Kubernetes Full Stack Application

Marco Jakob

April 19, 2019

Contents

1 Introduction

This Introduction should give an end to end overview to develop a sample Application with multiple microservices and a Databases completely on Kubernetes.Furthermore, the application should run on a local machine as well as on simple Docker containers witout any manual changes. The Application generates random numbers, saves them with a Timestamp on a database, displays the new number on a Frontend and also shows graphs from previous andom numbers based on their producerservices Id. The goal of the whole application is primarly to develop an end to end Application fully cloud native wit all configurations based on requirements for cloud native applications. The second goal is to show graphically the self healing process of Kubernetes, namely when we delete a random generator Pod, it should automatically create a new Random generator Pod with a new Id.

2 Prerequisites

To create the application it is assumed that we already have a machine with the following Software running:

- Java Development Kit (minimum 1.8, JAVA_HOME set)
- Maven build tool (MAVEN_HOME and M2_HOME set)
- Python 3.6
- node package manager
- Docker
- Angular (minimum version 6) and Angular CLI

Not Required but recommendet

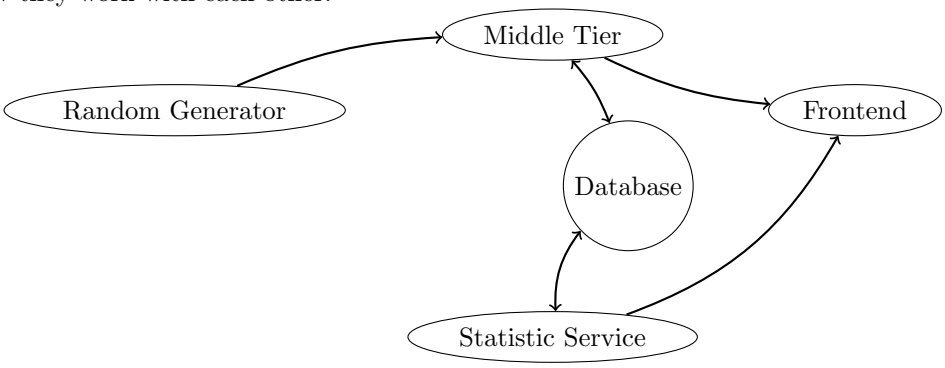
- Visual Studio Code
- Spring tool suite

2.1 Technology

Service	Technology
Random Number Generator circle	Python Flask
Database to store previous Numbers	Mysql Database
Microservice for requesting new Numbers	Spring Boot Java
Microservice for gathering Statistics	Spring Boot Java
Frontend	Angular with Spring Boot
Routing Server	Nginx
Server OS	Ubuntu Server 18.4.2 LTS

2.2 Model

The Following Picture should give an overview about the different Services and how they work with each other.



The random generator service creates a JSON file with an unique id and a random number. The middle tier service calls the random generator, saves the number with the id, saves it in the database and forwards it to the frontend. The statistic service is fetched from the frontend to deliver historic data for the graph and the table with all counts from previous calls.

3 Development

3.1 Database

In the further sections we will use a plain mysql docker container as a database. In order to use the Application completely locally, a local mySQL database has to be installed and a schema for the random generator application needs to be provided for a specific randgenuser. If there is not yet a local mySQL database installed it can be done with the following command:

```
sudo apt-get update
sudo apt-get install mysql-server
```

After the installation log in to MySQL as root.

```
sudo mysql
```

Now the Database and the user can be created and permissions to the respective schema will be given.

```
1 CREATE database rand_numbers;
2 CREATE USER 'random_user'@'localhost' IDENTIFIED BY '
      random_password';
3 GRANT ALL PRIVILEGES ON rand_numbers.* TO 'random_user'@'localhost'
      ;
```

With this, the Database is ready to used by the Random Generator Example from localhost.

3.2 Random Generator

The Random Generator will be a verry simple Python app which has an unique Id per Service instance. It will return his id and a new randomly generated number. For this application we create a folder called generator with the following Structure.

```
generator
├── Dockerfile
├── rand_gen.py
└── requirements.txt
```

The Dockerfile is needed to create the Docker Image which will be used from Kubernetes to create the random generator service. rand_gen.py is the complete Random Generator Python application based on Flask[1].

```
1  from flask import Flask, jsonify
2  import random
3  import uuid
4
5  app = Flask(__name__)
6  user_id = uuid.uuid4().int
7
8  @app.route('/')
9  def random_number():
10     object = {'id' : str(user_id),
11               'randNumber': str(random.randint(0,1000))}
12     return jsonify(object)
13
14 if __name__ == '__main__':
15     app.run(port=5050,host='0.0.0.0')
```

In requirements.txt are the Python packages specified which are needed to be installed. In our case we need flask. Therefore, this file contains only one line which is:

```
Flask==1.0.2
```

[1] More information about Flask can be found on the official Homepage `http://flask.pocoo.org/`

3.3 Middle Tier

The Middle Tier Application is created with Spring Initializer. Unter the following link `https://start.spring.io/` Helps to bootstrap very fast a simple Spring Boot application. In our case the fields should be filled out as followed:

Project	Maven Project
Language	Java
Spring Boot	2.1.3 (All versions would apply)
Group	ch.toky.randgen
Artifact	middletier
Name	MiddleTier
Description	Middle Tier Service for Random Generator Application
Packaging	Jar
Java Version	8
Dependencies	Web, JPA, MySQL, Rest Repositories (Actuator)

Actuator is optional in case you want to monitor your application. The Website will create a zip file to download. This zipfile has to be extracted to the desired Project folder.

```
middletier
├── Dockerfile
├── mvnw
├── mvnw.cmd
├── pom.xml
└── src
```

Spring Initializer already created mostly the whole structure. the Dockerfile has to be created only.

