Dockerized Dashboard

Group 15

November 2019

Current URL for the project can be found here *(Note: still in discussion with Scott about docker support and the URL might change, will update as needed)*:

http://csci491-01.cs.montana.edu:8080/

1 Objective

The idea came to me during my junior year, I had completely missed a few school assignments and was frustrated that I had no technology to solve this problem. The project's origins were innocent, a simple program which would read a list of manually entered assignments and their correlating due dates. It ran on a raspberry pi with a junk monitor I had found. The program ran completely in the terminal and displayed information using a tui (terminal user interface). The display was a calendar which would show what assignment was due when.

I quickly became frustrated with configuration and management of my program. If I wanted anything changed I needed to ssh into the pi, update the code, and restart the program. This is when the modern version of the dashboard program was born. I envisioned a modular, dockerized, and fully salable system. Configured remotely from an admin console, each device could dynamically display information I cared about. This is the program I set out to create, and began to refer to as "dash".

2 Summary

Dash is what you would call a micro service architecture. This allows you to break down concerns into individual services, which run on a technology called docker. Docker is the new hotness in the web world, it can be thought of as way to bundle a program together into an individual unit called an *image*. Once running, this image is referred to as a *container*. Containers can communicate to each other in any way that you would make two computers talk to each other. In Dash's case, this was all done through the *http* protocol.

Dash currently has 4 images: server, apis, mongo, and nginx. Server is where most of the magic happens. It is running a simple web-server on nodejs. This

serves the ‘dashboard‘ and ‘admin‘ pages and holds all of the ReactJS code for the individual modules that are rendered to the dashboard pages. Apis is the image responsable for all communication to the outside world. It is a golang rest api that contains bindings to 3rd party apis like google calendar, firebase, and openweathermap. All requests are piped through a cache system which ensures that multiple requests wont flood the 3rd party api servers, resulting in costly bills to me. Mongo is a base image for a mongodb instance. This is where all of the data is held, the cache entries for the apis image, and the admin console settings for the server image. Finally there is a nginx image. This image only acts as a reverse proxy for the individual docker images, allowing requests to refer to the internal images as ”/apis” instead of ”/10.0.1.234” to request api data.

Because it is important that each dashboard can automatically update its display without refreshing, websockets are a core technology that is used. At its most basic level, a websocket will allow a server to communicate to clients through a tcp connection that is left open. This means that as a server, I can tell all of my clients to update their displays because new data is available. The server image uses this technology heavily, allowing instant updates of data, and configuration updates.

3 Members

All work was done by me, Matthew Nitschke, with the help of some friends, and coworkers for questions about systems architecture and docker.

4 Conclusion

Through out the life of this project, I learned an immense amount of knowledge on how to archetype a system. Docker, micro-services, websockets, large code bases, react lifecycle, and no-SQL databases were all new to me and needed to be researched.

I learned about what it takes to support software, specifically errors and error reporting. Initially I was using nodejs as my apis server, but because of how node handles errors, I was receiving cryptic information. The solution was to use golang which is very specific on its error reporting. I found a service called sentry which monitors any and all errors that are thrown. This allowed me to hunt down a few difficult lifecycle race condition bugs.

Looking back on the project, I would make a greater emphasis on preventing scope creep. The project grew too big too fast, and as a single developer I found it difficult to support everything. Given a more focused scope, I might have been able to target the problem I was trying to solve better, and end up with a better project.

5 References

- Docker docs: https://docs.docker.com/
- Express docs: "https://expressjs.com/"
- Socketio docs: https://socket.io/docs/
- Mongoose docs: https://mongoosejs.com/docs/api.html
- Mux docs: https://godoc.org/github.com/gorilla/mux
- Golang Google Calendar docs: https://godoc.org/google.golang.org/api/calendar/v3
- Microservice Archatecture: https://microservices.io/
- Sass docs: https://sass-lang.com/documentation
- Parceljs docs: https://parceljs.org/