Scraping for Students' futures: An Indepth look at the InternAloha Revamping Process

Caliana Fortin

Abstract

InternAloha was made to remedy the internship problem that Radgrad had. Through web scraping InternAloha was able to help Radgrad increase the amount of internships students had available to them. This paper looks at how the InternAloha web application was implemented and details the revamping process of the structure, and goals of the original InternAloha system.

The Beginning of My Scraper Journey

This fall 2021 semester I was introduced to the web application known as InternAloha. InternAloha was a student-implemented web application that uses JavaScript and Puppeteer to scrape websites that have computer science internships, such as Indeed or SimplyHired. According to the article How To Scrape a Website Using Node.js and Puppeteer by Gbadebo Bello “Web scraping is the process of automating data collection from the web. The process typically deploys a ‘crawler’ that automatically surfs the web and scrapes data from selected pages” [5].

The main goal of InternAloha is to remedy the Radgrad internship problem by scraping websites for computer science internships [7]. Website scraping allows for there to be an abundant amount of internships available that fulfill a widescope of interests. Previously, Radgrad internships had to be manually input. This resulted in only a few select internships being available thus as a result InternAloha was made.

Since I was just getting my feet wet with scraping I needed to understand the general design of the InternAloha system and how it worked. I used the article How To Scrape a Website Using Node.js and Puppeteer by Gbadebo Bello to understand the web scraping process [5]. The website details the basic design of a simple scraper, which is as follows:

```
const scraperObject = {
    url: 'http://books.toscrape.com',
    async scraper(browser){
        let page = await browser.newPage();
        console.log(`Navigating to ${this.url}...`);
        // Navigate to the selected page
        await page.goto(this.url);
        // Wait for the required DOM to be rendered
        await page.waitForSelector('.page_inner');
        // Get the link to all the required books
```

```
            let urls = await page.$$eval('section ol > li',
              links => {
                // Make sure the book to be scraped is in
                   stock
                links = links.filter(link =>
                link.querySelector('.instock.availability >
                i').textContent !== "In stock")
                // Extract the links from the data
                links = links.map(el => el.querySelector('h3
                      > a').href)
                return links;
            });
            console.log(urls);
        }
    }
```

From this example I had a general sense of how the scrapers for InternAloha were designed. Looking at the InternAloha system each scraper was implemented in their own file then imported into a main file. Each scraper could be initiated by using console commands like the following: `npm run scrape -- -s nsf`

And would return the following output:

```
$ npm run scrape -- -s nsf

> scraper@2.0.0 scrape
> ts-node -P tsconfig.buildScripts.json scrape.ts "-s"
"nsf"

11:19:24 WARN NSF Launching NSF scraper
11:19:41 INFO NSF Wrote 100 listings.
11:19:41 INFO NSF Wrote statistics.
$
```

Looking closer at each individual scraper and using the basic design of a simple scraper, we can identify the general trend in how the websites were scraped. For websites that did not require a login, the general design was as follows: First the scraper would visit the website's url. Then it would go to the search bar and type “Computer Science Internships” and hit enter. From there the scraper would actually start scraping the website by looking for things like position, location, and job description. Finally, after compiling all the data the code would put it all into a JSON file. From there this JSON data could then be uploaded to Radgrad, where the internships would be available.

Each scraper that did not need to login utilized this basic design. Scrapers like Angellist and Student Opportunity Center utilized a similar design except after going to the website's url we would have the code login then start scraping.

Finally, after learning about the basic design of the InternAloha scrapers I wanted to implement one myself, so I could get a stronger grasp on scraping.

Learning the Ins and Outs of Scraping

As referenced above web scraping is "... the process of automating data collection from the web " [5]. This simply means that we employ the use of a computer to go to web sites and extract the data we want. In the case of InternAloha, we visit websites like Indeed or LinkedIn and extract the computer science internships listings.

