SOFTWARE PRODUCTION ENGINEERING PROJECT REPORT

SPECTRUM AUCTION VISUALIZATION WEBSITE

P SAI RAKSHITH, MVS CHARAN, S SAI SANKEERTH, P VAMSHI KRISHNA

IMT2015032,IMT2015023,IMT2015040,IMT2015516

INTRODUCTION

About The Application

Spectrum auction data is a valuable set of data many want to know about and understand something out of it. This data is available in various publications and articles, in the form of huge tables which are very hard to decipher.This website has various visualizations of Spectrum Auction data.So, it is helpful if we are able to visualize the data such that any person will be able to understand. We worked on three different visualizations. We added the extra visualizations on a pre-existing website.

Why DevOps

DevOps is crucial in continuous deployment and monitoring. Our project at it's core is a Visual Analytics project. It is of utmost importance that we can easily update the data for the visualizations and the website keeps on being deployed continuously. DevOps also allows us to roll out new visualizations.We also wanted to use continuous monitoring to gain insights from the client's usage of the website.

SOFTWARE DEVELOPMENT CYCLE

Scope of Project

- Improve the Heatmap visualization for world data.
- Develop the Treemap visualization for world data.
- Develop the India Map visualization for India data.
- Develop a complete DevOps pipeline which includes continuous integration , testing , deployment and monitoring
- Hoste the website on a AWS instance , so that it can be accessed on the public domain.

Project Architecture and Workflows

Basic Architecture

The application has two main components:

- Front-End : A Javascript-based web application which produces D3.JS based visualizations of the spectrum data.
- Back-End: Using Jersey , we developed REST-API endpoints to access data from the MYSQL server.
- Back-End Database:A MYSQL server which has all the spectrum data.

The front-end accesses data from the back-end using REST API.

Functionalities

We've worked on three different visualizations in the website, namely the HeatMap visualization, the TreeMap visualization and the IndiaMap visualization.

HeatMap

The HeatMap visualization is basically a grid structure which has countries and the bands as the axes. Each grid cell will be coloured according to the value that should be present there and gives the value when the mouse is put on the cell. Year should be chosen using a slider on the top. The bottom of the visualization has the colour spectrum which shows the colour for each interval. The images below show the top and bottom view.

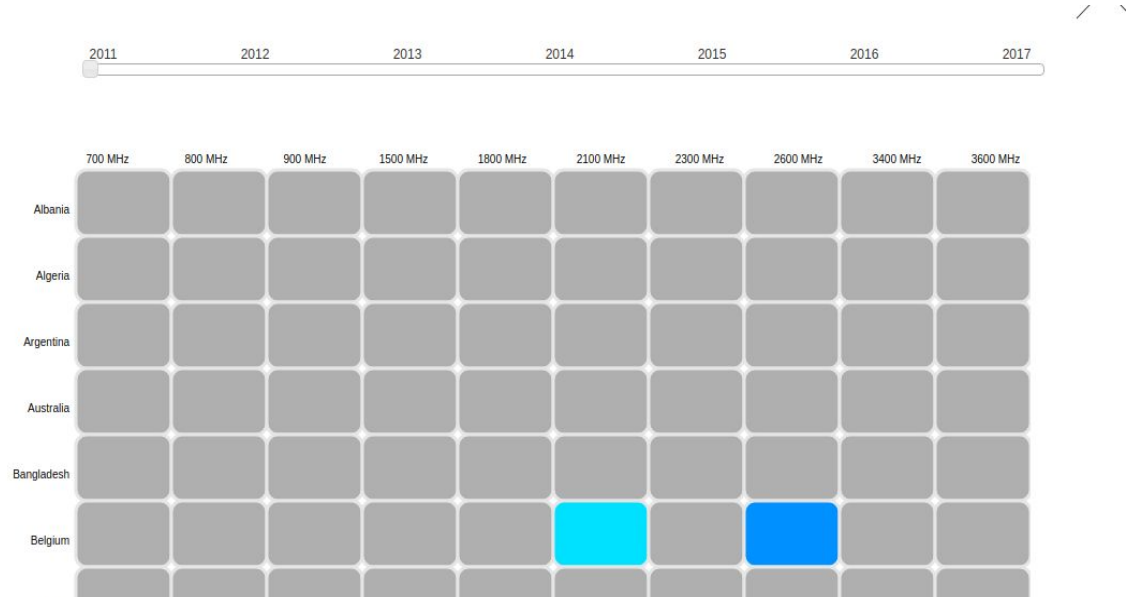


Figure 1

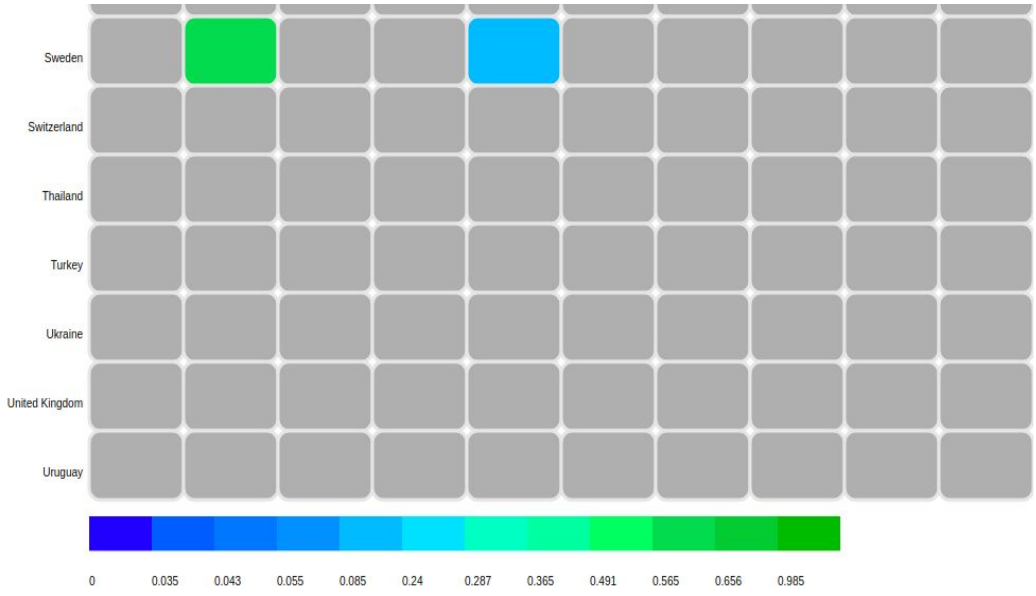


Figure 2

TreeMap

The previous visualization is good for comparing across different values but not the best one because the visualization has the blocks separated as there are many countries and bands. So, we need to figure out a way to get rid of the empty spaces in the visualization.

For this, we used treemap visualization of the same data, where we can break down the visualization at different levels. First, it shows the data split according to years, and each year block contains the sum of the values in that year. When we choose an year, it breaks down to the next level where it shows the aggregate for each country. Now, when we choose a country, it breaks down to the next level where it shows the value for each band.

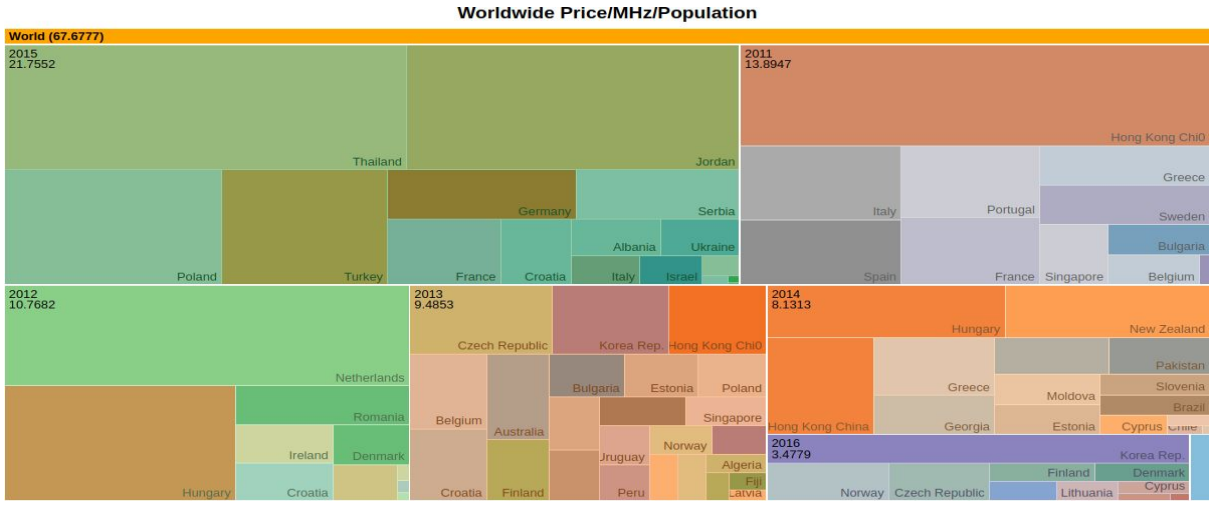


Figure 3

IndiaMap

The IndiaMap visualization helps us depict the data of spectrum auction prices of the network circles of India. For this, we used the map of India to show the states and then we can view the value band in a bar graph on the side. Year needs to be chosen beforehand to open the map.

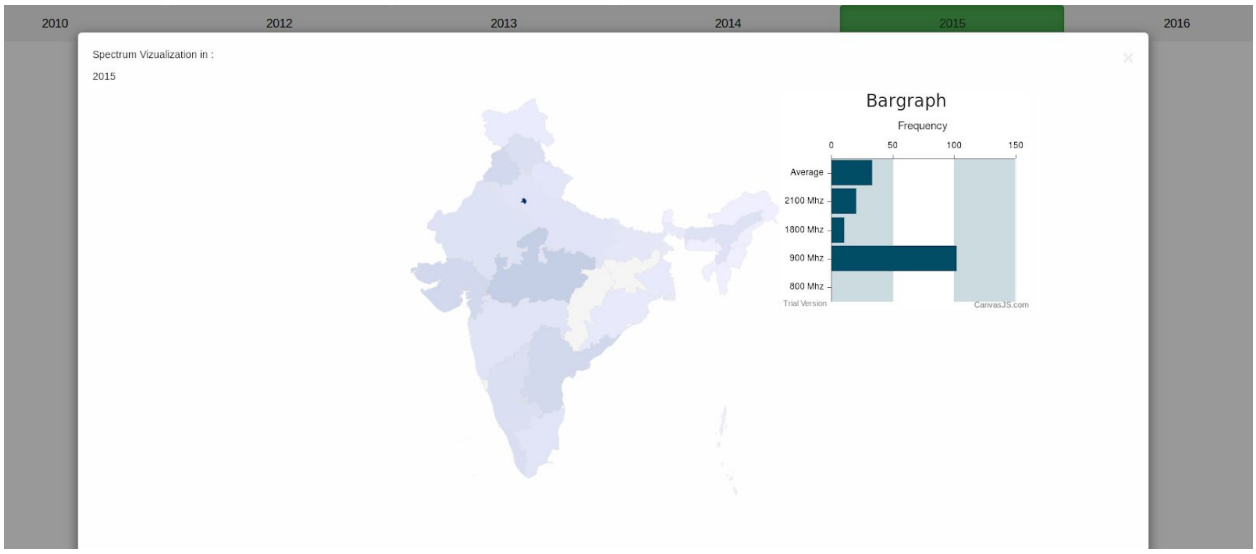


Figure 4

Rest Services

REST stands for Representational State Transfer.REST is an architectural style which is based on web standards and HTTP protocols for developing applications that can be accessed over the network. JDKs JAX-RS is the Java API for creating REST Services.

Annotations

The JAX-RS API defined annotations like @GET , @POST , @PUT , @Path etc which are used to provide meta-data around the web resource.

1) **@Path** :- The Path annotations value is used as a relative URI path for Java methods and classes. So now this becomes the path for the respective class or method. We can even embed numbers/variables in the path value which then can be used during runtime in order for the resource to respond to the request. These variables are placed inside curly brackets.

EX :- @Path("heatmap/{year}") here the year variable is embedded inside the path.

2)**@Produces** :- The Produces annotation specifies what Media Type the resource will produce.

HTTP Methods

The PUT, GET, POST and DELETE methods are typical used in REST based architectures. @GET , @PUT , @POST and @DELETE are resource annotations defined by JAX-RS which correspond to similarly named HTTP methods. We only read data from our database and hence only used GET Methods.

GET :- The GET method gives a read only access to the Resources. In the GET method the Resources are never changed.

Jersey

Jersey is an Open Source Framework used for developing RESTful web services in java that provides support for JAX-RS APIs.Rest Services in our project are written in Java using Jersey.

Functions

Three REST APIs have been written in our project which are HeatMap , IndiaMap and TreeMap.

1)**HeatMap** :- The 'heatmap' REST API call returns a stringified version of a JSON Array consisting of JSON objects with the country , band , value and year attributes based on the year which are present in the worldgriddata table in the DataBase. The year variable is embedded into the path annotation which is retrieved using the PathParam annotation. This REST API is used as a GET request.It is accessed through the url

'webapi/Rest_Service/heatmap/{year}'

2)**IndiaMap** :- The 'indiamap' REST API call returns a stringified verson of a JSONArray consisting of JSON Objects with different attributes based on Year. The Year variable is retrieved similar to the HeatMap function.This REST API is used as a GET request.It is accessed through the url

'webapi/Rest_Service/indiamap/{year}'

3)**TreeMap** :- The 'treemap' REST API returns a stringified version of a JSONArray consisting of JSON Objects with different attributes. The Year variable is retrieved similar to the HeatMap function.This REST API is used as a GET request.It is accessed through the url

'webapi/Rest_Service/treemap'

Rest call logging

The rest calls are logged using log4j library present in java

1.Timestamp(i.e date,hour,min,sec)

2.Name of the rest API called

3.Result(i.e success or failure)

4.Year parameter that is passed to the API call.

The year parameter to the treemap API call is made 0000 as it doesn't contain an year parameter.

Ex

27/Apr/2019:13:58:45 Indiamap Success 2012

27/Apr/2019 is the date

13:58:45 is the time

Indiamap is the rest API call

Success is the result

2012 is the year parameter passed to the Indiamap call

Source Control Management

We are using Git for SCM. Git helps us in tracking changes and updates in the source code and allows us to merges changes from different sources into a single project. Our entire project is currently stored as a GitHub repository. The GitHub repository contains the required source code (for building artifacts) and other configuration files.

GitHub Repo: https://github.com/SugguSaiSankeerth/DevOps

Build

Building .war file

We used maven to create the war file of the application. The war file is the executable version of the application, which has the executables of the java code and the html files to display the webpages.

Creating Docker Images

Now, we create two docker images, one for the application and the other for its database. The application image will pull tomcat and we copy the war file of the application into the desired location inside the docker image. All these instructions of the image build will be given in the file named “Dockerfile”. The instructions in this file are as follows.

```
FROM tomcat

COPY target/DevOps.war /usr/local/tomcat/webapps/DevOps.war
```

For creating the database docker image, we have to pull mysql into the image and create the database in this container. For this, we need to copy the dump file of our database into the desired location inside the docker image. All these instructions of the image build will be given in the file named “mysql.Dockerfile”. The instructions in this file are as follows.

```
FROM mysql:5.7

COPY spectrumdatadump.sql /docker-entrypoint-initdb.d
```

Artifacts

The artifacts built in the first step is the DevOps.war file. Using this and mysql dump file, we create the docker images as explained above.

Test

Testing in our project is done using the Newman CLI Tool.

Postman

Postman is an API development tool which provides functionality to build,test and modify API's .

Newman

Postman provides testing sandbox where JavaScript tests can be written and executed for an API. The postman is hooked up in the build using Newman which is the command line collection runner for postman.

Testing in Postman

Automated Testing can be done by using the Postman Collection Runner. Postman Collection is a group of saved requests which can be organized into folders.The Postman Collection Runner is used to execute a complete collection, with multiple requests and tests. Postman's Newman can access the collection through the command line to integrate with CI/CD tools.

Test Cases

There are three rest API calls in our project.All of them are GET calls and return object is json .

We have created multiple test cases for each API call and can be seen below

Figure 5

For each rest call the following test cases are created.

1.Checking if all the expected columns are present in json or not.

Ex

```
pm.test("test columns", function () {
tests["has country"] = responseBody.has("country") ;
tests["has year"]=responseBody.has("year");
tests["has band"] = responseBody.has("band");
tests["has value"] = responseBody.has("value");
});
```

Figure 6

checks whether that resulting json has the columns country,year.brand,etc or not

2.As our data is formatted year wise,the presence of data for all the years is

Ex:

```
tests["year 2011"] = responseBody.has("2011");
```

Figure 7

Checks whether resulting json has data pertaining to year 2011 or not

3.Some data points are considered at random and checked if the attributes matches with the expected values for that data point.

Ex:

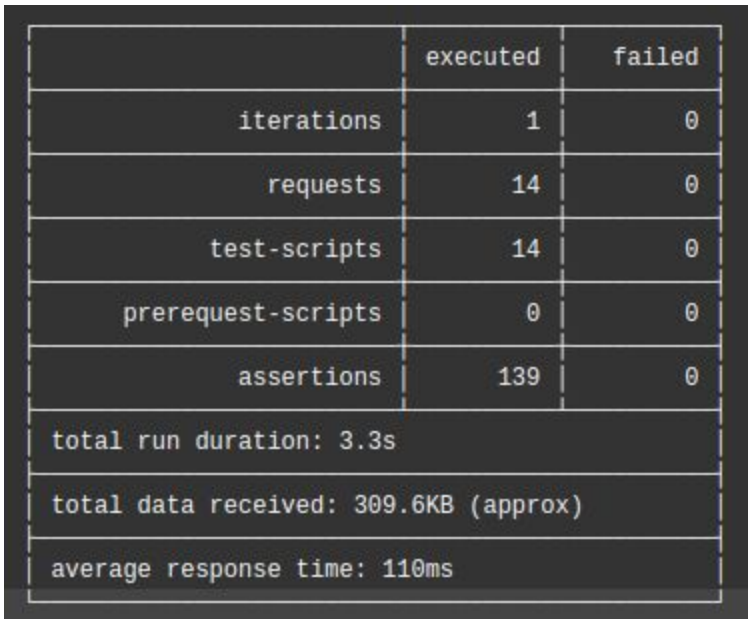
```
pm.test("test random columns 2", function () {
    var jsonData = pm.response.json();
    var testflag=0;
    for(var i=0;i<jsonData.length;i++)
    {
        var obj=jsonData[i];
        if(obj.country=="Bulgaria")
        {
            if(obj.band=="700")
            {
                testflag=testflag+1;
            }
        }
    }
    pm.expect(testflag).to.be.above(0);

});
```

Figure 8

It checks whether there is at least one json object in the returned json array which has country as Bulgaria and band as 700.

We have a total of 139 assertion-based tests which are stored in a APITests.postman_collection.json file. The Newman CLI Tool uses this collection to run the appropriate tests.



	executed	failed
iterations	1	0
requests	14	0
test-scripts	14	0
prerequest-scripts	0	0
assertions	139	0
total run duration: 3.3s		
total data received: 309.6KB (approx)		
average response time: 110ms		

Figure 9

Deploy

We are using Rundeck and an AWS instance to continuously deploy our web application.

Rundeck

We used Rundeck to configure our post-build actions. Once our docker containers are successfully built and tested , we configure a rundeck job to deploy those docker containers onto our AWS cloud instance.

Our job is configured in the following way

wget docker-compose.yml

docker pull *web_api_docker*

docker pull *mysq_docker*

sudo docker-compose up

This pulls the latest docker container and configurations files from DockerHub and GitHub. It than runs the containers.

The next step is to make this job run on the AWS instance. This is done configuring the AWS cloud instance as a node from Rundeck and telling Rundeck to run the above mentioned job on the cloud node.

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
<node name="spectrum_ec2" description="Spectrum Website" tags=""
  hostname="13.229.102.182" osArch="amd64" osFamily="unix"
  osName="Linux" osVersion="4.15.0-1032-aws"
  username="ubuntu" ssh-authentication="privateKey"
    ssh-key-storage-path="keys/ec2-key.pem"
/>
</project>
```

Figure 10

Source:https://raw.githubusercontent.com/SugguSaiSankeerth/DevOps/master/ec2_resources.xml

This is the resources.xml file that we are using to configure the cloud node. This file is auto-generated before the rundeck job is run.

AWS Instance and AWS-CLI

The ec2_resources.xml file needs to be generated every time before the rundeck job is run on the AWS instance. This is to ensure that the xml file has latest public IP of the cloud node. Every time the aws instance is stopped and started , the public IP of the cloud node gets changed. We used a shell script to generate the xml file. AWS-CLI is a command line tool which takes in AWS access token and upon authorization , can be used to obtain the IP address of the cloud node.

aws ec2 describe-instances --filters "Name=tag:Name,Values=elk_medium" --query 'Reservations[*].Instances[*].PublicIpAddress' --output text

This command returns the current public IP of the cloud instance. This value is later injected into the ec2_resources.xml

The purpose of using a shell script is to automate the DevOps pipeline ,eliminating the need to update the public IP in the xml file manually.

Monitoring

By the end of deployment stage , our web application is up and running from the aws instance and is accessible via the cloud instance's public IP.
For monitoring our web application , we are using an ELK stack which is run using docker containers. The ELK stack is run independently from the CI/CD pipeline. This enables us to continuously monitor the web application irrespective of it's deployment.
An important configuration that needs to be done before we can monitor the web application is setting up connection between our ELK containers and web-application containers. This done by configuring the ***volumes*** in the appropriate docker-compose files. Basically , the web-app containers write the log files into a local directory(in the cloud instance) and the ELK containers access the same log files to monitor and create visualizations.

The ELK stack has several important components as follows:

Figure 11

Source: https://logz.io/learn/complete-guide-elk-stack/#intro

MetricBeats and LogStash

We used MetricBeats to collect data corresponding to Docker Containers and the MYSQL Server. We use LogStash to aggregate and process the logs generated by our web application. We parse the obtained logs and then send it to ElasticSearch.

```
Filter{
grok{
        match => {
                "message" =>
"%{MONTHDAY}/%{MONTH}/%{YEAR}:%{HOUR}:%{MINUTE}:%{SECOND}%{SPACE}%{WORD:rest_call}%{SPACE}%{WORD:result}%{SPACE}%{WORD:rest_call_year}"
        }
}
}
```

LogStash uses the above mentioned custom grok filter to extract the required fields from our log messages.

ElasticSearch

Elasticsearch is the search and analysis system. It is the place where your data is finally stored and it creates the appropriate indexes for our data. These indexes are then used to obtain search and analysis results.

Time	_source
May 3rd 2019, 21:09:52.212	rest_call_year: 2016 host: d308f484f7d3 rest_call: Indiamap message: 01/May/2019:10:01:17 Indiamap Success 2016 path: /logstash/logs.out @version: 1 @timestamp: May 3rd 2019, 21:09:52.212 result: Success _id: DsRafmoB8yof7Ay_Do9- _type: doc _index: logstash-2019.05.03 _score: -
May 3rd 2019, 21:09:52.212	rest_call_year: 2016 host: d308f484f7d3 rest_call: Indiamap message: 01/May/2019:10:01:17 Indiamap Success 2016 path: /logstash/logs.out @version: 1 @timestamp: May 3rd 2019, 21:09:52.212 result: Success _id: HsRafmoB8yof7Ay_Do9- _type: doc _index: logstash-2019.05.03 _score: -
May 3rd 2019, 21:09:52.212	rest_call_year: 2016 host: d308f484f7d3 rest_call: Indiamap message: 01/May/2019:10:01:17 Indiamap Success 2016 path: /logstash/logs.out @version: 1 @timestamp: May 3rd 2019, 21:09:52.212 result: Success _id: EMRafmoB8yof7Ay_Do9- _type: doc _index: logstash-2019.05.03 _score: -
May 3rd 2019, 21:09:52.212	rest_call_year: 2016 host: d308f484f7d3 rest_call: Indiamap message: 01/May/2019:10:01:17 Indiamap Success 2016 path: /logstash/logs.out @version: 1 @timestamp: May 3rd 2019, 21:09:52.212 result: Success _id: EcRafmoB8yof7Ay_Do9- _type: doc _index: logstash-2019.05.03 _score: -
May 3rd 2019, 21:09:52.212	rest_call_year: 2016 host: d308f484f7d3 rest_call: Indiamap message: 01/May/2019:10:01:17 Indiamap Success 2016 path: /logstash/logs.out @version: 1 @timestamp: May 3rd 2019, 21:09:52.212 result: Success _id: EsRafmoB8yof7Ay_Do9- _type: doc _index: logstash-2019.05.03 _score: -
May 3rd 2019, 21:09:52.212	rest_call_year: 2016 host: d308f484f7d3 rest_call: Indiamap message: 01/May/2019:10:01:17 Indiamap Success 2016 path: /logstash/logs.out @version: 1 @timestamp: May 3rd 2019, 21:09:52.212 result: Success _id: E8RafmoB8yof7Ay_Do9- _type: doc _index: logstash-2019.05.03 _score: -

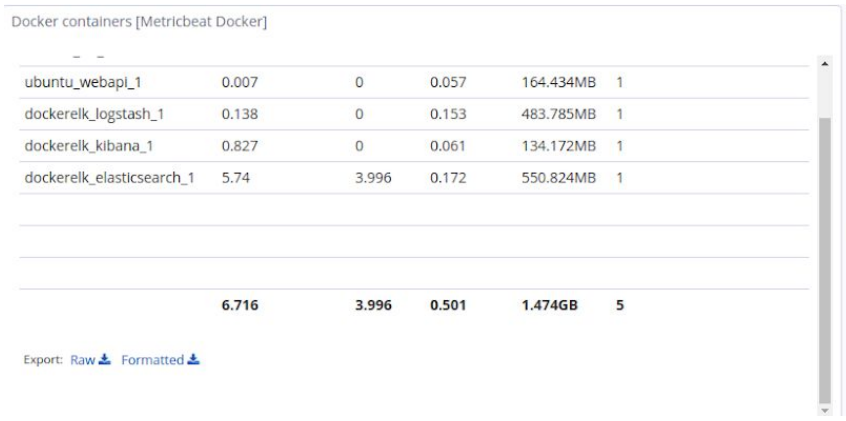
Figure 12

Elasticsearch is used to create such structured indexes upon which we can run our custom queries and also build visualizations.

Kibana

Kibana allows you to build graphs and dashboards to help understand the data.
We created the following custom dashboard to visualize a few crucial metrics in our web applications.

Docker Metrics



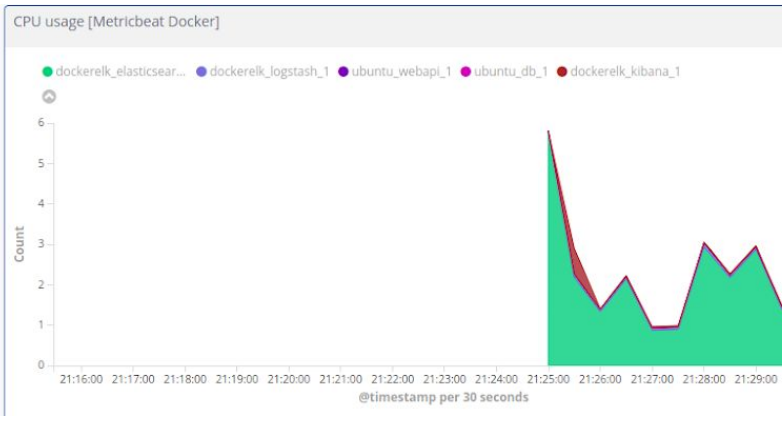


Figure 13

These help us see what docker containers are running at the moment and measure a few metrics such as RAM usage and CPU usage by the containers.

SQL Metrics

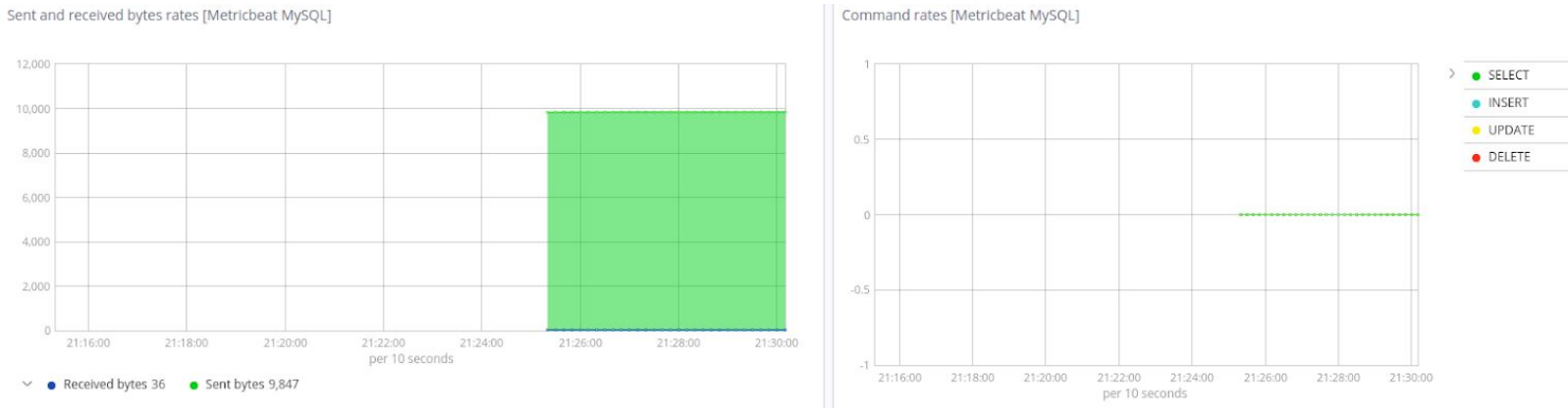


Figure 14

The SQL Metrics show us the amount data sent and received by the MySQL server and the kinds of statements processed by the MySQL server. Our web application only retrieves data from the MySQL database and hence we can only see SELECT statements and the amount of bytes sent from the server.

Web Application Metric

We can use the indexes created by ElasticSearch to create our own visualizations. We have created to pie graphs to visualize which specific years and which specific endpoints are requested by the clients.

Figure 15

We can gain very useful insights regarding our client's behaviour from such visualizations. For example , if we observe that the clients usually check data from the year 2015 , we can cache the data for optimized performance

AWS Metrics

Since , we are using an AWS EC2 instance for deployment and monitoring , multiple AWS-Metrics are also available to us via the AWS Console.

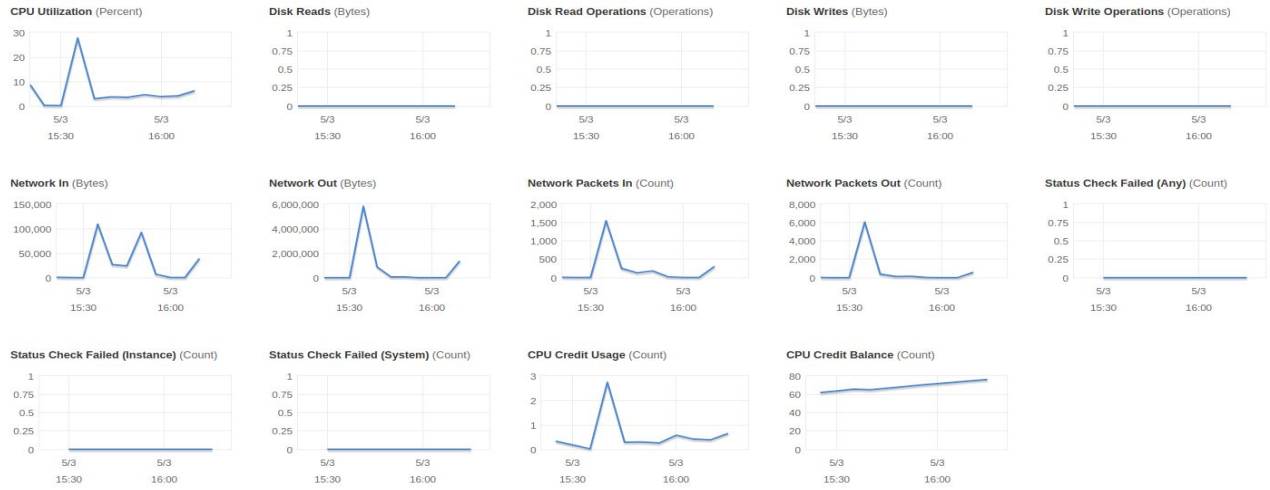


Figure 16

These further help us in monitoring the resources used up by our cloud instance.

CI/ CD PIPELINE

The CI/CD pipeline is built using combination of different tools. We use github as the VCS and this github repository is linked to a pipeline project in Jenkins. Jenkins helps us to take care of continuous integration. The Jenkins pipeline takes care of build, test and publishing to docker-hub. The Jenkins project is a pipeline project, so it needs a script file called "Jenkinsfile" which defines stages of the pipeline and the commands of each stage. The advantage of using such type of project is that we can define parallel stages. For our project, we used this feature to define two parallel processes, one for setting up the test and keep the application running for test, and the other for running the test. Once the test is done, the docker images are allowed to be pushed to docker hub repository and a rundeck job is triggered which asks a desired node to deploy the application by pulling the images from docker hub. This combination of Jenkins, Docker, NPM and Rundeck complete the CI/CD pipeline. We used Blue Ocean to visualize the flow and progress of stages. The stages are explained in detail below. Monitoring should be continuous, so it won't be included in the pipeline as we should be able to monitor the past states of the application even when the application is not running.

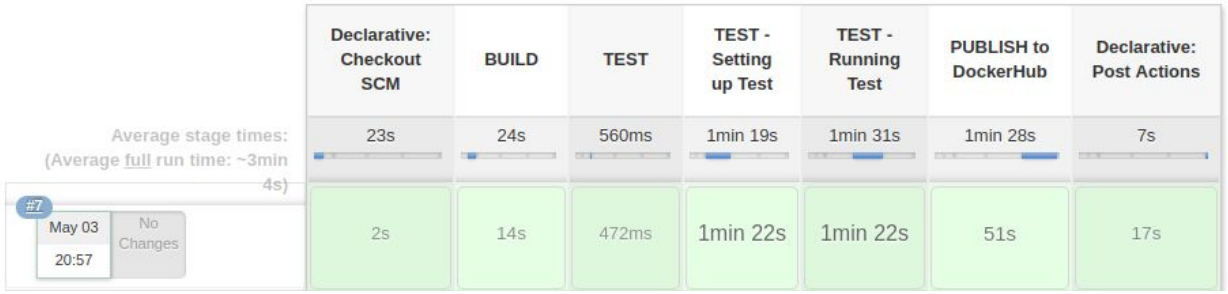


Figure 17

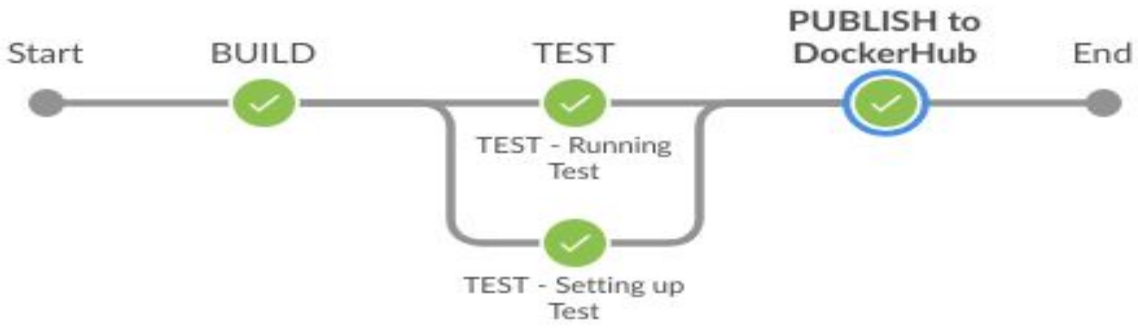


Figure 18

Checkout SCM

We have configured the Jenkins pipeline to poll the GitHub repository every 2 mins. Whenever it sees that there are any new changes in the GitHub repo , it automatically starts the pipeline. All we need to do is commit and push changes to the GitHub repo and the web application's continuous deployment is started.

Build

```
stages
{

        stage('BUILD')
        {
                steps
                {
                        sh 'mvn package'
                        sh 'docker build -t mvscharan9/spectrum_website:webimg .'
                        sh 'docker build -t mvscharan9/spectrum_website:mysqlimg -f mysql.Dockerfile .'
                }
        }
```

Figure 19

The war file of our application is created using the maven package command and then creating the docker images for our web application and mysql database using the Dockerfile and mysql.Dockerfile respectively. The mysql.Dockerfile uses spectrumdatadump.sql to create the required database.

Test

```
stage('TEST')
{
        parallel
        {
                stage('TEST - Setting up Test')
                {
                        steps
                        {
                                sh 'docker-compose up'
                        }
                }

                stage("TEST - Running Test")
                {
                        steps
                        {
                                script
                                {
                                        sh 'sleep 60'
                                        sh 'npm install'
                                        try
                                        {
                                                sh 'npm run api-tests-production'
                                                currentBuild.result = 'SUCCESS'
                                                sh 'docker-compose stop'
                                        }
                                        catch(Exception ex)
                                        {
                                                currentBuild.result = 'ABORTED'
                                                sh 'docker-compose stop'
                                                error('Test Cases Failed')

                                        }
                                }


                        }
                }
        }
}
```

Figure 20

In the pipeline for the testing, there are two sub stages which run in parallel.

TEST - Setting up Test

In this the docker containers are created using the docker compose up command

TEST - Running Test

In this ,the test cases are executed in the above created containers.

To make sure that containers are created before the testing starts,a waiting time of 60 seconds is added to this phase.

'npm install' will install all modules listed as dependencies in package.json

The test cases are executed using the command 'npm run api-tests-production' which invokes the script specified in package.json which runs the test cases

After testing, the docker containers are deleted using the docker compose down command.The pipeline continues only if all the test cases are passed,else the pipeline is aborted.

Publish to DockerHub

Once the test is successful , the docker images of both web application and mysql server are pushed to Dockerhub where the images are later retrieved for deployment.

Post-Build

```
post
{
        always
        {
                sh 'echo "Pipeline Finished"'
        }

        success
        {
                sh 'chmod +x aws_script/sample.sh'
        sh 'aws_script/./sample.sh'
        sh 'sleep 15'
                sh 'curl --location --request POST "http://localhost:4440/api/21/job/a6521c89-c162-4c3a-99c0-7f13cb08e391/run" \
  --header "Accept: application/json" \
  --header "X-Rundeck-Auth-Token: qIC6nrPc8Z0lObKmzcfA0OmKu8rmP4fI" \
  --header "Content-Type: application/json" \
  --data ""'
        }
}
```

Figure 21

This stage contains the post build actions i.e deployment.These will be executed only if all the previous stages are successful.
In the success block first the permissions of sample.sh is changed so that it can be accessed from jenkins and then it is executed.Then a sleep of 15 seconds is kept so than rundeck can take the updated resource.xml file.Then the job is triggered in rundeck by accessing its API using curl.

RESULTS AND DISCUSSIONS

Results

We have successfully divided the SDLC across different machines. We have a local computer which builds and runs the Jenkins Pipeline , has the RunDeck job configurations and instructs the cloud instance to deploy the web application. We have a different AWS cloud instance which deploys the application and is also used to monitor it.

Discussions

A few hurdles that we faced during the project and solutions we used to counter them.

1. **AWS Instance Configuration:**
   The AWS instance both deploys and monitors the web application. This requires it to run a total of 5 docker containers for continuous deployment and monitoring.
   For this we need atleast 4GB of ram and 16 GB of hard disk space.
2. **Rundeck Node Configuration:**
   The public IP of our cloud node changes every time we start and stop the EC2 instance. This would require us to manually change the public IP of the node every time before the Rundeck Job is executed. Instead of doing this , we wrote a script which use AWS-CLI tool to inject latest IP into the node configuration file.

FUTURE WORK

We have converted the json and csv files for the three visualizations we've built into sql tables and used them to complete the visualizations. But the other visualizations built previously when we received the application code still takes the data in the form of json and csv files. So, as future work, we can convert all these files into mysql tables and write REST API functions to replace the json and csv file reading into REST API calls in the javascript code. The other aspect the project is lacking is the access to pipeline. Right now, the one system which has Jenkins and Rundeck start the job. The contributors of the source code can always make changes through github and edit the pipeline by editing the

Jenkinsfile. But one physical system should always have Jenkins and Rundeck running on it for this to happen. Instead, we can have Jenkins and Rundeck configured on a server and have them always running. So the contributors can simply make a change to the code, or the pipeline itself and Jenkins adn Rundeck on the server will handle the rest of the job.

CONCLUSION

Thus, the SDLC is completed using various tools to build the pipeline and monitor. To summarize the whole, a Github repository contains the source code and acts as the version control system, Jenkins does the continuous integration by pulling the updated code and build docker images, test them using npm and then push the valid docker images to docker hub. A Rundeck job gets triggered after the updated images are pushed to docker hub. The Rundeck job asks the AWS EC2 instance to deploy the application on it by pulling the docker images and running the containers. On the EC2 instance, we can continuously monitor the application, database metrics and docker metrics using ELK.

REFERENCES

- https://www.vogella.com/tutorials/REST/article.html
- https://jersey.github.io/documentation/latest/index.html
- https://www.journaldev.com/9170/restful-web-services-tutorial-java
- https://learning.getpostman.com/docs/postman/collectionruns/integrationwithjenkins
- https://www.geeksforgeeks.org/introduction-postman-api-development/
- https://www.getpostman.com/collection
- https://documenter.getpostman.com/view/95797/rundeck/7TNfX9k#8fc45ac9-fd17-8726-afd2-f9c8b6f0aabc
- https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-configure.html
- https://github.com/deviantony/docker-elk