EE 496 Report: InternAloha: Revamping Scrapers

Author: Jatin Pandya

Date: November 29, 2021

Abstract:

In this report, we take a look at the service InternAloha which is a scraper that scrapes different job sites for internships. In the report, I will go over the basics of InternAloha and scraping, the design of InternAloha, and the technical aspects that go into InternAloha.

1 Introduction

The original goal of RadGrad was to increase the retention rate in the ICS program. When a pilot study was conducted, a common comment students had was regarding the "Internship Opportunities" in the application[1]. InternAloha was the solution to that comment. InternAloha is a system that goes out to different sites such as LinkedIn and Glassdoor and scrapes internships related to Computer Science and Computer Engineering. InternAloha uses Puppeteer a web scraping tool that is used with Node.JS to scrape sites as its base and then builds the InternAloha scraper on top. And then the scraper goes through scraping the data from each site using Puppeteer functions as its base and gets the information that way. This information is then processed and then posted on a front-end site. The original InternAloha front-end site displayed the results from the scrapers. Now, this information is posted on the RadGrad site where the results are then filtered based on the student's interest, GPA, activities, and skills which gives students an idea of what internships are out there that they would be interested in[1]. This saves the student time from going through job sites to find internships and figuring out which ones they would be interested in.

The design of the scrapers starts off with the scraper starting up the browser and then grabbing the URL's or elements of each internship the scraper finds, and then going to each of the URL's or elements and then grabbing the data off the page. Once the data is scraped off the page it is then formatted into a specific format, and then is put into a JSON (JavaScript Object Notation) and then inserted into an array of internships and then later one last check is done to make sure that the internships are related to Computer Science and Computer Engineering by checking for terms such as "software engineering" and "hardware engineering".

The following image below (figure 1) shows the internships showing up in the RadGrad client based on the user jatinp@hawaii.edu's information. The information that jatinp@hawaii.edu gave to determine this is classes taken and interests. In this project, we aim to revamp the InternAloha scraper by improving the implementation of scraping in general to make the process of creating a scraper easier for users and to improve the way we get data from the sites we scrape so we aren't just scraping all internships that match our search criteria but also checking position names and descriptions to make sure they match for students.

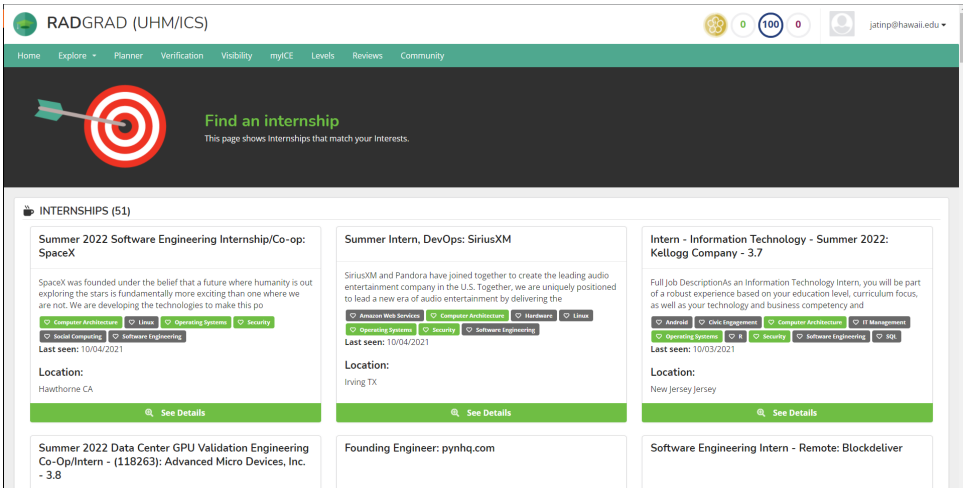

Figure 1 Image of InternAloha Scrape Results on RadGrad

In this report, I will go over the background information for scraping, what we did in this project in relation to the design, then going over an example of a scraper that I implemented that is in production and comparing it to the previous iteration of that scraper, and then going over future plans of the InternAloha project.

