Building a custom news aggregator using Python, MySQL, and Node.js

Capstone Project

Paolo Gomez
Computer and Information Science
SUNY Polytechnic Institute
Utica, NY
gomezpa@sunypoly.edu

***Abstract*— The importance of the news cannot be overlooked. News plays a crucial role in informing people about current events, issues, and developments that affect their lives. It allows people to stay informed about what is happening in their communities, their country, and the world. With the growth of the internet, there has been an increase in the amount of information available online. In today's fast-paced world, staying up to date with the latest news is more important than ever. As the amount of information available online continues to grow, there is a growing need for tools that allow users to efficiently access and organize this information. In this paper, we will explore the process of scraping news articles and data from the New York Times using Python. Additionally, I will cover how we can store this data in a MySQL database and use Node.js to build a custom web application to display the articles.**

Keywords: MySQL, Python, Node.js

I. Introduction

News plays a vital role in our daily lives, keeping us informed about current events happening in the world. Accessibility to news plays a crucial role in our democracy. It allows people to make informed decisions and be less susceptible to misinformation. News is also an important tool for holding those in power accountable. Access to news ensures that people are aware of the actions of those in power. It allows us to hold these individuals accountable for their decisions and actions.

However, keeping up to date with current events in a fast-paced world is a difficult task to do. Especially with the amount of information that exists on the Internet. It can be difficult to keep track of everything and find the news that matters most to us. Manually browsing through multiple websites to find relevant articles can be time-consuming. This is where tools like web scraping come into play.

In this report, we will explore the process of scraping news articles and data from the New York Times using Python. I will demonstrate how we can use BeautifulSoup, a Python library, to extract data from RSS feeds and HTML pages. We will also cover how we can store this data in a MySQL database and use Node.js to build a custom web application, which will display the articles. The web app will be hosted on a Raspberry Pi 4 Model B and will be hosted on my home network.

Please note that the New York Times' Terms of Service states in Section 4 that using "robots, spiders, scripts, service, software or any manual or automatic device, tool, or process designed to data mine or scrape the Content, data or information from the Services, or otherwise access or collect the Content, data or information from the Services using automated means [4]." I am just doing this project for research purposes and to get a better understanding of how scrapers work. Therefore, I strongly advise against scraping articles from the New York Times, and I do not condone or encourage such actions.

II. Background

A. What is an RSS feed?

RSS (Really Simple Syndication) is a web feed that allows users and applications to access updates to websites in an easy-to-read file called XML[2]. Information fetched by a user's RSS feed reader converts the file and the latest updates from website into an easy-to-read format. The feed takes headlines, summaries, and update notices, and then links back to articles on your favorite website's page. The content found in feeds are distributed in real time, so that the top results on the RSS feed are always the latest published for a website.

Figure 1: Example of NYT's RSS feed

Figure 1 above shows an example of what an RSS feed looks like. The file is written in XML and for this project, I will be going through different feeds and scraping data from

specific tags. The use of RSS feeds guarantees getting the most recent articles from the NYT.

B. *Setting up the Raspberry Pi*

For this project, I used a Raspberry Pi 4 Model B, and it was running Ubuntu Server 22.10. Installing Ubuntu on the Pi was easy to do with the Raspberry Pi Imager software.

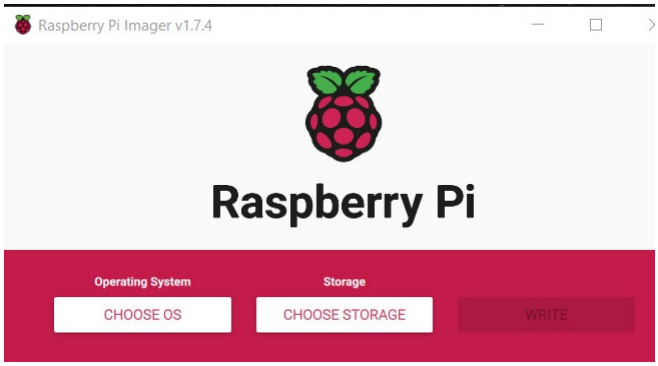


Figure 2: Screenshot of the software used to install Ubuntu.

Figure 2 above shows the menu of the Raspberry Pi Imager software. The software includes a quick way to install the Ubuntu operating system with the "Choose OS" menu. Figure 3 below shows the options for OS that one can choose to install.

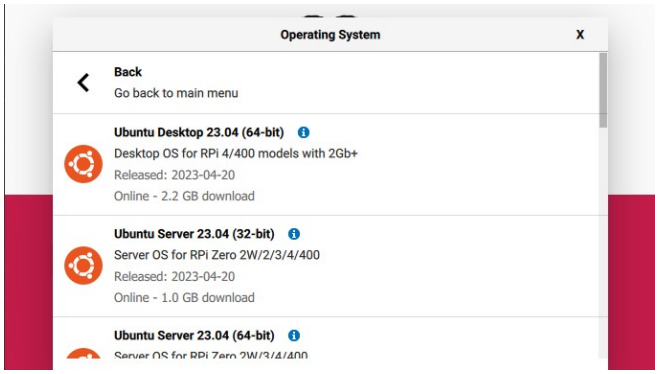


Figure 3: Screenshot of OS menu

Next, you can choose which storage device you would like to install the OS on. Finally, you can click on "Write" and the software takes care of the rest. To make things easier, you can press Ctrl + Shift + X on your keyboard and an advanced options menu pops up.

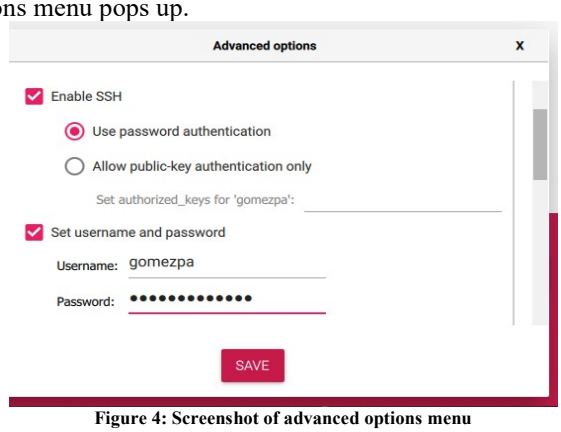


Figure 4: Screenshot of advanced options menu

The advanced options menu can be used to setup a username and password and enable SSH. Once the software finished installing the OS on a micro-SD card, I booted the Pi up. I enabled SSH in the advanced options menu because I did not have an extra monitor. I just found the IP address of the Raspberry Pi on my home network and used that to SSH into it. Once logged in, I performed a full update on the OS and installed MySQL, and Node.js. MySQL is needed to create the database needed for the website. Node.js is used to create the server-side website for this project. Additionally, I installed UncomplicatedFirewall (UFW) which is a simple to use firewall on Ubuntu.

C. *Creating a Virtual Environment*

Beautiful Soup is a Python library that is used for pulling data out of HTML and XML files. It works with a parser to provide idiomatic ways of navigating, searching, and modifying the parse tree[1]. It can be installed with the system package manager or with pip3. For this project, I created a Python environment for the following reasons.

- Access to Python libraries is easier to install and manage.
- Isolated from other environments and does not affect other Python projects.

I created a directory, 'Capstone', that contains pip, the interpreter, scripts, and libraries. Figure 5 below shows results of using the command `python3 -m venv Capstone`.

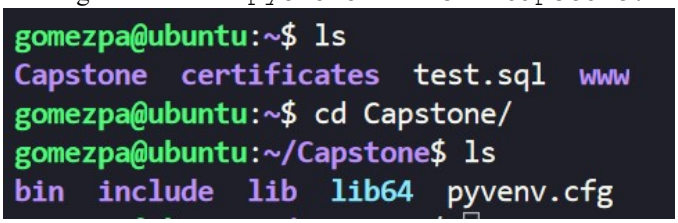


Figure 5: Contents of the 'Capstone' directory.

