Micro-services Boot Camp

This boot camp involves an exercise in designing an application specifically built for deploying and maintaining in modern cloud environments. As part of this exercise, we will try to solve a very simple problem by designing the solution as a collection of loosely coupled services and orchestrating them within a self contained and isolated network.

Purpose

This boot camp serves twin purposes, that is to initiate new comers, to micro-services by demonstrating the usefulness of micro-services as well as demonstrating the typical challenges in deploying the services in a cloud native environments. It also demonstrates the usefulness of containerisation in the context of micro-services and cloud native deployments.

The Problem

The problem is to generate and mine `Lucky Hashes`, which are essentially hex encoded SHA-256 hashes starting with "0". The hashes are computed from secure random tokens (hex encoded random bytes) of a given size and stored in a data store. The problem also includes **real-time** display of the mining process.

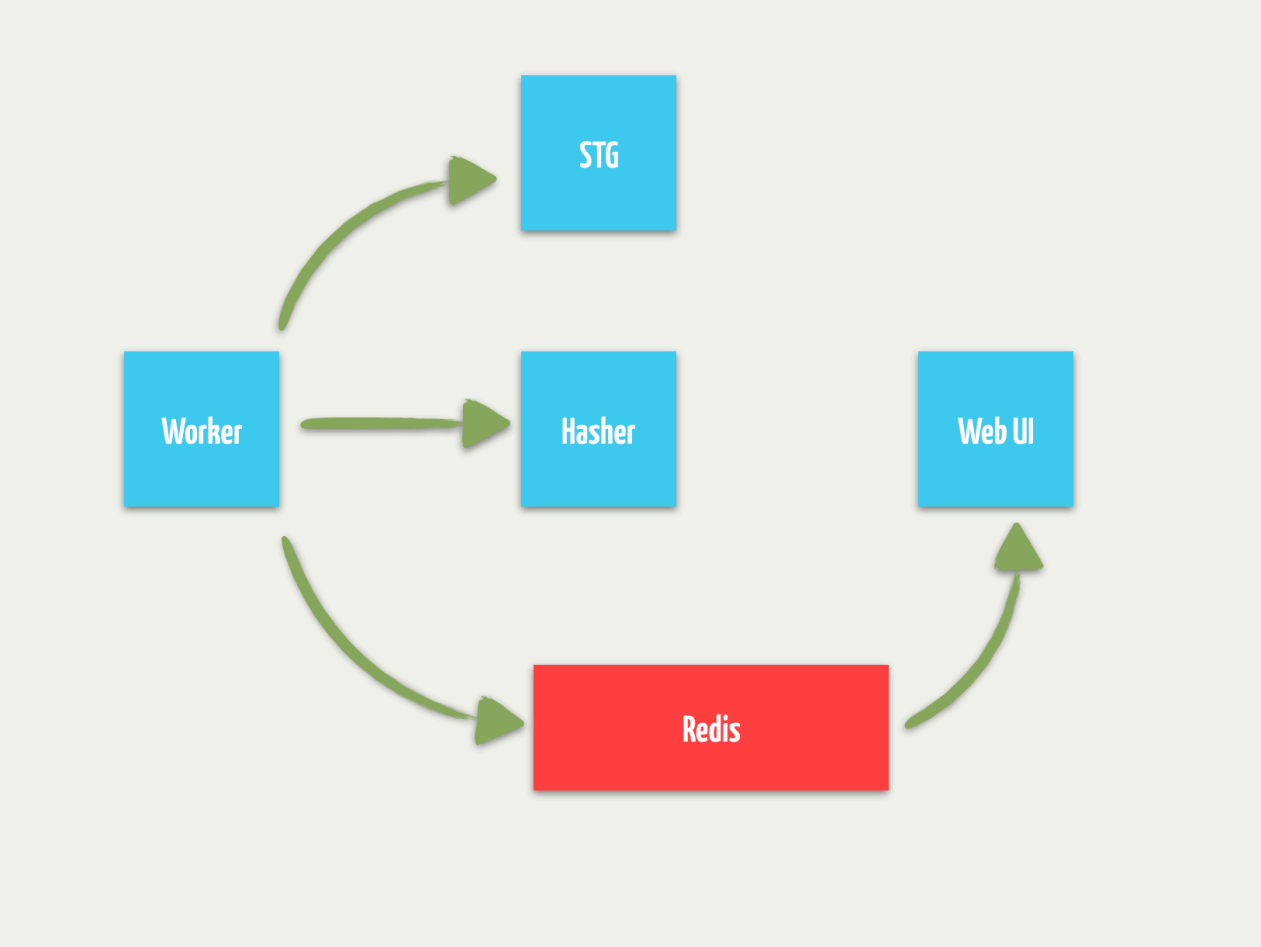
Applying Microservices to solve the problem