2 Related Work

The technology that the InternAloha service uses is called web scraping. According to Imperva, “ Web scraping is the process of using bots to extract content and data from a website. Unlike screen scraping, which only copies pixels displayed onscreen, web scraping extracts underlying HTML code and, with it, data stored in a database. The scraper can then replicate entire website content elsewhere.” The top 5 web scraping technologies that exist right now according to PopUpSmart.com are: Scrape.do, Scrapingdog, AvesAPI, ParseHub, and Diffbot. InternAloha differs from these existing technologies as they all charge for using them, while InternAloha is open source. InternAloha is also based on another web scraping technology called Puppeteer. Puppeteer is a node.js library that uses the chromium dev tools to grab the data. According to Puppeteer, it runs headless on default which means that there is no graphical user interface to interact with. Additionally, these web scraping tools allow for scraping on any site, while InternAloha does allow other sites to be scrapped but its main design is for internships from job sites.

Scraping being a popular way of grabbing data off of sites to use it for other purposes can run into legal issues. Most sites do say in their Terms and Service that any form of scrapping is prohibited. The reason why many sites do not want people to scrape their sites is that it could give a competitive advantage to competitors, scrapers might be scraping data that could be considered copyright, scraping can also cause sites to be flooded with requests which could cause the site to crash, and since each site has their own specific security measures they need to follow and scraping can be a security issue for them.

There was a lawsuit with a small data analytics company hiQ Labs against LinkedIn. What hiQ Labs wanted to do was to scrape public LinkedIn profiles but LinkedIn ended up serving hiQ a cease and desist notice. This then led hiQ to file a lawsuit against LinkedIn for the right to scrape and to prevent LinkedIn from invoking the Computer Fraud and Abuse Act (CFAA), the Digital Millennium Copyright Act (DMCA), California Penal Code § 502(c), or any trespassing law that could be used against hiQ . The trial was decided on the 9th of September 2019 with the United States Court of Appeals for the Ninth Circuit ruling in favor of hiQ labs stating that user’s private data was compromised as hiQ labs was only grabbing data that was publicly available.

What this means is that scraping for the most part is legal as long as the data that we are scraping is publicly available and is not hidden behind any login. Knowing these constraints InternAloha only scrapes data it can only find on a public page without requiring any login credentials which is considered “fair use”.

Additionally, InternAloha uses Google Cache as a scrape data which is legal and doesn't run the risk of companies coming after InternAloha. InternAloha also goes ahead and limits its activity on pages to avoid looking like a bot and to make the scrapers seem more "human" so sites do not IP(Internet Protocol) ban users of the scrapers. This means that InternAloha does not flood sites with requests to grab data. If a website does have a login and only displays the data that InternAloha needs, InternAloha needs to ask permission from the site owners to be able to scrape the site. Additionally, InternAloha also checks the site's robots.txt to make sure that when scraping, InternAloha is not disobeying the rules the sites have when it comes to scraping.

3 Design

The design of InternAloha scrapers is that they open a browser and then they go to a specific site and go ahead and grab URL's or element's and go to each one of them and then scrape the data from each page and then format and put it into a JSON and then into an array which is processed one more time before being sent into a JSON file. The original scraper did this in every scraper file which meant that lines of code were being reused over and over again across several files. These lines of code were when starting up the browser, going to a certain page, and fetching data from the page.