There are many ways to scrape a website, but in our case we need something that works with Javascript, that's where Puppeteer comes in. Currently for the InternAloha system Puppeteer is the tool that we will be using to scrape websites. According to Puppeteer documentation, Puppeteer is ".. a Node library which provides a high-level API to control Chromium or Chrome over the DevTools Protocol" [10]. Basically, Puppeteer is like an add on that we can download which provides us with the ability to scrape websites using JavaScript. Puppeteer works by using Chrome's DevTool Protocol which is simply a way for Puppeteer to "communicate" to the Chrome browser and the pages in the browser [2].

In order to use Puppeteer to scrape a webpage we need to first know how to open the browser and navigate to a webpage . The documentation provides the following to show how to do that [10]:

```
const puppeteer = require('puppeteer');

(async () => {
 const browser = await puppeteer.launch();
 const page = await browser.newPage();
 await page.goto('https://www.google.com');
 // other actions...
 await browser.close();
})();
```

In this code we can see that in order to use Puppeteer we must employ the use of async(). "async()" is simply a function that is asynchronous which allows us to change up the flow of our code, which is why we use the await keyword to tell the computer to wait until this action is completed before going to the next instruction. The reason we use the async function is because of the way websites work. Websites vary in the amount of time they load, so we need to wait until the website is loaded until we do anything, thus the reason we use async() and await.

To actually start getting data from a webpage we have to employ the use of the Puppeteer class called page. The page class allows us to control the behavior of how we scrape a webpage. There

are a lot of functions from the page class that are useful for scraping but I will only cover the ones commonly used in the InternAloha web application.

Page.waitForSelector(selector) allows us to wait for the selector to appear on the page. The waitForSelector() function returns true if the selector is there, else it throws an error. This function is particularly useful when we know the element may not be on the page immediately. For InternAloha page.waitForSelector() is often used in conjunction with another page function like page.$(selector) or page.type().

Page.$(selector) returns the value of the selector we have chosen. Like for example a position title or a location of a job. If there is no selector that matches the given selector then null is returned.

Page.setDefaultTimeout(timeout) simply allows us to wait for a given amount of milliseconds before we do anything. This is particularly useful for waiting for an element to be loaded.

Page.type(selector, text) allows us to type into the given selector. InternAloha uses this to type into the search bar for computer science internships. If it can type in the selector then a value of true is returned, if it can't then the request is rejected and an error is thrown.

Page.click(selector) clicks on the selector of our choice. If it can click on the selector then a value of true is returned, if it can't then the request is rejected and an error is thrown. For InternAloha we often use this function to click on the internship or search button.

Page.evaluate(pageFunction) this allows us run a function in context of the information on the page. A return value of true is returned along with pageFunctions return value. For InternAloha page.evaluate() is often used to create an array from the value of the selectors on the page, like for example:

```
const urls = await page.evaluate(() =>
Array.from(document.querySelectorAll('a[class="table--adv
anced-search__title"]'), a => a.href));
```

This piece of code returns an array of urls using page.evaluate to create that array.

Page.goto(url) simply goes to the url of the website that we want to go to. In the case of InternAloha page.goto() goes to the url of the website that holds computer science internships, like for example indeed.com or glassdoor.com .

In general these are the base page functions that InternAloha uses to scrape web pages. Now that we know about the tools that we need to scrape a website we must also learn about the legality of scraping websites. The main thing that makes InternAloha unique is that we aren't trying to steal information from those websites and market it as our own, but rather increase "visibility" of that information to students. InternAloha still provides the original link of the website that it came from that students can click on, thus increasing the "visibility" of that information.

