🌐 E2E Pruebas externas de principio a fin

Pruebas de aplicaciones web de caja negra. Puppeteer para comprobación de existencia, navegación, tamaño, velocidad y otras métricas.

> *"Si un usuario final percibe una mal rendimiento en tu website, su siguiente click probablemente sea en tu-competencia.com"*
>
> -- ✍ **Ian Molyneaux**

Lo primero es **software que funcione**. Y cuando hablamos de una *web* la primera garantía debe ser la existencia y la descarga rápida de contenido relevante.

Todo lo demás vendrá después. Paro si el usuario no encuentra rápido lo que busca... entonces si puede se irá, y si no puede irse estará a disgusto. Ninguna de las dos situaciones es deseable.

Pero, afortunadamente, es muy sencillo realizar unos primeros test que garanticen estos aspectos tan básicos de nuestros sistemas.

La herramienta que os propongo utilizar es Puppeteer. Aunque todo se puede realizar con otras como Cypress o Playwright. La primera la usaremos para pruebas más funcionales, y la segunda es muy reciente y habrá que darle algo de tiempo...

Lo interesante no es el software que usemos, sino **tener claro cual es el objetivo de la prueba**. Que en este caso es tremendamente simple.

- ✅ Garantizar que una ruta existe.
- ✅ Comprobar que el contenido es el esperado.
- ✅ Validar que sus métricas entran en un rango esperado.

Son casi pruebas de humo para comprobar que algo no arde nada más enchufarlo. Pero por **su simplicidad para implantarlas y su coste tan asequible de ejecución** son la primera recomendación que puedo dar a un *web tester*.

Vamos a usar *Puppeteer* para realizar una pruebas muy básicas. Pero vamos a **prepararlo y automatizarlo** de forma que sea muy sencillo y barato realizarlas con frecuencia.

Para seguir este tutorial no se requiere ningún conocimiento previo sobre testing. Pero se recomienda familiarizarse con la terminología leyendo el artículo Filosofía y patrones

Puppeteer

Lo primero es tener las herramientas adecuadas. Instalamos Puppeteer con el comando `yarn add puppeteer`. Y empezamos a programar. Os propongo desarrollar un script Node que tome como dependencia a Puppeteer, lo lance y lo cierre.

```
const { getBrowser, closeBrowser, takeScreenshot } = require(`./lib/puppets`);
async function test() {
  const { browser, pagePuppet } = await arrangeBefore();
  await cleanAfter(browser);
}
async function arrangeBefore() {
  const browser = await getBrowser();
  const pagePuppet = await browser.newPage();
  return { browser, pagePuppet };
}
async function cleanAfter(browser) {
  await closeBrowser(browser);
}
```

Veamos en detalle:

El browser

La sección de preparación de cualquier test debe dejarlo listo para la ejecución de pruebas. Habitualmente **se preparan objetos** de negocio, ficheros, servicios o como en este caso se configura *Puppeteer* para visitar páginas en modo oculto y a la resolución que determinemos.

```
exports.getBrowser = async function getBrowser() {
  if (!browser) {
    browser = await puppeteer.launch({
      headless: false,
      defaultViewport: { width: 1920, height: 1080 },
      devtools: false
    });
  }
  return browser;
};
```

Y ahora aun par de pruebas.

La página

A partir de este momento ya empezamos con **lo que debería ocurrir**. Y lo más sencillo es determinar que una página existe y devuelve un código http válido. Acostúmbrate a la sintaxis asíncrona con `async-await` porque todo esto se ejecutará siempre en segundo plano.

```
module.exports = async function (pagePuppet) {
  const inputPageUrl = `https://www.bitademy.com/`;
  await given(`A the url ${inputPageUrl}`, async () => {
    await when(`we visit it`, async () => {
      const response = await pagePuppet.goto(inputPageUrl, { waitUntil: `load` });
      let actual = response.ok();
      let expected = true;
      then(`respond with an ok status code`, actual, expected);
    });
  });
};
```

Fíjate en la auditoria tan explícita que se hace. En un test siempre querrás saber lo que está pasando. Recuerda que es una herramienta para el desarrollador, es decir, para ti. Así que haz que te resulte cómoda, agradable y util.