Now inside of the Project /src/main/java create the following Project Structure and files.

```
ch.toky.randgen.middletier
├── MiddletierApplication.java
├── ch.toky.rand-gen.middle-tier.model
│   ├── PodStat.java
│   └── RandomNumber.java
└── ch.toky.rand-gen.middle-tier.repository
    └── PodStatRepository.java
```

The file MiddleTierApplication.java should already be in place. as Spring Boot Initializr created this with the complete Folder structure. the packages controller, model and repository need to be created. Inside of controller, A file named MiddletierController.java needs to be created with the following content.

Within the model package two new Files have to be created: PodStat.java and RandomNumber.java.

PodStat is an Entity and therefore the class has to be annotated with @Entity. It contents the following fields with their respective getter and Setter Methods.

```
    @Id
    @GeneratedValue(strategy= GenerationType.AUTO)
    private Long podStatID;
    private String id;
    private Long timeStamp;
    private Long counter;
```

The class RandomNumber is only a DTO which is retreived from the random generator Service. Therefore there is no need for a annotation. Only the following Fields needs to be declared with their getters and setters:

```
    private String id;
    private Long randNumber;
```

In the repository package the file PodStatRepository.java will be created. As this java file is not a class but an interface it needs to be changed to interface and extends JpaRepository¡PodStat, Long¿ and it will be annotated with @Repository Now those two methods can be created

```
    @Query("Select count(ps.id) from PodStat ps where ps.id = ?1")
    Long countUniqueId( String id);

    @Query("Select count(ps.id) from PodStat ps")
    Long findMaxCount();
```

Witin the controller all the endpoints are declared. Therfore it will be annotaded with @RestController

Those fields are needed within the controller and are therefore declared first.

```
    @Autowired
    private ObjectMapper objectMapper;
    @Autowired
    private PodStatRepository podStatRepository;
    private Map<String, String> env = System.getenv();
```

As there should not be any hardcoded url according to the 12 Factor application [2], the Url will be retreived through a System Variable.

Now we create a controller method with the business logic inside:

```
1   @RequestMapping(value = "/", produces = "application/json")
2   public RandomNumber getRandom() {
3
4       RandomNumber randNum = getNewRandomNumber();
5       Long maxCountId = podStatRepository.countUniqueId(randNum.
    getId());
6       Long maxCountOverall = podStatRepository.findMaxCount();
7       PodStat tmpPod = new PodStat();
8       tmpPod.setId(randNum.getId());
9       tmpPod.setTimeStamp(maxCountOverall+1);
10      tmpPod.setCounter(maxCountId +1);
11      podStatRepository.save(tmpPod);
12
13      return randNum;
14
15  }
```

The above Method listens on the path / and returns the Random number after saving it to the database with the actual count.

Now the last Method is used to retreive the Random Number from the Random Generator Service.

```
1   private RandomNumber getNewRandomNumber() {
2
3       String randGenUrl=env.get("RANDOM_GENERATOR_URL");
4       String URL = randGenUrl;
5       RestTemplate restTemplate = new RestTemplate();
6
7       return   restTemplate.getForObject(URL, RandomNumber.class)
    ;
8
9   }
```

Above Source code contains the Controller and the Service, which should be separated optimally. Next two files represents POJOs which are going to be used from controller and provided by Repositories.

The last thing which needs to be done for the Middle Tier Service is to setup the default values for the Database Connection.

The datasource configurations are going to be overwritten by Environment Variables once they run inside of a Docker Container.

[2] Accessible at `https://12factor.net/`

3.4 Statistic Service

The Statistic Service has the same setup as the Middle Tier client. Therefore the Project will be created too with Spring Boot initializer.

Project	Maven Project
Language	Java
Spring Boot	2.1.3 (All versions would apply)
Group	ch.toky.randgen
Artifact	stattier
Name	Stattier
Description	Statistic Tier Client for Random Generator Application
Packaging	Jar
Java Version	8
Dependencies	Web, JPA, MySQL, Rest Repositories

For the statistic Tier Service we need to create the following Package structure:

```
ch.toky.randgen.statservice
├── StatserviceApplication.java
├── ch.toky.randgen.statservice.controller
│   └── Controller.java
├── ch.toky.randgen.statservice.model
│   ├── PodStat.java
│   ├── Series.java
│   ├── SeriesItem.java
│   └── Stat.java
├── ch.toky.randgen.statservice.repository
│   └── PodStatRepository.java
└── ch.toky.randgen.statservice.service
    └── StatService.java
```

This time, we will create a seperate controller class. Therefore, inside of StatserviceApplication.java will only be the main method which calls the Spring Boot applicaton. Which means we do not have to change this file as Spring Boot already did everything needed for us.

Controller.java will be annotated with @RestController as it contains all our endpoints.

next, we need the StatService class in this controller, this is why we inject it with @Autowired.

```
1     @Autowired
2     StatService statService;
```

Now we can also create our two endpoints, one for all stats, and one with the history.

```
    @RequestMapping(method = RequestMethod.GET, value = "/")
    public List<Stat> getAllPodStats(){

        return statService.getAllPodStats();
    }

    @RequestMapping(method = RequestMethod.GET, value = "/history")
    public List<Series> getAllPodStatHistory(){

        return statService.getPodHistory();

    }
```

We could leave the method out as GET is anyway the default method but we leave it here for for understanding.

