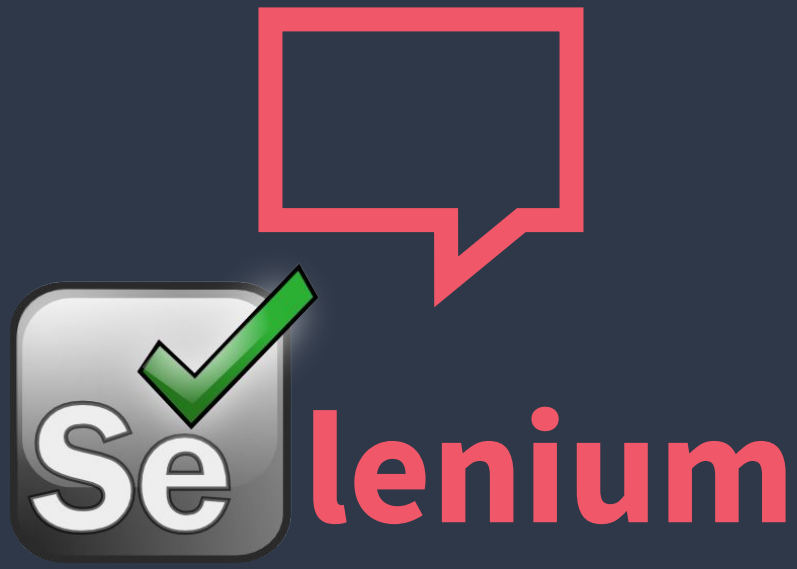

Selenium

SELENIUM

Web Browser Automation Tool

Qué es Selenium?

- Selenium automatiza navegadores. Es una API que nos permite interactuar con los componentes gráficos de aplicaciones web.
- Sirve para crear tests robustos y tests de regresión para aplicaciones web.
- Generalmente se utiliza con el propósito de testear aplicaciones web, pero no está limitado solamente a eso. Por ejemplo, se pueden automatizar tareas de administración.
- Esta herramienta nos da la facilidad de utilizar el mismo código para soportar distintos browsers en distintos sistemas operativos.
- Actualmente soporta los browsers más importantes como Firefox, Google Chrome, Internet Explorer, Safari y otros más incluyendo los de mobile web.

Selenium WebDriver

- Selenium contiene un servidor que abre un navegador y establece una comunicación con él.
- El servidor espera a que el cliente le envíe comandos para ser ejecutados y envía estos comandos al navegador.
- El WebDriver es un cliente de este servidor: permite enviar comandos desde un lenguaje de programación (Node.js por ejemplo)

Testing

UNIT TESTS

- Unit Tests es código que testea unidades de código. Pueden ser funciones, módulos o clases.