Para homogenizar los mensajes te propongo que uses la terminología que tomo prestada del BDD. Se basa en usar los tres sucesos de toda prueba. *Given, when, then*. Es decir **dada** una situación de partida, **cuando** el usuario realiza una acción, **entonces** el sistema debería responder adecuadamente.

El contenido

Este caso de comprobar **existencia** es habitual, aunque mucho más habitual será comprobar **contenido**. Por ejemplo si la página existe pero queremos comprobar que es la adecuada. Para ello se usan métodos auxiliares para obtener acceso a la respuesta.

```
module.exports = async function (pagePuppet) {
  const inputPageUrl = `https://www.bitademy.com/`;
  await given(`A the page at ${inputPageUrl}`, async () => {
    await when(`we get its title`, async () => {
      await pagePuppet.goto(inputPageUrl, { waitUntil: `load` });
      const actual = await pagePuppet.title();
      const expected = `bitAdemy`;
      then(`it is ${expected}`, actual, expected);
    });
    await when(`we download all the content`, async () => {
      await pagePuppet.goto(inputPageUrl, { waitUntil: `networkidle2` });
      const content = await pagePuppet.content();
      const kiloByte = 1024;
      const maximumKiloBytes = 30;
      const maximunExpected = kiloByte * maximumKiloBytes;
      const actual = content.length < maximunExpected;
      const expected = true;
      then(`it is smaller than ${maximunExpected} bytes`, actual, expected);
    });
  });
};
```

En este caso comprobando título y tamaño de la descarga. Ojo a las esperas. Porque en algunas situaciones nos basta esperar a la respuesta básica del server y en otros necesitamos esperar a todo el contenido.

🎭 Pruebas de aplicaciones web con Puppeteer

Puppeteer para emulación, evaluación del contenido y seguimiento de la prueba.

> *"Hacer las pruebas de existencia y rendimiento al terminar el desarrollo o tras las pruebas funcionales es como tomar el pulso o hacer una analítica a un paciente que ya está muerto."*
>
> -- ✍ **Scott Barber**

Normalmente vas a querer comprobar algo más que la existencia básica de una web. Hay tantas posibles pruebas que hacer como situaciones pueda vivir un usuario. Pero para empezar te voy a mostrar las más utilizadas.

Emulación

Algo típico es que despliegues una aplicación web, pero quieras comprobar que se ejecuta correctamente en distintos dispositivos. Ya que Puppeteer usa por debajo un Chrome, podemos usar las capacidades de emulación que tiene.

```
module.exports = async function (pagePuppet) {
  await given(`Any page of my site`, async () => {
    const inputPageUrl = `https://www.bitademy.com`;
    const inputUserAgent =
      'Mozilla/5.0 (iPhone; CPU iPhone OS 11_0 like Mac OS X) AppleWebKit/604.1.38
(KHTML, like Gecko) Version/11.0 Mobile/15A372 Safari/604.1';
    await pagePuppet.setUserAgent(inputUserAgent);
    await pagePuppet.setViewport({ width: 375, height: 812 });
    await when(`we visit emulating an iPhone`, async () => {
      await pagePuppet.goto(inputPageUrl, { waitUntil: `load` });
      const actual = await pagePuppet.evaluate(() => {
        const nav = document.getElementById(`main-navigation`);
        const style = window.getComputedStyle(nav, ``);
        return style.getPropertyValue(`visibility`);
      });
      const expected = `hidden`;
      then(`it hides main-navigation`, actual, expected);
    });
  });
};
```

De esta forma puedes determinar si se aplican o no ciertos estilos, la validez de tus *media queries*...

Evaluación

Hablando de evaluar; te habrás fijado en la función `evaluate` de primer orden que admite un *callback* para ejecutar. La clave está en entender cuándo esa función se va a ejecutar. `Evaluate` ejecutará el *callback* de

forma asíncrona una vez descargada la página. Por ejemplo lo uso para comprobar que no tenemos links vacíos en una web.

```
module.exports = async function (pagePuppet) {
  await given(`A site url`, async () => {
    const inputPageUrl = `https://www.bitademy.com`;
    await when(`we scrap it for empty links`, async () => {
      await pagePuppet.goto(inputPageUrl, { waitUntil: `load` });
      const actual = await pagePuppet.evaluate(() =>
        window.find(`a:is(:not([href]),[href=""],[href="#"])`)
      );
      const expected = false;
      then(`have no one`, actual, expected);
    });
  });
};
```

> Cualquier expresión JavaScript que pongamos se ejecutará como si la hubiéramos escrito en la consola del Chrome.

