# **How to Refactor a Monolith to a Microservices Application**



A **Monolith application** is a single unified unit. All the components of an application is baked into a single process and share one database. The **Microservice** is more flexible because each service is a separate, independent and deployable service with its own database. A service can be worked on separately without damaging the whole application.

For the Udacity Cloud Developer Nanodegree third project, i worked on refactoring a monolith app to microservices.

For this project, the **prerequisites** are PostgreSQL client, NodeJS v12.14 - 14.15, Ionic CLI utility v6, Docker desktop, [**Travis account**](https://www.travis-ci.com/) (using your GitHub account), AWS CLI v2 - create an IAM user with admin privileges, copy it's access key and configure locally, kubectl CLI utility (to communicate with Kubernetes clusters).

aws configure

# Run a sample command

aws iam list-users

Fork and clone the [**project starter code**](https://github.com/udacity/cd0354-monolith-to-microservices-project.git).

psql -h mypostgres-database-1.c5szli4s4qq9.us-east-1.rds.amazonaws.com -U [your-username] postgres

**STEP 1**

* Create an S3 bucket to store the pictures that will be uploaded later. Add the necessary bucket policy and CORS configuration in the permissions tab.
* Create a PostgreSQL database using AWS RDS. This will be used to store the metadata e.g., User credentials.
* Edit the RDS' security group inbound rules to allow traffic from anywhere.
* Test the connection from the local PostgreSQL client.
* To allow the app access to the PostgreSQL database and S3 bucket created earlier, set up the environment variables.

source set\_env.sh

echo $POSTGRES\_USERNAME

echo $URL

For Linux:

setx POSTGRES\_USERNAME

setx POSTGRES\_PASSWORD abcd1234

setx POSTGRES\_HOST abcdefshg-database-1.c5szli4s4qq9.us-east-1.rds.amazonaws.com

......

For windows:

I used windows, so using the setx command stored them permanently in my environment variables. As I wouldn't want my credentials in my git repository, I added the set\_env.sh filename to the .gitignore file of the project directory.

**STEP 2**

Run the application locally. Start the backend application before the frontend application since it depends on the backend's API.

* Open a new terminal and navigate to the project directory and run these commands:

cd udagram-api/

npm install .

npm run dev

After running the backend, visit the [**http://localhost:8080/api/v0/feed**](http://localhost:8080/api/v0/feed) in your web browser to verify that the application is running. You should see a JSON payload

No alt text provided for this image

* Open a new terminal, navigate to the project directory and run these commands

cd udagram-frontend/

npm install -f

ionic build

//Run your application locally

ionic serve

A link appears after running the ionic serve. Visit **http://localhost:8100** in your web browser to verify that the application is running. You should see the Udagram web interface.

**Leave the udagram-api running before running the udagram-frontend**. This is because the frontend depends on the backend's API.

**STEP 3**

This part of the project is to refactor the monolith application to microservices

* Create two new directories with the names: ./udagram-api-feed and ./udagram-api-user
* Copy the backend code (udagram-api) into the above individual services.

.

├── mock # Common and no change

├── node\_modules # Auto generated. Add this into the .gitignore and .dockerignore

├── package-lock.json # Auto generated. Do not copy.

├── package.json # Common and no change

├── src

│ ├── config # Common and no change

│ ├── controllers/v0 # Keep either/feed or/users. Delete the other folder

│ ├── index.router.ts #Remove code related to other (either feed or users)

│ └── model.index.ts #Remove code related to other (either feed or users)

│ ├── migrations #Remove the JSON related to other (either feed or users)

│ └── server.ts # Remove code related to other (either feed or users)

├── Dockerfile # Create NEW, and common

└── .dockerignore # Add "node\_modules" to this file

Each of the services should have the following directory structure:

* The dockerfiles for the above 2 services should look like this

## Build

FROM beevelop/ionic: latest AS ionic

# Create app directory

WORKDIR /usr/src/app

# Install app dependencies

# A wildcard is used to ensure both package.json AND package-lock.json are copied

COPY package\*.json ./

RUN npm ci

# Bundle app source

COPY . .