Regardless of this fact there are still some legal issues about scraping that need to be addressed. Many websites like Indeed have implemented tools to help protect themselves from web scrapers, thus the question stands if websites are deliberately trying to stop us from scraping their website, should I be scraping their website? If we look at the terms and services of Indeed we find the following:

> Use of any automated system or software, whether operated by a third party or otherwise, to extract data from the Site (such as screen scraping, crawling, reproducing, duplicating, copying, selling, trading or reselling) is prohibited. Publisher agrees that it is solely responsible for (and that Indeed has no responsibility or liability to it or to any third party for) any breach of Publisher's obligations under these IPP Terms and for any consequences (including any loss or damage which Indeed may suffer) of any such breach. [9]
>
> Use of any automated system or software, whether operated by a third party or otherwise, to extract data from the Site (such as screen scraping or crawling) is prohibited. Indeed reserves the right to take such action as it considers necessary, including issuing legal proceedings without further notice, in relation to any unauthorized use of the Site. [9]
>
> You agree not to access (or attempt to access) the Site by any means other than through the interface that is provided by Indeed, unless you have been specifically allowed to do so in a separate, written agreement with Indeed. You agree that you will not engage in any activity that interferes with or disrupts the Site (or the servers and networks which are connected to the Site). Unless you have been specifically permitted to do so in a separate, written agreement with Indeed, you agree that you will not crawl, scrape, reproduce, duplicate, copy, sell, trade or resell the Site for any purpose. [9]
>
> Without limiting the foregoing and by way of example only, users may not: [9]
> - Scrape the Indeed Resume Search database
> - Scrape or otherwise replicate any Indeed content for competitive purposes

From this information we can conclude that Indeed does not allow scraping if it means that we publish that information as our own or if we use that information commercially. However, it also states that if they do find you scraping their website they reserve the right to issue legal action.

The situation even gets cloudier if you consider the HiQ Labs v. LinkedIn case [3]. In 2019 LinkedIn sent HiQ a cease-and-desist order, asking it to stop accessing and copying data from its servers. HiQ filed a lawsuit against LinkedIn in California, seeking injunctive relief and a declaratory judgment to prevent LinkedIn from invoking the Computer Fraud and Abuse Act (CFAA), the Digital Millennium Copyright Act (DMCA), California Penal Code 502(c), or the common law of trespass against HiQ. In 2021 the supreme court ruled that ".. any data that is publicly available and not copyrighted is fair game for web crawlers" [11]. This means we cannot scrape data that is protected behind a login.

Taking into account the Supreme Court's ruling we can deduce that as long as we are scraping information that is publicly available in a respectful way and not using it commercially then scraping is entirely legal.

Respectful scraping entails not visiting the webpage too often and navigating it at an appropriate speed. We don't want to mess up the website by scraping it, thus by scraping it at moderate speed we can limit the harm we do.

Now that I know the ins and outs of scrapping I can actually start addressing some of InternAloha's problems.

Addressing the Elephant in the Room: InternAloha

The original InternAloha application had numerous problems ranging from efficiency and legality. The biggest issue I personally saw was the way the scrapers were implemented. Each scraper was implemented in a different way. This made it hard to identify the most efficient implementation and basic design of a scraper. Below is a picture showing the implementation of two different scrapers:

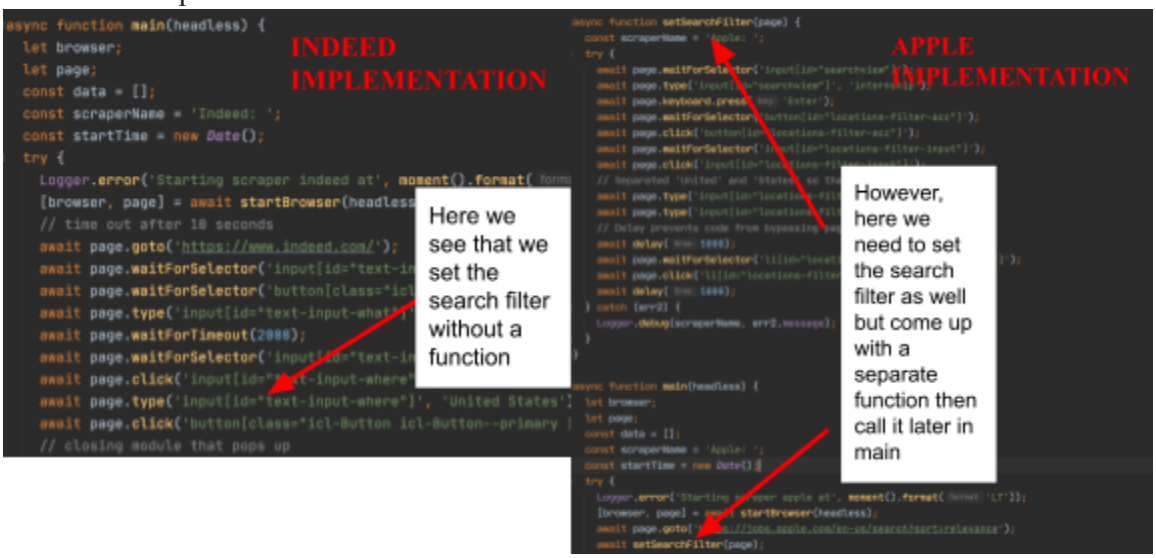

Figure 1

In the Indeed scraper we generally just scrape the website in the main function and set the filter here but for the apple scraper we come up with a function that sets the search filter called setSearchFilter. This is just one example showing the disconnected nature of the original code. The non-cohesive code made it extremely hard for me to begin implementation of my own scraper. A lot of the code was incredibly hard to read. Some code had embedded if statements which made it hard to figure out what the code actually does. Below shows a picture of some of the common if statements that I would find.

```
if (posted.includes('day') || posted.includes('days')) {
  daysBack = 0;
} else if (posted.includes('month') || (posted.includes('months'))) {
  // 'a month ago...'
  if (posted.includes('a')) {
    daysBack = 30;
  } else {
    daysBack = posted.match(/\d+/g) * 30;
  }
} else {
  daysBack = posted.match(/\d+/g);
}
date.setDate(date.getDate() - daysBack);
const time = date;
let state = '';
// if there is no state tag
if (!location.match(/([^,]*)/g)[2]) {
  state = 'United States';
} else {
  state = location.match(/([^,]*)/g)[2].trim();
}
```

Figure 2 code from the original Chegg scraper

This code is confusing because no comments are provided explaining what it's doing. From an outsider perspective I assume it's setting the variable called daysBack based on different if statements. If the posted object included day then we set dayBack to 0, but if it doesn't we set the variable to something else. It would be better to include comments telling you what this piece of code achieves and possibly why something is done in a certain way. Without explanation it is super difficult to improve or even fix the code without knowing what it does.

The main issue about scraping is that websites are volatile, meaning that websites like chegg or indeed are likely to change the design of their website. For InternAloha this means that our scrapers are very likely to break, thus meaning we need an efficient way to fix a scraper. InternAloha contained a lot of spaghetti code which made it incredibly hard to maintain it. Spaghetti code is simply defined as: " Unstructured and hard-to-maintain code caused by lack of style rules or volatile requirements. This architecture resembles a tangled pilde of spaghetti in a bowl " [6].

The faults of the original InternAloha were the things we wanted to fix in InternAloha 2.0 . The first thing to tackle in the new system was establishing the basic design structure of each scraper. From previous evaluations of the original scrapers it was deduced that each scraper had the following flow: go to website, navigate/scrape website using page functions, and finally compile data.

Each scraper was structured according to the flow that was identified. A scraper template was implemented to show new developers how to implement a scraper. The template is as follows:

```
import { Scraper } from './Scraper';

const prefix = require('loglevel-plugin-prefix');

export class TemplateScraper extends Scraper {
  constructor() {
    super( {name, url}: { name: 'template', url: 'https://testscrapersite.com' });
  }

  async launch() {
    await super.launch();
    prefix.apply(this.log, options: { nameFormatter: () => this.name.toUpperCase() });
    this.log.warn(`Launching ${this.name.toUpperCase()} scraper`);
  }

  async login() {
    await super.login();
    // if you need to login, put that code here.
  }

  async generateListings() {
    await super.generateListings();
    // here is where you traverse the site and populate your this.Listings field with the listings.
  }

  async processListings() {
    await super.processListings();
    // here is where you do any additional processing on the raw data now available in the this.listings field.
  }

}
```

Figure 3 showing the structure that scrapers should follow

Each scraper is implemented into their own class and extends the parent class Scraper. This simply allows the scraper to have access to all of the parent classes functions. We create an instance of the scraper using a constructor specifying the name of our scraper and the general url of our scraper.

