Web Scraping Independent Study

Introduction:

Web scraping is the process of extracting specific bits of information from web pages, usually with the help of 'crawler' scripts that collect and process the said information. Regardless of implementation, any scraping process involves the following steps:

- fetch the contents of the target page
- extract data of interest from the page's markup
- parse through the data to filter and transform it as needed
- write the results to a database or hard drive

One of the most popular language picks for scraping applications is JavaScript, which offers a number of packages for fetching and analyzing web data. To be more specific, data extraction can be easily accomplished with the help of modules like 'axios.js' or 'request', parsing - with the help of 'cheerio.js', and saving - with the helps of 'fs' or 'puppeteer'.

Initially, JavaScript was used to build client-side interfaces working exclusively for browsers, meaning that it could not run independent applications such as scrapers and crawlers. However, with the introduction of NodeJS, JavaScript could now dabble into the realm of web-independent applications. NodeJS is a programming environment made exclusively for running JavaScript applications, thus allowing them to interact with computer resources directly (e.g. reading/storing files on computer memory).
As opposed to most environments, which deal with concurrency by creating a separate thread for each request, NodeJS efficiently interleaves everything within a single process. This allows Node.js to efficiently handle thousands of concurrent requests to a single server without introducing much overhead, making it a highly versatile choice for scraping applications.

Fetching Data (Axios.js):

As mentioned earlier, the first step in developing a scraper application is finding an efficient method of sending requests to other web hosts and retrieving the information of interest. Gladly, most informational and forum websites are static, meaning that their contents do not diverge between consecutive requests due to asynchronous background operations. Communication with such hosts can be accomplished using any client tool that allows to send and receive HTTP requests.
In JavaScript, one of the most popular of such tools is Axios.js, a promise-based HTTP client compatible browsers and NodeJS applications. This library supports the Promise API and allows us to use Node's native HTTP module for sending requests, thus avoiding the need for any additional dependencies.

Axios provides a set of shorthand methods for performing different types of requests, most common of which are .get() and .post() (standing for GET and POST requests, respectively). These methods expect to be passed two arguments: the URL of the target host and a configuration object with request details (optional).

As an example, one can make a POST request using the axios .post() method to forward data to a given endpoint and trigger events. This snippet simply instructs Axios to send a POST request to /login with an object of key/value pairs as its data. Axios will automatically store this data in a JSON object and send it back to the client

```
axios.post('/login', {
  firstName: 'John',
  lastName: 'Doe'
})
.then((response) => {
  console.log(response);
}, (error) => {
  console.log(error);
```

```
});
```

Once an HTTP POST request is made, Axios returns a promise object that indicates whether the initial request was fulfilled or rejected. To handle the response, we can chain-call the .then() method on the returned object. In case of success (i.e. promise fulfilled), the structure executes the procedure passed as the first argument, while in case of error (i.e. promise rejected) - the one passed as the second argument.

Alongside sending information, axios also allows us to request data from other servers by calling the .get() method. The get() method requires two parameters: the URL of the service endpoint and an object that contains the properties we want to send to our server. The following snipped of code requests the contents of the r/nodejs subreddit's page and displays the returned data upon success:

```
const axios = require('axios')
axios.get('https://www.reddit.com/r/nodejs.json')
 .then((response) => {
  console.log(response)
 })
 .catch((error) => {
  console.error(error)
 });
```

Parsing Data (RegEx/Cheerio):

Once we request and obtain the source code, we can move on to the next step in the process - parsing markup to extract information of interest.
The easiest and most straightforward to get started with web scraping without any dependencies is by using regular expressions. To do so, one would just parse the HTML string that we have fetched with the HTTP client using regular expressions. While great for simpler input, this method is relatively inefficient in larger applications, as regular expressions tend to grow increasingly complicated when parsing nuanced text.
To further elaborate, regular expressions are patterns used to match character combinations in strings, and in JavaScript, they are represented as objects with a number of properties and methods that help us in parsing over text. We can pass these patterns as arguments to string matching methods (such as .match() or .matchAll()) to extract the desired bits of information.
As an example, let's say a string variable htmlString contains markup which at some point includes a <label> element. To extract the text stored between the tags, we could use the .match() method with a selector tailored specifically to this instance:

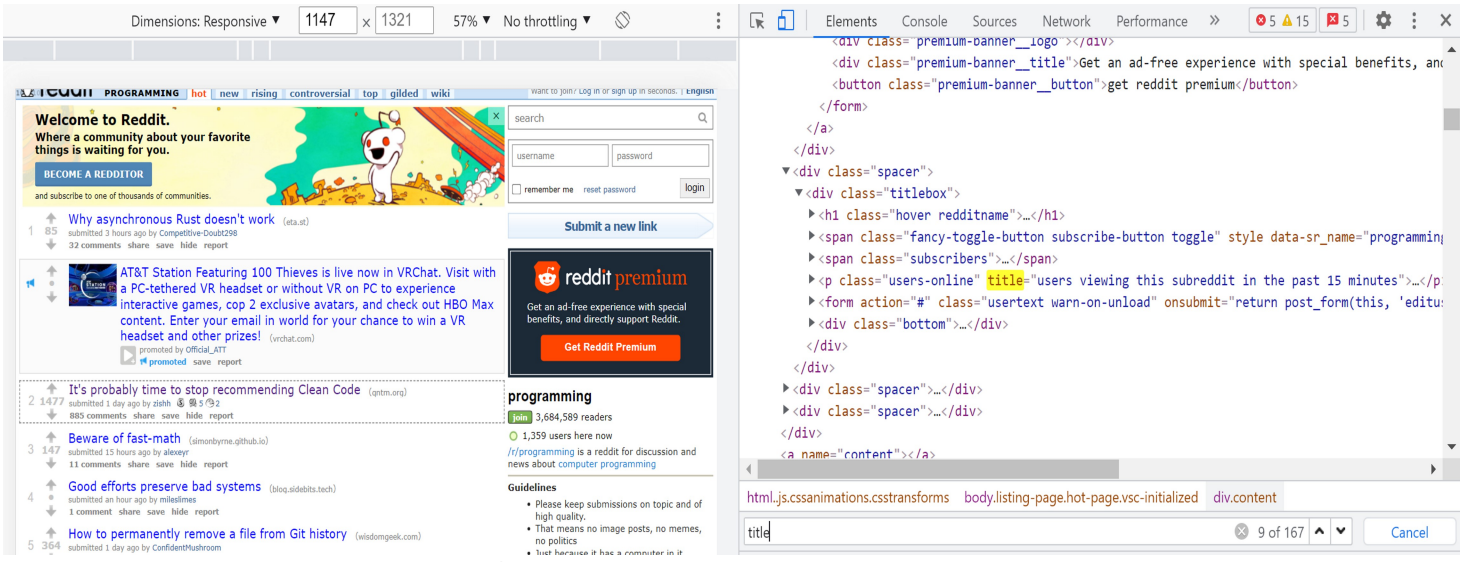
```
const htmlString = '...<label>Username: John Doe</label>..'
const result = htmlString.match(/<label>(.+)<\/label>/)
console.log(result[1].split(": ")[1])
//This code prints out 'Username: John Doe' to the console.
```

For more intricate analysis, it is preferred to rely on dedicated HTML parser libraries, and NodeJS offers a few to choose from. One of the most efficient and lightweight parser libraries is Cheerio.js, which provides a server-side traversal/selection interface based off of jQuery syntax. According to its documentation, Cheerio parses raw HTML markup and allows to conveniently manipulate the resulting data structure while removing any inconsistencies related to document structure and bookkeeping.
The following example involves a Cheerio-based program that crawls the static rendering of the r/programming subreddit and extracts a list of post titles:

```
//import dependencies
const axios = require('axios');
const cheerio = require('cheerio');
const getTitlesList = async () =>
{
 try
```

```
{
  //request the markup of the page
  const { data } = await axios.get('https://old.reddit.com/r/programming/');
  const $ = cheerio.load(data);
  const titlesList = [];
  //iterate through each extracted title and store it in the list
  $('div > p.title > a').each(  (_idx, el) =>
  {
    const postTitle = $(el).text()
    titlesList.push(postTitle)
  } );
  return titlesList;
 }
 catch (error)
 {
  throw error;
 }
};
//upon succesful extraction, display the list in the console
getTitleList()
.then((titlesList) => console.log(titlesList));
```

After importing the required libraries, we define an asynchronous function called getTitlesList(), which would crawl and extract the data from the target page. First, we obtain the HTML contents of the website using the aforementioned axios.get() method. Next, we feed the data to the cheerio.load() method, which returns an '$' object later used to parse the markup.
To determine the exact HTML selectors that identify target elements, we need to inspect the target web page using Chrome Dev Tools. Upon closer examination, we can see that post titles are contained within link elements (tag <a>) belonging to the div's of class 'title'; closer examination shows that all titles can be selected using the 'title' class selector. Following the Cheerio's jQuery-like command structure, we can conveniently obtain a list of all title elements with the single selector $('div > p.title > a').
Next, given that we want to access each title individually, we would iterate over all entries using $'s .each() member function. To extract the contents of each element (referred to as 'el' in this case), we apply the .text() method and save the result in an array of all post titles. As we run the program, we get a display of this list of about 30 post titles, ready for further evaluation and analysis.



Caption: screenshot of source code inspection using Chrome Dev Tools

Dynamic Options(Puppeteer):

Nevertheless, it's worth noting that while Cheerio is capable of traversing website elements like a browser, it does not produce a visual rendering, apply CSS, or execute embedded JavaScript. Thus, Cheerio 's utility is limited to mostly static websites, which are not heavy on JavaScript or interactive CSS. For dynamic websites that rely on asynchronous embedded JavaScript, it's best to resort to headless browsers such as 'Puppeteer' or its more intricate counterpart 'Nightmare'.