Next we are going to declare the Entities and DTOs.

For simplicity i will only show all fields but of cource also the getter and setter methods needs to be implemented.

Fields for PodStat:

```
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long podStatID;
    private String id;
    private Long timeStamp;
    private Long counter;
```

Series will be a DTO and therefore not annotated with Entity we have also a contructor in order to initialize a new Array List.

```
    private String name;
    private List<SeriesItem> series;

    public Series(String name){
        this.name = name;
        this.series = new ArrayList<SeriesItem>();
    }

    public void appendSeriesItem(SeriesItem seriesItem){
        this.series.add(seriesItem);
    }
```

SeriesItem is the second DTO class we are going to use and therefore as well not annotated with @Entity

```
    private Long name;
    private Long value;

    public SeriesItem(Long name, Long value){
        super();
        this.name = name;
        this.value = value;
    }
```

Stat Class:

```
    private String id;
    private Long counter;

    public Stat(String id, Long counter) {
        this.id = id;
        this.counter = counter;
    }
```

Now we can implement the Repository (annotated with @Repository) as an interface which extends JpaRepository¡PodStat, String¿

```
    @Query("SELECT count(pd.id) FROM PodStat pd WHERE pd.id = ?1")
    Long countUniqueId(String id);

    @Query("SELECT distinct ps.id FROM PodStat ps")
    List<String> findUniqueIds();

    @Query("SELECT ps from PodStat ps WHERE ps.id = ?1")
    List<PodStat> findByIds(String id);
```

Now the last file is the most important one as there we will have all the business logig in it.

We need to annotate it with @Service in order to make it available for Spring Boot to create a Bean out of it.

```
    @Autowired
    PodStatRepository podStatRepository;

    private Iterable<String> getAllIds(){
        Iterable<String> source = podStatRepository.findUniqueIds()
    ;

        return source;
    }

    public List<Stat> getAllPodStats(){
        Iterable<String> source = getAllIds();
        List<Stat> podStatistics = new ArrayList<Stat>();

        source.forEach( id -> {
            podStatistics.add(new Stat(id, podStatRepository.
    countUniqueId(id)));

        });

        return podStatistics;
    }

    public List<Series> getPodHistory(){
        List<PodStat> source = podStatRepository.findAll();

        Map<String, Series> podHistorys = new HashMap<>();
        source.forEach( entry ->{
            if(podHistorys.get(entry.getId()) == null){
                podHistorys.put(entry.getId(), new Series(entry.
    getId()));
            }
            podHistorys.get(entry.getId()).appendSeriesItem(new
    SeriesItem(entry.getTimestamp(), entry.getCounter()));

        });

        List<Series> plainSeries = new ArrayList<>(podHistorys.
    values());

        return plainSeries;

    }
```

Now that all the java fils are created we can also create the Dockerfile for this Service.

3.5 Frontend

The frontend Application is build with Spring Boot and Angular together.

Creating Spring Boot Application:

Project	Maven Project
Language	Java
Spring Boot	2.1.3 (All versions would apply)
Group	ch.toky.randgen
Artifact	frontend
Name	Frontend
Description	Frontend Service
Packaging	Jar
Java Version	8
Dependencies	Web, Rest Repositories

Until now we only have the controller part of our User interface with the Spring boot application, we need to create the Angular partnow.

Therefore, we create a new Project directly with Angular CLI within the folder where all the other Projects are stored.

```
ng new ui-frontend
```

Now to try if the project has been set up correctly. Change path to the new project folder and run `ng serve`. Once the server is ready we should be able to access http://http://localhost:4200 and see a sample home page created by angular cli.

As the frontend should be only one single application we will now move all folders into the Spring boot application folder.

in the Folder where all Projects are stored we execute those commands.

```
mv ui-frontend/src/* frontend/src/
rm -rf ui-frontend/src/
mv ui-frontend/* frontend/
rm -rf ui-frontend
```

In order to serve all the frontend files from the Spring Boot application we need to tell angular to save all compiled files into the target/classes/static folder. In order to achieve this, we need to change the output path option in the file angular.json

```
1           "options": {
2             "outputPath": "target/classes/static",
3             "index": "src/index.html",
```

Now every time the Angular Project is built with `ng build` it will be compiled and copied directly to the path where spring boot serves static files. Every time the spring boot application has started we will get the static web files under the root path /.

Now we are going to modify the Angular Project itself.

First we create all interfaces with the following commands.

```
ng generate interface models/randomNumber
ng generate interface models/seriesItem
ng generate interface models/series
ng generate interface models/podStatistic
```

Within Interface RandomNumber, declare those two fields.

```
    id: string;
    randNumber: number;
```

Within Interface PodStatistic

```
    id: string;
    counter: number;
```

Interface SeriesItem

```
    name: number;
    value: number;
```

Interface Series

```
    name: string;
    series: SeriesItem;
```

In order to have some empty Objects we also need to declare the following Classes within the folder implementation.

```
ng generate class RandNumberImpl
ng generate class PodStatisticImpl
```

Where RandomNumberImpl implements RandomNumber and PodStatisticImpl implements PodStatistic Within those classes we define all Strings as empty Strings and all numbers as 0. This prevents us from getting an error from the Html due to an undefined object.

```
ng generate service statistic
ng generate service fetch-number
```

this created 4 new files for us two typescript and two spec.ts files which would be needed for the end to end tests. Both Services also need to be declared in the provders array from *app.module.ts* if not yet done from angular cli.