With this new directory, we can use pip3 to install the Python libraries we need to create the scraper. For this project, I installed the following libraries:

- MySQL
- BeautifulSoup
- Requests
- Html-parser
- Lxml

The MySQL library is necessary to establish a connection with the database. The requests library allows us to send HTTP requests extremely quickly. We will be using this library to grab the code from the RSS feed link. Then, we will use Beautiful Soup and a parser, html-parser or lxml, to extract the data from the HTML or XML file. Finally, we use the SQL connection to store the data in the database.

D. *Installing and creating the MySQL database*

On Ubuntu, we can install the database using the command `sudo apt install mysql-server`. I logged in as root and changed the root password for security reasons. Then I created a user '`gomezpa`' and granted the user permissions with the MySQL command "`GRANT CREATE, ALTER, DROP, INSERT, UPDATE, INDEX, DELETE,`

SELECT, REFERENCES, RELOAD on *.* TO 'gomezpa'@'localhost' WITH GRANT OPTION;". This user will be in the Node.js app and Python code to establish a connection to the tables we need in the database.

With the 'gomezpa' user, I created two databases 'test' and 'NYT'. The 'test' database will be used to hold the 'users', 'nyt_us', 'favorites', 'nytHome_US', 'nytToday_US', amd 'nytToday_US' tables. Figure 6 below shows the tables that reside inside the 'test' database.

```
mysql> show tables;
+----------------+
| Tables_in_test |
+----------------+
| favorites      |
| nytHome_US     |
| nytToday_NY    |
| nytToday_US    |
| nyt_us         |
| users          |
+----------------+
```

Figure 6: The tables inside of the test database.

The 'users' table holds the user's email address, username, hashed password, and an id that automatically increments.

The 'nyt_us' table holds the scraped data from the RSS feed like the title, description, date published, the link to the full article, and an article_id that automatically increases.

The three tables 'nytHome_US', 'nytTodayUS', and 'nytHomeUS' hold the same information as the 'nyt_us' table except it has an additional column that holds the full article. The 'nytHome_US' table holds the scraped data from the NYT's Home Page (U.S.) RSS feed. The 'nytToday_US' table holds the scraped data from the NYT's U.S News RSS feed. Finally, the 'nytToday_NY' table holds the scraped data from the NYT's New York RSS feed.

The 'favorites' has three columns, an id that automatically increments, a user id that references the id in the 'users' table, and an id the that references the id in the 'nyt_us' table. This table is used when a user favorites an article found from the 'nyt_us' table. When this happens, the user's id and the article id is stored in the 'favorites' tables.

```
mysql> show tables;
+---------------+
| Tables_in_NYT |
+---------------+
| news_article  |
| news_items    |
+---------------+
2 rows in set (0.01 sec)
```

Figure 7: The tables stored in the 'NYT' database.

Figure 7 above shows the two tables stored in the 'NYT' database, 'news_article' and 'news_items'. The 'news_article' table stores the full text of an article and an 'article_id'. The 'news_items' table holds the title, description, link to the article, and two IDs, 'news_id' and 'article_id'. The 'article_id' is used as a foreign key to the 'article_id' field in the 'news_items' table.

With the creation of these tables, we can successfully store the data scraped and use it to display on a web application.

E. Installing Node and packages

This project will be using Node.js to create the web application that will display the data scraped with BeautifulSoup. To make sure there aren't problems with package dependencies, I created a new directory that will hold all the Node packages necessary.

```
gomezpa@ubuntu:~$ ls
Capstone  certificates  test.sql  www
gomezpa@ubuntu:~$ cd www
gomezpa@ubuntu:~/www$ ls
Capstone_Proposal.txt    app.js  node_modules  nytTodayNY.py  package-lock.json  public         scraper.py    test.js
Capstone_Report_Log.txt  db.js   nytHomeUS.py  nytTodayUS.py  package.json       rss_scraper.py  scraperv2.py  views
```

Figure 8: The contents inside the 'www' directory.

First, I created the 'www' directory that will hold the Node modules and files for the web app itself. Figure 8 above shows the 'www' directory that currently holds all my project files.

Once the directory was created, we can install Node through their website. By installing Node, you also install npm (node package manager), which is used to install the packages necessary for our app. The dependencies needed are the following and can be installed with `npm install`:

- Bcrypt (used for password hashing)
- Dotenv (used to load environment variables from a .env file)
- EJS (generate HTML markup with plain JavaScript)
- Express
  - Express-session
- MySQL

Express and express-session is a Node.js web application framework that provides broad features for building web and mobile applications.

```
const express = require('express');
const app = express();

app.set('view engine', 'ejs');

app.get('/', function(req, res) {
    res.render('home');
});

app.listen(8080, function(err) {
    if (err) console.log(err);
    console.log("Site started on port 8080");
});
```

Figure 9: Basic app.js file

Now that the packages were installed, I created a simple Node application called app.js. Figure 9 above shows an app that is listening on port 8080. It has a route to the file 'home' and is using the EJS view engine. This means that we can render .ejs files. We can start the program with the command `node app.js`. We can tell it was successful when the terminal outputs "Site started on port 8080". For now, the '/' route will display nothing until we add code to the '`home.ejs`' file.

III. Process

Now that I have created a basic web app, the database, and downloaded the necessary packages, I began creating the Python script that will scrape the RSS feed that I give it. This script will need to connect to the database and store the information that was scraped. Once the scripts are fully functional, I will create a webpage that will present the articles in a card format.

A. Creating Python scripts with Beautiful Soup and MySQL

To begin, I used BeautifulSoup's documentation to understand how to scrape content.

1. Choose the website we want to scrape and inspect it to see how its HTML or XML is structured.
2. Start writing your Python script by importing the necessary modules (requests and BeautifulSoup)
3. The code must make a request to the website and store the response.
4. Using BeautifulSoup, we can parse through the HTML or XML and extract the information we want.

```
#!/home/gomezpa/Capstone/bin/python

from datetime import datetime
from bs4 import BeautifulSoup
import requests

NYT_US = requests.get('https://rss.nytimes.com/services/xml/rss/nyt/US.xml', timeout=10)

soup = BeautifulSoup(NYT_US.content, 'xml')
items = soup.find_all('item')

for item in items:
    title = item.title.text
    description = item.description.text
    link = item.link.text
    pubDate = item.pubDate.text
```

Figure 10: Basic scraper using Beautiful Soup and requests.

Figure 10 above is a screenshot of the basic scraper I made with the help of the documentation[3]. First, I import beautifulsoup and requests with the `import requests` and `from bs4 import BeautifulSoup` commands. Next, a request is made to the link https://rss.nytimes.com/services/xml/rss/nyt/US.xml, and the response is stored in the variable '`NYT_US`'. The response is parsed through with Beautiful Soup and extract information from the 'item' elements. Figure 1 has an example of what response looks like. In that example, we can see that there are `<item>` tags that hold the tags we want:

- `<title>`
- `<description>`
- `<pubDate>`
- `<link>`

Beautiful Soup provides the '`find_all`' method to find all mentions of a certain element in the XML response. In this case, I use '`find_all`' to find all the `<item>` tags and use a '`for`' loop to iterate through each `<item>` tag. The loop extracts the text content of the tags, listed above, using the '`.text`' attribute.

```
Meet the House Republicans Who Democrats Hope Will Defect on the Debt Limit
A long-shot Democratic effort to force a debt-limit increase to the floor hinges on at l
onsidered the likeliest.
https://www.nytimes.com/2023/05/03/us/politics/house-republicans-debt-limit-biden-democr
Wed, 03 May 2023 21:28:02 +0000

Guard Who Fatally Shot Man at a San Francisco Walgreens Won't Be Charged
The San Francisco district attorney, Brooke Jenkins, said the security guard "believed h
https://www.nytimes.com/2023/05/03/us/san-francisco-walgreens-shooting.html
Wed, 03 May 2023 23:04:23 +0000
```

Figure 11: Output of the file scraper.py

After adding a print statement to code, I saved the file as '`scraper.py`' and ran it. Figure 11 above shows the output of the script. The script printed the title, description, link, and date published on the terminal. This demonstrates that the script is working fine, and we can move to the next step, scraping the articles from the `<link>` tag.