The primary concern of this project is to develop an application composed of four micro-services deployed in a self contained and isolated network, which in unison are responsible for mining the `Lucky Hashes` and displaying the progress of mining. The required micro-services are listed below:

1. Secure Token Generator `stg` - This is a RESTFUL webservice responsible of generating secure random tokens.
2. `Hasher` - This too is a RESTFUL webservice which takes a token as input, computes a SHA-256 hash of the token and responds with the hex encoded representation of the hash.
3. `Worker` - This is a service which reacts to timer events to perform the task of storing and mining of the `Lucky Hashes` by orchestrating the other micro-services.
4. `WebUI` - This is a service responsible for dispalying the rate of mining through a web page by updating a line chart in the web page at real-time.

The hashes should be stored in redis.

The diagram below depicts the high-level architecture of the solution:



Services:

Stg - Secure Token Generator

This service should be implemented as a restful webservice adhering to the following specification:

Request:

Property	Value
VERB	GET
URL	`http://stg/tokens/:size`
Content-Type	`application/json`

> NB: `:size` is a restful url parameter for specifying the size in bytes of the random token before hex encoding.

Response:

Property	Value
Content-Type	`application/json`
Body	{ "token": "b35b06bd3f24237...074aa09c5263d"}

This service should be implemented in Go Programming language. Use of any opensource web framework is permitted. Make sure the implementation includes the following:

1. Unit tests
2. Build script `Makefile`
3. `Dockerfile`
4. `.gitignore`
5. README.md containing all relevant information like build and setup instructions and instructions of starting the service, etc.

Hasher

This service should also be implemented as a restful webservice adhering to the following specification:

Request:

Property	Value
VERB	POST
URL	`http://hasher/`
Content-Type	`application/json`
Body	{ "token": "b35b06bd3f24237...074aa09c5263d"}

Response:

Property	Value
Content-Type	`application/json`
Body	{ "hash": "73483cb06571211730db...abc336c14349170af6a"}

> N.B: The token received as input argument (in request body) should be hex decoded before and computing the hash and the computed hash should be hex encoded before forming the response body.

This service should be implemented in Go Programming language. Use of any opensource web framework is permitted. Make sure the implementation includes the following:

1. Unit tests

2. Build script `Makefile`
3. `Dockerfile`
4. `.gitignore`
5. README.md containing all relevant information like build and setup instructions and instructions of starting the service, etc.

Worker

This should be implemented as a standalone worker service which reacts to timer events and performs the following tasks:

1. Call `stg` service to generate secure token.
2. Call `hasher` to generate hash of the token.
3. Check if the hash is `Lucky`
4. Store the hashes in `redis`
5. Publish notification data in a redis topic/channel.

The service should be implemented in Go Programming language. Use of any opensource library for communicating with redis (e.g: `go-redis`) is permitted. Make sure the implementation includes the following:

1. Unit tests
2. Build script `Makefile`
3. `Dockerfile`
4. `.gitignore`
5. README.md containing all relevant information like build and setup instructions and instructions of starting the service, etc.