We will now modify first the fetch-number service
withing the Constructor argument field we pass a HttpClient.

```
  constructor(private _http: HttpClient) { }
```

Therefore you need to make sure to implement the following Line.

```
import { HttpClient } from '@angular/common/http';
```

```
getRandomNumber(): Observable<RandomNumber>{
  return this._http.get<RandomNumber>('randomNumber');
}
```

In the Statistic Service File we declare the same HttpClient to be passed to the constructor.

```
constructor(private _http: HttpClient) { }
```

Then the following methods are going to be declared:

```
getStatistics(): Observable<PodStatistic[]>{

  return this._http.get<PodStatistic[]>('statistics');
}

getHistory(): Observable<Series[]>{

  return this._http.get<Series[]>('history');
}
```

As we are using Material design, we need to create a seperate Module to manage all depedencies which are used for Material design.

```
ng generate module material
```

Within the newly created *material.module.ts* file we add the following Modules into imports and exports array.

```
  CommonModule,
  MatButtonModule,
  MatCheckboxModule,
  MatCardModule,
  MatTableModule
```

In order to be able to use our newly created Material Module within our app, we need to add it to the import array from *app.module.ts* along with all the following imports.

```
  BrowserModule,
  MaterialModule,
  HttpClientModule,
  NgxChartsModule,
  BrowserAnimationsModule
```

By now we cannot import all the declared Module Dependencies. this is because we use a lot of external libraries which we now have to install via npm.
in order to use Angular Material [3]

```
npm install --save @angular/material \\
@angular/cdk @angular/animations
```

[3] documentation via following link `https://material.angular.io/guide/getting-started`

Ngx-Charts Module is to display the statistics as a graph. [4]
It can be installed with the following command.

```
npm install @swimlane/ngx-charts --save
```

Now we can start adding the business logic. Within the Main Component file *app.component.ts*. First we declatre the Variables which we are going to use.

```
randomNumber: RandomNumber = new RandNumberImpl();
statistics: PodStatistic[] = [new PodStatisticImpl()];
dataSource: any;
multi: Series[];
displayedColumns: string [] = ['id', 'counter'];
```

In the constructor, we provide our two services as Private.
next we create the Methods which fetches the data from the service.

```
getNewNumber(){
  this.fetchNumberService.getRandomNumber()
  .subscribe(
    data =>{this.randomNumber = data;});
};

getStatistics(){
  this.statisticService.getStatistics()
  .subscribe(
    data =>{
      this.dataSource = new MatTableDataSource<PodStatistic>(
data);});
}

getHistory(){
  this.statisticService.getHistory()
  .subscribe(
    data => {this.multi = data;})
};
```

We also need to define a method Called *ButtonClick* as we want to use it from the Html file. within this Method we will do nothing more that just calling all fetch methods.

```
buttonClick(){
  this.getNewNumber();
  this.getStatistics();
  this.getHistory();
}
```

From *ngOnInit* we call only *buttonClick*().
then we also have a onSelect which we need ot provice for the graph;

```
onSelect(event) {
  console.log(event);
}
```

[4]Documentation can be found here `https://swimlane.gitbook.io/ngx-charts/`

As a last thing in this file we need to add some configuration Variables.

```
  // Graph
  view: any[] = [700, 400];
  showXAxis = true;
  showYAxis = true;
  gradient = false;
  showLegend = true;
  showXAxisLabel = true;
  xAxisLabel = 'Sum of all Calls';
  showYAxisLabel = true;
  yAxisLabel = 'Calls per Pod';
  timeline = true;
  autoScale = true;
  colorScheme = {
    domain: ['#5AA454', '#A10A28', '#C7B42C', '#AAAAAA']
  };
```

the Last file we need to create for the Frontend is the html from our component Class.

```
<section class="mat-typography">
  <h1>Random Number</h1>

  <mat-card>
    <mat-card-header>
      <p>Id: {{randomNumber.id}}</p>
    </mat-card-header>
    <mat-card-content>
      <p>Number: {{randomNumber.randNumber}}</p>
    </mat-card-content>
  </mat-card>
  <button mat-button (click)="buttonClick()">Get New Number</button
    >

  <mat-card>
    <mat-card-header>
      <p>Statistics Table</p>
    </mat-card-header>
    <mat-card-content>
      <table mat-table [dataSource]="dataSource">
        <ng-container matColumnDef="id">
          <th mat-header-cell *matHeaderCellDef>ID</th>
          <td mat-cell *matCellDef="let element">{{element.id}}</td
    >
        </ng-container>
        <ng-container matColumnDef="counter">
          <th mat-header-cell *matHeaderCellDef>Counter</th>
          <td mat-cell *matCellDef="let element">{{element.counter
    }}</td>
        </ng-container>
        <tr mat-header-row *matHeaderRowDef="displayedColumns"></tr
    >
        <tr mat-row *matRowDef="let row; columns: displayedColumns;
    "></tr>
      </table>
    </mat-card-content>
  </mat-card>
```

```
33
34   <ngx-charts-line-chart
35    [view]="view"
36    [scheme]="colorScheme"
37    [results]="multi"
38    [gradient]="gradient"
39    [xAxis]="showXAxis"
40    [yAxis]="showYAxis"
41    [legend]="showLegend"
42    [showXAxisLabel]="showXAxisLabel"
43    [showYAxisLabel]="showYAxisLabel"
44    [xAxisLabel]="xAxisLabel"
45    [yAxisLabel]="yAxisLabel"
46    [autoScale]="autoScale"
47    [timeline]="timeline"
48    (select)="onSelect($event)"></ngx-charts-line-chart>
49
50 </section>
```

Now it is time to go through the Server side Part of the Frontend, which is from our Spring Boot frontend. Like we did it for all the previous Spring boot Applications, we will first create all folders and .java files. From within src/main/java

```
ch.toky.randgen.frontend
├── FrontendApplication.java
├── ch.toky.randgen.frontend.model
│   ├── PodStat.java
│   ├── RandomNumber.java
│   ├── Series.java
│   └── SeriesItem.java
└── ch.toky.randgen.frontend.service
    └── RestService.java
```

All classes under the Model Package, represent our DTOs like we had it for all the previous services.