However, a new problem arose when moving to the next step. When you visit the NYT article, you are presented with a "paywall" that makes sure you are logged in or create an account to get access to the article, Figure 14. This then made me wonder what the response would be if the script made a request to a specific site.

```
#!/home/gomezpa/Capstone/bin/python

from bs4 import BeautifulSoup
import requests

NYT_US = requests.get('https://www.nytimes.com/2023/05/04/us/politics/tyre-nichols-autopsy.html', timeout=10)

soup = BeautifulSoup(NYT_US.content, 'html.parser')

print(soup)
```

Figure 12: Edited previous code to print response.

```
gomezpa@ubuntu:~/www$ ./test.py
<html><head><title>nytimes.com</title><style>#cmsg{animation: A 1.5s;}@k
eyframes A{0%{opacity:0;}99%{opacity:0;}100%{opacity:1;}}</style></head>
<body style="margin:0"><p id="cmsg">Please enable JS and disable any ad
blocker</p><script data-cfasync="false">var dd={'rt':'c','cid':'AHrlqAAA
AAMAnEhtSaqkgLIAR33r9A==','hsh':'499AE34129FA4E4FABC31582C3075D','t':'bv
','s':17439,'e':'4f6a2cc6da7ec1ba8e55a057507f699330aabc27bb5075e1c515619
f21b4d8da','host':'geo.captcha-delivery.com'}</script><script data-cfasy
nc="false" src="https://ct.captcha-delivery.com/c.js"></script></body></
html>
```

Figure 13: Output of the edited script.

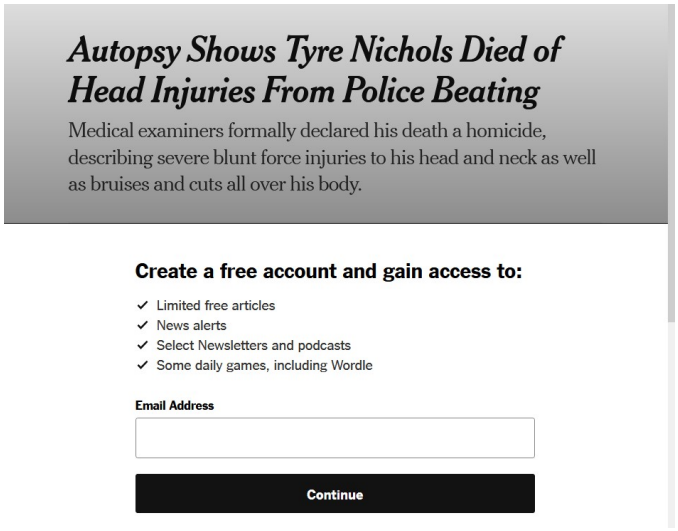


Figure 14: Popup appears on article prompting users to create an account.

I edited 'scraper.py', Figure 12, to print out the response, Figure 13, received after the script made a request to the link https://www.nytimes.com/2023/05/04/us/politics/tyre-nichols-autopsy.html. The results were not what I fully expected. HTML does not show any indication of the pop up or the article. Instead, the output from Figure 13 says to "enable JavaScript and disable any AdBlocker". Upon further research, the result of Figure 13 and Figure 14 was caused by the lack of a User-Agent.

B. What is a User-Agent?

The NYT has a client-side "paywall" in place that blocks users from reading their content once they hit their complementary article limit. I say "paywall", but it requires the user to sign up to continue reading the site. A client-side paywall is one that loads after the initial content from the page is loaded. This paywall is coded in JavaScript which runs in the user's browser, meaning that we could disable JavaScript on our end with the cost of losing some functionality. That is where User-Agents come in.

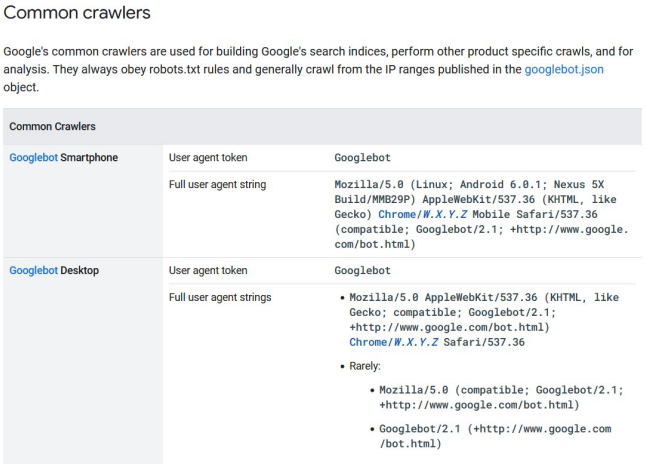
A User-Agent is a short bit of text that describes the Software/Browser (the "Agent") that is making the request to a website[4]. This text is encapsulated inside the request header, and it allows servers and network peers to identify the "Agent". For example, if you make a request to a website with a mobile phone, then the server would be able to identify the "Agent" is a mobile phone.

In the script from Figure 12, we are not passing any parameter that includes what kind User-Agent we "are". Instead, we are just demanding a request and the NYT website treats this as a bot, requiring you to fill out a captcha. Luckily enough, the requests library allows you to include headers as a parameter. However, adding any User-Agent to the header will not be enough because the request does not contain any cookies with credentials to the site. We can overcome this problem with the help of Google's own User-Agent for their Googlebot.

C. Googlebot User-Agent

If you search up a couple of sentences from an article from the NYT, Google's first result will be a link to that article. Sometimes, the Google search result will show part of the article that is behind the paywall. This is possible because the webserver that gaves us the result was using a Googlebot User-Agent to retrieve the information from the NYT.

Google has documented that they use crawlers and fetchers to perform actions for its products either automatically or triggered by user request. A crawler is any program that is used to automatically discover and scan websites following links from one site to another[5]. Google's crawler uses their own specific User-Agent to let webservers know who is making the requests. Figure 15 below shows the common User-Agents Google uses to crawl through websites.



Common crawlers

Google's common crawlers are used for building Google's search indices, perform other product specific crawls, and for analysis. They always obey robots.txt rules and generally crawl from the IP ranges published in the googlebot.json object.

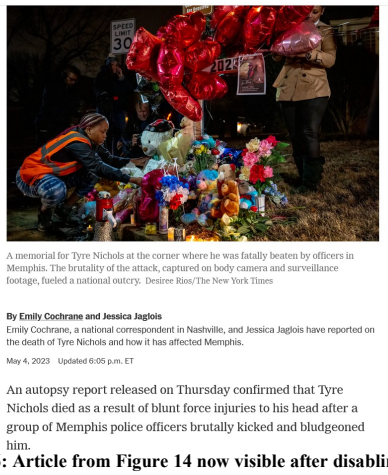
Common Crawlers		
Googlebot Smartphone	User agent token	Googlebot
	Full user agent string	Mozilla/5.0 (Linux; Android 6.0.1; Nexus 5X Build/MMB29P) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/W.X.Y.Z Mobile Safari/537.36 (compatible; Googlebot/2.1; +http://www.google.com/bot.html)
Googlebot Desktop	User agent token	Googlebot

	Full user agent strings	• Mozilla/5.0 AppleWebKit/537.36 (KHTML, like Gecko; compatible; Googlebot/2.1; +http://www.google.com/bot.html) Chrome/W.X.Y.Z Safari/537.36
		• Rarely:
		  • Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.html)
		  • Googlebot/2.1 (+http://www.google.com/bot.html)

Figure 15: Google's common crawlers

D. Using Beautiful Soup to scrape articles

Now that we have identified which User-Agents to use with the request, we must inspect the website of the article and identify which tags we want scrape. Using the same website from Figure 14, I disable JavaScript, shown in Figure 16 below, and inspect the code of the website.