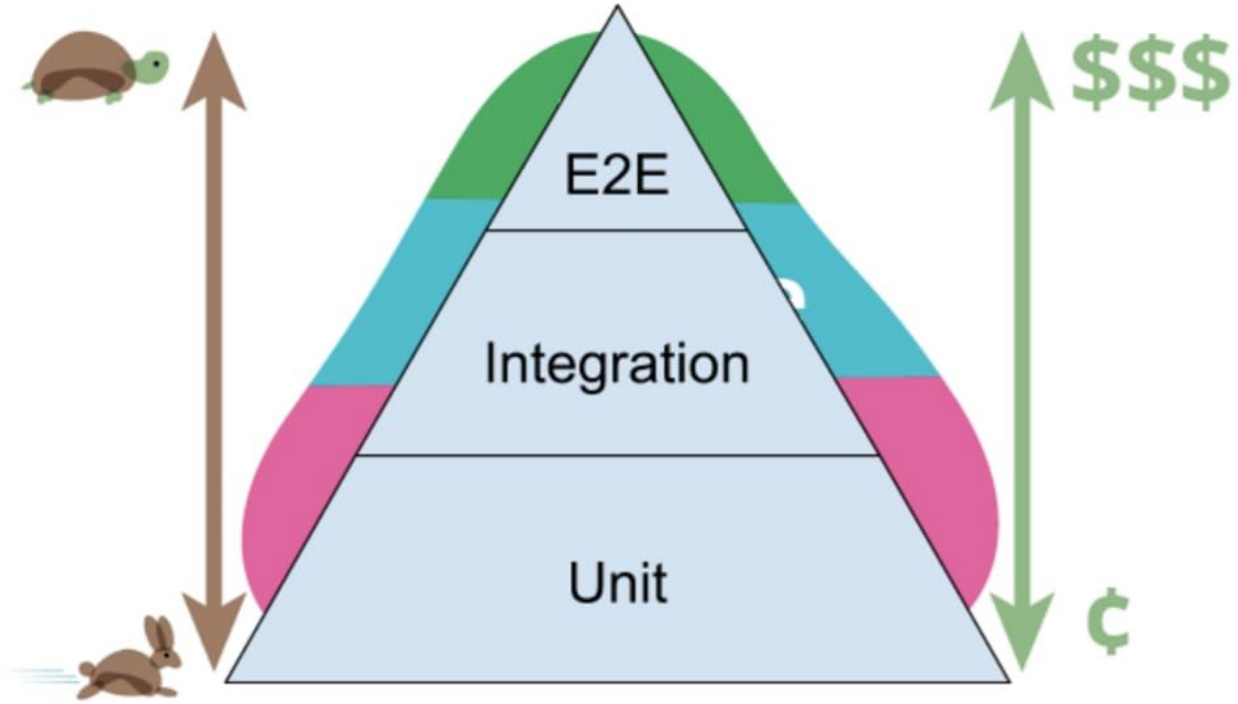
INTEGRATION TESTS

- Son tests que tienen el objetivo de chequear la integración de dos o más units (por ejemplo: módulos, clases, etc)

END TO END TESTING

- End to end (e2e) testing es el proceso de ejecutar un escenario de test contra un navegador real para testear todas las capas de la aplicación.

Testing



Alcance de la Herramienta

END TO END TESTING

- Selenium es una buena herramienta para crear tests end to end, ya que puede manejar cualquier navegador web utilizando una API estandarizada. Además, tiene soporte activo.

E2E: slow and flaky

- Este tipo de tests suelen ser muy **lentos**. No es factible hacer un único test que loguee al usuario y simule toda la feature. Todo test entonces, debe comenzar con un nuevo browser, y un nuevo login. Tener muchos E2E tests, como habría que tener para testear una página web, será sumamente lento y correr tests tiene que ser rápido.

- Además son **flaky**. Flaky tests son tests que a veces pasan y otras fallan. Cuanto más operaciones de I/O tiene el test, más flaky se vuelve. Los tests en Selenium también suelen estar muy ligados a la estructura HTML de la aplicación, por lo que un pequeño cambio puede producir fallas en tests.

Funcionalidades: Métodos del driver

Basados en la librería para Javascript
Funcionalidades similares se pueden encontrar para el resto de los lenguajes.
https://seleniumhq.github.io/selenium/docs/api/javascript/index.html

Navegación

- this.get(url) → Promise<undefined>
- this.close() → Promise<undefined>
- this.navigate() → Navigation
- this.quit() → Promise<undefined>

Manejo de Tabs

- this.getAllWindowHandles() → Promise<Array<string>>
- this.switchTo() → TargetLocator

Funcionalidades: Métodos del driver

Búsqueda de Elementos

- **this.findElement(locator) → WebElementPromise**
- **this.findElements(locator) → Promise<Array<WebElement>>**

Selectores o "Locators"

- By.className(name) → By
- By.css(selector) → By
- By.id(id) → By
- By.linkText(text) → By
- By.name(name) → By
- By.xpath(xpath) → By

Funcionalidades: Métodos del driver

Otros

- this.getCurrentUrl() → Promise<string>
- this.getTitle() → Promise<string>

Ejecución de Scripts

- this.execute(command) → Promise<(T|null)>
- this.executeScript(script, ...args) → IThenable<(T|null)>

- this.takeScreenshot() → Promise<string>

Funcionalidades: Métodos de WebElement

Permiten realizar aserciones sobre nuestros componentes.

