Tuto Test

*Les tests pour une application web React servent à garantir la qualité et la stabilité de l'application en détectant les erreurs et en vérifiant que les fonctionnalités se comportent comme prévu. Voici les différents types de tests couramment utilisés pour une application web React :*

Les tests unitaires :

Ils permettent de tester individuellement chaque composants d’une application, Les tests unitaires sont importants pour détecter les erreurs précocement, isoler les problèmes, fournir une documentation vivante, faciliter la maintenance, accélérer le développement et améliorer la qualité du code.

Outils : Jest et React Testing Library (*npm install --save-dev @testing-library/react* *Et JEST : npm install –save-dev jest*

Important ! Les fichiers test doivent terminer par .test afin de permettre

Création d’un dossier tests dans les dossier src

Création d’un fichier AddTodo.test.js

Nous pouvons maintenant coder les tests en JavaScript avec le framework React, ainsi que les bibliothèques de test React Testing Library et Jest.

Fonctions React Testing Library :

Voici quelques-unes des fonctions les plus couramment utilisées de la bibliothèque de test React Testing Library :

* render(component) : Rend le composant dans un environnement de test virtuel pour pouvoir interagir avec lui et effectuer des assertions.
* screen.getByText(text) : Sélectionne un élément en fonction de son contenu textuel.
* screen.getByRole(role) : Sélectionne un élément en fonction de son rôle (par exemple, "button", "input", etc.).
* screen.getByPlaceholderText(placeholderText) : Sélectionne un élément en fonction de son attribut de texte de substitution (placeholder).
* screen.getByTestId(testId) : Sélectionne un élément en fonction de son attribut data-testid.
* screen.getAllByRole(role) : Sélectionne tous les éléments correspondant à un rôle spécifique.
* screen.queryBy\* : Fonctions similaires aux getBy\*, mais elles renvoient null si l'élément n'est pas trouvé au lieu de générer une exception.
* fireEvent.\* : Permet de déclencher des événements sur des éléments simulés (par exemple, fireEvent.click(element) pour simuler un clic sur un élément).
* waitFor : Permet d'attendre que des conditions asynchrones se réalisent, par exemple l'affichage d'un élément à l'écran ou la fin d'une action asynchrone.
* act : Permet de synchroniser les mises à jour d'état de React pendant les tests.

Fonctions de JEST :

Voici quelques-unes des fonctions couramment utilisées dans le framework de test Jest :

* jest.fn() : Crée une fonction simulée (mock function) à des fins de test
* test(name, fn) : Déclare un nouveau test avec une fonction de test associée.
* expect(value) : Utilisé pour effectuer des assertions sur les valeurs.
* toBe(value) : Vérifie si une valeur est égale à une autre valeur spécifiée.
* toEqual(value) : Vérifie si une valeur est égale à une autre valeur spécifiée, en effectuant une comparaison récursive pour les objets et les tableaux.
* toBeDefined() : Vérifie si une valeur est définie (non undefined).
* toBeUndefined() : Vérifie si une valeur est undefined.
* toBeNull() : Vérifie si une valeur est null.
* toBeTruthy() : Vérifie si une valeur est évaluée comme true.
* toBeFalsy() : Vérifie si une valeur est évaluée comme false.
* toHaveBeenCalled() : Vérifie si une fonction a été appelée.
* toHaveBeenCalledTimes(number) : Vérifie le nombre de fois où une fonction a été appelée.
* toHaveBeenCalledWith(...args) : Vérifie si une fonction a été appelée avec des arguments spécifiques.
* toThrow() : Vérifie si une fonction lance une erreur.

