Chapitre 10
Sujets divers

Plan

- Utilisabilité
- Tests unitaires
- Extras

Utilisabilité

Utilisabilité

"Degré selon lequel un produit peut être **utilisé**, par des **utilisateurs** identifiés, pour atteindre des **buts** définis avec **efficacité**, **efficience** et **satisfaction**, dans un **contexte** d'utilisation spécifié"

Utilisabilité

Pourquoi parler de l'**utilisabilité**?

==> Votre application web pourrait être la plus belle et la plus performante, si elle est **inutilisable**, elle ne sera jamais vraiment populaire…

Pensez à Slack…

Utilisabilité

- Utiliser les **supports** fournis par les appareils mobiles au maximum
  - GPS
  - Caméra
  - Micro
  - Gestuelles
  - Accéléromètres
  - Stockage interne ou cloud
- Connaissez votre **utilisateur type**

- **KISS** : Keep It Simple Stupid

Utilisabilité

- **Faciliter** la vie à l'utilisateur

  - Autocomplétion
  - Réduire la quantité de clics pour obtenir une information

- Respecter les **standards**

  - Cohérence et familiarité
  - Ex :
    - Erreurs - rouge
    - Avertissement - jaune
    - Succès - vert

Utilisabilité

- Donner du **feedback**
  - Barres de progression
  - Sablier/Loading animation
  - Messages d'erreurs / de succès clairs
    - Ne pas dévoiler de stacktraces à l'utilisateur
    - Unholy error
    - Localisation des messages

- Aider l'utilisateur à **se retrouver** dans l'application
  - Fil d'ariane (breadcrumb)

Utilisabilité

- UI Design patterns
  - Designing Web Interfaces - O'Reilly Media

- Dotez vous d'un *styleguide* et assurez vous de le respecter
  - Composants réutilisables et leur style
  - Patterns (Pourquoi une modale? Comment faire des tableaux?)

- Les mêmes patterns devraient être identiques partout dans votre application...

Utilisabilité

Quelques très bons *styleguides*

- Material Design
  https://material.io/guidelines/material-design/introduction.html

- iOS UI Guidelines
  https://developer.apple.com/ios/human-interface-guidelines/overview/design-principles/

- styleguides.io
  http://styleguides.io/

Tests

Tests

Il existe plusieurs méthodes et outils pour tester des applications web. Voyons les différents types de tests et quelques exemples d'outils reliés à ces méthodes.

- Unitaires
- Intégration
- Bout en bout (End-to-end / E2E)
- Snapshot testing

Tests unitaires

Un test unitaire implique de tester un composant/objet de l'application isolé de tout autres composants. On peut faire des tests unitaires du code front-end ou backend.

Les outils pour construire une suite de tests unitaires en javascript est composé de:

- Framework de test
  - Jasmine
  - Mocha
- Framework d'assertion
  - Jasmine inclus son module d'assertion
  - Chai
- Un test runner
  - Karma (permet d'exécuter les tests dans plusieurs navigateurs)
  - Mocha inclus son propre runner

Tests unitaires

Les tests unitaires permettent de prendre le contrôle sur un composant et de vérifier son comportement

- Entrées / Sorties
- Interactions (click, hover, focus, etc)
- Exceptions lancées / attrapées
- Appels de fonctions (paramètres d'entrée et de sortie)
- AJAX (manipuler le retour du serveur, erreurs, latence, etc)

Tests d'un composant Vue

```
// MyComponent.vue
<template>
 <span>{{ message }}</span>
</template>
<script>
 export default {
   data () {
     return {
       message: 'hello!'
     }
   },
   created () {
     this.message = 'bye!'
   }
 }
</script>
```

```
import Vue from 'vue'
import MyComponent from 'path/to/MyComponent.vue'
// Here are some Jasmine 2.0 tests, though you can
// use any test runner / assertion library combo you prefer
describe('MyComponent', () => {
       // Inspect the raw component options
       it('has a created hook', () => {
         expect(typeof MyComponent.created).toBe('function')
       })
       // Evaluate the results of functions in
       // the raw component options
       it('sets the correct default data', () => {
         expect(typeof MyComponent.data).toBe('function')
         const defaultData = MyComponent.data()
         expect(defaultData.message).toBe('hello!')
       })
       // Inspect the component instance on mount
       it('correctly sets the message when created', () => {
         const vm = new Vue(MyComponent).$mount()
         expect(vm.message).toBe('bye!')
       })
       // Mount an instance and inspect the render output
       it('renders the correct message', () => {
         const Ctor = Vue.extend(MyComponent)
         const vm = new Ctor().$mount()
         expect(vm.$el.textContent).toBe('bye!')
       })
})
```

Tests d’intégration

Un test d'intégration est un test qui a pour but de valider les interactions entre plusieurs composants d’une application.