A memorial for Tyre Nichols at the corner where he was fatally beaten by officers in Memphis. The brutality of the attack, captured on body camera and surveillance footage, fueled a national outcry. Desiree Rios/The New York Times

By Emily Cochrane and Jessica Jaglois

Emily Cochrane, a national correspondent in Nashville, and Jessica Jaglois have reported on the death of Tyre Nichols and how it has affected Memphis.

May 4, 2023 Updated 6:05 p.m. ET

An autopsy report released on Thursday confirmed that Tyre Nichols died as a result of blunt force injuries to his head after a group of Memphis police officers brutally kicked and bludgeoned him.

Figure 16: Article from Figure 14 now visible after disabling JavaScript.

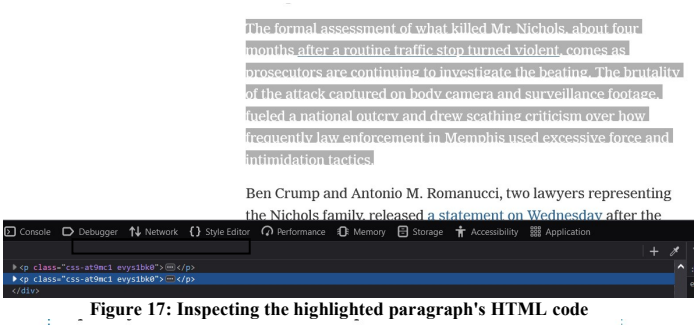


Figure 17: Inspecting the highlighted paragraph's HTML code

Figure 18: Inspecting the next paragraph seen in Figure 17.

I started by highlighting a random paragraph in the article, as shown in Figure 17 above. Upon inspecting it, I noticed that it had a <p> tag with a class name set to '`css-at9mc1 evys1bk0`'. Then I checked the next paragraph that had the same class name, shown in Figure 18 above. With this information, I learned that every article in the NYT uses the same class name attribute for the <p> tag. Now we can edit the 'scraper.py' file to meet our needs.

For this report, I created a new file called 'rss_scraper.py'. It is a heavily modified version of the 'scraper.py' file but with the same concept. First, I created a list of User-Agents and URLs for BeautifulSoup to use as shown in Figure 19 below. The list '`user_agents`' contains four different Googlebot User-Agents.

```
user_agents = [
   "Googlebot/2.1 (+http://www.google.com/bot.html)",
   "Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.html)",
   "Mozilla/5.0 AppleWebKit/537.36 (KHTML, like Gecko; compatible; Googlebot/2.1; +http://www.google.com/bot
.html) Chrome/*.*.*.* Safari/537.36",
   "Mozilla/5.0 (Linux; Android 6.0.1; Nexus 5X Build/MMB29P) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/
*.*.*.* Mobile Safari/537.36 (compatible; Googlebot/2.1; +http://www.google.com/bot.html)"
]

urls = [
    'https://rss.nytimes.com/services/xml/rss/nyt/US.xml',
    'https://rss.nytimes.com/services/xml/rss/nyt/World.xml',
    'https://rss.nytimes.com/services/xml/rss/nyt/NYRegion.xml'
]
```

Figure 19: The list of User-Agents and URLs

```
def get_urls(url):
    for url in urls:
        headers = {'User-Agent': random.choice(user_agents)}
        r = requests.get(url, headers=headers, timeout=10)
        soup = BeautifulSoup(r.content, 'xml')
        items = soup.find_all('item')

        for item in items:
            title = item.title.text
            description = item.description.text
            link = item.link.text
            links.append(link)
            article = get_content(link)
            print(title, "\n", description, "\n", link,"\n----------------
icle,"\n")
```

Figure 20: Screenshot of the function get_urls()

Next, the main `()` function runs the `get_urls()` function, shown in Figure 20 above, which does two things:

a) First it does exactly what 'scraper.py' did but we randomly select one of the User-Agents from the list to send the request.

b) Then we use the link from the `<link>` tag found in the RSS feed and pass it to the function `get_content()`

The `get_content function()`, shown in Figure 21 below, makes a request to the website of the article using another random User-Agent from the list. The response is saved and parsed with BeautifulSoup to find all <p> tags with a class attribute set equal to '`css-at9mc1`'. Then the `for` loop appends the scraped text to the string variable called '`text`' and returned back to the `get_url()` function.

```
def get_content(link):
    headers = {'User-Agent': random.choice(user_agents)}
    r = requests.get(link, headers=headers)
    soup = BeautifulSoup(r.content, 'html.parser')
    items = soup.find_all('p', class_='css-at9mc1')
    text = ""
    for item in items:
        text += item.text + " "
    return text
```

Figure 21: Screenshot of the function get_content().

Once the program was saved, I ran it, and the output can be seen in Figure 22. The script was successful and printed out the full article. However, this scraper was more ambitious and was scraping three NYT RSS feed links. Each RSS feed has around 20 articles, so we are scraping about 60 articles on average. With rss_scraper.py and scraper.py being a success, I started on the process of storing this scraped information in the tables.

```
New York's Hotel-to-Housing Push Seemed Lost. Now a Project Has Begun.
 Developers announced a plan to transform a Hilton hotel near Kennedy International Airport into affordabl
 place through a pandemic-era state program.
 https://www.nytimes.com/2023/05/03/nyregion/nyc-hotel-conversion-housing.html
--------------------------------------
 The halting effort to transform struggling New York City hotels into much-needed affordable housing appea
n airport hotel in Queens is set to be turned into a housing development with more than 300 units. The pro
sday, is the first under a 2021 state program designed to capitalize on the dip in tourism during the pand
and cheaper. Other places, notably California, moved relatively quickly to turn thousands of hotel rooms i
w York, the program has been widely panned. Developers claimed it would not do enough to help them overcom
 million attached to the program was not enough to compensate for development costs on a large scale. But
ncement on Wednesday of the roughly $150 million conversion of the Hilton near Kennedy International Airpo
 and cheaper than building new apartment buildings — can be an effective way of adding more affordable hou
he $150 million will come from the state program, initially created in 2021 by the Housing Our Neighbors w
e bigger," said David Schwartz, a principal for Slate Property Group, one of the developers of the convers
ve ever done." The apartments at the Hilton could be rented out within two years, the developers said, aft
ms, heating systems and other parts of the building. The announcement on Wednesday reflects how officials
 new ways to add more affordable housing. The pandemic also provided a new opportunity to consider whether
 to better use. The issue is urgent in New York City, as housing construction has lagged behind job growth
omelessness remains at record levels. City officials are looking at a number of ways to loosen restriction
ca Katz, the city's chief housing officer, including legalizing basement apartments and converting office
ve been slow to take off because of the rebound in tourism and because hotels were needed to shelter an in
```

Figure 22: Final output of the rss_scraper.py file.

E. Storing data in tables

I started by modifying the existing file, '`scraper.py`' and establishing a connection to the database engine. Figure 23 below shows how to establish the connection. I am using a .env file to hold my environment variables, like username and password.

```
load_dotenv()

mydb = mysql.connector.connect(
    host="localhost",
    user=os.getenv("user"),
    password=os.getenv("password")
)

mycursor = mydb.cursor()
```

Figure 23: Screenshot of the MySQL connector.

Once connected, I modified the for loop to do two things 1) check if the information already exists, 2) insert or update the information into/in the table. This is done with the `mycursor.execute()` function. Figure 24 shows the code below, which checks first and according to entries in the table, it will either insert a new entry to the table or update the table. To do this, the code will check if the title or link scraped was already saved.

```
for item in items:
    title = item.title.text
    description = item.description.text
    link = item.link.text
    pubDate = item.pubDate.text
    pubDate = datetime.strptime(pubDate, '%a, %d %b %Y %H:%M:%S %z').strftime('%Y-%m-%d')

    mycursor.execute("SELECT * FROM nyt_us WHERE title=%s OR link=%s", (title, link))
    result = mycursor.fetchall()
    if not result:
        sql = "INSERT INTO nyt_us (title, description, link, pubDate) VALUES (%s, %s, %s, %s)"
        val = (title, description, link, pubDate)
        mycursor.execute(sql, val)
    else:
        sql = "UPDATE nyt_us SET description=%s, pubDate=%s WHERE title=%s OR link=%s"
        val = (description, pubDate, title, link)
        mycursor.execute(sql, val)
```

Figure 24: Screenshot of the edited for loop.