Seguimiento

Además de lanzar el script y ver directo lo que ocurre, también puedes querer ver lo que pasó una vez terminado. Es decir comprobar qué ha ocurrido y tener un rastro; sobre todo si se ejecuta en modo desatendido o sin visualización.

```
exports.takeScreenshot = async function takeScreenshot(pagePuppet) {
  const timeStamp = new Date().getTime();
  const shotPath = path.join(process.cwd(), 'images', `${timeStamp}.png`);
  await pagePuppet.screenshot({
    path: shotPath,
    fullPage: false
  });
};
```

La función `screenshot` saca instantáneas de las pantallas que permitirán analizar tranquilamente lo que ocurrió. Recuerda que la prueba la haces para ti. El *log* o rastro de lo sucedido es tu principal activo al terminar la prueba.

Pruebas de rendimiento web con Lighthouse

Lighthouse para comprobación tamaño, velocidad, SEO y otras métricas.

> "*Cualquier optimización que no sea sobre el cuello de botella es una ilusión de mejora.*"
>
> -- ✍ **Federico Toledo**

Es decir, que antes de optimizar hay que medir y saber qué es lo que falla. Para medir de forma automática tras cada deploy, o cuando se establezca, te propongo que uses Lighthouse.

Seguro que conoces *lighthouse* como un complemento de *Chrome*. Pero aquí te voy a enseñar como integrarlo con *Puppeteer* para realizar una pruebas de rendimiento. Pero vamos a **prepararlo y automatizarlo** de forma que sea muy sencillo y barato realizarlas con frecuencia.

Para seguir este tutorial no se requiere ningún conocimiento previo sobre testing. Pero se recomienda familiarizarse con la terminología leyendo el artículo Filosofía y patrones

Lighthouse

Este producto de Google lo puedes usar de distintas formas manualmente. Para automatizarlo vamos a desarrollar un script Node que tome como dependencia a su librería desde npm

Seguiremos las misma premisas que empleamos con Puppeteer para las pruebas básicas. Es decir la famosa *triple A* **Arrange-Act-Assert**. Incluso con una última *cuarta A* **After** para organizar el código. Y el *Given, when, then* para informar al usuario del test, es decir al programador, copiado del *behavior-driven development*.

Arrange

Esta fase es un poco tediosa, pero te puede valer para todos los tests que hagas con *Lighthouse*. En esencia lanza una instancia de *Chrome* y se conecta a ella. Es en esa instancia en la que se cargará tu página usando *Puppeteer* y contra la que se lanzarán las peticiones de métricas.

```
async function arrangeBrowser() {
  const chrome = await launchChrome();
  console.info(`chrome.port : ${chrome.port}`);
  const browser = await connectToChrome(chrome);
  return { chrome, browser };
}

async function launchChrome() {
  const chrome = await chromeLauncher.launch(config);
  config.port = chrome.port;
  console.log(`Chrome launched at port: ${chrome.port}`);
  return chrome;
}
async function connectToChrome(chrome) {
  const resp = await util.promisify(request)
```

```
(`http://localhost:${chrome.port}/json/version`);
  const { webSocketDebuggerUrl } = JSON.parse(resp.body);
  const browser = await puppeteer.connect({ browserWSEndpoint:
webSocketDebuggerUrl });
  console.log(`Browser connected at url: ${browser._connection._url}`);
  return browser;
}
```

Act

Esta fase es más simple, aunque aquí si que tendrás que trabajar cada test. *Lighthouse* es capaz de tomar cientos de métricas y generar informes *json* o *html*. Eso está muy bien, pero exige que algo o alguien los procese después.