Mes tests unitaires :

Le premier test sera « Ajoute une todo lorsque le bouton est cliqué »

* Créé une fonction simulée avec jest.fn ()
* Rendre le composant AddTodo dans un environnement de test avec render(component)
* On sélectionne l’input correspondant à l’ajout de todo (avec le placeholder = « Add a todo » )
* Sélection du bouton avec screen.getByText("Add")
* On change la valeur de l’input avec fireEvent.change()
* Clic sur ce bouton avec fireEvent.click
* Vérifier que la fonction est appellé qu’une seule fois avec toHaveBeenCalledTimes(1)
* Vérifier si une fonction a été appelée avec la valeur entrée dans l’input grâce à toHaveBeenCalledWith(...args)

Le deuxième test sera « Ajoute une todo lorsqu'on presse Enter »

* Même logique que au dessus sauf qu’au lieu d’appuyer sur le bouton avec .click, on simule la pression de la touche enter avec .keyDown

Le troisième test sera « Ne doit pas ajouter de todo lorsque l'input est vide »

* Créé une fonction simulée avec jest.fn ()
* Rendre le composant AddTodo dans un environnement de test avec render(component)
* On simule le clic sur le bouton
* Puis on vérifie simplement que la fonction simulé n’a pas été appellée avec .not.toHaveBeenCalled()

Le quatrième sera « un message doit apparaitre (Chargement en cours ...) lors de l'ajout d'une todo »

* Créé une fonction simulée avec jest.fn ()
* Rendre le composant AddTodo dans un environnement de test avec render(component)
* On sélectionne l’input correspondant à l’ajout de todo (avec le placeholder = « Add a todo » )
* Sélection du bouton avec screen.getByText("Add")
* On change la valeur de l’input avec fireEvent.change()
* Clic sur ce bouton avec fireEvent.click
* Repérer le texte « chargement en cours » et vérifier sa présence dans le document

Différences test unitaire/ test d’intégration

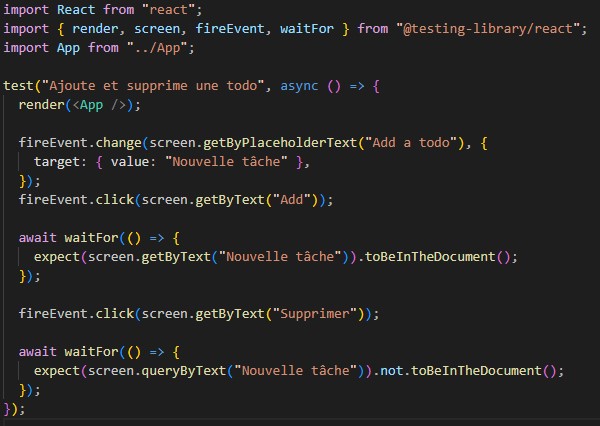
Les tests unitaires se concentrent sur la validation des fonctionnalités individuelles de manière isolée, tandis que les tests d'intégration vérifient le bon fonctionnement et les interactions entre les différentes parties d'un système.

Les tests d’intégration :

Ils permettent de tester l’ensemble de l’application en vérifiant les interactions entre les différents composants, donc le bon fonctionnement de l’application dans son ensemble. Ces tests permettent de détecter les éventuels problèmes d’intégration et contribuent à garantir la qualité et la stabilité de l'application en vérifiant que tous les composants se connectent et fonctionnent correctement ensemble.

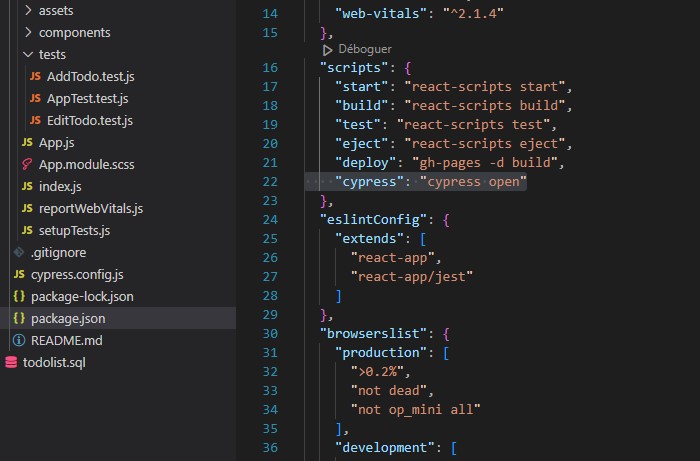
Pour effectuer ce test j’ai repris les mêmes bibliothèques que pour les tests unitaires. J’ai choisi de tester l’interaction entre deux composants de l’application qui sont AddTodo et TodoItem c’est-à-dire que je vais ajouter une todo puis la supprimer :

