriture des cas de test après leur conception

La conception est une étape essentielle et peu automatisable. Néanmoins, après avoir conçu son test, si ce test est destiné à être scripté, il faut l'écrire. Cette écriture se doit être rigoureuse et, si possible, permettre une maintenance limitée.

Les outils de MBT comme Yest et MaTelo aident fortement à l'automatisation de cette partie en proposant même d'aller jusqu'à l'écriture de script automatisés pour Yest et une exécution de ces derniers pour MaTelo.

1.4.5. Création des Fiches d'anomalies

Les fiches d'anomalies sont les livrables particulièrement importants dans lesquels il faut mettre autant d'information que possible, afin que n'importe quel acteur sur ce logiciel (métier, testeur, développeur) puisse les reproduire et les comprendre. Ces paramètres peuvent être très nombreux et il est facile d'en oublier. Là encore l'automatisation peut nous aider en préremplissant des informations permettant de reproduire l'anomalie.

Processus Simplifié de Test :

Ce qu'il faut comprendre de manière générale Tester un logiciel c'est réaliser l'exécution d'un programme pour y trouver des défauts. Dans le cas des tests automatisés nous exécutons un script sur le logiciel pour tester ses fonctionnalités.

Ce processus est valable pour tout type de test :

Etape 1 : On définit un cas test (CT) à exécuter

Un cas de tests correspond à un scénario de tests à exécuter afin de mesurer la bonne réalisation, ou non, du résultat attendu, selon une donnée de tests (Jeux de données ou JDD). Cela permet de

comparer la réalité avec le résultat attendu. De ce fait, la rédaction de chacun des cas de tests doit lister :

- Le type de test à exécuter et son scénario d'exécution.
- Les données de tests qui doivent être utilisées.
- Le résultat attendu.

Etape 2 : On détermine les données de test (DT) à fournir au CT (concrétisation)

Les données de test sont des données passées en entrée à un cas de test. Il est primordial d'avoir les bonnes données adaptées à nos cas de test.

Etape 3 : On indique le résultat que l'on attend pour ce CT (les résultats attendus)

Ce sont les résultats attendus de l'exécution des cas de test.

Etape 4 : On exécute le script de test testant le CT sur les DT.

Le script représente le code qui lance le programme sur les données de test ou jeu de données.

Etape 5 : On compare le résultat obtenu aux résultats attendus (verdict)

Les résultats obtenus après l'exécution du script sont comparés aux résultats attendus pour donner un verdict qui représente le résultat du test.

Etape 6 : On rapporte le résultat du test : succès / échec

Si les résultats sont conformes on rapporte le test à succès, dans le cas contraire à échec.

Pour une meilleure compréhension nous y reviendrons avec un exemple pratique dans le chapitre 2.

Les dérives liées aux tests automatiques :

Conclusion

Pour de réussir l'automatisation de toute activité de test, il faut définir :

- Les objectifs liés à l'automatisation de l'activité en question
- Identifier les solutions existantes et étudier un retour sur investissement (financier ou non) potentiel
- Choisir une solution, qui peut être un logiciel acheté ou un développement interne, si l'on décide d'initier l'automatisation de l'activité à la suite du point précédent
- Implémenter la solution à travers un POC (Proof Of Concept)
- Prioriser les tâches
- Faire un bilan et mesurer les avancées

CHAPITRE II : PRATIQUES DES TESTS AUTOMATISES DANS LA GRANDE DISTRIBUTION (CAS DE L'ENSEIGNE GRAND FRAIS)

1. Présentation de l'enseigne Grand frais

Cf document fonctionnel de formation de la Grande Distribution

2. Présenter des applications de Sigale

Cf document fonctionnel de formation de la Grande Distribution

3. Présentation de l'environnement de travail

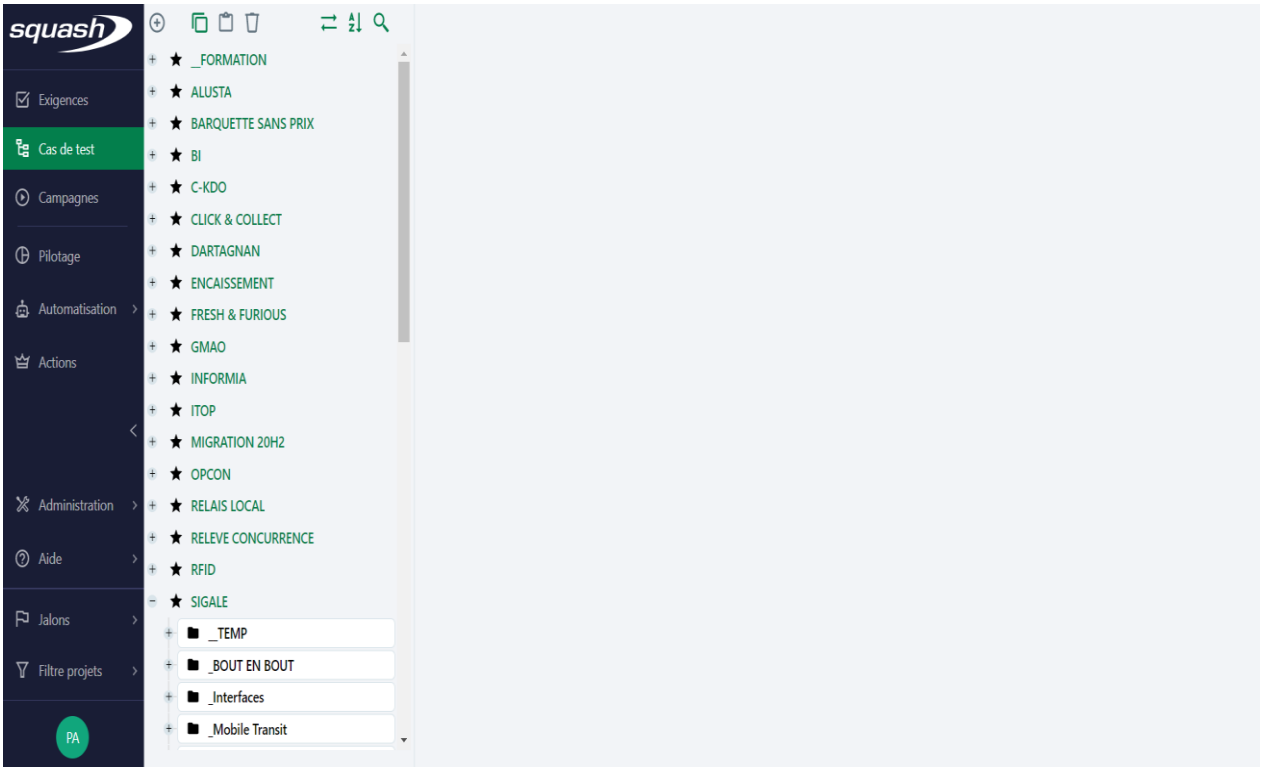
3.1. Gestionnaire de cas de Tests : SQUASH

Squash est une suite d'outils pour concevoir, automatiser, exécuter et industrialiser les tests logiciels. Basée sur un socle open source, la solution est modulable et facilement intégrable. Elle s'adapte à tous vos contextes projet : Cycle en V, Agile et agilité à l'échelle type SAFE.

Dans le cas de l'entreprise Prosol, les projets adopte la méthode agile. Le Squash TM est utilisé pour :

- Concevoir et exécuter les cas de tests
- Gérer les différentes campagnes de test
- Gérer les TNR
- Paramétrer les jeux de données pour les tests

Le lien d'accès Squash TM pour Prosol : http://squash.prosol.pri:8080/squash/



3.2. Gestion de tests automatisés : JENKINS

Jenkins : qu'est-ce que c'est ?

Jenkins est un ordonnanceur permettant l'administration et la centralisation des tâches liées à l'intégration continu. Il s'agit d'un logiciel open source, développé à l'aide du langage de programmation Java. Il permet de tester et de rapporter les changements effectués sur une large base de code en temps réel. En utilisant ce logiciel, les développeurs peuvent détecter et résoudre les problèmes dans une base de code et rapidement. Ainsi les tests de nouveaux builds peuvent être automatisés, ce qui permet d'intégrer plus facilement des changements à un projet, de façon continue. L'objectif de Jenkins est en effet d'accélérer le développement de logiciels par le biais de l'automatisation. Jenkins permet l'intégration de toutes les étapes du cycle de développement.

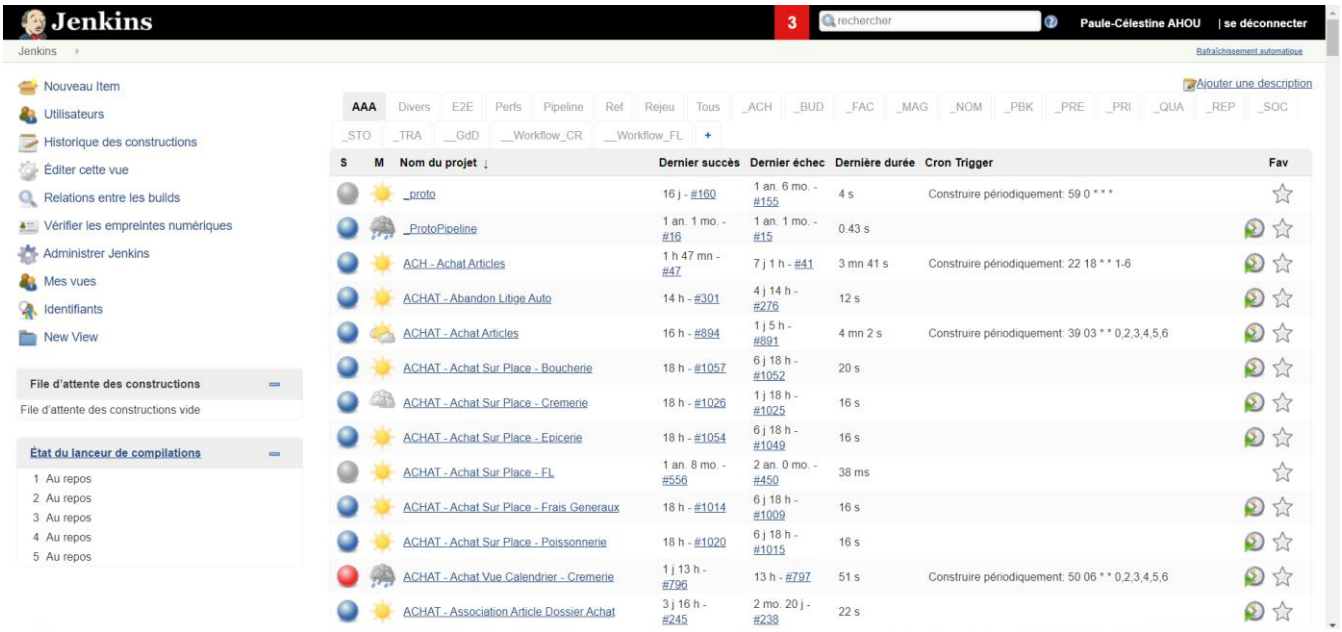
Quels sont les avantages de Jenkins ?

