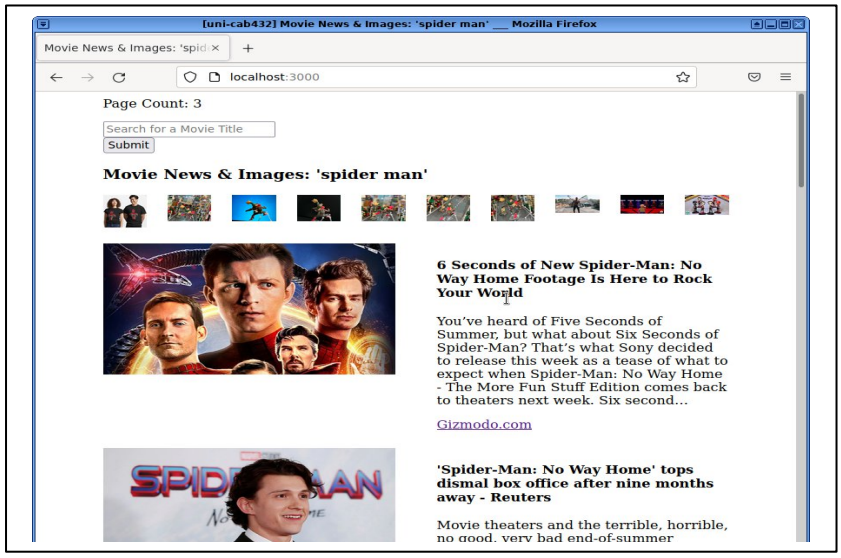
2022

Movie News & Images



CAB432 Assignment 1

Jack Muir

N10467009

Contents

Introduction

Mash-up Purpose & Description

This application allows a person to submit a search query containing a movie title. If a movie title is returned, the application will return and display a list news articles containing the article photo, title, shortened description, and a hyperlink to the original source article. It additionally queries photos from Flickr using the movie title.

Services Used

The Movie Database API

Enables the search for a large collection of movie and TV show titles from a given query. If a query is successful, it returns information including the title, release date, popularity, and more.

Endpoint:

https://api.themoviedb.org/3/search/movie/?api_key=<api_key>&include_adult=false&query=<URI_encoded_query>

Docs:

https://developers.themoviedb.org/3/getting-started/introduction

NewsAPI

Returns articles from thousands of news sources matching a given query. If a query is successful, the article photo, title, description, and source are displayed on the page.

Endpoint:

https://newsapi.org/v2/everything/?q=<URI_encoded_query>&pageSize=10&apiKey=<api_key>

Docs:

https://newsapi.org/docs/endpoints

Flickr API

Returns a list of photos matching a given query. If a query is successful, a json object containing a large photo URL (*b_url*), the page URL (*p_url*), and the photo's title is appended to a list of JSON objects. They are then simply hyperlinked and linked to on the rendered page.

Endpoint:

https://api.flickr.com/services/rest/?method=flickr.photos.search&key_key=<api_key>&tags=<query>&per_page=10&format=json&media=photos&nojsoncallback=1

Docs:

https://www.flickr.com/services/api/

Persistence Service

This application utilized in this application was an Amazon S3 bucket.

Mash-up Use Cases and Services

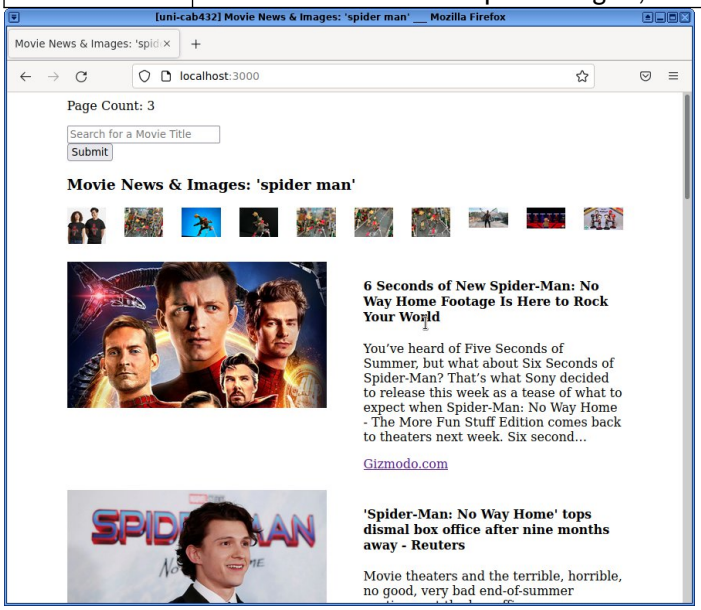
Your User Stories should go here. The basic structure is provided, and you should fill in the role and then the action and the good result that follows. Underneath the formal statement of the User Story,

you can then tell us how you have implemented this service – basically you would tell us how you get input from the user and then use this to get information from an API and then use those results in another one. This is at a semi-technical level – introduce it at a high level and then give more detail, but stop well short of code excerpts. You should then use screenshots to illustrate the process

To illustrate some of what we want, I will use an example relating to travel:

Discover Movie News

As a	Movie watcher
I want	To find new articles relating to a movie query
So that	I can find news articles pertaining to, and potentially related to, my query



The user can interact and perform queries using the search field at the top of the page. Then, located beneath the Flickr image results, the articles can be seen displayed one after another down the length of the page.

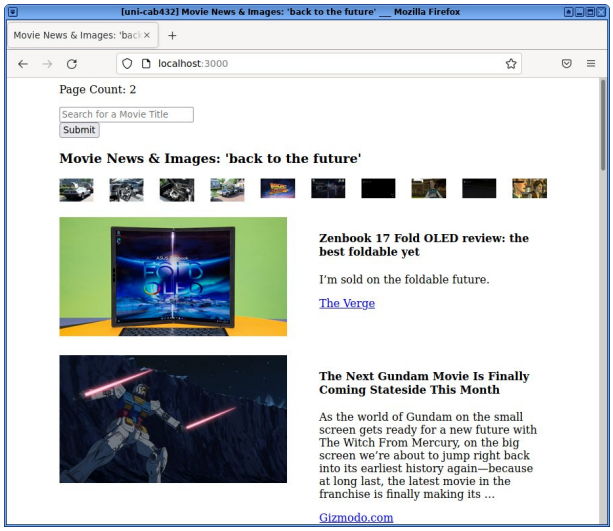
Discover Movie Images

As a	Movie watcher
I want	Discover images relating to a movie query
So that	I can discover images uploaded to Flickr relating to my query

Preceding the NewsAPI article results, after the search field, an inline list of images can be seen that relate to the original query

Discover New Topics

As a	Internet browser
I want	Discover new information vaguely relating to my query
So that	I can potentially learn something new



On occasion, if the movie title being queried through NewsAPI contains more generic keywords, the results returned can contain information not pertaining to the original query. This can lead to some instances were the user can see news articles that might leave them curious.

Technical breakdown

Architecture and Data Flow

The architecture of this application is straightforward and simple, containing limited complexity with a focus on readability. The project folder structure was initially generated using express-generator. with an additional ‘./api’ folder for organizing the functions that call the individual APIs. The app.js was, for all intents and purposes, left unmodified. So for the purpose of this discussion, we will ignore it.

The index.js route is the primary and only route that express links to. Within this file we see two routes defined for the root (‘/’). One is a GET request, and the other a POST. The GET route is rudimentary and uninteresting, only defining the page title, updating the persistence counter on each page load, and rendering an index.pug page blank of any content.In contrast, the POST route pieces together and links the applications together.

A POST request containing a query string (*req.body.moviename*) is made to this route from the input form, first defined in the layout.pug, and then extended by the index.pug template.As aforementioned, a separate file is defined for each API call is defined in the ‘./api’ folder for organizational purposes. The index.js file includes the API functions accordingly. Each file and their associated function(s) will be covered individually in the order they are called from within index.js. If no query was set in the body of the POST request, then an error page is rendered on the page informing the user as such.

```
// The Movie Database API details
const options = {
    api_key: '89f29b52094db728b20171bd16abed84',
    hostname: 'https://api.themoviedb.org',
    adult: 'false',
    path: '/3/search/movie'
}

// Search The Movie Database for movie titles based on query
async function searchTheMovieDB(query) {
    return new Promise((resolve, reject) => {
        const encodedQuery = encodeURIComponent(query);
        const url = options.hostname + options.path +
                '?api_key=' + options.api_key +
                '&include_adult=' + options.adult +
                '&query=' + encodedQuery;
        axios.get(url).then((response) => {
            resolve(response.data)
        }).catch(error => {
            reject(error)
        })
    })
}
```

Firstly, the movie title is retrieved from The Movie Database with the *themoviedb.searchMovie(query)* promise. The function itself is just an *axios.get(url)* promise. Upon fulfillment of the promise, the consuming code simply ensures the returned results does not exceed the maximum (defined as 10) before continuing.

```
// NewsAPI details
const options = {
    api_key: '6f617da0fe3f479789448f115668a0ce',
    hostname: 'https://newsapi.org',
    path: '/v2/everything',
    page_size: 10
}

// Search NewsAPI for articles based on query
async function searchNewsAPI(query) {
    return new Promise((resolve, reject) => {
        const encodedQuery = encodeURIComponent(query)
        const url = options.hostname + options.path +
                    '?q=' + encodedQuery +
                    '&pageSize=' + options.page_size +
                    '&apiKey=' + options.api_key;
        axios.get(url).then((response) => {
            resolve(response.data)
        }).catch(error => {
            reject(error)
        })
    })
}
```

Secondly, a list of news articles is retrieved from NewsAPI. Before continuing, it is important to note that the call to the NewsAPI is only done using the first movie title response (*newsapi.searchNewsAPI(response.results[0].original_title)*). The basis for this decision was made on concerns for the overall latency of the application, and of conserving the number of consecutive API calls. The function itself is just an *axios.get(url)* promise.

Following the fulfillment of the promise, once again the consuming code ensure the number of articles returned does not exceed the maximum before concatenating the articles into a single list of JSON objects (*relevant_articles = [{...}, {...}, {...}]*).

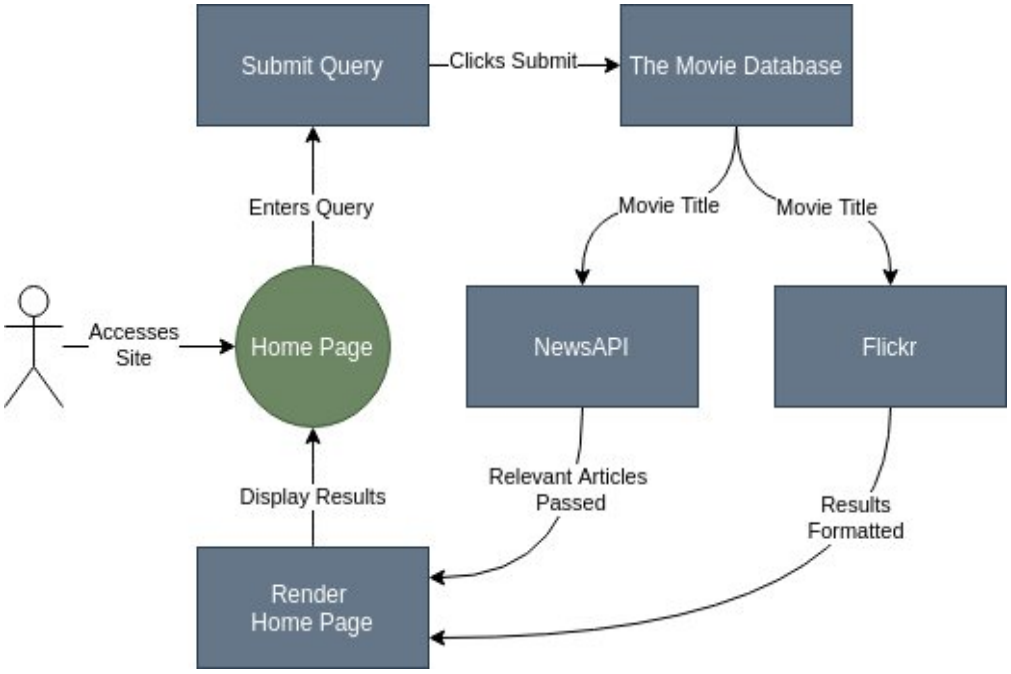
```
// Flickr API details
const options = {
    hostname: 'https://api.flickr.com',
    path: '/services/rest/?',
    method: 'flickr.photos.search',
    api_key: "82aff6fdd8d1593e6d5c3eaca5d090ce",
    format: "json",
    media: "photos",
    nojsoncallback: 1,
    number: 10
}

// Search Flickr for images based on query
async function searchFlickr(query) {
    return new Promise((resolve, reject) => {
        const url = options.hostname + options.path +
                    'method=' + options.method +
                    '&api_key=' + options.api_key +
                    '&tags=' + query +
                    '&per_page=' + options.number +
                    '&format=' + options.format +
                    '&media=' + options.media +
                    '&nojsoncallback=' + options.nojsoncallback;

        axios.get(url).then((response) => {
            // Return promise with JSON object containing, b_url, p_url, and photo title
            resolve(formatFlickrImagesURL(response.data))
        }).catch(error => {
            reject(error)
        })
    })
}
```

Once the articles relevant articles are retrieved, we see the last of the API promise-fulfillment functions finally occur. Once again the call to the Flickr API is only done with the movie title of the first response (*flickr.searchFlickr(response.results[0].original_title)*), for the same reasoning given prior. The *flickr.searchFlickr* function utilizes an *axios.get(url)* promise before pre-processing the response.

```
// Formet data from Flickr response and return JSON object
function formatFlickrImagesURL(data) {
    var photos = []
    for(i = 0; i < options.number; i++) {
        const photo = data.photos.photo[i];
        const b_url = "https://farm" + photo.farm + ".static.flickr.com/" + photo.server + "/" + photo.id + "_" + photo.secret + "_" + "b.jpg"
        const p_url = "https://www.flickr.com/photos/" + photo.owner + "/" + photo.id;
        const title = photo.title
        const json = {'b_url': b_url, 'p_url': p_url, 'title': title}
        photos[i] = json
    }
    return photos
}
```

The function, *formatFlickrImagesURL(response.data)*, pre-processes the response by looping through the number of defined results (set as 10), then building URLs to Flickr assets/pages. The core functionality of this code was re-purposed from the Flickr tutorial exercises earlier this semester. It builds the link to the uploaded image on Flickr (b_url), links to the publicly-facing page (*p_url*), and returns the photo title. Those variables are placed into a JSON object before being appended to a list. The function returns this list and finally resolves the Flickr promise with that object.

The consuming code of the Flickr promise contains the rendering code for the page. As such, it firstly updates the page counter using the same page counter function as in the GET root route; and secondly, it renders the page by passing lists containing the page title, the current page counter, the relevant articles, and the list of formatted JSON objects contained in the response from the consuming code from the Flickr promise.



Deployment and the Use of Docker

This application was successfully deployed with Docker throughout testing. In the prior section discussing the architecture of the whole application, I explained simplicity in its structure. Knowing this, it was unnecessary to deploy this application as a set of individual Docker images and utilize Docker Compose. Thus, only a simple Dockerfile was required. The Dockerfile uses Alpine Linux as its base image for its small, simple, and minimalistic nature. Compared to Ubuntu, the much smaller image size and fewer package dependencies sped up the process of building new Docker images. The rest of the Dockerfile follows the general process of building a NodeJS application in Docker: copy the project folder, apk update packages, apk install the required packages, set the WORKDIR to the copied project folder, export the running port, npm install the dependencies, and *npm start* the app.

The Dockerfile itself can be found in the appendix of this report.

Test Plan

Because of the scope of this assignment and the limited complexity of this application, a manual testing process was utilized for evaluation.

Task	Expected Outcome	Result	Screenshot Appendix #
Search for movie title from home page	Movie title results retrieved	PASS	1
Search for news articles based on movie title	Given valid query, NewsAPI returns results containing news articles related to the movie title	PASS	2, 3, 4
Search for Flickr images relating to movie title	Given valid query, Flickr returns related to the movie title	PASS	2, 3, 4
Display Flickr images on page	Flickr images are displayed in-line on home page	PASS	2, 3, 4
Click Flickr image	A working hyperlink to original photo	PASS	3
Display news article title, image, and description	News article divs are rendered as rows on the home page	PASS	2, 3, 4
Display and hyperlink to original news article	A working hyperlink to source article is displayed beneath description	PASS	4

Display Flickr photo title	Hovering the Flickr photo displays its title	FAIL	N/A
Page counter updated on S3	On each page GET/POST, page counter is incremented and updated on S3 bucket	PASS	5
Page counter displayed	Correctly display current page counter on home page	PASS	1, 2, 3, 4

Difficulties / Exclusions / Unresolved & Persistent Errors

Overall, I encountered few pressing issues throughout this assignment. Once the data was collected and the overall structure of the assignment was planned, integrating the individual APIs into a cohesive application was an uncomplicated affair.

The final assignment presented in this report differs from the initial project brief. The initial idea as nearly identical in concept to this assignment, but instead the queries would search for related music artists/songs, then searched NewsAPI for results based on the results. The tutor responded positively overall to the idea and agreed the use cases defined was consisted with the assignment criteria. However, they had concerns about the data that could be pulled from the Musicmatch API. With this feedback I shifted to The Movie Database API instead.

Presently, I do not believe there to be many, if any, outstanding bugs. At least none that have been encountered.

There can be issues encountered where Flickr images cannot be found given a particular movie title. In the future, additional error handling could be added to ignore the results gracefully render the page without the images. An example can be found in Figure 7 in the appendix.

Finally, the last "problem" has to do with the implementation of the persistence portion of this assignment. It is functional according to the given criteria. However, I encountered problems when I wrote a function to directly return the current count variable from its Amazon S3 bucket; the function, no matter how it was written, when called from within index.js in order to display on the page, would not work as intended. The workaround to this problem was to declare a local variable in the s3-counter.js file. This is seen in Figure 8 in the appendix.

Extensions

There are a several improvement points that can extend this application.

Improvement 1

When the movie titles are returned from The Movie Database, it can often contain movies that contain the same movie title. For example, a query for "back to the future" can return movie titles for "Back to the Future" as well as the sequels "Back to the Future 2" and "Back to the Future 3." A future improvement would be to identify such sequels and remove them from the returned movie titles. This ties into one of the current (and intentional) limitations of the app in its current form. Currently it only searches the title of the first returned movie in the NewsAPI and the Flickr API. The reasoning for this was explain previously.

Improvement 2

The application easily be extended to also search for TV shows relating to the original query. This would be best implemented as a user-selectable toggle: search for movies or search for TV shows.

Improvement 3

If the NewsAPI function was expanded to search for multiple movie titles at once, there's the possibility for duplicate news articles subsequently returned. Such duplicates could be compared and thus eliminated from the results.

Improvement 4

Another blatant improvement is the UI and styling of the home page for the app. Currently it has basic UI elements displayed. The reasoning for this was due to a lack of experience working with the Pug and its limited documentation.

Improvement 5

Finally, it would be useful to implement a future to allow the user to filter the total number of results desired to be displayed on the page. This would allow the user a greater degree of control over their search results.

User Guide

The usage of this application is very rudimentary and straightforward. After the page has been loaded, the user can enter their desired movie query into the search box and either click submit or click *enter* on their keyboard. Several screenshots displaying the usage of the application are displayed in the appendix.

Analysis

Question 1: Vendor Lock-in

This mash-up as it stands is highly dependent on the intermingling of the API results. Whilst it would be inconvenient, it would still be possible to replace the individual APIs to an alternative.

The first API called in this application is The Movie Database API. The functionality it provides is rudimentary and straightforward to replace with another API should it be required. Currently, TMDB is only needed to return the movie title based off an original query. This functionality could be replicated by the IMDB API. The JSON response from it includes a list of movies and TV shows, and each object contains the movie title.

https://imdb-api.com/API

NewsAPI as it stands could almost completely replaced by a service implementing the Google News API. The same information currently displayed from NewsAPI is present in a JSON response.

https://newscatcherapi.com/google-news-api

There exists a multitude of alternatives that can replace the functionality provided by the Flickr API in this application. The Flickr API as used here only return the original photo page, a link to the

Likewise for the persistence service, it would be possible to swap it to another offering that can achieve the same functionality required for the page counter. For example, swapping from an currently using an object in an Amazon S3 bucket to a simple table in Amazon DynamoDB.

https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html

Question 2: Container Deployment

In content of this application, I do not believe there to be any disadvantages to the use of container-based deployment. As it currently stands, this assignment had non-substantial scope compared to a full web application deployment. Additionally, the resource load of running this application is low, so it would be impractical and unnecessary to deploy an individual VM for each running instance.

The use of container-based deployment allows for greater speed and more granular control the deployment of software on a system. Compared to full virtual machines, containerized services lack the code for an entire, complete operating system. Instead, they contain a small “guest” OS image that can negotiate the system’s resources as needed that are required to run an individual application or back-end service. This overall leads to containers have significantly lower overhead compared to running beneath a full hypervisor (whether it be a Type 1 or Type 2) and a resource-heavy guest OS.

A more complex application with multiple moving components has many advantages utilizing containers from a life cycle management perspective, provided the applications themselves are stateless and the persistence is reliable. An updated version of an existing containers could be “dropped in” with near zero noticeable downtime.

If a more complicated version of this application would be deployed at scale, a couple categories of persistence would be considered. For example, using a relational database for storing user credentials, S3 to pull/push Blob stores for other persistent objects, shared in-memory caches to store the results from recent searches, and CDN services to store static website content.

Appendix

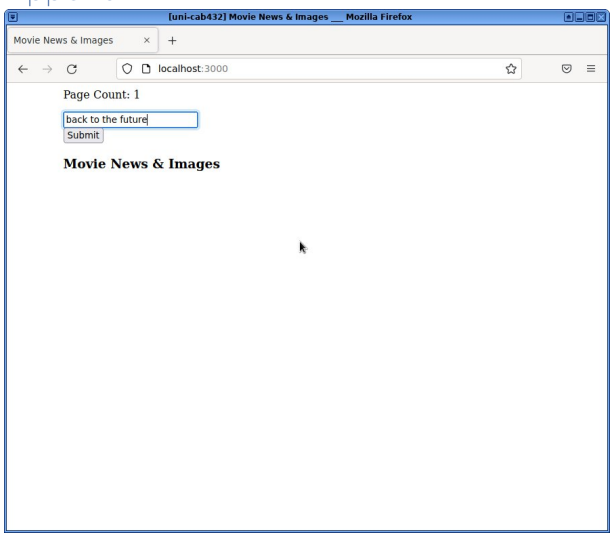


Figure 1 Search for movie title

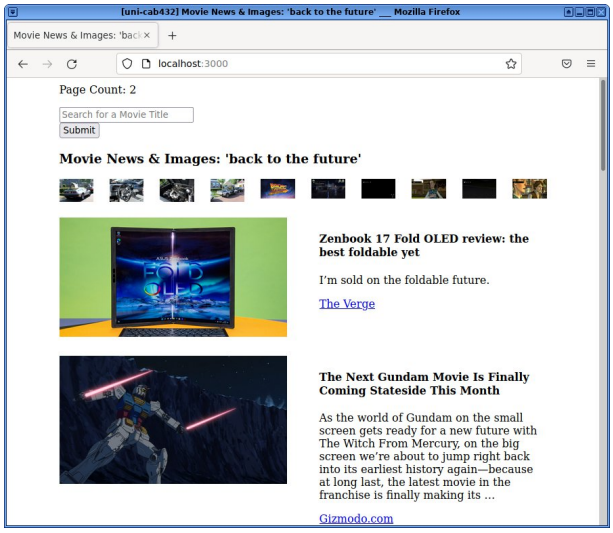


Figure 2; All elements correctly displayed (Flickr, News Articles, page count updated)

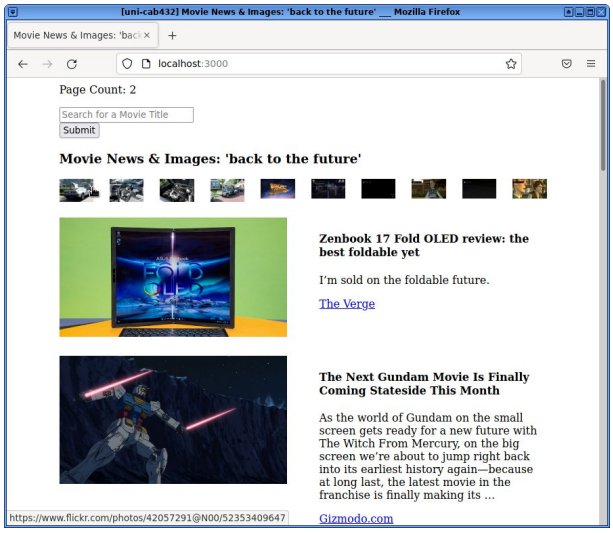


Figure 3 Working hyperlink to Flickr image page

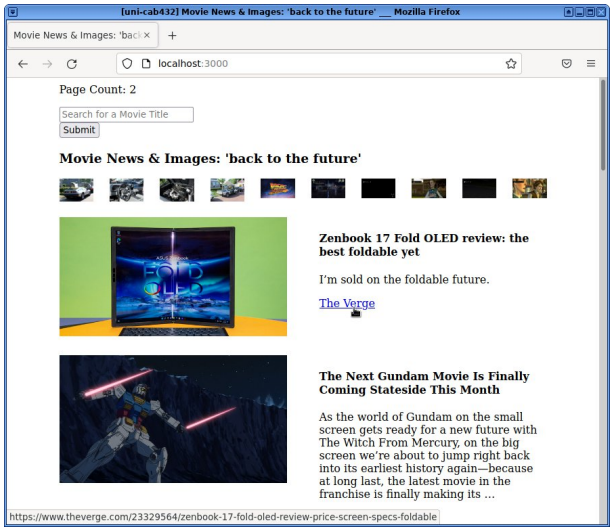


Figure 4 Working hyperlink to article page

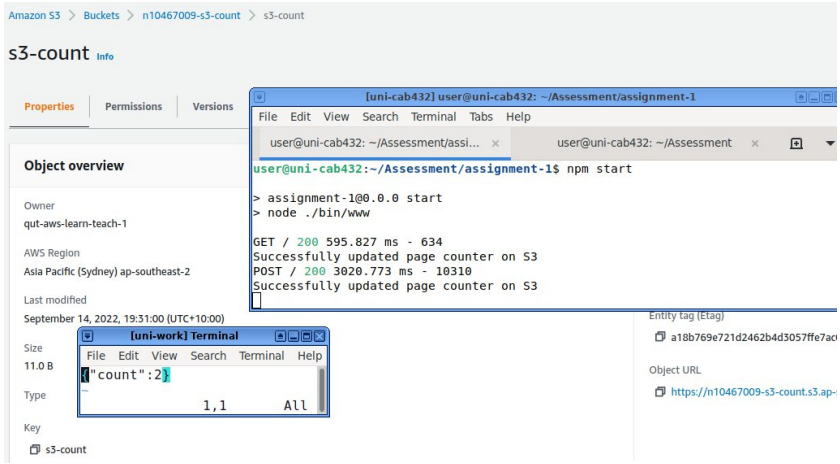


Figure 5 Page count updated on Amazon S3 bucket

```
FROM alpine:latest
LABEL MAINTAINER "Jack Muir"
COPY assignment-1 /app
RUN apk update && apk add --no-cache nodejs npm
ADD assignment-1 /app
WORKDIR /app
EXPOSE 3000
RUN npm install
RUN npm install aws-sdk
CMD npm start
```

Figure 6 Dockerfile

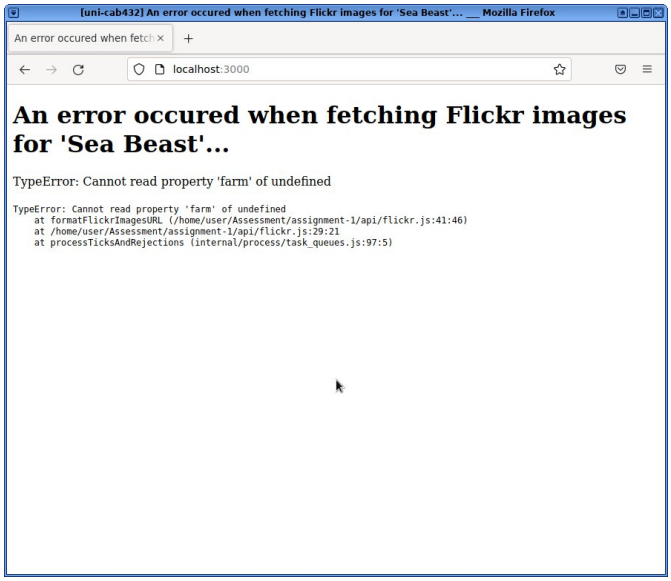


Figure 7 Error fetching Flickr images

```
// Site counter object
const s3Count = {
    count: 0
}

// Criteria-adherent (?) workaround to issues returning data from S3 bucket and displaying as variable
let count = 0;

function getCount() {
    return count;
}

var newDeployment = true

// Update the local variable page count and update on S3
function s3UpdateCounter() {
    if(newDeployment) {
        newCount = s3Count
        newCount.count++
        const body = JSON.stringify(newCount)
        // Update S3 object
        s3.putObject({Bucket: s3Bucket, Key: s3Object, Body: body}, function(err, data) {
            if(err) {
                console.log(err.message)
            } else {
                console.log("Successfully updated page counter on S3")
                newDeployment = false
            }
        })
    } else {
        s3.getObject({Bucket: s3Bucket, Key: s3Object}, (err, result) => {
            if(result) {
                const resultJSON = JSON.parse(result.Body)
                const newResultJSON = resultJSON
                newResultJSON.count++
                const body = JSON.stringify(newResultJSON)

                // Update S3 object
                s3.putObject({Bucket: s3Bucket, Key: s3Object, Body: body}, function(err, data) {
                    if(err) {
                        console.log(err.message)
                    } else {
                        count = newResultJSON.count
                        console.log("Successfully updated new page counter on S3")
                    }
                })
            }
        })
    }
}
```

Figure 8 Persistence implementation