- this.clear() → Promise<undefined>
- this.click() → Promise<undefined>
- this.getCssValue(cssStyleProperty) → Promise<string>
- this.getText() → Promise<string>
- this.isDisplayed() → Promise<boolean>
- this.isEnabled() → Promise<boolean>
- this.isSelected() → Promise<boolean>
- this.sendKeys(...args) → Promise<undefined>
- this.submit() → Promise<undefined>

Funcionalidades: Wait Until

- Durante la ejecución del test puede ser necesario esperar a que se muestren ciertos elementos antes de ejecutar una acción, de lo contrario el driver no lo encontrará y arrojará una excepción.
- Para esto utilizamos el siguiente método del driver:

this.wait(condition, timeout, message) → (IThenable<T>|WebElementPromise)

- Selenium nos provee maneras de definir condiciones para utilizar en el wait usando la clase **until.**
- **Ejemplo**

```
    var query =
driver.wait(until.elementLocated(By.name('q')));
```

Funcionalidades: Wait Until

Until Conditions

- alertIsPresent() → Condition
- elementIsDisabled(element) → Condition
- elementIsEnabled(element) → Condition
- elementIsNotSelected(element) → Condition
- elementIsNotVisible(element) → Condition
- elementIsSelected(element) → Condition
- elementIsVisible(element) → Condition
- elementLocated(locator) → Condition
- elementTextContains(element, substr) → Condition
- elementTextIs(element, text) → Condition
- elementTextMatches(element, regex) → Condition
- elementsLocated(locator) → Condition
- titleContains(substr) → Condition
- titleIs(title) → Condition
- titleMatches(regex) → Condition
- urlContains(substrUrl) → Condition
- urlIs(url) → Condition
- urlMatches(regex) → Condition

Asincronismo en Javascript con Selenium

- JavaScript es un lenguaje asincrónico. Llas operaciones de I/O suelen ser asincrónicas, debido a que el código JS corre en un único thread. Si el thread se bloquea, no se permite correr código en paralelo. Para eso, en lugar de bloquear el thread se proveen **callbacks** que se llamarán cuando la operación finaliza.

```
fs.readFile('source.txt', (_, content) => {
 fs.writeFile('target.txt', content, () => {
   console.log('done!')
 })
})
```

- Los callbacks pueden dar lugar a código difícil de seguir. Una alternativa es utilizar **promises**, que permiten escribir callbacks de una manera más legible usando *then*:

```
readFilePromise('source.txt')
 .then(content =>
writeFilePromise('target.txt', content))
 .then(() => console.log('done'))
```

Asincronismo en Javascript con Selenium

- Para evitar por completo el uso de callbacks, se agregó una manera de decirle a JavaScript que espere a que se resuelva la promise y continúe ejecutando, usando las directivas **async** y **await**:

```
async function main() {
 const content = await readFilePromise('source.txt')
 await writeFilePromise('target.txt', content)
 console.log('done')
}
main()
```

- **Todas las funciones de Selenium WebDriver son asincrónicas y devuelven promesas, por lo que podemos utilizar async y await para simular el flujo sincrónico en el test.**

Asincronismo en Javascript con Selenium

EJEMPLO CON CALLBACKS

```
const {Builder, By, Key, until} = require('selenium-webdriver');

let driverPromise = new Builder()
   .forBrowser('firefox')
   .build();

driverPromise.then(driver => driver.get('http://www.google.com/ncr'))
 .then(() => driver.findElement(By.name('q')))
 .then(element => element.sendKeys('webdriver', Key.RETURN))
 .then(() => driver.wait(until.titleIs('webdriver - Google Search'), 1000))
 .then(() => driver.quit())
```