Jenkins présente plusieurs avantages. Il s'agit d'un outil open source fédérant une vaste communauté proposant sans cesse de nouvelles améliorations et autres perfectionnements. Le logiciel est facile à installer, et plus de 1000 plugins sont disponibles. Si un plugin correspondant à vos besoins n'existe pas, vous pouvez le créer vous-même et le partager avec la communauté. Autre avantage : Jenkins est également gratuit. Enfin, en tant qu'outil développé avec Java, il peut être porté sur toutes les principales plateformes logicielles.

Par ailleurs, Jenkins se distingue de la plupart des autres outils d'intégration continue par plusieurs points. Tout d'abord, Jenkins est adopté de manière bien plus large que ses concurrents. Au total, on dénombre 147 000 installations actives et plus d'un million d'utilisateurs autour du monde. L'autre force de Jenkins est son interconnexion avec plus de 1000 plugins permettant de l'intégrer à la plupart des outils de développement, de test et de déploiement.

Comment fonctionne Jenkins ?

Un développeur insère son morceau de code dans le répertoire du code source. Jenkins, de son côté, vérifie régulièrement le répertoire pour détecter d'éventuels changements. Lorsqu'un changement est détecté, Jenkins prépare un nouveau build. Si le build rencontre une erreur, l'équipe concernée est notifiée. Dans le cas contraire, le build est déployé sur le serveur test. Une fois le test effectué, Jenkins génère un feedback et notifie les développeurs au sujet du build et des résultats du test.



3.3. Présentation de Git :

Qu'est-ce que Git ?

C'est un système de contrôle de version (version control system) et un ensemble d'outils logiciels pour :

- Mémoriser et retrouver différentes versions d'un projet.
- Faciliter le travail collaboratif.

Il a été initialement développé par Linus Torvald pour faciliter le développement du noyau Linux. C'est un logiciel libre / open source disponible sur toutes les plates-formes.

Pour l'installation

- Téléchargez le programme d'installation de Git pour Windows sur le site officiel : http://git-scm.com/download/win
- Une fois téléchargé, exécutez le programme d'installation (Git-2.33.1-64-bit.exe). Cela ne prendra qu'une minute.

Qu'est-ce que Gitlab ?

GitLab est une application web basée sur git qui permet de gérer :

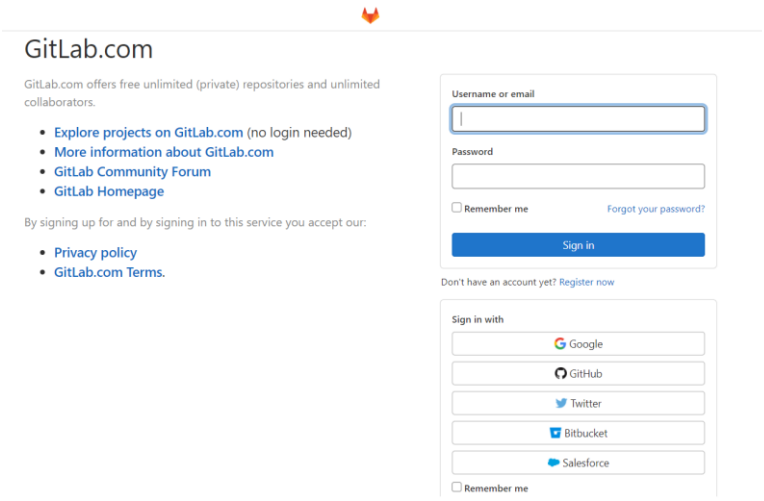
- Le cycle de vie de projets git.
- Les participants aux projets (rôles, groupes, etc.).
- La communication entre ces participants.

Elle a été développée par une société privée :

- La ''Community Edition'' est libre.
- Un service gratuit en ligne

Pour créer son compte Gitlab, rendez-vous sur le site officiel :

https://gitlab.com/users/sign_in?__cf_chl_jschl_tk__=pmd_m7OOUklmRxPNPH5xU_4DD2soRHrhsvDEQaPwRYUuKmo-1635432275-0-gqNtZGzNAlCjcnBszQq9



3.4. Présentation de Cypress

Cypress est un outil d'automatisation de test open source similaire à Sélénium Web Driver, WebDriverIO, TestCafe et autres qui dispose d'une communauté active et réactive.

Cypress requiert des connaissances en JavaScript et s'adresse plutôt aux personnes ayant des affinités techniques. Une autre particularité de Cypress est sa capacité à contrôler le trafic qui passe sur son IHM notamment avec des bouchons. Néanmoins, certaines APIs, comme les API fetch, ne sont pas encore supportées.

Les tests sont assez rapides à écrire et propose des simplifications. Le lancement des tests peut se faire de deux manières : soit par ligne de commande ou directement avec l'IHM. Les résultats et logs fournis sont assez complets et aident les développeurs à déboguer rapidement.

Cypress propose une version payante qui contient un Dashboard en ligne, tout le reste est gratuit.

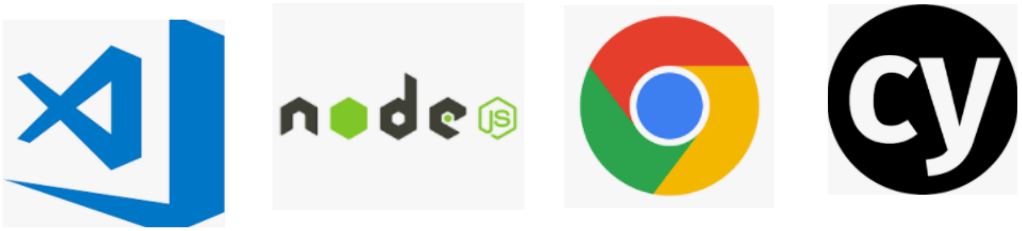
3.5. Description de l'existant :

Les tests automatisés étaient développés avec le Framework Protactor. Malheureusement Google envisage de le supprimer. Pour pallier à ce problème nous utilisons maintenant le Framework Cypress pour nos tests automatiques.

3.6. Déploiement de l'environnement de travail

Pour travailler avec Cypress vous devez remplir certaines conditions. Les principales conditions sont notamment :

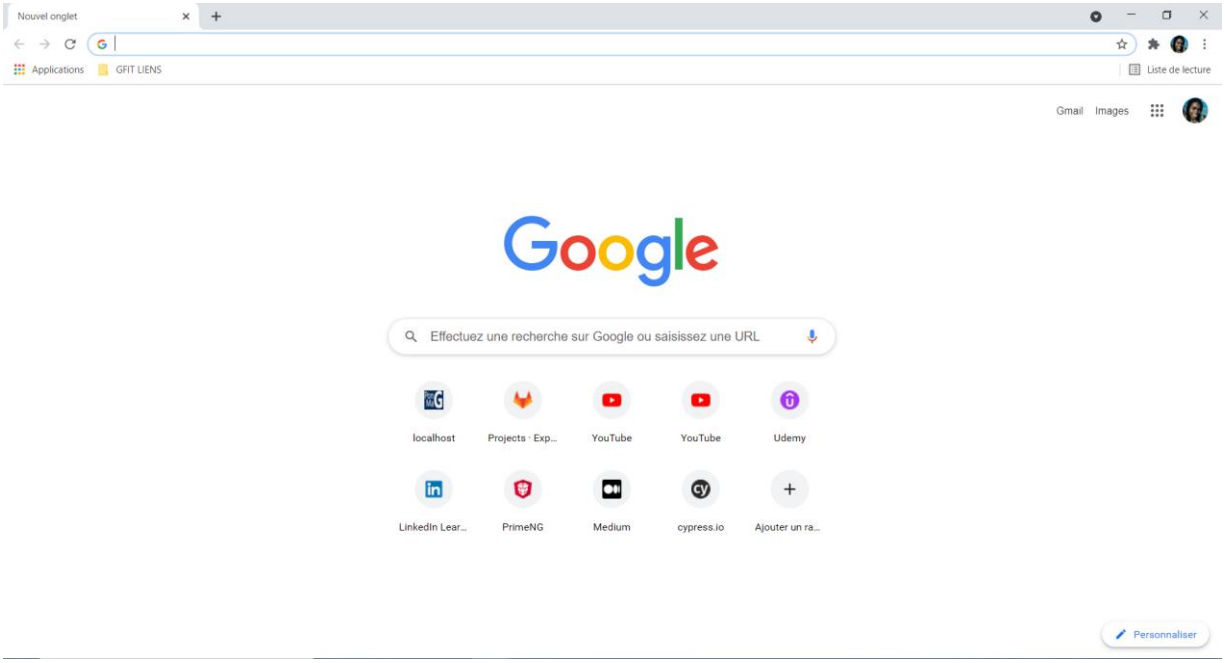
- Avoir un IDE installé sur son ordinateur (Visual Studio Code de préférence)
- Installer Node Js
- Avoir un Navigateur (Google Chrome de préférence)



Installer le navigateur Chrome

Google Chrome est un navigateur Web rapide et gratuit. Avant de le télécharger, vérifiez si Chrome est compatible avec votre système d'exploitation et assurez-vous de disposer de la configuration système requise.