* Rendre le composant App dans un environnement de test avec render(component), ainsi on a accés à tous les compossant
* On sélectionne l’input correspondant à l’ajout de todo (avec le placeholder = « Add a todo » )
* Sélection du bouton avec screen.getByText("Add")
* On change la valeur de l’input avec fireEvent.change()
* Clic sur ce bouton avec fireEvent.click
* On vérifie que le todo est bien ajouté à l’écran
* On clique sur le bouton « supprimer »
* On vérifie que le todo n’apparait plus à lécran



Les test End-to-End :

Outils : CYPRESS (npm i cypress --save-dev)

Après avoir télécharger CYPRESS, il faut ajouter le script de démarrage dans le fichier package.json :

Puis exécuter la commande npx cypress open dans votre terminal. Cela créera le dossier de configuration et les exemples de tests par défaut.

Maintenant que le Framework est téléchargé et prêt. On peut coder un parcours utilisateur représenter par trois tests dans le fichier spes.cy.js .

Mon Premier test sera « Ajoute une tâche et la supprime »

Comment réaliser ce test ?

* Sélectionner l’input avec le sélecteur placeholder= ‘Add a todo’ et cy.get pour selectionner
* Simuler l’envoi de la valeur « Nouvelle tâche » avec cy.type()
* Créer un événement de clic sur le bouton « Add » avec cy.click()
* Vérifier si la tâche est visible sur la page grâce à cy.should qui permet de créer une assertion pour savoir si la condition donnée est vraie ou fausse, ici la condition sera « be.visible ».
* Attendre un peu…. Avec cy.wait()
* Supprimer la tâche en selectionnant le bouton supprimer avec cy.contains et cy.click
* Vérifier que la tâche a été supprimer de la liste avec cette fois ci la condition « not.exist »

Le deuxième test sera « Ajoute une tâche et la modifie »

Comment réaliser ce test ?

* Sélectionner l’input avec le sélecteur placeholder= ‘Add a todo’
* Simuler l’envoi de la valeur « Todo à modifier » avec cy.type()
* Créer un événement de clic sur le bouton « Add » avec cy.click()
* Attendre une seconde Avec cy.wait(1000)
* Sélectionner et cliquer sur le bouton « Modifier »
* Sélectionner l’input avec le selecteur placeholder="Edit a todo"
* Effacer le contenu de l’input avec cy.clear()
* Saisir du texte avec cy.type(‘Todo modifié’)
* Sélectionner le bouton ‘Save’ et simuler le clic avec cy.click()
* Vérifié que la tâche a été modifiée avec la condition « be.visible »

Le troisième test sera « Supprime toutes les tâches existantes »

Comment réaliser ce test ?

* Sélectionner tous les éléments <li> qui se trouve dans la liste
* On itère sur chaque élément <li> trouvé dans l'étape précédente avec la méthode .each()
* On utilise cy.wrap pour effectuer une action qui sera de supprimer l’élèment
* On vérifie la suppression de tous les élèments

Le code

Il faut maintenant coder tout ça. Tout d’abords, on utilise la fonction describe() de Cypress pour définir une suite de tests appelée "Todo App". La fonction describe() prend deux arguments : une description de la suite de tests et une fonction de rappel (callback) contenant les tests associés.