Asincronismo en Javascript con Selenium

EJEMPLO SIN CALLBACKS

```
async function main() {
   let driver = await new Builder()
       .forBrowser('firefox')
       .build();
 await driver.get('http://www.google.com/ncr')

 const element = await driver.findElement(By.name('q'))
 await element.sendKeys('webdriver', Key.RETURN)
 await driver.wait(until.titleIs('webdriver - Google Search'), 1000)
 await driver.quit()
}
main()
```

Buenas Prácticas

PAGE OBJECT PATTERN

- Es común que al escribir tests e2e se empiece a repetir el código. Además, el testing e2e está muy ligado al código HTML, lo que ocasiona que al modificarlo se arruinen tests creados anteriormente. Esto podría generar mucho trabajo.

- La solución para este problema son los Page Objets. Estos centralizan la lógica de una página con el objetivo de realizar end to end testing.

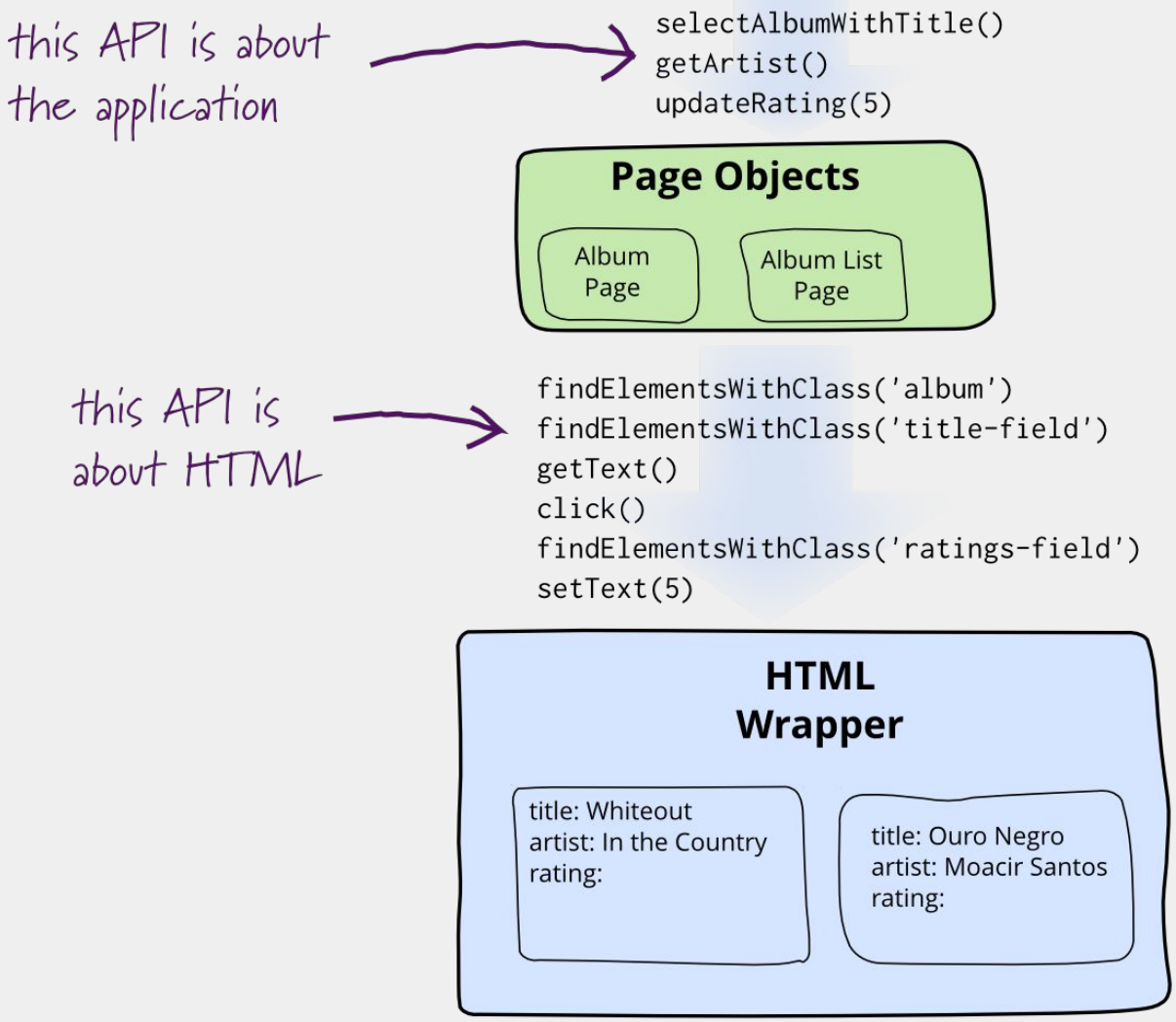
- Basta con crear un objeto JavaScript (no se necesitan librerías para construir un page object)