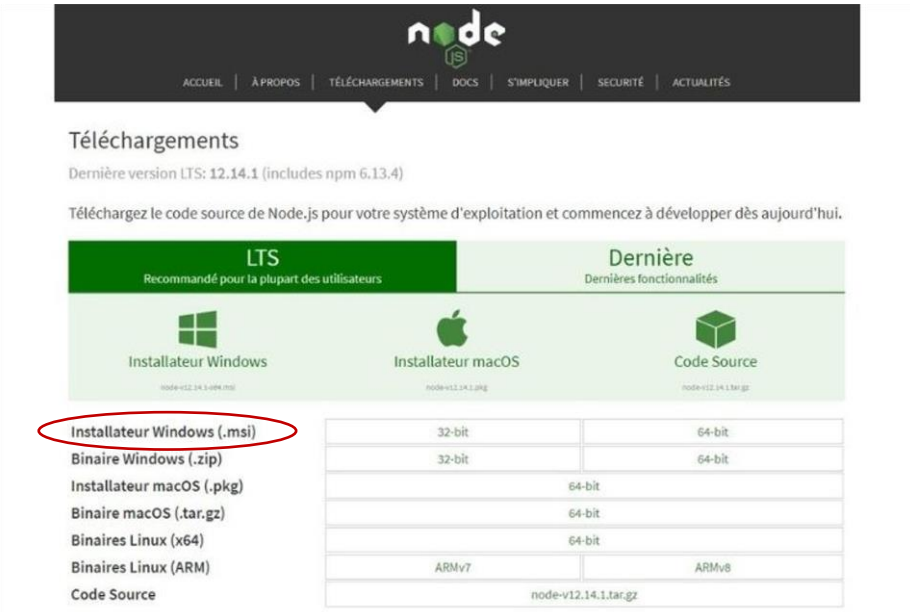
- Téléchargez le fichier d'installation : https://www.google.com/intl/fr/chrome/
- Si vous y êtes invité, cliquez sur Exécuter ou sur Enregistrer.
- Si vous choisissez Enregistrer, double-cliquez sur le téléchargement pour lancer l'installation.
- Lancez Chrome :



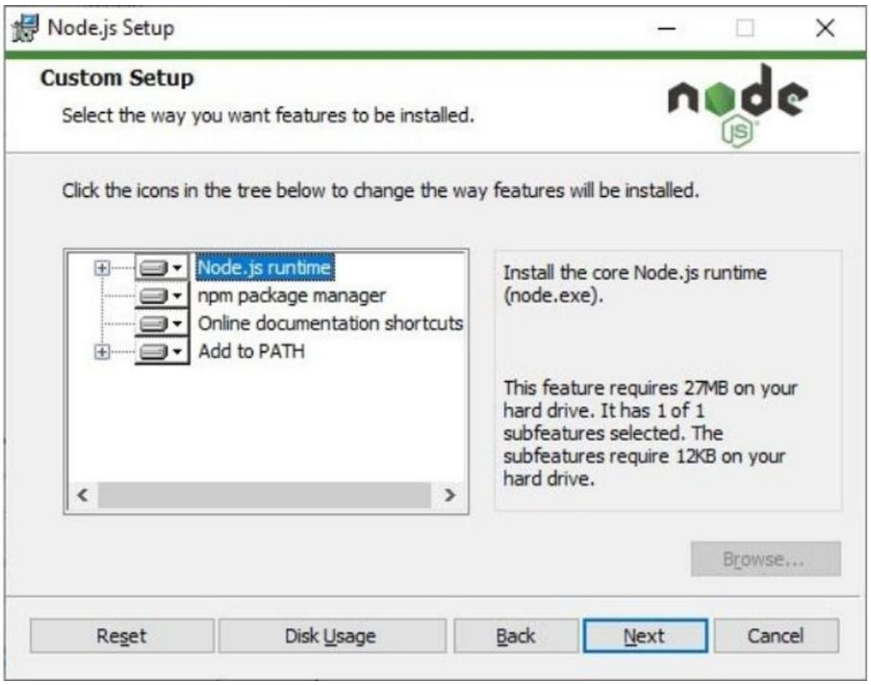
Installation de Node Js

Comme pour la plupart des logiciels, la procédure pour installer Node.js sur Windows commence par une visite sur la page du site officiel : https://nodejs.org/fr/download/

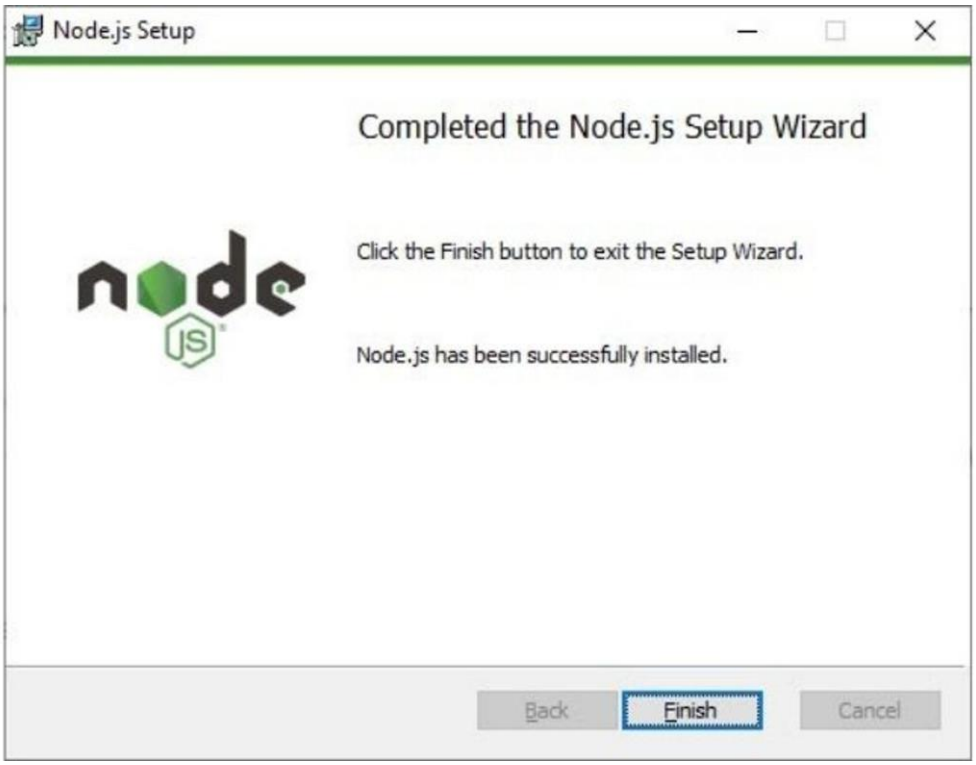
Choisissez l'installateur Windows (.msi)



Après les habituelles étapes d'acceptation de la licence et du choix du dossier d'installation, le programme d'installation vous propose de choisir vos fonctionnalités et leur méthode d'installation.



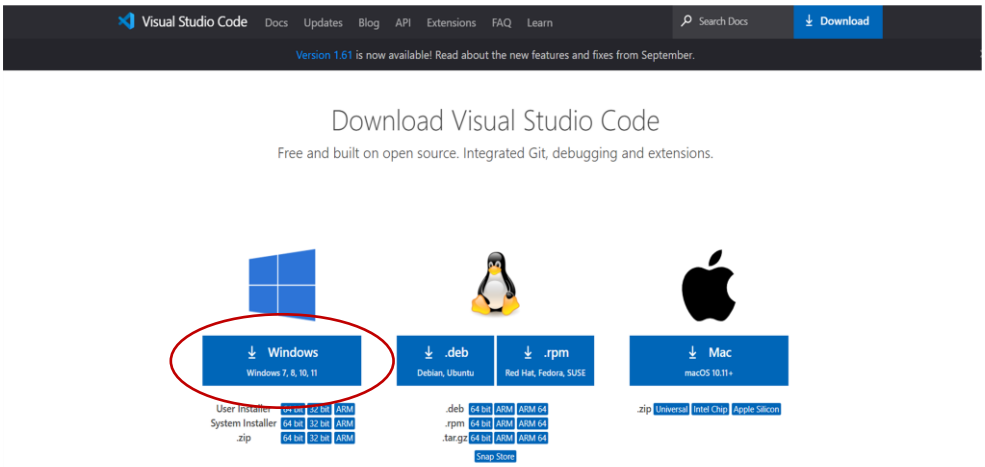
L'installation se lance et se termine, normalement, par un joli bouton "Finish". Bravo, vous avez installé Node.js sur Windows !



Installation Visual studio code

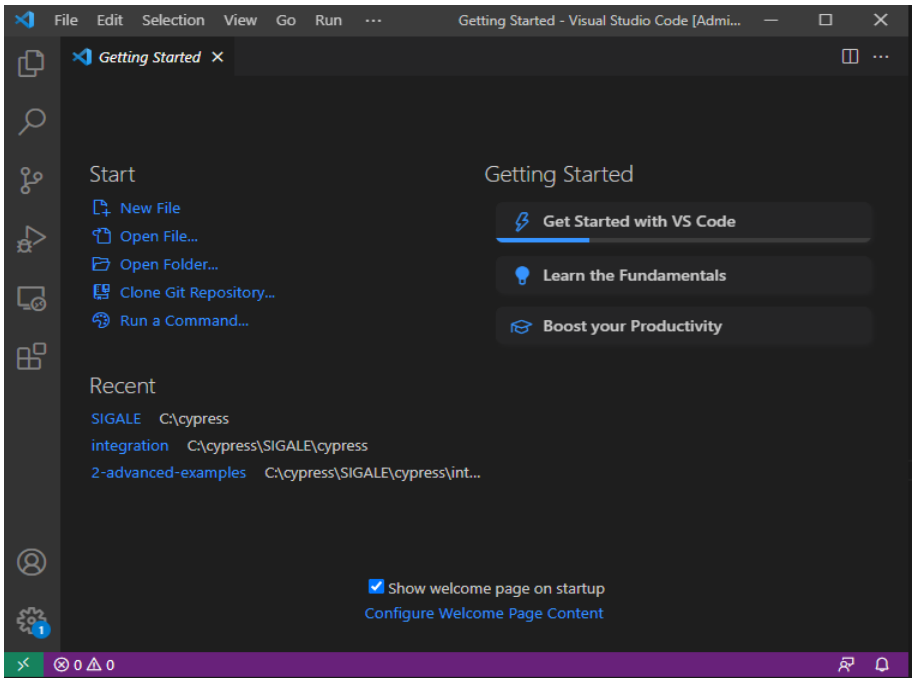
Installation

- Téléchargez le programme d'installation de Visual Studio Code pour Windows.
- Une fois téléchargé, exécutez le programme d'installation (VSCodeUserSetup-x64-1.61.2.exe). Cela ne prendra qu'une minute.



- Par défaut, VS Code est installé sous :

  C:\users\{username}\AppData\Local\Programs\Microsoft VS Code.