Mi propuesta minimalista y enfocada al paso de la prueba es que pidas una métrica concreta y la valides contra un resultado esperado.

```
module.exports = async function () {
  await given(`A deployed site`, async () => {
    const inputPageUrl = `https://www.bitademy.com`;
    const { chrome, browser, chrome_config } = await arrangeBrowser();
    await when(`we get the page audit scores`, async () => {
      const audits = await getAudits(inputPageUrl, chrome_config);
    });
  });
};
// Ver código real en el laboratorio
exports.getAudits = async function getAudits(url, chrome_config) {
  lh_desktop_config.settings.skipAudits = null;
  lh_desktop_config.settings.onlyAudits = [
    'first-meaningful-paint',
    'speed-index',
    'first-cpu-idle',
    'interactive'
  ];
  const lh_audits = await lighthouse(url, chrome_config, lh_desktop_config).then(
    results => results.lhr.audits
  );
  return mapToSimpleArray(lh_audits);
};
```

Vale, pedir tan pocas métricas quizá sea poco realista. Pero recuerda dos cosas antes de pedir a lo loco. **Las pruebas deben ser rápidas**. Y deben usarse con un objetivo concreto, **detectar y un cuello de botella y corregirlo**.

Assert

Esta es la parte más sencilla. Determina el umbral de rendimiento aceptable y compáralo con el resultado obtenido. Por ejemplo yo aquí estoy midiendo el *speed-index* , que es el criterio principal, y lo comparo contra el umbral que recomiendan en google.

```
const minimumExpected = 0.89;
const expected = true;
const score = audits.find(a => a.id === 'speed-index').score;
const actual = score > minimumExpected;
then(`Speed Index faster than ${minimumExpected}`, actual, expected);
```

After

Al acabar tus pruebas deberías liberar los recursos, que3 en este caso es simplemente desconectar y cerrar la instancia de *chrome*

```
browser.disconnect();
await chrome.kill();
```

> En el laboratorio tienes más ejemplos de lo que es capaz *Lighthouse*. Y si aún quieres más puede mirar este otro repositorio aún más completo AtomicBuilders/muon

Con esto tienes para empezar en el mundo de las pruebas de las aplicaciones web, ojo que **empezamos por lo más fácil de probar**. Pero algo salimos ganando, al menos comprobar que las páginas respondan rápido. Ya veremos cómo garantizar que además lo hagan correctamente.

 Pruebas de un API Rest

Pruebas de comunicaciones JSON

> *"Nunca hay pruebas suficientes que demuestren que un software está bien, pero un único test puede mostrar que está mal"*
>
> -- ✍ **Amir Ghahrai**

Hemos visto que *Puppeteer* es un automatizador de Chrome. Los navegadores solemos usarlo para ver páginas web pero también podemos, y como programadores debemos, usarlos para inspeccionar las comunicaciones de datos *json*. Con esto vas a poder probar tu API.

Supertest

Para facilitar dichas inspecciones usaremos una librería con un nombre pretencioso: `supertest`. En esencia nos permite hacer peticiones *http*.

Veamos la llamada get más sencilla posible. el *hello world* de las API.

```
async function getHello() {
  const inputHostUrl = `https://api-base.herokuapp.com`;
  await given(`the API url ${inputHostUrl}`, async () => {
    const inputEndPoint = `/api/pub/hello`;
    await when(`we call the ${inputEndPoint} endPoint`, async () => {
      const response = await request(inputHostUrl).get(inputEndPoint);
      const actual = response.body.message;
      const expected = `Hola Mundo`;
      then(`respond with the Hola Mundo message`, actual, expected);
    });
  });
}
```

CRUD

Algo más elaborado sería probar la comunicación completa. Dejo la muestra de cómo enviar una *payload* con el método *post*

```
async function postProject() {
  const inputHostUrl = `https://api-base.herokuapp.com`;
  await given(`the API url ${inputHostUrl}`, async () => {
    const inputEndPoint = `/api/pub/projects`;
    await when(`we post to the ${inputEndPoint} endPoint`, async () => {
      const inputProject = { name: 'start testing', dueDate: '2020-12-31' };
      const response = await
request(inputHostUrl).post(inputEndPoint).send(inputProject);
      const actual = response.body.name;
```

```
        const expected = 'start testing';
        then(`respond with the same object`, actual, expected);
        const expectedStatus = 201;
        then(`respond with status code 201`, response.status, expectedStatus);
      });
    });
  }
```

Como ves, se pueden comprobar respuestas, códigos, cabeceras... Es decir, se trata de automatizar las llamadas al API y comprobar que las respuestas entran dentro de lo esperado. Todo ello sin frameworks especiales de pruebas, sólo JavaScript con alguna librería de ayuda.