Buenas Prácticas

PAGE OBJECT PATTERN

- Encapsula una página HTML (o una parte) utilizando una API específica, permitiendo manipular elementos sin tener que escribir código específico.
- Debe permitir a un cliente realizar y observar todo lo que un ser humano puede hacer, a través de una interfaz que sea fácil de utilizar.
- No necesariamente tiene que existir un Page Object por página, sino uno por cada elemento significativo de la página.
- Este patrón se puede aplicar de igual forma a cualquier tecnología UI (no solamente a HTML)

Buenas Prácticas - Page Object Pattern



Buenas Prácticas

Sobre Page Object…

- No manipular objetos de Selenium directamente en el test, como por ejemplo WebElements. Siempre hacerlo a través del page object correspondiente.

Asserts

- Hay distintas opiniones sobre si las aserciones deben incluirse o no en los Page Objets.
  - Si se incluyen, se evitan aserciones duplicadas en los scripts de test. Además, facilita crear mensajes de errores y soporta un estilo de API: **TellDontAsk**.
  - Pero por otro lado, si se incluyen, se mezclan las responsabilidades de proveer acceso a información de la página y la lógica de testificación.

  En nuestro caso, decidimos no incluirlas.
- Incluir mensajes claros en los asserts.

Ejemplo de PageObject - LoginPage

```
const webdriver = require("selenium-webdriver");
const By = webdriver.By;
const until = webdriver.until;

module.exports = function(driver){
  const selectors = {
      nameinput: By.name("nick"),
      ageinput: By.name('age'),
      cityinput: By.name('city'),
      loginButton: By.name('Login-button'),
  };
  return {
      url: 'http://localhost:3000/login',

...
}
```

Ejemplo de PageObject - LoginPage

```
waitUntilVisible: async function () {
    await driver.wait(until.elementLocated(selectors.ageinput));
    await driver.wait(until.elementLocated(selectors.cityinput));
    await driver.wait(until.elementLocated(selectors.nameinput));
},

navigate: async function () {
    await driver.navigate().to(this.url);
    await this.waitUntilVisible();
},

login: async function (name, age, city) {
    await driver.findElement(selectors.nameinput).sendKeys(name);
    await driver.findElement(selectors.ageinput).sendKeys(age);
    await driver.findElement(selectors.cityinput).sendKeys(city);
    await driver.findElement(selectors.loginButton).click();
}
```

Ejemplo de PageObject - ChatsPage

```
const selectors = {
  userName: By.id('userName'),
  userAge: By.id('userAge'),
  userCity: By.id('userCity')
};
waitForProfileIsVisible: async function() {
  await driver.wait(until.elementLocated(selectors.userName), 10000);
  await driver.wait(until.elementLocated(selectors.userAge), 10000);
  await driver.wait(until.elementLocated(selectors.userCity), 10000);
},
getUserName: async function() {
  return await driver.findElement(selectors.userName).getText();
},
getUserAge: async function () {
  return await driver.findElement(selectors.userAge).getText();
},
getUserCity: async function() {
  return await driver.findElement(selectors.userCity).getText();
},
```

