TP-D3: Deliverable 3
Term Project: Total-Death Coronavirus Mapping

Nate Snyder (n8snyder)
Matt Whitehead (matt1985)
Andrew Villaume (aclaude)
CS370: Operating Systems
Shrideep Pallickara
December 9, 2020

INTRODUCTION

For our semester project, our team created a map displaying the total number of deaths as caused by the Covid-19 pandemic in the United States. We implement this, primarily, with the use of leaflet - an open source JavaScript library for the creation of the map. Leaflet is deployed in a RESTful API server, running within a Docker container. The API server specifies two endpoints, one of which is a data-endpoint. In a separate container is a database, which communicates, using kubernetes, with the data-endpoint for transfer. The transferred data is then parsed, and made available using the API's second endpoint, the web server one. Ultimately, when the user makes a request to the API server, they are met with an html webpage containing the JavaScript leaflet mapping. The JavaScript is run on their machine, and further interacts with the web-server endpoint to display the map.

DATA SOURCE

Sourcing the data was the first problem our team had run-into, and an integral one in that whatever data we had access to would be shaping the implementation of our map (how the data would define the location of cases, dating of cases, etc.). As such, we needed to resolve what type of data we would be working with to move forward with project development. We quickly found that there were a number of available sources to contend with, and an equal number of unique frameworks holding the data. Initially, our team was keen on using a data lake. A repository of raw data, this is a system we had little experience with; though, the use of Amazon Web Services made accessing a particular Covid-19 data lake we had found to be seamless. Querying the data proved to be another challenge. Amazon Web Service charges a querying fee per the terabyte, and, while not particularly expensive at $5 per terabyte, considering further our team expected to query a size of data much less than that, we still opted to pursue a free alternative source. This came in the form of a data repository by the Center for Systems Science and Engineering at Johns Hopkins University. Such source provided a number of statistics relating to daily reports from around the world. Our team decided to consider only data from the United States, and proceeded to question how such data should be implemented.

DISPLAYING MAP DATA

In exploring how to implement the data available to us within our map, we run into a problem poised in our initial project proposal. That is, how to - if even we should - manipulate the coronavirus data. In this is a greater question which may provide us insight to the former. This being, what can we purposefully display with our map, and thus what data may need to be manipulated? While a case could be made for displaying updated daily statics, our team found an answer in displaying the total-deaths on a state-by-state basis. This, we felt, gave greater insight into how the disease was affecting the United States, both in showing the devastating impact and displaying the sheer volume of cases. For this, data would require no manipulation, rather, we needed only to find the number of deaths attributed to each state. Such data was not readily available in the repository provided by Johns Hopkins, thus, we simply differed to using

a new source as provided by The New York Times. Still, we would use state's latitude and longitudes included in the Johns Hopkins repository for the creation of leaflet's map markers.

SERVER

Coming into the project creating a web server seemed like a fairly large task as none of our team members had ever built one before. Yet this process turned out to be reasonably straightforward thanks to the help and advice of our Teaching Assistants. The TAs in CS370 and the video tutorials they created on Infospaces were of major assistance in developing our project overall, but in specifically setting up our first web server. Taking our TAs advice we began by creating our project using Maven. Maven is a project management tool that is used for Java-based projects build, dependency, and documentation. Maven would allow us to implement a RESTful API with Java's Spark framework. Once our default Maven project was built we added dependencies like the Maven compiler plugin and Java Spark core to ensure that we could build our project and utilize Java's Spark micro framework.

The next step was creating a Java RESTfulServer class that we would use to develop our server. We started by importing Spark's microserver and Spark's request and response objects. These would allow us to build the framework and the functionality to modify the response and request objects. We then created a Spark microserver to serve as an endpoint and listen on a specified port, which in our case was port 8080. We then added functions to configure Spark's RESTful API request routes. These routes allow for HTTP requests and responses between the server and client so status information and requested content can be shared freely. Finally all we had to do was create a main function that would call the constructor of our RESTful server. In essence our RESTful server constructor configures our Spark microserver to listen for requests and defines all the routes.

We then wanted to use our server to communicate with a separate database that held the coronavirus data. After some research we decided on MySQL. MySQL is an open-source relational database management system. How and why we chose MySQL and this type of data is discussed in its own section. In our server class we used our request routes to request coronavirus data from MySQL. Once our server received a response we created a function to parse the coronavirus data. We were specifically interested in the state name, latitude, longitude, and total deaths. Our goal was to use the latitude and longitude information to set markers on our map and use the state name and total deaths information to display above these markers. So once our data was parsed correctly we created a JavaScript object that we would use in the creation of our map and markers.