Installation de Cypress

Installer Cypress avec npm install dans l(invite de commande ou dans le terminal de Visual studio code. Aller sur le répertoire de votre projet : C:\cypress\SIGALE\

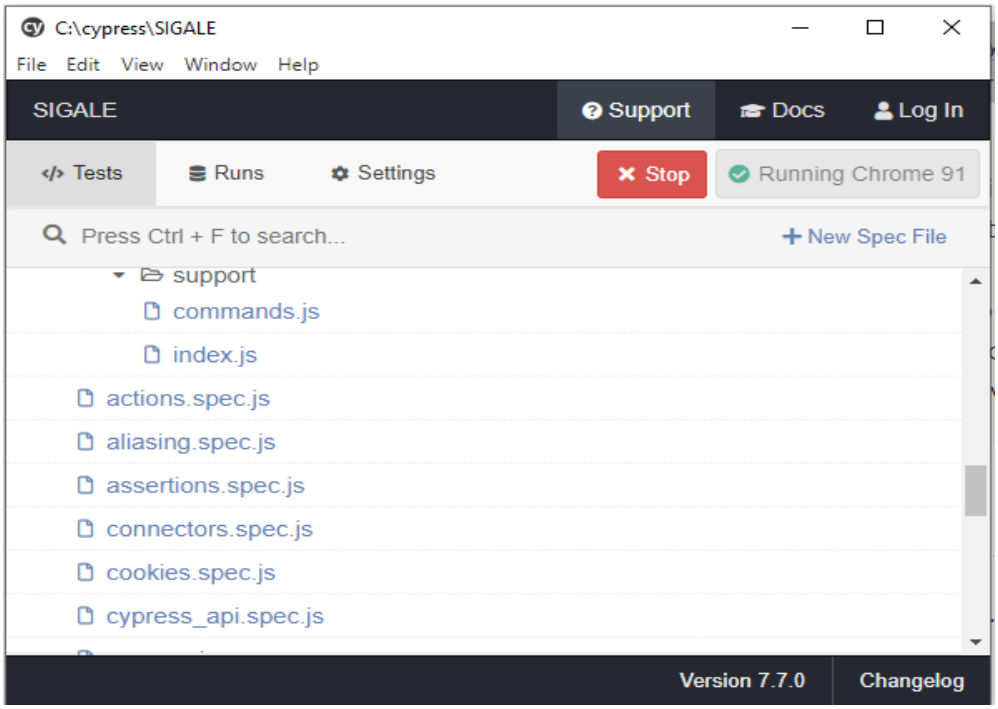
```
cd C:\cypress\SIGALE\
```

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

```
/Users/bmann/Desktop/playground
$ npm install cypress --save-dev

> cypress@0.20.3 postinstall /Users/bmann/Desktop/playground/node_modules/cypress
> node index.js --exec install

Installing Cypress (version: 0.20.3)

 ✔ Downloaded Cypress
 ✔ Unzipped Cypress
 ✔ Finished Installation /Users/bmann/Desktop/playground/node_modules/cypress/dist/Cypress.app

You can now open Cypress by running: node_modules/.bin/cypress open

https://on.cypress.io/installing-cypress
+ cypress@0.20.3
updated 1 package in 44.615s
$ ./node_modules/.bin/cypress open
```

Lancer Cypress :

```
npx cypress open
```



3.7. CREATION DE VOTRE PREMIER TEST

3.7.1. Description des possibilités

Commençons la rédaction notre test, deux options s'offrent à vous.

Un test peut être un test de réussite dit Test passant, le principe est d'inculquer un chemin qu'on va qualifier de positif à notre logique.

L'inverse n'étant pas proscrit, vous pouvez aussi opter pour un chemin négatif ou Test non passant.

3.7.2. Cas de test passant

Un cas de test passant est un test qui produit un résultat sans erreur et entraine une autre action. Prenons l'exemple d'un utilisateur qui souhaite se connecter à une application. Il renseigne un bon Login et le mot de passe qui y est associé. Dans ce cas, le résultat sera l'authentification réussie et l'action qui sera lancée est l'affichage de la page d'accueil.

3.7.3 Cas de test non passant

Un cas de test non passant est un test qui n'est pas censé produire un résultat. Un peu obscur ? Vous allez comprendre.

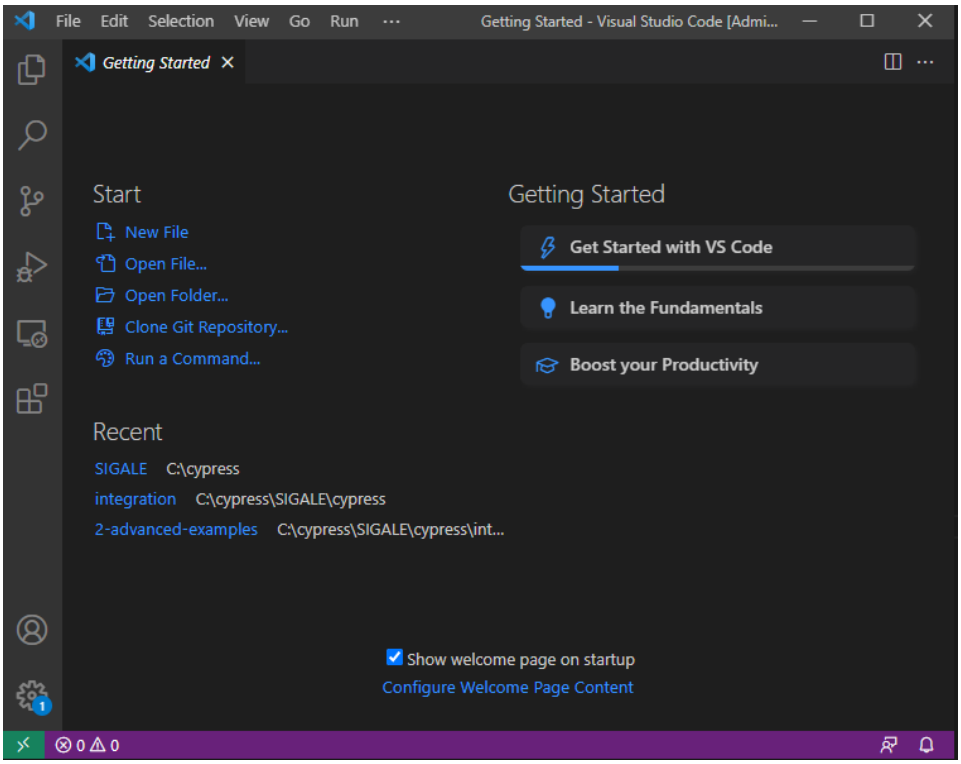
Prenons l'exemple d'un utilisateur qui en voulant s'authentifier renseigne un bon Login mais fait une erreur dans son mot de passe. Dans ce cas, le résultat doit être une incapacité à se connecter. Si vous préférez avoir un le bon Login et le mauvais Mot de passe ne vous sert à rien et n'entraine aucune action à sa suite.

3.7.4 Rédaction du test automatisé

Tous les tests que vous allez rédiger et présenter sont réalisés sur SIGALE Qualité (URL : http://app1.int.sigale.prosol.pri:9094/login).

Pour commencer ouvrez votre IDE, dans mon cas il s'agit de Visual Studio Code.

Une fois que c'est fait, choisissez votre fichier de test et ouvrez-le : **C:\cypress\SIGALE\cypress**



Dans les prochaines parties, nous mettrons en pratique les cas décrits dans la **Description des possibilités**.

Pour ce faire, adoptons une approche heuristique en utilisant une carte mentale et utilisons un logiciel tel que Freemind pour répondre à ce besoin.

FreeMind permet de créer des cartes mentales et des diagrammes pour mettre en évidence des connexions entre vos idées grâce à sa structure systémique. Les idées sont réorganisées afin de mettre en relief de nouvelles informations et de booster la créativité. Il vous aide ainsi à organiser en plans vos concepts et vos projets dans le but de représenter graphiquement un concept. Classifier

par groupes et hiérarchiser des données favorise ainsi leur mémorisation, leur compréhension et leur planification.

- **Chercher l'identifiant ou l'attribut de la balise dans le DOM**

Première étape, lancer une page Web ici il s'agit de SIGALE Qualité.

Qu'est-ce que le DOM ?

Le Document Object Model ou **DOM** (pour modèle objet de document) est une interface de programmation pour les documents HTML, XML et SVG à laquelle vous avez accès en effectuant un clic droit avec souris puis Inspecter ou encore en appuyant la touche F12 sur votre clavier.

Il vous donne la visibilité sur les balises HTML et leurs attributs qui permettront à nos tests de localiser précisément l'élément qu'on lui demande de tester.

3.7.5 Cas de test passant

Exemple de Test Passant sur SIGALE Qualité:

```
describe('cas passant',() =>{
    it('tout bon', () => {
        cy.visit('http://app1.int.sigale.prosol.pri:9094')
        cy.viewport(1920, 1080)
        cy.get('[name="login"]').type('lunettes')
        cy.get('[name="password"]').type('glasses')
        cy.get('[type="submit"]').click()
        cy.get('.titre > div').should('be.visible')
        cy.get('[class="text-danger text-center mt-2"]').should('not.exist')
    })
})
```

Résultat dans le navigateur :

3.7.6 Cas de test non passant

Exemple de Test non Passant sur SIGALE Qualité:

```
describe('cas non passant',() =>{
    it('mauvais MDP', () => {
        cy.visit('http://app1.int.sigale.prosol.pri:9094')
        cy.get('[name="login"]').type('lunettes')
        cy.get('[name="password"]').type('Azer01234')
        cy.get('[type="submit"]').click()
        cy.get('[div="_ngcontent-vnw-c83"]').should('not.exist')
        cy.get('[class="text-danger text-center mt-2"]').should('be.visible')
    })
})
```

Résultat dans le navigateur :