Ejemplo de PageObject - ChatsPage

```
openChat: async function(name) {
  await driver.findElement(By.css('[data-nick="' + name + '"]')).click();
},
sendMessage: async function(message) {
  await driver.findElement(By.id("message-input")).sendKeys(message);
  await
driver.findElement(By.id("message-input")).sendKeys(Key.RETURN);
},
getChatHeader: async function() {
  return await driver.findElement(By.id("chat-header")).getText();
},
checkLastSentMessage: async function(name, message) {
  return await checkLastMessage(By.className('sent'), name, message)
},
checkLastReceivedMessage: async function(name, message) {
  return await checkLastMessage(By.className('replies'), name,
message)
},
```

Ejemplo de PageObject - ChatsPage

```
async function checkLastMessage(selector, name, message) {
   const webElements = await driver.findElements(selector);
   if (webElements.length < 1) {
      return false;
   }
   const lastMessage = webElements[webElements.length - 1];
   const messageNick = await
lastMessage.getAttribute("data-message-nick");
   const messageData = await
lastMessage.getAttribute("data-message");
   return  messageNick === name &&  messageData === message;
}
```

Ejemplo de PageObject - Groups Page

```
clickOnNewGroup: async function() {
      await driver.findElement(By.id("newGroup")).click();
  },
  clickOnChats: async function() {
      await driver.findElement(By.id("chats")).click();
  },
  setGroupName: async function(groupName) {
      await driver.findElement(By.id("groupName-input")).sendKeys(groupName);
      await driver.findElement(By.id("groupName-input")).sendKeys(Key.RETURN);
  },
  selectGroupMember: async function(nick) {
      await driver.findElement(By.css('[data-nick="' + nick + '"]')).click();
  },
  createGroup: async function() {
      await driver.findElement(By.id("addNewGroup")).click();
  },
  openGroup: async function(groupName) {
      await driver.findElement(By.css('[data-group-id="' + groupName + '"]')).click();
  }
}
```

Buenas Prácticas

Screenshots

- Crear capturas de pantalla puede ayudar a ahorrar mucho tiempo cuando se investiga la causa de un test fallido.
- Método del WebDriver: this.takeScreenshot() → Promise<string>

Orden de Prioridad de Selectores

Cuando buscamos un elemento en la página, debemos tener en cuenta el siguiente orden de prioridad:

1. By.id(id)
2. By.name(name)
3. By.className(className)
4. By.tagName(name)
5. By.linkText(linkText)
6. By.partialLinkText(linkText)
7. By.xpath(xpathExpression)
8. By.cssSelector(selector)

Buenas Prácticas

Wait: Sleep, Implicit and Explicit

- ...Explicit wait:

En la espera explícita nosotros escribimos el código para que el proceso espere por un elemento

- ...Implicit wait

En la espera implícita le decimos al WebDriver que en cualquier búsqueda espere determinados segundos por el elemento y ya no hace falta que pongamos código para decir explícitamente que se debe esperar.

- No utilizar THREAD SLEEP ya que reduce los tiempos de ejecución de los tests.

Mocha + React + Selenium

- Mocha es un framework de JavaScript para testing que corre sobre Node.js, útil para testing asincrónico.
- https://mochajs.org/

ESTRUCTURA DE LOS TESTS CON MOCHA

```
describe("messages test", function () {
beforeEach(async () =>
    { driver = await new webdriver.Builder().forBrowser('chrome').build();
    loginPage = require('../test/pages/login')(driver);
    tabsSwitcher = require('../test/utils/tabsSwitcher')(driver);
    chatsPage = require('../test/pages/chats')(driver);
    await loginPage.navigate();
    await tabsSwitcher.init();});
it("Open private chat and send message", async() => {...};
after(function (done) {
    driver.quit().then(() => done()); });
```

