Automated Testing

Test automation is an efficient way of validating that your application code works as intended. While Electron doesn't actively maintain its own testing solution, this guide will go over a couple ways you can run end-to-end automated tests on your Electron app.

Using the WebDriver interface

From ChromeDriver - WebDriver for Chrome:

> *WebDriver is an open source tool for automated testing of web apps across many browsers. It provides capabilities for navigating to web pages, user input, JavaScript execution, and more. ChromeDriver is a standalone server which implements WebDriver's wire protocol for Chromium. It is being developed by members of the Chromium and WebDriver teams.*

There are a few ways that you can set up testing using WebDriver.

With WebdriverIO

WebdriverIO (WDIO) is a test automation framework that provides a Node.js package for testing with WebDriver. Its ecosystem also includes various plugins (e.g. reporter and services) that can help you put together your test setup.

Install the testrunner

First you need to run the WebdriverIO starter toolkit in your project root directory:

```
npx wdio . --yes
```

This installs all necessary packages for you and generates a `wdio.conf.js` configuration file.

Connect WDIO to your Electron app

Update the capabilities in your configuration file to point to your Electron app binary:

```
export.config = {
  // ...
  capabilities: [{
    browserName: 'chrome',
    'goog:chromeOptions': {
      binary: '/path/to/your/electron/binary', // Path to your Electron binary.
      args: [/* cli arguments */] // Optional, perhaps 'app=' + /path/to/your/app/
    }
  }]
  // ...
}
```

Run your tests

To run your tests:

```
$ npx wdio run wdio.conf.js
```

With Selenium

Selenium is a web automation framework that exposes bindings to WebDriver APIs in many languages. Their Node.js bindings are available under the `selenium-webdriver` package on NPM.

Run a ChromeDriver server

In order to use Selenium with Electron, you need to download the `electron-chromedriver` binary, and run it:

```
npm install --save-dev electron-chromedriver
./node_modules/.bin/chromedriver
Starting ChromeDriver (v2.10.291558) on port 9515
Only local connections are allowed.
```

Remember the port number `9515`, which will be used later.

Connect Selenium to ChromeDriver

Next, install Selenium into your project:

```
npm install --save-dev selenium-webdriver
```

Usage of `selenium-webdriver` with Electron is the same as with normal websites, except that you have to manually specify how to connect ChromeDriver and where to find the binary of your Electron app:

```
const webdriver = require('selenium-webdriver')
const driver = new webdriver.Builder()
  // The "9515" is the port opened by ChromeDriver.
  .usingServer('http://localhost:9515')
  .withCapabilities({
    'goog:chromeOptions': {
      // Here is the path to your Electron binary.
      binary: '/Path-to-Your-App.app/Contents/MacOS/Electron'
    }
  })
  .forBrowser('chrome') // note: use .forBrowser('electron') for selenium-webdriver
<= 3.6.0
  .build()
driver.get('http://www.google.com')
driver.findElement(webdriver.By.name('q')).sendKeys('webdriver')
driver.findElement(webdriver.By.name('btnG')).click()
driver.wait(() => {
  return driver.getTitle().then((title) => {
    return title === 'webdriver - Google Search'
  })
}, 1000)
driver.quit()
```

Using Playwright

Microsoft Playwright is an end-to-end testing framework built using browser-specific remote debugging protocols, similar to the Puppeteer headless Node.js API but geared towards end-to-end testing. Playwright has experimental Electron support via Electron's support for the Chrome DevTools Protocol (CDP).

Install dependencies

You can install Playwright through your preferred Node.js package manager. The Playwright team recommends using the `PLAYWRIGHT_SKIP_BROWSER_DOWNLOAD` environment variable to avoid unnecessary browser downloads when testing an Electron app.

```
PLAYWRIGHT_SKIP_BROWSER_DOWNLOAD=1 npm install --save-dev playwright
```

Playwright also comes with its own test runner, Playwright Test, which is built for end-to-end testing. You can also install it as a dev dependency in your project:

```
npm install --save-dev @playwright/test
```

:::caution Dependencies This tutorial was written `playwright@1.16.3` and `@playwright/test@1.16.3`. Check out Playwright's releases page to learn about changes that might affect the code below. :::

:::info Using third-party test runners If you're interested in using an alternative test runner (e.g. Jest or Mocha), check out Playwright's Third-Party Test Runner guide. :::

Write your tests

Playwright launches your app in development mode through the `_electron.launch` API. To point this API to your Electron app, you can pass the path to your main process entry point (here, it is `main.js`).

```
const { _electron: electron } = require('playwright')
const { test } = require('@playwright/test')

test('launch app', async () => {
  const electronApp = await electron.launch({ args: ['main.js'] })
  // close app
  await electronApp.close()
})
```

After that, you will access to an instance of Playwright's `ElectronApp` class. This is a powerful class that has access to main process modules for example:

```
const { _electron: electron } = require('playwright')
const { test } = require('@playwright/test')

test('get isPackaged', async () => {
  const electronApp = await electron.launch({ args: ['main.js'] })
  const isPackaged = await electronApp.evaluate(async ({ app }) => {
    // This runs in Electron's main process, parameter here is always
    // the result of the require('electron') in the main app script.
    return app.isPackaged
  })
```

```
  console.log(isPackaged) // false (because we're in development mode)
  // close app
  await electronApp.close()
})
```