3.7.7 PROCESSUS D'EXECUTION

Commandes de lancement

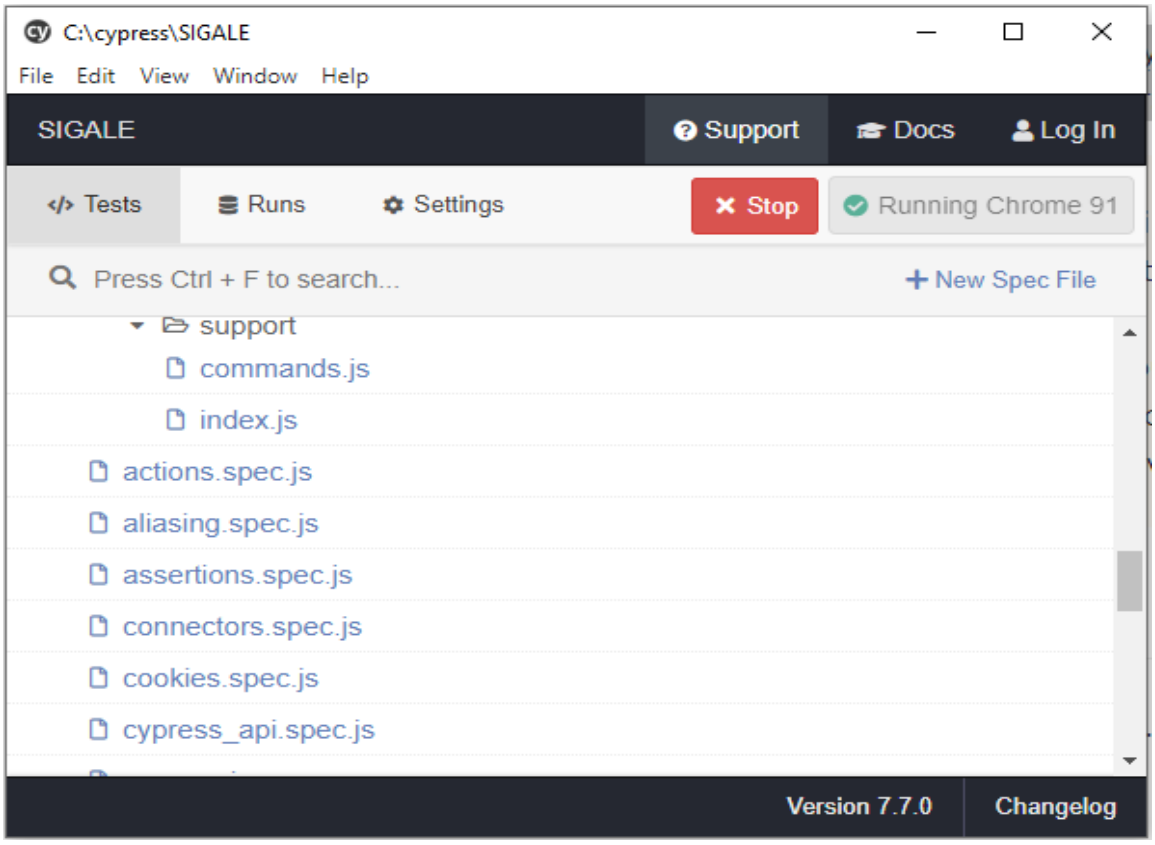
Lançons votre test dans le navigateur via une commande à saisir dans le Terminal de votre IDE ou dans l'invite de commande mais assurez-vous d'être dans le bon répertoire :

```
PS C:\cypress\SIGALE> npx cypress open
```

Rôle du navigateur

Cypress ouvre le test dans un navigateur installé sur votre système. Si vous avez Google Chrome, il sera choisi en priorité (mais tout ceci reste paramétrable).

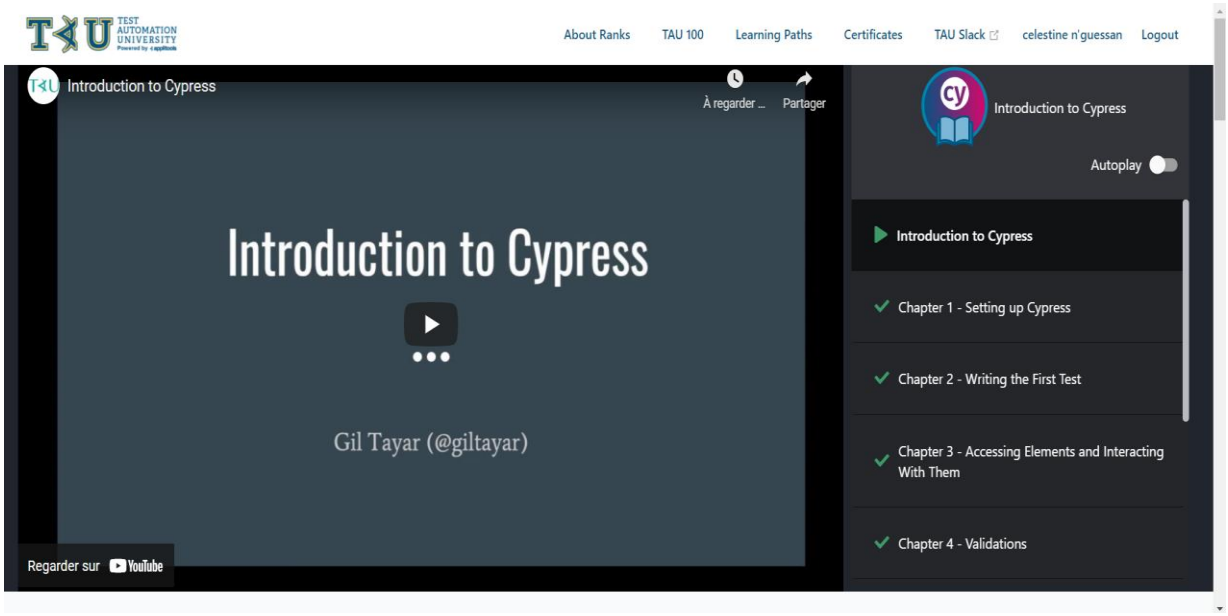
Si le test rencontre un problème lors de son exécution, il se chargera aussi d'afficher l'erreur et de vous indiquer à quelle ligne de votre code le test échoue.



Pour aller plus loin dans Cypress consulter ces différents cours en faisant les quiz pour obtenir des certificats pour votre apprentissage personnel :

https://testautomationu.applitools.com/cypress-tutorial/

https://testautomationu.applitools.com/advanced-cypress-tutorial/



CHAPITRE III : Exemple de processus de développement des Tests D'intégration Automatisés Cypress pour les applications Sigale DONS et Sigale QUALITE

1. Cloner le Projet : http://happybox.sigale.prosol.pri/gitlab/recette/cypress

- Pour git
  - Ajouter son nom et son email

```
git config --global user.name "Prénom NOM"
git config --global user.email "MY_NAME@example.com"
```

  - Ajouter les alias

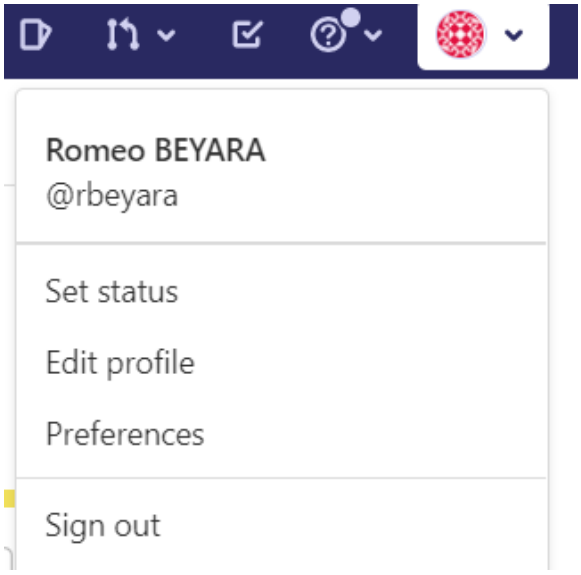
```
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.ci commit
git config --global alias.st status
```

  - Créer sa clé ssh (**sans passphrase**) et l'uploader sur gitlab

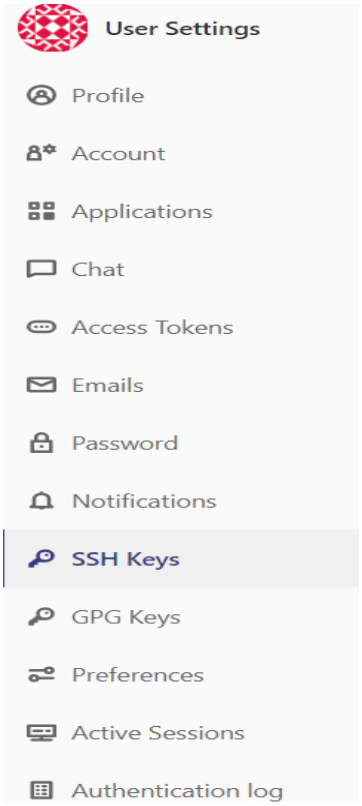
```
ssh-keygen
```

Vous devez générer une clé ssh avec la commande "ssh-keygen", cela va créer une clé qui sera dans un fichier. Normalement "C:\Users\"nom utilisateur"\.ssh\id_rsa.pub"

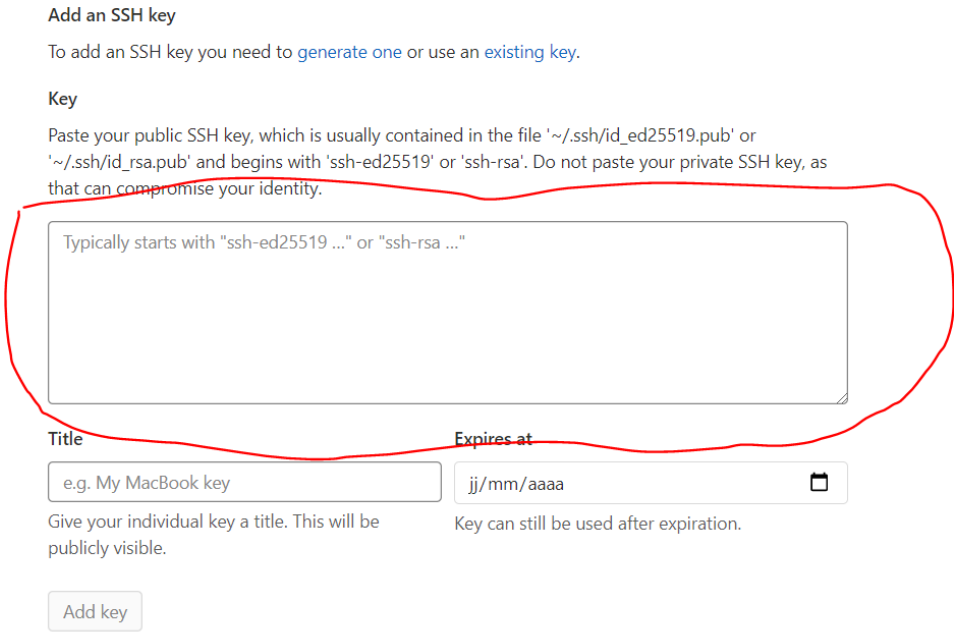
Maintenant sur Gitlab connectez-vous et aller sur votre profil :