Login Test

```
it("Log user", async() => {
   await loginPage.login(user.name, user.age, user.city);
   await chatsPage.waitForProfileIsVisible();
   const loggedUserName = await chatsPage.getUserName();
   const loggedUserAge = await chatsPage.getUserAge();
   const loggedUserCity = await chatsPage.getUserCity();

   assert(loggedUserName, user.name);
   assert(loggedUserAge, user.age);
   assert(loggedUserCity, user.city);
});
```

PrivateMessage Test

```
it("Open private chat and send message", async() => {
   // Logueo la primera persona
   var user1Tab = await tabsSwitcher.getTabIdentifier();
   await loginPage.login(user.name, user.age, user.city);

   // Logueo la segunda persona
   var user2Tab = await tabsSwitcher.openNewTab();
   await tabsSwitcher.switchTab(user2Tab);
   await loginPage.navigate();
   await loginPage.login(receiver.name, receiver.age, receiver.city);

   // El user1 va a enviar un mensaje al otro user
   await tabsSwitcher.switchTab(user1Tab);
   await chatsPage.openChat(receiver.name);
   await chatsPage.sendMessage(message);
```

PrivateMessage Test

```
var chatHeader = await chatsPage.getChatHeader();
  assert(chatHeader, receiver.name);

  var checkReceivedMessage = await
chatsPage.checkLastSentMessage(user.name, message);
  assert(checkReceivedMessage, true);

  await tabsSwitcher.switchTab(user2Tab);
  await chatsPage.openChat(user.name);

  chatHeader = await chatsPage.getChatHeader();
  assert(chatHeader, user.name);

  checkReceivedMessage = await
chatsPage.checkLastReceivedMessage(user.name, message);
  assert(checkReceivedMessage, true);
});
```

GroupCreation Test

```
it("Create group and send message", async() => {
   // Logueo la primer persona
   await loginpage.login(user1.name, user1.age, user1.city);
   await chatsPage.waitForProfileIsVisible();

   var user1Tab = await tabsSwitcher.getTabIdentifier();
   var user2Tab = await tabsSwitcher.openNewTab();
   await tabsSwitcher.switchTab(user2Tab);

   // Logueo la segunda persona
   await loginpage.navigate();
   await loginpage.login(user2.name, user2.age, user2.city);
   await chatsPage.waitForProfileIsVisible();

   // El user1 va a crear el nuevo grupo
   await tabsSwitcher.switchTab(user1Tab);
   await chatsPage.clickOnNewGroup();
   await chatsPage.setGroupName(groupName);
```

GroupCreation Test

```
 await chatsPage.selectGroupMember(user2.name);
   await chatsPage.createGroup();
   await chatsPage.openGroup(groupName);

   await chatsPage.sendMessage(message);
   var checkSentMessage = await
chatsPage.checkLastSentMessage(user1.name, message);
   assert(checkSentMessage, true);

   await tabsSwitcher.switchTab(user2Tab);
   await chatsPage.clickOnChats();
   await chatsPage.openGroup(groupName);
   checkSentMessage = await
chatsPage.checkLastReceivedMessage(user1.name, message);
   assert(checkSentMessage, true);
});
```

Alternativas a selenium-webdriver para JavaScript

- Selenium-webdriver suele ser considerado muy verboso. Existen otras librerías que encapsulan la interacción con el Selenium Server como por ejemplo:
  - Nightwatch.js
  - webdriver.io
  - Protractor (especializado para Angular)
  - Chimp.io
- Cada una de estas librerías agrega alguna "magia" a la API de webdriver, lo que puede tener desventajas al momento de debugging.