It can also create individual Page objects from Electron BrowserWindow instances. For example, to grab the first BrowserWindow and save a screenshot:

```
const { _electron: electron } = require('playwright')
const { test } = require('@playwright/test')

test('save screenshot', async () => {
  const electronApp = await electron.launch({ args: ['main.js'] })
  const window = await electronApp.firstWindow()
  await window.screenshot({ path: 'intro.png' })
  // close app
  await electronApp.close()
})
```

Putting all this together using the PlayWright Test runner, let's create a `example.spec.js` test file with a single test and assertion:

```
const { _electron: electron } = require('playwright')
const { test, expect } = require('@playwright/test')

test('example test', async () => {
  const electronApp = await electron.launch({ args: ['.'] })
  const isPackaged = await electronApp.evaluate(async ({ app }) => {
    // This runs in Electron's main process, parameter here is always
    // the result of the require('electron') in the main app script.
    return app.isPackaged;
  });

  expect(isPackaged).toBe(false);

  // Wait for the first BrowserWindow to open
  // and return its Page object
  const window = await electronApp.firstWindow()
  await window.screenshot({ path: 'intro.png' })

  // close app
  await electronApp.close()
});
```

Then, run Playwright Test using `npx playwright test`. You should see the test pass in your console, and have an `intro.png` screenshot on your filesystem.

```
☁  $ npx playwright test

Running 1 test using 1 worker
```

```
  ✓  example.spec.js:4:1 › example test (1s)
```

:::info Playwright Test will automatically run any files matching the `.*(test|spec)\.(js|ts|mjs)` regex. You can customize this match in the Playwright Test configuration options. :::

:::tip Further reading Check out Playwright's documentation for the full Electron and ElectronApplication class APIs. :::

Using a custom test driver

It's also possible to write your own custom driver using Node.js' built-in IPC-over-STDIO. Custom test drivers require you to write additional app code, but have lower overhead and let you expose custom methods to your test suite.

To create a custom driver, we'll use Node.js' `child_process` API. The test suite will spawn the Electron process, then establish a simple messaging protocol:

```
const childProcess = require('child_process')
const electronPath = require('electron')

// spawn the process
const env = { /* ... */ }
const stdio = ['inherit', 'inherit', 'inherit', 'ipc']
const appProcess = childProcess.spawn(electronPath, ['./app'], { stdio, env })

// listen for IPC messages from the app
appProcess.on('message', (msg) => {
  // ...
})

// send an IPC message to the app
appProcess.send({ my: 'message' })
```

From within the Electron app, you can listen for messages and send replies using the Node.js `process` API:

```
// listen for messages from the test suite
process.on('message', (msg) => {
  // ...
})

// send a message to the test suite
process.send({ my: 'message' })
```

We can now communicate from the test suite to the Electron app using the `appProcess` object.

For convenience, you may want to wrap `appProcess` in a driver object that provides more high-level functions. Here is an example of how you can do this. Let's start by creating a `TestDriver` class:

```
class TestDriver {
  constructor ({ path, args, env }) {
    this.rpcCalls = []
```

```
    // start child process
    env.APP_TEST_DRIVER = 1 // let the app know it should listen for messages
    this.process = childProcess.spawn(path, args, { stdio: ['inherit', 'inherit',
'inherit', 'ipc'], env })

    // handle rpc responses
    this.process.on('message', (message) => {
      // pop the handler
      const rpcCall = this.rpcCalls[message.msgId]
      if (!rpcCall) return
      this.rpcCalls[message.msgId] = null
      // reject/resolve
      if (message.reject) rpcCall.reject(message.reject)
      else rpcCall.resolve(message.resolve)
    })

    // wait for ready
    this.isReady = this.rpc('isReady').catch((err) => {
      console.error('Application failed to start', err)
      this.stop()
      process.exit(1)
    })
  }

  // simple RPC call
  // to use: driver.rpc('method', 1, 2, 3).then(...)
  async rpc (cmd, ...args) {
    // send rpc request
    const msgId = this.rpcCalls.length
    this.process.send({ msgId, cmd, args })
    return new Promise((resolve, reject) => this.rpcCalls.push({ resolve, reject }))
  }

  stop () {
    this.process.kill()
  }
}

module.exports = { TestDriver };
```

In your app code, can then write a simple handler to receive RPC calls:

```
const METHODS = {
  isReady () {
    // do any setup needed
    return true
  }
  // define your RPC-able methods here
}
```

```
const onMessage = async ({ msgId, cmd, args }) => {
  let method = METHODS[cmd]
  if (!method) method = () => new Error('Invalid method: ' + cmd)
  try {
    const resolve = await method(...args)
    process.send({ msgId, resolve })
  } catch (err) {
    const reject = {
      message: err.message,
      stack: err.stack,
      name: err.name
    }
    process.send({ msgId, reject })
  }
}

if (process.env.APP_TEST_DRIVER) {
  process.on('message', onMessage)
}
```

Then, in your test suite, you can use your `TestDriver` class with the test automation framework of your choosing. The following example uses `ava`, but other popular choices like Jest or Mocha would work as well:

```
const test = require('ava')
const electronPath = require('electron')
const { TestDriver } = require('./testDriver')

const app = new TestDriver({
  path: electronPath,
  args: ['./app'],
  env: {
    NODE_ENV: 'test'
  }
})
test.before(async t => {
  await app.isReady
})
test.after.always('cleanup', async t => {
  await app.stop()
})
```