In the newly revamped scraper, takes the original scraper and goes ahead and adds a few features to improve the development of the scrapers and the usability of the scrapers. One of these improvements is a Scraper superclass. What this superclass does is take lines of code which is used multiple times across different files and create functions out of them. It also has a sequence of superclass functions that can be run while running the subclass function. What this means is when creating a child function of a parent function we can call the parent function to start things off and then write the rest of the particular code in the child function. In particular, when we are scraping and we want to get data, what we had to originally do is use this JavaScript function that was in a separate file and use it whenever we need to scrape data. Instead of doing this every time we instead now use functions getValues and getValue which are in the superclass of the scraper and use each one depending on whether we need just 1 or multiple values from scraping. The following image shows the implementation of both getValue and getValues.

```
63  /**
64   * Return a list of field values based on selector.
65   * Use this function when you expect the selector to match a list of elements in the page.
66   * @param selector The nodes to be selected from the current page.
67   * @param field The field to extract from the nodes returned from the selector.
68   */
69  async getValues(selector, field) {
70    return await this.page.$$eval(selector, (nodes, field) => nodes.map(node => node[field]), field);
71  }
72
73  /**
74   * Return a single field value based on selector.
75   * Use this function when you expect the selector match only a single element in the page.
76   * @param selector The node to be selected from the current page.
77   * @param field The field to extract from the node returned from the selector.
78   * @throws Error if there is no element matching the selector.
79   */
80  async getValue(selector, field) {
81    return await this.page.$eval(selector, (node, field) => node[field], field);
82  }
```

Figure 2: Implementation of getValues and getValue

Another change to the scraper was the way we handle listings. In the newly revamped version of the scraper, we have a specific class called listing which holds the specific information for each internship listing. It includes the variables URL, this gives us where users can go to apply for the specific internship they are looking at, which is a string, position, this tells the user what the position of the internship is, which is a string, location, this tells us where the internship is located, which is an object of strings of the city, state, and country, description, this tells us what the internship is about, which is a string, company, this tells us which company is hosting the internship, which is a string, contact, this tells us who to contact if there are any questions the student may have about the internship, which is a string, posted, this tells us when the internship was posted, which is a string, due, this is used to figure out when the internship application is due, which is a string, and lastScrape which a date which tells us the when we scraped this. Above that is a Listings object which holds an array of Listing. This object provides us with the structure to be able to create the array so we can hold all of the listings. From there we can use the function addListing to add to the listing array. We can use the function length to get the length of the listings. And then at the end, we can use the function writeListings to be able to write the listings to a JSON file. Compared to the original implementation we would just use a basic structure of a Listing but there was no Listing object.

```
export class Listing {
  public url: string;
  public position?: string;
  public location?: { city: string, state: string, country: string };
  public description?: string;
  public company?: string;
  public contact?: string;
  public posted?: string;
  public due?: string;
  public lastScraped: Date;
```

Figure 3: Listing class that we used when scraping

```
/** Each instance holds an array of writeable Listing objects. */
export class Listings {
  private commitFiles: boolean;
  private listings: Listing[];
  private listingDir: string;
  private name: string;
  private log;

  constructor({ listingDir, name, log, commitFiles }) {
    this.listingDir = listingDir;
    this.listings = [];
    this.name = name;
    this.log = log;
    this.commitFiles = commitFiles;
  }

  addListing(listing) {
    this.listings.push(listing);
  }

  length() {
    return this.listings.length;
  }

  writeListings() {
    try {
      const suffix = this.commitFiles ? 'json' : 'dev.json';
      const file = `${this.listingDir}/${this.name}.${suffix}`;
      const data = JSON.stringify(this.listings, replacer: null, space: 2);
      fs.writeFileSync(file, data, options: 'utf-8');
      this.log.info(`Wrote ${this.listings.length} listings.`);
    } catch (error) {
      this.log.error(`Error in Listings.writeListings: ${error}`);
    }
  }
}
```

Figure 4: Listings class that we use when scraping to hold each Listing

More specifically is the scrape function which follows a specific pattern to scrape. The scrape function first launches the scraper by starting the browser and going to a specific link that is given. It then logs into that scraper if necessary and then it goes ahead and starts to generate the listings. This is done by grabbing the URLs and creating the listings by scraping each listing for a specific structure for the listings array and taking the data and putting it into the listings object. And then finally there is a process listings function that takes the listings we generated and checks to see if the internships are the ones we need or to process the listings to format the listings in a certain way to make them look more uniform than the rest of the scrapers. If there is an error that gets caught at any step throughout this process it is then sent to the catch statement where it is then printed out to let the user know about what the error was. And then at the end,

the browser closes and then listings get written into the JSON file, and then a statistics file is also made letting the user know how long the scraper took and how many listings were created.

```
async scrape() {
  try {
    await this.launch();
    await this.login();
    await this.generateListings();
    await this.processListings();
  } catch (error) {
    const message = error['message'];
    this.errorMessages.push(message);
    this.log.error(message);
  } finally {
    await this.close();
    await this.writeListings();
    await this.writeStatistics();
  }
}
```

