Brazil Project – PGDAS

Contents

[Project Description 1](#_Toc65054253)

[Puppeteer 2](#_Toc65054254)

[Pros 2](#_Toc65054255)

[Cons 2](#_Toc65054256)

[Implementation 2](#_Toc65054257)

[Reference 3](#_Toc65054258)

[Playwright 3](#_Toc65054259)

[Pros 3](#_Toc65054260)

[Cons 4](#_Toc65054261)

[Implementation 4](#_Toc65054262)

[Reference 4](#_Toc65054263)

[Cheerio 4](#_Toc65054264)

[Pros 5](#_Toc65054265)

[Cons 5](#_Toc65054266)

[Implementation 5](#_Toc65054267)

[Reference 5](#_Toc65054268)

[Jsdom 5](#_Toc65054269)

[Pros 5](#_Toc65054270)

[Cons 6](#_Toc65054271)

[Implementation 6](#_Toc65054272)

[Reference 6](#_Toc65054273)

[TestCafe 6](#_Toc65054274)

[Pros 6](#_Toc65054275)

[Cons 7](#_Toc65054276)

[Implementation 7](#_Toc65054277)

# Project Description

* Validate and implement web scraping strategies to generate PGDAS.
* Create a mock page to avoid sending inaccurate information to Brazilian revenue service.
* Create a Proof of Concept (PoC) to generate PGDAS using different technologies.
* Compare issues and performance using Selenium, Puppeteer and other suggested technologies.
* Test a simpler approach, using HTTP requests without automated testing tools.
* Create strategies to avoid blocks for excessive requests.

Initially suggested tools for web scrapping with NodeJS:

# Puppeteer

* Puppeteer is a Node library which provides a high-level API to control [headless](https://developers.google.com/web/updates/2017/04/headless-chrome) Chrome or Chromium over the [DevTools Protocol](https://chromedevtools.github.io/devtools-protocol/). It can also be configured to use full (non-headless) Chrome or Chromium.

## Pros

* Reduce execution time for applications having heavy GUI loading.
* Omit image load to speed up execution.
* Ability to switch on/off GUI based testing.
* Crawling ability.
* Capture a timeline trace of site to help diagnose performance issues.
* Less system set-up to spin up more test systems quickly.
* Testing in latest Chromium versions.
* Leveraging DevTools or its Protocol features.

## Cons

* Puppeteer is limited to Chrome browser only for now, until Firefox support is completed.
* Puppeteer scripting only available in JavaScript for Node.js, and it is unclear if other languages will be supported in the future.
* Puppeteer has a smaller testing community using the tool currently, there is more test-specific support for Selenium.

## Implementation

* Contains files in a following manner:
* webScrapping.js contains Scrape()
* breakCaptcha.js contains Image() and solver() and imports info() from PA.js
* PA.js contains info function and imports as follows:

1. checkDeclaration() from checkDeclaration.js
2. rectify() from rectify.js
3. calculationPeriod() from calculationPeriod
4. Activities() from Activities
5. clear() from clearHistory

* calculationPeriod imports revenueCondition() from revenueCondition

## Reference

* <https://developers.google.com/web/tools/puppeteer/get-started>
* <https://www.toptal.com/puppeteer/headless-browser-puppeteer-tutorial>

# Playwright

* Playwright enables fast, reliable and capable automation across all modern browsers.

## Pros

* Support for all browsers
* Test on Chromium, Firefox and WebKit. Playwright has full API coverage for all modern browsers, including Google Chrome and Microsoft Edge (with [Chromium](https://www.chromium.org/)), Apple Safari (with [WebKit](https://webkit.org/)) and Mozilla Firefox.
* Cross-platform WebKit testing. With Playwright, test how your app behaves in Apple Safari with WebKit builds for Windows, Linux and macOS. Test locally and on CI.
* Headless and headful. Playwright supports headless (without browser UI) and headful (with browser UI) modes for all browsers and all platforms. Headful is great for debugging, and headless is faster.
* Fast and reliable execution
* Auto-wait APIs. Playwright interactions [auto-wait for elements](https://playwright.dev/docs/actionability) to be ready. This improves reliability and simplifies test authoring.
* Timeout-free automation. Playwright receives browser signals, like network requests, page navigations and page load events to eliminate the need for sleep timeouts that cause flakiness.
* Lean parallelization with browser contexts. Reuse a single browser instance for multiple parallelized, isolated execution environments with [browser contexts](https://playwright.dev/docs/core-concepts).
* Resilient element selectors. Playwright can rely on user-facing strings, like text content and accessibility labels to [select elements](https://playwright.dev/docs/selectors). These strings are more resilient than selectors tightly-coupled to the DOM structure.
* Powerful automation capabilities
* Multiple domains, pages and frames. Playwright is an out-of-process automation driver that is not limited by the scope of in-page JavaScript execution and can automate scenarios with [multiple pages](https://playwright.dev/docs/multi-pages).
* Powerful network control. Playwright introduces context-wide [network interception](https://playwright.dev/docs/network) to stub and mock network requests.
* Modern web features. Playwright supports web components through [shadow-piercing selectors](https://playwright.dev/docs/selectors), [geolocation, permissions](https://playwright.dev/docs/emulation), web workers and other modern web APIs.
* Capabilities to cover all scenarios. Support for [file downloads](https://playwright.dev/docs/network) and [uploads](https://playwright.dev/docs/input), out-of-process iframes, native [input events](https://playwright.dev/docs/input), and even [dark mode](https://playwright.dev/docs/emulation).