In the ‘rss_scraper.py’ file, I created a new function called `inputDB` that takes the ‘`title`’, ‘`description`’, ‘`link`’, and ‘`article`’ variables as parameters. This function is called in the `get_urls()` function after calling the `get_content()` function. Figure 25 below shows the code for the `inputDB()`.

```
def inputDB(title, description, link, article):
    mycursor.execute('SELECT * FROM news_article WHERE article_text=%s',(article,))
    result = mycursor.fetchone()
    if result:
        articleID = result[0]
    else:
        sql = "INSERT INTO news_article (article_text) VALUES (%s)"
        val = (article,)
        mycursor.execute(sql, val)
        articleID = mycursor.lastrowid

    mycursor.fetchall()

    mycursor.execute("SELECT * FROM news_items WHERE news_title=%s OR news_link=%s", (title, link))
    result = mycursor.fetchall()
    if not result:
        sql = "INSERT INTO news_items (news_title, news_description, news_link, article_id) VALUES (%s, %s,
%s, %s)"
        val = (title, description, link, articleID)
        mycursor.execute(sql, val)
    else:
        sql = "UPDATE news_items SET news_description=%s WHERE news_title=%s OR news_link=%s"
        val = (description, title, link)
        mycursor.execute(sql, val)

    mycursor.fetchall();
```

Figure 25: Screenshot of the inputDB function.

The code will first check if the ‘news_article’ table already has the article saved. If not, then it is inserted, but if it already exists, then the ‘`article_id`’ is stored. Afterwards, the function will do the same thing that Figure 24 demonstrates. The only difference is that if the entry does not exist, then it will save the articleID from the news_article table and save it as the ‘`article_id`’ for the ‘`news_items`’ table.

Once I completed writing the script for both files, I ran both scripts and sure enough, I got new entries added to the tables. Figures 25 and 26 below show the entries added to the ‘nyt_us’ and ‘news_items’ tables.

```
------------------------------------------------------------------+-----------+
| id   | title                                                                                      | description

                                                                                | link
                                                               | pubDate    |
+------+-------------------------------------------------------------------------------------------+------------------------------
---------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------+-------------------------------------------------
------------------------------------------------------------------+-----------+
|    1 | Friends and Family Mourn Lisa Marie Presley at Graceland                                   | The service on Sunday, open to the p
ublic, was followed by a procession for visitors.
                                                                                | https://www.nytimes.com/2023/01/22/us/lisa-marie-presley-memorial.h
tml                                                            | 2023-01-22 |
|    2 | When Students Change Gender Identity, and Parents Don’t Know                               | Educators are facing wrenching new t
ensions over whether they should tell parents when students socially transition at school.
                                                                                | https://www.nytimes.com/2023/01/22/us/gender-identity-students-pare
nts.html                                                       | 2023-01-22 |
|    3 | Women’s March Holds Nationwide Rallies on 50th Anniversary of Roe                          | The annual march, which was started
in 2017 as a reaction to the election of President Donald J. Trump, this year focused on abortion rights.
                                                                                | https://www.nytimes.com/2023/01/22/us/politics/womens-march-roe-wad
e.html                                                         | 2023-01-23 |
|    4 | Florida Gives Reasons for Rejecting A.P. African American Studies Class                    | The state’s Department of Education
cites examples of what it calls “the woke indoctrination” of students.
                                                                                | https://www.nytimes.com/2023/01/21/us/florida-ap-african-american-s
tudies.html                                                    | 2023-01-22 |
```

Figure 26: Entries added to the 'nyt_us' table

```
mysql> select * from news_items
    -> ;
+---------+------------------------------------------------------------------------------------------+----------------------------
---------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------+-------------------------------------
-----------------------------------------------------------------------------------------------+-----------+
| news_id | news_title                                                                               | news_description

                                                                                           | news_link
                                                                                               | article_id |
+---------+------------------------------------------------------------------------------------------+----------------------------
---------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------+-------------------------------------
-----------------------------------------------------------------------------------------------+-----------+
|       1 | After School Shooting, Nashville Grieves and Ponders Its Divisions                       | For decades, Nashville has pro
spered while finding common ground between urban and rural, left and right, state and city. In a partisan era, that’s becoming much harder.
                                                                                           | https://www.nytimes.com/2023/03/31/us/n
ashville-shooting-guns-transgender-rights.html                                                 |         1 |
|       2 | Nebraska’s Fight Over Transgender Care Turns Personal and Snarls Lawmaking               | One state senator has been fil
ibustering for weeks, trying to block a bill that would ban transition-related medical treatment for young people.
                                                                                           | https://www.nytimes.com/2023/03/30/us/n
ebraska-filibuster-transgender-bills.html                                                      |         2 |
|       3 | G.O.P. Lawmakers Override Kentucky Governor’s Veto on Anti-Trans Law                     | The Kentucky measure bans acce
ss to gender-transition care for young people, and West Virginia’s governor signed a similar bill on Wednesday. Passage of bans also appears immin
ent in Idaho and Missouri.                                                                 | https://www.nytimes.com/2023/03/29/us/k
entucky-anti-transgender-bill.html                                                             |         3 |
|       4 | U.S. Border Policies Have Created a Volatile Logjam in Mexico                            | As the United States has crack
ed down on border entries, Mexico is bearing the burden of housing and feeding tens of thousands of desperate migrants.
                                                                                           | https://www.nytimes.com/2023/03/28/us/m
exico-border-migrants-shelters.html                                                            |         4 |
```

Figure 27: Entries added to the 'news_items' table.

After successfully verifying the scripts are working, I created a cron job that runs both scripts every four hours, guaranteeing we scrape the most recent articles.

F. Creating the HTML card elements

Now that we have set up the scripts, we can finally move to creating the website that holds the article’s content. For this project, a basic web application that points to different links. One of the routes that we will create is called ‘nyt.ejs’. The file will present to users a title, description, link, and the date published, in a card format. To create the card format, I will be using HTML, EJS, and CSS. Figure 28 below shows the code for the card that is in the file ‘nyt.ejs’.

```
    <div class="container">
        <% news.forEach(function(item) { %>
            <div class="card">
                <div class="card-body">
                    <h4 class="card-title"><%= item.title %></h4>
                    <hr>
                    <p class="card-text"><%= item.description %>
                    <hr>
                    <p class="card-text">Published on: <%= item.pubDate %></p>
                    <hr>

                    <% if (username) { %>
                        <a href="<%= item.link %>" class="btn" style="float: left">Read More</a>
                        <form action="/favorite" method="POST">
                            <input type="hidden" name="article_id" value="<%= item.id %>">
                            <button type="submit" id="favorite" class="btn" style="float: right">Add to favorite</button>
                        </form>
                    <% } else { %>
                        <a href="<%= item.link %>" class="btn" style="float: left">Read More</a>
                        <button type="submit" id="favorite" class="btn" style="float: right">Login to favorite</button>
                    <% } %>
                </div>
            </div>
        <% }); %>
    </div>
</div>
```

Figure 28: Screenshot of the 'nyt.ejs' file.

The code above uses a for loop ‘`news.forEach(function(item))`’ to iterate over each news item in the ‘`news`’ array. The loop generates an HTML card element with the information mentioned above displayed inside of them. If the user is logged in, the card will display a button to add the article to your favorites. If they are

not logged in, then the button will say to log in first. When clicked it sends a POST request to the '`/favorite`' route with the '`article_id`' hidden input field. The '`item`' object contains the properties for the news item such as '`title`', '`description`', '`pubDate`', and '`link`'.

```
app.get('/nyt', function(req, res){
    console.log("NYT")
    db.getData(function(data){
        db.getFullData(function(fullArticle){
            res.render('nyt', {username: req.session.username, news:data, fullArticle: fullArticle});
        });
    });
});
```

Figure 29: Screenshot of the edited app.js file.