Fields from PodStat:

```
1     private Long podStatID;
2     private String id;
3     private Long timeStamp;
4     private Long counter;
```

Fields from RandomNumber:

```
1     private String id;
2     private Long randNumber;
```

Series:

```
1     private String name;
2     private List<SeriesItem> series;
```

SeriesItem:

```
1     private Long name;
2     private Long value;
```

All fields from above classes must have as well their coresponsive getter and setter methods.

To fetch all the Data we need to declare witin *RestService.java* all the methods which communicates to Statistic Service and to Middle Tier Service.

As this service should be injected within Main Application we need to annotate this Class with @Service.

Fields from RestService:

```
1     private Map<String, String> env = System.getenv();
2     private final String statURL = env.get("STAT_URL");
3     private final String randNumURL = env.get("RAND_NUM_URL");
4     private final String historyURL = env.get("HISTORY_URL");
5     private RestTemplate restTemplate = new RestTemplate();
```

As one can see, we will retreive all endpoints via environment Variables which are either declared directly within the docker Container or via Kubernetes Environment Variable configurations.

Now we declare the methods.

```
1     public List<PodStat> getStatistics(){
2         PodStat[] tmpStats = restTemplate.getForObject(statURL,
      PodStat[].class);
3         return Arrays.asList(tmpStats);
4     }
5
6     public RandomNumber getRandomNumber(){
7         return restTemplate.getForObject(randNumURL, RandomNumber.
      class);
8     }
9
10    public List<Series> getHistory(){
11        Series[] series = restTemplate.getForObject(historyURL,
      Series[].class);
12        return  Arrays.asList(series);
13    }
```

This was the last method for *RestService.java*. Within the Main Application file FrontendApplication.java we will also declare our endpoints we provide to the Angular application. Therefore we need to add the @RestController annotation besides the @SpringBootApplication which should already be in place.

There is one dependency with @Autowired and this is our previously declared RestService.

```
@Autowired
private RestService restService;
```

Now declare all endpoint methods.

```
@GetMapping("/statistics")
public List<PodStat> getStatistics(){
 return restService.getStatistics();
}

@GetMapping("/randomNumber")
public RandomNumber getRandomNumber(){
 return restService.getRandomNumber();
}

@GetMapping("/history")
public List<Series> getHistory(){
```

Within the root directory of the project we can place the Docker file like we did it for all the other services. Except that we now have to add another stage for the angular build.

4 Docker

4.1 Creating the Docker Images

As we were always using multi stage builds we are now able to build the Project directly with Docker without any manual execution of some maven build commands or angular build commands. Mainly for our Frontend it makes or live easier, as we do not need to run ng build and after that mvn clean package. With this we can also always be sure to have the latest version of the jar file, as we are building the jar together with the Docker Image.

There is one huge disadvantage. As every docker Container has its own environment and we cannot mount a hostpath during the build, we need to download all the dependencies for each build which takes quite some time.

This takes a lot of time. There are plugins which makes this much easier. For example the spotify docker Maven plugin which would enable us to build the Dockerfile directly with maven [5]. There is also a maven plugin to execute the angular build before building the jar file [6]. All of them can make our life easier but as we want to dive a little bit deeper into Docker, we will do it with plain Dockerfiles.

Following we describe the Dockerfiles for each service and then as well how to build the Dockerimage. **Database** the Dockerfile can be stored in a seperate empty folder as we do not need to *ADD* or *COPY* any source files.

```
1  FROM mysql:5.6
2
3  # set default values for Database name, user and passwords
4
5  ENV MYSQL_ROOT_PASSWORD=password
6
7  ENV MYSQL_USER=random_user
8
9  ENV MYSQL_PASSWORD=random_password
10
11 ENV MYSQL_DATABASE=rand_numbers
```

creating the docker image

```
docker build -t rand-database-image .
```

[5] https://github.com/spotify/docker-maven-plugin
[6] https://github.com/eirslett/frontend-maven-plugin

Generator

Wihin the root directory of the generator application.

```
FROM python:3.6-alpine

COPY requirements.txt /app/requirements.txt

COPY rand_gen.py /app/rand_gen.py

WORKDIR /app

RUN pip install -r requirements.txt

ENTRYPOINT ["python"]

CMD ["rand_gen.py"]
```

The flag -t inicates that the name of the image and the dot at the end tells docker to use the Dockerfile from the current working directory.

```
docker build -t rand-gen-image .
```

Middle Tier Docker Image

```
FROM maven:3.5-jdk-8 AS builder

COPY ./pom.xml /app/pom.xml

COPY ./src /app/src

RUN mvn -f /app/pom.xml  package -DskipTests

FROM anapsix/alpine-java:latest

COPY --from=builder /app/target/middletier-0.0.1-SNAPSHOT.jar /opt/
    middle-tier.jar

# limit memory space

ENV JAVA_OPTS='-Xmx200m'

ENV SPRING_DATASOURCE_URL=jdbc:mysql://rand-gen-database:3306/
    rand_numbers

ENV SPRING_DATASOURCE_USERNAME=random_user

ENV SPRING_DATASOURCE_PASSWORD=random_password

ENV RANDOM_GENERATOR_URL=http://rand-gen-tier:5050/

ENTRYPOINT ["java"]

CMD ["-jar", "/opt/middle-tier.jar"]
```

We are using anapsix [7] base image, as we want to keep the Docker image as small as possible. This will also be the case for all the following Java Dockerfiles.