After creating an instance we have an async function called launch which calls our parents function launch using super to launch the chromium browser. After, if the website requires us to be logged in we have a function called login() that logs us in to that site, if we don't need to login you can just delete the function. After this we have a function called generateListings() which is where we actually traverse the website and scrape data from that website and put it into a JSON file. Finally if we need to do anything extra to that scraped we can process it in the function processListings().

The generateListings() also has its own structured flow that it should follow. After gathering and extracting all the internship listings you have created a new Listing object with all the appropriate fields such as position, location, url and description. After creating the Listing object you need to add that object to Listings which hold the whole array of all the internship listings we obtained.

The next thing we tackled was identifying duplicated code fragments. Any pieces of code that were commonly used in many scrapers were implemented into a utilities file, like for example retrieving the value of a field was a common piece of code used in every scraper so a getValue function was implemented in the utilities file. Below is the getValue function:

```
/**
* Return a single field value based on the selector.
* Use this function when you expect the selector to match
only a single element in the page.
```

```
* @param selector The node to be selected from the
current page.
* @param field The field to extract from the node
returned from the selector.
* @throws Error if there is no element matching the
selector.
*/
async getValue(selector, field) {
 return await this.page.$eval(selector, (node, field) =>
node[field], field);
}
```

Observing the code, you can see that a comment statement is provided detailing exactly what the function does and what it returns. This makes it incredibly easy to understand how to use the function.

The final thing we tackled was how each individual scraper was implemented. The main goal of transferring InternAloha's original code to InternAloha 2.0 was to simplify it. We needed to reimplement the code from scratch using our new found structure. InternAloha 2.0's new structure helped us to reduce multiple duplicated fragments thus reducing the size of the original code. New strategies like going to the direct link instead of navigating using the search bar filter allowed us to get rid of like 10-20 lines of code.
Overall the main rules for implementing the scraped was to follow the given structure in the template, write comments where needed, and simplify code when you can.

The last issue Intern Aloha 2.0 tackles is the issue of legality. After identifying the legality of scraping through research, we have started to reach out to companies like Indeed or LinkedIn to avoid legal issues. However, taking into account the supreme court's ruling of the HiQ Labs v. LinkedIn case [3] we can deduce that as long as we are scraping information that is publicly available in a respectful way and not using it commercially then scraping is entirely legal.

As a result of learning this Intern Aloha 2.0 has been implementing more human-like scraping, by employing the use of tests, a Puppeteer plugin called StealthPlugin, setting the user agent, and using Google cache to avoid detection.

In order to make sure our bots acted as human-like as possible, websites like https://bot.incolumitas.com/ and https://antoinevastel.com/bots/datadome helped us to test if we were navigating websites "respectfully" initial test revealed that our bots did not behave like a human at all. However as a result of this a thing called user agents was discovered which would help our scrapers appear less bot-like.

According to this article titled User Agents for WebScraping 101 by Josh Vanderwillik User agents are defined as follows "the user agent string helps the destination server identify which browser, type of device, and operating system is being used " [8]. The article also states that the reason you should use different user agents is because without a user agent a website can

automatically detect that you are a bot, but with a user agent it makes it a bit more difficult to identify if your scraper is a bot or not. Some websites completely block scrapers that are obvious bots and others only allow user agents they believe are adequate enough to perform scraping. As a general rule of thumb you should always aim to use a real user agent. The website NetWorking Howtos has an article that displays all of the common and real user agents used [1].