- Existen librerías que automatizan la creación de código de tests con Selenium, por ejemplo:
  - http://seleniumbuilder.github.io/se-builder
  - Esta app "graba" las acciones que se ejecutan en un navegador y las reproduce en tests, exportable a otros browsers

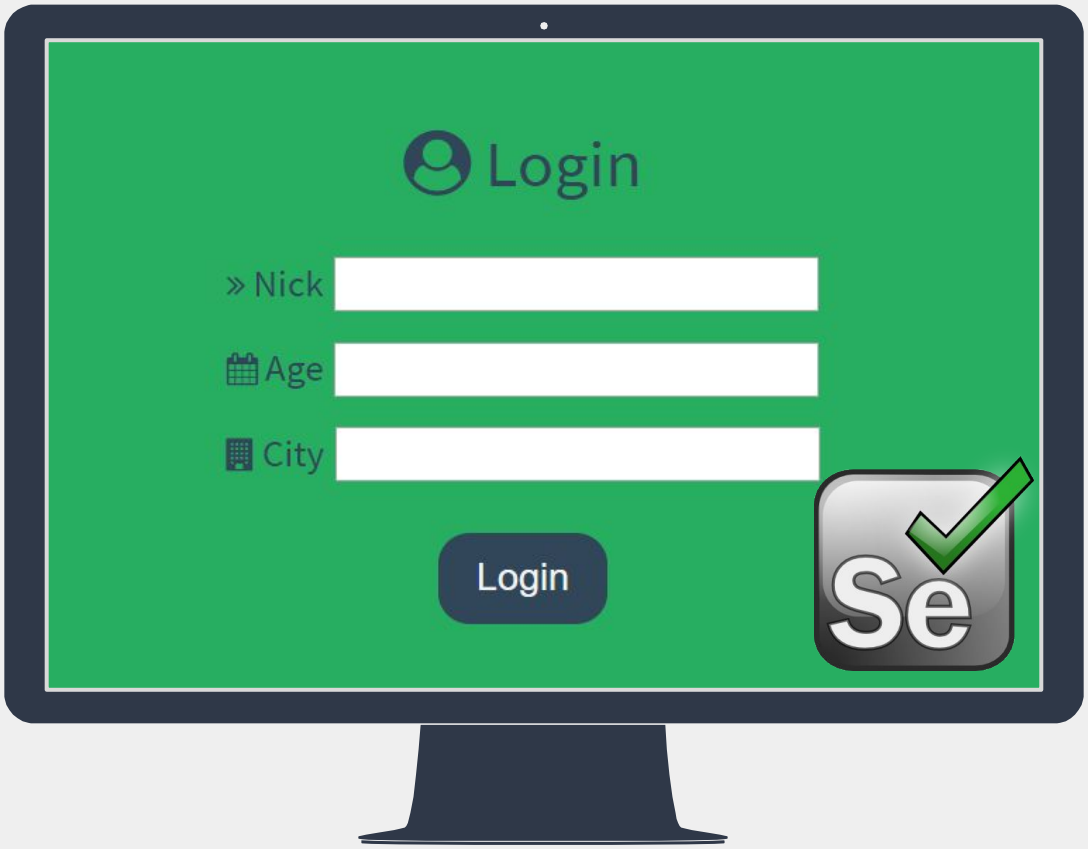
Selenium: Instalación

1. Instalar selenium-webdriver usando npm
   - **npm install selenium-webdriver**
2. Descargar e instalar los drivers correspondientes a los browsers en los que se va a correr los tests. Se pueden encontrar en la página de selenium-webdriver.
   - Por ejemplo, en nuestro caso utilizamos chromedriver.
3. Instalar un Selenium Server local, o utilizar uno remoto.
   - Selenium Server funciona como un proxy entre los tests y los drivers específicos para cada browser.
   - Para utilizarlo localmente, es necesario instalar el JDK y descargar la última versión desde Selenium. Luego, el server se corre con el comando:
   - **java -jar selenium-server-standalone-2.45.0.jar**
   - Para configurar uno remoto se puede usar:

```
let driver = new webdriver.Builder()
    .forBrowser('firefox')
    .usingServer('http://localhost:4444/wd/hub')
    .build();
```

Doc: **https://www.npmjs.com/package/selenium-webdriver**

Link útil: **https://gist.github.com/ziadoz/3e8ab7e944d02fe872c3454d17af31a5**



demo

Gracias!

Preguntas?

Maddonni, Axel
Martinez, Manuela
Rabinowicz, Lucia