MySQL

We are utilizing MySQL as a database to store our data. At first we planned on using SQLite as our database, however we ran into issues that would have made that choice

impossible. The reason for using SQLite is that it is very simple, it does not have a user or password or server to connect to, it is just a single file which can easily be shared or moved. The issue that we ran into was that with a SQLite database existing in a container, there is no network connection to connect to the database. The normal procedure for connecting is with the path to the database file. Because we wanted the web server and database to be in separate containers, they would not have been able to communicate due to containers inability to access each other's file systems. MySQL fit our desired use of a database with a server that is toned down compared to other databases, such as Postgres.

MAP

To present our coronavirus data we felt an interactive map would be both the most user-friendly and aesthetically pleasing. Our teams' goal was to present the map once the user loads the website. The total coronavirus deaths would be defined within each state of the interactable map. The user would then have the ability to move through the map, zoom in or out, and click on each state and view the total coronavirus deaths presented as a pop-up.

Our first instinct was to use Google Maps API. Google Maps API lets you customize maps with your own images or utilize four basic map types (roadmap, satellite, hybrid, and terrain). This seemed ideal, but the first issue we encountered was that to implement a dynamic map we would need to create it using JavaScript. This was an issue as none of the members of our team had any prior experience with the JavaScript programming language. Luckily, Google points you to some excellent JavaScript tutorials [1], which would come in useful later in our project, and its documentation [2] on implementing the Maps API is very easy to follow. The first step was declaring the DOCTYPE within the HTML that allows for our content to be cross-browser compliant and render the best across platforms. We then used Google Maps API HTML formatting, so we could output a basic map in the suggested style. The last addition was our JavaScript file. This turned out to be much easier than we previously thought as Google Maps API could be inlined directly into our HTML file. This simply required us to set HTML script tags around Google's provided source. The source is the URL where Google Maps API is loaded from and includes all of the symbols and definitions we needed. It should be noted that to use Google Maps API an API key [3] is required in the src code.

Now that we had a working map, our next goal was to get our Spark Java to return our HTML file. We found our solution within the Spark documentation. First we needed to define our HTML and CSS files within the source's resources directory, so our server could have a path to access our files. We then passed the relative path to our files to our server using Spark's staticFiles.location() method [4]. Once our path was set up then all we had to do was redirect our HTML file to our URL by using the method response.redirect().

To view our new map we built our Maven project and ran our web server, but upon loading our site our map turned out to be fairly unusable. The map was overlayed with a “For development purposes only” layer. Apparently Google has changed its terms of use and only certain APIs are free. This was disappointing, but likely saved us a lot of research and headache as we were already concerned that we would not be able to implement the clickable marker functionality we wanted to with Google Maps.

We still wanted to implement a dynamic map and after more research we settled on OpenStreetMap [5]. OpenStreetMap provides community managed, open-source mapping that is free to use with citation to its contributors. This was exactly what we were looking for and, better yet, our discovery of OpenStreetMap led us to Leaflet. Leaflet is an open-source JavaScript library designed specifically for interactive maps. Leaflet is simple, clear, and has tons of documentation [6] and community resources and examples. While it was a pain going through all that hard work to implement Google Maps, discovering Leaflet and OpenStreetMaps ended up making our lives much easier in the long run.

Lucky for us, the implementation of Leaflet was extremely similar to that of Google Maps. We began by inlining script to pull the Leaflet CSS and library and JQuery library. These allow for simpler website styling and JavaScript usage. We also created a path between our server and HTML to redirect our HTML to our website. Next we diverted from our Google Maps implementation as we wanted to keep our map and marker renders in a separate Javascript file for readability and organization sake. Following the Leaflet documentation we created a map using a tilelayer from OpenStreetMap and added initial attributes like latitude, longitude, and zoom so our map view was set over the United States when our site is loaded. To add our markers we had to pass in the JavaScript object [7] we created from calling the coronavirus database. From this JavaScript object we parsed out the state, latitude, longitude, and total deaths. We used the latitude and longitude to set the location of the markers on the map and within each state and added the state name and total deaths data to the marker pop-up. This marker pop-up can be viewed by the user by clicking on the marker itself. The final step in getting our map and markers to render was to inline a source to our map JavaScript file into our HTML and wrap it in script tags.

CONTAINERS

Our application utilizes two containers. One container holds the web server and the other holds the MySQL database. For the web server container, we decided to have the build process use a two part process. The first step uses the Maven image from Docker Hub to compile the web server into an executable which is saved within the image’s file system. The second stage of the build process creates an image to run the web server. To do this, the image uses alpine java and copies the executable from the first stage image and moves it to a new image. By using this two step process, building the docker image does not depend on the builder having Maven and it

reduces the amount of dependencies and space used by the final image [8]. Initially, the build process was only a single stage of using a java image and copying the executable from a local directory. The issue with this approach was that there was a prerequisite of compiling before being able to build the image. This would be impossible in some situations, such as having GitHub automatically build and push to Docker Hub.

For the database container, we wanted to have the data be preloaded in the database. Because we are using a MySQL server, the data is stored in the /var/lib/mysql directory. Our initial thought was to copy this directory into the image. The problem with this approach is that that directory contains all databases that exist locally, which would have then been copied into the image. To avoid any local dependencies, we decided to use a similar multi-step build process as the web server container. The first stage creates the database and table and then inserts the data into the table. To do this, we used a sql file which contains all of the necessary commands. Another issue we ran into was that the /var/lib/mysql directory is defined as a volume, and docker ignores all changes made to a volume after being declared. To solve this problem, the data is stored in a temporary directory of /initialized-db. The second step of the build process copies the data from /initialized-db into /var/lib/mysql as a declaration [9].

KUBERNETES

In order to solve issues related to connectability, reliability and deployment, we utilize Kubernetes. By default, the two containers are not able to communicate with each other, which means that the web server would be unable to query the database. To solve this issue, the two containers are placed within the same pod. This makes it so they have the same localhost, which means they can easily connect through the usual means. With these containers are in the same pod, we need to provide a way to connect to the web server when sending a request from the outside. To accomplish this, we define a service which redirects incoming requests to the web server's port of 8080 [10]. Kubernetes also automatically solves the issue of the containers crashing by relaunching them when they go down.

In the future, if there is a large amount of traffic to the site, we could create replicas of the pod to provide more entry points into the app. Usually it would not be a good idea to replicate a container with a database due to potential inconsistencies between them. Inconsistencies can occur when one gets written to, then they would all need to be synced with the updated data. This would not be an issue with our app because our use of the database is purely to read from it. This means that when we do want to update the data, we would perform the same process as for updating the web server. If the pod were to be replicated, one of the potential downsides would be from having both containers in the same pod. The issue would be that there would potentially be more or fewer database instances than would be needed. For instance, if the web server is struggling to handle requests but the database is not, then creating replicas of the pod would

result in too many instances of the database. If duplication were to happen in the future, it may be worthwhile to decouple the number of web server instances from database instances.

CONCLUSION

Our term project proved to be a useful exploration of new technologies, development processes, and collaboration. Initially, in brainstorming, we explored the possibility of implementing interactive features on the map such as timelapses, the visual spread of data, and more. We quickly found this to be an overhead outside of our scope for this project, and refocused our efforts. From this, we learned the importance of having a strong framework from which to build from and remaining in frequent contact throughout the duration of the project to continually set our bases and reassess the current build. By creating such a framework and remaining in contact, we were able to continually think broadly about any design decision, thus giving us the ability to adjust our direction and work-through problems while still maintaining progress. We learned, too, the importance of starting small and building incrementally. By first developing our back-end and setting a base to build from, we could gradually move towards the realization of our project with the insurance that previous components were in working condition. With the full realization of our project and the final map, we are happy with what we have built and what we have learned.

BIBLIOGRAPHY

[1]. JavaScript Tutorial. *W3Schools*. Retrieved December 09, 2020, from https://www.w3schools.com/js/

[2]. Maps JavaScript API. *Google*. Retrieved December 09, 2020, from https://developers.google.com/maps/documentation/javascript/overview

[3]. Using API keys. *Google*. Retrieved December 09, 2020, from https://cloud.google.com/docs/authentication/api-keys

[4]. Documentation. *Spark Java*. Retrieved December 09, 2020, from http://sparkjava.com/documentation

[5]. Use OpenStreetMap. *OpenStreetMap*. Retrieved December 09, 2020, from https://wiki.openstreetmap.org/wiki/Use_OpenStreetMap

[6]. Leaflet API reference. *Leaflet*. Retrieved December 09, 2020, from https://leafletjs.com/

[7]. Maloney, A. (2014, January 23). Creating An Interactive Map With Leaflet and OpenStreetMap. Retrieved December 09, 2020, from https://asmaloney.com/2014/01/code/creating-an-interactive-map-with-leaflet-and-openstreetmap/

[8]. Baker, A. (2020, March 05). Three Ways to Create Docker Images for Java. Retrieved December 09, 2020, from https://codefresh.io/docker-tutorial/create-docker-images-for-java/

[9]. Roy, M. (2018, June 08). Creating a docker mysql container with a prepared database scheme. Retrieved December 09, 2020, from https://serverfault.com/questions/796762/creating-a-docker-mysql-container-with-a-prepared-database-scheme

[10]. Using a Service to Expose Your App. (2020, July 27). Retrieved December 09, 2020, from https://kubernetes.io/docs/tutorials/kubernetes-basics/expose/expose-intro/

[11] Almukhtar, S. et al. (2020, April 01). Missouri Covid Map and Case Count. Retrieved December 07, 2020, from https://www.nytimes.com/interactive/2020/us/missouri-coronavirus-cases.html

[12] Dong, E., Du, H., & Gardner, L. (2020). CSSEGISandData/COVID-19. Retrieved December 07, 2020, from https://github.com/CSSEGISandData/COVID-19