For InternAloha 2.0 we implemented the use of user agents using a package which provided us with all of the common user agents.

For some websites the google cached website was used. Google cached websites are snapshots of the websites that are cached in google's server. Usually the websites are just a few weeks old which is good enough for the purposes of gathering internships for students. Scraping google cached websites provide complete protection from detection, making it the best and safest option for scraping.

Overall, InternAloha 2.0 reimplemented the original InternAloha in order to make the code more readable, efficient, and overall easier to use. InternAloha 2.0 also helped to address the legality of scraping.

After understanding the overall improvements that InternAloha 2.0 did we can start looking closely at the Apple and Chegg scraper I personally implemented.

The Re-Implementation of the Apple and Chegg Scraper

The very first scraper I implemented after we found the general structure we wanted was the Apple scraper. Looking at the original Apple scraper there was a noticeable flow of how the website was scraped. First we would set the search filter of the apple website, then we would get the total amount of pages and save that into a variable, after we would iterate through each page and gather the total urls for each page, finally after getting all the urls we would iterate through each url and extract information such as position, location, and description.

```
async generateListings() {
  await super.generateListings();
  let pageNum = 1;
  const listingsTable = '#tblResultSet';

  // pageUrl returns an URL containing the specified page number.
  const pageUrl = (pageNum) =>
    `https://jobs.apple.com/en-us/search?location=united-states-USA&sort=relevance&search=internship&page=${pageNum}`;

  // Get the first page of Internship listings.
  await super.goto(pageUrl(pageNum));
```

Figure 4 Shows the implementation of directly going to the filtered link

This was the general flow that I was aiming for however I discovered that the search filter code was not necessary to scrape the apple website instead it was more efficient to just set the filters in the urls so instead of going to the website like this: https://jobs.apple.com/en-us/search?sort=relevance. I would navigate to the website that already had the filters in the url so like this: https://jobs.apple.com/en-us/search?location=united-states-USA&sort=relevance&search=internship&page=1
This already sets the filter of the location and the type of job we are looking for which is intern.

```
while (await super.selectorExists(listingsTable)) {
  // Collect the URLs to the listings on this page.
  let urls = await super.getValues( selector: 'a[class="table--advanced-search__title"]', field: 'href');
  this.log.info(`Processing page ${pageNum} with ${urls.length} listings.`);
  // Retrieve each URL, extract the internship listing info.
  for (let i = 0; i < urls.length; i++) {
    const url = urls[i];
    await this.page.goto(url);
    const company = 'Apple';
    const position = await super.getValue( selector: 'h1[itemprop="title"]', field: 'innerText');
    const description = await super.getValue( selector: 'div[id="jd-description"]', field: 'innerText');
    const state = await super.getValue( selector: 'span[itemprop="addressRegion"]', field: 'innerText');
    const city = await super.getValue( selector: 'span[itemprop="addressLocality"]', field: 'innerText');
    const location = { city, state, country: 'USA' };
    const listing = new Listing({ url, position, location, company, description });
    this.listings.addListing(listing);
  }
  // Increment the pageNum and get that page. If we get a page without listings, then listingsTable selector won't be on it.
  await this.page.goto(pageUrl(++pageNum), { waitUntil: 'networkidle0' });
}
}
```

Figure 5 shows the implementation of the while loop and for loop

In order to make the code more readable a while loop and a nested for loop was used. In the original Apple scraper a for loop was nested inside a for loop, which made things a bit hard to understand. In my implementation it is clear that the while loop works as long as the table is present and once we go through each url we increment to the next page until there is no table element. The code is very straightforward and works relatively well as of now. The scraper is able to extract and add the correct number of listings in the correct format.