RUN ionic build

## Run

FROM nginx:alpine

COPY --from-ionic usr/src/app/www /usr/share/nginx/html

FROM node:12

# Create app directory

WORKDIR /usr/src/app

# Install app dependencies

# A wildcard is used to ensure both package.json AND package-lock.json are copied where available (npm@5+)

COPY package\*.json ./

RUN npm ci

# Bundle app source

COPY . .

EXPOSE 8080

CMD [ "npm", "run", "prod" ]

* Refactor the frontend application. To do this, add a Dockerfile to the udagram-frontend directory

**NB**: *Add .dockerignore file to each of the services above and mention node\_modules in that file. This ensures that the node\_modules file will not be included in the Dockerfile COPY command.*

* Create a new directory ./udagram-reverseproxy. This helps add a layer between frontend and backend APIs so that the frontend uses only a single endpoint.

FROM nginx:alpine

COPY nginx.conf /etc/nginx/nginx.conf

* Create a Dockerfile as:
* Create a nginx file. This will expose the 8080 port

worker\_processes 1;

events { worker\_connections 1024; }

error\_log /dev/stdout debug;

http {

sendfile on;

upstream user {

server backend-user:8080;

}

upstream feed {

server backend-feed:8080;

}

proxy\_set\_header Host $host;

proxy\_set\_header X-Real-IP $remote\_addr;

proxy\_set\_header X-NginX-Proxy true;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

proxy\_set\_header X-Forwarded-Host $server\_name;

server {

listen 8080;

location /api/v0/feed {

proxy\_pass http://feed;

}

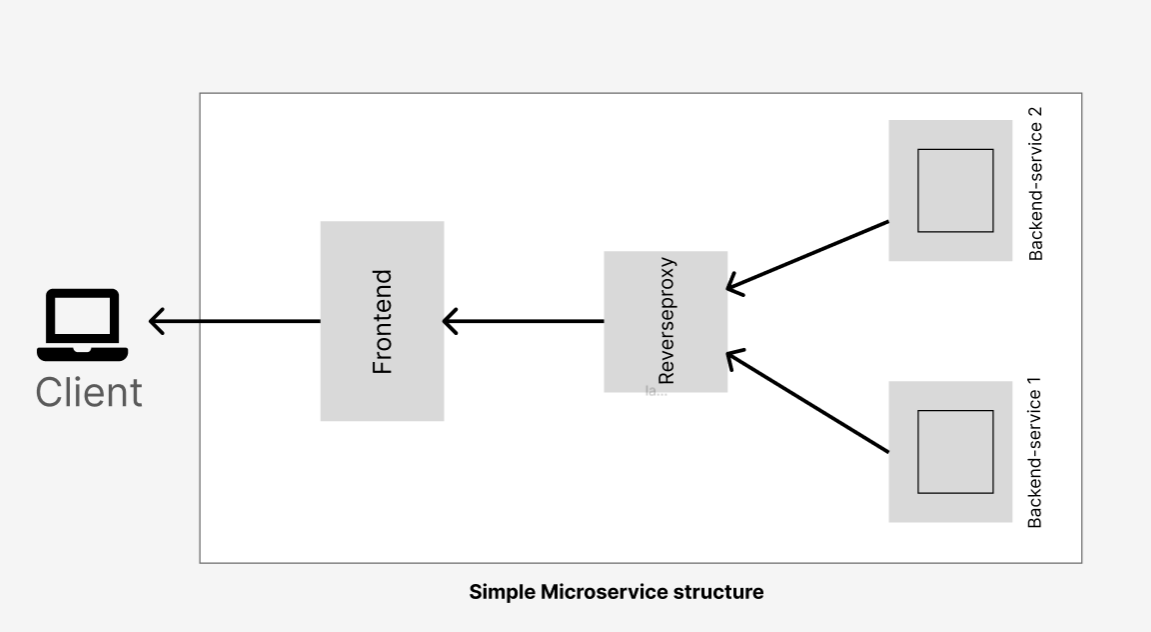
location /api/v0/users {

proxy\_pass http://user;

}

}

}



.

├── udagram-api-feed

│ └── src

├── udagram-api-user

│ └── src

├── udagram-frontend

│ └── src

└── udagram-reverseproxy

The project directory had the following structure:

* Create images. In the parent directory, create a **docker-compose-build.yaml** file. This creates an image for each individual service.

version: "3

services:

reverseproxy:

build:

context: ./udagram-reverseproxy

image: reverseproxy

backend\_user:

build:

context: ./udagram-api-user

image: udagram-api-user

backend\_feed:

build:

context: ./udagram-api-feed

image: udagram-api-feed

frontend:

build:

context: ./udagram-frontend

image: udagram-frontend:local

docker-compose -f docker-compose-build.yaml build --parallel

// Remove unused images

docker image prune --all

* Run the command below in the project directory

**Tip:** *The docker-compose command didn't work for me, so I took out the hyphen in docker-compose and ran the command. It worked!!*

docker compose up

* Create containers using the images created earlier. Create a **docker-compose.yaml** file in the project directory then run this command

**Tip:** *Ensure the images are built and containers running by using this command*

docker images

* Visit [**http://localhost:8100**](http://localhost:8100/) in your web browser to ensure the application is running. You should see the Udagram interface.

**STEP 4**

* Log into [**http://hub.docker.com/**](https://hub.docker.com/) and create four public repositories corresponding to your local docker images. The names of the repositories must be exactly the same as in the *docker-compose-build.yaml* file.
* Create a *.travis.yml* file in the project directory. The file should contain codes to automatically read Dockerfiles, build and push images to DockerHub.

language: node\_js

node\_js:

  - 13

services:

  - docker

# Pre-testing installs

install:

  - echo "nothing needs to be installed"

# Scripts to be run such as tests

before\_script:

  - echo "no tests"

script:

  - docker --version # print the version for logging

  - docker build -t udagram-api-feed ./udagram-api-feed

  - docker tag udagram-api-feed $DOCKER\_USERNAME/udagram-api-feed:latest

  - docker build -t udagram-api-user ./udagram-api-user

  - docker tag udagram-api-user $DOCKER\_USERNAME/udagram-api-user:latest

  - docker build -t udagram-frontend ./udagram-frontend

  - docker tag udagram-frontend $DOCKER\_USERNAME/udagram-frontend:latest

  - docker build -t reverseproxy ./udagram-reverseproxy

  - docker tag reverseproxy $DOCKER\_USERNAME/reverseproxy:latest

after\_success:

  - echo "$DOCKER\_PASSWORD" | docker login -u "$DOCKER\_USERNAME" --password-stdin

  - docker push $DOCKER\_USERNAME/udagram-api-feed

  - docker push $DOCKER\_USERNAME/udagram-api-user

  - docker push $DOCKER\_USERNAME/udagram-frontend

  - docker push $DOCKER\_USERNAME/reverseproxy

* Push changes to GitHub to trigger your build on Travis. It might fail showing *Udagram-backend failed* after 3-4 minutes. Go to settings > Environment variables and add your docker username and password. Restart the build. It should show a successful job.
* Verify that the newly built images are available in your DockerHub account.

**STEP 5**

apiVersion: eksctl.io/v1alpha5

kind: ClusterConfig

metadata:

  name: Cluster

  region: us-east-1

* Create a cluster.yaml file using EKSCTL. I left out the node groups in the cluster.yaml file. I created the node groups in the AWS console.

eksctl create cluster -f Cluster.yaml

* Run the command to create the cluster. It should take 15-20 minutes to create the cluster.

kubectl create configmap myconfig --from-literal=POSTGRES\_HOST=abcdef-1.c8.us-east-1.rds.amazonaws.com --from-literal=POSTGRESS\_DB=kojos --from-literal=AWS\_BUCKET=arn:aws:s3:::abcde --dry-run=client -o yaml

* The newly created cluster should be available in the AWS console. Go to the cluster, scroll down to Node groups > Edit group. Create the node groups with the necessary policies.

kubectl get nodes

* Create a kubeconfig for Amazon EKS. Once kubectl is configured to communicate with your EKS cluster. Run this command
* Create *env-configmap.yaml* and save all your non-confidential environments variables in that file. PostgreSQL username and password are not stored in this file.
* Create an *env-secret.yaml* file. This has the Docker and Postgres username and passwords.

kubectl create secret generic secrets --type=string --from-literal=POSTGRES\_USERNAME=chrysanthemum --dry-run=client -o yaml

* Create an *aws-secret.yaml* file to store your AWS login credentials using the previous command but this has the aws\_access\_key\_id and aws\_secret\_access\_key. Copy the keys and paste them into an encoding field website to convert them to base64.

kubectl create service clusterip backend-feed --tcp=8080 --dry-run=client -o yaml

kubectl create deployment --image=chrysanthemum/udagram-backend-api .....

kubectl apply -f env-configmap.yaml

kubectl apply -f env-secret.yaml

kubectl apply -f aws-secret.yaml

* Apply the variables and secrets.
* Create deployment and service files for each service.

kubectl apply -f backend-feed-deployment.yaml

kubectl apply -f backend-feed-service.yaml

// Do the same for the other files

* Apply the deployment and service files for all four services.

kubectl get pods

//Check deployments

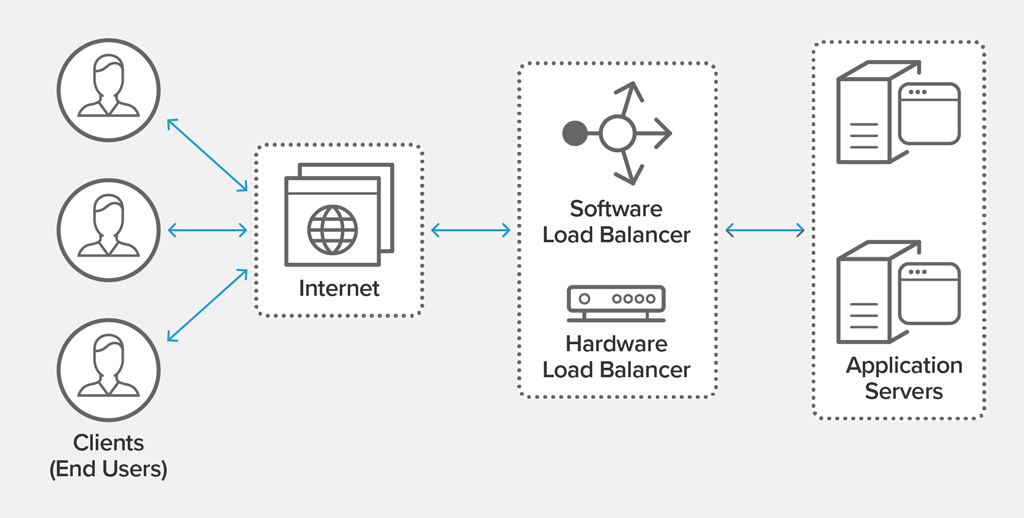
kubectl get deployments

* Verify the pods are running and deployment is fine
* Connect Kubernetes services to connect with the application. Create a service object that exposes the frontend and reverseproxy deployments. The command below creates an external load balancer and assigns a fixed, external IP to the service.

kubectl expose deployment frontend --type=LoadBalancer --name=portfrontend

kubectl expose deployment reverseproxy --type=LoadBalancer --name=portreverseproxy

The load balancer distributes incoming traffic across several backend servers. This reduces the strain on each server and makes the servers more efficient and function properly.



kubectl get services

This command lists the services and the two services created with the previous command with their IPs.

* Use the IP to access the application.

**STEP 6**

* Update the ***udagram-frontend/src/environments/environment.ts***file. Replace the word localhost in the ***http://localhost:8080/api/v0*** string with the external IP of the reverseproxy. Do the same for ***udagram-frontend/src/environments/environment.ts*** string reverseproxy external IP.
* Build a new frontend image and push to DockerHub. Run the command from the ./udagram-frontend directory

docker build . -t [DockerHub-username]/udagram-frontend:v1

docker push [DockerHub-username]/udagram-frontend:v1

* Redeploy the new frontend image to the Kubernetes cluster. Update the image tag in the frontend-deployment.yaml file

# Run these commands from the /udagram-deployment directory

kubectl set image deployment frontend frontend=[DockerHub-username]/udagram-frontend:v1

* Check the deployed application on the external IP of the portfrontend.

Now the application should run. The Udagram interface will pop up.

**Solution to the project can be found in my**[**GitHub repo**](https://github.com/BenedictaUche/cd0354-monolith-to-microservices-project.git)**.**