Cette définition est très large car il peut s’agir de l’interaction entre deux (ou plusieurs) classes de votre code ou deux (ou plusieurs) services de votre application.

Un test d’intégration habituel est de tester la couche REST de l’application en envoyant différentes requêtes et en testant les différentes réponses.

Tests d'intégration

Il existe différents outils pour faire des tests d'intégrations.

- Rest Assured rest-assured.io (Java)
- Chakram dareid.github.io/chakram/ (JS)
- Certains opteront simplement pour un client REST et une librairie d'assertions

Tests end-to-end

Un test end-to-end (e2e) est un test qui a pour but de valider l'exécution du début jusqu'à la fin d'une fonctionnalité de l'application.

Ces tests passent donc par la couche visuelle (application web ou native), la couche réseau, la couche serveur (API) et même la base de donnée.

On utilise ces tests souvent pour valider qu'une fonctionnalité respecte les besoins et afin d'éviter toutes régression. Les tests peuvent même aller jusqu'à valider des captures d'écrans pour éviter toutes régressions visuelles.

Tests e2e

Il existe différents outils pour faire des tests end to end. La majorité des outils sont basé sur Selenium qui permet de prendre le contrôle d'un navigateur.

- Selenium
  - Nightwatch nightwatchjs.org
  - Webdriver.io webdriver.io
  - Cypress cypress.io
- Chrome Headless
  - github.com/GoogleChrome/puppeteer

Tests e2e

Étant donné que les tests end-to-end se base sur l'interaction visuelle avec l'application, ces tests peuvent s'avérer **fragiles** car changer un bouton ou un formulaire d'endroit peut faire planter votre suite de test. Il faut donc être vigilants sur la façon dont on construit ces tests.

Ex: éviter de valider une erreur par son texte, valider si la boîte d'erreur est présente dans le DOM (par son ID unique par exemple)

Test e2e

```
// For authoring Nightwatch tests, see
// http://nightwatchjs.org/guide#usage

/**
 * Test that user can login and see dashboard.
 */
module.exports = {
  'default e2e tests': function (browser) {
    // automatically uses dev Server port from /config.index.js
    // default: http://localhost:8080
    // see nightwatch.conf.js
    const devServer = browser.globals.devServerURL

    browser
      .url(devServer)
      .waitForElementVisible('#app', 5000)

      // Assert that user can see login.
      .assert.elementPresent('.login')
      .setValue('.js-login__username', 'demouser')
      .setValue('.js-login__password', 'testpass')
      .click('.js-login__submit')
      .pause(1000)

      // Assert that user can see dashboard.
      .assert.containsText('.ev-dashboard__heading h1', 'This is the dashboard')
      .pause(2000)
      .end()
  }
}
```

Source https://github.com/prograhammer/vue-example-project

Snapshot testing

Difficile de tester les changements purement **visuels** de l'application…

Le snapshot testing permet de faire des comparaisons **visuelles** de l'application et de déceler les différences potentielles.

https://jestjs.io/docs/en/snapshot-testing

https://vueschool.io/articles/vuejs-tutorials/snapshot-testing/

Comparaison des différentes méthodes de test

Comparaison

Unitaires

- Stabilité +++
- Contrôle +++

Étant donné qu'on "mock" les objets externes à l'objet testé, on peut contrôler au maximum le test.

- Qualité +

Un test unitaire qui passe garantit que le composant se comporte comme voulu mais ses interactions avec les autres ne sont pas garanties (c'est pourquoi on fait des tests d'intégration)

Intégration

- Stabilité +
- Contrôle +
- Qualité +++

E2E

- Stabilité ---

Les changements visuels ou de positionnement peuvent briser les tests. On veut éviter les tests trop précis qui rende le test "fragile"

- Contrôle -
- Qualité+++

Un test E2E valide l'entièreté de l'architecture, comprend le navigateur, le réseau, l'API, etc. Sa valeur sur la qualité est très grande.

Idéalement….

Idéalement on utilise un heureux mélange des trois techniques dépendamment des besoin de tests qu’on a besoin.

Qu'est-ce qu'on teste?

