FS20

DISTRIBUTED APPLICATION

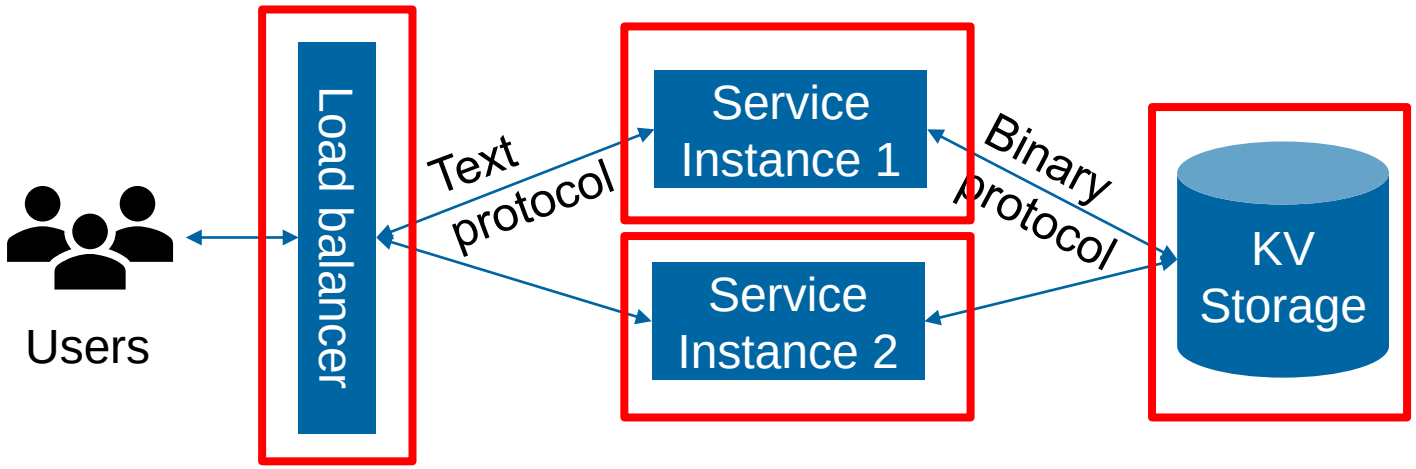
Docker / Docker compose example, Kubernetes

T. Bocek
thomas.bocek@hsr.ch

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

Multi Container

- **Docker compose**
  - Tool for multi-container Docker applications
  - Docker container, lecture 7
    - Most likely your setup has multiple docker containers in your CT
  - Installation
    - Ubuntu 20.04, apt install docker-compose
    - Other OS