[7] `https://hub.docker.com/r/anapsix/alpine-java/`

As you can also see, we are limiting the memory space for all Java applications. This is because the JVM would still see the whole Memory Space on the node and not one its own container. As a result of this it could claim to much memory space for one single service and this is what we want to avoid.

Finally we can create the docker image with the following Command.

Important is, we need to be in the root directory of the Projecct where also the Dockerfile is stored.

```
docker build -t rand-middle-image .
```

As previously mentioned, we will use multi stage builds. In this case we use a maven [8] base image as a builder.

[8] `https://hub.docker.com/_/maven/`

Statistic Service Docker Image For the image itself we are using anapsix base image as this is currently the smallest possible java base image `https://hub.docker.com/r/anapsix/alpine-java/dockerfile/` If you want to run the project on a raspberry, you would need to change the base image to hypriot as a Raspberry has a diffret underlaying CPU architecture which is based on ARM [9]

```
FROM maven:3.5-jdk-8 AS builder

COPY ./pom.xml /app/pom.xml

COPY ./src /app/src

RUN mvn -f /app/pom.xml  package -DskipTests

FROM anapsix/alpine-java:latest

COPY --from=builder /app/target/statservice-0.0.1-SNAPSHOT.jar /opt
    /stat-tier.jar

ENV JAVA_OPTS='-Xmx200m'

ENV SPRING_DATASOURCE_URL=jdbc:mysql://rand-gen-database/
    rand_numbers

ENV SPRING_DATASOURCE_USERNAME=random_user

ENV SPRING_DATASOURCE_PASSWORD=random_password

ENTRYPOINT ["java"]

CMD ["-jar", "/opt/stat-tier.jar"]
```

Basically the Dockerfile for this Service looks pretty much the same as the one for the Middle tier except that we do not need to declare an environment variable for the random generator.

```
docker build -t rand-stat-image .
```

[9] https://github.com/hypriot/rpi-java

Frontend Service Docker Image

As we have one more step within the Frontend service. We need to have one more stage. We need node package manager to build the angular application we are goint to use a node.js docker base image [10].

```
FROM node:10-alpine as frontbuilder

WORKDIR /app

COPY . .

RUN npm install

RUN npm run build

FROM maven:3.5-jdk-8 AS builder

COPY --from=frontbuilder app/ /app/

RUN mvn -f /app/pom.xml  package -DskipTests

FROM anapsix/alpine-java:latest

COPY --from=builder /app/target/frontend-0.0.1-SNAPSHOT.jar /opt/
    front-tier.jar

ENV JAVA_OPTS='-Xmx200m'

ENV STAT_URL=http://rand-stat-tier:8080/

ENV RAND_NUM_URL=http://rand-middle-tier:7070/

ENV HISTORY_URL=http://rand-stat-tier:8080/history

ENTRYPOINT ["java"]

CMD ["-jar", "/opt/front-tier.jar"]
```

```
docker build -t rand-front-image .
```

[10] Node Repository Docker hub `https://hub.docker.com/_/node/`

4.2 running with Docker

If you wanted to skip the whole development part and want to start directly with running the docker files, you can use the prebuilt docker images on docker Hub [11]. Therefore, just put toky03/ in front of every image name.

Create a Network

```
docker network create  -d bridge rand-network
```

Start the Database Container

```
docker run -d --name rand-database \
--network rand-network \
rand-database-image
```

Start the Random Generator App

```
docker run -d --name=rand-gen-tier \
--network rand-network \
rand-gen-image
```

Start the middle tier Application

```
docker run -d --name rand-middle-tier \
--network rand-network \
rand-middle-image
```

Starting the Stat Tier Application

```
docker run -d \
--name rand-stat-tier \
--network rand-network \
rand-stat-image
```

Start the Frontend Service

```
docker run -d -p 8080:8080 \
--network rand-network \
--name rand-frontend \
rand-front-image
```

All the default environment Variables are already set within the Dockerfiles therefore we can startup all the services one by one.

The only setup you need to check is if port 8080 on localhost is already allocatd. If this is the case, you need to bind the frontend container to another port than 8080 for example with $-p8081 : 8080$.

[11] `https://hub.docker.com/u/toky03`

5 Kubernetes

5.1 Installing Kubernetes

To play arround and get to know Kubernetes it is recommendet to install minikube [12] which is running inside a VM on your local machine. If you want to use Kubernetes on a Server and deploy PROD ready application then you have the option to rent directly a Kubernetes Cluster from a Vendor

- AWS EKS `https://aws.amazon.com/de/eks/`
- GKE Google Cloud `https://cloud.google.com/kubernetes-engine/`
- IBM Kubernetes Service `https://www.ibm.com/cloud/container-service`
- etc ...

You can also install Kubernetes on a plain server, we use ubuntu for this setup. Under the following link you can find the source for the following installation instruction[13]

Everything with a # in front of the command means this is executed as root and everything with $ means it is executed as a regular user.