Figure 5: Specific pattern to scrape

Additionally, the original scraper was all done in JavaScript instead of in TypeScript. The reason why TypeScript is a better language to work with instead of JavaScript as it is more reliable which can make it easier to refactor files, it is also really good in handling types and is a perfect use for large and complex projects.

Another thing the new scraper uses is a Node.JS package called commander. What the commander package does is help create a command-line interface and help process options. This is used to help select a specific scraper and whether to put the scrapers on debug mode or not, there are also other options that InternAloha Scraper 2.0 supports such as for multiple disciplines. This means internships can be searched for students in computer science and computer engineering.

When the scraping is done when listings and statistics are made for each discipline is that each discipline is given a folder and for each scraper run their results are being saved in listing/discipline-name or statistics/discipline-name. This also allows for other disciplines to take this and use it for their disciplines. Another thing that this scraper does is automatically generate statistics. These statistics include how long a scraper took to scrape a file when the scraping occurred, the number of errors that occurred during the scrape, the number of listings that were generated during the scrape, the name of the scrape, and finally the error messages that were given if there were any errors that were generated.

Something that was removed from the InternAloha revamped scraper was multi-scraper support or the original ability to run all scrapers at the same time. The reason why this was removed was because of the way Puppeteer handles threads. Puppeteer is not considered as “thread-safe” which means that if any errors occur they may not appear. To fix this issue InternAloha scraper now only supports running scrapers one at a time. If you would like to run multiple scrapers at the same time the recommended procedure is to create a script that opens several terminal or command line windows as necessary to run as many scrapers as you have. This is shown in figure 6. What this does is separate the scrapers into different processes so if there is an issue that occurs in one of the scrapers it will for sure display for the scraper that ran into that issue. The remaining issues of what needs to be implemented in scrapers are cleaning up the Simply Hired scraper, implementing the American Express scraper, implementing the Monster scraper, implementing the ACM scraper, implementing the LinkedIn Scraper, and implementing the Indeed scraper.

Figure 6: Multi-Scraper running on revamped scrapers

4 Implementation

For this project, I implemented the Cisco scraper. The new implementation of the Cisco scraper is based on the original Cisco scraper. The Cisco scraper works well where it gets all the internships that it can find. We can check this manually by going directly to the site and comparing the statistics to the website. The Cisco scraper also does not crash and is stable so it can be run in production. Most of the changes in the Cisco Scraper have had to do with unnecessary code that could be simplified. For example, in the original implementation while getting data for the listings there is a loop that occurs which fetches data three times for no reason and puts it into a result array which is then returned to the scraping method. While in the new implementation there are only three lines and to get the data. The older implementation uses a function from a JavaScript file while this uses a function from the superclass. The below two images show the implementation of how each gets data. The top image shows the original implementation of getting the position data. And the bottom image shows the new implementation on how we get information from job postings.

```
async function getData(page) {
  const results = [];
  for (let i = 0; i < 3; i++) {
    // get title, location, description
    results.push(fetchInfo(page, 'h2[itemprop="title"]', 'innerText'));
    results.push(fetchInfo(page, 'div[itemprop="jobLocation"]', 'innerText'));
    results.push(fetchInfo(page, 'div[itemprop="description"]', 'innerHTML'));
  }
  return Promise.all(results);
}
```

Figure 7: Original Implementation of Get Data

```
    positions.push(await super.getValue('h2[itemprop="title"]', 'innerText'));
    descriptions.push(await super.getValue('div[itemprop="description"]', 'innerText'));
    const location = await super.getValue('div[itemprop="jobLocation"]', 'innerText');
    const locationTuple = location.split(', ');
```

