# School of Computing
# Year 4 Project Technical Specification

Project Title: **Q&A Extraction from Factual Text**

Student Name: **Traian Svinti**
Student ID: **1443218**
Stream: **CASE4**

Project Supervisor: **Yvette Graham**

Completion Date: 19/05/2019

# Table of Contents



# 1. Glossary

- **NLTK -** Natural Language Toolkit for Python.
- **spaCy -** Alternative to NLTK.
- **Postgres** - Database for storing information types.
- **Docker** - Containerisation tool for faster and simpler deployment.
- **Python** - Main programming language used.
- **Django -** Backend framework that connects the UI to the database.
- **AWS -** Amazon Web Server is used as the deployment server of the product.

# 2. Motivation

Motivation for this project came from the search of a challenging project that was accomplishable in the time frame allocated. In my search for inspiration, I came across many NLP based projects, with a large number implementing Q&A systems where the user would input a question and an answer would be found. NLP being a very challenging area in computing I delved deeper into this looking for an appropriate use case in this area, educational being a very strong and motivating area.

With further research, I had an idea of the opposite of a generic Q&A system, similar implementation but instead, the user would input the answer (in the form of factual text) and questions would be generated from the text in a comprehension style. To myself this seemed like a challenging and interesting approach to automating this sort of teaching/quizzing which is very beneficial especially to younger people.

# 3. Research

I began to research this idea, with very little found. To my surprise, one paper was written on a similar approach to this, which I read but could only find use of the dissection of sentences and how the question would be formed from the input, which required much more research.

Continuing research, I needed to find a suitable way to analyze the inputted text. I found NLTK, which is the leading toolkit used in natural language processing with Python. After testing and beginning work, the amount of analysis being done started to noticeably slow down the system and results. This led to further research being done until I found spaCy, a strong alternative to NLTK that done everything that NLTK done but with 80% more efficiency and 20% more accuracy; this became the final toolkit to be used in the text processing.

I needed a better understanding of how questions are created in the english language and how sentences were structured. A lot of my research was in a very non-computing area of language.

Language is such a complicated area to make computers understand and process with no limit to the amount of research that can be done, once I worked on what parts of the input to form questions from, which noun chunks to use and part of speech tag sequencing, I needed to begin to form questions and implement the idea, reducing my research.

# 4. Design

The design of this project directly followed the specification of the Function Specification that was created before the implementation of this project. This was achieved through deep analysis of requirements and the system needing to be implemented before any action was taken on implementation. Little modification was needed to achieve this system.

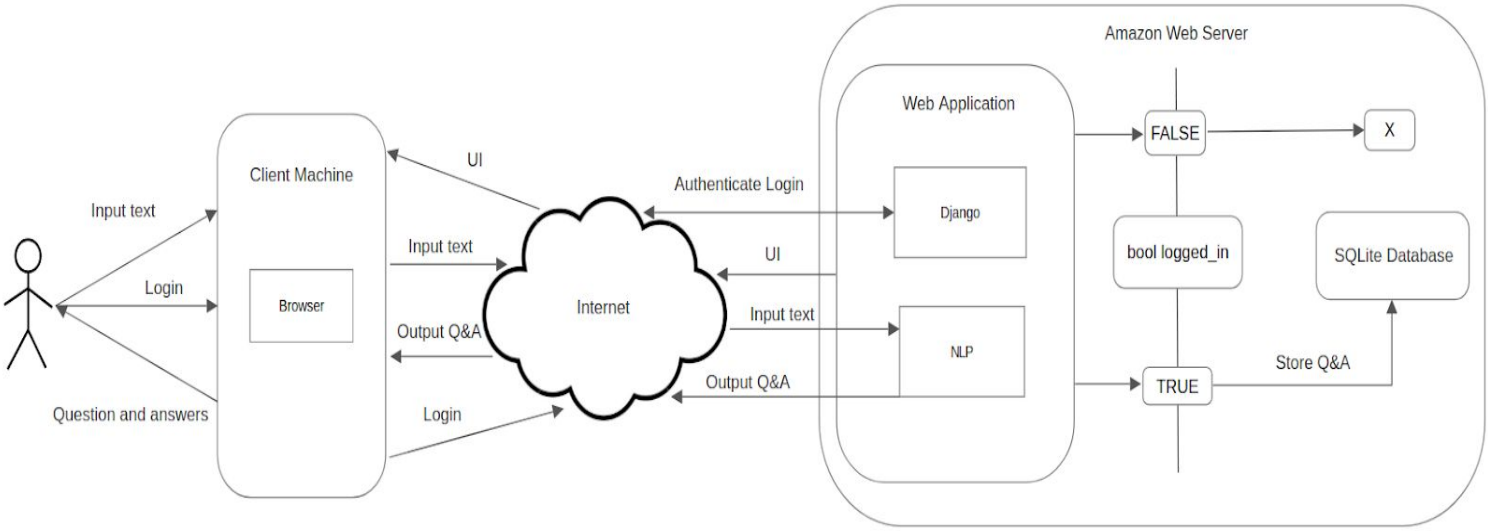


*Figure 1 - System Architecture Diagram*

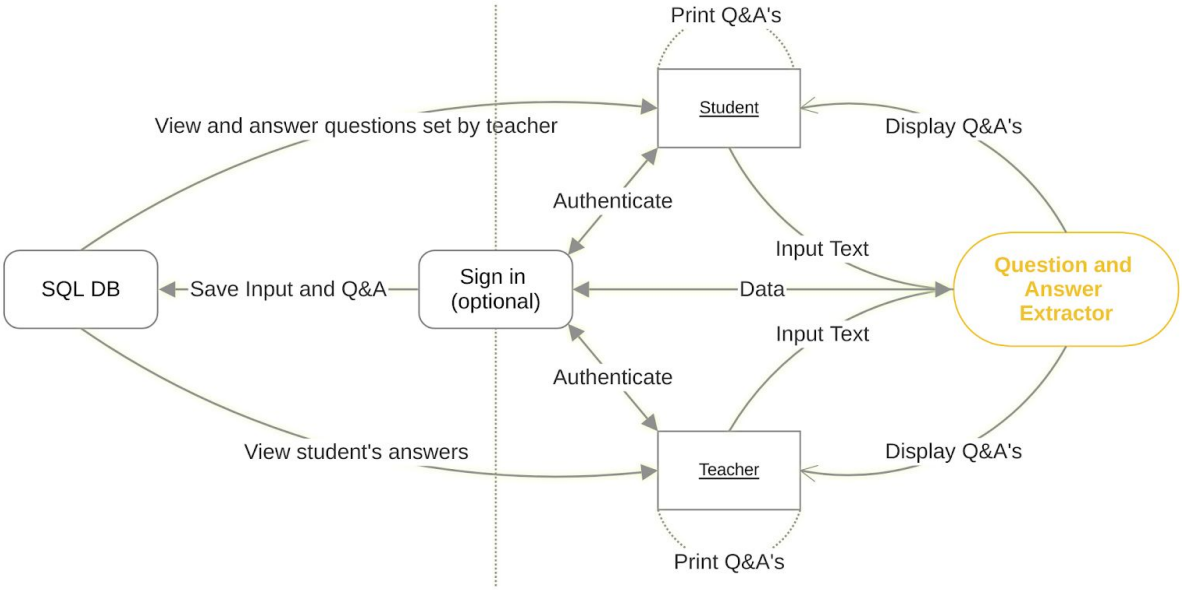


*Figure 2 - Context Diagram*

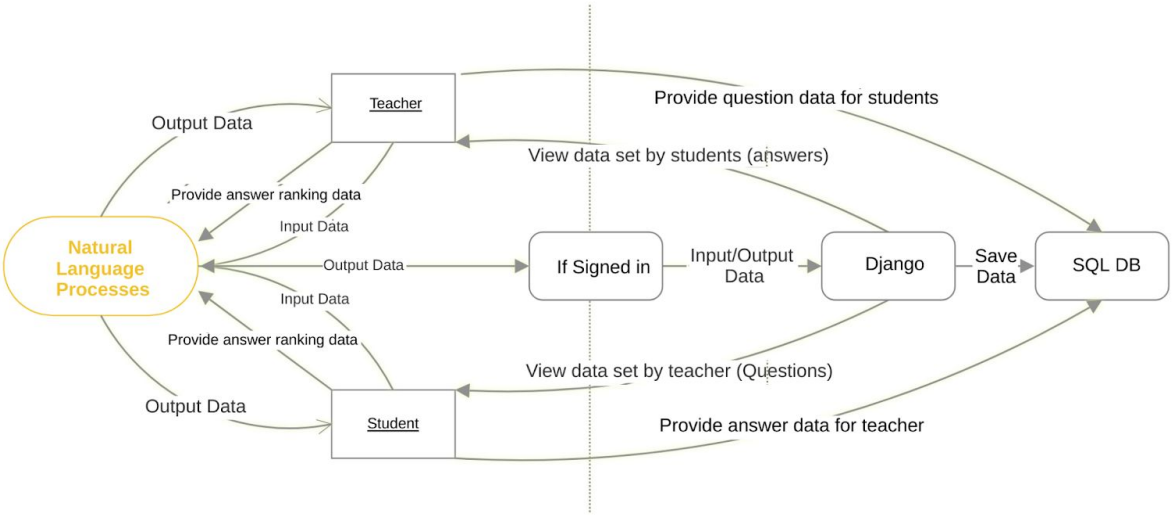


*Figure 3 - Data Flow Diagram*

# 5. Implementation

My aim of this project was to choose a complicated objective, create a full back to front end project and use as many tools as possible that I would not have previously had the chance to use.

These goals led to the use of a wide technical stack such as AWS for server deployment, NGINX for network routing, Docker for containerization for faster and hassle free deployments, Django as the chosen framework, Postgres as a strong and sustainable production database and spaCy for the backend text analysis.

As previously mentioned, spaCy is used for the text analysis. This uses pretrained models to analyze and extract things like entities, monetary values, dates and other information from the text based on datasets which spaCy have. This also provides the framework for noun chunk analysis, part-of-speech tagging of the input and visualisation of the text based on it's properties which helped to understand how this needed to be implemented.

AWS is used to deploy the final project to make it available for everyone to use, no matter where that person is located. This combined with Docker, Django and NGINX, provided great knowledge to myself about how these types of systems work together to create a project of this category.

# 6. Sample Code

I wanted to implement this project as correctly and optimised as I could. A lot of refactoring occurred to ensure the best use of classes and functions. I closely followed a very strict programming structure for Python called PEP8 using highly decoupled classes in an OOP style. This in turn not only made the code easier to look at but provided a better testing structure.

```
class ProcessInput:
    nlp = en_core_web_sm.load()

    def __init__(self, input_):
        self.input_ = input_

    # Tokenise sentence
    def tok_sent(self):
        return self.nlp(self.input_).sents

    # Get SpaCy doc
    def get_nlp_doc(self, sent):
        return self.nlp(str(sent))

    # Get part of speech tags
    def pos_tag(self, sent_doc):
        pos_tagged = [[tok.text, tok.pos_] for tok in sent_doc]
        return pos_tagged

    # Get noun chunks
    def n_chunk(self, sent_doc):
        chunks = [[chunk.text, chunk.root.head.text]
                  for chunk in sent_doc.noun_chunks]
        return chunks
```

*Figure 4 - "question_extractor.py" in "extractor_libs" - Process inputted text*

This class is very clear, it takes the input and pre-processes it before the question creation.

```
class QuestionCreator:
    def __init__(self, input_, answer_):
        self.input_ = input_
        self.answer_ = answer_

    def advanced_q(self, pos_tagged):
        POS_LIST = [
            'VERB', 'ADV', 'CCONJ', 'DET',
            'ADJ', 'NOUN', 'ADP',
            'PROPN', 'PART', 'NUM', 'PROPN'
        ]

        q_list = []
        extracted = ''
        wh_object = 'who'

        for X in self.input_.ents:
            if pos_tagged[0][0] in str(X):
                if str(X.label_) == "ORG":
                    wh_object = "what"

            if advanced_q_dict.get(X.label_) is not None:
                questions = advanced_q_dict.get(X.label_)

                for i, tup in enumerate(pos_tagged):
                    if extracted != '':
                        if tup[1] in POS_LIST \
                           and extracted.split()[-1] == pos_tagged[i - 1][0]:
                            extracted += ' ' + str(tup[0])
                    else:
                        if tup[1] == POS_LIST[0] \
                           and pos_tagged[i - 1][0] != 'I':
                            if tup[0][0].isupper():
                                tup[0] = 'was ' + tup[0].lower()
                            extracted = str(tup[0])

                if extracted != '' and len(extracted.split()) > 2:
                    q_list = [q.replace('{EXTRACTED}', extracted)
                              .replace('{WH}', wh_object)
                              for q in questions]
        return q_list
```

*Figure 5 - "question_extractor.py" in "extractor_libs" - Create advanced questions*

There are two different types of question generation that I have implemented. This is the more advanced type which is based from the part-of-speech tagged words. These tags in certain sequences can output a fluently sounding question. Of course, the program needs to know the question word, which is added from the “q_dicts.py”. This is a dictionary of sentence templates such as filler words that the program would have no way of outputting as it is not available in the inputted text.

```
    def n_chunk_q(self, noun_chunks, q_size):
        q_list = []
        used_ents = []

        for X in self.input_.ents:
            if n_chunk_q_dict.get(X.label_) is not None \
               and str(X) not in used_ents:
                used_ents.append(str(X))
                questions = n_chunk_q_dict.get(X.label_)
                questions = [questions[random.randrange(len(questions))]]

                n_chunk = [chunk[1] for chunk in noun_chunks
                           if chunk[0] == X.text]

                if n_chunk != []:
                    q_list += [q.replace('{VERB}', n_chunk[0].lower()).replace
                               ('{ENT}', X.text)
                               for q in questions]
                else:
                    if X.label_ == "DATE":
                        if X.text[-1] == 's':
                            n_chunk = 'are'
                        else:
                            n_chunk = 'in'
                    else:
                        n_chunk = 'is'

                    q_list += [q.replace('{VERB}', n_chunk).replace
                               ('{ENT}', X.text)
                               for q in questions]
        return q_list

    def q_decider(self, q_list, q_size):
        decided_list = []

        if len(q_list) > 1:
            while len(decided_list) < q_size and len(q_list) > 0:
                chosen = q_list[random.randrange(len(q_list))]
                decided_list.append(chosen)
                q_list.remove(chosen)
        else:
            decided_list = q_list
```

*Figure 6 - “question_extractor.py” in “extractor_libs” - Create noun_chunk questions*

This second approach to question creation is less advanced, using extracted noun chunks from the inputted text, filler words are used to create a fluently sounding sentence which correlates to the input.

The “q_decider” function simply determines which questions to actually output to limit the generation of questions.

This is code from the extractor_library. Due to using Django as the framework, code is spread throughout the ‘src’ directory in a structured manner. Below is a screenshot of the ‘src’ directory structure showing 2 levels deep. This also followed a very strict structure.

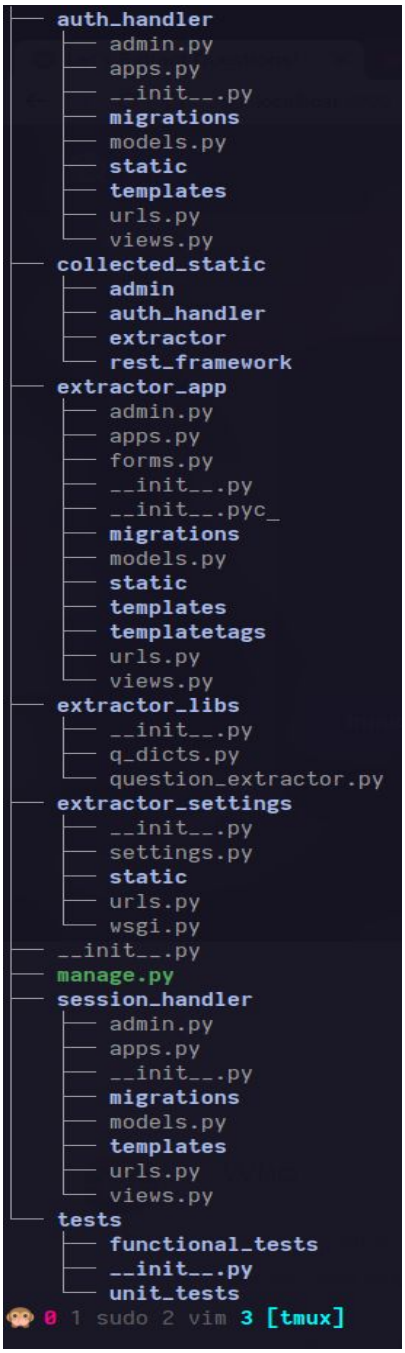


*Figure 7 - Project code directory structure of ‘src’, 2 levels deep*

# 7. Problems Solved

In the course of completing this project, problems arose in many different forms whether backend or frontend based. All problems were resolved and major ones are outlined below.

***Problem 1***

A major problem I had was how the extractor library would pass the question and answers to the frontend for visual parsing. In a Django template, codeblocks are used to be able to loop and filter in the html code itself. It makes sense that the extractor would return a list of questions and an answer for each question. One answer can have multiple questions associated with it. I was left with the question of "How will I link the answer to the list of associated questions?". This was solved using the below method.

Returned from the extractory library is a list of lists. The inner lists would contain questions, with the final value of the list being the answer. This made it very easy to output the questions visually by looping the inner list and first checking if the current value was the last value in the list, if so, display on the answer side, if not, display on the question side. The program will then move to the next list of questions and answer, which would be displayed under the previous.

***Problem 2***

A problem that occurred in unit testing had to do with the RequestParser class. This class take a POST request from the frontend which is passed from the *views.py* file. At first I was unsure of how to test a method independently which takes a request object (type QueryDict) as input. With further research, I found that I can use a tool called Pickle. This allowed me to save a sample request in a file, which can then be accessed and fed in testing the class.

***Problem 3***

A rather pressing problem that was encountered was the file input versus the text input. The text input worked as expected, with the same text placed in the file, using the same analysis and creation process, the file input would output very unexpected results.

Upon further research, I found that the problem was in how the text was encoded. Txt files would use a specific encoding which needed to be decoded to utf-8 compared to the text input. Newline characters needed to be removed forcefully as they were appearing in parsing and were unneeded. This was resolved by using the below python code when parsing the text in the file:

```
12                if form.is_valid():$
11                    input_ = request.FILES['my_uploaded_file'] \$
10                            .read().decode("utf-8", "strict") \$
 9                            .replace('\n', ' ')$
 8                    input_ = input_.strip()$
```

*Figure 8 - File input decoding, parsing and stripping*

***Problem 4***

A problem that was previously mentioned was how to display lots of information for the user in a sorted way. This was resolved by implementing Bootstrap collapse method in an accordion style. A bigger problem arose in that this information was displayed in codeblocks and looped over to be accessed, this provided difficulty for using this Bootstrap class as this is not how it was intended to be used. To make matters worse, I needed these collapsable to be nested.

I resolved this issue by using the names of the collapse-(value of loop index) to provide different numbers for the html code to be able to differentiate the objects but still have relations.

```
<div id="collapse-{{ forloop.parentloop.counter }}-{{ forloop.counter }}" class="collapse"
```

***Problem 5***

Another issue encountered was in the testing stages of the UI tests. There was a major issue that hours were spent on where the page source being returned by Selenium was Javascript based which was very strange. This meant that I was unable to find any elements on the webpage.

The problem was found to be within the communication between localhost:8000 and the Selenium Hub itself where, even though they were linked, returned the incorrect/malformed data.

This was resolved by using extractor.thesvinti.com as the URL instead of local testing, which solved the problem once the systems volume was added to the docker container to stop the browsers from crashing.

```
chrome:
image: selenium/node-chrome:3.141.59$
volumes:
- /dev/shm:/dev/shm
depends_on:
- selenium-hub
environment:
 - HUB_HOST=selenium-hub
 - HUB_PORT=4444
logging:
driver: none
```

# 8. Results

## Main Component Results

In regards to the main component, the extraction of questions, the results, both positive and negative are outlined below:

*Sample text:*
"Monrovia was named after James Monroe, who was president of the United States in 1822. In that year a group of freed U.S. slaves, sponsored by a U.S. society, started a new settlement on the continent of their ancestors. As more settlers arrived from the United States and from the Caribbean area, the area they controlled grew larger. In 1847 Monrovia became the capital of the new country of Liberia."

*Questions:*
Q1. What was named after James Monroe?
Q2. Why is James Monroe mentioned?
Q3. What happened in 1822?
Q4. Why is that year mentioned?
Q5. Who arrived from the United States and from the Caribbean area?
Q6. United States appears in the text, why is this the case?

Question 1, 3 and 5 are the result of the previously mentioned *'advanced_q'* function. This is an example of a good outputted question from the text. The problem with only having this function is that, depending on the input, it will find it more difficult to produce any sort of output at all, which is why the function '*n_chunk_q' is also needed.*

The remainder of the questions are the output of '*n_chunk_q'.* Most questions are fluently accurate in a general sense, analyzing question 3 we can see this is not the case and shows the problem of why NLP is a difficult task. The year is mentioned in the previous sentence, but the analyzer cannot link that the two sentences are related, simply finding a statement about a year and outputting the question "*Why is that year mentioned?".* Unfortunately it is the case that this approach to question extraction and every other, is subject to mistakes as such.

Nonetheless, I do believe that the webapp delivers on its promise of extracting questions from inputted text with acceptable results.

## Testing Results

I have implemented three different testing types to ensure that the quality of the software is sufficient.

***Unit Testing (Component Testing)***

I executed unit testing using a Python library called Pytest. This was done using the method of 'test driven development' meaning that tests were written before any code was written, knowing the input and what the expected output was. Pytest integrates with Django to create a testing suite.

Tests were written in the same form as the code is written, using classes and functions. Below is an extract of one test which returns the part-of-speech tags for each word:

```
test_input = "This is a test. Monrovia was named after James Monroe.
Passing this test is easy."
```

Firstly, I needed to preprocess this sentence by getting the text in a 'doc' type which the processor accepts and then passing this to the function in the extractor library:

```
doc = test_process_input.get_nlp_doc(test_sent)
test_pos_tagged += [test_process_input.pos_tag(doc)]
```

This is then asserted against an expected output as shown below:

```
    def test_pos_tag_return_list(self):
        assert test_pos_tagged == [
            [
                ['This', 'DET'],
                ['is', 'VERB'],
                ['a', 'DET'],
                ['test', 'NOUN'],
                ['.', 'PUNCT']
            ],
            [
                ['Monrovia', 'PROPN'],
                ['was', 'VERB'],
                ['named', 'VERB'],
                ['after', 'ADP'],
                ['James', 'PROPN'],
                ['Monroe', 'PROPN'],
                ['.', 'PUNCT']
            ],
            [
                ['Passing', 'VERB'],
                ['this', 'DET'],
                ['test', 'NOUN'],
                ['is', 'VERB'],
                ['easy', 'ADJ'],
                ['.', 'PUNCT']
            ]
        ]
```

*Figure 9 - Asserting function output to expected output*

Using this process, I have 14 unit tests covering every part of the extractor library, resulting in a total library coverage of 93%. **Tests are set to run every time the the containers are ran, ensuring that if something changes and a test fails, it will be noticed.**

***Functional Testing (UI tests)***

The UI was tested using Selenium. **A fully decoupled UI testing Suite** was created using docker where there is a main **Selenium Hub in it's own container.** This communicates with the actual extractor container. **Two extra containers are created and started, one that runs chrome and one that runs firefox independently as nodes connected to the Selenium hub.**

Due to UI testing being a **slow processes of running these tests** (due to the running of Chrome and Firefox headless) I was unable to automatically run these tests at the same time as the unit tests to avoid the delay in deployment. Nonetheless, tests were created to ensure system UI functionality, below is some sample code of how tests were implemented:

```
2 s/t/f/test_index_extractor.py  2 s/e/t/e/index.html
23 class Test_Extractor(unittest.TestCase):$
22 $
21     def setUp(self):$
20         caps = {$
19             'browserName': os.getenv('BROWSER', 'chrome')$
18         }$
17         self.browser = webdriver.Remote($
16             command_executor='http://localhost:4444/wd/hub',$
15             desired_capabilities=caps$
14         )$
13 $
12     def test_extractor_homepage(self):$
11         browser = self.browser$
10         browser.get('https://extractor.thesvinti.com/')$
 9 $
 8         time.sleep(1)$
 7         self.assertIn('Extract Questions', browser.page_source)$
 6 $
 5     def test_extractor_go_to_login(self):$
 4         browser = self.browser$
 3         browser.get('https://extractor.thesvinti.com/')$
 2 $
 1         btn = browser.find_element_by_xpath('//*[@id="navbarNav"]/ul/li[6]/a')$
31         btn.click()$
 1 $
 2         time.sleep(1)$
 3         self.assertIn('Login', browser.page_source)$
 4 $
 5     def test_extractor_go_to_register(self):$
 6         browser = self.browser$
 7         browser.get('https://extractor.thesvinti.com/')$
 8 $
 9         btn = browser.find_element_by_xpath('//*[@id="navbarNav"]/ul/li[7]/a')$
10         btn.click()$
11 $
12         time.sleep(1)$
13         self.assertIn('Sign Up!', browser.page_source)$
```

*Figure 10 - UI Testing - Asserting page_source output to expected output*

As can be seen, once a button is clicked it will redirect the browser to that link and check if the new page source contains the unique text depending on the page.

Multiple UI tests were written to ensure that nothing breaks and would be run after re-building the project to ensure functionality is correct and as expected.

***System Testing - Regression Testing***

Regression testing was implemented constantly and thoroughly throughout the implementation of this system. This was achieved through automated tests on every docker container startup.

Below the flow is outlined:

1. Unit test (component test) is written to test the component which has not yet been created.
2. Code is created to implement a feature.
3. This code component is tested with the pre-created unit test.
4. Code is integrated with all other code to ensure compatibility.
5. The unit test for the component is run independently again, to ensure passing.
6. The containers would be rebooted (and sometimes fully pruned to ensure maximal test coverage on new system as well as a pre-used system and database). Once the containers start up again, unit tests and functional tests are automatically run at startup (entrypoint) to ensure that no other code is failing due to the new component.

This strategy proved very useful and thorough to help iron out issues with new components and their integration.

***System Testing - Load Testing***

Load testing was implemented by mimicking multiple users using the system simultaneously. This was achieved using Selenium and the running of all UI test concurrently. Due to the way the UI tests were implemented, wait times were added to properly simulate user interaction.

This type of testing showed no issues due to the infrastructure used and the efficiency of the system and the load balancing done with AWS. Up to 50 users were tested running the same test simultaneously but in different orders and a log of any issues being recorded.

Below is the code that the script ***autorun-parallel-ui-tests*** calls:

```
from subprocess import Popen$
$
$
processes = []$
$
for counter in range(1):$
    chrome_cmd = 'export BROWSER=chrome && python3 src/tests/functional_tests/test_extractor.py'$
    firefox_cmd = 'export BROWSER=firefox && python3 src/tests/functional_tests/test_extractor.py'$
    processes.append(Popen(chrome_cmd, shell=True))$
    processes.append(Popen(firefox_cmd, shell=True))$
$
for counter in range(1):$
    processes[counter].wait()$
```

The range can be increased on both loops to mimic more/less users. This runs UI tests on both Chrome and Firefox ensuring **cross-browser compatibility.**

***System Testing - Recovery Testing***

I wanted to ensure that if the system would crash for whatever reason in deployment that it would attempt to auto-recover and reboot the containers, clearing any errors. This was implemented by writing a shell script that would check if the system is running, if it was not, it would boot the system. A Cronjob would run every minute doing this check and running this script meaning that even if I manually killed the AWS instance and started it again, I would not need to manually access the system to start the containers, this would automatically recover with no data lost.

For testing, I attempted to kill the containers and the instance in different way to simulate failure, each time the containers were able to reboot in less than 1 minute as the job was run.

***User Testing***

With regards to user testing, I wanted to have a wide range of different types of users from technical to non-technical and NLP field related. Each user received a Plain Language Statement which was agreed to before attempting to use the system and answer the survey.

With the area of NLP and language in general being so flexible, creating a system that can accurately create questions for all types of input is not in the scope of a student project due to limitations such as data amount, data type, cost of storing this data and time involved. This system works best with historic type documents that have a lot of information that questions can be extracted from. Nonetheless, the liberty of input was given to the users, below are the questions and analysis, a total of 13 responses:

1. "Is the website intuitive? Does it provide ease of use and is it pleasing to the eye?"
   Percentage: 0% - 100% = 94%

   Some extra feedback was provided for this question such as:
   - "It's straightforward and quick. The design is nice and simple!"
   - "Very good and intuitive UI, easy to use and navigate around website."
   - "Interactive front end is nice. Good use of colors."
   - "Very nice looking UI."
   - "Very neat UI/UX, with the exception being the button for submitting copy pasted text is a bit too low on the page making it hard to see."

   The one negative feedback was taken into account already, as different resolutions did indeed make the button hard to see, this was resolved.

2. "Are the questions outputted accurate to the inputted text?"
   Percentage: 0% - 100% = 74%

   Some extra feedback was provided for this question such as:
   - "Sometimes questions were a bit strange but generally this works well!"
   - "Dependent on the text, there were both good questions and some ill formed questions"

3. "Are the questions outputted advanced?"
   Percentage: 0% - 100% = 77%

   Some extra feedback was provided for this question such as:
   - "The questions are great and well intricate and detailed"
   - "your system is perfect"
   - "System operates exactly as specified."
   - "I found the movement of some icons cool"

4. “Did you notice anything strange while using the system?”
   No = 12
   Yes = 1

   Some extra feedback was provided for this question such as:
   - “Not strange but a few things I noticed. When signing up I purposely entered a different password to the confirm password section. I got a message that stated it was incorrect but it cleared all the previous fields, but I would have liked for the previous fields to remain.”

5. “Do you have any suggestions?”
   No = 10
   Yes = 3

   Some extra feedback was provided for this question such as:
   - “Maybe a way to input a scanned copy of a textbook would be a nice addition”
   - “After you submit the text there is the "Select All" option, I think saying something like "Select All Questions you wish to be saved" would be more appropriate. When I first did it I wasn't 100% what "Select All" meant”
   - “Align the text content of the website to the left.”

All this feedback is much appreciated and has been/will be taken into consideration for any features or reparations to improve the system. I was pleased to know that no major bugs have been found across multiple different browsers, screen resolutions and user types.

# 9. Future Work

This project has an endless application of potential features and improvements. Future work that I would have liked to have the time to implement would be to give the user the choice to input a start time for a session and a duration.

Another feature I would like to implement is the ability to search questions and session and filter based on a specific users answers and grades.

A very appropriate feature would be the option to input a scanned .pdf of a textbook and using OCR, extract the contents of that text and then feed it to the preprocessor and question creator.

I do not believe any of these features would be difficult to implement but time was against me and I needed to use it wisely.