Client

- Sérialisation / Désérialisation
- Échappement
  - Ne jamais faire confiance au contenu retourné par le serveur
- Appels qui nécessitent une validation, les formulaires, les « inputs »
  - Tester les cas limites, XSS, Caractères spéciaux, etc.
- Appels Ajax qui pourraient retourner des messages d'erreur et qu'on voudrait notifier l'utilisateur
  - Idéalement *mocker* les appels Ajax afin spécifier le code d'erreur auquel on s'attend (Jasmine permet de *mocker* la fonction $.ajax…)
- Appels qui affichent une confirmation à l'utilisateur

Serveur

- Sérialisation / Désérialisation
- Échappement
  - S'assurer qu'il n'est pas possible d'ajouter du code dans la base de données ou dans l'application
- Validation des formulaires
  - Côté serveur et côté client
- Tests de toute la logique d'affaire
  - Chaque objet devrait être testé unitairement
  - Chaque interaction client/serveur devrait être testée avec un test d'intégration / fonctionnel.

Qu’est-ce qu’on teste?

Client

- Différents navigateurs IE11, Edge , Chrome, Firefox, Opera
  - Permet de trouver différentes erreurs CSS et JavaScript.
- Différents appareils / OS / Résolution d’ écran *(difficile à automatiser…)*
  - Windows, MacOS, iPhone, Android, Écran Rétina, etc.
- Applications utilisant des dates, « datepicker »
  - Tester les différents fuseaux horaires, DST (Daylight Saving Time)
- Les pertes de connexions
- Applications utilisant certains périphériques (Géolocalisation, Webcam, etc.)
  - Tester lorsque les périphériques ne sont pas disponibles

Qu’est-ce qu’on ne teste pas?

Client

- Le message d’erreur contient le texte « Une erreur est survenue »
  - On teste plutôt que la méthode showError() a bien été appelée lors d’une erreur 500

Serveur

- Voir le cours de qualité/métriques...

Extras

Préprocesseurs CSS

Vous avez probablement eu une certaine difficulté à maintenir une architecture CSS propre et bien découpée…

⇒ Manque d'héritage, de variables, de nesting, de *mixins*…

Solution : Transpiler un autre language en CSS

SASS : http://sass-lang.com/

LESS : http://lesscss.org/

autres…

Préprocesseurs CSS

Un petit aperçu de **SASS**...

```
// Fichier base.scss

$font-stack: Helvetica, sans-serif;
$primary-color: #333;


// Fichier body.scss

@import “base”
body {
  font: 100% $font-stack;
  color: $primary-color;
}
```

TypeScript

De la même manière, il est parfois difficile de s'y retrouver dans une architecture JavaScript, ou de s'éviter des problèmes de **typage…**

⇒ Serait-il possible de combiner la flexibilité du JavaScript avec le typage des langages plus traditionnels?

⇒ **Solution** : transpiler un autre langage en JavaScript

TypeScript : http://www.typescriptlang.org/

CoffeeScript : http://coffeescript.org/

Dart : https://www.dartlang.org/

(pas d'ActionScript svp…)

TypeScript

```
class Student {
    fullname : string;
    constructor(public firstname, public middleinitial, public lastname) {
        this.fullname = firstname + " " + middleinitial + " " + lastname;
    }
}
interface Person {
    firstname: string;
    lastname: string;
}
function greeter(person : Person) {
    return "Hello, " + person.firstname + " " + person.lastname;
}
var user = new Student("Jane", "M.", "User");
document.body.innerHTML = greeter(user);
```

TypeScript

Fonctionnalités expérimentales

Service Workers

https://developer.mozilla.org/fr/docs/Web/API/Service_Worker_API

Le *caching* d'une application web est un sujet particulièrement épineux.

Les **service workers** tentent de régler plusieurs problèmes:

- Caching des fichiers
- Faciliter les déploiements de l'application
- Mode *offline*

L'implémentation est plus difficile qu'elle en a l'air!

Web Assembly

https://webassembly.org/

Le JavaScript possède toujours ses limites, notamment au niveau de la performance vs des langages plus traditionnels.

WebAssembly permet d'exécuter du C++/Rust dans le navigateur! Technologie très prometteuse pour l'industrie du jeu vidéo notamment.

IndexedDB

https://developer.mozilla.org/fr/docs/Web/API/API_IndexedDB

Le *state* d'une application web peut devenir parfois très complexe - nécessitant plus qu'un simple local storage (queries, index etc.)

IndexedDB offre une *database* similaire à SQL directement dans le navigateur - fonctionne de pair avec les Service Workers.

À SUIVRE....