But first we need to add the Kubernetes to our installation repository.

```
# curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg \
| apt-key add

# apt-add-repository \
"deb http://apt.kubernetes.io/ kubernetes-xenial main"
# apt install kubeadm

# swapoff -a
```

Now we have Kubeadm installed, we can continue with the server setup based on the instruction from Kubernetes itself [14]

The pod network will be Callico.

```
# kubeadm init --apiserver-advertise-address=<ip-address> \
--pod-network-cidr=192.168.0.0/16
```

Switch back to non root user and execute those commands.

```
$ mkdir -p $HOME/.kube
$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Now we just need to deploy the pod network.

[12] `https://kubernetes.io/docs/tasks/tools/install-minikube/`
[13] `https://linuxconfig.org/how-to-install-kubernetes-on-ubuntu-18-04-bionic-beaver-linux`
[14] `https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/`

```
$ kubectl apply -f \
https://docs.projectcalico.org/v3.3/getting-started/
kubernetes/installation/hosted/rbac-kdd.yaml
$ kubectl apply -f \
https://docs.projectcalico.org/v3.3/getting-started/
kubernetes/installation/hosted/
kubernetes-datastore/calico-networking/1.7/calico.yaml
```

Wait until all pods are up and running. `kubectl get pods --all-namespaces`

If we are using a single node cluster, we need to tell Kubernetes that it is allowed to deploy pods on the master node (which is the only node in the cluster)

```
$ kubectl taint nodes --all node-role.kubernetes.io/master-
```

Now we are done with the setup and we can start deploying pods to the Kubernetes cluster.

5.2 Deploy Services

Kubernetes Commands

Kubernetes is managing all entities according to specifications. Which means we need to declarare our desired state and kubernetes will do all the magic for us. Furthermore, it will also constantly check if the actual state is matching with the desired state and adjust if neccessary. `kubectl` is used to configure kubernetes [15]. In our case we declare all our desired states within yaml files and post them via `kubectl create -f <filename.yml>` to kubernetes. the create command means that the declared ressources in the yaml file *Deployment*, *Service*, *Secret*, $ConfigMap$ needs to be created. the $-f$ flag means it should read it from the following file.

after that we can user `kubectl get pods` to show the all pods within the default namespace and `kubectl get svc` to show all services.

Configuration

Secrets Secrets[16] are a special construct within Kubernetes to save credentials like username and passwords. For our application, we create for each item a seperate file with the following commands.

```
echo -n 'rnduser' > ./username.txt
echo -n 'rndpwd' > ./password.txt
echo -n 'rootrndpwd' > ./rootpassword.txt
```

Then create the secret with the following command:

```
kubectl create secret generic rand-secret-file \
--from-file=./username.txt \
--from-file=./password.txt \
--from-file=./rootpassword.txt
```

This created the secret with the name $rand-secret-file$ for us.

Another very helpful item within Kubernetes are $ConfigMaps$ It is more or less the same as a secret but not with hidden items. We use a $ConfigMap$ to manage our Environment Variables for our Application.

Environment Variable configuration for the whole application.

```
1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: random-config-env
5    creationTimestamp: 2019-04-13T13:31:07+00:00
6    namespace: default
7  data:
8    SPRING_DATASOURCE_URL: jdbc:mysql://database-svc:3306/rand_nums
9    STAT_URL: http://random-statistic-svc:8080/
10   RAND_NUM_URL: http://middle-tier-svc:7070/
11   HISTORY_URL: http://random-statistic-svc:8080/history/
12   RANDOM_GENERATOR_URL: http://random-generator-svc:5050/
13   MYSQL_DATABASE: rand_nums
```

[15] https://kubernetes.io/docs/reference/kubectl/overview/

[16] https://kubernetes.io/docs/tasks/inject-data-application/distribute-credentials-secure/

All the Applications are running inside of a docker container and are managed within a Kubernetes Pod (which is the atomar unit of Kubernetes). Pods can go down and restart if for example something went wrong within the pod or as well in case of re-deployments. In such a case a Pod receives a new IP. But we need to have a stable Endpoint where we can access the Pod and this is achieved via a Service. Services have a stable endpoint where we can define a name which will be resolved by kubernetes internal dns. We can then give each Service such a name and we can access the Service from pods (if they have the right labels [17]).There is one special Service we have and this is our Frontend service. We need to make the Service available from outside the cluster. In kubernetes we can therefore declare the Service type as *NodePort* which means the port from the Service will then be allocated to localhost from the host node. If we are running it on a multi node cluster the port will be the same on all nodes.

Database A database is always quite special for a deployment as the whole Container setup is meant to run stateless. Which means all the data is lost once a container dies. Mostly this is very bad for a database as there all the relevant data should be persisted.

Docker itself provides therefore persistent Volumens which can be mounted by a Container.

Within Kubernetes we create therefore a PersistentVolume [18]. A Pod can then have a persistentVolumeClaim. Kubernetes will then search for a persistent Volume which would satisfie the requirements from the claim and allocates it to the claim.

A pod can then be linked to a persistent volume via the persistent volume claim. In the following yaml file we declare first a Persistent volume and afterwards the persistent volume claim.

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: mysql-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 10Mi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv-claim
spec:
```

[17] https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/
[18] https://kubernetes.io/docs/concepts/storage/persistent-volumes/

```
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Mi
```

Within the Pod we would then link the pod to the pv claim via the following arguments.

```
    spec:
      containers:
          # container specifications ...
        volumeMounts:
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
      volumes:
      - name: mysql-persistent-storage
        persistentVolumeClaim:
          claimName: mysql-pv-claim
```

Nevertheless, this is not realy required for our application because we only need the database as a provisional storage.

