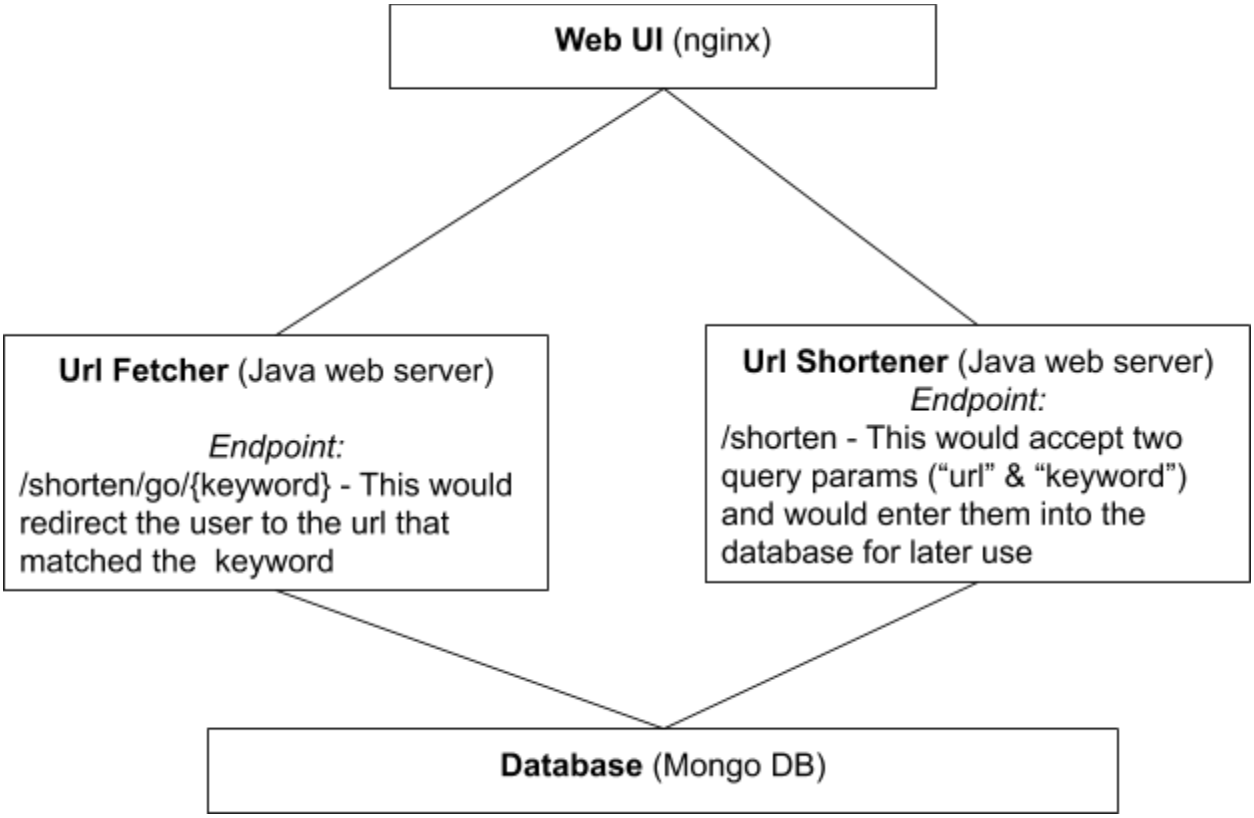
**For the sake of example, and ease of explanation we will use “ab.co” as our domain that we would setup to be pointed to our project’s ip.*

**Project Objective**

The goal of this application is to allow users to submit a long link such as a news article, and provide a keyword related to the link. For example the user could provide the link (https://www.reuters.com/article/us-space-exploration-bezos/billionaire-bezos-unveils-moon-lander-mockup-embraces-trumps-lunar-timetable-idUSKCN1SF1WI), which is a story about Jeff Bezos, along with the keyword “Bezos” and then we would return the link ab.co/go/bezos, which would allow for a shorter, easier to read link that would redirect to the story.



**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 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.

**The technical logic behind the user flow**

When a user wants to shorten their link they go to our main web page located at ab.co/index.html. Once on the page they are presented with two text input boxes as well as one button. These two inputs accept the link and the keyword, and the button will start the process of shortening the link. There is not much validation done on the input other than the fact that the input fields are not empty. Next, the web ui will send a GET request to one of the backend services dedicated to shorten links that is located on port 84, to the endpoint “ab.co/shorten”, and pass in two http query parameters of “url” which is the link provided, and “keyword” which is the keyword the user provided. Then the shortener service will connect to the database via the MongoDB Java driver, and create a record containing a JSON object formatted like “{“url”:”(User’s Link)”, “keyword”: “(User’s keyword)”}”. JSON data format was selected for the ease of serializing and deserializing it to and from the database. Once the record has been added to the database, the service will return a response that includes the shortened link that is generated based on a preformatted string of “ab.co:82/shorten/go/” + keyword. The web ui will then print this link on the screen. This process can be repeated for as many links as the user has, and they will be printed in a continuing list on the web ui. Now onto accepting the newly shortened link and redirecting the user. This is done purely as a backend service, there is no web ui component for this. Once the user connects to the

shortened link the backend fetching service, located on port 82, will separate the keyword from the link and then start to look for the record in the database that contains the keyword. If no match can be found it will display an error message. If a match is found, it will return a small bit of html from the backend that redirects the webpage to the original link that the user provided.

**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.