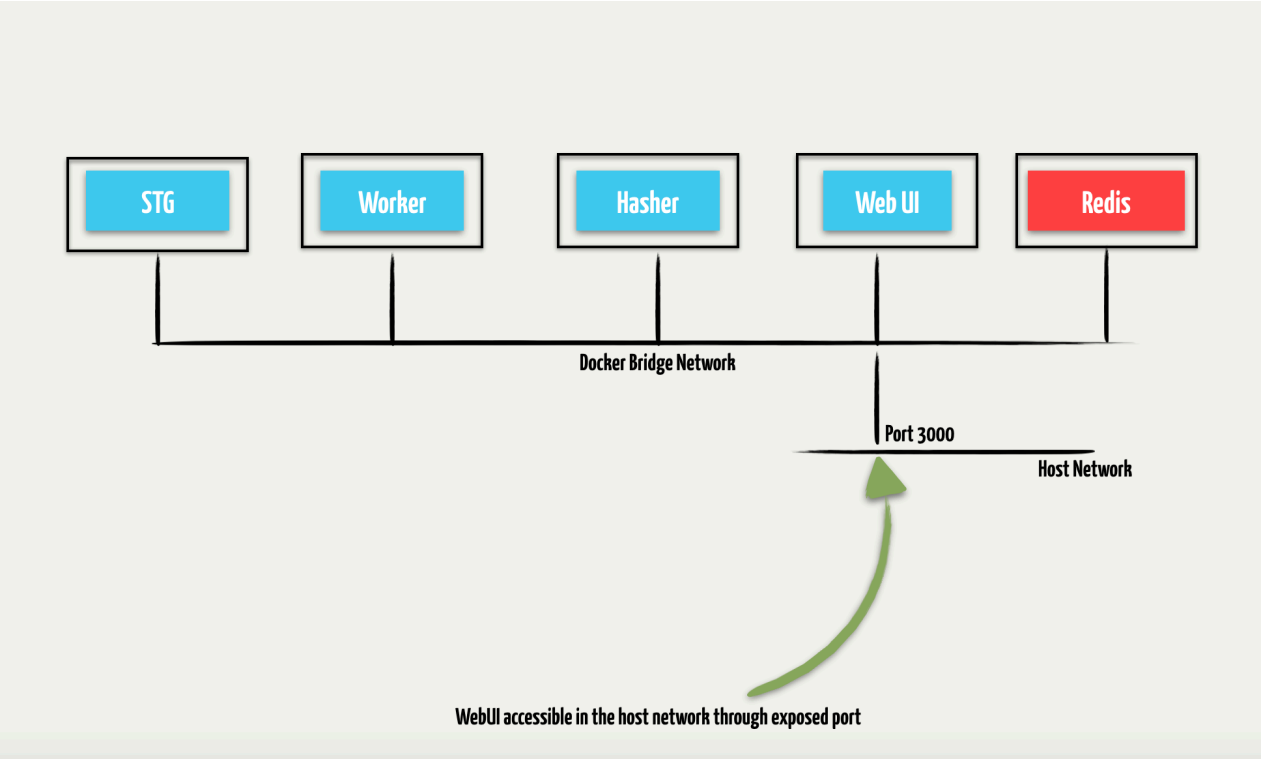
Web UI

The web based user interface should be implemented as a web application and should be developed using Node.js. The application should be able to serve a html page containing a line chart that is updated in real time. Use of Socket.io is recommended to introduce real-time features and Chartist for displaying line charts with javascript.

The application should be designed to subscribe to a redis topic/channel and should update the line chart whenever the worker publishes data in the topic/channel.

Composing the services

Finally compose the entire application comprising of the above services using Docker Compose. Make sure to expose the web ui to be available at host machine port 3000 (http://localhost:3000). The diagram below depicts the network layout of the services.



Repository Structure

The project should be organized in a directory structure identical to the structure shown below. After organizing your work in the structure specified publish the project to your github account.

```
.
├── docker-compose.yml
├── hasher
|   ├── Dockerfile
|   ├── Makefile
|   ├── go.mod
|   ├── go.sum
|   └── main.go
├── README.md
├── stg
|   ├── Dockerfile
|   ├── Makefile
|   ├── go.mod
|   ├── go.sum
|   └── main.go
├── webui
|   ├── Dockerfile
|   ├── index.html
|   ├── index.js
|   ├── package.json
|   └── package-lock.json
└── worker
    ├── Dockerfile
    ├── Makefile
```

```
    ├── go.mod
    ├── go.sum
    └── main.go
```

Learning Resources

- Microservices
- Go Programming Language
- Node.js
- Docker
- Docker Compose
- REST Web Services