À l'intérieur de cette suite de tests, on utilise également beforeEach() pour exécuter du code avant chaque test. Dans ce cas précis, la fonction beforeEach() visite l'URL "<http://localhost:3000>" à l'aide de cy.visit().

describe("Todo App", () => {

  beforeEach(() => {

    cy.visit("http://localhost:3000");

  });

Les trois tests seront encapsulés dans la fonction describe()

Voici un récupitulatif des fonctions utiles dans cypress :

 Sélection d'éléments :

* cy.get() : Sélectionne un élément DOM en fonction d'un sélecteur CSS, d'une classe, d'un ID, etc.
* cy.contains() : Sélectionne un élément DOM contenant un texte spécifique.
* cy.find() : Recherche les éléments enfants d'un élément sélectionné.

 Interaction avec les éléments :

* cy.click() : Clique sur un élément.
* cy.type() : Saisit du texte dans un champ de saisie.
* cy.clear() : Efface le contenu d'un champ de saisie.
* cy.select() : Sélectionne une option dans un élément <select>.

 Navigation :

* cy.visit() : Accède à une URL spécifique.
* cy.go() : Navigue en avant ou en arrière dans l'historique du navigateur.

 Assertions :

* cy.should() : Effectue des assertions sur un élément sélectionné.
* expect() : Effectue des assertions sur des valeurs ou des conditions spécifiques.

 Gestion des requêtes réseau :

* cy.request() : Effectue une requête HTTP personnalisée.
* cy.route() : Intercepte et modifie les requêtes réseau.

 Contrôle des tests :

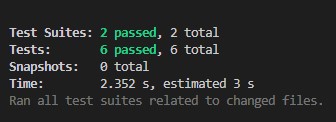
* cy.wait() : Attend un certain délai, un événement ou une condition avant de poursuivre les tests.
* cy.then() : Exécute une fonction personnalisée dans le contexte actuel des tests.

 Gestion des fichiers :

* cy.fixture() : Charge un fichier de données de test.
* cy.readFile() : Lit le contenu d'un fichier sur le disque