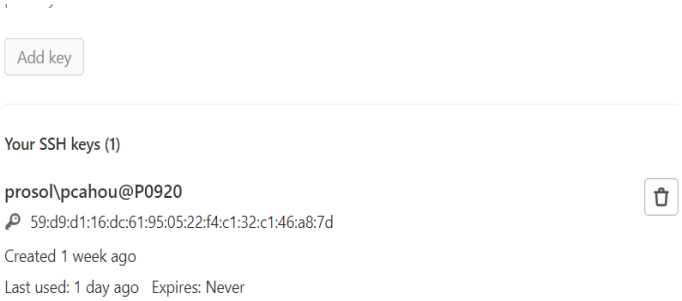
Cliquez sur onglet SSH Keys :



Vous copiez le contenu du fichier id_rsa.pub ensuite cliquez sur add Key

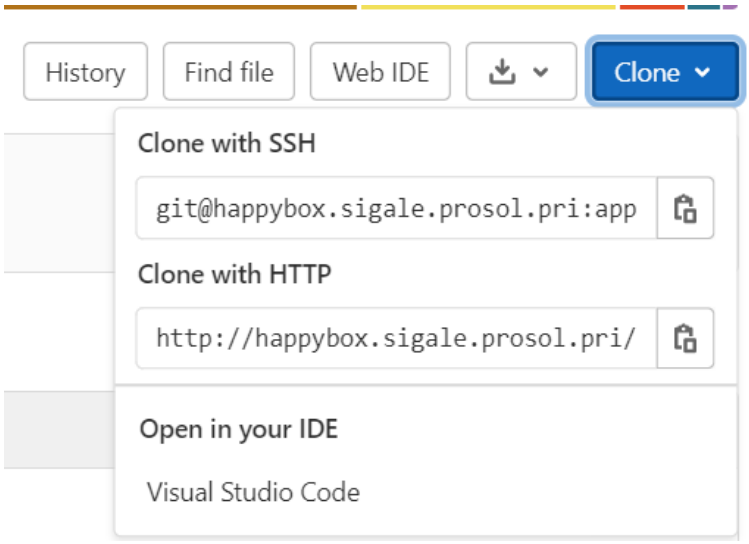


Cette action génère votre clé SSH



Maintenant vous allez sur le projet à cloner :

http://happybox.sigale.prosol.pri/gitlab/recette/cypress/

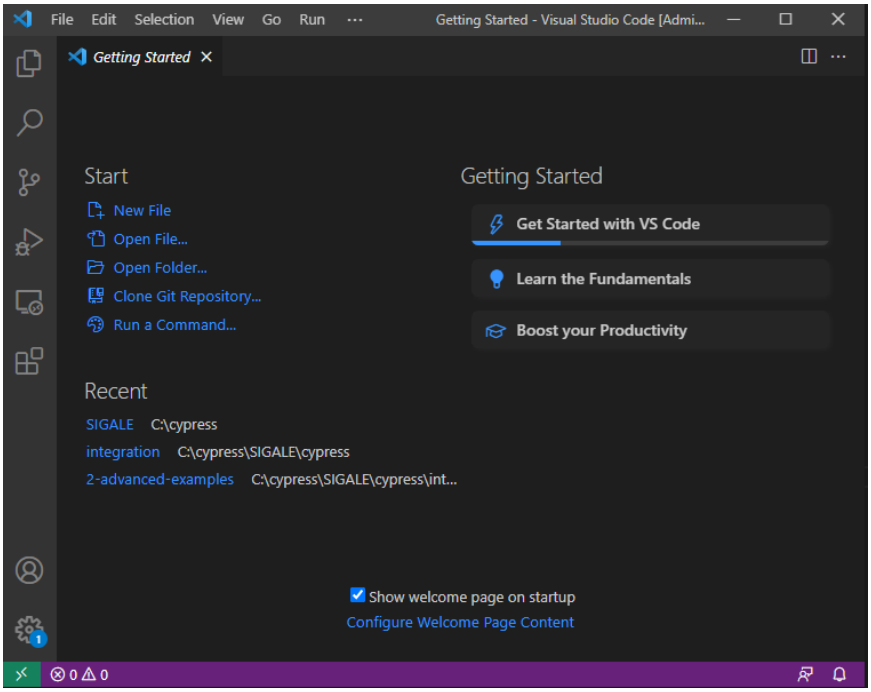


Vous cliquez sur clone et vous copiez soit le 1ier lien ou le deuxième lien

Ouvrez votre IDE Visual Studio Code :

Une fois que s'est fait choisissez votre ficher de test et ouvrez-le :

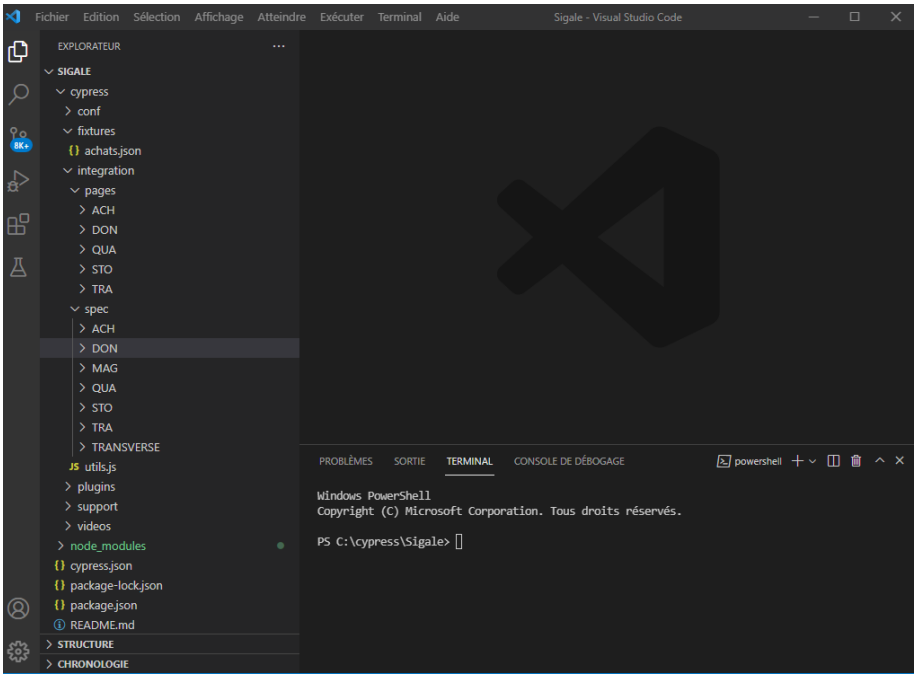
C:\cypress\SIGALE\cypress



Aller dans le terminal de Visual Studio Code et lancer la commande :

```
C:\cypress\SIGALE\> git clone http://happybox.sigale.prosol.pri/gitlab/recette/cypress.git
```

Vous obtenez cette image :



Le projet a été bien cloné :

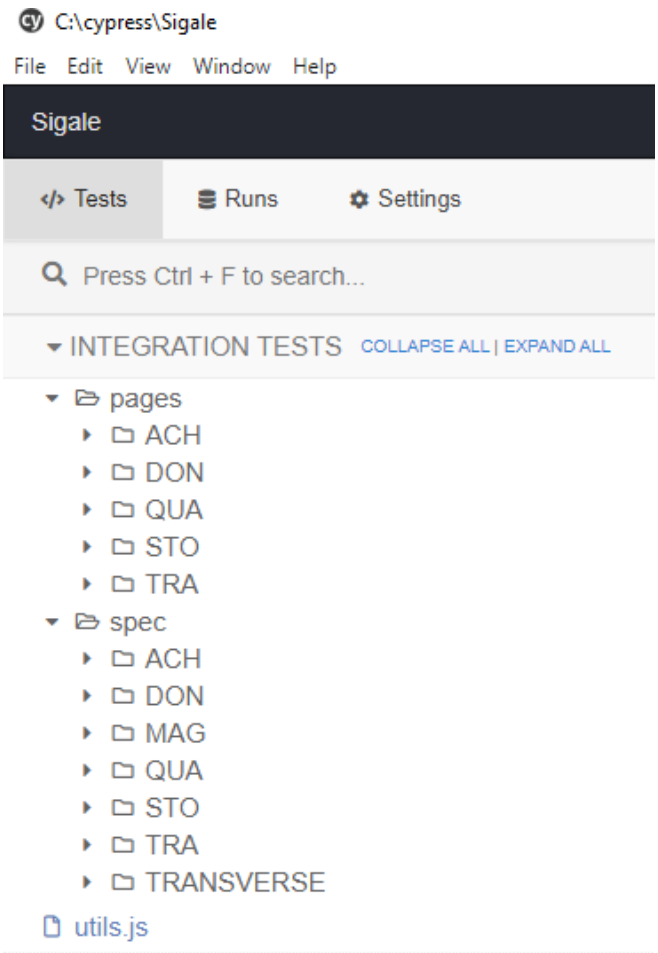
Aller dans le terminal de Visual studio code sur la racine du projet et lancer la commande pour installer Cypress dans le projet

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

Ensuite lancer cette commande pour ouvrer Cypress :

```
PS C:\cypress\Sigale> npx cypress open
```

Cypress ouvre le projet dans le navigateur :



2. Prise en main du Projet :

1. Créer un répertoire dédié qui contiendra tous les tests (*.spec.js) dans le répertoire spec.
   Soit :
   C:\cypress\Sigale\cypress\integration\spec\QUA
   C:\cypress\Sigale\cypress\integration\spec\DON

2. Créer un répertoire dédié qui contiendra tous les Pages Objects (*.page.js) dans le répertoire pages.
   Soit :
   C:\cypress\Sigale\cypress\integration\pages\QUA
   C:\cypress\Sigale\cypress\integration\pages\DON

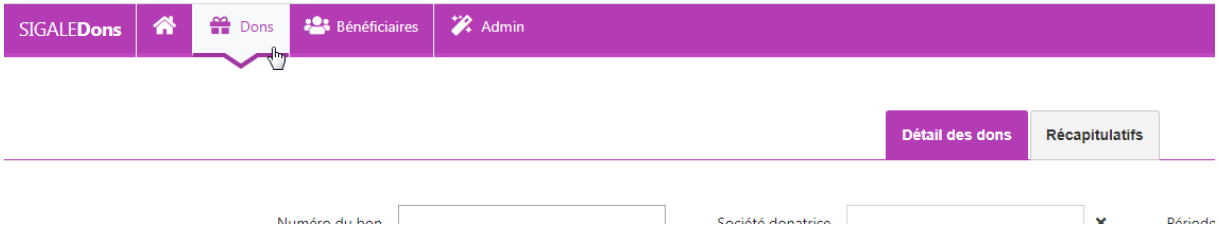
3. Sur la base du fichier C:\cypress\Sigale\cypress\integration\pages\ACH\menu.js créer les fichiers :
   C:\cypress\Sigale\cypress\integration\pages\QUA\menu.js
   C:\cypress\Sigale\cypress\integration\pages\DON\menu.js
   Les adapter en s'inspirant du modèle.