- **Example docker containers for (lecture 7)**
  - From lecture 7: Dockerfile

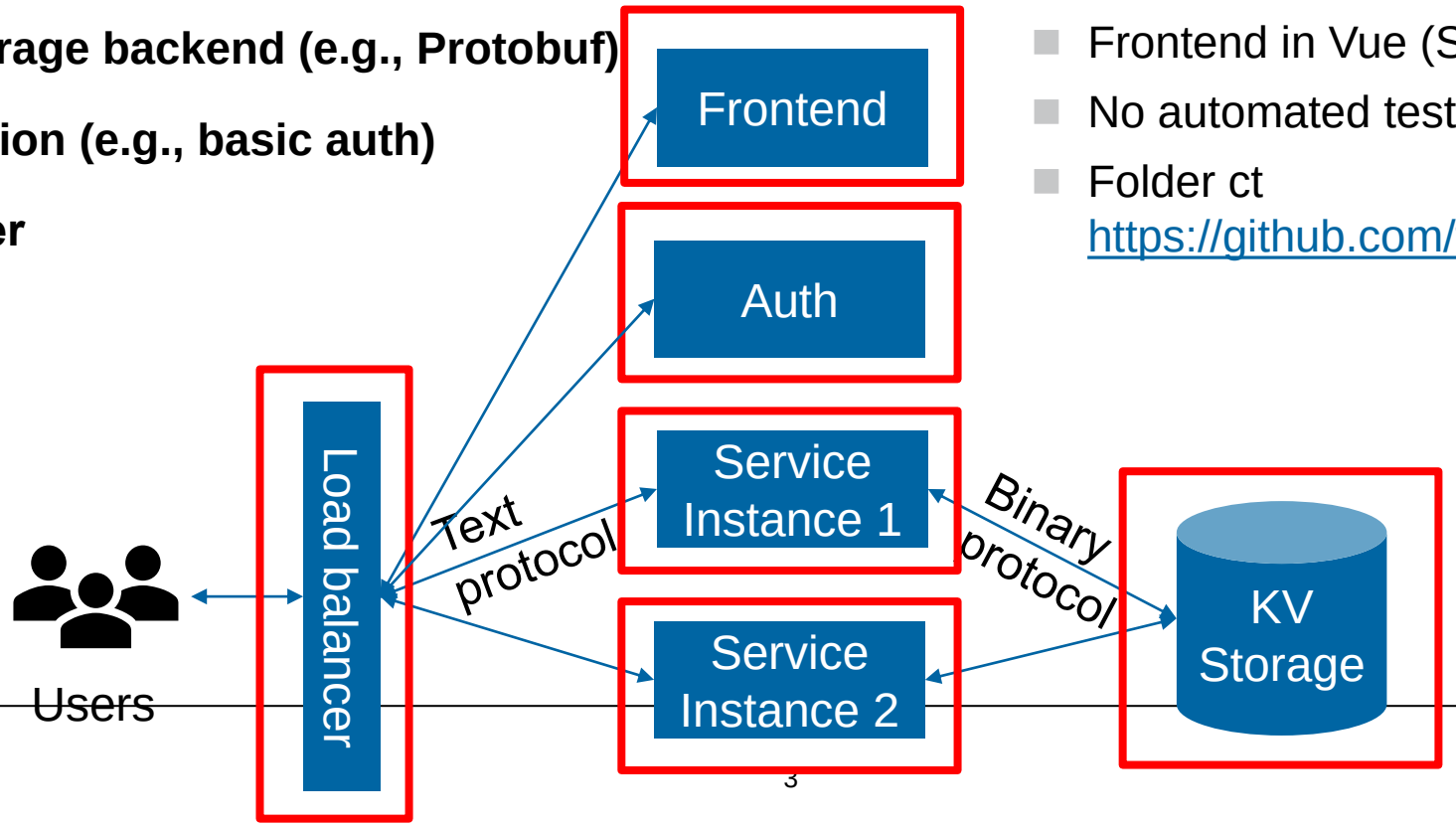
```
#Dockerfile
FROM golang:alpine AS builder
WORKDIR /build
COPY server-stateful.go .
RUN apk --no-cache add git
RUN go get -d -v
RUN go build server-stateful.go

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=builder /build/server-stateful .
CMD ["./server-stateful", "-p", "8081"]
```

Challenge Task

- **Simple Frontend (e.g., HTML, Vue, React)**
- **Loadbalancer(e.g., traefik)**
- **Two instances of a service (your choice) (e.g., REST / HTTP / Protobuf)**
- **One KV storage backend (e.g., Protobuf)**
- **Authentication (e.g., basic auth)**

→ **Coin Tracker**

- **Design decisions**
  - Auth done via separate service
  - Dockerized / docker-compose (no VM)
    - That's why this lecture took so long to record
  - Frontend in Vue (SPA)
  - No automated testing
  - Folder ct https://github.com/tbocek/FS20



KV Storage

- **Design decision**
  - No DB (in-memory KV is enough)
  - No problem to use DB, every DB also can store KV
  - Golang:
    - kvm = make(map[string]string)
- **Binary protocol: grpc**
  - Lecture 3
  - Uses HTTP/2 and protobuf for serialization
  - Protocol definition in proto3
  - Generate code
  - Ugly: I copied code from storage to service
  - Better: have source code or the proto file in a shared repository