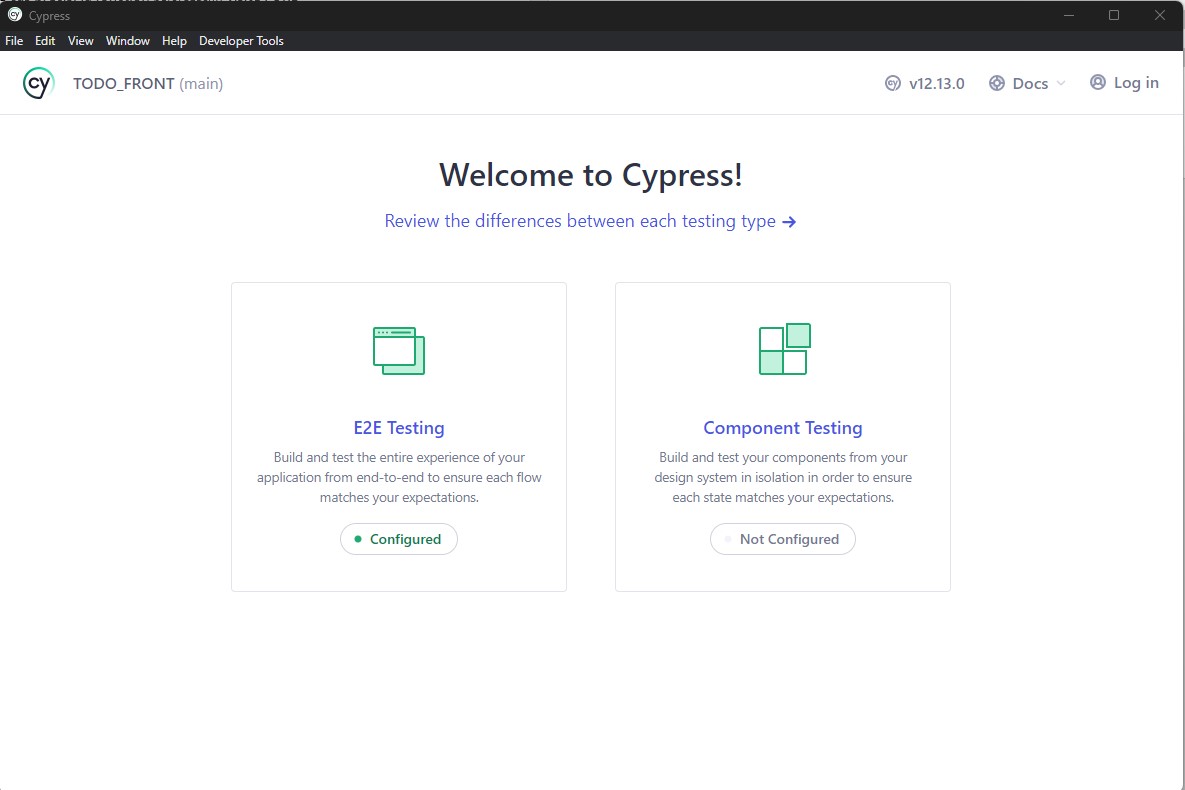
Résultats

Le Résultat des tests unitaires et intégration :

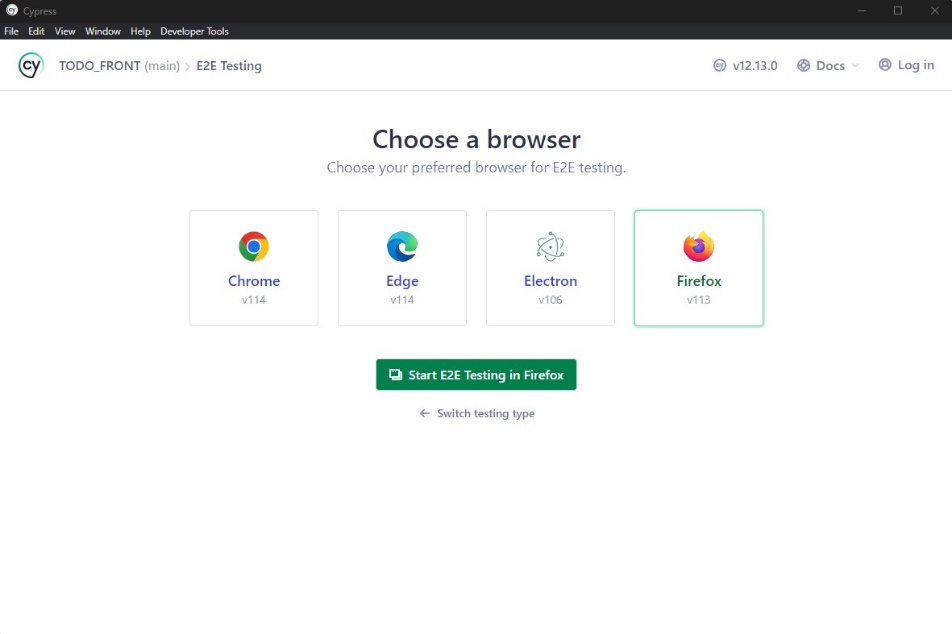


Le Résultat des tests end-to-end :