Now moving on to my implementation for the Chegg scraper. The Chegg scraper was probably one of the more annoying scrapers I had to try to implement. Navigating to the website myself I noticed there was an obvious issue. The Chegg website employed the use of infinite scroll. Infinite scroll is basically when you scroll down a page and more content is loaded. This was significantly more difficult than the Apple scraper. Also when I was navigating the Chegg website I noticed that sometimes when scrolling it would force me back all the way to the first internship listing.

My plan of attack was to reference the original Chegg scraper, but upon further investigation the original scraper did not work and plus it was a mess of spaghetti code. The next logical step was

research. I googled “puppeteer and infinitescroll”. From my google search I found a lot of stack overflow responses which helped with my quest of creating the Chegg scraper.

So my code works as follows: We go to the filtered link, and we create an array of all the internship boxes on the page. Below shows an image of the internship boxes, which held the listing.

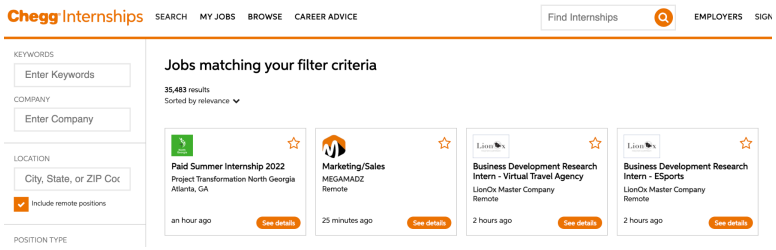

Figure 6 internship boxes

```
async generateListings() {
  await super.generateListings();
  let pageNum = 1;
  const listingsTable = '#tblResultSet';

  // pageUrl returns an URL containing the specified page number.
  const pageUrl = (pageNum) =>
    `https://jobs.apple.com/en-us/search?location=united-states-USA&sort=relevance&search=internship&page=${pageNum}`;

  // Get the first page of Internship listings.
  await super.goto(pageUrl(pageNum));

  while (await super.selectorExists(listingsTable)) {
    // Collect the URLs to the listings on this page.
    let urls = await super.getValues( selector: 'a[class="table--advanced-search__title"]', field: 'href');
    this.log.info(`Processing page ${pageNum} with ${urls.length} listings.`);
    // Retrieve each URL, extract the internship listing info.
    for (let i = 0; i < urls.length; i++) {
      const url = urls[i];
      await this.page.goto(url);
      const company = 'Apple';
      const position = await super.getValue( selector: 'h1[itemprop="title"]', field: 'innerText');
      const description = await super.getValue( selector: 'div[id="jd-description"]', field: 'innerText');
      const state = await super.getValue( selector: 'span[itemprop="addressRegion"]', field: 'innerText');
      const city = await super.getValue( selector: 'span[itemprop="addressLocality"]', field: 'innerText');
      const location = { city, state, country: 'USA' };
      const listing = new Listing({ url, position, location, company, description });
      this.listings.addListing(listing);
    }
    // Increment the pageNum and get that page. If we get a page without listings, then listingsTable selector won't be on it.
    await this.page.goto(pageUrl(++pageNum), { waitUntil: 'networkidle0' });
  }
}
```

Figure 7 Chegg scraper implementation

From there we enter a loop which loops through all the currently displayed internship boxes we found. Then in a promise statement (to ensure that this action follows through) we click on the nth element of our array holding the internship boxes. After the page loads of the internship we gather the relevant information like the position, location, url, and company. We add the information into a listing and add the listing. Finally navigate back, wait for the page to load and start the whole process over by refreshing the newly loaded elements into the elements array. Figure 7 shows the implementation.