- **Dockerfile: as shown in lecture, multi-stage building in first stage, running in second**

```
//run with: protoc -I . storage.proto --go_out=plugins=grpc:.
syntax = "proto3";
package main;

service Storage {
    rpc GetKey(Key) returns (Value) {}
    rpc PutKeyValue(KeyValue) returns (Empty) {}
}

message Key {
    string key = 1;
}

message Value {
    string value = 1;
}

message KeyValue {
    string key = 1;
    string value = 2;
}

message Empty {}
```

Service

- **Design decision**
  - Connect to storage via grpc
  - REST interface /portfolio GET and POST
  - Bad: hardcoded IP address, password
- **Dockerfile: as shown in lecture, multi-stage building in first stage, running in second**
  - Multi-container setup: wait until ready: wait4ports
- **Now this we can test via our REST interface**
  - Fire up both docker containers
  - Use your favorite REST client
- **Auth**
  - Check if user is allowed: JWT, lecture 6

```
#Dockerfile
#to build image: docker build -t service .
#to run: docker run -it --rm -p 8085:8080 service
FROM golang:alpine AS builder
WORKDIR /build
COPY storage.pb.go server-stateless.go ./
RUN apk --no-cache add git
RUN go get -d -v
RUN go build server-stateless.go storage.pb.go

FROM alpine:latest
WORKDIR /root/
RUN apk --no-cache add wait4ports
COPY --from=builder /build/server-stateless .
CMD ["./server-stateless", "-p", "8080", "-s", "8082"]
```

Auth

- **Design decision**
  - Auth not combined with load balancer, behind load balancer
  - JWT tokens can easily be obtained and used scalable with service
  - REST protocol, its accessed via browser
  - Only auth, no refresh token, no roles, etc.
- **Dockerized**
- **Idea**
  - Browser calls auth, provides username, password, gets back token (e.g., in the Header)
  - Browser stores locally this token, uses it to access resources with:
    - 'Authorization': 'Bearer ‘ + token
- **Bad: hardcoded username, password ☺**
- **Hardcoded key, expiry time, id, …**

```
claims := &Claims{
  Username: creds.Username, //this could be something app specific
  StandardClaims: jwt.StandardClaims{
    ExpiresAt: time.Now().Add(time.Hour * 24 * 5).Unix(),
    Id:        "15b54ed3-d1ed-4122-8e10-6c2f462ee398",
    Subject:   creds.Username,
  },
}
token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
tokenString, err := token.SignedString(jwtKey)
```

Frontend

- **Design decision**
  - Vue SPA frontend
  - Webpack bundler
  - Caddy as the HTML server
    - Why caddy? – I never used it before
    - Simple setup
  - With routing (frontend)
    - History mode: if set, reconfigure traefik
- **Dockerized**
- **“only 2 dependencies”**
  - 460 packages

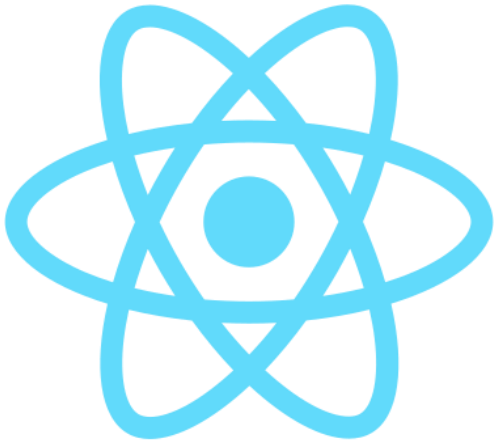
```
{
  "dependencies": {
    "vue": "^2.6.11",
    "vue-router": "^3.1.6"
  },
  "devDependencies": {
    "css-loader": "^3.5.3",
    "vue-loader": "^15.9.2",
    "vue-template-compiler":
"^2.6.11",
    "webpack": "^4.43.0",
    "webpack-cli": "^3.3.11",
    "webpack-dev-server": "^3.10.3"
  }
}
```

```
http://:8080
root * /site/
file_server
```

```
#Dockerfile
#to build image: docker build -t frontend .
#to run: docker run -it --rm -p 8083:8080 frontend
FROM node:alpine AS builder
WORKDIR /app
COPY *.js *.json ./
COPY vue ./vue
RUN yarn install
RUN ./node_modules/.bin/webpack

#Dockerfile
FROM caddy:latest
COPY Caddyfile /etc/caddy/Caddyfile
COPY index.html /site/
COPY --from=builder /app/dist /site/dist/
```