Figure 8: New Implementation of Get Data

Another change to the implementation of the Cisco scraper is the way URL's are gathered. In the original implementation, there was an if statement and a while statement. The if statement checks if there is no next page link if there is a next page link it goes into a while loop. Both the inside of the if statement and the while loop are the same except in the while statement it checks for the next page link and clicks for the next page. In the new implementation that is

changed into first getting all URL's on the first page, then checking if there is a next page link and then if there are the loop starts and clicks on the next page and then grabs the URL's on that page and adds it on into the array. It keeps doing that until there the next page link no longer shows up. This method takes fewer lines of code and is also compartmentalized into the code from the superclass.

```
29      let hasPage = await page.$('div[class="pagination autoClearer"] a:last-child');
30      // Case when there is no "next" link.
31      if (hasPage === null) {
32        const urls = await page.evaluate(() => Array.from(
33          document.querySelectorAll('table[class="table_basic-1 table_striped"] tbody tr td[data-th="Job Title"] a'),
34          a => a.href,
35        ));
36        urlList.push(urls);
37      }
38      // Case where there is a next link
39      while (hasPage !== null) {
40        const urls = await page.evaluate(() => Array.from(
41          document.querySelectorAll('table[class="table_basic-1 table_striped"] tbody tr td[data-th="Job Title"] a'),
42          a => a.href,
43        ));
44        urlList.push(urls);
45        await page.click('div[class="pagination autoClearer"] a:last-child');
46        hasPage = await page.$('div[class="pagination autoClearer"] a:last-child');
47      }
```

Figure 9: Previous Implementation of Get URL

```
28
29      // Get all of the URLs to the Internship pages.
30      urls = urls.concat(await super.getValues(urlSelector, 'href'));
31      while (await super.selectorExists(nextLink)) {
32        await this.page.click(nextLink);
33        urls = urls.concat(await super.getValues(urlSelector, 'href'));
34      }
```

Figure 10: New Implementation of Get URL

A large change to the implementation of the newly implemented Cisco scraper is the way it scrapes data and puts it into an array and how it handles Listings. In the original implementation, it loops through a two-dimensional array of URLs, goes to the URL and then gets the data, and then puts it into a JSON object and then into an array. In the new implementation, a single for loop is run through the URLs goes to each URL, and then grabs the data. It then puts it into a Listing object which is then put into a Listing array. At the end of getting the data and putting it into the array in the first implementation, the JSON file gets written and then the browser is closed. But in the new implementation, the writing to the JSON file and the closing of the browser is done in the superclass after the process listings function. In the close function and writeListings function, which is a function inside the Listing object file.

```
    const lastScraped = new Date();
    for (let i = 0; i < urlList.length; i++) {
      for (let j = 0; j < urlList[i].length; j++) {
        await page.goto(urlList[i][j]);
        const [position, location, description] = await getData(page);
        data.push({
          url: urlList[i][j],
          position: position,
          company: 'Cisco',
          location: {
            city: location.match(/^([^,]*)/)[0],
            state: location.match(/([^ ,]*)$/)[0],
          },
          lastScraped: lastScraped,
          description: description,
        });
      }
    }
    await writeToJSON(data, 'cisco');
    await browser.close();
```

Figure 11: Old Implementation of Listing Loop

```

    // Get positions, descriptions, and locations.
    for (const url of urls) {
      this.log.debug(`Processing: ${url}`);
      await this.page.goto(url);
      positions.push(await super.getValue('h2[itemprop="title"]', 'innerText'));
      descriptions.push(await super.getValue('div[itemprop="description"]', 'innerText'));
      const location = await super.getValue('div[itemprop="jobLocation"]', 'innerText');
      const locationTuple = location.split(', ');
      locations.push({ city: locationTuple[0], state: locationTuple[1], country: '' });
    }

    // Create the listings from the parallel arrays.
    for (let i = 0; i < urls.length; i++) {
      const listing = new Listing({ url: urls[i], position: positions[i], location: locations[i], company: 'Cisco', description: descriptions[i] });
      this.listings.addListing(listing);
    }
  }
```

