

# Website Cheating Bot Detector

Final Handoff Report

Oregon State University

## **Introduction**

In today's modern age of online classes, academic dishonesty is becoming a bigger issue as more and more answers to tests and assignments are getting posted online. And with no in-person classes to hold students accountable, more and more students are resulting to cheating in class.

This document describes the components that make up our bot which, through the use of a database, scours the internet, searching for and reporting websites containing answers to course material.

From the EECS Project Description:

The project is to create a bot that would scour the internet to detect websites that are posting exams, quizzes and assignments with answers for Oregon State University courses. By detecting these sites that have posted the information to assist students in academic dishonesty cases, it would allow us to notify the appropriate website hosts that they will need to remove the information from their websites, and it will allow Oregon State University to notify the faculty member that their course has been compromised.

## **User Perspective**

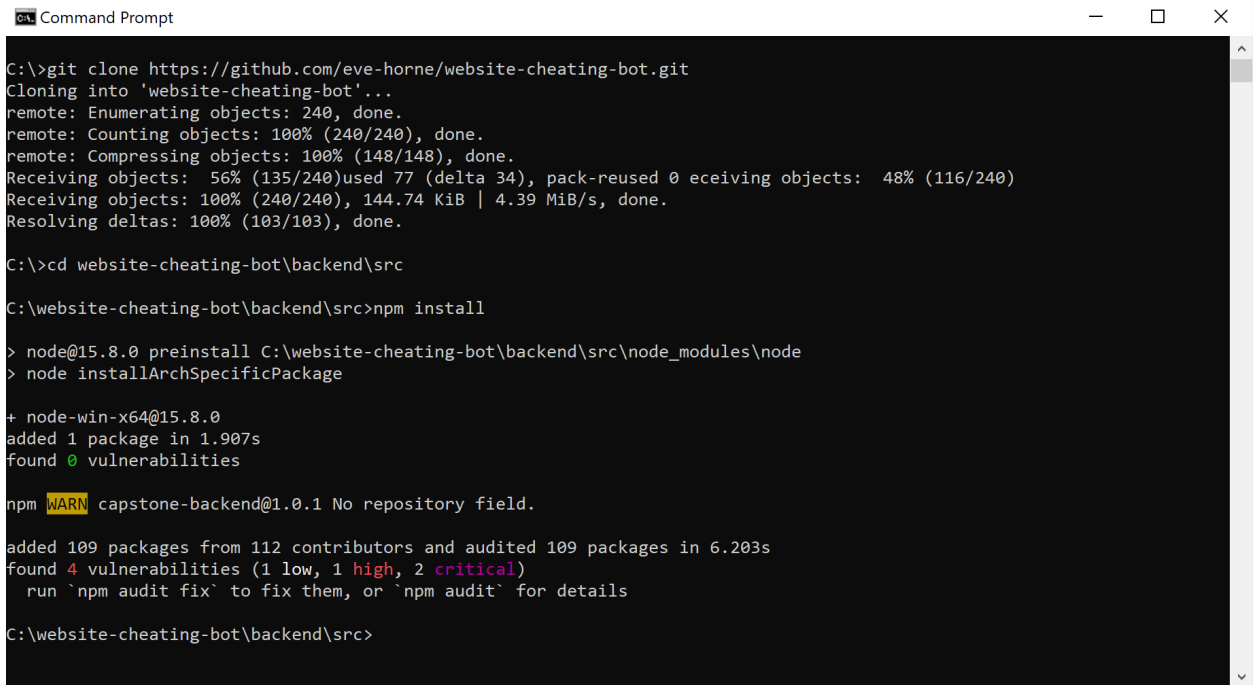
From the end-user's perspective, our project is a two-part website interface. The first part of the website is the query page where users can type out the course material that they want the bot to search for. And the second part of the website is the dashboard page where users can see the results of their search, which are in the form of website URLs. We hope that by providing this two-part interface, the end-user is able to easily understand and run our bot, as well as see the results of their search.

## **Usage Instructions**

### **a. Server Setup**

1. To set up the server, open up a command terminal. On Windows 10 you can click on the search bar, type "cmd", then press enter. If using a Mac, you can click on the search bar and type "terminal" and then press enter.
2. Next, type "git clone <https://github.com/eve-horne/website-cheating-bot.git>" and hit enter.

3. After this, type “cd website-cheating-bot\backend\src” and hit enter.
4. Next type “npm install”, hit enter, and wait a few seconds. This command downloads the files needed to run the server. Here’s what the above commands should look like:



```
Command Prompt
C:\>git clone https://github.com/eve-horne/website-cheating-bot.git
Cloning into 'website-cheating-bot'...
remote: Enumerating objects: 240, done.
remote: Counting objects: 100% (240/240), done.
remote: Compressing objects: 100% (148/148), done.
Receiving objects: 56% (135/240)used 77 (delta 34), pack-reused 0 eceiving objects: 48% (116/240)
Receiving objects: 100% (240/240), 144.74 KiB | 4.39 MiB/s, done.
Resolving deltas: 100% (103/103), done.

C:\>cd website-cheating-bot\backend\src

C:\website-cheating-bot\backend\src>npm install

> node@15.8.0 preinstall C:\website-cheating-bot\backend\src\node_modules\node
> node installArchSpecificPackage

+ node-win-x64@15.8.0
added 1 package in 1.907s
found 0 vulnerabilities

npm WARN capstone-backend@1.0.1 No repository field.

added 109 packages from 112 contributors and audited 109 packages in 6.203s
found 4 vulnerabilities (1 low, 1 high, 2 critical)
  run `npm audit fix` to fix them, or `npm audit` for details

C:\website-cheating-bot\backend\src>
```

5. After that, you need to follow your hosts process for generating SSL certificates. Once you have your key and certificate files, replace the placeholder text in the app.js file with the location of your files.
6. After that, move the frontend folder to whatever static html hosting location your host has. From here you will need to replace all references of “hax.services” with the domain name of your server.
7. Start your hosting server (apache or python3 http), then start the backend api with “node app.js” in the backend folder.

### **b. Generating a Query**

To generate a query, go to the project website at the domain you've installed this on and fill out the Course ID, Questions, and Email. Alternatively a CSV file can be uploaded instead of typing out the questions.

Once the "Submit" button is clicked, the unique query code will be generated and displayed. It is very important that this code is saved as it is needed to view the results of the scraper. This code will also be emailed to the email specified in the Email field.

### **c. Seeing Results**

To view the results, head over to the Dashboard page using the navigation bar, and enter either the course code, or the unique query code supplied earlier. Once the "Submit" button is clicked, the results of the query will be displayed.

## **Software and System Description**

The software and systems of our project work together in a few different ways.

### **a. Query Website Interface to Server**

First, our website interface uses the VueJS framework, and sends all of the user-inputted data to our backend nodeJS server. The way this works is our query website interface uses an HTML form for the user-inputted data. This query interface also has a corresponding JavaScript file that then uses VueJS and an XMLHttpRequest to receive the user-inputted data, and send it to the nodeJS server.

### **b. Server to Database and Bot**

Once the nodeJS server is up and running, it will connect to the MongoDB database and sit on the network, waiting for a network request. And once the network request containing the user-inputted data is received, it starts interacting with the database.

The way this works is once the server is running, it creates a MongoDB client using the login information for the database. And whenever a network request is received, the server calls our scrape() function, which uses the child\_process library to execute the python scraper itself, for each piece of user-inputted data received. And as well as calling this scrape() function, the server also emails the user their specific query token. Then, once the results of the scraper are received, the server stores those results in the database, using the MongoDB client created earlier. And finally, once all of the scraper instances have returned, the server then marks that query as done in the server.

### **c. Bot to Server**

The interaction from bot to server is much simpler than server to bot. The way it works is whenever the scraper finishes its scrape, it simply prints out the results. But since the bot instances were created using the `child_process` library, the results aren't actually printed out, but instead returned to the server.

### **d. Dashboard Website Interface to Server and Database**

Similarly to the query interface, the dashboard interface uses VueJS and a corresponding JavaScript file to connect to the server and database. The way it works is it again uses an XMLHttpRequest to send a network request to the server, passing the user's unique query ID to the `getQuery()` function. This function then uses that unique query ID to receive both the initial user-inputted data and the results of the scraper bot. This data is then displayed on the dashboard interface.

## **Developmental Tools Used**

We used a variety of development tools through the production of this project:

### **a. Website Interface**

- i. VueJS
- ii. HTML and CSS
- iii. JavaScript
- iv. Bootstrap

### **b. Server**

- i. NodeJS
- ii. Express
- iii. BodyParser
- iv. PHP for initial unit testing
- v. Apache
- vi. OpenSSL

### **c. Database**

- i. MongoDB

### **d. Bot**

- i. Python
- ii. Googlesearch
- iii. BeautifulSoup

### **e. DevOps**

- i. Bash for environment initialization
- ii. DigitalOcean

## Team Member Responsibilities

### a. Keaton Perry

- i. Implemented the initial website.
  - 1. Used HTML, JavaScript, and CSS to create the website. HTML form was used for the user-inputted data, which was processed by a PHP file.
  - 2. Later updated this to process the data through the nodeJS server instead of a PHP file.
- ii. Implemented the initial server.
  - 1. Used nodeJS, express, and handlebars for the server.
  - 2. Used an app.post network request function to receive the data from the website.
  - 3. Used the child\_process library to execute the scraper from the nodeJS server.
  - 4. Later updated this server to no longer rely on handlebars.
- iii. Documented changes to the project throughout the code freeze and final progress report.

### b. Eve Horne

- i. Implemented final Vue.JS logic and front end design
  - 1. Used HTML, JavaScript (vueJS), and CSS to create the final front end view
- ii. Implemented final backend logic
  - 1. Used nodeJS, express, mongo client, spawn
  - 2. Implemented endpoints for querying and recalling jobs
  - 3. Implemented email notification system with nodemailer
- iii. Deployed a live playground version of the application, including with letsencrypt https support on a digitalocean droplet.

## Weekly Task Breakdown

### a. Keaton Perry

- i. Registered for the EECS Project Showcase.
- ii. Updated our nodeJS server to no longer use handlebars.
- iii. Researched backend hosting solutions offered through the university.
- iv. Implemented error checking into the server code to catch any errors thrown by the python scraper.

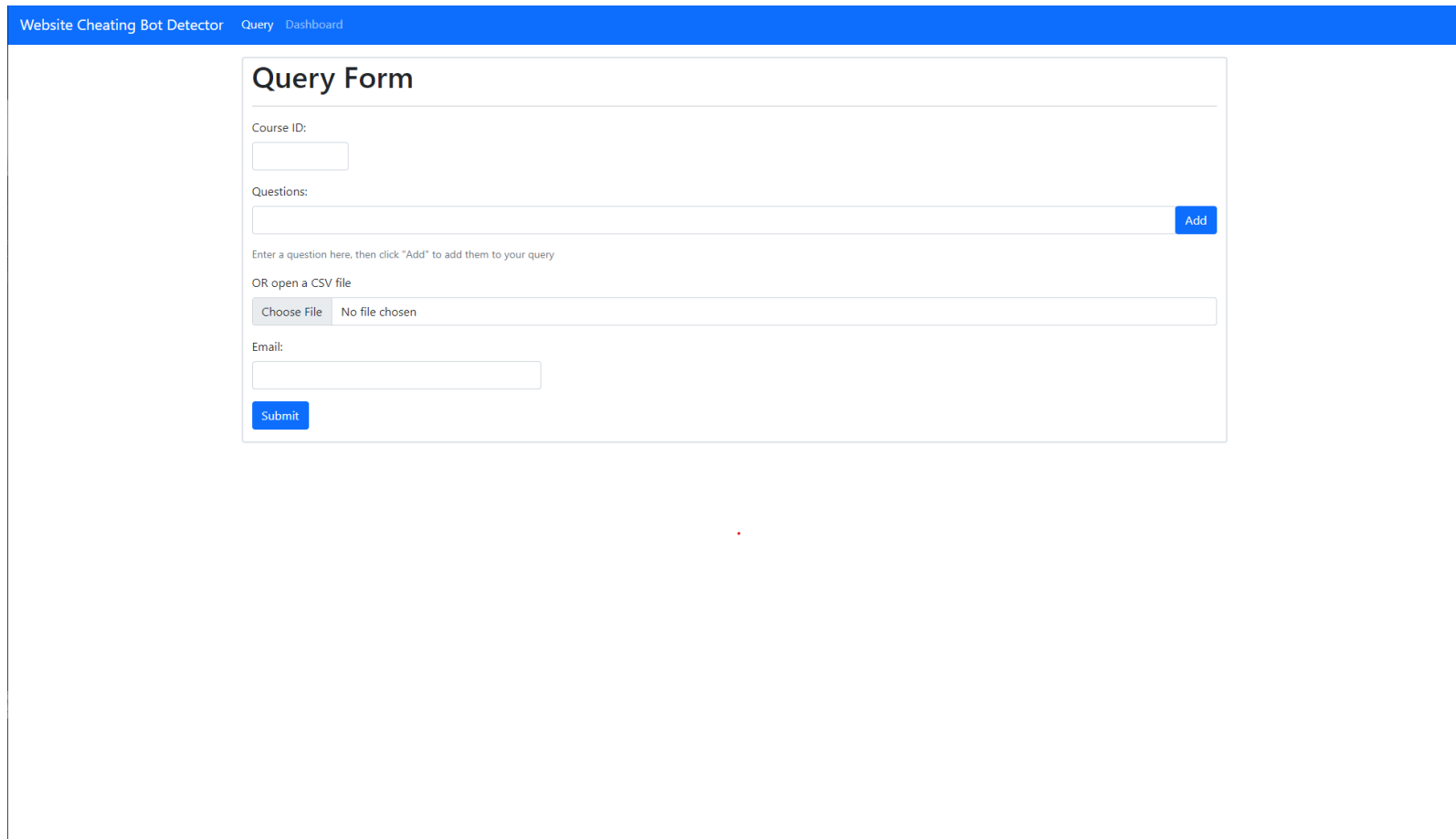
- v. Finished updating the design document and turned in the Code Freeze assignment. Also met with Professor Pfeil and our client to go over the final details of the project.
- vi. Finished and turned in the Final Handoff Report as well as getting client verification.
- vii. Updated EECS project showcase.
- viii. Assisted with identifying and fixing bugs.
- ix. Finished and turned in both the Final Handoff Report and Client Verification.

**b. Eve Horne**

- i. Registered for EECS Project showcase and set up showcase links
- ii. Started re-implementation of front-end html & css layout and created a hosted mongodb instance in atlas.
- iii. Added post endpoint in the backend to kick off the python scraper, and log the results in a mongodb collection.
- iv. Added a query endpoint to the backend allowing users to view the results of scraping jobs. Implemented a dashboard webpage to utilize this endpoint. Implemented email notifications with nodemailer and a gmail account.
- v. Worked on design document and deployed a playground version of the application for code freeze
- vi. Implemented more backend features discussed in sync up meeting with client. Added the ability to search for all results for a given class ID
- vii. Updated EECS project showcase
- viii. Started identifying and fixing bugs found in live application, fixed major issue with pages caching then site and not showing changes to sites.
- ix. Finished and turned in Final Handoff Report and client verification.

## Visuals

Screenshot of our working website:



The screenshot shows a web application interface for a 'Website Cheating Bot Detector'. The top navigation bar is blue and contains the text 'Website Cheating Bot Detector', 'Query', and 'Dashboard'. The main content area is white and features a 'Query Form' with the following elements:

- Course ID:** A text input field.
- Questions:** A text input field with an 'Add' button to its right.
- Instructions:** A line of text stating 'Enter a question here, then click "Add" to add them to your query'.
- File Upload:** A section titled 'OR open a CSV file' containing a 'Choose File' button and the text 'No file chosen'.
- Email:** A text input field.
- Submit:** A blue button at the bottom left of the form.

## Conclusion

From the EECS Project Page, the deliverables for this project are to “to have a successful bot that would alleviate the manual scouring we currently do on various websites to have our content removed and no longer shared with students seeking to commit academic dishonesty. The goal is to reduce the availability of information to support students who are seeking to cheat within their course, and to reduce the amount of academic dishonesty cases within the college.”

We feel confident that our finished product meets these requirements, as well as the requirements provided to us at the beginning of the year.