   Rq : les valeurs contenues dans l'objet this.onglets sont subjectives et seront exploitées à partir des tests.

   Ex :
   accueil : ['rechercheLot', 'syntheseAchatsCourants'],
   seront appelés sous la forme :

```
var sNomPage = 'accueil';
it ('Onglet [SYNTHESE DES ACHATS COURANTS] - Click', () => {
    menu.clickOnglet(sNomPage, 'syntheseAchatsCourants')
})
```

4. En s'inspirant de la PO C:\cypress\Sigale\cypress\integration\pages\ACH\besoins-besoins_consolides_fournisseurs.page.js
   créer les pages contenant les sélecteurs contenus sur les applications Sigale QUALITE et DONS.
   1 PO correspondant à 1 page Web et 1 onglet.
   Subdiviser en onglets si nécessaire.
   Exemple :



Donnera :
dons-detail_dons.page.js
don-recpaitulatif.page.js

Donnera :
beneficiares-beneficiaires.page.js
beneficiaire-suivi_attestations.page.js

Le but de cette règle de nommage et de permettre de retrouver plus rapidement dans quel fichier (PO) se trouve le sélecteur concerné.

Dans la mesure de possible, créer des noms de méthodes qui soient explicite en mentionnant dans l'ordre :

a. Le type de composant (Exemple : bouton, champ input, liste déroulante)
b. La fonction associée au composant (Ex : Enregistrer)
   Ex : buttonCreerBesoinConsolFourn()

Si le sélecteur est contenu dans une popin, utiliser la convention de nommage suivante :

a. pP (Pour « PoPin »)
b. le nom de la popin (Ex : bcf -> Création Besoin Consolidé Fournisseur)
c. Le type de composant (Exemple : liste déroulante , bouton, champ input)
d. La fonction associée au composant (Ex : Liste des plateformes)
   Ex : pPbcfListBoxPlateforme()

Cf. exemples : C:\cypress\Sigale\cypress\integration\pages\ACH\besoins-besoins_consolides_fournisseurs.page.js

5. Sur la base du test C:\cypress\Sigale\cypress\integration\spec\ACH\IHM_achats.spec.js créer les tests d'IHM suivants :
   C:\cypress\Sigale\cypress\integration\spec\DON\IHM_dons.spec.js
   C:\cypress\Sigale\cypress\integration\spec\QUA\IHM_qualite.spec.js

Les différentes tâches liées au Projet

1. Mise en place d'un TA pour l'IHM des applications Dons et Qualité

   Sous tâches :

   1. Cypress Tuto :

      Suivre et assimiler les concepts de ces tutoriels :

      https://testautomationu.applitools.com/cypress-tutorial/

      https://testautomationu.applitools.com/advanced-cypress-tutorial/

   2. Cypress Socle DON ou Qualité

      Sur la base du document : T:\GFIT\Etudes\Refonte SI\15_Recette\TEST\PROJETS\SIGALE\Automatisation\TA\Cypress\Développement TA Cypress v1.0.docx

      Préparer :

      - ✓ L'arborescence des répertoires
      - ✓ Le fichier menu.js

   3. Cypress IHM DON ou IHM QUALITÉ

      Référencer dans les Pages Object les éléments du DOM qui seront exploités ultérieurement.

      Respecter :

      - La convention de nommage des fichiers

      - La convention de nommage des éléments du DOM

      Cf. Wiki : http://happybox.sigale.prosol.pri/gitlab/recette/cypress/-/wikis/home

      Créer le TA chargé de vérifier la présence de tous ces éléments.

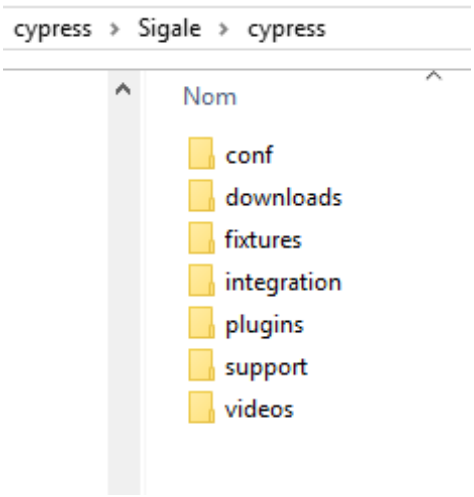
Documentation sur les bonnes pratiques de développement des TA

Architecture du Projet

L'arborescence des répertoires est la suivante :

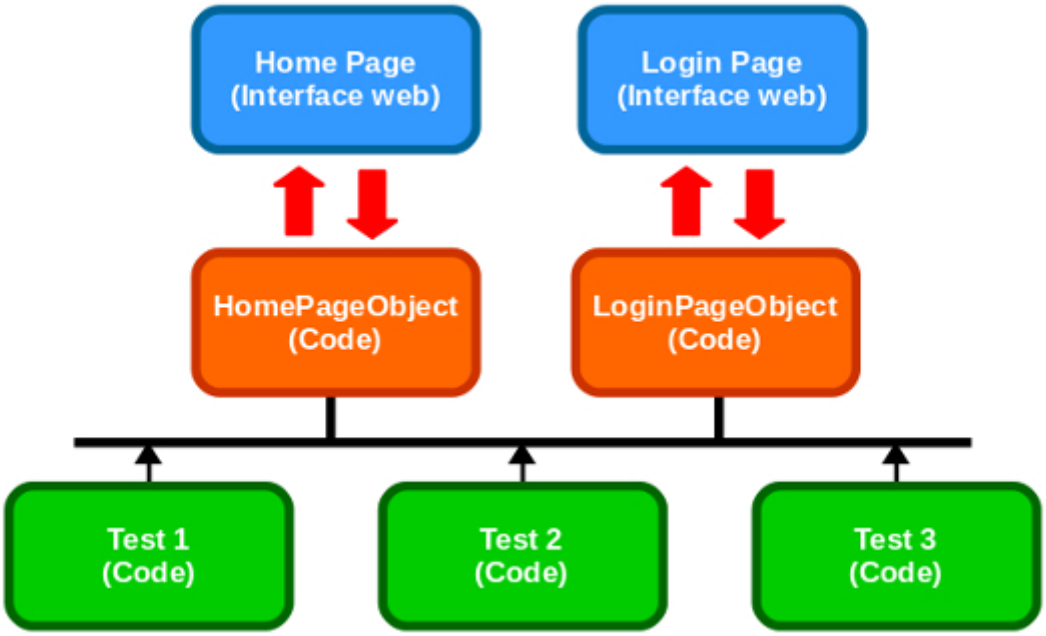
- conf
  - environnements
- fixtures
- integration
  - pages
  - spec
- plugins
- screenshots
- support
- videos

Sur Windows :



Utilisation des Page Object Model (POM).

Le design pattern "page object" est un modèle de conception qui permet de structurer un code de tests automatisés dans le but d'éviter les problèmes de maintenabilité.
Les Page Object doivent permettre de manipuler, observer et rechercher des éléments d'une page.



Les Avantages :

- La réduction de la quantité de code dupliqué
- La réutilisabilité des classes page Object pour différents tests
- Une maintenance plus facile en cas de modification de l'interface utilisateur
- Un code plus lisible et plus compréhensible

Le nom des éléments contenus dans ces pages doit conserver une cohérence de nommage afin de permettre de retrouver rapidement et sans ambiguïté un élément du DOM.

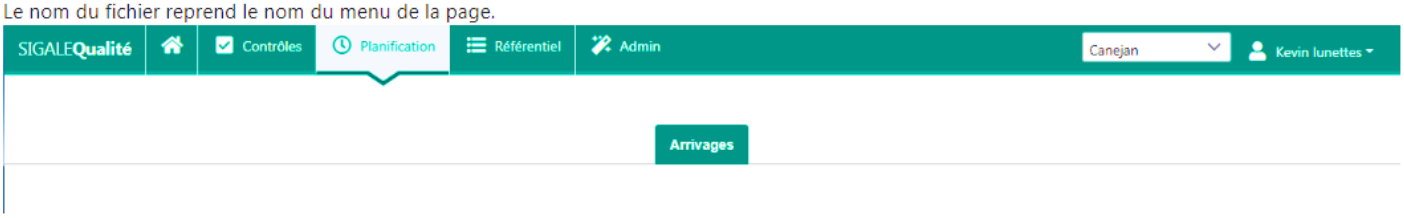
Règles de nommage

1. Objet de page

Dans la mesure du possible, appuyez sur la structure du site web pour déterminer le nom du fichier.

Si la page contient 0 ou 1 onglet

Le nom du fichier reprend le nom du menu de la page.



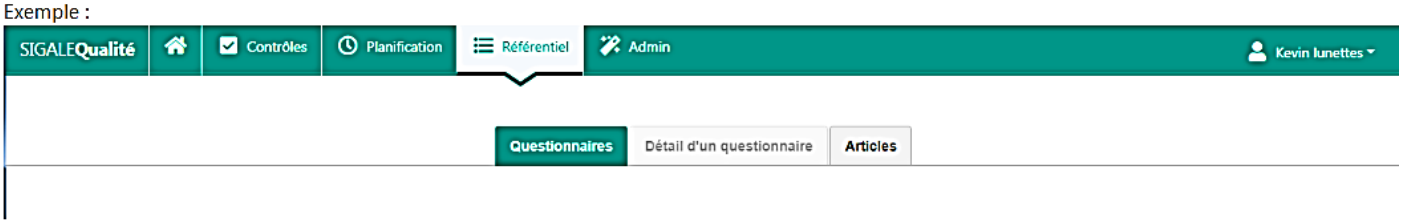
Exemple de nom de fichier :

- *planification.page.js*

Si la page contient plus d'un onglet,

Au nom du menu ajouter le nom de l'onglet séparé par un caractère souligné (_).

