Brazil Project – PGDAS

Contents

[Project Description 1](#_Toc64910621)

[Puppeteer 2](#_Toc64910622)

[Pros 2](#_Toc64910623)

[Cons 2](#_Toc64910624)

[Implementation 2](#_Toc64910625)

[Reference 3](#_Toc64910626)

[Playwright 3](#_Toc64910627)

[Pros 3](#_Toc64910628)

[Cons 4](#_Toc64910629)

[Implementation 4](#_Toc64910630)

[Reference 4](#_Toc64910631)

[Cheerio 4](#_Toc64910632)

[Pros 4](#_Toc64910633)

[Cons 5](#_Toc64910634)

[Implementation 5](#_Toc64910635)

[Reference 5](#_Toc64910636)

[Jsdom 5](#_Toc64910637)

[Pros 5](#_Toc64910638)

[Cons 6](#_Toc64910639)

[Implementation 6](#_Toc64910640)

[Reference 6](#_Toc64910641)

[Nightmare 6](#_Toc64910642)

# 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/" \t "_blank)) 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>

# Nightmare

* Nightmare is a high-level browser automation library from [Segment](https://segment.com/).
* The goal is to expose a few simple methods that mimic user actions (like goto, type and click), with an API that feels synchronous for each block of scripting, rather than deeply nested callbacks.
* It was originally designed for automating tasks across sites that don't have APIs, but is most often used for UI testing and crawling.
* Under the covers it uses [Electron](http://electron.atom.io/), which is similar to [PhantomJS](http://phantomjs.org/) (a headless web browser scriptable with JavaScript) but roughly [twice as fast](https://github.com/segmentio/nightmare/issues/484#issuecomment-184519591) and more modern.

Pros: