SENG3011 20T1

Testing Details

Medics

Abanob Tawfik	z5075490
Kevin Luxa	z5074984
Lucas Pok	z5122535
Rason Chia	z5084566

API Testing

We tested our API extensively during development to ensure that it would respond appropriately to user requests. Initially, our testing involved manual inspection of the API response. Later, we automated the testing process to help speed up our workflow.

Environment

Most of our testing was performed locally (with the API running on localhost) on a Windows machine. Later in development, we also tested our live API via our Swagger UI’s “try it out” functionality.

Manual tests

We performed manual testing using Postman, a popular tool used in API testing. In Postman, we specified parameter values by entering key-value pairs in the ‘Params’ tab, which automatically updates the query string. Postman also handles the URL encoding of the data. Shortly after clicking the ‘Send’ button, the response appears in the panel below. We then checked the received response and compared it to the expected result.

Automated Tests

For our automated tests, we created a shell script to send requests and compare the responses from our API with the expected results. Our inputs took the form of small shell scripts which contain variable assignments for each parameter that we wanted to test. For example, here is an input file for testing the start and end date parameters:

```
#!/bin/sh
# getting multiple articles 1

start_date=2010-03-03T00:00:00
end_date=2010-04-25T00:00:00
```

The main testing script sources the input files and generates a query string containing all of the parameters defined by the input file. It then sends a request to the API via the **curl** command, formats the response, removing all variable output such as access time, and stores the output in a file.

```
curl -s -G "http://localhost:5000/api/Test/GetArticles?$query" -H "accept: */*" |
python3 -m json.tool |
sed -e 's/"accessed_time": ".*"/"accessed_time": "REMOVED"/' > "$output_filename"
```

The script then uses the **diff** command to compare the output with the expected output. If they match, the script prints “Test passed”, otherwise it prints “Test failed”.

Testing Data

While testing our API, we needed to ensure that the expected results did not change frequently. Otherwise, we would need to constantly update them. There were two possible solutions we thought of to resolve this: (1) always specifying a date range from the past, and (2) creating test articles and storing them in a separate test database. We opted for the latter, as it guaranteed that the data used to test our API would not change, which meant that we wouldn't need to update our expected results too often. Furthermore, it was unlikely that regular articles from the CDC site would enable us to cover all the cases we wanted to test.

Our test data changed frequently over the course of development. There were a number of reasons for this:

1. We began with a set of 10 articles stored in our test database, which were designed to test regular cases. Later in development, we devised and organised test data to cover a wider variety of cases.

2. We realised that in order to simplify database querying and make querying more efficient, the articles, reports and locations that we store in our database would need to contain more data than the objects that we return to the end user.

   For example, exact dates in the objects returned to the user need to be strings in the format "yyyy-MM-dd HH:mm:ss", and components of the date that are unknown could be replaced with "xx". However, if we simply stored this string in the database, we would need to perform more computation while responding to a query to check whether the date matched their search criteria. We resolved this issue by including two more fields in each article and report object that contain the earliest and latest possible dates that could be interpreted from a date string. For example, the earliest and latest possible dates that could be interpreted from "2020-02-xx xx:xx:xx" are 2020-02-01 00:00:00 and 2020-02-29 23:59:59.

   Thus, in addition to the Article, Report and Place objects that we return to the user, we created StoredArticle, StoredReport and StoredPlace objects that contain additional information, and methods for converting these to their non-stored equivalents. However, the amount of additional information we need to store was uncertain, so the structure of our StoredArticle, StoredReport and StoredPlace objects changed over time.

The main disadvantage of manually creating test data is that it is time consuming. In order to save as much time as possible, we divided our test cases into categories and created test data with the bare minimum amount of information needed to test each category. For example, the test data that we created for our date tests did not contain any diseases, syndromes or locations, or any lengthy text, as these were irrelevant. Meanwhile, all of the test data we created for our location tests contained the same date ranges, as the dates were irrelevant. Below are examples of test data that we used for our date and location tests.

```
{
    "url": "url1",
    "date_of_publication_start": "2010-03-05T12:00:00.000+00:00",
    "date_of_publication_end": "2010-03-05T12:00:00.000+00:00",
    "date_of_publication_str": "2010-03-05 12:00:00",
    "headline": "Headline of url1",
    "main_text": "This is the main text for url1.",
    "keywords": [],
    "reports": [
        {
            "diseases": [],
            "syndromes": [],
            "event_date_start": "2010-03-01T00:00:00.000+00:00",
            "event_date_end": "2010-03-04T23:59:59.000+00:00",
            "event_date_str": "2010-03-01 xx:xx:xx to 2010-03-04 xx:xx:xx",
            "locations": []
        }
    ]
},
```

Example of test data used to test the start date and end date parameters

```
{
    ...
    "reports": [
        {
            "diseases": [],
            "syndromes": [],
            "event_date_start": "2014-01-01T00:00:00.000+00:00",
            "event_date_end": "2014-12-30T23:59:59.999+00:00",
            "event_date_str": "2014-01-01 xx:xx:xx to 2014-12-30 xx:xx:xx",
            "locations": [
                {
                    "country": "Australia",
                    "location": "Sydney, New South Wales",
                    "geonames_id": "34567",
                    "location_names": [
                        "Australia",
                        "Sydney",
                        "New South Wales",
                        "NSW"
                    ]
                }
            ]
        }
    ]
},
```

Example of test data used to test the location parameter

We organised our tests and test data into these categories: (1) start_date and end_date, (2) timezone, (3) key terms, (4) location, (5) max and offset, and (6) errors. This organisation of our test cases made it easy to modify and add new tests.

We also set an allowed range of dates for test data in each category. For example, test data for our date tests must only contain dates between 2010 and 2012, while test data for our location tests must only contain dates in 2014. Doing this meant that adding a new article to test a case in one category would not affect the expected results of test cases in other categories. The details of how we divided dates among categories is in the file TestScripts\Data\README.txt.

Test Case Development

The development of our test cases mirrored the development of our API - as we updated our API implementation to handle more parameters and edge cases, we devised test cases for testing these parameters/cases. We began our implementation assuming that all input parameters were valid (the normal case), and then added error handling later.

We first implemented and tested the date matching functionality. Handling the start date and end date parameters was most important, as these were the only parameters end users needed to provide with their request. We designed our tests of these parameters to be extremely stringent, so that slight inaccuracies would result in a failed test. For example, we created a test article whose event date was 2010-03-01 xx:xx:xx to 2010-03-04 xx:xx:xx, and then a test case which specified the end date parameter as 2010-02-28T23:59:59. If our implementation contained any slight issues with date parsing, we were likely to fail this test.

Next, we implemented and tested key term matching. When dealing with key terms, normalisation was important, as we wanted queries containing the same words to give the same result, regardless of case or spacing. For example, a request that includes “   coronavirus” as a key term should get the same result as a request that includes “Coronavirus” as a key term. Thus, our test cases for the key terms parameter include terms with odd capitalisation and spacing.

Then, we implemented location matching. Location matching was very challenging as we wanted to match locations at different levels. For example, a request for the location “New South Wales” should retrieve all articles with reports that reference locations in New South Wales, including all of the suburbs in NSW, and not just articles that referenced NSW. Therefore, while testing the location parameter, we created multiple articles that referenced different locations in the same country/state, and then sent requests for those locations as well as the country/state itself. For example, we created articles that referenced Sydney, Newcastle, and Adelaide, and then sent requests for Sydney, New South Wales, and Australia.

Finally, we handled and created tests for the max and offset parameters, which allow users to specify which articles they want to be returned out of a range, and then handled error cases.

The table below contains an overview of our test cases.

Category	Test Cases
start_date, end_date	● No articles retrieved ○ Requested date range is earlier than all report dates ○ Requested date range is later than all report dates ○ Requested date range is between report dates ● Requested date range is completely inside a report's date range ● Report's date range is completely inside a requested date range ● Requested date range overlaps with beginning or end of a report's date range ● Start date is the same as end date ● Only one result ● Multiple results ● Article with multiple reports
timezone	● Checking that the default is UTC ● Checking a neural timezone (i.e., ±00:00) ● Checking a positive timezone (e.g., +01:00) ● Checking a negative timezone (e.g., -01:00) ● Checking a timezone further away from 0
key_terms	● No key terms provided ● One key term provided ● Using a general search term (e.g., "virus") ● Using a specific search term (e.g., "ebola") ● Two key terms provided ● More than two key terms provided ● Case insensitivity ● Leading and trailing whitespace and empty key terms or key terms consisting of only whitespace
location	● No location provided ● Searching for a fictional location ● Searching for a location without reports ● Searching for a suburb/city (e.g., Sydney) ● Searching for a state/province (e.g., New South Wales) ● Searching for a country (e.g., Australia)

max and offset	● max is 1 ● max is less than the number of matching articles ● max is greater than the number of matching articles ● No max provided (defaulted value is 25) ● max is capped at 50, no matter what the user specifies ● offset is 0 ● No offset provided (default value is 0) ● offset is greater than 0 and less than the number of matching articles ● offset is greater than the number of matching articles
errors	● Start and end date both not provided ● Start date provided, end date not provided ● Start date not provided, end date provided ● Invalid date ○ “t” instead of “T” ○ Time not specified ○ February 29 on some years ○ Invalid time ● End date before start date ● Ambiguous timezone abbreviation ● Unknown timezone abbreviation/name ● Invalid limit ● Invalid offset

Output

The test script outputs the API response for each test (with variable output removed) to the folder TestScripts\Output. To generate the expected output, we ran our script without the **diff** checking, and then manually inspected the output to check that it was correct. After verifying all the output, and regenerating the expected output if it was incorrect, we copied these files into the expected results folder (TestScripts\Expected). We were then easily able to test our API while making changes and improvements.

Limitations

There were a few limitations of our testing.

Firstly, we had planned to load test our API using Loader.io, a load and scalability testing service, but eventually did not have time for it. This wasn’t a huge limitation as our API will only be known to other members of the course, so we don’t expect a large amount of traffic.

Secondly, because our API simply extracts matching reports from our database, which is populated by our web scraper, our tests do not test the correctness of our web scraper at all. They assume that our web scraper will correctly extract all the required fields such as the disease, syndrome, date range and location from the CDC articles. In reality, our web scraper will rarely be perfect, and we will need to inspect each extracted article manually.