In the `app.js` file, I added a new Express.js route handler which defines the route '`/nyt`'. When a GET request is made to the new endpoint, it retrieves data from the database using two different functions, `db.getData()` and `db.getFullData()`. Figure 29 above shows the changes added to the `app.js` file.

```
const mysql = require('mysql')
require('dotenv').config()

const testConnection = mysql.createConnection({
  host: '127.0.0.1',
  port: 3306,
  user: process.env.user,
  password: process.env.password,
  database: 'test'
});
testConnection.connect((err) => {
  if (err) throw err;
  console.log("Connected to test database");
});
exports.getData = (function(callback) {
  testConnection.query('SELECT * FROM nyt_us', (err, results) => {
    if (err) throw err;
    callback(results);
  });
});
```

Figure 30: Code for the db.js file

After saving the changes for the app.js file, I created the new db.js file. Figure 30 above shows the code for a Node.js module the exports a function called `getData()`. The code initializes a connection to the '`test`' database. Then the function `getData()` is defined as an exported method, which takes a '`callback`' parameter. Inside the function, the '`testConnection.query()`' method is used to execute the SQL query "`SELECT * FROM nyt_us`". The result of the query will then be processed as needed.

So, the `db.js` file makes a query to the '`test`' database and exports the results to the `app.js` file. The `app.js` file retrieves and displays this data when a GET request is sent to the '`/nyt`'. The 'news' property is set equal to the data retrieved from the db.getData() function, which is an array. Finally, in the '`nyt.ejs`' file, uses a '`for`' loop to iterate over each news item in the '`news`' array and for each item, an HTML card element is generated.

G. Creating a "favorite" feature

Now that I finished writing the code for the HTML card elements, I started working on the "favorite article" feature. I previously demonstrated how to create the 'favorites' table and in Figure 28, I wrote the code to create a favorite button. In the code, I used EJS to create an '`if`' statement to check if the user is logged in or not. If they are logged in, then the user will be able to click on the button. This will send a POST request the '`/favorites`' route and pass the variable '`article_id`'. This variable is the ID of the article stored in the '`nyt_us`' database.

Next, I created another Express.js route handler that defines a route for the '`/favorites`' endpoint. Figure 31 below shows the new route I created in the `app.js` file.

```
app.post('/favorite', (req, res) => {
    const username = req.session.username;
    const article_id = req.body.article_id;
    const user_id = req.session.user_id;
    console.log(username, user_id, article_id);

    const sqlCheck = "SELECT * FROM favorites WHERE user_id = ? AND nyt_us_id = ?";
    const values = [user_id, article_id];
    user.query(sqlCheck, values, function(err, result) {
        if (err) throw err;
        if (result.length > 0) {
            console.log("Article is already in favorites!")
            res.send('<script>alert("Article already in favorites!"); window.history.back();</script>')
        } else {
            const sqlInsert = "INSERT INTO favorites(user_id, nyt_us_id) VALUES (?, ?)";
            const values = [user_id, article_id];
            user.query(sqlInsert, values, function(err, result) {
                if (err) throw err;
                console.log("Article added to favorites!");
                res.redirect('/nyt?favorite_added=true');
            });
        }
    });
});
```

Figure 31: Code for the new '/favorite' endpoint.

Here is a brief overview of what is happening in the code:

- First, the handler extracts the username, article_id, and user_id from the request object. This request object will be made with the login feature.
- Next, we establish connection to the database and query the '`favorites`' table to check if the article is already in the user's favorites list. This is done by preparing the SQL statement and saving it to the `sqlCheck` variable. I pass the values `user_id` and `article_id` to the query statement.
- If the article already exists, the handler will send an alert to the user informing them that the article is already in the favorites table.
- Finally, if the `article_id` and `user_id` do not exist, the handler will insert the article into the favorites table using the SQL statement in the `sqlInsert` variable. It will then redirect them to the '/nyt' endpoint with a query parameter, indicating that their article has been added to their favorites list.

Now, if a user was logged in, they can favorite any article they desire in the '`/nyt`' route. The next step is to create a route that allows the user to view their favorite articles. In this case, we can use the same EJS '`for`' loop shown in Figure 28 to create HTML card elements. This time, however, the '`for`' loop will only display articles from the '`favorites`' table.

```
<h1> Favorite Page </h1>
<div class="container">
    <% favorite.forEach(function(item, index) { %>
        <div class="card">
            <div class="card-body">
                <h4 class="card-title"><%= item.title %></h4>
                <hr>
                <p class="card-text"><%= item.description %></p>
                <hr>
                <p class="card-text">Published on: <%= item.pubDate %></p>
                <hr>
                <a href="<%= item.link %>" class="btn">Read More</a>
            </div>
        </div>
    <% }); %>
</div>
```

Figure 32: EJS code for the '/favorite' route.

First, I created a new file, '`favoritePage.ejs`', which will display the card element to the user. Figure 32 above shows a screenshot of the new '`for`' loop. This time, the code loops through the array of favorite articles passed to the template via the "favorite" variable. It will generate the same card format generated in Figure 28.

Next, I edited the `app.js` file to include a new handler for the '`/favoritePage`' endpoint. The code for the new handler can be seen in the figure below.

```
app.get('/favoritePage', (req, res) => {
    console.log("Favorites")
    db.getFavorite(function(favorite) {
        res.render('favoritePage', {username:req.session.username, favorite:favorite});
    });
});
```

Figure 33: Screenshot of the new handler.

The snippet above listens for GET requests made to the '`/favoritePage`', which will call the function db.getFavorite. This function is used to query the database and check if there are any entries in the user's favorites. The handler takes two arguments, the name of the file to render, '`favoritePage`', and an object that contains variables to be passed to the file.

Finally, the db.js file was edited to fetch the list of favorite articles for the current user. In the screenshot below, the function uses the MySQL module's 'query' method to execute a SQL JOIN query.

```
exports.getFavorite = (function(callback) {
    testConnection.query('SELECT nyt_us.id, nyt_us.title, nyt_us.description, nyt_us.link, nyt_us.pubDate FROM nyt_us INNER JOIN
        if (err) throw err;
        callback(results);
    });
});
```

Figure 34: Code for the query function.

The SQL statement used to make the query is the following:

```
SELECT news_items.news_title,
news_items.news_description,
news_items.news_link,
news_article.article_text
FROM news_items
JOIN news_article ON
news_items.article_id =
news_article.article_id
GROUP BY news_items.news_title,
news_items.news_description,
news_items.news_link,
news_article.article_text;
```

The SQL statement above performs a JOIN operation between the two tables, '`news_items`' and '`news_article`', using their respective article ID columns. The results from the query will be returned to the `app.js` file which will render the new '`favoritePage`' route.