The last thing that I discovered while scraping the Chegg website is that it would often list non computer science related listings. I had to implement the process listings function to fix this problem. After getting all the listings, I created an array called word which held all the relevant words I wanted to search for in the description. An example of some of the words I was looking for was “computer science” , “programming”, and “software”. Basically my plan was that if

some of the words in the description matched my words array then it probably is a computer science internship. However, if it did not match I would delete it. Figure 8 shows the implementation of the process listing function.

```
// here is where you do any additional processing on the raw data now available in the this.listings field.
async processListings() {
  await super.processListings();
  this.log.info('Processing Listings');
  var removedCount = 0; //the number of non relevant listings removed
  const words = ['Computer Science', 'programming', 'software']; //the words we want to search for in the description

  // TODO: Replace this with filter() at some point.
  this.listings.forEach(function (listing, index, object) {
    //This tests to see if some words from our array are in the description, returns true if there is
    const test = words.some( predicate: word => listing.description.includes(word));
    //If the test fails then we remove the element from the listings
    if (!test) {
      //splice will remove the non-matching object
      removedCount += 1;
      object.splice(index, 1);
    }
  });
  this.log.info(`Removed ${removedCount} nonrelevant listings, total now: ${this.listings.length()}`);
}
}
```

Figure 8 Process listings implementation

Compared to the old Chegg scraper that could only get 23 listings, my implementation could get 200-300 listings. Overall the implementation for the Chegg scraper works pretty well, but of course later when I get a chance I would like to use filter() opposed to foreach() for the processListings function. Also the amount of code used to write the new Chegg scraper is much smaller and more concise. The original Chegg scraper utilized 187 lines, but my implementation uses only 106 lines.

Overall, I feel like I added enough comments for someone to understand what I am doing and help debug the code if needed.

The Future for InternAloha

The new main goal of InternAloha now is to keep it up to speed with all the new scraping tools or techniques. I think it is important to make sure that each scraper is working to the best of their ability, so I think eventually some of the scrapers may have to be improved.

Currently the Chegg scraper takes about an hour or two to run, so I think possibly looking into how it could run faster is a possibility for the future.

InternAloha has great potential for being completely automated. I think it would be cool to have the InternAloha system upload the JSON file to Radgrad itself rather than having to do it manually.

New web scrapers could be added. I think it would be nice if we added the functionality of looking for internships for either computer engineering or computer science. We could add the

functionality to already production working scrapers. This would also help the computer engineering side of Radgrad.

References

[1] "Common User Agent List." *Networking HowTos*, 2 June 2017, https://www.networkinghowtos.com/howto/common-user-agent-list/.

[2] *Chrome DevTools Protocol - Github Pages*. https://chromedevtools.github.io/devtools-protocol/.

[3] "Data Scraping Survives! (at Least for Now) Key Takeaways from 9th Circuit Ruling on the HIQ vs. Linkedin Case." *The National Law Review*, https://www.natlawreview.com/article/data-scraping-survives-least-now-key-takeaways-9th-circuit-ruling-hiq-vs-linkedin.

[4] https://www.scotusblog.com/case-files/cases/linkedin-corp-v-hiq-labs-inc/

[5] DigitalOcean. "How to Scrape a Website Using Node.js and Puppeteer." *DigitalOcean*, DigitalOcean, 17 May 2021, https://www.digitalocean.com/community/tutorials/how-to-scrape-a-website-using-node-js-and-puppeteer.

[6] Flower, Zachary. "Fix Spaghetti Code and Other Pasta-Theory Antipatterns." *SearchSoftwareQuality*, TechTarget, 29 May 2020, https://searchsoftwarequality.techtarget.com/tip/Fix-spaghetti-code-and-other-pasta-theory-antipatterns.

[7]"Internaloha: Internaloha." *InternAloha Blog RSS*, https://internaloha.github.io/documentation/.

[8] Josh Vanderwillik. "User Agent for Web Scraping." *Bright Data*, 15 Nov. 2021, https://brightdata.com/blog/how-tos/user-agents-for-web-scraping-101.

[9] "Legal." *Job Search*, https://www.indeed.com/legal.

[10] *Pptr.dev*, https://pptr.dev/.

[11] Waterman, Tom. “Web Scraping Is Now Legal.” *Medium*, Medium, 27 Apr. 2020, https://medium.com/@tjwaterman99/web-scraping-is-now-legal-6bf0e5730a78.