Exemple :



Exemples de noms de fichiers :

- *planification_questionnaires.page.js*
- *planification_detail-questionnaire.page.js*
- *planification_articles.page.js*

---

2. Essais

Dans la mesure du possible, nommez le test en commencant par le nom de la fonctionnalité testée suivie de l'action réalisée sur cette fonctionnalité.
Exemple :

- *campagne_creation.spec.js*
- *campagne_modification.spec.js*
- *campagne_bilan.spec.js*
- *campagne_bilan_export.spec.js*
- *campagne_suppression.spec.js*

Cette règle de nommage permet de regrouper fonctionnellement les tests.

Nom des éléments

Dans la mesure de possible, créez des noms de méthodes qui soient explicites en mentionnant dans l'ordre :

1. Le type de composant (Exemple : bouton, champ input, liste déroulante)
2. La fonction associée au composant (Ex : Enregistrer)

Ex : *buttonCreerBesoinConsolFourn()*

Si le sélecteur est contenu dans une popin, utiliser la convention de nommage suivante :

1. pP (Pour « PoPin ou Popup »)
2. le nom de la popin (Ex : bcf -> Création Besoin Consolidé Founisseur)
3. Le type de composant (Exemple : liste déroulante , bouton, champ input)
4. La fonction associée au composant (Ex : Liste des plateformes)

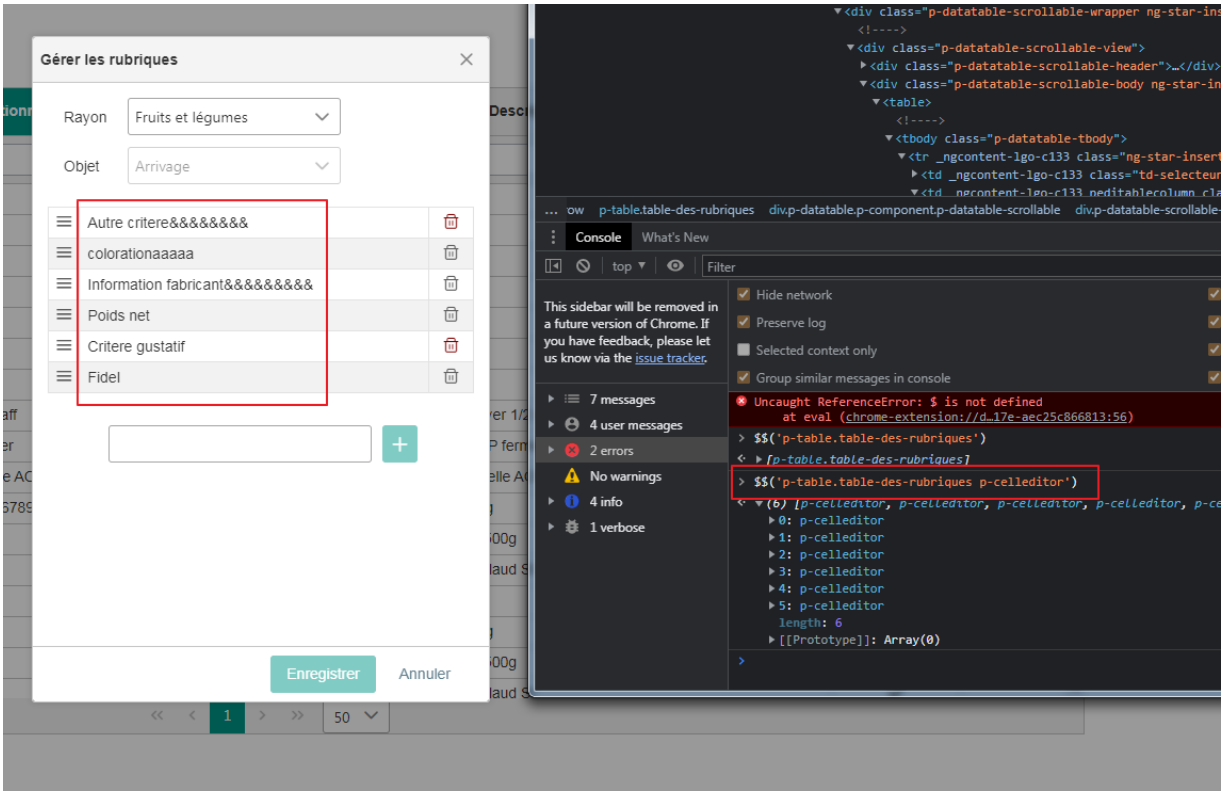
Ex : *pPbcfListBoxPlateforme()*

Il faut être homogène et cohérent lorsque vous écrivez le code

Des fois c'est écrit : POPIN XXXXXX >
des fois c'est écrit : Popin [XXXXX] >
La seconde écriture est la bonne.

Bien choisir vos sélecteurs afin d'avoir des éléments assez uniques et éviter qu'ils soient trop long.

Utiliser des sélecteurs CSS les plus précis possibles et le moins dépendant du Xpath (la moindre sélection sous l'élément sélectionné rend le sélecteur caduque)



Il faut plus de rigueur au niveau de la présentation du code :

- Indentation
- Espacement des blocs de codes (conserver un homogénéité 1 à 2 lignes selon le contexte)
- Regroupement entre les éléments identiques (au niveau des sélecteurs)
- Structurer et organiser le code afin qu'il soit lisible

Un exemple :

Tu proposes :

```
buttonCreerQuestionnaire() { return cy.get('app-footer-bar button.sans-icone').eq(0) }
buttonModifier() { return cy.get('.footerBar button').eq(1) }   //('app-footer-bar button span.fa-pencil-alt')
buttonDuppliquer() { return cy.get('.footer-bar button').eq(2) };                    //('app-footer-bar button span.fa-copy')
buttonGererRubrique() { return cy.get('app-footer-bar button.sans-icone').eq(1) }
buttonImprimer() { return cy.get('app-footer-bar button em.fa-print') };
```

Je propose :

```
buttonCreerQuestionnaire() {  return cy.get('#p-tabpanel-11 app-footer-bar button.sans-icone').eq(0) }
buttonModifier() {            return cy.get('#p-tabpanel-11 app-footer-bar button span.fa-pencil-alt') }
buttonDuppliquer() {          return cy.get('#p-tabpanel-11 app-footer-bar button span.fa-copy') }
buttonGererRubrique() {       return cy.get('#p-tabpanel-11 app-footer-bar button.sans-icone').eq(1) }
buttonImprimer() {            return cy.get('#p-tabpanel-11 app-footer-bar button em.fa-print') };
```

Vois tu une différence ?

Indiquer dans les commentaires à quel popin fait référence le sélecteur.

Il peut y avoir plusieurs popin sur une même page.

Exemple :

```
41    this.tdCommande                    = element.all(by.css('td.datagrid-commande span'));
42    this.tdPrevision                   = element.all(by.css('td.datagrid-previsionDeCommandeJPlus1 span'));
43    this.tdStock                       = element.all(by.css('td.datagrid-stockMagasin span'));
44
45    //--- Popin [Magasins n'ayant pas commandé pour la date d'expédition du {dd/mm/yyyy}] ----------------------------------------
46    this.buttonPPFermer                = element(by.xpath('//*[@id="form-magasins-sans-commande"]/modal/div/div[4]/a'));
47
48    //-- Popin [Choix des besoins consolidés fournisseur à transmettre] ------------------------------------------------------------
49    this.pDataGridBesoinsConsolidesFour = element.all(by.css('.popup-selection-bcf th'));
50
51    this.pCheckBoxAllClients           = element.all(by.css('.popup-selection-bcf th input')).get(0);
52    this.pCheckBoxListeClients         = element.all(by.css('.popup-selection-bcf td input'));
53
```

Ne pas s'appuyer sur le texte des boutons pour les localiser.

Le site étant multilingues, le TA doit pourvoir continuer de fonctionner quel que soit la langue sélectionnée.

Ne pas mettre d'accent dans le nom des méthodes ou fonctions utilisés dans le code.

Le nom des méthodes doit respecter la convention de nommage.

Si le sélecteur est contenu dans une popin, utiliser la convention de nommage suivante :

1. pP (Pour « PoPin ou PoPup »)
2. le nom de la popin (Ex : bcf -> Création Besoin Consolidé Founisseur)
3. Le type de composant (Exemple : liste déroulante , bouton, champ input)
4. La fonction associée au composant (Ex : Liste des plateformes)

Ex : *pPbcfListBoxPlateforme()*

Respecter le nom des composants (liste déroulante ? Liste de check box ?, etc.)

Eviter du code mort et les doublons

Quand vous souhaiter intervenir sur un élément précis d'un sélecteur qui existe en plusieurs exemplaires (Exemple : Les checkBox contenus dans une DataGrid), si vous avez besoin de sélectionner le premier élément trouvé dans votre test, n'indiquez pas :dans la PO

```
checkboxQuestionnaires() { return cy.get('.xxxxxxx').eq(0) }
```

et dans le test (exemple) :

```
checkboxQuestionnaires().click();
```

Il vaut mieux faire :dans la PO

```
checkboxQuestionnaires() { return cy.get('.xxxxxxx') }
```

et dans le test (exemple) :

```
checkboxQuestionnaires().eq(0).click();
```

Ceci est plus facilement maintenable !La PO retourne un tableau d'éléments et le test choisi l'élément cible. C'est plus facile à maintenir et c'est plus polyvalent pour les prochains tests qui exploiterons par exemple le dernier élément de la série...

```
checkboxQuestionnaires().eq(34).click();
```

ou mieux !

```
checkboxQuestionnaires().last().click();
```

Pour ces 2 Tests la PO est inchangée.Et c'est un peu l'esprit de la centralisation / mutualisation des PO...

Choisir des sélecteurs uniques possible afin d'éviter d'utiliser le moins possible
La commande eq().
Exemple :

Ce que tu proposes :

```
buttonDemarrerControle() {       return cy.get('.footerBar.btn-toolbar button').eq(0) }
buttonReprendreControle() {      return cy.get('.footerBar.btn-toolbar button').eq(1) }
buttonVisualiserControle() {     return cy.get('.footerBar.btn-toolbar button').eq(2) }
buttonImprimerResultat() {       return cy.get('.footerBar.btn-toolbar button').eq(3) }
```

La Bonne astuce :

```
buttonDemarrerControle() {       return cy.get('.footerBar > :nth-child(1)') }
buttonReprendreControle() {      return cy.get('.footerBar > :nth-child(2)') }
buttonVisualiserControle(){      return cy.get('.footerBar > :nth-child(3)') }
buttonImprimerResultat() {       return cy.get('.footerBar > :nth-child(4)') }
```