H. Creating a register and login feature

With the new favorite feature implemented, I created a login and register form that works. They are modal forms that were created using HTML and CSS. Figure 35 below shows the code for the register form.

```
<span onclick="document.getElementById('id01').style.display='none'" class="signup-close animate" title="Close Menu">&times;</span>
<form class="modal-content animate" action="/register" method="POST">
  <div class="signup-container">
    <h1>Sign Up</h1>
    <p>Please fill in this form to create an account.</p>
    <hr>
    <label for="username"><b>Username</b></label>
    <input type="text" placeholder="Enter username" name="username" class="signup-input" required>

    <label for="email"><b>Email</b></label>
    <input type="email" placeholder="Enter Email" name="email" class="signup-input" required>

    <label for="passwd"><b>Password</b></label>
    <input type="password" placeholder="Enter Password" name="password" class="signup-passwd" required>

    <label for="passwd-repeat"><b>Repeat Password</b></label>
    <input type="password" placeholder="Repeat Password" name="passwordRepeat" required>

    <p>By creating an account you agree to the Terms & Privacy</p>

    <div class="clearfix">
      <button type="button" onclick="document.getElementById('id01').style.display='none'" class="signupcancelbtn signupbtn">Cancel</button>
      <button type="submit" class="signupbtn">Sign Up</button>
    </div>
  </div>
</form>
```

Figure 35: HTML/EJS code for a register form

The form requests the user to input a username, an email address, and a password they want to use. When the form is submitted, it is sent to the '`/register`' endpoint as a POST request.

```
<span onclick="document.getElementById('id02').style.display='none'" class="login-close animate" title="Close Menu">&times;</span>
<form class="modal-content animate" action="/login" method="POST">

  <div class="imgcontainer">
    <img src="/images/profile.png" alt="Avatar" class="avatar">
  </div>

  <div class="login-container">
    <label for="username"><b>Email</b></label>
    <input type="text" placeholder="Enter username" name="username" required>

    <label for="passwd"><b>Password</b></label>
    <input type="password" placeholder="Enter password" name="password" required>

    <button type="submit" class="loginbtn">Login</button>

    <label>
      <input type="checkbox" checked="checked" name="remember"> Remember me
    </label>
  </div>

  <div class="login-container" style="background-color:#f1f1f1">
    <button type="button" onclick="document.getElementById('id02').style.display='none'" class="logincancelbtn">Cancel</button>
    <span class="passwd">Forgot your password?</span>
  </div>

</form>
```

Figure 36: HTML/EJS code for the login form

The code for the login form is just like the register form. The screenshot above, Figure 36, shows the code for the login form. This form asks the user for a username and password. Like the register form, when the login form is submitted, the form is sent to the '/login' endpoint as a POST request.

This code should be enough to create a basic login and register form. You can add CSS to make the forms look good, which is what I did. Next, I will demonstrate how to make the forms fully functional with SQL queries.

I. Dynamic queries vs parameterized queries

This website will be accessed by anybody who has the link. If an attacker wants to do an SQL injection attack, then they must find vulnerable user inputs within my website. In this case, the login and registration forms on the site are susceptible to malicious SQL queries. According to OWASP, SQL injection flaws are created when SQL statements are constructed dynamically by concatenating strings[6]. In these strings, the attacker can insert malicious SQL code to view sensitive information or deleting records.

```
String query = "SELECT account_balance FROM user_data WHERE user_name = "
             + request.getParameter("customerName");
try {
    Statement statement = connection.createStatement( ... );
    ResultSet results = statement.executeQuery( query );
}
...
```

Figure 37: Example of a SQL injection flaw.

Figure 37 above shows an example of using SQL statements and concatenating strings. The parameter "customerName" is appended to the query which makes it flawed. This code, written in Java, is unsafe and would allow an attacker to inject code into the SQL query. For example, the attacker could enter "john; SHOW TABLES;" in the "customerName" parameter. This would allow the attacker to identify what tables exist in the database. With that information, they have unauthorized access to the database.

Instead of using dynamic queries, OWASP states that using [6] "prepared statements with variable binding (aka parameterized queries) is how all developers should first be taught how to write database queries." Parameterized queries are effective in preventing SQL injection attacks because the SQL code is separate from the data values. The database engine, MySQL, treats the bound values as data, rather than as part of the SQL statement.

```
// This should REALLY be validated too
String custname = request.getParameter("customerName");
// Perform input validation to detect attacks
String query = "SELECT account_balance FROM user_data WHERE user_name = ? ";
PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, custname);
ResultSet results = pstmt.executeQuery( );
```

Figure 38: Example of a parameterized query.

Figure 38 above uses a PreparedStatement, which is Java's way of implementing a parameterized query. Unlike Figure 33, if the attacker were to input "john; SHOW TABLES;" in the "customerName" parameter, the parameterized query would not be vulnerable. Instead, the query would look for a "customerName" that literally matches "john; SHOW TABLES;".

J. Implementing parameterized queries to prevent SQL injection

Currently, the web app does not have any way of submitting SQL queries to the database when the user enters their information in either form. To fix this, I edited the app.js file. I created two app.post() functions that will route the HTTP POST requests to the specified path. The first function will route the HTTP POST requests to "/register".

```
app.post('/register', function(req, res) {
    var username = req.body.username;
    var email = req.body.email;
    var sqlCheck = "SELECT * FROM users WHERE username = ? OR email = ?";
    var sqlInsert = "INSERT INTO users (username, email, password) VALUES (?,?,?)";

    if (req.body.password == req.body.passwordRepeat) {
        var valueCheck = [username, email];
        var hash = bcrypt.hashSync(req.body.password, saltRounds)
        var valueInsert = [username, email, hash];
        userRegister.query(sqlCheck, valueCheck, function(err, result) {
            if (err) throw err;
            if (result.length > 0) {
                console.log("Username or email already exists!");
                res.send('<script>alert("Username or email already exists! Please choose a diffrent username or email."); window.history.back();</script
>');
            } else {
                userRegister.query(sqlInsert, valueInsert, function (err, result){
                    if(err) throw err;
                    console.log("User registered successfully!");
                    res.redirect('/');
                });
            }
        });
    } else {
        console.log("Passwords do not match!")
        res.send('<script>alert("Passwords do not match! Please try again."); window.history.back();</script>');
    }
    console.log(username, email, hash);
});
```

Figure 39: Code for "/register" route.

Figure 39 above shows the code for the first function that handles user registration. First, the code extracts the values for 'username', 'emai', and 'password' from the register form. Next, I define two SQL statements 'sqlCheck' and 'sqlInsert'. I am using parameterized queries and substituting user-supplied values in the SQL statements instead of concatenating the values directly into the statements. The code will then check if the passwords match. If they match, a hash of the value 'password' is generated using the bcrypt library. After generating the hash, the code creates two arrays, 'valueCheck' and 'valueInsert'. These arrays will be used to substitute the '?' placeholders in the SQL statements and passed as parameters to the 'userRegister' function. The first query function takes the 'sqlCheck' and 'valueCheck' values as parameters to check if there is a user with the same email address or username. If it does not exist in the table, the second query is sent. The second query takes the 'sqlInsert' and 'valueInsert' values as parameters. The second query inserts the user's email address, username, and hashed password into the database, creating a new user.

```
app.post('/login', function(req, res) {
    var username = req.body.username;
    var password = req.body.password;
    var sqlCheck = "SELECT password FROM users WHERE username = ?";
    var valueCheck = [username];

    if (username && password) {
        userRegister.query(sqlCheck, valueCheck, function(err, result) {
            if (err) throw err;
            if(result.length > 0) {
                var savedHash = result[0].password;
                bcrypt.compare(password, savedHash, function(err, match) {
                    if (err) throw err;
                    if (match) {
                        req.session.loggedin = true;
                        req.session.username = username;
                        res.redirect('/');
                    } else {
                        console.log("Incorrect password!");
                        res.send('<script>alert("Incorrect password!"); window.history.back();</script>');
                    }
                });
            } else {
                console.log("User not found!");
                res.send('<script>alert("User not found!"); window.history.back();</script>');
            }
        });
    } else {
        console.log("Username or password not provided!");
    }
});
```

Figure 40: Code for the "/login" route.

Figure 40 above shows the code that handles user login functionality. The code extracts the values for 'username' and 'password' from the login form. Next, I created another parameterized query, 'SELECT password FROM users WHERE username = BINARY ?', which will select the hashed password from the 'users' table where the username matches the provided username from the user. This query is executed and if it returns a result, the code will use the 'bcrypt.compare()' function. This function will compare the provided password to the hashed password stored in the database. If they match, the user will be logged in.

Additionally, the username is stored in an object, '`req.session.username`'. This object will be used to check if the user has logged in, allowing the user to save the articles they want.