Vue.js Introduction

- Vue (pronounced /vjuː/, like view) is a progressive framework for building user interfaces.
- …On the other hand, Vue is also perfectly capable of powering sophisticated Single-Page Applications when used in combination with modern tooling and supporting libraries.
- 5 Best JavaScript Frameworks in 2017
  - **Angular, React, Vue**, Ember, Meteor
- Top 7 Most Popular JavaScript Frameworks in 2018
  - **Angular, React**, Ember, Aurelia, Meteor, Polymer, **Vue**
- **Vue – low learning curve**



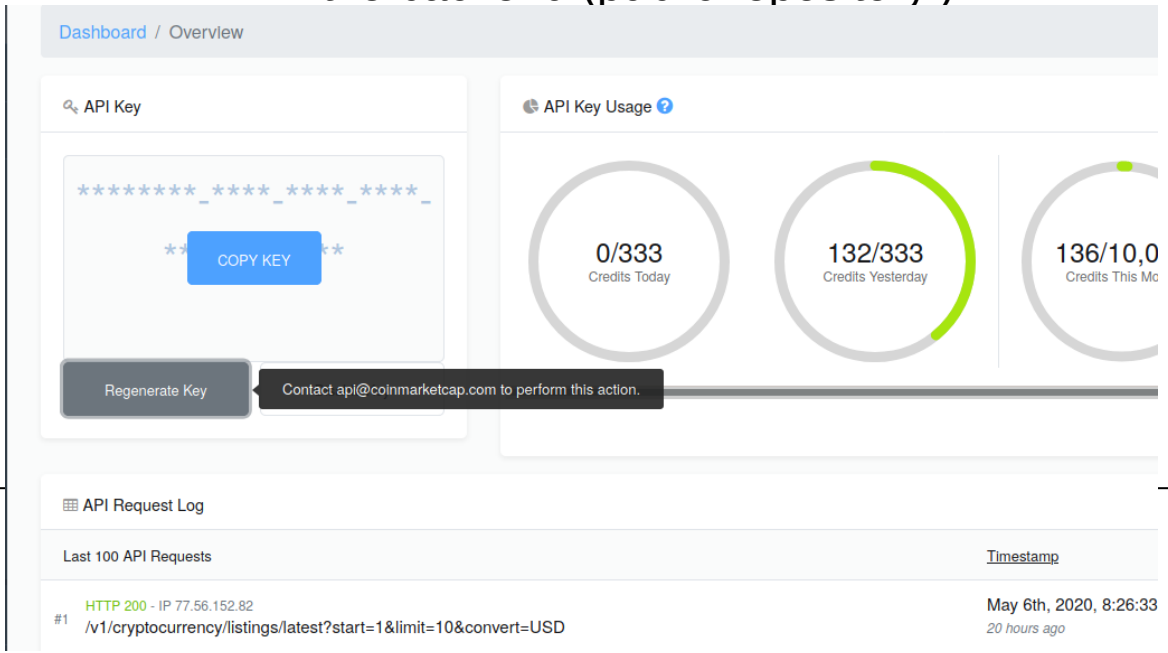
Vue.js Introduction

- **Simple Vue example**
  - Load Vue library
- **Vue built-in template syntax**
  - {{name}}
- **Reactive**
  - The data and the DOM are now linked, and everything is now reactive
  - vm.name = ‘hallo’

```
<!DOCTYPE html>
<head>
    <meta charset="UTF-8">
    <title>Hello!</title>
    <script src="https://cdn.jsdelivr.net/npm/vue"></script>
</head>
<body>
    <div id="app">
        <h2 class="hello-title">Hello {{name}}!</h2>
    </div>
    <script type="text/javascript">
        var vm = new Vue({
            el: '#app',
            data: {
                name: 'Hello Vue!'
            },
            created: function () {
                fetch('/spa/'+window.location.hash.substring(1))
                    .then(response => response.json())
                    .then(body => {
                        this.name = body.value;
                    });
            }
        });
    </script>
</body>
</html>
```

API proxy

- **Fetch coinmarketcap data**
  - Directly in browser vs. backend
- **Design decision**
  - Due to CORS and providing an API key, the call is done by the backend to fetch the data
  - Bad: should be protected via JWT token
  - Super bad: never ever do this: store API key in the backend (public repository!)

