GMIT
INSTITIÚID TEICNEOLAÍOCHTA NA GAILLIMHE-MAIGH EO
GALWAY-MAYO INSTITUTE OF TECHNOLOGY

Metal Hub Éire

James Curley

Bachelor of Engineering (Honours) in Software and Electronic Engineering

Galway-Mayo Institute of Technology

2019/2020

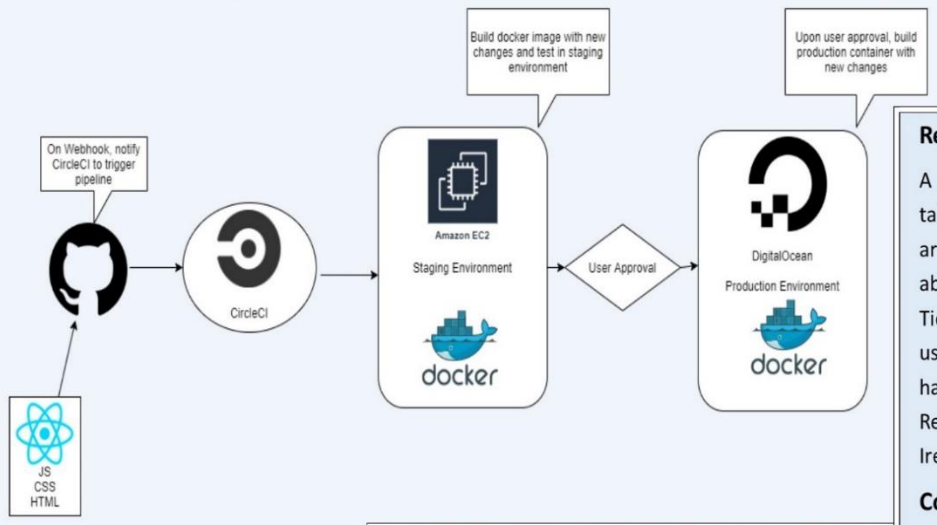
Metal Hub Éire

A Full Stack web application containing up to date metal music oriented concert listings, podcasts and news articles with the ability to check for new content at the click of a button.

GMIT
INSTITIÚID TEICNEOLAÍOCHTA NA GAILLIMHE-MAIGH EO
GALWAY-MAYO INSTITUTE OF TECHNOLOGY

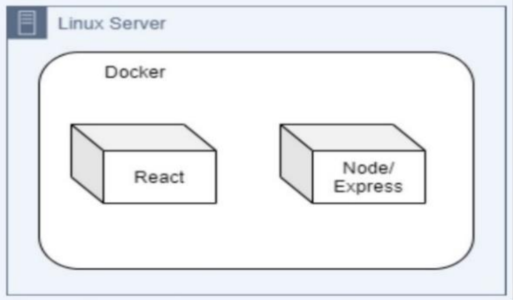
Web Scraping & Database

Researching web scraping techniques led me to pythons "BeautifulSoup" library. HTML is parsed for specific elements and class names and values are extracted and stored in an array. A postgres connection is established and the array is converted to a table and stored in the database. A script is written for each URL to be scraped. The scripts are ran by a Jenkins server on the production VM triggered by the user from the frontend.


On Webhook, notify CircleCI to trigger pipeline
JS
CSS
HTML
CircleCI
Build docker image with new changes and test in staging environment
Amazon EC2
Staging Environment
docker
User Approval
Upon user approval, build production container with new changes
DigitalOcean
Production Environment
docker

Containerisation

Docker allows the app to be created and deployed to any environment. Services like react, nginx and express are ran in parallel in separate containers in complete isolation.



Design and Implementation

Frontend: Created with React JS using modern technologies like react hooks for altering state and lifecycle and component libraries such as reactstrap and material-ui.

Backend: NodeJS server using express to handle requests to the database and implement CORS security measures for comms between front and back end.

Continuous Deployment: A CD Pipeline created using Circle CI detects merges to the master branch and triggers a workflow deploying the new changes to an AWS Staging VM. User approval is required to move to the next stage, deployment to a Production VM.

Result

A functional web app displaying tables of concert listings, news articles and podcasts with the ability to check for new content. Ticketmaster's discovery API was used to fetch listings of upcoming hard rock/metal events in the Republic of Ireland and Northern Ireland.

Conclusion

The purpose of this project was to become proficient in using some of the most popular full stack tools and to successfully research, design and implement a highly scalable web application from scratch and to create a functional pipeline to allow seamless integration of new features while delivering zero down time deployments.

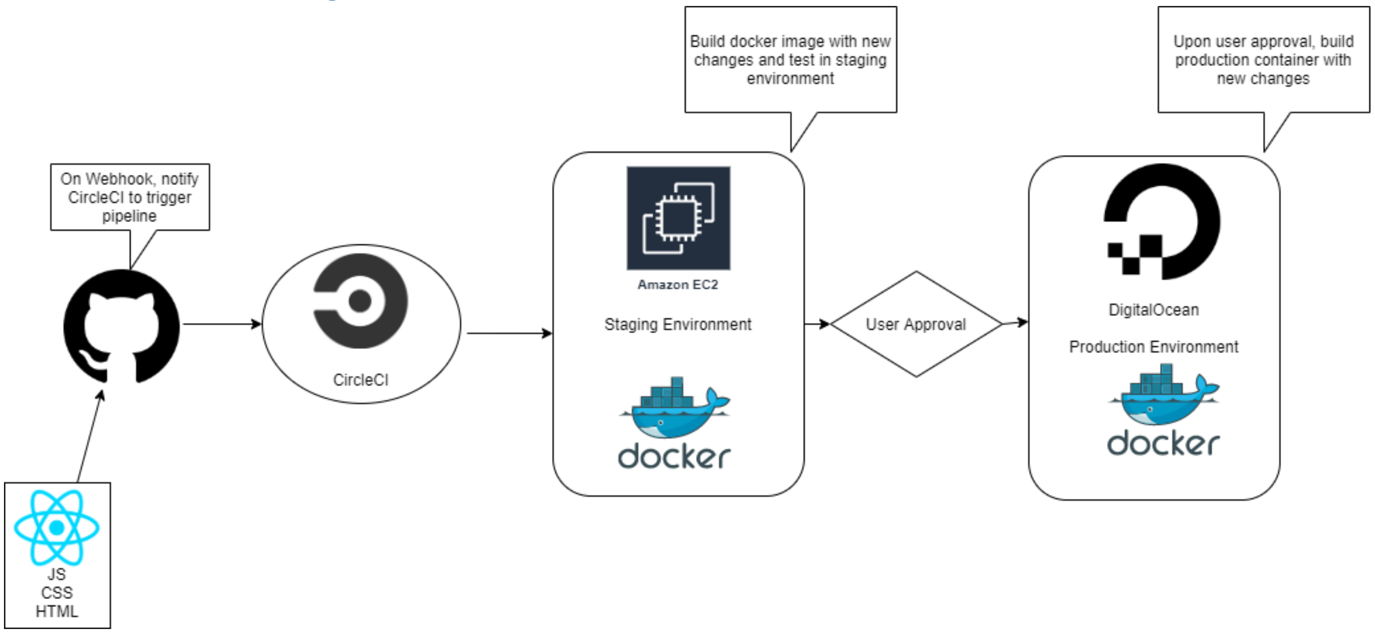
Links:

Metal Hub Éire

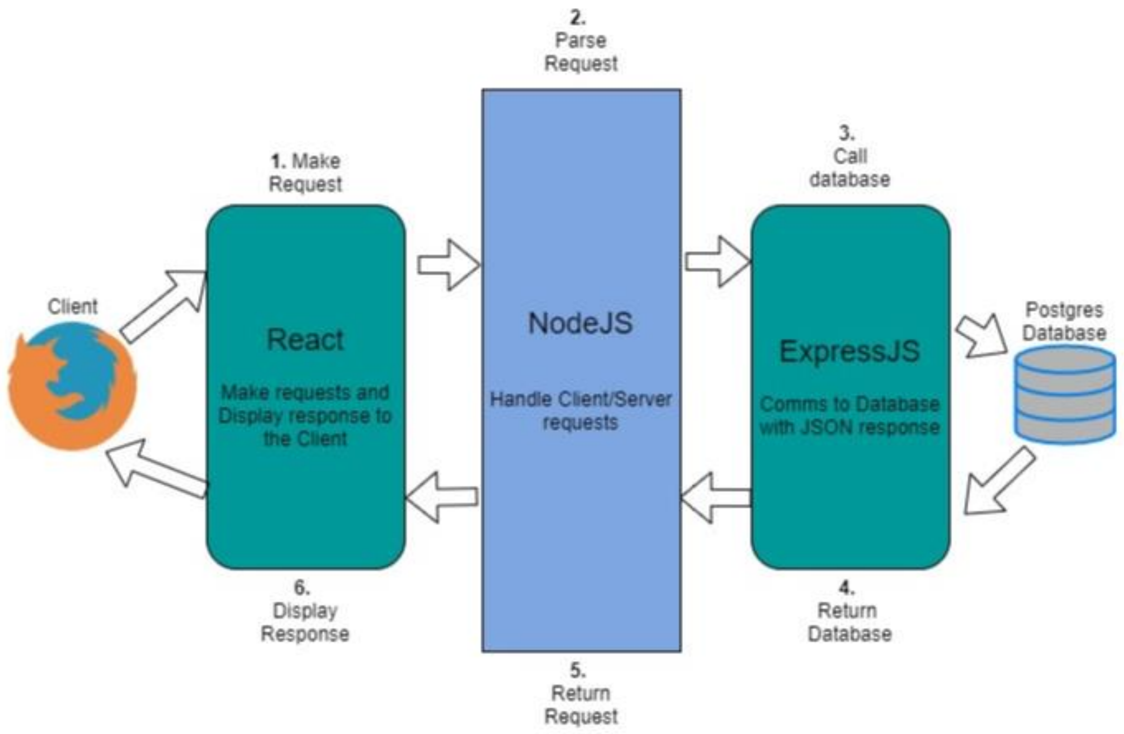
GitHub

Web Scraper Scripts

CD Flow Architecture Diagram


On Webhook, notify CircleCI to trigger pipeline
JS
CSS
HTML
CircleCI
Build docker image with new changes and test in staging environment
Amazon EC2
Staging Environment
docker
User Approval
Upon user approval, build production container with new changes
DigitalOcean
Production Environment
docker

PERN Stack Architecture Diagram



Declaration

This project is presented in partial fulfilment of the requirements for the degree of Bachelor of Engineering in Software & Electronic Engineering at Galway-Mayo Institute of Technology.

This project is my own work, except where otherwise accredited. Where the work of others has been used or incorporated during this project, this is acknowledged and referenced.

James Curley

____

Acknowledgements

Mairtin O'Conghaile – My Project Supervisor, for encouragement and advice throughout the course of the project.

Medium.com – For an abundance of brilliant tutorials on React, Docker, CD, Postgres, PERN stack.

DigitalOcean.com – For some fantastic tutorials and free webinars on Ubuntu server setup, Nginx, Docker for Ubuntu.

Docker.com – For providing excellent free webinars and documentation on docker tutorials.

Table of Contents

1 Summary

The main goal of this project was to design, develop and deploy a highly scalable web application with a solid infrastructure behind it to allow User acceptance testing followed by seamless deployments.

The application I created is called Metal Hub Éire and it acts as a hub for all the metal music related content that I, and metal music fans in Ireland like me read daily. You could think of it as an app like 'Skyscanner', where the user checks for data they want on one application, but the results displayed are collected from multiple external sources.

The data is collected using Pythons BeautifulSoup web scraping library and stored in an SQL database. The Frontend is created using React with NodeJS for Backend. ExpressJS handles communications between front and back end. Jenkins CI provides a means to run my web scraper Python scripts Asynchronously. CircleCI is used to create a 2-stage deployment pipeline that first deploys to a staging VM where User Acceptance Tests can be performed and if the tests pass, the app is deployed to a Production VM using docker to containerise express, react and Nginx.

1.1 Project Scope

This project will consist of the research, design, implementation and evaluation/conclusion of building *Metal Hub Éire.* The App will contain tabular data of concert listings from two sources, Ticketmaster's Hard Rock/Metal events listings and DME (Dublin Metal Events) Listings as well as a table of news articles scraped from Blabbermouth.net consisting of 100 news articles dated from newest to oldest and a table containing podcasts scraped from *The Metal Cell Podcast* from fireside FM's website. This table will consist of numbered podcasts sorted by date, specifying the duration of each podcast and a button which opens a modal to listen to the podcast.

1.2 Technologies Used

Data : My database is populated with data collected using Pythons BeautifulSoup Library and the Python Psycopg2 postgres connector for Python.

PERN Stack – Postgres, Express, React, Node.

I chose this stack for a few reasons. I wanted to use an SQL database system rather than a document-oriented database program because we were beginning MySQL in college, so I used Postgres so I could gain experience using both SQL systems.

I had been researching React vs Angular for Web Frameworks and because I had used Angular before I decided to try React for something new.

CI Tools : I use Jenkins to asynchronously run my scripts and CircleCI for the deployment pipeline.

Containerisation : I use docker in both Development and Production to containerise the front and back ends of the project.

Cloud Services : I use both DigitalOcean and AWS Elastic Compute 2 to host my Ubuntu servers. AWS for a Staging Virtual Machine to perform User Acceptance Tests (UAT) and DigitalOcean for a production Virtual Machine.

1.3 What was Accomplished

- A fully functioning web application accessible via: http://206.189.165.104:3000/
- Proficiency in using the technologies mentioned above. The confidence to create PERN stack applications from research up to deployment.
- A fully functional application deployed to a production environment on an Ubuntu Virtual Machine.
- A framework for future development of the application, a pipeline that provides seamless integration of future releases. A sandbox or 'Staging' environment for User Acceptance Testing (UAT) on an EC2 Ubuntu Virtual Machine prior to deploying to production.

2 Introduction

2.1 Problem Statement

The inspiration for this project came from a problem I've noticed ever since I stopped using Facebook. As a music enthusiast one of the main reasons I used Facebook was to keep up to date with upcoming events and music news. The problem is having to scroll through tons of useless information and videos of no interest just to see a buried post about a new concert announcement or a news article announcing a new tour or album release. Obviously visiting each site every day to check for content would be the best way of keeping up to date however having all this in one location would be the most efficient solution.

2.2 Solution

I thought the idea of having a hub that contains concert listings, news and podcasts in one place would be a more efficient replacement for Facebook giving me the information I'm interested in without scrolling mindlessly for hours on end. However, the solution would have to guarantee the data was correct and up to date to be more practical than visiting the individual source sites.

Like most web applications scalability is important here. It must be designed in such a way that new features can be easily integrated without rewriting or refactoring existing code. I also want new releases to be integrated to production with zero downtime.

This report will go through the steps and challenges I faced whilst creating this application in the same order as I completed it.

3 Research

3.1 Web Scraping

The first step was to decide how I could collect data from a website. Using a website's official API is by far the best way of extracting the data that you want from a webpage but the site's I will be using aside from Ticketmaster, do not provide API's for public use.

Python's 'BeutifulSoup' web scraping library is the next best thing. This was a perfect option for me as I had no experience with Web Scraping or even Python prior to this project so it required a steep learning curve. Web Scraping is also becoming popular in areas like machine learning and AI so it's a great time to learn it.

It works by performing a GET request to a URL using the 'requests' library. The response is the entire page's HTML. This response can then be parsed for the necessary data and stored in an array.

A Python script can be written for each source I want to scrape.

3.2 Data Storage

MySQL vs PostgreSQL vs MongoDB

I had decided earlier on that I was going to use an SQL database as I was just learning MySQL for the first time around the same time this project began, so that eliminated Mongo. After trawling a few comparison sites, Postgres came out on top. Slightly faster read speeds & faster handling of complex queries sold it for me.

3.3 Cloud Services

DigitalOcean vs AWS

This was going to be a big area as I needed everything apart from the development environment to be cloud based. I chose DigitalOcean as my main cloud service provider. This decision was based simply on the price plans DigitalOcean offers vs. AWS. While both offer free credits for students, a DigitalOcean droplet costs $5 a month after the credits run out whereas AWS pricing is difficult to calculate which made me wary of varied monthly charges depending on usage. Because I want to keep developing the app well after the credits run out, I went with the DigitalOcean option.

I do use AWS as well for my staging environment which I will explain further on.

3.4 Web Application Frameworks

Angular vs React

Deciding what Framework to use for my front end was a decision that required more thought. I had previous experience with Angular which is written with TypeScript. However, I had no experience using JavaScript and It was a language I've wanted to learn for a while. One main advantage of using Angular seemed to be better methods for form validation, however this didn't apply to me as I wouldn't be using forms. Between Angular, React and Vue, react had the medium learning curve which suited me as this was only one aspect of the project I can't devout all my time to it. It is also the most sought-after framework of 2019 according to indeed.com, so I settled for ReactJS.

"As of this writing in August 2019, Indeed.com lists nearly 15,000 jobs requiring React.js skills in the US. There are 13,500 Angular-related jobs and 2,200 Vue.js-related jobs. Of the three frameworks, React.js may be the best skill to prioritize for both engineering team leads and developers wishing to expand their web development skills [1]"

3.5 CI Tools

Jenkins vs CircleCI

Jenkins

I would need CI tools to serve two purposes.

1. To run my web scraping Python scripts either on a scheduler or on some trigger and provide a build log.
2. For Deployment.

For the first requirement, Jenkins is the only free tool that provides an API token to allow a build to be triggered remotely which might come in handy.

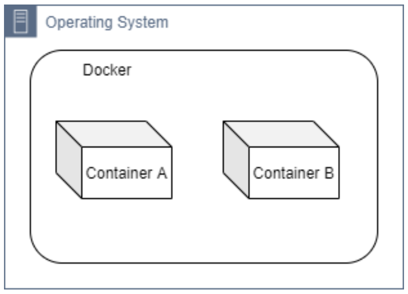
EDIT: As of April 2020 CircleCI, released API V2 which does allow remote triggering among other new API endpoints.

For Deployment, both options will provide pipeline features for deploying but CircleCI benefits by not requiring a dedicated server and is more lightweight as it doesn't rely on third party plugins to hook into GitHub, send failure emails etc. The downside is that a basic (free) account allows up to 2500 credits per week with the average workflow costing between 30-60 credits. You can also run only one job at a time, so this is something to consider especially if you require running jobs in parallel. However, these caveats won't affect this project and because It's always good to try something new I went with CircleCI for deployment and Jenkins for running my scripts.

3.6 Containerisation

Containerisation is widely used not only by major software and IT companies, but even by companies like the financial service firm, JP Morgan & Chase and Business MGMT Consultant Neudesic [2]. It provides a deployment solution by packaging and installing an application in an operating system-less container isolated from the rest of the host OS. It is essentially a virtual machine without the operating system so the configuration of the host OS will not affect the container. This way the application can be replicated anywhere it is deployed.

Docker is a full stack container management suite which is responsible for the creation, deployment and management of containers**.** I use docker in a local development environment, a staging environment and a production environment. A Dockerfile defines the steps involved in creating the container including installing the application and all dependencies. With Docker Compose, a yaml file can run multiple Dockerfiles. This way we can run our packages as services in parallel using multiple ports. This is the approach I will take for running react and node in separate containers. I will break down the Dockerfile and docker-compose yaml file and explain each command in chapter 6.



Container architecture.

3.7 Other Tools and IDE’s used

- Pycharm: IDE for Python development by Jetbrains
- Visual Studio Code: IDE for React (JavaScript/HTML/CSS) development
- MobaXterm: SSH client for remote computing
- Postman: For testing API Endpoints
- Heidi: SQL tool that provides a UI for viewing/manipulating databases
- Git: For version control

4 Design

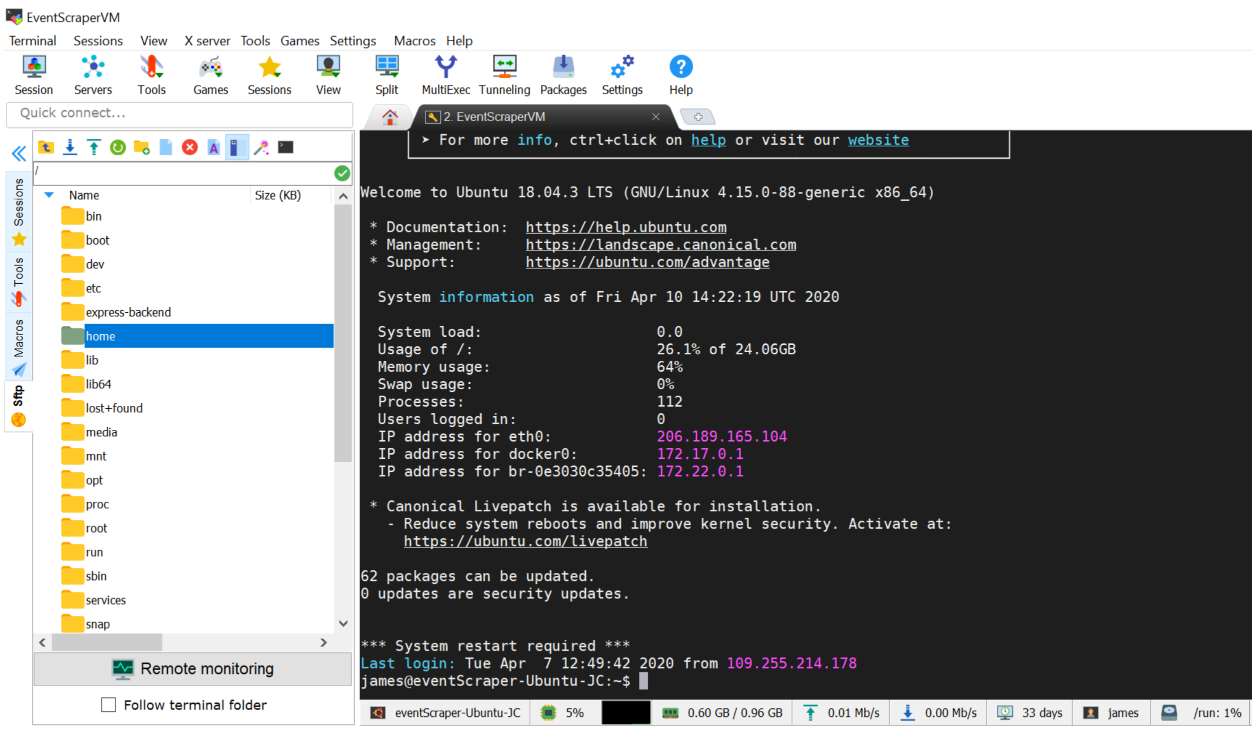
This chapter involves three sections.

(I) **Data**: Obtaining the data and populating the database and how to automate the process of checking for new data.
(II) **Client**: The PERN stack's react based frontend.
(III) **Server**: The PERN Stack's Express/NodeJS backend.

Note: For this project I am using **two** GitHub repositories. One for my front and back end and one for the Python based web scraping scripts.

4.1 Data

My database is located on my DigitalOcean droplet which I access through MobaXterm which provides a colourful terminal with a clickable linux directory that makes directory navigation quicker. It also displays IP addresses logged in including the docker containers IP as well as memory, disk and CPU usage.


EventScraperVM
Terminal Sessions View X server Tools Games Settings Macros Help
Session Servers Tools Games Sessions View Split MultiExec Tunneling Packages Settings Help
Quick connect...
2. EventScraperVM
Sessions Tools Macros Sftp
Name Size (KB)
bin
boot
dev
etc
express-backend
home
lib
lib64
lost+found
media
mnt
opt
proc
root
run
sbin
services
snap
Remote monitoring
Follow terminal folder
eventScraper-Ubuntu-JC 5% 0.60 GB / 0.96 GB 0.01 Mb/s 0.00 Mb/s 33 days james /run: 1%

```
 ➤ For more info, ctrl+click on help or visit our website

Welcome to Ubuntu 18.04.3 LTS (GNU/Linux 4.15.0-88-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Fri Apr 10 14:22:19 UTC 2020

  System load:                      0.0
  Usage of /:                       26.1% of 24.06GB
  Memory usage:                     64%
  Swap usage:                       0%
  Processes:                        112
  Users logged in:                  0
  IP address for eth0:              206.189.165.104
  IP address for docker0:           172.17.0.1
  IP address for br-0e3030c35405:   172.22.0.1

 * Canonical Livepatch is available for installation.
   - Reduce system reboots and improve kernel security. Activate at:
     https://ubuntu.com/livepatch

62 packages can be updated.
0 updates are security updates.

*** System restart required ***
Last login: Tue Apr  7 12:49:42 2020 from 109.255.214.178
james@eventScraper-Ubuntu-JC:~$
```

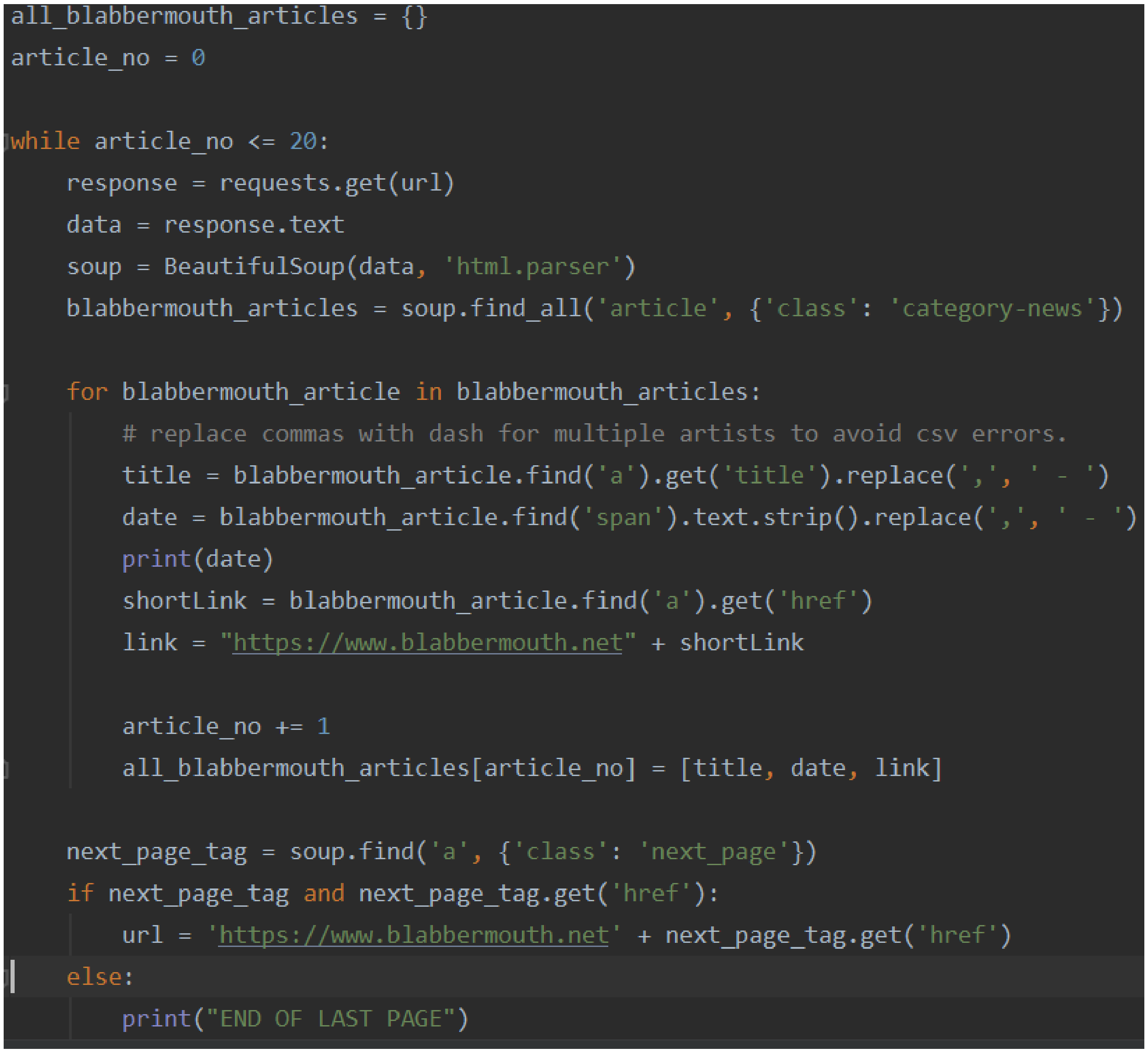
Server view after logging in

4.1.1 Web Scraping Script breakdown

I learned the basics of web scraping by following this short free Udemy course which was also my introduction to python.

Here I will break down the script that scrapes blabbermouth.net news articles. The other scripts are similar with different parsing methods used depending on the incoming data.

Step 1: Get the data



```
all_blabbermouth_articles = {}
article_no = 0

while article_no <= 20:
    response = requests.get(url)
    data = response.text
    soup = BeautifulSoup(data, 'html.parser')
    blabbermouth_articles = soup.find_all('article', {'class': 'category-news'})

    for blabbermouth_article in blabbermouth_articles:
        # replace commas with dash for multiple artists to avoid csv errors.
        title = blabbermouth_article.find('a').get('title').replace(',', ' - ')
        date = blabbermouth_article.find('span').text.strip().replace(',', ' - ')
        print(date)
        shortLink = blabbermouth_article.find('a').get('href')
        link = "https://www.blabbermouth.net" + shortLink

        article_no += 1
        all_blabbermouth_articles[article_no] = [title, date, link]

    next_page_tag = soup.find('a', {'class': 'next_page'})
    if next_page_tag and next_page_tag.get('href'):
        url = 'https://www.blabbermouth.net' + next_page_tag.get('href')
    else:
        print("END OF LAST PAGE")
```

(i) Make a GET request to the URL using the requests library

(ii) An array and a counter are declared to store and count each data item

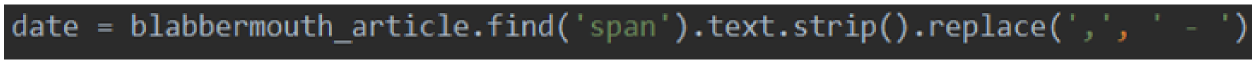
(iii) I only want 100 articles to improve execution speed but usually we want to loop while true

(iv) We start at the outer most HTML div that still contains all the data needed. So, within ‘category-news’ lie the title, date and link to each news article.

(v) The Items are then added to the array

(vi) We check for a next page URL, in this case ‘/page2’ and append it to the original URL and loop again until the last page

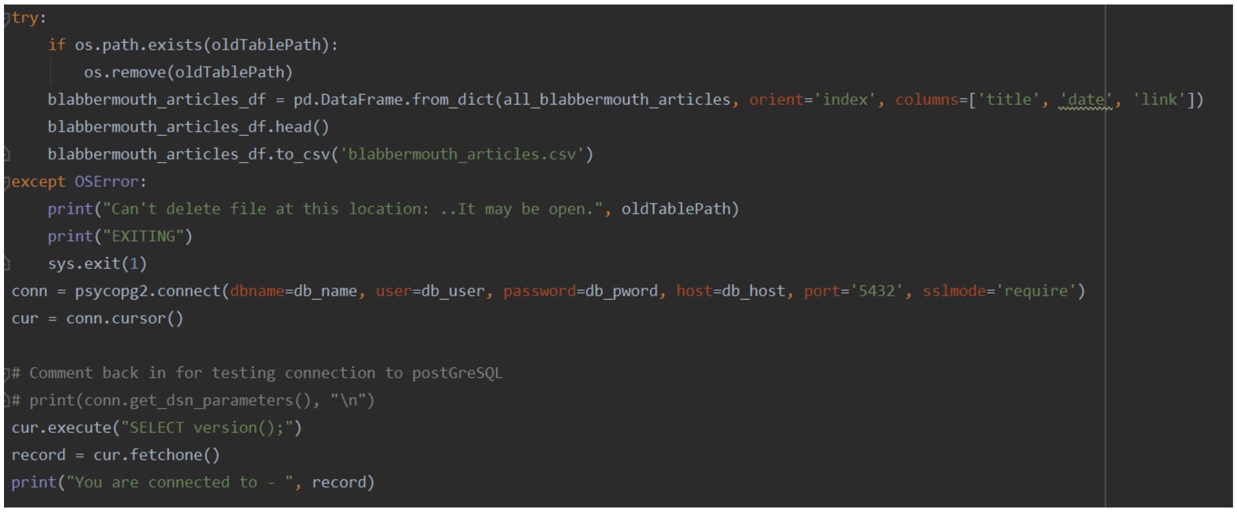
Note:



```
date = blabbermouth_article.find('span').text.strip().replace(',', ' - ')
```

Here we get the article date from between 2 span tags. It prints a blank line after the date so we use the ‘strip’ method to remove whitespaces and replace commas with dashes because this data will end up in a csv file and a comma will be read as an extra column.

Step 2: Create dataframe and connect to database



```
try:
    if os.path.exists(oldTablePath):
        os.remove(oldTablePath)
    blabbermouth_articles_df = pd.DataFrame.from_dict(all_blabbermouth_articles, orient='index', columns=['title', 'date', 'link'])
    blabbermouth_articles_df.head()
    blabbermouth_articles_df.to_csv('blabbermouth_articles.csv')
except OSError:
    print("Can't delete file at this location: ..It may be open.", oldTablePath)
    print("EXITING")
    sys.exit(1)
conn = psycopg2.connect(dbname=db_name, user=db_user, password=db_pword, host=db_host, port='5432', sslmode='require')
cur = conn.cursor()

# Comment back in for testing connection to postGreSQL
# print(conn.get_dsn_parameters(), "\n")
cur.execute("SELECT version();")
record = cur.fetchone()
print("You are connected to - ", record)
```

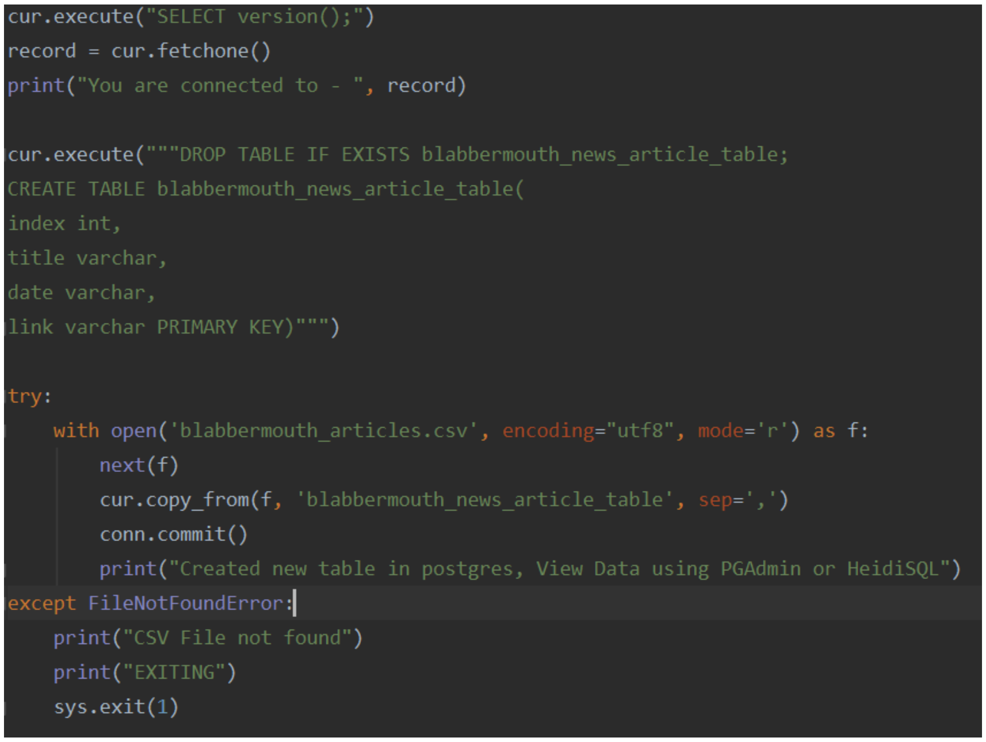
(i) If old csv exists, remove it. Create dataframe from array using the ‘pandas’ library to put items into a tabular format

(ii) Create csv file

(iii) Psycopg2 is Python's Postgres connector and gets passed the credentials to my database which are stored as environment variable to keep them hidden from GitHub.

(iv) The cursor method allows us to perform SQL commands.

Step 3: Create New Table



```
cur.execute("SELECT version();")
record = cur.fetchone()
print("You are connected to - ", record)

cur.execute("""DROP TABLE IF EXISTS blabbermouth_news_article_table;
CREATE TABLE blabbermouth_news_article_table(
index int,
title varchar,
date varchar,
link varchar PRIMARY KEY)""")

try:
    with open('blabbermouth_articles.csv', encoding="utf8", mode='r') as f:
        next(f)
        cur.copy_from(f, 'blabbermouth_news_article_table', sep=',')
        conn.commit()
        print("Created new table in postgres, View Data using PGAdmin or HeidiSQL")
except FileNotFoundError:
    print("CSV File not found")
    print("EXITING")
    sys.exit(1)
```

(i) We print if connection is established, this comes in handy when troubleshooting Jenkins build logs later.

(ii) Finally, we create a table, copy in the contents of the csv file and exit the cursor.

4.1.2 Web Scraper automation Architecture

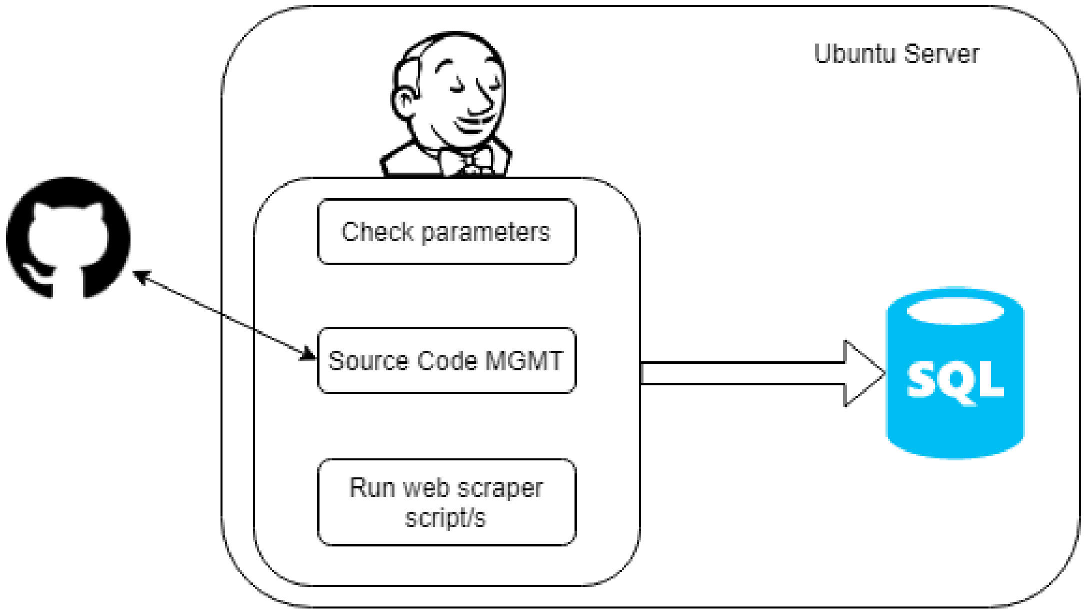
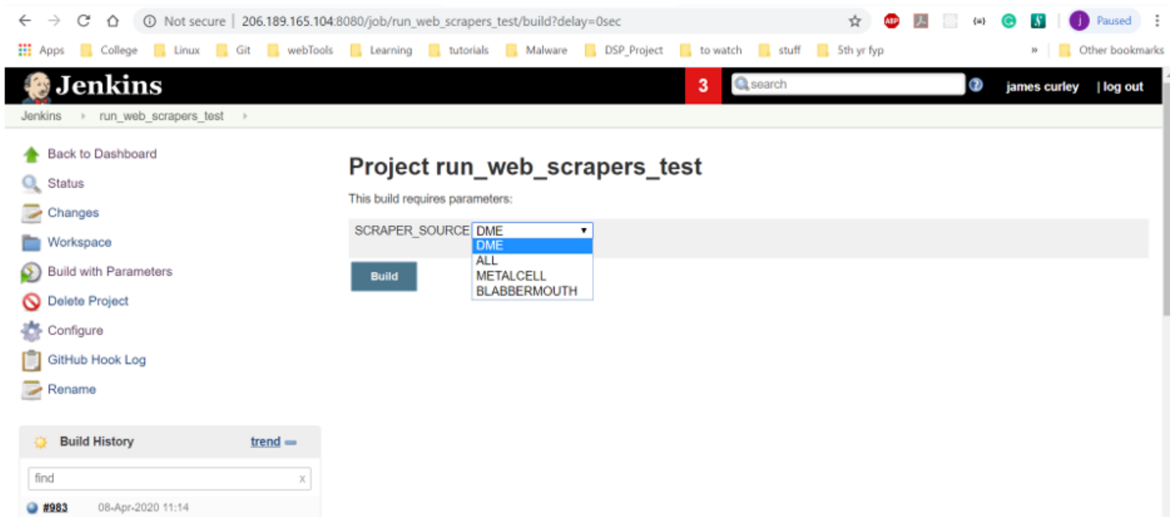


Fig 1. Jenkins installed on Ubuntu Server

I created a parameterized job on Jenkins. It has a multiple-choice parameter as seen in Fig 2 to select which (or All) python script to run. This parameter value can be passed in an API call to trigger a remote build using a Jenkins API token which we will make use of later.



Project run_web_scrapers_test

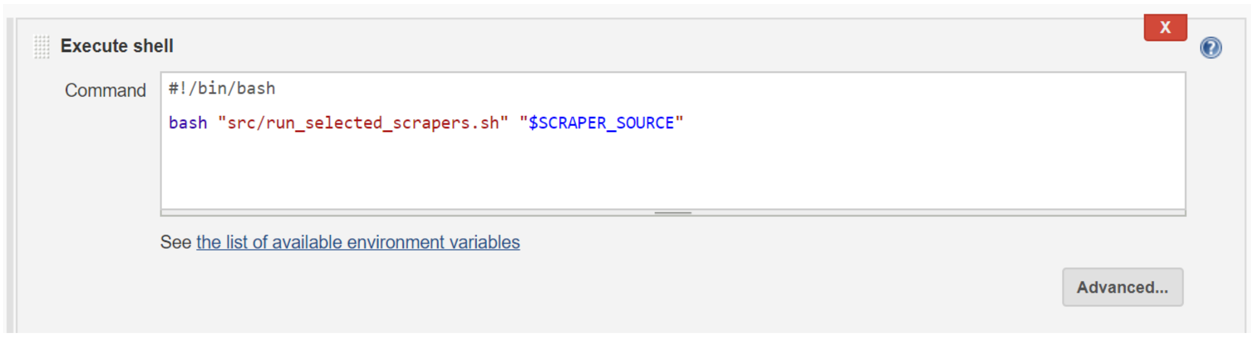
This build requires parameters:

SCRAPER_SOURCE DME

DME
ALL
METALCELL
BLABBERMOUTH

Fig 2.

I have specified my GitHub master branch in source control management, so Jenkins has access to my repo. Jenkins features execute shells which run shell scripts as seen in fig 3, so I pass the build parameter into a shell script named run_web_scrapers.sh which uses if/else conditionals to select the script to be ran. The script runs and the database is updated.



```
#!/bin/bash
bash "src/run_selected_scrapers.sh" "$SCRAPER_SOURCE"
```

Fig 3.

4.2 Client

This is the User Interface (UI) of the application created with Facebook's React. This is what the user see's and can interact with. It is made of independent, reusable blocks of code called **Components**. Components follow the Singular Purpose Principle. A component is a JavaScript class or function that can take in inputs known as 'props' and return a react element that determines how the UI should look.

Class vs Functional Components

Class components use different syntax, requiring you to extend *Component*. You are also required to create a *render* function before returning an element. The render function allows you to alter components state and lifecycle.

Functional components or *Stateless Components* originally were just plain JavaScript functions so you could not use the setState() method. The setState() method is used for example, to turn on/off a loading icon or disable a button after a press. So, if you wanted to change state you were required to use classes.

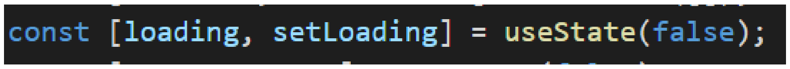
React Hooks

With the release of React 16.8 In March 2019, Hooks were introduced. These allowed the use of setState() among other state related features in functional components reducing the need to write class components. This means simple code sharing between components using custom hooks which in turn helps with rendering speed. Hooks are being gradually adopted in React to replace classes.

I have written most of my application using functional components making use of react hooks.

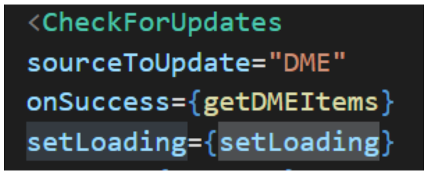
Here is an example of the useState hook.

A state item is created and initialised to false.



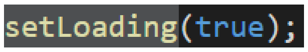
```
const [loading, setLoading] = useState(false);
```

The item is passed as a prop to a component called 'CheckForUpdates'.



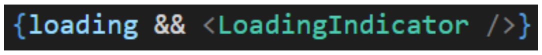
```
<CheckForUpdates
sourceToUpdate="DME"
onSuccess={getDMEItems}
setLoading={setLoading}
```

Here it can change state to true.



```
setLoading(true);
```

The state hook picks up the state change and calls a method called "LoadingIndicator" which displays a spinner while a function executes.



```
{loading && <LoadingIndicator />}
```

Props, which stands for ‘properties’ are inputs, similar to function parameters and can be passed between components to hold a value or change state. setLoading from the example above is an example of a prop.

node JS

expressjs

4.3 Server

The server’s API communicates to the database through a RESTful architecture. The responses are in the JSON format.

4.3.1 Express

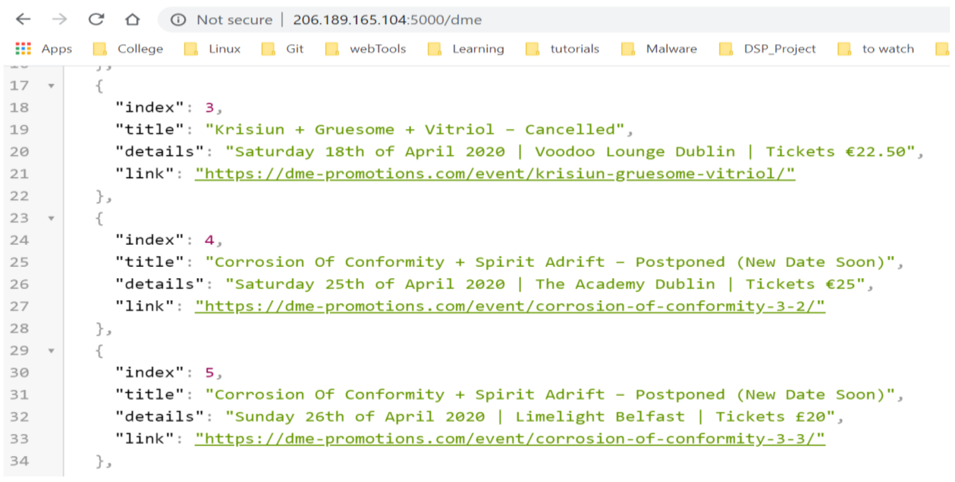
Express is a structured web framework that handles multiple requests to the same URL. It allows you to write responses to specific URL paths.

For example:

http://206.189.165.104:5000/dme

http://206.189.165.104:5000/metalcell

So, for my application I will have a page to display data from each web scraper source and append it to the URL path. The default express port number is 5000 which I haven’t changed so to view the JSON response of a query you can navigate to http://<domain>:5000/<queryPath> which will display like this:



```
  {
    "index": 3,
    "title": "Krisiun + Gruesome + Vitriol – Cancelled",
    "details": "Saturday 18th of April 2020 | Voodoo Lounge Dublin | Tickets €22.50",
    "link": "https://dme-promotions.com/event/krisiun-gruesome-vitriol/"
  },
  {
    "index": 4,
    "title": "Corrosion Of Conformity + Spirit Adrift – Postponed (New Date Soon)",
    "details": "Saturday 25th of April 2020 | The Academy Dublin | Tickets €25",
    "link": "https://dme-promotions.com/event/corrosion-of-conformity-3-2/"
  },
  {
    "index": 5,
    "title": "Corrosion Of Conformity + Spirit Adrift – Postponed (New Date Soon)",
    "details": "Sunday 26th of April 2020 | Limelight Belfast | Tickets £20",
    "link": "https://dme-promotions.com/event/corrosion-of-conformity-3-3/"
  },
```

4.3.2 NodeJS

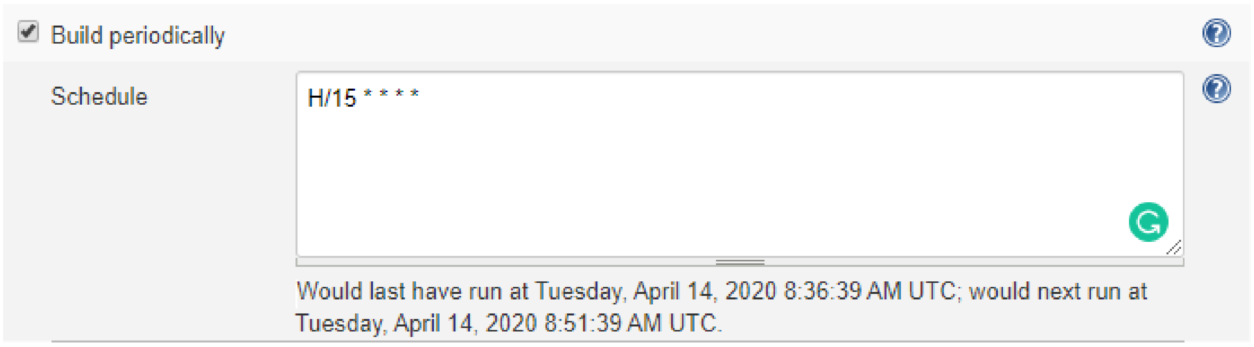
NodeJS is a server-side JavaScript runtime environment that allows JavaScript code to run outside of a web browser. It allows you to listen to HTTP requests, network traffic, access local files and access databases directly, which is one of my primary uses for it.

4.4 Checking for New Content

To provide an exceptional user experience this app would have to display the latest content available. This proves difficult when you are depending on web scraping as ideally you would want the web scraper script to execute when the site adds new content. Some of the bigger companies provide API's for developers to use for this purpose. You usually contact the company or set up a developer account, receive a token and can begin testing API endpoints with a tool like Postman which allows you to perform GET, POST, PUT and DELETE requests and view the response in different formats including XML and JSON. This is the approach I used for getting my Ticketmaster content. For the rest of the content I had to choose the most efficient of the methods explained below.

4.4.1 CRON Scheduler

Cron schedulers provide a way for CICD tools to run jobs/scripts at specified time intervals. They allow a user to set the minute, hour, day of month, month of year and day of the week. I had considered using CRON to run the web scraper scripts every 5 minutes. Below is an example of how to configure CRON to do this in Jenkins.



Build periodically ✓

Schedule

```
H/15 * * * *
```

Would last have run at Tuesday, April 14, 2020 8:36:39 AM UTC; would next run at Tuesday, April 14, 2020 8:51:39 AM UTC.

However, having data that is potentially 5 minutes out of date is still not quite good enough, so I decided against it.

4.4.2 Check for Updates Component

Jenkins provide an API to trigger jobs remotely and even allows the passing of parameters into the API to start a parametrised build. This will allow me to have a “check for updates” button that will trigger a build and run the web scrapers to update the database then reload the calling component displaying the new up to date data.

4.4.3 Ticketmaster API

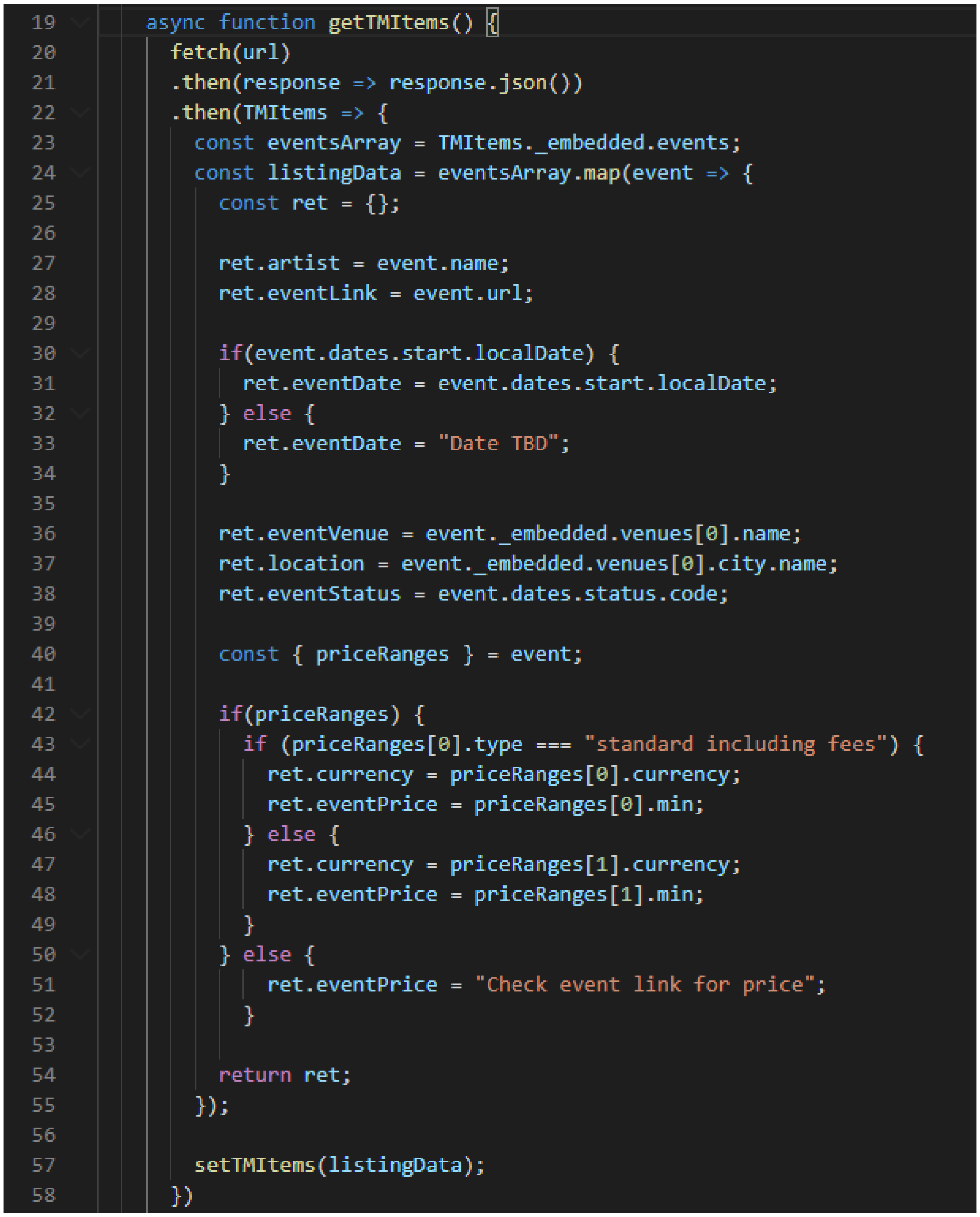
Ticketmaster Discovery API is available to anyone who creates an account and registers an application. I was able to fetch Hard Rock/Metal events taking place in the Locale of Ireland which includes Northern Ireland.

Here is a snippet of the URL I perform a GET request to excluding the domain name.

```
'/events?apikey=' + process.env.REACT_APP_TM_API_KEY +
'&includeTBA=yes&includeTBD=yes&dmaId=608&genreId=KnvZfZ7vAvt'
```

First, I specify the API key Ticketmaster issued to me (Seen here as an environment variable for security reasons). I include events that have a status of 'To Be Announced' and 'To Be Dated'. The DMA ID is the locale to be included followed by the genre ID. This returns roughly a 9000-line JSON response for about 20-25 events.

From this I parse the response to get Artist Name, City, Venue, Date, Ticket Link, Status (Important at the moment, most are cancelled due to COVID-19) and the Price of tickets. The image below shows some of the parsing process.



```
async function getTMItems() {
  fetch(url)
  .then(response => response.json())
  .then(TMItems => {
    const eventsArray = TMItems._embedded.events;
    const listingData = eventsArray.map(event => {
      const ret = {};

      ret.artist = event.name;
      ret.eventLink = event.url;

      if(event.dates.start.localDate) {
        ret.eventDate = event.dates.start.localDate;
      } else {
        ret.eventDate = "Date TBD";
      }

      ret.eventVenue = event._embedded.venues[0].name;
      ret.location = event._embedded.venues[0].city.name;
      ret.eventStatus = event.dates.status.code;

      const { priceRanges } = event;

      if(priceRanges) {
        if (priceRanges[0].type === "standard including fees") {
          ret.currency = priceRanges[0].currency;
          ret.eventPrice = priceRanges[0].min;
        } else {
          ret.currency = priceRanges[1].currency;
          ret.eventPrice = priceRanges[1].min;
        }
      } else {
          ret.eventPrice = "Check event link for price";
        }

      return ret;
    });

    setTMItems(listingData);
  })
```

A fetch is performed. The event name and ticket URL are inside the same object. The if else logic next is to catch events with a status of TBD as these do not have a key named “localDate”.

The venue and city name are inside one of two arrays inside a JSON object. The status value is inside the same object.

There are two arrays named “priceRanges”. One contains the price including booking fee and the other without. This complicates things because I want to choose the price that includes the booking fee. I can differentiate these by finding a “type” key with a string value of “standard including fees”. I also take want the currency as some events are in Northern Ireland.

5 Implementation

Structure

```
\---.circleci
   |---config.yml
\---.git
\---services
   +---express-backend
   +---react-frontend
.env
.gitignore
deploy.sh
deploy_staging.sh
docker-compose.yml
docker-compose-prod.yml
rsync_exclude.txt
```

5.1 Frontend

Frontend Structure

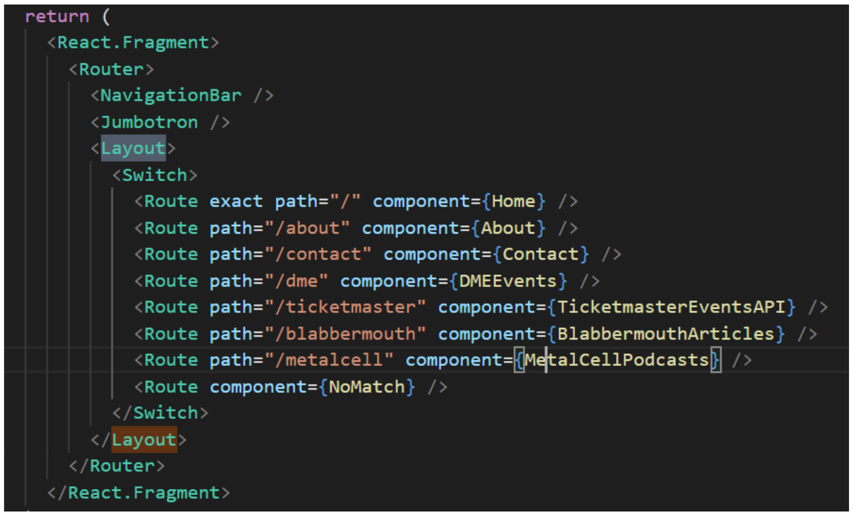
```
\---react-frontend
          +---nginx
                    |---nginx.conf
          +---node_modules
          +---public
                    |---index.html
          \---src
                    +---assets
                              |---cropped_maiden.jpeg
                    +---components
                              |---CheckForUpdates.js
                              |---Dropdown.js
                              |---Jumbotron.js
                              |---Layout.js
                              |---NavigationBar.js
                    +---Blabbermouth
                              |---Blabbermouth
                              |---BlabbermouthDT.js
                    +---DME
                              |---DMEDataTable.js
                              |---DMEEvents.js
```

```
        +---MetalCell
                |---MetalCellDT.js
                |---MetalCellMp3Modal.js
                |---MetalCellPodcasts.js
        +---Ticketmaster
                |---TicketmasterAPI.js
                |---TicketmasterDTAPI.js
        About.js
        App.css
        App.js
        Contact.js
        Home.js
        index.css
        index.js
        NoMatch.js
    docker-compose.yml
    docker-compose-prod.yml
    Dockerfile
    Dockerfile-prod
    package.json
    package-lock.json
    README
```

Routing

The website contains a navigation bar designed and styled using the NavBar component from the react-bootstrap library. This nav bar features a home page, an about page and contact page. The homepage features ‘reactstrap’ cards that contain link to each page.

Routing is handled by the React-Router library and can be viewed in the App.js file.

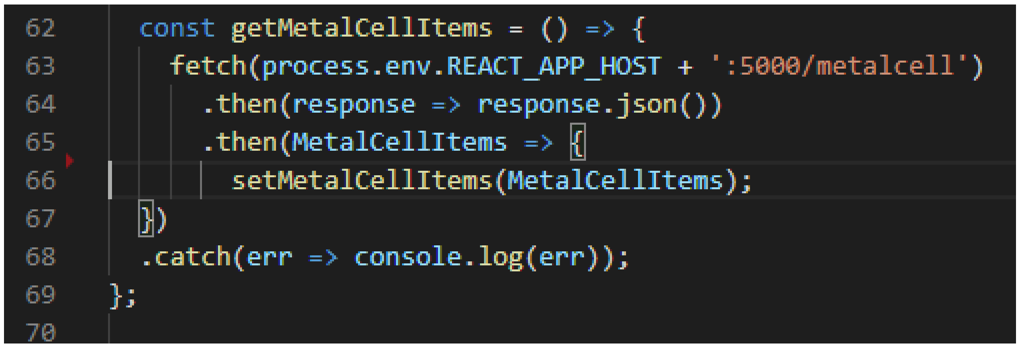


```
return (
  <React.Fragment>
    <Router>
      <NavigationBar />
      <Jumbotron />
      <Layout>
        <Switch>
          <Route exact path="/" component={Home} />
          <Route path="/about" component={About} />
          <Route path="/contact" component={Contact} />
          <Route path="/dme" component={DMEEvents} />
          <Route path="/ticketmaster" component={TicketmasterEventsAPI} />
          <Route path="/blabbermouth" component={BlabbermouthArticles} />
          <Route path="/metalcell" component={MetalCellPodcasts} />
          <Route component={NoMatch} />
        </Switch>
      </Layout>
    </Router>
  </React.Fragment>
```

Fig 1. Routing

Each of my web scraper sources has their own page displaying a table of data. Two components are required for each source.

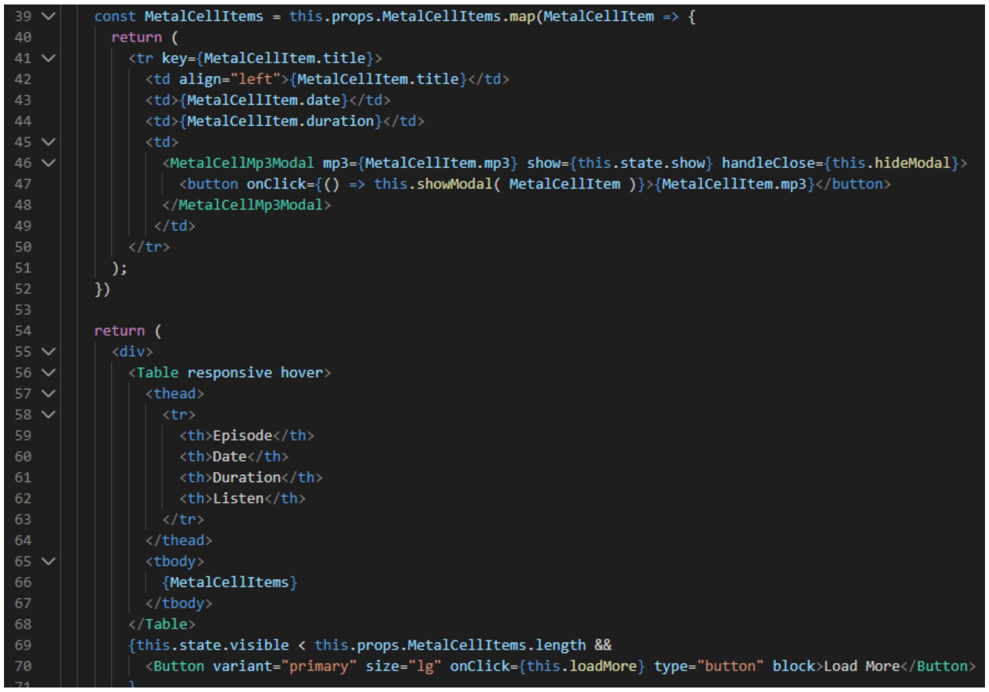
The first component contains methods to display a success/failure snackbar after checking for updates and a LoadingIndicator method that displays a spinner while the CheckForUpdates component executes. This component also performs the communication to the backend to get the data. This is done using Reacts fetch method and returns a JSON response object. The object (MetalCellItems) is then passed to the second component.



```
const getMetalCellItems = () => {
  fetch(process.env.REACT_APP_HOST + ':5000/metalcell')
    .then(response => response.json())
    .then(MetalCellItems => {
      setMetalCellItems(MetalCellItems);
  })
  .catch(err => console.log(err));
};
```

Fig 2.

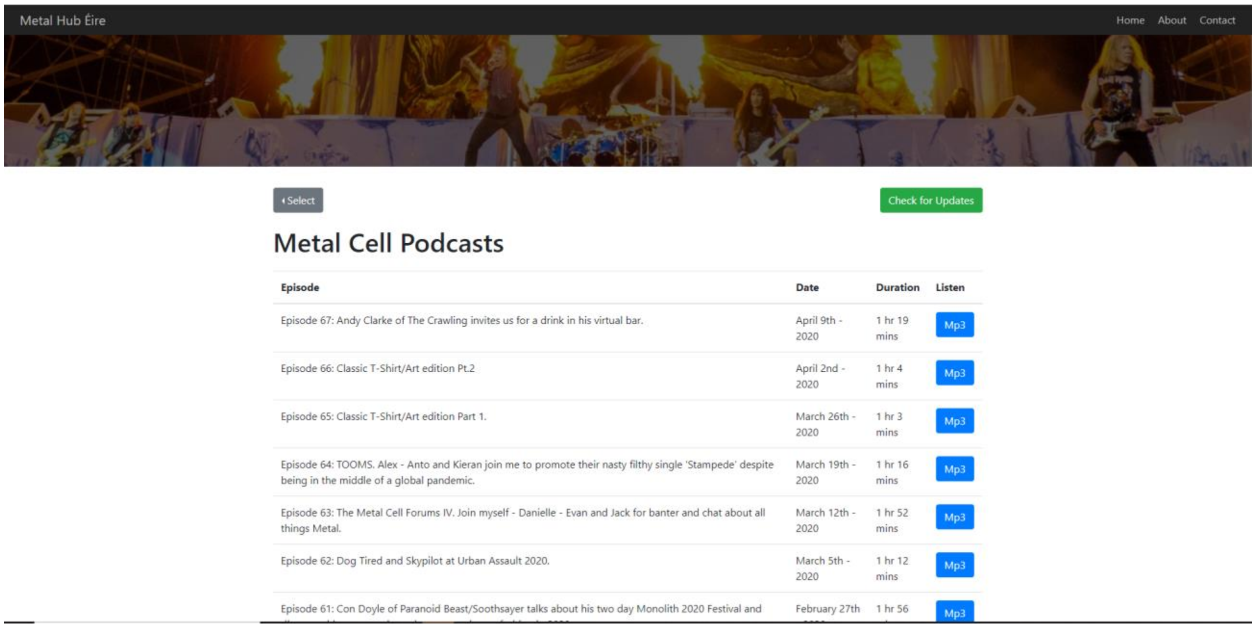
The second component renders the table, parses the object and maps the items to their respective table columns.



```
const MetalCellItems = this.props.MetalCellItems.map(MetalCellItem => {
  return (
    <tr key={MetalCellItem.title}>
      <td align="left">{MetalCellItem.title}</td>
      <td>{MetalCellItem.date}</td>
      <td>{MetalCellItem.duration}</td>
      <td>
        <MetalCellMp3Modal mp3={MetalCellItem.mp3} show={this.state.show} handleClose={this.hideModal}>
          <button onClick={() => this.showModal( MetalCellItem )}>{MetalCellItem.mp3}</button>
        </MetalCellMp3Modal>
       </td>
    </tr>
  );
})

return (
  <div>
    <Table responsive hover>
      <thead>
        <tr>
          <th>Episode</th>
          <th>Date</th>
          <th>Duration</th>
          <th>Listen</th>
        </tr>
      </thead>
      <tbody>
        {MetalCellItems}
      </tbody>
    </Table>
    {this.state.visible < this.props.MetalCellItems.length &&
      <Button variant="primary" size="lg" onClick={this.loadMore} type="button" block>Load More</Button>
```

Fig 3.

The result is a table like this

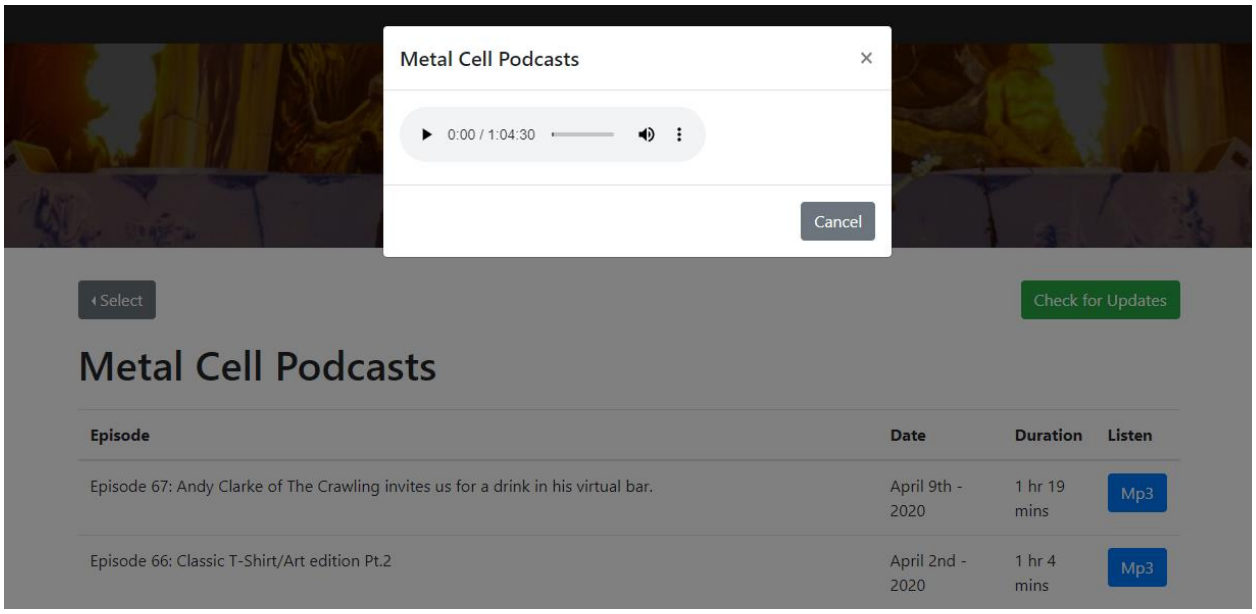


Metal Cell Podcasts

Episode	Date	Duration	Listen
Episode 67: Andy Clarke of The Crawling invites us for a drink in his virtual bar.	April 9th - 2020	1 hr 19 mins	Mp3
Episode 66: Classic T-Shirt/Art edition Pt.2	April 2nd - 2020	1 hr 4 mins	Mp3
Episode 65: Classic T-Shirt/Art edition Part 1.	March 26th - 2020	1 hr 3 mins	Mp3
Episode 64: TOOMS. Alex - Anto and Kieran join me to promote their nasty filthy single 'Stampede' despite being in the middle of a global pandemic.	March 19th - 2020	1 hr 16 mins	Mp3
Episode 63: The Metal Cell Forums IV. Join myself - Danielle - Evan and Jack for banter and chat about all things Metal.	March 12th - 2020	1 hr 52 mins	Mp3
Episode 62: Dog Tired and Skypilot at Urban Assault 2020.	March 5th - 2020	1 hr 12 mins	Mp3
Episode 61: Con Doyle of Paranoid Beast/Soothsayer talks about his two day Monolith 2020 Festival and	February 27th	1 hr 56	Mp3

Fig 4.

For this particular table containing podcasts the mp3 links are collected during the web scraping and then each link is passed as a prop to a reactstrap modal component and is playable with a simple HTML5 Audio element.



Metal Cell Podcasts

Episode	Date	Duration	Listen
Episode 67: Andy Clarke of The Crawling invites us for a drink in his virtual bar.	April 9th - 2020	1 hr 19 mins	Mp3
Episode 66: Classic T-Shirt/Art edition Pt.2	April 2nd - 2020	1 hr 4 mins	Mp3

Fig 5.

5.1.1 Checking for New Content

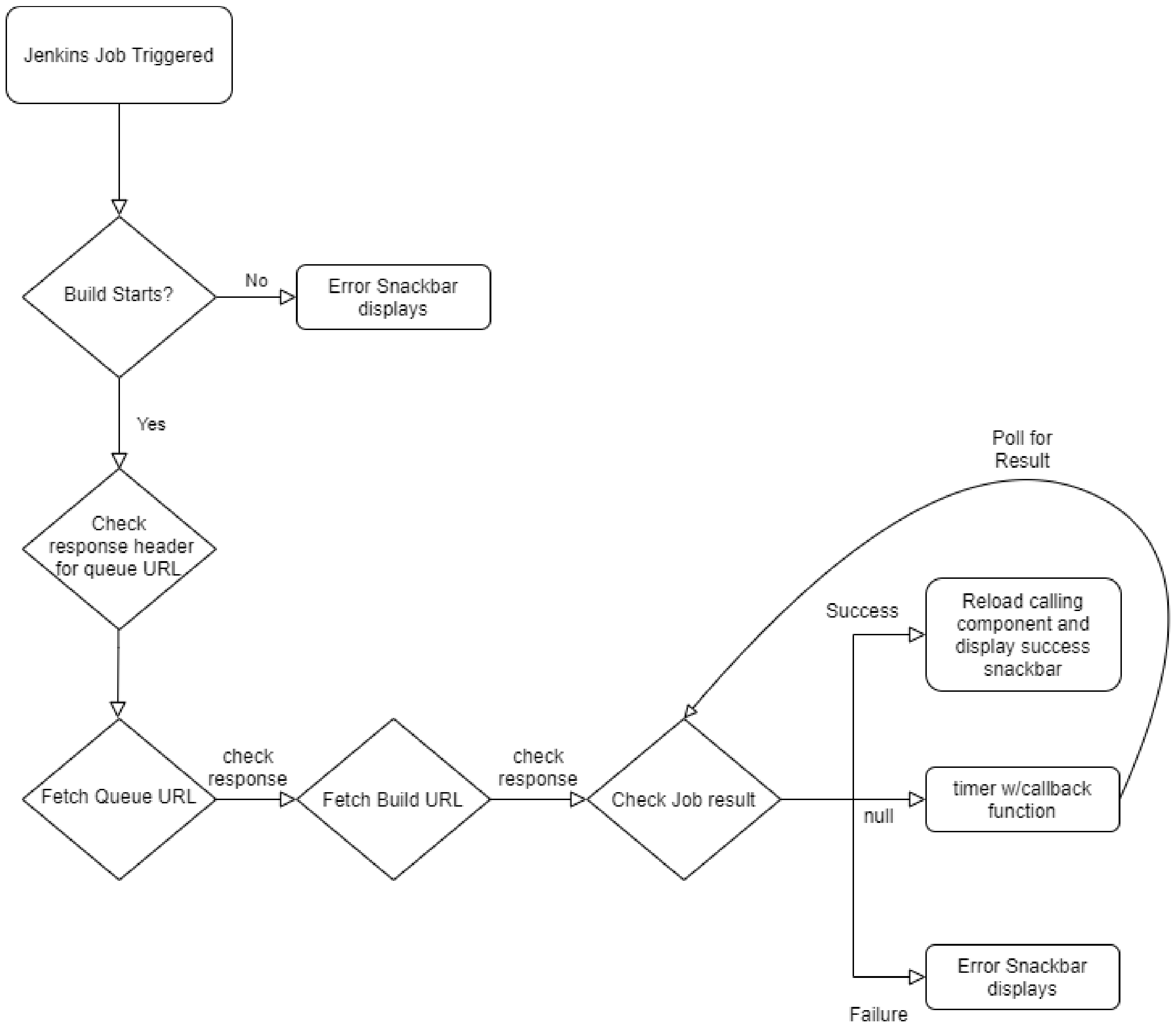


Fig 6.

Sequence of events involved in checking for updates:

(i) The name of the calling component is passed as a prop to the CheckForUpdates component.

```
<CheckForUpdates
  sourceToUpdate="DME"
```

(ii) The **sourceToUpdate** prop is passed into the Jenkins API that triggers the build.

```
<Jenkins_host> +
"view/All/job/run_web_scrapers_test/buildWithParameters?S
CRAPER_SOURCE=" +  sourceToUpdate + "&token=" +
process.env.REACT_APP_JENKINS_TOKEN;
```

(iii) Jenkins takes this prop as a parameter and passes it into the shell script and runs the specified script.

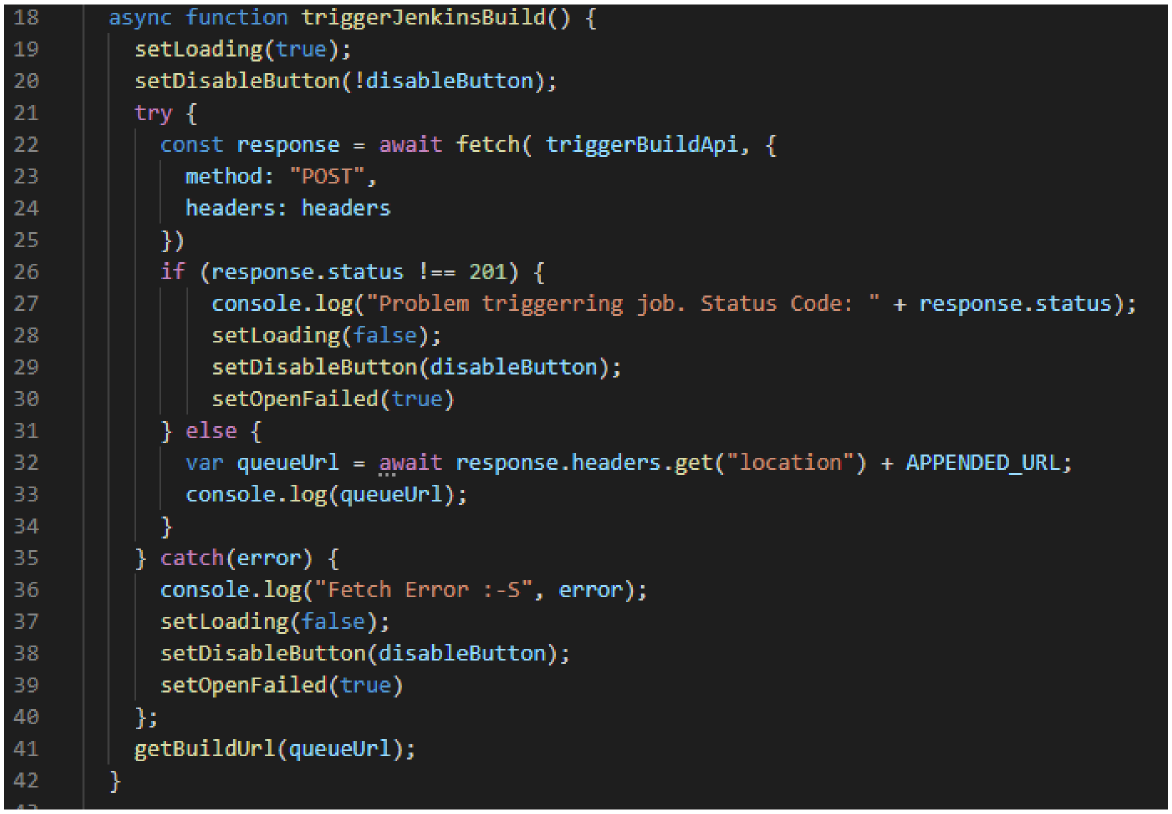
(iv) The script scrapes the webpage and overwrites the database table content.

(v) Upon a successful build the calling component is reloaded asynchronously to display the content.

Note: If the check for updates button is clicked on the homepage all scripts are ran and the whole site is updated.

5.1.2 How the Job result is determined

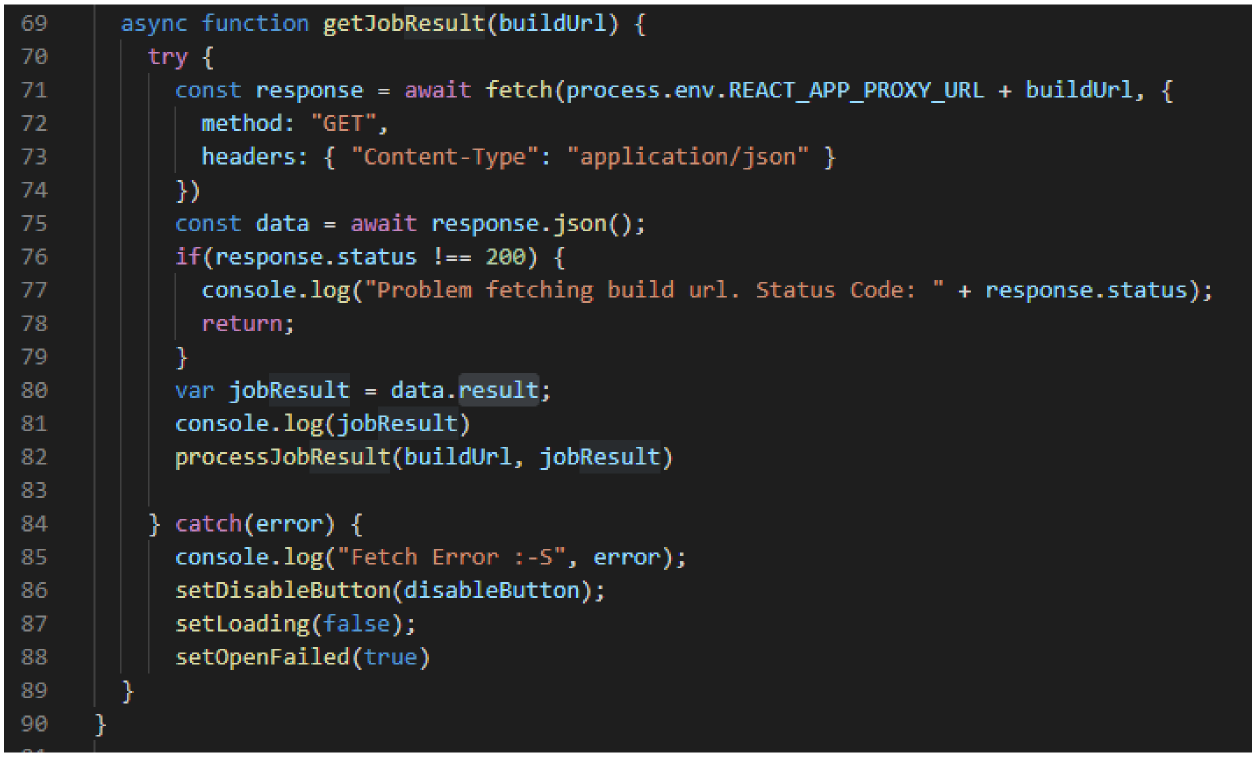
As we can see from the flow chart in fig 6, when a job is triggered a queue URL can be obtained by parsing the response header. Once we get the queue URL it is passed to another async function that obtains the build URL.



```
async function triggerJenkinsBuild() {
  setLoading(true);
  setDisableButton(!disableButton);
  try {
    const response = await fetch( triggerBuildApi, {
      method: "POST",
      headers: headers
    })
    if (response.status !== 201) {
        console.log("Problem triggerring job. Status Code: " + response.status);
        setLoading(false);
        setDisableButton(disableButton);
        setOpenFailed(true)
    } else {
      var queueUrl = await response.headers.get("location") + APPENDED_URL;
      console.log(queueUrl);
    }
  } catch(error) {
    console.log("Fetch Error :-S", error);
    setLoading(false);
    setDisableButton(disableButton);
    setOpenFailed(true)
  };
  getBuildUrl(queueUrl);
}
```

Fig 7.

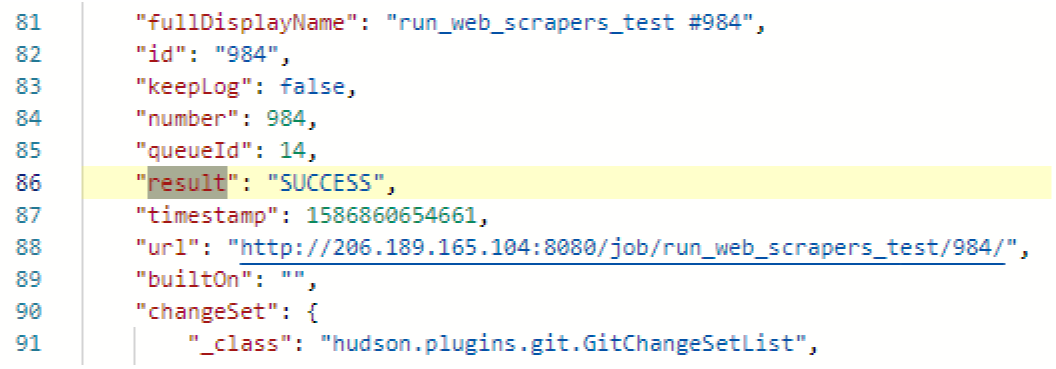
We can fetch this queue URL and parse the response to obtain the build URL and pass this to another async function that gets the JobResult.



```
async function getJobResult(buildUrl) {
  try {
    const response = await fetch(process.env.REACT_APP_PROXY_URL + buildUrl, {
      method: "GET",
      headers: { "Content-Type": "application/json" }
    })
    const data = await response.json();
    if(response.status !== 200) {
      console.log("Problem fetching build url. Status Code: " + response.status);
      return;
    }
    var jobResult = data.result;
    console.log(jobResult)
    processJobResult(buildUrl, jobResult)

  } catch(error) {
    console.log("Fetch Error :-S", error);
    setDisableButton(disableButton);
    setLoading(false);
    setOpenFailed(true)
  }
}
```

Fig 8.

From testing with Postman, I can see the JSON response contained a key called "result" which can hold one of three values, null (while build is ongoing), SUCCESS or FAILURE.



```
    "fullDisplayName": "run_web_scrapers_test #984",
    "id": "984",
    "keepLog": false,
    "number": 984,
    "queueId": 14,
    "result": "SUCCESS",
    "timestamp": 1586860654661,
    "url": "http://206.189.165.104:8080/job/run_web_scrapers_test/984/",
    "builtOn": "",
    "changeSet": {
        "_class": "hudson.plugins.git.GitChangeSetList",
```

Fig 9.

One more function is called and simple “if” conditionals determine how to handle each result value.

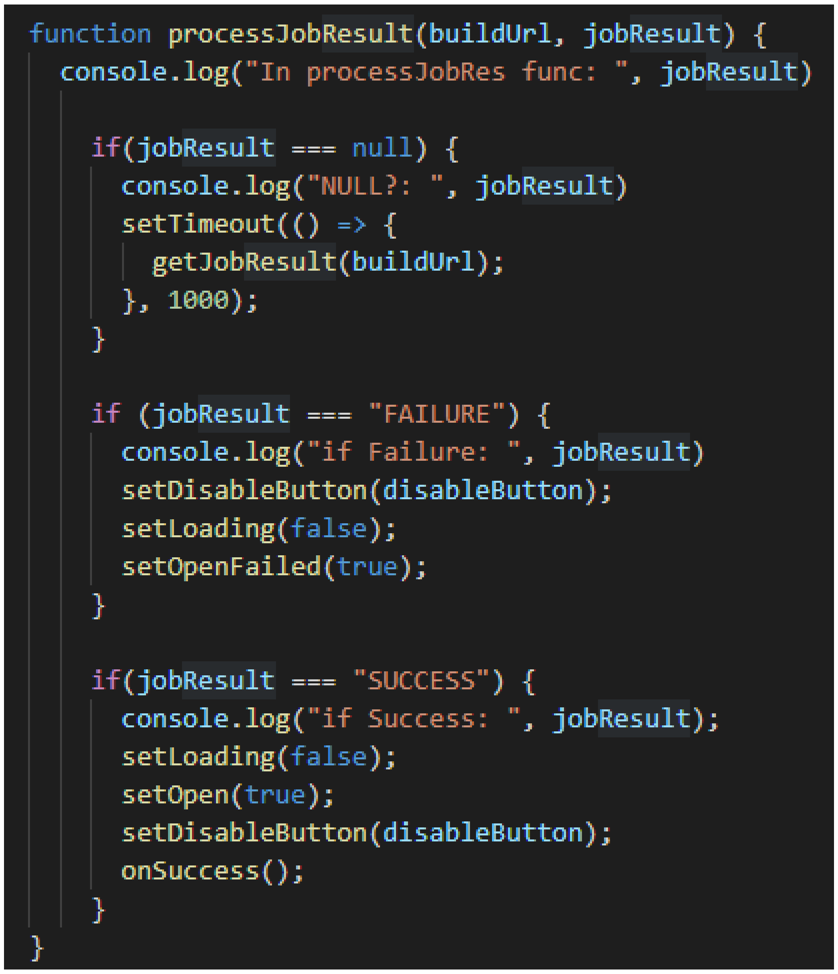

```
function processJobResult(buildUrl, jobResult) {
  console.log("In processJobRes func: ", jobResult)

    if(jobResult === null) {
      console.log("NULL?: ", jobResult)
      setTimeout(() => {
        getJobResult(buildUrl);
      }, 1000);
    }

    if (jobResult === "FAILURE") {
      console.log("if Failure: ", jobResult)
      setDisableButton(disableButton);
      setLoading(false);
      setOpenFailed(true);
    }

    if(jobResult === "SUCCESS") {
      console.log("if Success: ", jobResult);
      setLoading(false);
      setOpen(true);
      setDisableButton(disableButton);
      onSuccess();
    }
}
```

Fig 10.

A “null” value indicates that the build has not yet finished. In this we use “setTimeout()” to set a timer to execute a callback function which will allow me to poll the JSON response a little over every second to check for a SUCCESS or FAILURE value.

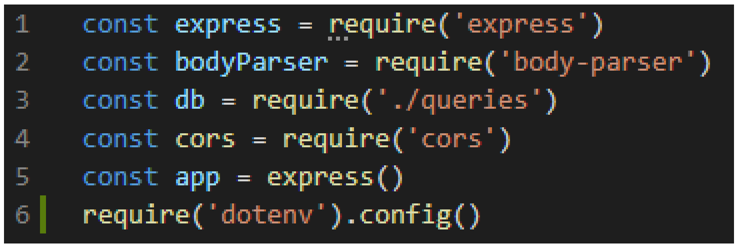
5.2 Backend

Backend Structure

```
+---express-backend
|   \---node_modules
        |---.dockerignore
        |---.env
        |---docker-compose.yml
        |---docker-compose-prod.yml
        |---Dockerfile
        |---Dockerfile-prod
        |---package.json
        |---package-lock.json
        |---queries.js
        |---server.js
```

5.2.1 Server.js

This is my NodeJS Server file. We will first go through the requirements defined at the start of the file.



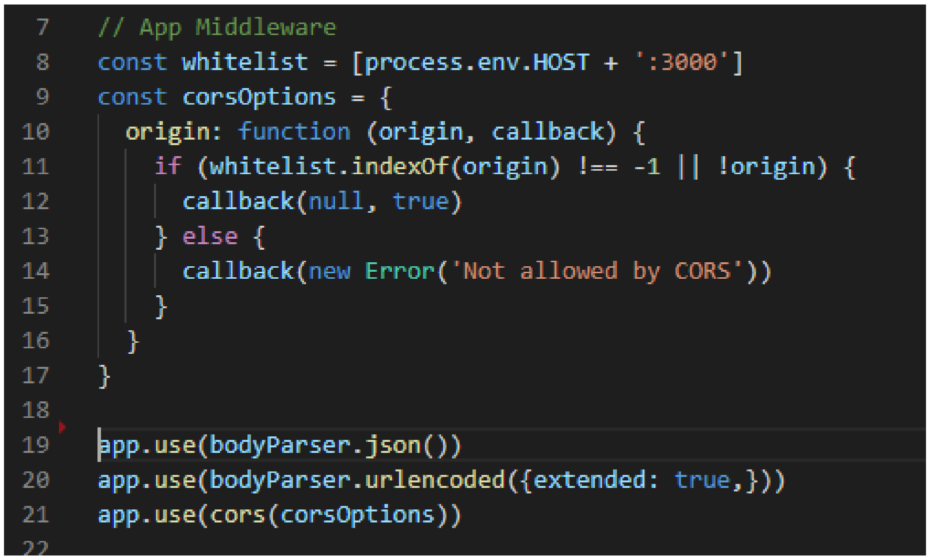
```
const express = require('express')
const bodyParser = require('body-parser')
const db = require('./queries')
const cors = require('cors')
const app = express()
require('dotenv').config()
```

Fig. 11

1. Express – The Framework for the server
2. Body-parser – Middleware used to parse incoming request bodies
3. db – Imports my queries.js file containing all of my Postgres queries
4. Cors (Cross-origin resource sharing) – Prevents unauthorised access to my API data by enforcing same-origin policy and only allowing whitelisteds host/s to fetch data from a database on a remote host.
5. dotenv – node package that lets express consume environment variables.

6. I also use nodemon which lets your node server update after saving code changes without re-running the server.

Next, the middleware is configured.

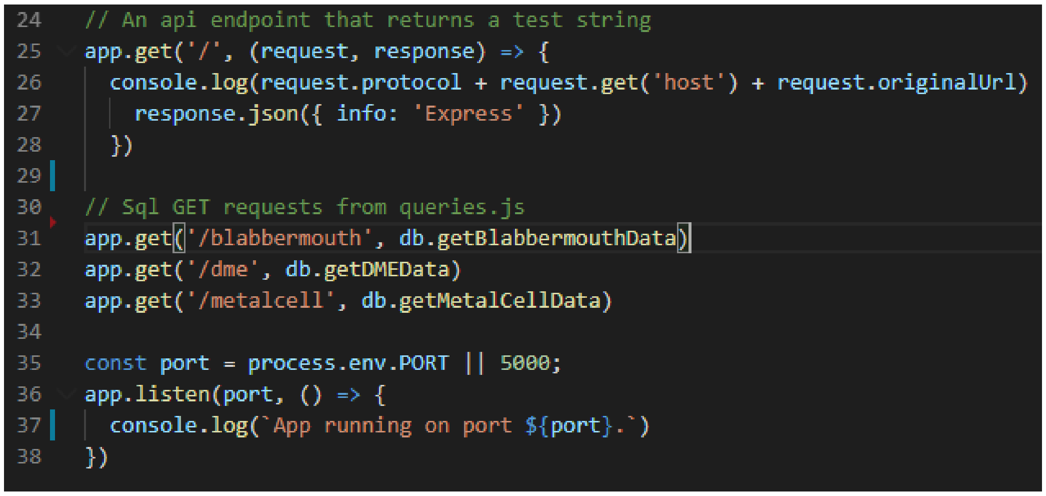


```
// App Middleware
const whitelist = [process.env.HOST + ':3000']
const corsOptions = {
  origin: function (origin, callback) {
    if (whitelist.indexOf(origin) !== -1 || !origin) {
      callback(null, true)
    } else {
      callback(new Error('Not allowed by CORS'))
    }
  }
}

app.use(bodyParser.json())
app.use(bodyParser.urlencoded({extended: true,}))
app.use(cors(corsOptions))
```

Fig 12.

1. First, I whitelist my host, which is hidden in an environment file on my local PC, the staging environment server and the production server.
2. Check if origin has been whitelisted, if not the error message will be displayed in the response header.
3. We instruct express to use JSON while parsing the request body
4. “Extended: true” instructs body-parser to use a complex algorithm to allow the parsing of nested objects etc.
5. We then tell express to use the Cors options defined earlier.

Finally..

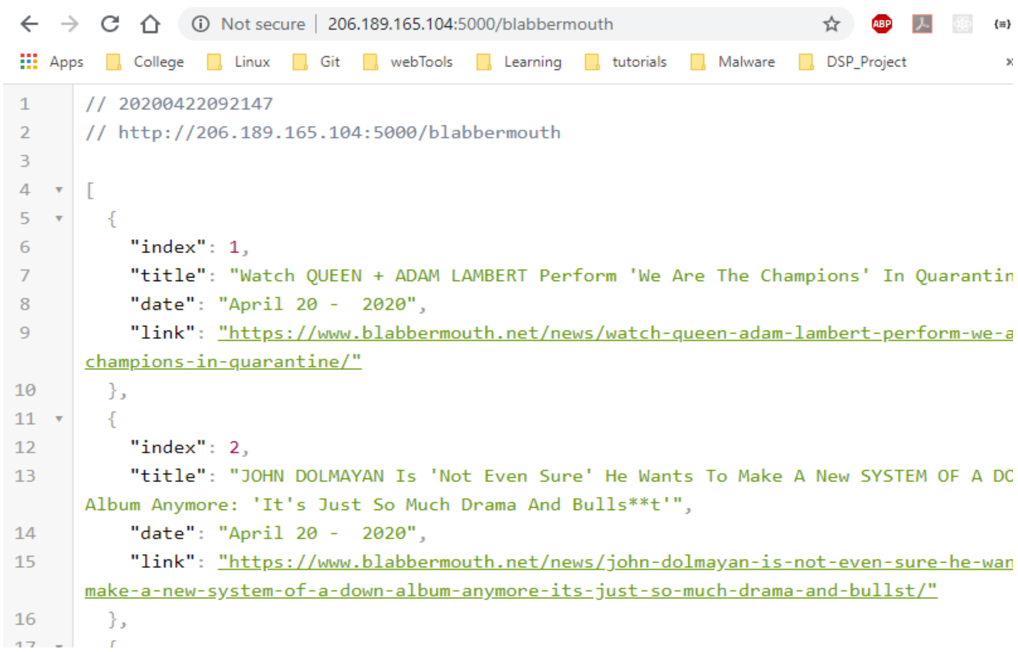


```javascript
// An api endpoint that returns a test string
app.get('/', (request, response) => {
  console.log(request.protocol + request.get('host') + request.originalUrl)
    response.json({ info: 'Express' })
  })

// Sql GET requests from queries.js
app.get('/blabbermouth', db.getBlabbermouthData)
app.get('/dme', db.getDMEData)
app.get('/metalcell', db.getMetalCellData)

const port = process.env.PORT || 5000;
app.listen(port, () => {
  console.log(`App running on port ${port}.`)
})
```

Fig 13

1. An endpoint that returns a string to indicate the server is running, which is handy when debugging.
2. Get requests are made to each page.
3. Next I tell express to listen in on port 5000.

Now I can view my data on port 5000



Not secure | 206.189.165.104:5000/blabbermouth
Apps College Linux Git webTools Learning tutorials Malware DSP_Project

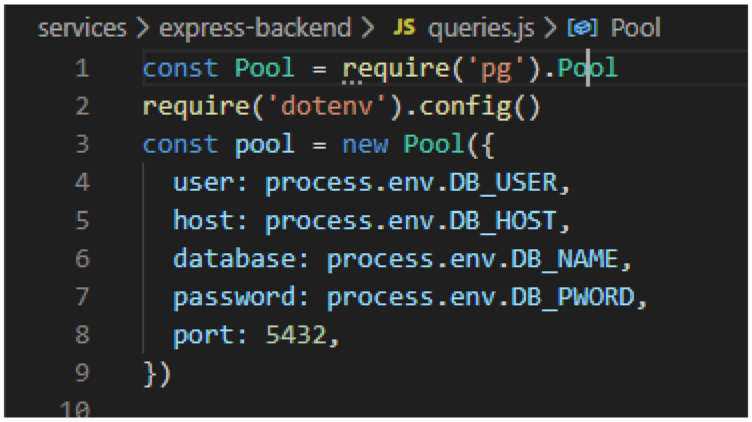
```json
// 20200422092147
// http://206.189.165.104:5000/blabbermouth

[
  {
    "index": 1,
    "title": "Watch QUEEN + ADAM LAMBERT Perform 'We Are The Champions' In Quarantin
    "date": "April 20 -  2020",
    "link": "https://www.blabbermouth.net/news/watch-queen-adam-lambert-perform-we-a
champions-in-quarantine/"
  },
  {
    "index": 2,
    "title": "JOHN DOLMAYAN Is 'Not Even Sure' He Wants To Make A New SYSTEM OF A DC
Album Anymore: 'It's Just So Much Drama And Bulls**t'",
    "date": "April 20 -  2020",
    "link": "https://www.blabbermouth.net/news/john-dolmayan-is-not-even-sure-he-wan
make-a-new-system-of-a-down-album-anymore-its-just-so-much-drama-and-bullst/"
  },
```

Fig 14.

5.2.2 Queries.js

This file contains my postgres database credentials and handles the connection to the database and also contains my SQL queries.

First, we set up a connection pool



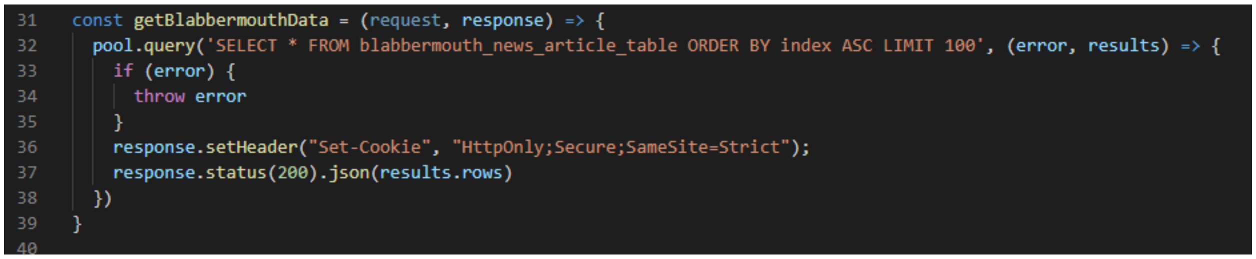
services > express-backend > JS queries.js > Pool

```javascript
const Pool = require('pg').Pool
require('dotenv').config()
const pool = new Pool({
  user: process.env.DB_USER,
  host: process.env.DB_HOST,
  database: process.env.DB_NAME,
  password: process.env.DB_PWORD,
  port: 5432,
})
```

Fig 15.

1. The dotenv package is used here to store our credentials as environment variables.
2. The postgres library is imported and we set our Username (DigitalOcean IP), the database name, password and port number.

Query structure



```javascript
const getBlabbermouthData = (request, response) => {
  pool.query('SELECT * FROM blabbermouth_news_article_table ORDER BY index ASC LIMIT 100', (error, results) => {
    if (error) {
      throw error
    }
    response.setHeader("Set-Cookie", "HttpOnly;Secure;SameSite=Strict");
    response.status(200).json(results.rows)
  })
}
```

1. An arrow function is used the takes two parameters
2. A query is made using the postgres pool class to validate credentials and perform the query and catch any errors.
3. The response headers are set, set-cookie is sent by the server in response to an HTTP request and a cookie is created on the client side. HttpOnly prevents cross site scripting attacks.

6 Deployment

For the development phase I had been using docker desktop for windows to containerise my front and back end. This allowed me to learn how to write and test Dockerfiles and docker-compose yaml files locally before getting to the production stage. For deploying to staging and production environments I installed docker for linux on my AWS EC2 instance which I use for staging and on my DigitalOcean droplet which I will use for production.

6.1 React Dockerfile for Production

In this section we go through the React & Express Dockerfiles I use for production; I have commented over each command used to spin up the React container to explain its purpose. This is a multi-stage build; the second stage is building the Nginx Image.

```
# Stage 1
# build environment from a node base image
FROM node:12.2.0-alpine as build

#set the working directory
WORKDIR /react-frontend

# add the node_modules folder to $PATH
ENV PATH /react-frontend/node_modules/.bin:$PATH

# Copies package.json & package-lock to workdir to identify required npm packages
COPY package*.json /react-frontend/

# Install dependencies, clean cache and removed any unused npm packages
RUN npm install --silent \
&& npm cache clean --force \
&& npm prune

# Copy entire react folder to /  react-frontend directory in the container
COPY . /react-frontend

# Create a Production optimised react build
RUN npm run build

#Stage 2
# NGINX Alpine based image
FROM nginx:alpine
```

```
# copy the build folder (produced by RUN command above) from react to the
# root of nginx (www) container
COPY --from=build /react-frontend/build /usr/share/nginx/html


#Remove default nginx conf file to route through react
RUN rm /etc/nginx/conf.d/default.conf

#Replace with custom one (Located Here)
COPY nginx/nginx.conf /etc/nginx/conf.d

# expose port 80 to the outer world
EXPOSE 80

# start nginx
CMD ["nginx", "-g", "daemon off;"]
```

6.2 Express Dockerfile for Production

```
# Build Environment from node base image
FROM node:12.2.0-alpine

# Set working directory
WORKDIR /express-backend

# Add the node_modules folder to $PATH
ENV PATH /express-backend/node_modules/.bin:$PATH

# Copies package.json & package-lock to WORKDIR to identify required npm packages
COPY package*.json /express-backend/

# Install dependencies, clean cache and removed any unused npm packages
RUN npm install --silent \
&& npm cache clean --force \
&& npm prune

# Copy entire express-backend folder to /express-backend directory in the container
COPY . /express-backend
# Run the server
#CMD ["npm", "run", "start:prod"]
```

6.3 Docker Compose for Production

The docker-compose tool is used to run multiple containers at once. The docker-compose file is located at the root of the project and can be executed by running :

```
docker-compose up -d --build
```

```
version: '3.2'

services:
  react-prod:
    container_name: react-prod
    build:
      context: ./services/react-frontend
      dockerfile: Dockerfile-prod
    volumes:
     - './services/react-frontend:/react-frontend'
     - '/react-frontend/node_modules'
    ports:
      - '3000:80'
    environment:
      - NODE_ENV=production

  express-backend:
    container_name: express-prod
    build:
      context: ./services/express-backend
      dockerfile: Dockerfile-prod
    volumes:
      - './services/express-backend:/express-backend'
      - '/express-backend/node_modules'
    ports:
      - 5000:5000
    environment:
      - NODE_ENV=production
```

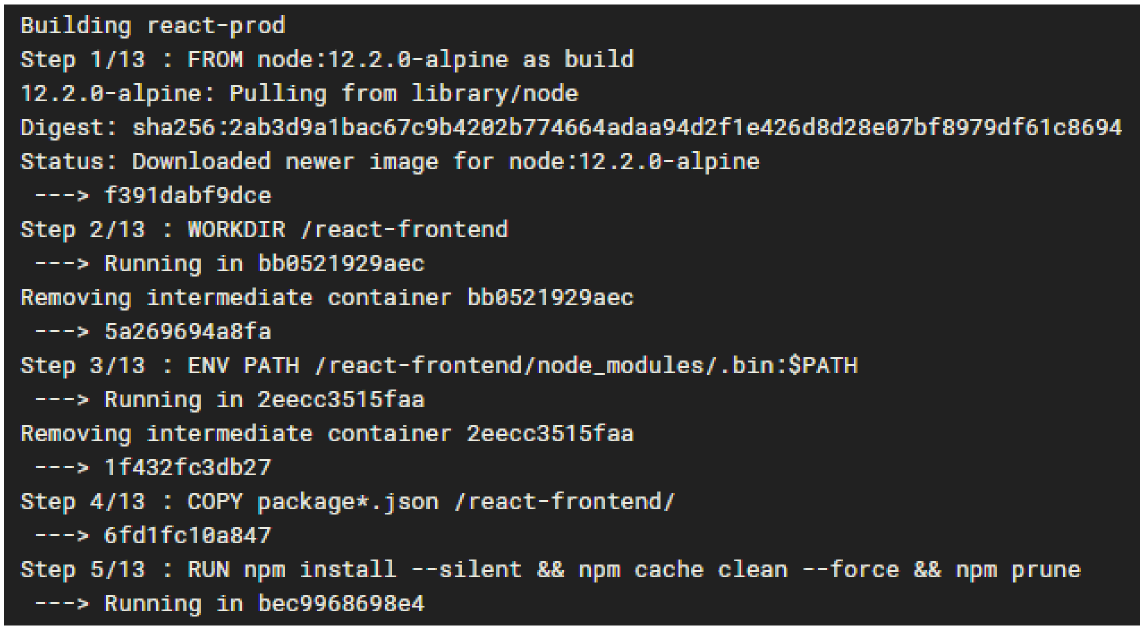
We have two services, the first is react-production. We name the container first, then navigate to the react-frontend directory using the context keyword and specify the name of the Dockerfile to be ran (Dockerfile-prod). A Host Volume is created first that mounts the host directory to the container and then an anonymous volume is created for the `node_modules` directory. Without this anonymous volume the `node_modules` directory would be overwritten when the host directory mounts.

We specify port 80, the default Nginx port and set `NODE_ENV=production` environment variable so we don't install any dev-dependencies which are not needed in production.

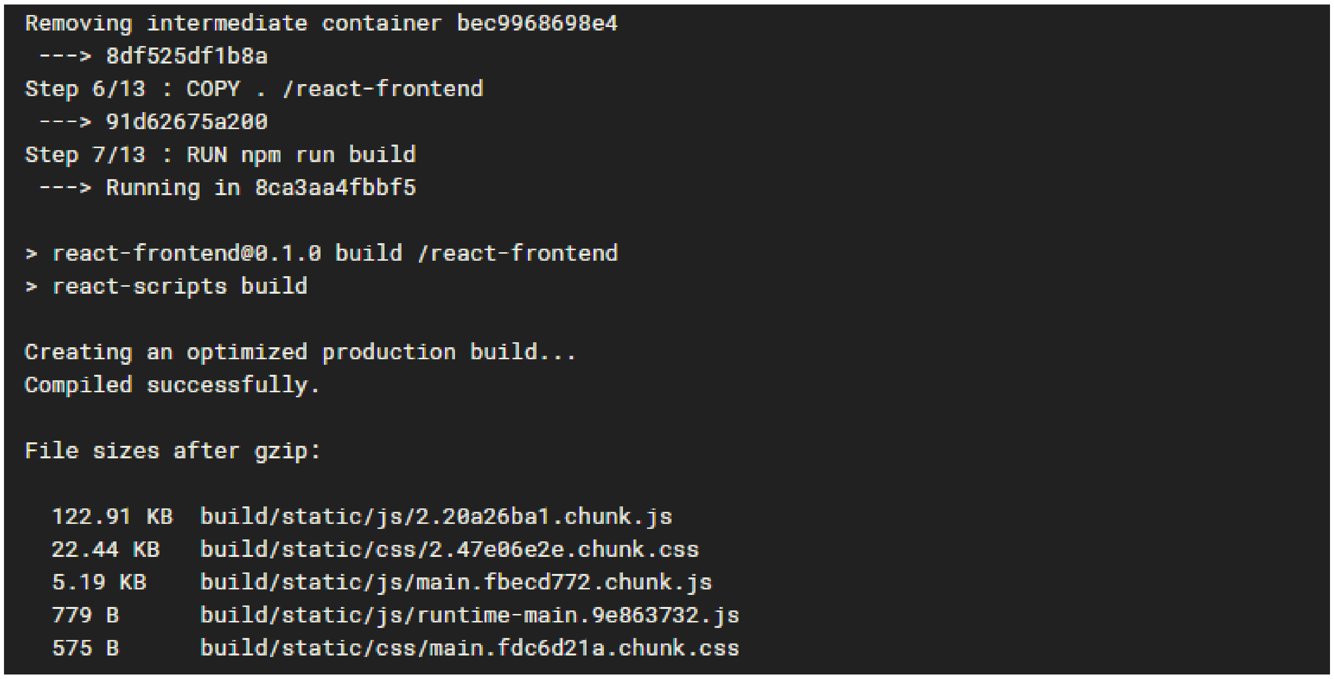
The second service is our node/express container. We name the container. Then navigate to the express-backend directory using the context keyword and specify the name of the Dockerfile (Dockerfile-prod). The same process is repeated with creating volumes ensuring the `node_modules` directory is not overwritten. Port 5000 is our node port, so we expose it and then let the `NODE_ENV=production`

The output of running `docker-compose up -d --build` should show us each command being executed. It should look like the image below:

1. **Building the react image**

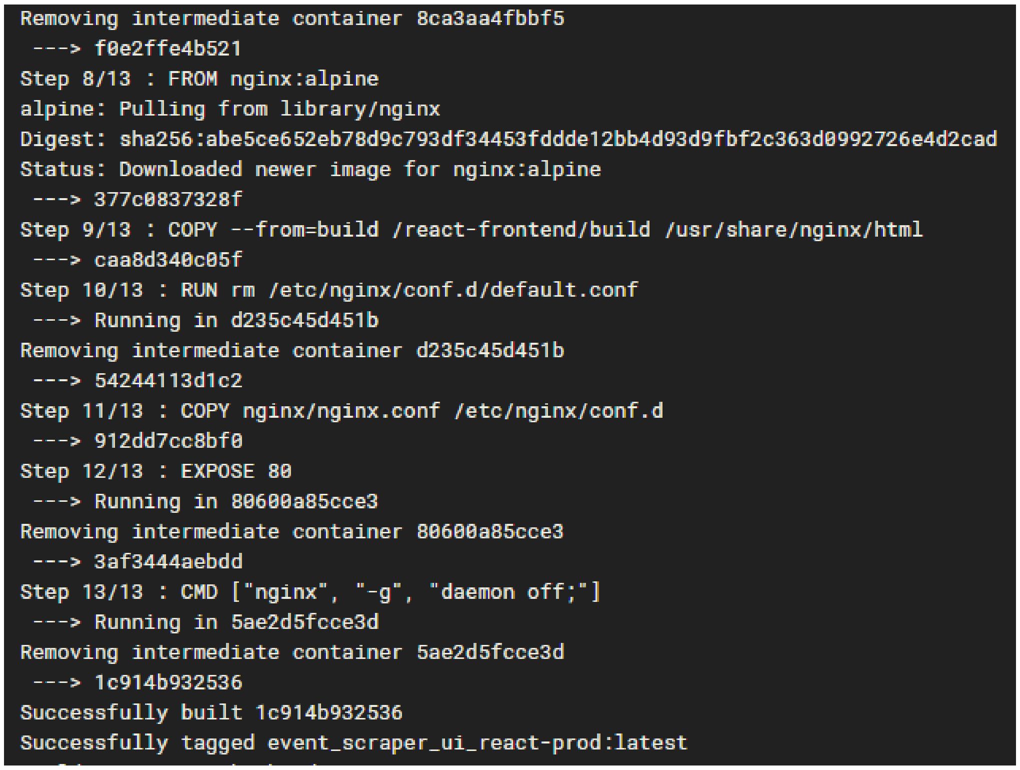


```
Building react-prod
Step 1/13 : FROM node:12.2.0-alpine as build
12.2.0-alpine: Pulling from library/node
Digest: sha256:2ab3d9a1bac67c9b4202b774664adaa94d2f1e426d8d28e07bf8979df61c8694
Status: Downloaded newer image for node:12.2.0-alpine
 ---> f391dabf9dce
Step 2/13 : WORKDIR /react-frontend
 ---> Running in bb0521929aec
Removing intermediate container bb0521929aec
 ---> 5a269694a8fa
Step 3/13 : ENV PATH /react-frontend/node_modules/.bin:$PATH
 ---> Running in 2eecc3515faa
Removing intermediate container 2eecc3515faa
 ---> 1f432fc3db27
Step 4/13 : COPY package*.json /react-frontend/
 ---> 6fd1fc10a847
Step 5/13 : RUN npm install --silent && npm cache clean --force && npm prune
 ---> Running in bec9968698e4
```



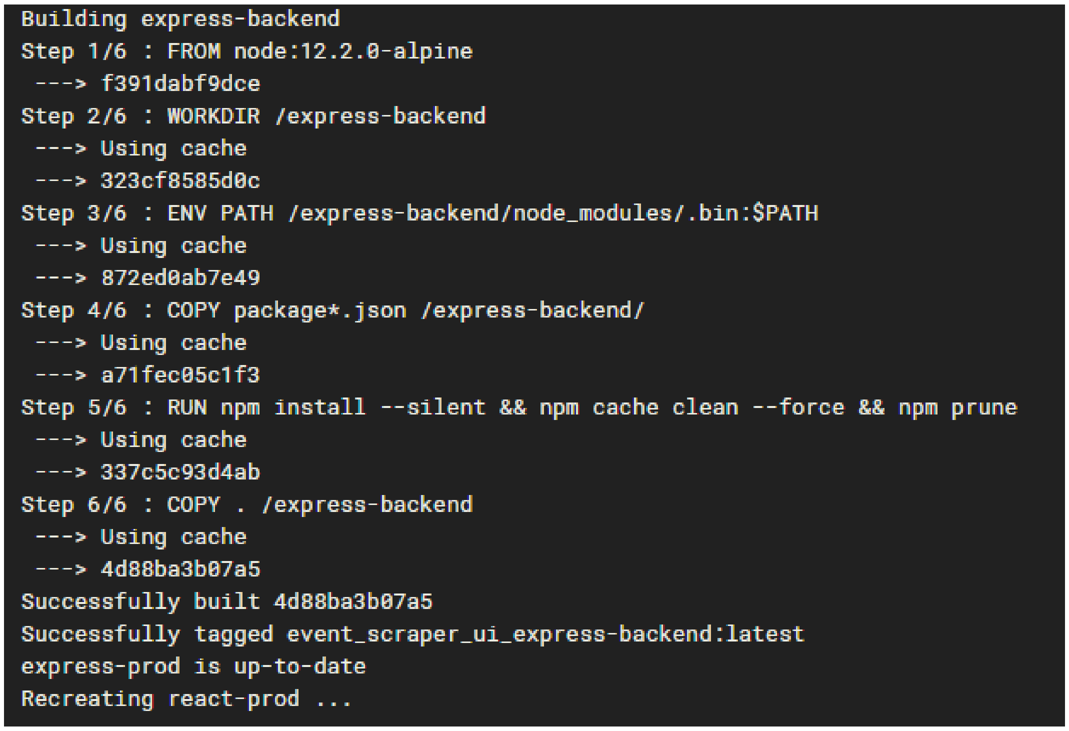
```
Removing intermediate container bec9968698e4
 ---> 8df525df1b8a
Step 6/13 : COPY . /react-frontend
 ---> 91d62675a200
Step 7/13 : RUN npm run build
 ---> Running in 8ca3aa4fbbf5

> react-frontend@0.1.0 build /react-frontend
> react-scripts build

Creating an optimized production build...
Compiled successfully.

File sizes after gzip:

  122.91 KB  build/static/js/2.20a26ba1.chunk.js
  22.44 KB   build/static/css/2.47e06e2e.chunk.css
  5.19 KB    build/static/js/main.fbecd772.chunk.js
  779 B      build/static/js/runtime-main.9e863732.js
  575 B      build/static/css/main.fdc6d21a.chunk.css
```

2. Building the Nginx Image



```
Removing intermediate container 8ca3aa4fbbf5
 ---> f0e2ffe4b521
Step 8/13 : FROM nginx:alpine
alpine: Pulling from library/nginx
Digest: sha256:abe5ce652eb78d9c793df34453fddde12bb4d93d9fbf2c363d0992726e4d2cad
Status: Downloaded newer image for nginx:alpine
 ---> 377c0837328f
Step 9/13 : COPY --from=build /react-frontend/build /usr/share/nginx/html
 ---> caa8d340c05f
Step 10/13 : RUN rm /etc/nginx/conf.d/default.conf
 ---> Running in d235c45d451b
Removing intermediate container d235c45d451b
 ---> 54244113d1c2
Step 11/13 : COPY nginx/nginx.conf /etc/nginx/conf.d
 ---> 912dd7cc8bf0
Step 12/13 : EXPOSE 80
 ---> Running in 80600a85cce3
Removing intermediate container 80600a85cce3
 ---> 3af3444aebdd
Step 13/13 : CMD ["nginx", "-g", "daemon off;"]
 ---> Running in 5ae2d5fcce3d
Removing intermediate container 5ae2d5fcce3d
 ---> 1c914b932536
Successfully built 1c914b932536
Successfully tagged event_scraper_ui_react-prod:latest
```

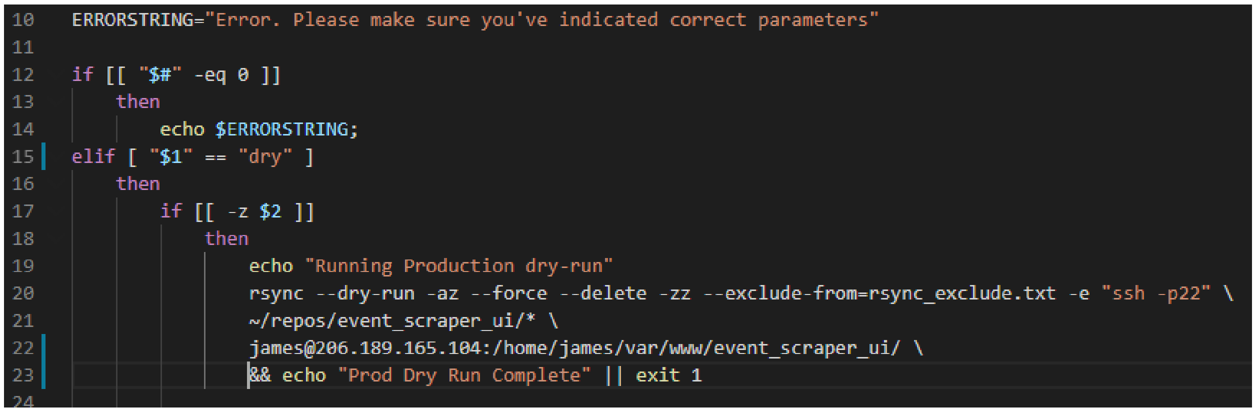
3. Finally, the node/express image



```
Building express-backend
Step 1/6 : FROM node:12.2.0-alpine
 ---> f391dabf9dce
Step 2/6 : WORKDIR /express-backend
 ---> Using cache
 ---> 323cf8585d0c
Step 3/6 : ENV PATH /express-backend/node_modules/.bin:$PATH
 ---> Using cache
 ---> 872ed0ab7e49
Step 4/6 : COPY package*.json /express-backend/
 ---> Using cache
 ---> a71fec05c1f3
Step 5/6 : RUN npm install --silent && npm cache clean --force && npm prune
 ---> Using cache
 ---> 337c5c93d4ab
Step 6/6 : COPY . /express-backend
 ---> Using cache
 ---> 4d88ba3b07a5
Successfully built 4d88ba3b07a5
Successfully tagged event_scraper_ui_express-backend:latest
express-prod is up-to-date
Recreating react-prod ...
```

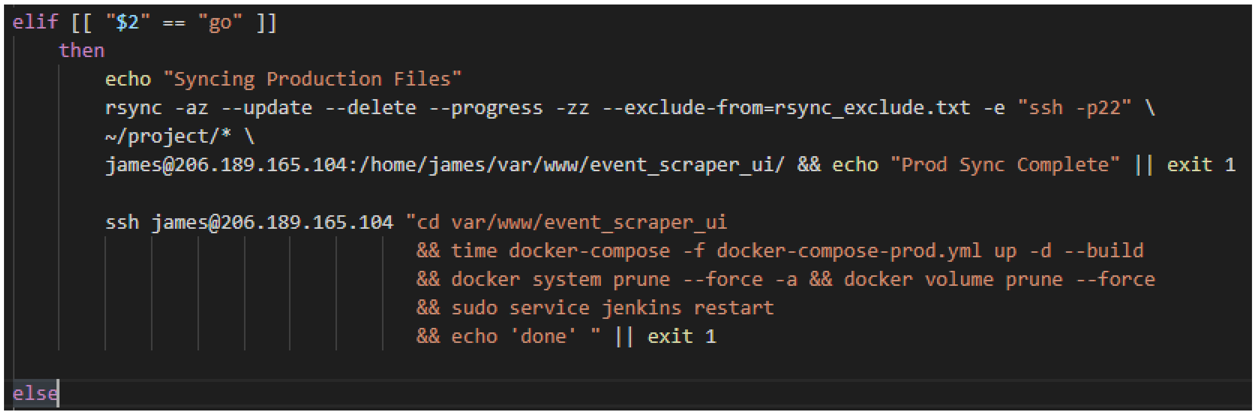
6.4 Deploy Script

The `deploy.sh` shell script will be used in my deployment pipeline which I will go through in the next section to synchronise files between my master branch on GitHub and my applications directory on the servers. Then it needs to run the docker compose file and spin up my production containers. Because staging and production are hosted on two different servers there are two different deploy scripts. The only difference is the IP of the Host. We will go through the production script below.



```
ERRORSTRING="Error. Please make sure you've indicated correct parameters"

if [[ "$#" -eq 0 ]]
    then
        echo $ERRORSTRING;
elif [ "$1" == "dry" ]
    then
        if [[ -z $2 ]]
            then
                echo "Running Production dry-run"
                rsync --dry-run -az --force --delete -zz --exclude-from=rsync_exclude.txt -e "ssh -p22" \
                ~/repos/event_scraper_ui/* \
                james@206.189.165.104:/home/james/var/www/event_scraper_ui/ \
                && echo "Prod Dry Run Complete" || exit 1
```

This script takes 2 parameters, "dry" and/or "live". Passing in "dry" will perform a dry run which will test the ssh connection to the VM and list all files that are to be copied without performing the copy operation. This is just for testing.



```
elif [[ "$2" == "go" ]]
    then
        echo "Syncing Production Files"
        rsync -az --update --delete --progress -zz --exclude-from=rsync_exclude.txt -e "ssh -p22" \
        ~/project/* \
        james@206.189.165.104:/home/james/var/www/event_scraper_ui/ && echo "Prod Sync Complete" || exit 1

        ssh james@206.189.165.104 "cd var/www/event_scraper_ui
                                  && time docker-compose -f docker-compose-prod.yml up -d --build
                                  && docker system prune --force -a && docker volume prune --force
                                  && sudo service jenkins restart
                                  && echo 'done' " || exit 1

else
```

Passing in "live" and "go" will perform the sync operation using the Rsync (Remote Synchronisation) tool.

Rsync syntax:

rsync <Options> <Source-Files-Dir> <User_Name>@<Remote-Host>:<Destination>

The rsync options used here performs the following

-**a |** Archive files during transfer. (-z is now deprecated, use -zz instead)

--update | Only sync files that are newer that the destinations copy

--delete | Deletes files at destination if not present in source.

--progress | Display progress of sync operation.

-zz | Compress files during transfer

--exclude-from=rsync_exclude.txt | Similar to gitignore, excludes any files or directories in this txt document located in the project root.

After the files have been synced it's time to spin up the docker containers with docker compose.

Docker system prune removes any old containers and images and docker volume prune removes any unused volumes. Jenkins is restarted because Nginx takes down the jenkins service when it builds.

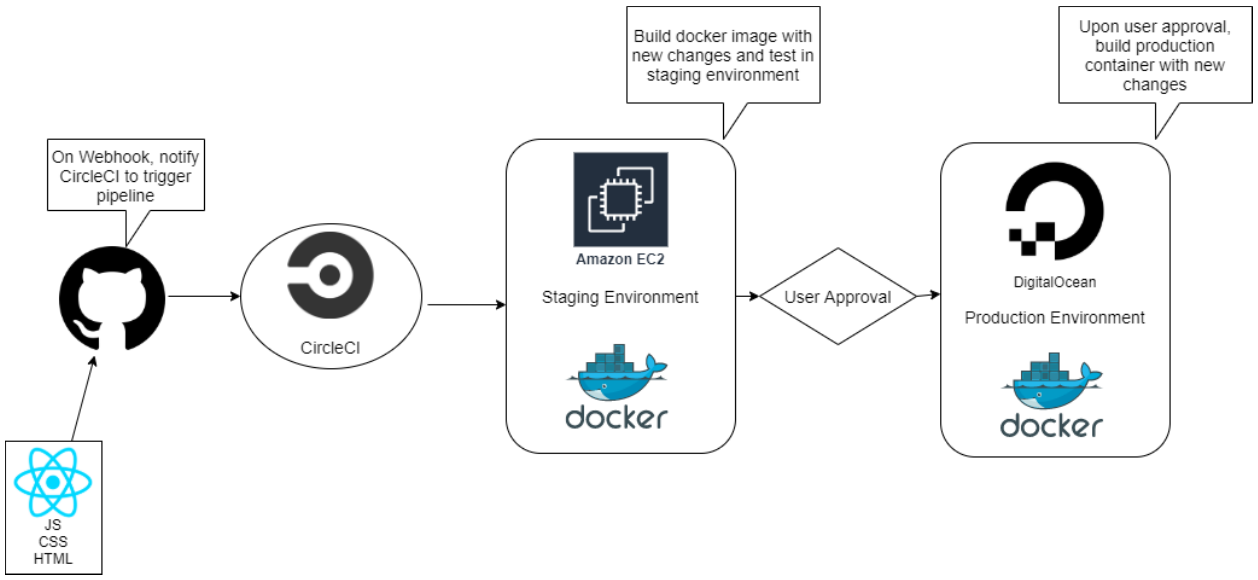
6.5 Continuous Deployment Pipeline

Now that the docker files and the deploy scripts are complete it's time to automate software deployment by creating a CD pipeline with CircleCI. This pipeline will provide a means to integrate and deploy future software updates seamlessly and with *zero downtime. With CircleCI, the term pipeline describes the highest-level configuration ran when you trigger jobs on your CircleCI projects. Beneath pipelines are workflows consisting of multiple jobs and like docker compose, these are configured using Yaml files. When you create a CircleCI account you sign up using GitHub and you can connect your account to a GitHub project's master branch. This is where the CircleCI config yaml file is located.

*Zero Downtime refers to an applications ability to update without going down or becoming unstable, so usability is completely unaffected.

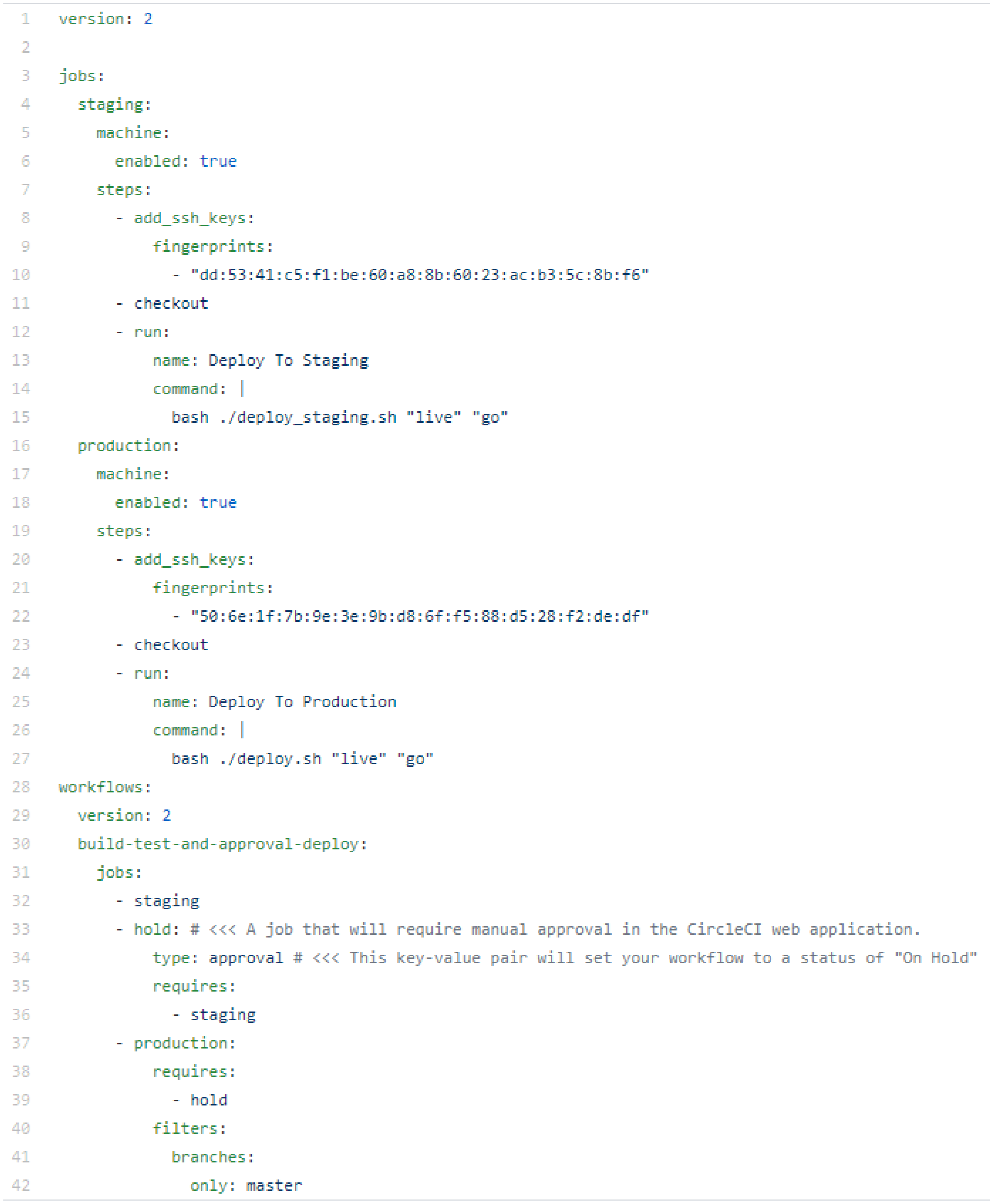
6.5.1 Workflow Configuration

Here we will look at the configuration file and explain line by line the steps involved in deploying code changes to the staging, and then the production VM. Below is an architecture diagram of the pipeline flow.


On Webhook, notify CircleCI to trigger pipeline
JS
CSS
HTML
CircleCI
Build docker image with new changes and test in staging environment
Amazon EC2
Staging Environment
docker
User Approval
Upon user approval, build production container with new changes
DigitalOcean
Production Environment
docker

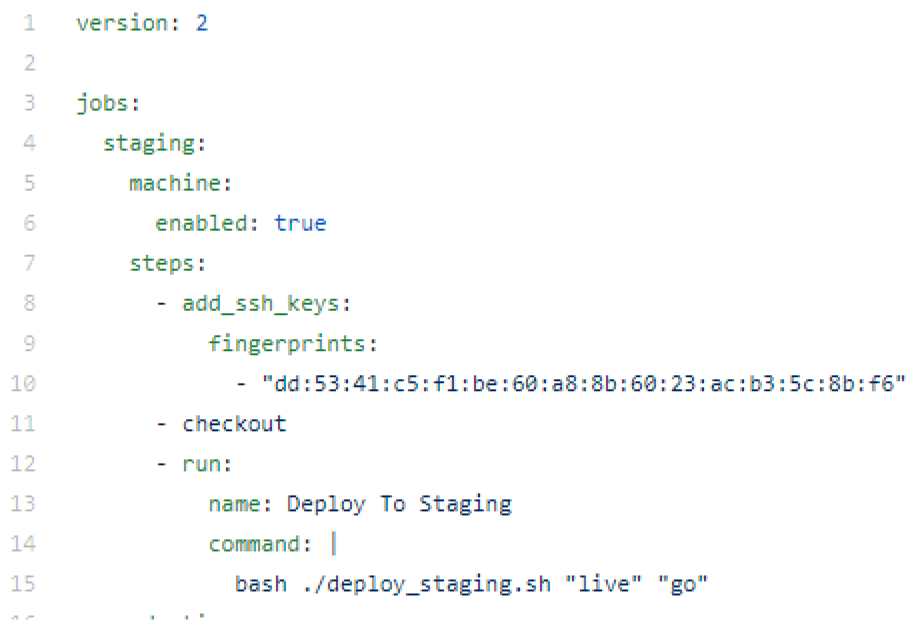
CD Architecture diagram

Below is the .circeci.config.yml file.



```yaml
version: 2

jobs:
  staging:
    machine:
      enabled: true
    steps:
      - add_ssh_keys:
          fingerprints:
            - "dd:53:41:c5:f1:be:60:a8:8b:60:23:ac:b3:5c:8b:f6"
      - checkout
      - run:
          name: Deploy To Staging
          command: |
            bash ./deploy_staging.sh "live" "go"
  production:
    machine:
      enabled: true
    steps:
      - add_ssh_keys:
          fingerprints:
            - "50:6e:1f:7b:9e:3e:9b:d8:6f:f5:88:d5:28:f2:de:df"
      - checkout
      - run:
          name: Deploy To Production
          command: |
            bash ./deploy.sh "live" "go"
workflows:
  version: 2
  build-test-and-approval-deploy:
    jobs:
      - staging
      - hold: # <<< A job that will require manual approval in the CircleCI web application.
          type: approval # <<< This key-value pair will set your workflow to a status of "On Hold"
          requires:
            - staging
      - production:
          requires:
            - hold
          filters:
            branches:
              only: master
```

Well we look at the first job here.



```yaml
version: 2

jobs:
  staging:
    machine:
      enabled: true
    steps:
      - add_ssh_keys:
          fingerprints:
            - "dd:53:41:c5:f1:be:60:a8:8b:60:23:ac:b3:5c:8b:f6"
      - checkout
      - run:
          name: Deploy To Staging
          command: |
            bash ./deploy_staging.sh "live" "go"
```

The first job is called “staging” which will deploy the application to an AWS EC2 linux server.

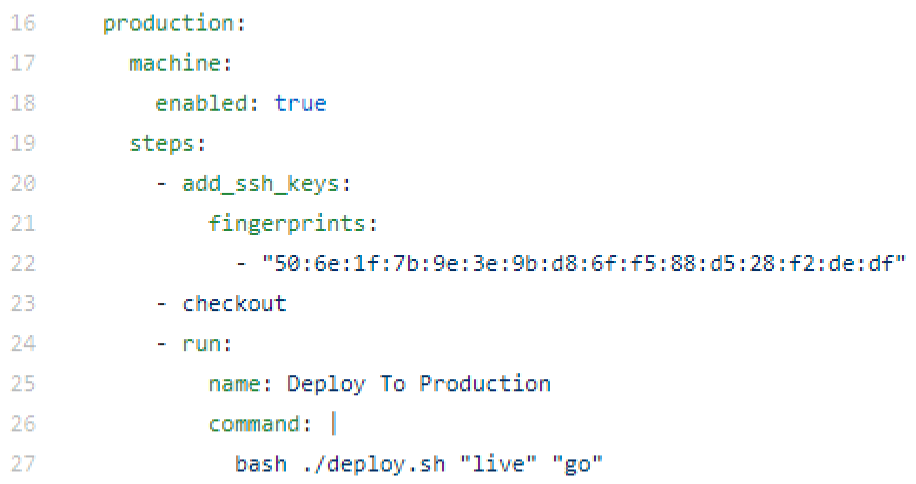
`Machine: enable: true` Is the executor type and the machine executor will run the job in a dedicated ephemeral (short lived) VM with default specifications: vCPU’s: 2, RAM: 7.5GB. Another option would be to use a docker machine however certain features that I require aren’t possible like using docker-compose with volumes.

`Steps: add_ssh_keys:` I have added an ssh key in the circleci settings to allow circleci access to my VM without inserting credentials. This way I can ssh into my VM and run my docker commands. The `fingerprint` value is created along with the key so as to keep sensitive credentials safe from version control.

`Steps: checkout:` Checks out source code to the default directory, named “~/Project/*”. When we looked at the rsync command in the deploy script this “~/Project” directory is the source directory in the command.

`Steps: run:` Used to execute command line programs like shell scripts, add environment variables. Here I run the `deploy_to_staging.sh` script and give it a `name` to make it easy to see in the logs and the to be ran, passing in the “live” and “go” parameters to perform a dry run and then do the Sync operation.

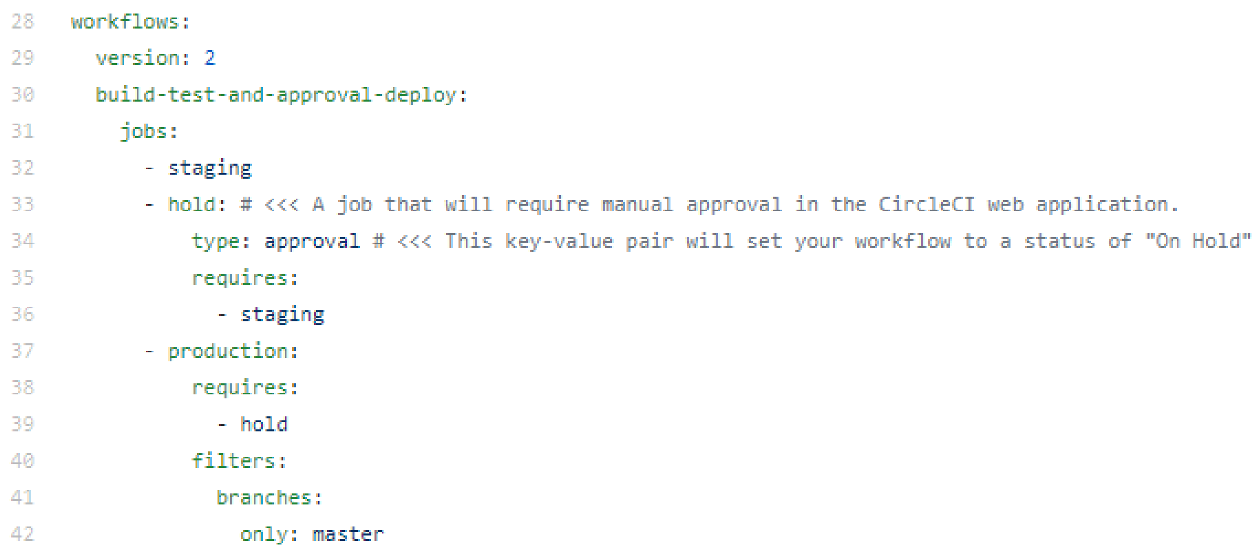
Now we will look at the second job



```yaml
production:
  machine:
    enabled: true
  steps:
    - add_ssh_keys:
        fingerprints:
          - "50:6e:1f:7b:9e:3e:9b:d8:6f:f5:88:d5:28:f2:de:df"
    - checkout
    - run:
        name: Deploy To Production
        command: |
          bash ./deploy.sh "live" "go"
```

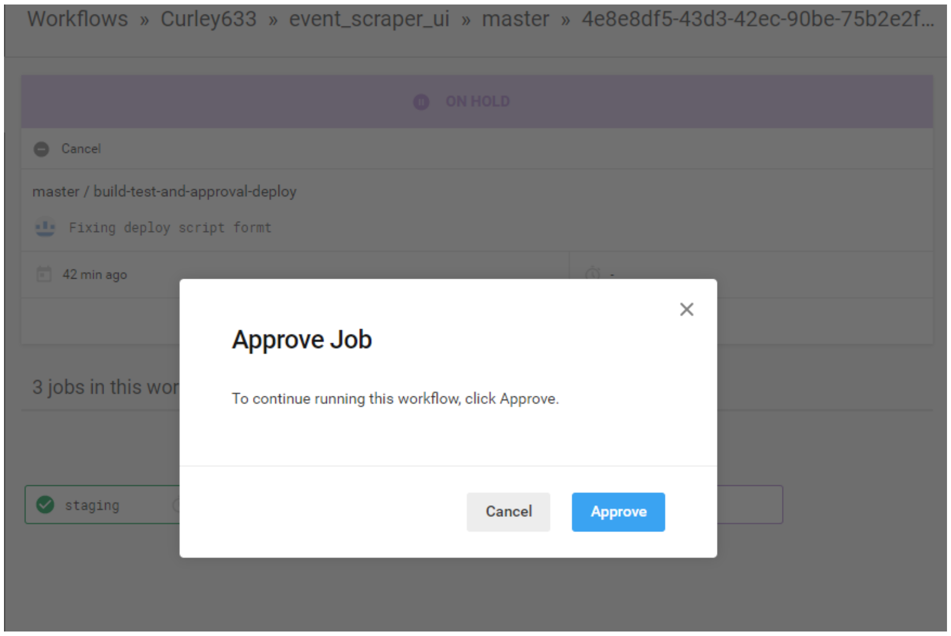
The only difference we see here is the name of the script being ran which deploys to the DigitalOcean Linux server rather than AWS.

Now we can look at the Workflow



```yaml
workflows:
  version: 2
  build-test-and-approval-deploy:
    jobs:
      - staging
      - hold: # <<< A job that will require manual approval in the CircleCI web application.
          type: approval # <<< This key-value pair will set your workflow to a status of "On Hold"
          requires:
            - staging
      - production:
          requires:
            - hold
          filters:
            branches:
              only: master
```

`build-test-and-approval-deploy:` This is the name of the workflow. We see after the first staging job the next job is called `hold:` This job pauses the workflow after the changes have been deployed to the staging VM. This allows the changes to be manually tested on an exact replica of the production VM before deploying to production. To continue the workflow the user must log into circleci and approve the hold job before continuing.


Workflows » Curley633 » event_scraper_ui » master » 4e8e8df5-43d3-42ec-90be-75b2e2f...

ON HOLD

Cancel

master / build-test-and-approval-deploy

Fixing deploy script formt

42 min ago

3 jobs in this wor

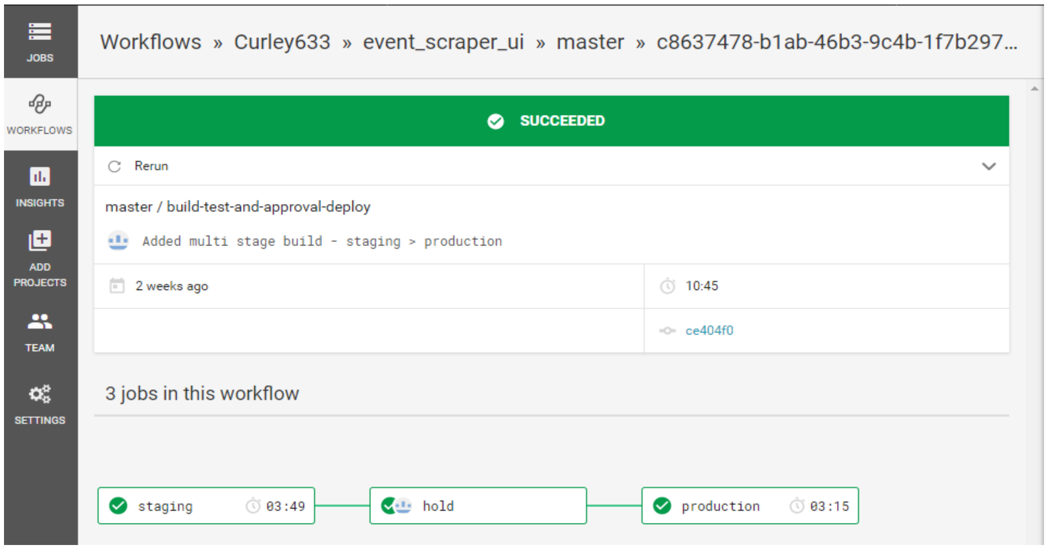
staging

Approve Job

To continue running this workflow, click Approve.

Cancel Approve

The `filters: branches: only: master` ensures only a code push or a merge to the master branch will trigger this pipeline. Below is an image of a successful run through pipeline.



Workflows » Curley633 » event_scraper_ui » master » c8637478-b1ab-46b3-9c4b-1f7b297...

SUCCEEDED

Rerun

master / build-test-and-approval-deploy

Added multi stage build - staging > production

2 weeks ago | 10:45

ce404f0

3 jobs in this workflow

staging 03:49 — hold — production 03:15

7 Problem Solving & Tutorials

This chapter describes problems I encountered while completing this project and provides solutions that hopefully others may find useful if using these tools. I will also list the tutorials I found particularly useful that helped me learn how to use some of the tools for this project.

7.1 Problems Encountered

7.1.1 Web Scraping

Problem 1 : Extracting 2 different div element values in one find_all.

Asked by me on stack overflow.

Solution: Answered by a stack overflow user here.

Problem 2 : String Parsing in Python. Commas in values being read as extra columns in the csv file.

Solution: Using String 'split', 'append' and 'replace' functions as demonstrated here

Problem 3 : In general, web scraping is susceptible to breaking if the website's structure changes, i.e. if a HTML classname is altered the script will break and the page will need to be inspected to find the new classname.

Solution: The user will have to check the page source code and identify the cause of the problem and alter the Python script.

7.1.2 Docker Problems

Problem 1 : docker-compose error - error starting 'userland proxy: listen tcp 0.0.0.0:5000: bind: Only one usage of each socket address'

Docker Commands I tried:

`docker ps -a :` to check all existing containers

`docker stop <ID> :` to stop all containers

`docker rm <ID> :` to remove all containers

`docker compose down :` to ensure nothing left up

Powershell Commands I tried:

```
Get-NetTCPConnection | where Localport -eq 5000 | select Localport,OwningProcess :
```

(finds PID of port 5000)

`Get-Process -ID 12740 | Select-Object *` : (details of app running on port 5000. It was docker)

`Stop-Process -ID <pid>` : Force (force kills the process)

Solution : Restarted docker engine and ran docker-compose successfully

Problem 2 : Cannot find module 'express' when running docker-compose

Diagnosis : Express is in the node_modules directory so node modules is not seen by docker-compose

Solution : Incorrect path in the docker-compose-prod.yml under VOLUMES:

Wrong: COPY package*.json ./express-backend/

Correct: COPY package*.json /express-backend/

Problem 3 : permission denied while trying to connect to the Docker daemon socket at unix://var

Solution : ‘sudo chmod 666 /var/run/docker.sock’ to give read and write permissions to the file.

7.1.3 Express Problems

Problem 1 : Express goes down when trying to pull from database

Diagnosis : Express container spins up but when a "fetch" takes place the express container goes down as it can’t connect to my DB.

Solution : Log into the server, enter `< sudo service postgresql start >` to start the postgres service.

7.1.4 React Problems

Problem 1 : TypeError [ERR_INVALID_ARG_TYPE] The “path” argument must be of type string. Received type undefined raised when starting react app

Symptoms : Displays on browser When spinning up the react container.

Solution : Upgrade ‘react-scripts’ node module in ‘services/react-frontend/package.json’ by manually changing the version from 3.3.0 to 3.4.0 and run "npm install".

Problem 2 : ‘TypeError: http://user:pass@domain/oauth/token is an URL with embedded credentials’

Occurs when making an API call to jenkins to trigger a build remotely.

Cause: Passing credentials in the URL is not allowed for security reasons.

Solution : Need to set the Authorization header:

```
var headers = new Headers();

headers.append('Authorization', 'Basic ' + btoa(JENKINS_USER +
":" + JENKINS_PWORD));
fetch('https://host.com', {headers: headers})
```

‘btoa’ encodes the username and password to a base64 string to avoid the credentials being accessed in the browsers devtools.

7.1.5 Server-side Python Problems

Problem 1 : ImportError: No module named <module_name>

Symptoms : Occurred during Jenkins job build while executing run_web_scrapers.sh

Cause : Jenkins pulls the scripts from GitHub but not the modules, the server has Ubuntu 18.04 with Python3.6 pre-installed but the modules or “site_packages “need to be installed on the server manually.

Solution : Locate the folder where 'site_packages' are installed on the server using this command: `python -m site` then Install the site_packages in the correct directory using 'pip3 install'. This could be automated but my scripts only use 4 site packages, so this way is quicker for now.

OR

Install packages globally using `pip3 install -h`

Problem 2 : Failed to install dot-env module

Diagnosis: Occurs when running `pip3 install dotenv`

Solution : Run `pip3 install python-dotenv`

7.1.6 Jenkins Problems

Problem 1 : Gmail blocks outgoing mail from Jenkins SMTP server in the event of job failure.

Solution :

(i) Switch on 'less secure apps' in https://myaccount.google.com/security to allow third party apps.

(ii) Enable 'displayUnlockCaptcha' by going to https://accounts.google.com/DisplayUnlockCaptcha and clicking 'Enable'.

Problem 2: Sh script can't recognise "(" or "[".

Occurs during Jenkins build when executing the run_web_scrapers.sh script in the Jenkins execute shell.

Cause/Solution : Incorrect shebang, `#!/bin/sh` instead of `#!/bin/bash` . Also use the 'bash' command instead of 'sh' command to execute the script because only bash can read the brackets contained in the script. While on some systems ‘sh’ is a symlink to ‘bash’ on others like Ubuntu it is a symlink to dash. This will cause the above error. You can check by running this command: `file -h /bin/sh` which will output something like this:

```
/bin/sh: symbolic link to dash
```

7.2 Tutorials

7.2.1 Web Scraping

Web Scraping Introduction

Describes Web scraping best practices, challenges and Caveats

How to add scraped data from a csv file into a Postgres table[5]

7.2.2 Docker

Docker for NodeJS and React, provides a great breakdown of the Dockerfile and the Docker Compose file for both dev and prod

Install and setup Docker on Ubuntu 18.04

How to write a dockerfile and docker-compose for Node/express + writing the server file

7.2.3 Jenkins

Handy article on Jenkins remote build API including passing parameters to the build

Ubuntu 18.04 server setup including firewall config and using the rsync command

How to install and setup Jenkins on Ubuntu 18.04

7.2.4 React

Information on React hooks and how they can be used to replace classes to write better components

The Reactstrap Library listing all components and the code needed to implement each one

7.2.5 Nginx

Install and Setup Nginx on Ubuntu 18.04

7.2.6 CICD

Great introductory article on the integration of a React-app with Nginx and CircleCI

8 Conclusion

This project has been by far the steepest learning curve of my degree and It was a great opportunity to delve into a specific area of the course that I was interested in. The goal was to learn how to implement a full stack application and develop a better understanding of the concepts behind CICD.

Looking back at my original project proposal I feel I covered all aspects I set out to learn. I have the confidence now to build a full stack project from the ground up. I have familiarised myself with two new languages, JavaScript and Python as well as a JavaScript framework, React which was by far my favourite aspect of the project. I have developed extensive knowledge in the area of web scraping and can see where it should and should not be used.

One area I was particularly satisfied with was using and testing API's and also how to parse API responses and extract nested data from JSON objects. I can see the benefits of using sturdy API's rather than web scraping if possible.

I have become proficient with configuring and using tools like docker, circle-ci, jenkins and understanding their capabilities. I think I made the right choice using both Circle-CI and Jenkins so I can compare the capabilities of both. The same for AWS and DigitalOcean, it was great to set up both services which will make decisions easier in the future with regard to pricing and specifications.

I have also received good feedback from non "tech-savvy" friends but who share similar music interests who find the app to be more efficient than searching Facebook pages or individual sites.

8.1 Links

Front and Backend code: https://github.com/Curley633/event_scraper_ui

Web Scraper Scripts: https://github.com/Curley633/event_scraper_project

Metal Hub Éire: http://206.189.165.104:3000/

Jenkins job to run web scrapers: http://206.189.165.104:8080/job/run_web_scrapers_test/

Prospects

I still have plenty of ideas to add to and improve the application including extra features which will improve my react knowledge, but also to restructure the architecture and use a microservice architecture which will involve dockerising postgres and jenkins. I also want to restructure the backend to use a Model, View, Controller (MVC) architecture which will allow me to move all my API calls to the backend which is better practice.

9 References

[1] 2019 Article comparing React/Angular/Vue.

[2] Docker Interesting Facts, Medium.com, https://medium.com/@tao_66792/interesting-facts-companies-and-the-use-of-docker-948baa8cf309

[3] Stack Overflow, "How to extract two div tags in BS find_all", stackoverflow.com, 2019.

[4] Stack Overflow, "split a comma separated list with links in with BeautifulSoup", Stackoverflow.com, 2009.

[5] An Introduction to Postgres with Python , Dataquest.io, 2019.