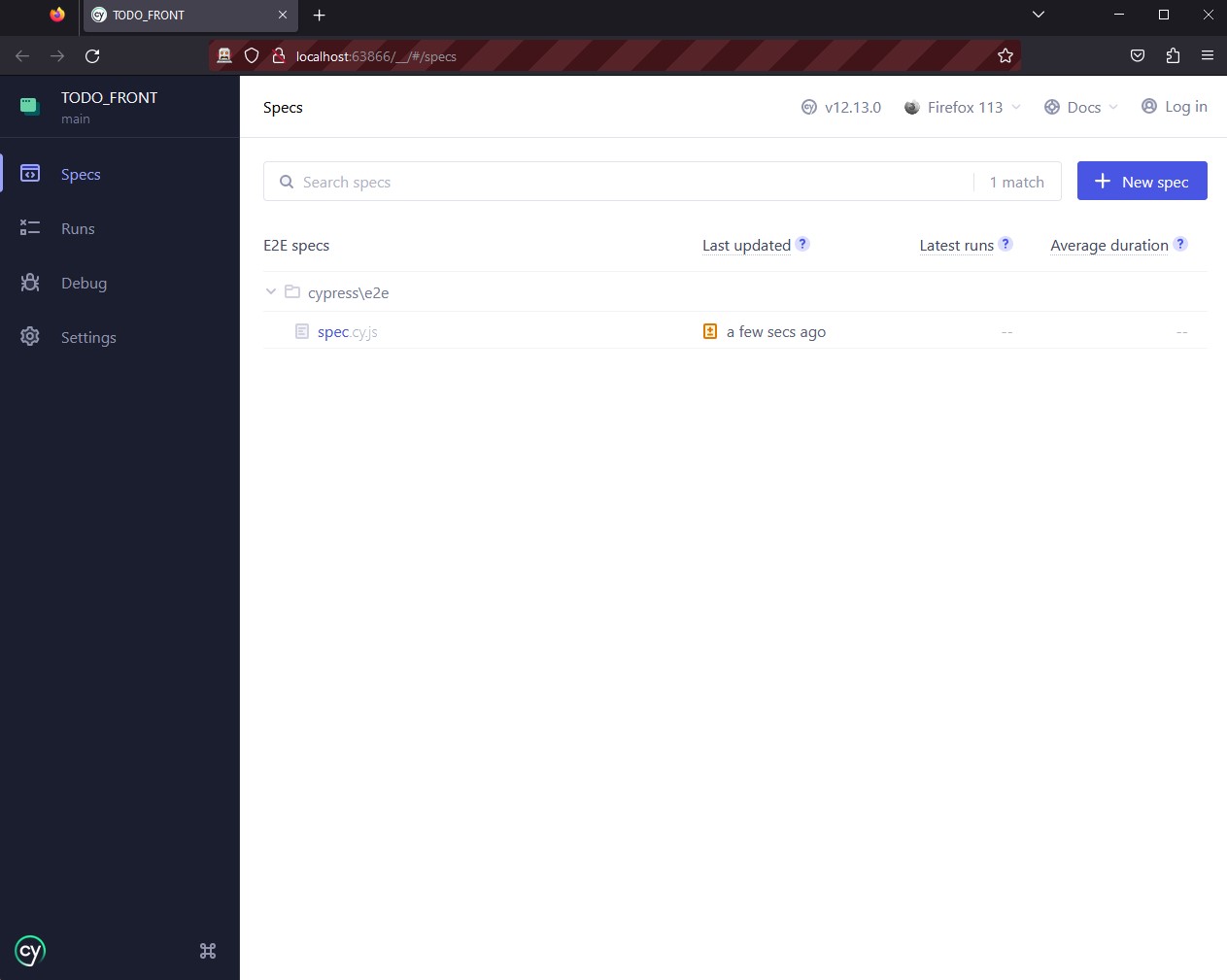
Tout d’abords, lancer le Framework CYPRESS avec npx Cypress open, on arrive sur cette page



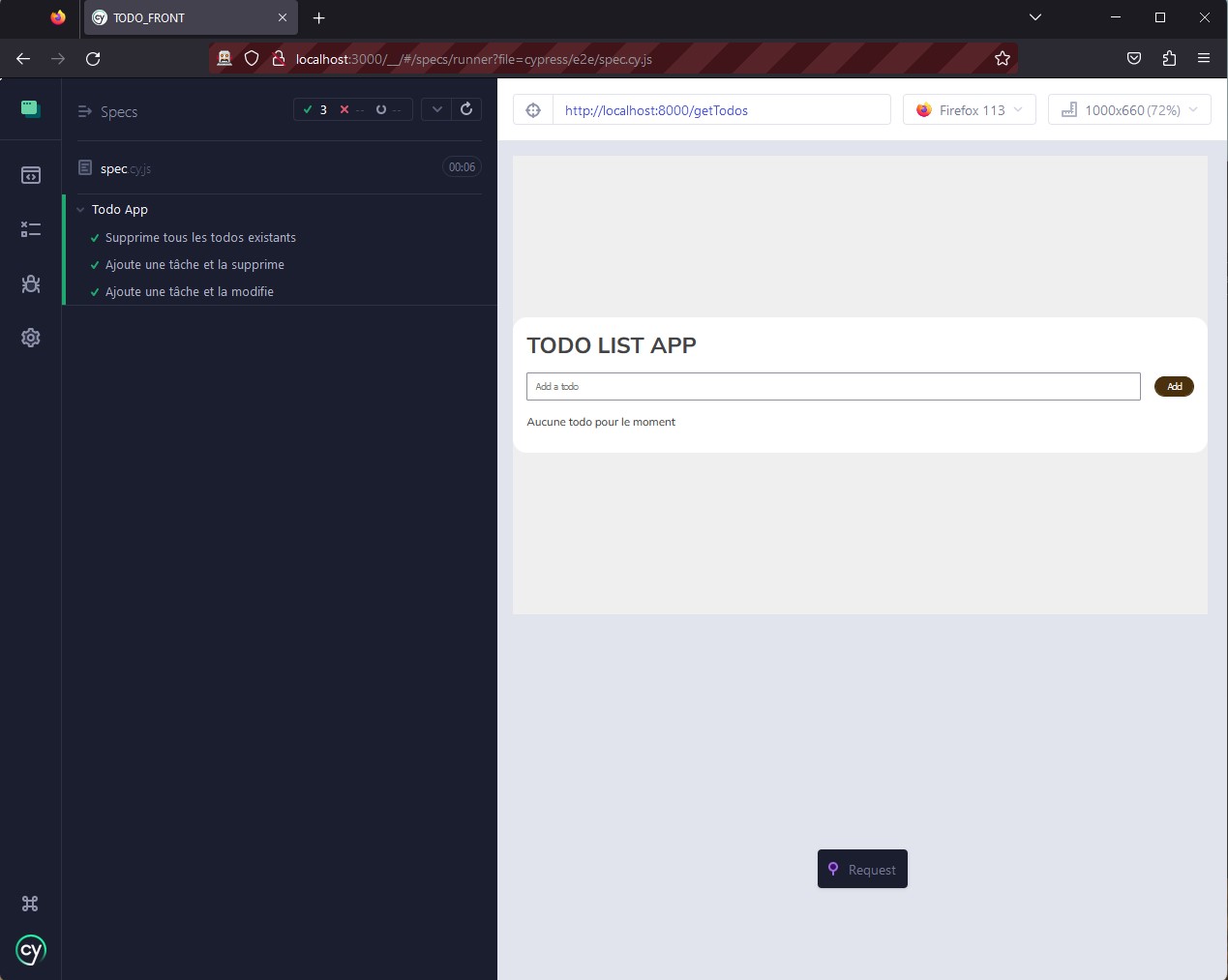
Selectionner E2E Testing



Selectionner votre navigateur préférée



Selectionner le fichier spec qui contient nos tests.



Les trois tests se sont lancés et ils ont tous réussit comme on peux observer sur la capture au dessus.

L’importance des tests pour les recruteurs

Les recruteurs demandent aux développeurs de savoir effectuer des tests car cela démontre leur compétence en matière de qualité du code, de maintenance, de collaboration en équipe et de respect des bonnes pratiques de développement. Les tests sont considérés comme une compétence essentielle pour garantir la production de logiciels de haute qualité.