Cons

* Legacy Edge and IE11 support. Playwright does not support legacy Microsoft Edge or IE11 ([deprecation notice](https://techcommunity.microsoft.com/t5/microsoft-365-blog/microsoft-365-apps-say-farewell-to-internet-explorer-11-and/ba-p/1591666)). The new Microsoft Edge (on Chromium) is supported.
* Java language bindings: The Playwright API cannot be used in Java or Ruby today. This is a temporary limitation as Playwright is built to support bindings for any language.
* Test on real mobile devices: Playwright uses desktop browsers to emulate mobile devices. If you are interested in running on real mobile devices, please [upvote this issue](https://github.com/microsoft/playwright/issues/1122).

Implementation

* Contains files in a following manner:
* webScrapping.js contains Scrape()
* breakCaptcha.js contains Image() and solver() and imports info() from PA.js
* PA.js contains info function and imports as follows:
  1. checkDeclaration() from checkDeclaration.js
  2. rectify() from rectify.js
  3. calculationPeriod() from calculationPeriod
  4. Activities() from Activities
  5. clear() from clearHistory
* calculationPeriod imports revenueCondition() from revenueCondition

## Reference

* <https://playwright.dev/docs/intro>
* <https://dev.to/dnature/web-scraping-scrape-data-from-your-instagram-page-with-nodejs-playwright-and-firebase-29d1#:~:text=Playwright%3A%20Playwright%20is%20a%20Node,%2C%20capable%2C%20reliable%20and%20fast>.

# Cheerio

* [Cheerio](https://cheerio.js.org/) is a Node.js library that helps developers interpret and analyze web pages using a jQuery-like syntax.

## Pros

* Cheerio provides developers with the ability to provide their attention to the downloaded data, rather than on parsing it.
* Familiar syntax: Cheerio implements a subset of core jQuery. It removes all the DOM inconsistencies and browser cruft from the jQuery library, revealing its truly gorgeous API.
* Lightening Quick: Cheerio works with a very simple, consistent DOM model. As a result, parsing, manipulating, and rendering are incredibly efficient. Preliminary end-to-end benchmarks suggest that cheerio is about 8x faster than JSDOM.
* Stunningly flexible: Cheerio can parse nearly any HTML or XML document.

## Cons

* It is not capable to parse JavaScript.
* It does not provide functionalities like taking screenshot and making pdf.
* Websites built with react or angular cannot be scraped with this.
* Cheerio can only works with raw HTML data.

## Implementation

* Dropped cheerio
* Reason: Cheerio parses markup and provides an API for traversing/manipulating the resulting data structure. It does not interpret the result as a web browser does. Specifically, it does not produce a visual rendering, apply CSS, load external resources, or execute JavaScript.

## Reference

* <https://cheerio.js.org/>
* <https://dev.to/diass_le/tutorial-web-scraping-with-nodejs-and-cheerio-2jbh>

# Jsdom

* jsdom is a pure-JavaScript implementation of many web standards, notably the WHATWG [DOM](https://dom.spec.whatwg.org/) and [HTML](https://html.spec.whatwg.org/multipage/) Standards, for use with Node.js. In general, the goal of the project is to emulate enough of a subset of a web browser to be useful for testing and scraping real-world web applications.

## Pros

* It *does* support the DOM, HTML, canvas, many other web platform APIs, and running scripts.
* If you are familiar with manipulating the DOM, then using JSDOM will be quite straightforward.
* One of the goals of jsdom is to be as minimal and light as possible.

## Cons

* The drawback of jsdom is that not everything can be simulated outside a real browser (you can’t take a screenshot for example) so using it will limit your test’s reach.
* jsdom is not a full browser, it does not perform layout or rendering, and it does not support navigation between pages.
* jsdom is awesome but it can be really hard to use if you need to parse non valid HTML you have to play with the [html5](https://github.com/aredridel/html5) parser module, which is super slow and the all thing is not dev friendly.

## Implementation

* Dropped jsdom
* Reason: jsdom is not a full browser: it does not perform layout or rendering, and it does not support navigation between pages. Also because of lack of proper documentation.

## Reference

* <https://www.npmjs.com/package/jsdom>
* <https://packages.tools.medtronicconnect.com/feeds/NPM/jsdom/9.9.1>
* <https://npmdoc.github.io/node-npmdoc-jsdom/build/apidoc.html>

# TestCafe

* TestCafe allows you to write tests using TypeScript or JavaScript (with its modern features like async/await). You get all the advantages of strongly-typed languages like rich coding assistance, painless scalability, check-as-you-type code verification, etc., by using TypeScript to write your TestCafe tests.

## Pros

* Fast. Browser is controlled from inside a web page. This makes test run inside a browser as fast as your browser can render page with no extra network requests.
* Simple Cross-Browser Support. Because TestCafe only launches browsers, it can automate browser on desktop or mobile. Unlike WebDriver, you don't need special version of browser and driver to prepare to run tests. Setup simplified. All you need is just a browser installed, and you are ready to go.
* Stable to Execution. Because a test is executed inside a browser, the network latency effects are reduced. Unlike WebDriver you won't hit stale element exceptions, or element not interactable exceptions, as from within a web browser all DOM elements are accessible.

## Cons

* Magic. Browsers executed in TestCafe are not aware that they run in test mode. So at some edges automation control can be broken. It's also quite hard to debug possible issues, as you don't know how actually a web page is parsed to inject automation scripts.
* No Browser Control. Because TestCafe do not control browser, you can't actually automate all users actions. For instance, TestCafe can't open new tabs or open a new browser window in incognito mode. There can be also some issues running tests on 3rd party servers or inside iframes.
* Simulated Events. Events like click or doubleClick are simulated by JavaScript internally. Inside WebDriver or Puppeteer, where those events are dispatched by a browser, called native events. Native events are closer to real user experience. So in some cases simulated events wouldn't represent actual user experience, which can lead to false positive results. For instance, a button which can't be physically clicked by a user, would be clickable inside TestCafe.

## Implementation