IV. Discussion

A. Results

After adding the features to the web application, I focused on styling the website. I followed tutorials from W3Schools to make everything that is seen on the website. I added a navigation bar that has tabs to a home page, about page, and news pages, Figure 41. The navigation bar also has the "Login" and "Sign Up" buttons, Figure 44, that get replaced with the person's username when they are logged in. When hovering over the username, a dropdown menu appears with a "Logout" and "Your favorites" button, Figure 45. Clicking the "Your favorites" button will load the Favorite Page, Figure 46.

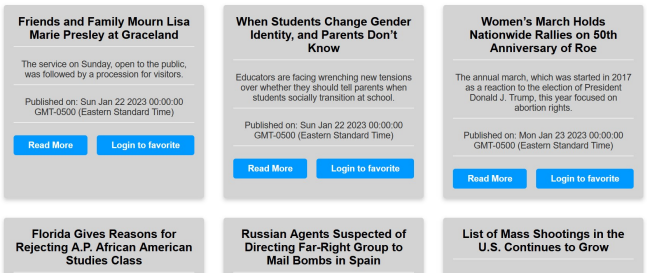


Figure 41: Page for the scraped NYT articles.

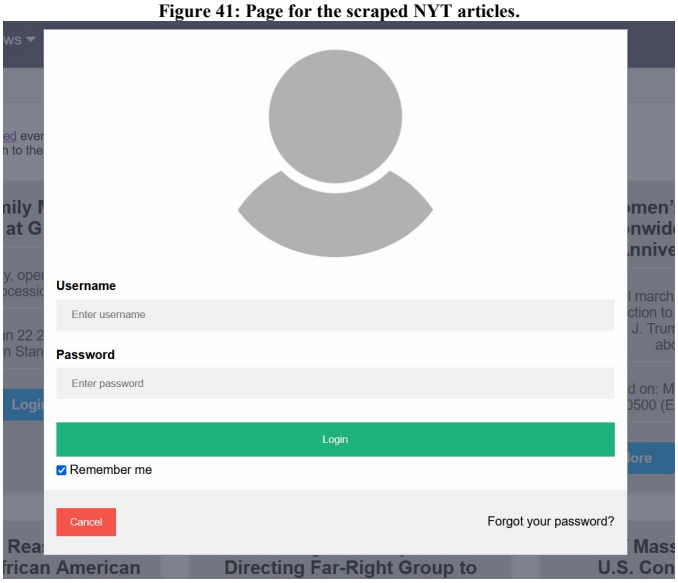


Figure 42: The modal login form.

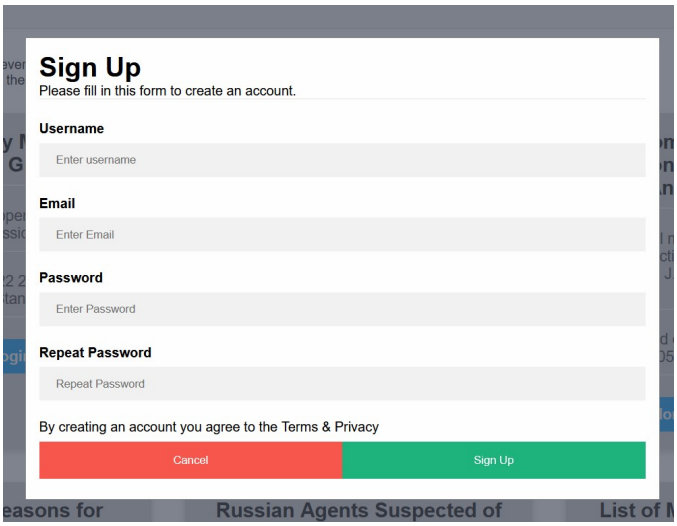


Figure 43: The modal register form.

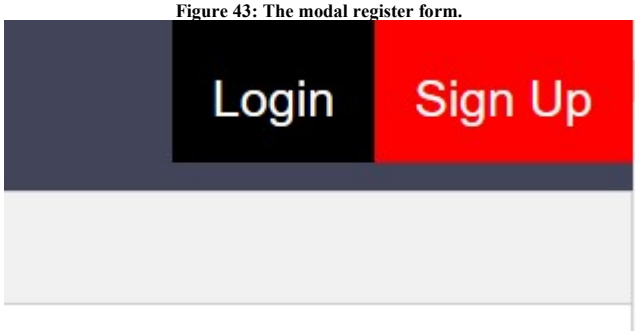


Figure 44: "Login" and "Sign Up" buttons found on the top-right of the page.

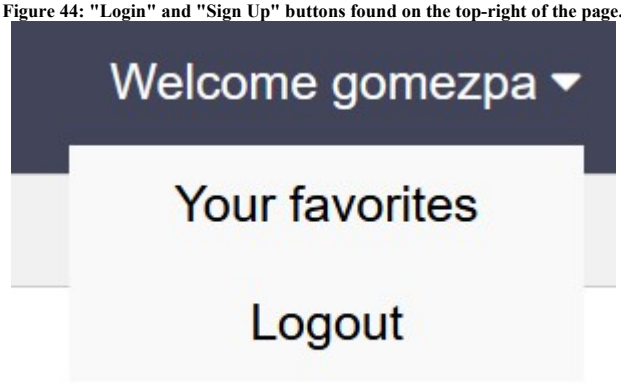


Figure 45: The buttons get replaced with the user's username and a dropdown menu.

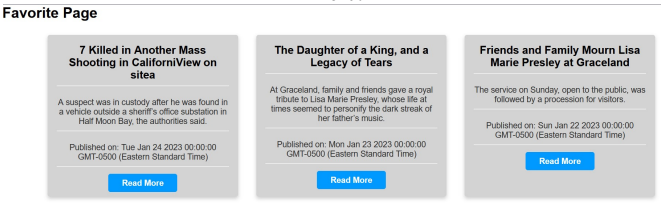


Figure 46: The favorite page rendered.

V. Conclusion

Staying up to date with the news and having accessibility to different news resources is vital. We can be informed on decisions made by people in power or be informed on the current events that impact our lives. With so much information and resources available online, it can be tricky to stay up to

date. In this report, I went over how to scrape RSS feeds from the NYT. The scraped data was aggregated to a database which we later use to access the contents inside. This information is displayed on a website in a HTML card format. Additionally, I worked on preventing SQL injection attacks. In the report, I explained that SQL flaws are introduced when a developer creates dynamic database queries constructed with string concatenation which contains user input. Instead, I used parameterized queries, which takes user input and treats it like data rather than executable code.

VI. Bibliography

[1] "Beautiful Soup Documentation — Beautiful Soup 4.4.0 documentation." Beautiful Soup Documentation. [Online]. Available: https://beautiful-soup-4.readthedocs.io/en/latest/. [Accessed: Apr. 24, 2023].

[2] "How Do RSS Feeds Work? | RSS.com," *RSS.com*, Jun. 25, 2018. [Online]. Available: https://rss.com/blog/how-do-rss-feeds-work/ [Accessed Apr 24, 2023]

[3] "Beautiful Soup Documentation — Beautiful Soup 4.4.0 documentation," *beautiful-soup-4.readthedocs.io*. [Online] Available: https://beautiful-soup-4.readthedocs.io/en/latest/#making-the-soup [Accessed Apr 24, 2023]

[4] "What is a user agent?," *WhatIsMyBrowser.com*, Feb. 19, 2022. [Online]. Available: https://www.whatismybrowser.com/detect/what-is-my-user-agent/faq/what-is-a-user-agent [Accessed Apr. 24, 2023]

[5] "Google Crawler (User Agent) Overview | Google Search Central | Documentation," *Google Developers*. [Online]. Available: https://developers.google.com/search/docs/crawling-indexing/overview-google-crawlers [Accessed Apr. 24, 2023]

[6] OWASP, "SQL Injection Prevention · OWASP Cheat Sheet Series," Owasp.org, 2021. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html [Accessed Apr. 25, 2023]