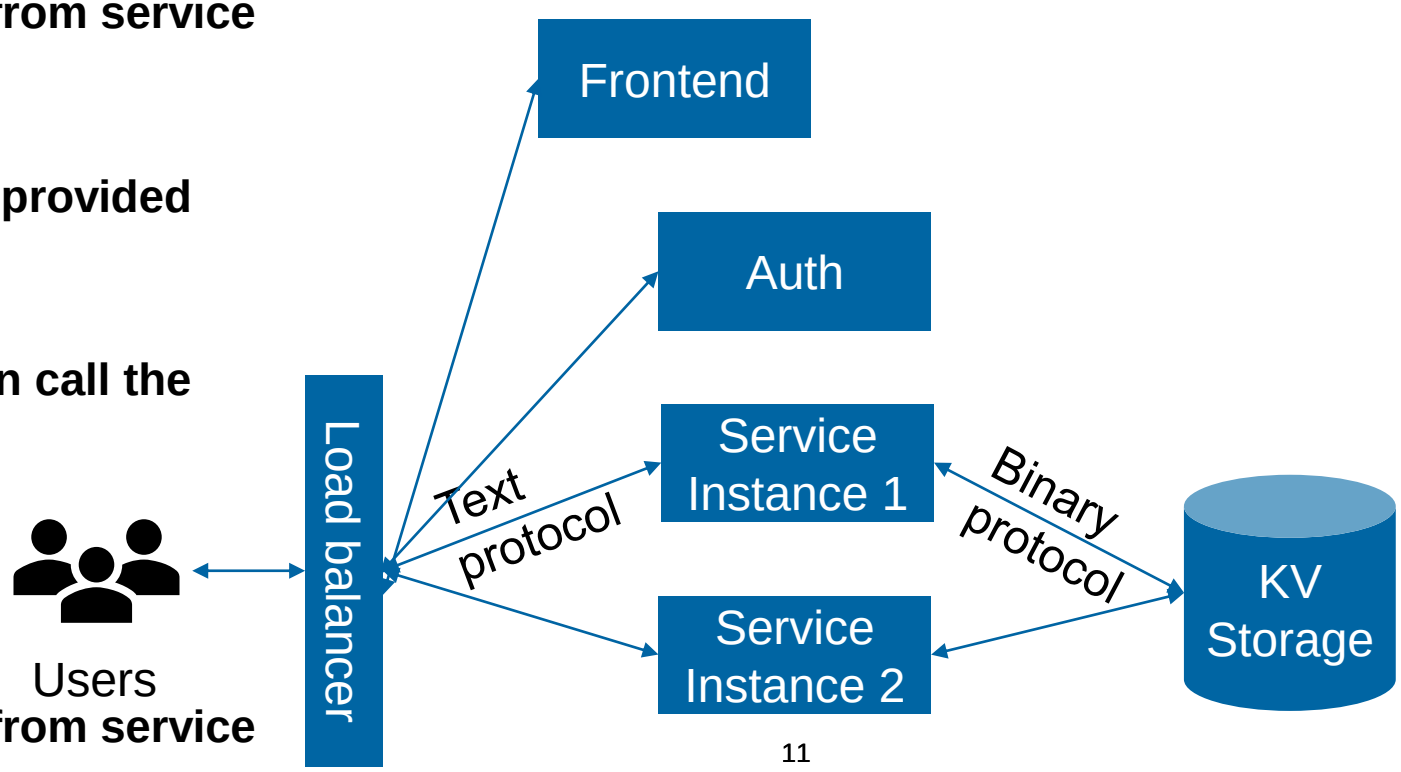


- **Fetch from coinmarketcap.com, serve connected client**

```
req, err := http.NewRequest("GET", URL, nil)
if err != nil {
  w.WriteHeader(400)
}
req.Header.Set("Accept", "application/json")
req.Header.Set("X-CMC_PRO_API_KEY", API_KEY)
client := &http.Client{}
res, err := client.Do(req)
if err != nil {
  w.WriteHeader(400)
}
w.Header().Set("Content-type", "application/json")
io.Copy(w, res.Body)
```

Auth Flow

- **User/browser: access frontend, get HTML/JS**
- **Frontend tells user/browser: go and fetch from service**
- **Loadbalancer decides: service 2**
- **Access to service 2 is denied, as no token provided**
- **User/browser: redirected to /login**
- **Login works without token, and anyone can call the auth backend service**
- **User/browser calls auth, with username, password, gets token, redirected to main page**
- **Frontend tells user/browser: go and fetch from service**
  - … until it reaches service 1 and 2 and this will work, as we have the token



Traefik

- **Loadbalancer**
  - Also in a docker container
  - Docker containers can access docker network:
    - Interface docker0
    - inet 172.17.0.1 netmask 255.255.0.0 broadcast 172.17.255.255
- **Design decision, pull image directly from dockerhub**
  - Configuration via docker-compose.yml
- **Assign external ports in compose file**

```
traefik:
    image: "traefik"
