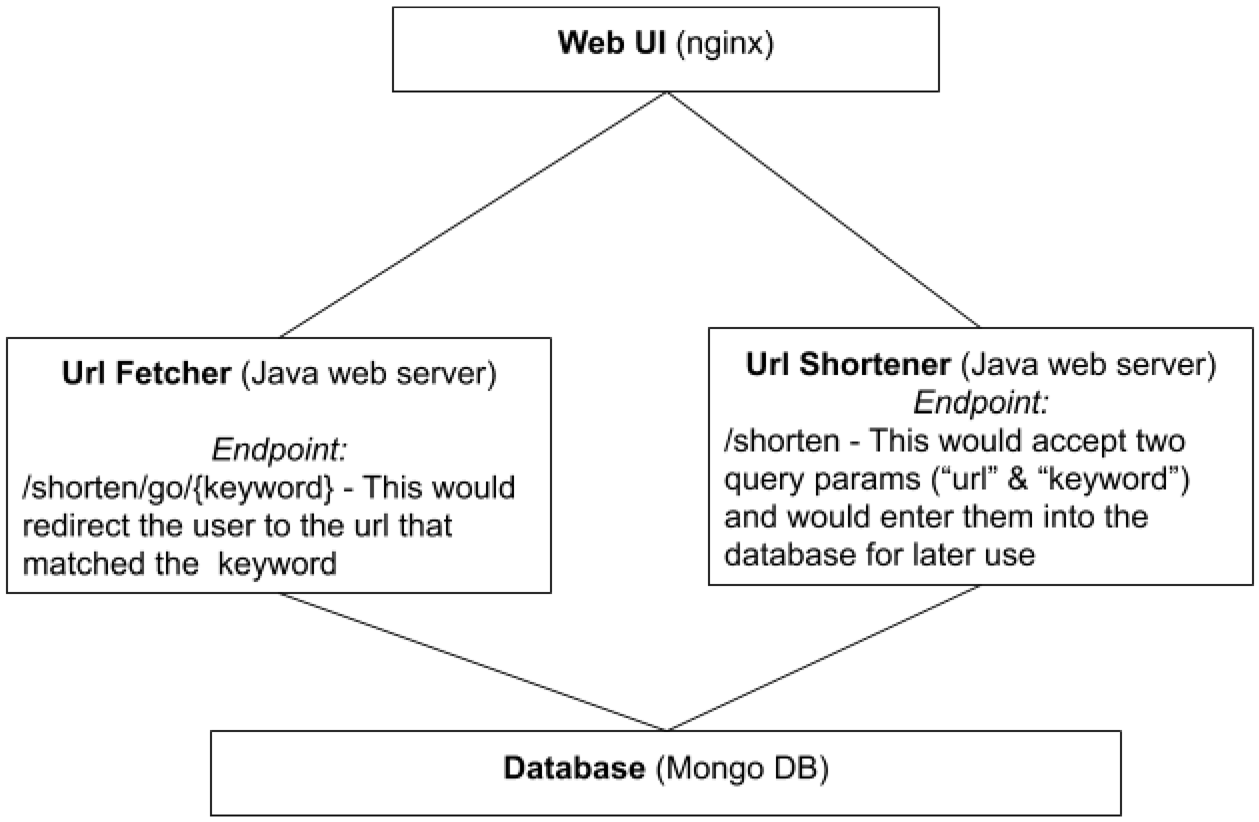
Web UI (nginx)
Url Fetcher (Java web server)
Endpoint:
/shorten/go/{keyword} - This would redirect the user to the url that matched the keyword
Url Shortener (Java web server)
Endpoint:
/shorten - This would accept two query params ("url" & "keyword") and would enter them into the database for later use
Database (Mongo DB)

Overview

Above is a visual of what our structure looked like. We opted to use nginx as the web frontend service since it came with an easy to set up Docker container. Then moving down the diagram, we opted to use two Java microservices. We assigned one of them to solely being in charge of accepting new url and keyword pairs from the user, and then storing them into the database. The other service had the job of fetching the url based on what keyword was provided, and then redirecting the user to that url. We opted for this splitting of the two main responsibilities of inserting and fetching from the database so that if we ever needed to scale the service up, we had an easy way to just scale up the part that was getting overwhelmed, i.e. if we had shortened a very popular link and our fetcher was getting requested a lot, we could scale up just the fetcher without wasting time and money scaling the shortener as well. Lastly, we opted to use MongoDB as our data layer of our project. There main reason we chose this was we had trouble setting up MySql using the pre-made docker images. Additionally, I have decent amount of experience with MySql, and I wanted to mess around with a different database structure, which Mongo provided (NoSql).

Current State

We have completed both the requirements of deliverable 2 and 3 at this point in time. We have the application fully working in docker compose infrastructure as well as it being accompanied by a fully automated profile and deployment on cloudlab.

Challenges

We experienced a decent number of challenges as apart of the project, and most of them related to the networking of Docker. It took us a bit of time to understand how the Docker networks were setup and how to access them from each different container. We found out that docker networks create an internal DNS to the network and the “domain” of each container is the name given to the container in the docker compose file. This was a major breakthrough in terms of connecting all of our services together. Figure this out required extensive google searching, as well as testing minor configuration changes until we found what worked. We did find one odd case in which this “domain as container name” did not hold up and it seem to vary depending on locally run environment vs cloudlab. This case was when we attempted to link the web ui to the two backend services. In this instance we had to use localhost instead of the container name to link them together. We also came across a bit of issues really to Cross origin scripting when attempting to connect our web ui to our backend. We solved this by forcing our backend services to accept all connections. Lastly, since the application was designed to be deployed on cloudlab, we would never know the “domain” that it would get put onto. Because of this, we opted to us the actual IP address of the node when providing “shortened” links. This causes minor bugs on most browsers as a security feature. Most browsers will not redirect to links that only have an IP address rather than a domain name. In a real production environment we would change this so it used the domain name rather than the IP address.