A more detailed example from Kubernetes can be found under this link [19]

Deployment

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: database-deployment
  labels:
    app: rand-gen
    location: database
spec:
  replicas: 1
  selector:
    matchLabels:
      app: rand-gen
      location: database
  template:
    metadata:
      labels:
        app: rand-gen
        location: database
    spec:
      containers:
      - name: database
        image: mysql:5.6
        env:
          - name: MYSQL_ROOT_PASSWORD
            valueFrom:
              secretKeyRef:
                name: rand-secret-file
                key: rootpassword.txt
          - name: MYSQL_USER
```

[19] https://kubernetes.io/docs/tasks/configure-pod-container/configure-persistent-volume-storage/

```
            valueFrom:
              secretKeyRef:
                name: rand-secret-file
                key: username.txt
          - name: MYSQL_PASSWORD
            valueFrom:
              secretKeyRef:
                name: rand-secret-file
                key: password.txt
          - name: MYSQL_DATABASE
            valueFrom:
              configMapKeyRef:
                name: random-config-env
                key: MYSQL_DATABASE
        ports:
        - containerPort: 3306
```

```
apiVersion: v1
kind: Service
metadata:
  name: database-svc
  labels:
    app: rand-gen
spec:
  selector:
    app: rand-gen
    location: database
  clusterIP: None
  ports:
  - port: 3306
```

Random Generator Now we also declare the deployment for the generator service. Notice that we have two replicas for the generator. this is because we want to have two diffrent ids as a source for our generator.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: random-generator-deployment
  labels:
    app: rand-gen
    location: generator
spec:
  replicas: 2 # to showcase diffrent generator ids
  selector:
    matchLabels:
      app: rand-gen
      location: generator
  template:
    metadata:
      labels:
        app: rand-gen
        location: generator
    spec:
      containers:
      - name: random-generator-container
        image: toky03/rand-gen-image:1.0
        ports:
```

```
        - containerPort: 5050
```

The corresponsing service.

```
apiVersion: v1
kind: Service
metadata:
  name: random-generator-svc
  labels:
    app: rand-gen
spec:
  selector:
    app: rand-gen
    location: generator
  ports:
  - protocol: TCP
    port: 5050
    targetPort: 5050
```

Middle Tier

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: middle-tier-controller
  labels:
    app: rand-gen
    location: middle
spec:
  replicas: 1
  selector:
    matchLabels:
      app: rand-gen
      location: middle
  template:
    metadata:
      labels:
        app: rand-gen
        location: middle
    spec:
      containers:
      - name: random-controller-container
        image: toky03/rand-middle-image:1.0
        envFrom:
        - configMapRef:
            name: random-config-env
        env:
          - name: SPRING_DATASOURCE_USERNAME
            valueFrom:
              secretKeyRef:
                name: rand-secret-file
                key: username.txt
          - name: SPRING_DATASOURCE_PASSWORD
            valueFrom:
              secretKeyRef:
                name: rand-secret-file
                key: password.txt
        ports:
        - containerPort: 7070
```

The service

```
apiVersion: v1
kind: Service
metadata:
  name: middle-tier-svc
  labels:
    app: rand-gen
spec:
  selector:
    app: rand-gen
    location: middle
  ports:
  - protocol: TCP
    port: 7070
    targetPort: 7070
```

Statistic Tier

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: random-statistic-deployment
  labels:
    app: rand-gen
    location: stat
spec:
  replicas: 1
  selector:
    matchLabels:
      app: rand-gen
      location: stat
  template:
    metadata:
      labels:
        app: rand-gen
        location: stat
    spec:
      containers:
      - name: random-statistic-container
        image: toky03/rand-stat-image:1.0
        envFrom:
        - configMapRef:
            name: random-config-env
        env:
          - name: SPRING_DATASOURCE_USERNAME
            valueFrom:
              secretKeyRef:
                name: rand-secret-file
                key: username.txt
          - name: SPRING_DATASOURCE_PASSWORD
            valueFrom:
              secretKeyRef:
                name: rand-secret-file
                key: password.txt
        ports:
        - containerPort: 8080
```

Service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: random-statistic-svc
  labels:
    app: rand-gen
spec:
  selector:
    app: rand-gen
    location: stat
  ports:
  - protocol: TCP
    port: 8080
    targetPort: 8080
```

Frontend

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend-deployment
  labels:
    app: rand-gen
    location: frontend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: rand-gen
      location: frontend
  template:
    metadata:
      labels:
        app: rand-gen
        location: frontend
    spec:
      containers:
      - name: frontend-container
        image: toky03/rand-front-image:1.0
        envFrom:
        - configMapRef:
            name: random-config-env
        ports:
        - containerPort: 8080
```

Service:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: frontend-svc
  labels:
    app: rand-gen
spec:
  selector:
    app: rand-gen
    location: frontend
  type: NodePort # make the service accessible from outside
  ports:
```

```
13   - protocol: TCP
14     port: 5123
15     targetPort: 8080
```

With the command `kubectl get svc` we will get all the Services and we should see that the frontend-svc should have the type NodePort. Now we can access the Application via the Ip address of our node and this port number.

Deleting the whole Application To remove all the configuration and services we can run the following commands and everything should be gone.

```
kubectl delete -f frontend.yml
kubectl delete -f stat-tier.yml
kubectl delete -f middle-tier.yml
kubectl delete -f generator.yml
kubectl delete -f database.yml
kubeclt delete -f config-env.yml
kubectl delete secret rand-secret-file
```