    ports:
      - "8080:80"
    volumes:
      - $PWD/traefik/traefik.toml:/etc/traefik/traefik.toml
      - $PWD/traefik/dynamic_load.toml:/etc/traefik/dynamic_load.toml
```

Connecting everything

- **docker-compose up --build --scale service=2**
- **Open browser: localhost:8080**
- **Development**
  - Run single parts locally, don’t restart everything
- **Deployment (use compose on single host)**
  - Docker Swarm / docker-compose
    - Swarm mode for managing cluster of Docker Engines
    - docker deploy --compose-file docker-compose.yml / docker stack deploy …
  - Kubernetes
    - Kompose – convert docker-compose.yml to Kubernetes
    - Kubernetes (K8s) is an open-source system for automating deployment (k3s)

```
#docker-compose.yml
#run with: docker-compose up --build --scale service=2
version: '3'
services:
  traefik:
    image: "traefik"
    ports:
      - "8080:80"
    volumes:
      - $PWD/traefik/traefik.toml:/etc/traefik/traefik.toml
      - $PWD/traefik/dynamic_load.toml:/etc/traefik/dynamic_load.toml
  apiproxy:
    build: ./apiproxy
    ports:
      - "8081:8080"
  storage:
    build: ./storage
    ports:
      - "8082:8080"
  frontend:
    build: ./frontend
    ports:
      - "8083:8080"
  auth:
     build: ./auth
     ports:
      - "8084:8080"
  service:
    build: ./service
    ports:
      - "8085-8086:8080"
```

Questions?

Dr. Thomas Bocek
thomas.bocek@hsr.ch