Figure 12: New Implementation of Listing Loop

```

  /**
   * Perform any final close-down operations.
   * Subclass: generally no need to override.
   */
  async close() {
    this.log.debug('Starting close');
    this.endTime = new Date();
    await this.browser.close();
    if (this.listings.length() < this.minimumListings) {
      this.log.error(`Generated listings (${this.listings.length()}) less than minimum listings (${this.minimumListings})`);
    }
  }

```

Figure 13: Closing of Browser

```
33
34  writeListings() {
35    try {
36      const suffix = this.commitFiles ? 'json' : 'dev.json';
37      const file = `${this.listingDir}/${this.name}.${suffix}`;
38      const data = JSON.stringify(this.listings, null, 2);
39      fs.writeFileSync(file, data, 'utf-8');
40      this.log.info(`Wrote ${this.listings.length} listings to ${file}.`);
41    } catch (error) {
42      this.log.error(`Error in Listings.writeListings: ${error}`);
43    }
44  }
45 }
```

Figure 14: Function that writesListings

The main differences between the two implementations are the way we scrape data and the way we process the data. This new implementation of the InternAloha scraper is better than the original scraper due to the way the code is handled. The code in the new implementation is compartmentalized and uses a class-based system. What this means is that the most used functions are coded in the superclass where it is easily able to be used and not have to copy and paste the function into the code multiple times. It also makes it easier to program another scraper as a lot of the details are taken care of and all that is needed is to figure out what elements/URLs are needed to scrape and then process through the results to format it better and to filter out the internships that do not need to be there.

5 Conclusion & Future Directions

The InternAloha project is used to scrape internships from different job sites to be displayed on the RadGrad Internship page. In this project, I revamped the original Cisco scraper by using TypeScript over JavaScript, implementing functions from a parent scraper class, and making the code more straightforward to understand. While on this project I learned how to work in a team effectively, setting time management habits while researching, studying, and working, and I learned how scrapers work and use that information to implement them. What remains to be completed in the InternAloha project is first the completion of the rest of the scrapers. The remaining issues from the GitHub issues page, of what needs to be implemented in scrapers are cleaning up the Simply Hired scraper, implementing the American Express scraper, implementing the Monster scraper, implementing the ACM scraper, implementing the LinkedIn Scraper, and implementing the Indeed scraper. Additionally, going forward from there we can add a country option to when running the scraper. This would allow users to be able to scrape internships in other countries. This is good as it can allow students to explore outside of the country and be able to gain new experiences and it can open students up to going to other places in the world rather than just looking at the mainland or just staying in Hawaii. Another future step is also to start contacting sites that require login to view the internships to see if we are allowed to scrape. This is so we can make sure we are not trespassing and to avoid any legal issues that could possibly come our way. A far in the future project would be to have listings be automatically sent to the RadGrad when running the scraper as opposed to right now where listings need to be pushed up to the repository.

References

[1]"InternAloha", *InternAloha*, 2021. [Online]. Available: internaloha.github.io. [Accessed: 20- Nov- 2021].

[2]"Web Scraping", *Imperva*, [Online]. Available: https://www.imperva.com/learn/application-security/web-scraping-attack/. [Accessed: 20- Nov- 2021].

[3]"README.md", *InternAloha*, 2020. [Online]. Available: https://github.com/internaloha/scrapers. [Accessed: 28- Nov- 2021].

[4]G. Dreimanis, "Why You Should Choose TypeScript Over JavaScript", *serokell*, 2020. [Online]. Available: https://serokell.io/blog/why-typescript [Accessed: 28- Nov- 2021].

[5]"Web Scraping Tools", *popupsmart*, [Online]. Available: https://popupsmart.com/blog/web-scraping-tools [Accessed: 20- Nov- 2021].

[6]"README.md", *Puppeteer*, 2020. [Online]. Available: https://github.com/puppeteer/puppeteer. [Accessed: 28- Nov- 2021].