Qunit

Typologie des tests

- Tests unitaires
  - Les tests unitaires consistent à tester individuellement les composants de l'application. On pourra ainsi valider la qualité du code et les performances d'un module.
- Tests d'intégration
  - Ces tests sont exécutés pour valider l'intégration des différents modules entre eux et dans leur environnement exploitation définitif. Ils permettront de mettre en évidence des problèmes d'interfaces entre différents programmes.
- Tests fonctionnels
  - Ces tests ont pour but de vérifier la conformité de l'application développée avec le cahier des charges initial. Ils sont donc basés sur les spécifications fonctionnelles et techniques.
- Tests de non-régression
  - Les tests de non-régression permettent de vérifier que des modifications n'ont pas altérées le fonctionnent de l'application. L'utilisation d'outils de tests, dans ce domaine, permet de facilité la mise en place de ce type de tests.

Typologie des tests

- Tests IHM
  - Les tests IHM ont pour but de vérifier que la charte graphique a été respectée tout au long du développement pour contrôler :
    - la présentation visuelle : les menus, les paramètres d'affichages, les propriétés des fenêtres, les barres d'icônes, la résolution des écrans, les effets de bord,…
    - la navigation : les moyens de navigations, les raccourcis, le résultat d'un déplacement dans un écran,…
- Tests de configuration
  - Une application doit pouvoir s'adapter au renouvellement de plus en plus fréquent des ordinateurs. Il s'avère donc indispensable d'étudier l'impact des environnements d'exploitation sur son fonctionnement.

Typologie des tests

- Tests de performance
  - Le but principal des tests de performance est de valider la capacité qu'ont les serveurs et les réseaux à supporter des charges d'accès importantes.
  - On doit notamment vérifier que les temps de réponse restent raisonnable lorsqu'un nombre important d'utilisateurs sont simultanément connectés à la base de données de l'application.
  - Pour cela, il faut d'abord relever les temps de réponse en utilisation normale, puis les comparer aux résultats obtenus dans des conditions extrêmes d' utilisation.
  - Une solution est de simuler un nombre important d'utilisateur en exécutant l'application à partir d'un même poste et en analysant le trafic généré.
  - Le deuxième objectif de ces tests est de valider le comportement de l'application, toujours dans des conditions extrêmes. Ces tests doivent permettent de définir un environnement matériel minimum pour que l'application fonctionnement correctement.
- Tests d’installation
  - Une fois l'application validée, il est nécessaire de contrôler les aspects liés à la documentation et à l'installation.
  - Les procédures d'installation doivent être testées intégralement car elles garantissent la fiabilité de l'application dans la phase de démarrage.
  - Bien sûr, il faudra aussi vérifier que les supports d'installation ne contiennent pas de virus.

Test end-to-end

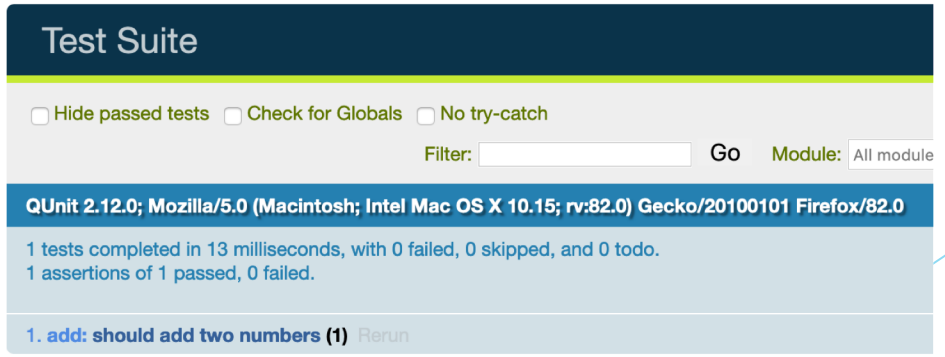
- Tester l’application comme dans le vrai monde avec un vrai utilisateur simulé.
- Ces tests lancent un navigateur, navigue dans l’application et interagit avec.
- Chonophage donc bien choisir les scénarios.

Exemple

- DL qunit.js

```
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width">
    <title>Test Suite</title>
    <link rel="stylesheet" href="https://code.jquery.com/qunit/qunit-2.12.0.css">
  </head>
  <body>
    <div id="qunit"></div>
    <div id="qunit-fixture"></div>
    <script src="https://code.jquery.com/qunit/qunit-2.12.0.js"></script>
</body>
</html>
```

```
  <script>
    const add = (a, b) => a + b;
    QUnit.module('add', function() {
      QUnit.test('should add two numbers', function(assert) {
        assert.equal(add(1, 1), 2, '1 + 1 = 2');
      });
    });
  </script>
```



Assertions

- assert.async()
  - Instruct QUnit to wait for an asynchronous operation.
- assert.deepEqual()
  - A deep recursive strict comparison.
- assert.equal()
- A non-strict comparison.
- assert.expect()
  - Specify how many assertions are expected to run within a test.
- assert.false()
  - A strict boolean false comparison.
- assert.notDeepEqual()
  - An inverted deep recursive comparison.
- assert.notEqual()
  - A non-strict comparison, checking for inequality.
- assert.notOk()
  - Check if the first argument is falsy.
- assert.notPropEqual()
  - A strict comparison of an object's own properties, checking for inequality.
- assert.notStrictEqual()
  - A strict comparison, checking for inequality.

- assert.notStrictEqual()
  - A strict comparison, checking for inequality.

- assert.ok()
  - Check if the first argument is truthy.
- assert.propEqual()
  - A strict type and value comparison of an object's own properties.
- assert.pushResult()
  - Report the result of a custom assertion.
- assert.rejects()
  - Test if the provided promise rejects.
- assert.step()
  - A marker for progress in a given test.
- assert.strictEqual()
  - A strict type and value comparison.
- assert.throws()
  - Test if a callback throws an exception.
- assert.timeout()
  - Set the length of time to wait for async operations before failing the test.
- assert.true()
  - A strict boolean true comparison.
- assert.verifySteps()
  - A helper assertion to verify the order and number of steps in a test.

Méthodes

- QUnit.module()
  - Group related tests under a common label.
- QUnit.start()
  - Start an async test suite.
- QUnit.test()
  - Add a test to run.
- QUnit.test.only()
  - Add a test that is exclusively run.
- QUnit.test.skip()
  - Add a test that will be skipped.
- QUnit.test.todo()
  - Add a test which expects at least one failing assertion.