The following snippet fulfills a similar task of extracting and printing post titles; this time, however, it's being applied to the newer (dynamic) version of Reddit's main page. This can be accomplished with the help of Puppeteer, a lightweight headless browser extension for NodeJS that allows to interact with a website by using nothing more than a couple, of lines of code. Firstly, we import the module with a 'require' statement, given that all the packages have already been installed with npm.
Next, we call puppeteer's .launch() method that boots up an instance of the browser and returns a promise, which is then resolved with 'await' (alternatively, we could use .then()). This method also takes an optional parameter to choose whether or not we want to see a visual rendering of browsing, and it defaults to invisibility. Then, we open a new implicit browser tab with the help of the .newPage() method and navigate it to the target by calling page.goto(url).
Afterwards, we select the overlaying html element ('siteTable') that we will search for the target title links 'p.title > a'. We can extract these elements and save them in an array by calling page.$$eval() method and passing the selector as an argument. Lastly, we print the array containing target elements, which is now ready for further method of processing of choice.

```
const puppeteer = require('puppeteer');
    let url = 'https//reddit.com/r/programming/'

async scraper(browser){

   const browser = await puppeteer.launch;
   //'open' new browser tab
   let page = await browser.newPage();
   console.log(`Navigating to ${this.url}...`);
   //navigate newly created tab to target webpage
   await page.goto(url);
  //get the outer DOM element
   await page.waitForSelector('#siteTable');
   // Get titles of all posts
   let titles = await page.$$eval('div > p.title > a');
   console.log(titles);
   await browser.close();
}
```

It's worth mentioning that the capabilities of the Puppeteer module go far beyond just retrieving data. According to its documentation, some of the module's many capabilities include:
Generate screenshots and PDFs of pages.
Automate form submission, UI testing, keyboard input, etc.
Create an up-to-date, automated testing environment. Run your tests directly in the latest version of Chrome using the latest JavaScript and browser features.
Test Chrome Extensions.
This next short snipped shows a modification of the previous code; here, instead of extracting data, we take a screenshot of the requested page and save it as a pdf.

```
async scraper(browser){

   const browser = await puppeteer.launch;
   //'open' new browser tab
   let page = await browser.newPage();
   console.log(`Navigating to ${this.url}...`);
```

```
    //navigate newly created tab to target webpage
    await page.goto(url);
    //take a screenshot of page and save it as jpeg
    await page.screenshot( { path: 'screenshot.png'} );
    //save a pdf rendering of the page
    await page.pdf( {path: 'page.pdf' } );
    await browser.close();

    }
```

Permanent Storage (FS):

Once all the required data is extracted, parsed, filtered, and modified, it is time to permanently save it to computer memory to allow for its further processing. One of the modules that greatly simplifies file IO is the NodeJS fs module. It's is quite straightforward to use; in our case, we only need the writing capabilities provided by the fs.writeFile() method. The method has the format fs.writeFile(file, data, [options, callback]), where file (required) is the name of the output file, data (required) is any structure one wishes to save, options allows to modify smaller functionalities (not required), and callback is the function executed upon failed completion to handle errors(not required).

Quite often, the data we obtain from parsing modules is returned in a form of sets, arrays, or objects. For improved consistency, it is best to transform the data into a standard format (like json) before outputting it to file. The following snippet illustrates one such example:

```
    const container  =[]
some_elements.each((idx, el) => {
  // create composite object for storing data
  const obj = { attr1: "", attr2: ""};
  //Retrieve some desired attributes
  obj.attr1 = $(el).children("some selector" ).text();
  obj.attr2 = $(el).children("some selector").text();
  container.push(obj);
});
//transform data to a json string
let transformed =  JSON.stringify(container);
// Write array to json file
fs.writeFile("output.json", transformed,
(err) => { if (err) {console.error(err);} });
```

Here, we save each attribute in an object literal, store all object literals in an array, and then transform the array into a formatted json strnig using the JSON.stringify() method. We then output the formatted data onto the output json file.

Future plans:

I believe my work so far has barely scratched the surface of the vast world of web scraping. Given that there is increasing demand in sentiment analysis and AI data training, this field is projected to keep expanding, introducing new convenient tools. My next step on this journey would most probably be to explore different methods of retrieving large sets of data and processing them in an efficient way. Moreover, I need to identify which sorts of data to look for to ensure that the results of my analysis are telling of a meaningful trend.
Moreover, I would love to get more familiar with the impressive capabilities of headless browsers like Puppeteer, as they appear to be an extremely useful tool in testing all sorts of web applications. Puppeteer seems to also have a number of more advanced siblings, such as Nightmare, which offer an even wider selection of tools for scraping and intricate UI testing.