Kubernetes Full Stack Application

Marco Jakob

April 4, 2019

Contents

1 Introduction **3**

2 Prerequisites **3**
2.1 Technology 3
2.2 Model 4

3 Create Services **4**
3.1 Database 4
3.2 Random Generator 4
3.3 Middle Tier 5
3.4 Statistic Service 9
3.5 Frontend 12
3.6 Creating the Docker Images 19
3.7 running with Docker 20

4 Installing Kubernetes **20**

5 Deploy Services to Kubernetes **20**

1 Introduction

This Introduction should give an end to end overview to deploy a sample Application with multiple Services and a Databases completely on Kubernetes. 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 Random Numbers based on their Producers Id. The Goal of the whole application is first to create an end to end Application fully Cloud native which runs as well locally without any changes. 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)
- Maven build tool
- Python 3.6
- Docker
- Angular (minimum version 6) with the ng command

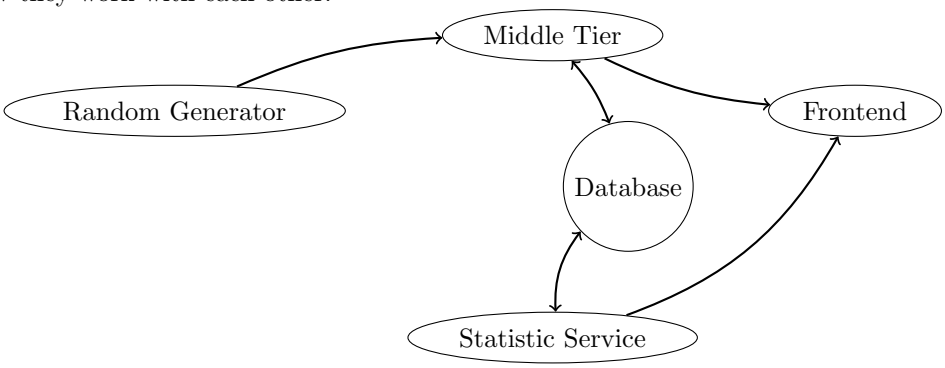
2.1 Technology

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 18.5.1 LTS

All the Applications are Running inside of an own Docker Container and Managed within a Kubernetes Pod and accessible via a Kubernetes Service where only the Frontend Service is bound to a Node Port.

2.2 Model

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



3 Create Services

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.

```
CREATE database db_example;
CREATE USER 'springuser'@'localhost' IDENTIFIED BY ThePassword;
GRANT ALL PRIVILEGES ON *.db_example TO 'springuser'@'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 Random Number. For this application we create a folder called RandGen with the following Structure.

`RandGen`

```
├── 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].

```python
from flask import Flask, jsonify
import random
import uuid

app = Flask(__name__)
user_id = uuid.uuid4().int

@app.route('/')
def random_number():
    object = {'id' : str(user_id),
              'randNumber': str(random.randint(0,1000))}
    return jsonify(object)

if __name__ == '__main__':
    app.run(port=5050,host='0.0.0.0')
```

In requirements.txt are the Python packages specified. In this case it is flask. therefore this file contains only one line which is

```
Flask==1.0.2
```

The Dockerfile to create the Docker image:

```
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"]
```

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:

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

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

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
```

all blue folders are already given. the Dockerfile still has to be created.

Now inside of the Project 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;
```

```
2 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 are created

```
1 @Query("Select count(ps.id) from PodStat ps where ps.id = ?1")
2 Long countUniqueId(String id);
3
4 @Query("Select count(ps.id) from PodStat ps")
5 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.

```
1 @Autowired
2 private ObjectMapper objectMapper;
3 @Autowired
4 private PodStatRepository podStatRepository;
5 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 a controller Method with the business logic can be declared:

```
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();
```

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

```
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.

```
1 spring.jpa.hibernate.ddl-auto=update
2 spring.datasource.username=random_user
3 spring.datasource.password=random_password
4 spring.datasource.url=jdbc:mysql://localhost:3306/rand_numbers
```

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

```
1  FROM maven:3.5-jdk-8 AS builder
2
3  COPY ./pom.xml /app/pom.xml
4
5  COPY ./src /app/src
6  ADD ~/.m2 /root/.m2
7  RUN mvn -f /app/pom.xml  package -DskipTests
8
9  FROM anapsix/alpine-java:latest
10
11 COPY --from=builder /app/target/middletier-0.0.1-SNAPSHOT.jar /opt/
       middle-tier.jar
12
13 ENV SPRING_DATASOURCE_URL=jdbc:mysql://rand-gen-database/
       rand_numbers
14
15 ENV SPRING_DATASOURCE_USERNAME=random_user
16
17 ENV SPRING_DATASOURCE_PASSWORD=random_password
18
19 ENV RANDOM_GENERATOR_URL=rand-gen-app
20
21 CMD java -jar /opt/middle-tier.jar
```

We are using anapsix base image, as we want to keep the Docker image as small as possible.

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.

```
1 docker build -t middle-tier .
```

Reference for using maven with Docker `https://hub.docker.com/_/maven/`

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/`

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.

There are also other options to tackle this for example with the spotify maven plugin [3]. Which would provide a new command for maven to build the dockerfile along with building the jar file. But as this Tutorial should give an intro with plain docker witout any additional plugins. We decided to do it with a multi stage build.

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.

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
```

For this time, we will leave the Main Class allone in the file StatserviceApplication.java. Which means we do not have to change this file as Spring Boot already did everything needed for us.

[3] Spotify Maven Plugin `https://github.com/spotify/docker-maven-plugin`

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.

```
@Autowired
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> getAllPodStatsHistory(){

 return statService.getPodHistory();
}
```

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

next we are going to declare all our Entities.

All of them are annotated with @Entity. 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 not annotated with @Entity as it is only a helper 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) {
 super();
 this.name = name;
 this.series = new ArrayList<SeriesItem>();
}
...

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

SeriesImtem is the second helper 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;
}
```

```
private String id;
private Long counter;

public Stat(String id, Long counter) {
 super();
 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(ps.id) from PodStat ps where ps.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(){

 Iterable<PodStat> source = podStatRepository.findAll();
```

```
22  Map<String, Series> podHistorys = new HashMap<>();
23  source.forEach( (entry) -> {
24   if(podHistorys.get(entry.getId()) == null) {
25    podHistorys.put(entry.getId(), new Series(entry.getId()));
26   }
27   podHistorys.get(entry.getId())
28   .appendSeriesItem(new SeriesItem(entry.getTimeStamp(), entry.
       getCounter()));
29  });
30
31  List<Series> plainSeries = new ArrayList<>(podHistorys.values());
32  return plainSeries;
33 }
```

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

```
1  FROM maven:3.5-jdk-8 AS builder
2
3  COPY ./pom.xml /app/pom.xml
4
5  COPY ./src /app/src
6  ADD ~/.m2 /root/.m2
7  RUN mvn -f /app/pom.xml  package -DskipTests
8
9  FROM anapsix/alpine-java:latest
10
11 COPY --from=builder /app/target/statservice-0.0.1-SNAPSHOT.jar /opt
       /stat-tier.jar
12
13 ENV SPRING_DATASOURCE_URL=jdbc:mysql://rand-gen-database/
       rand_numbers
14
15 ENV SPRING_DATASOURCE_USERNAME=random_user
16
17 ENV SPRING_DATASOURCE_PASSWORD=random_password
18
19 CMD java -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.

3.5 Frontend

The frontend Application is build with Spring Boot and

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

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

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 we can try if the project has been set up correctly with navigating to the new project folder and hit ng serve. Once the server is ready we should be able to access localhost:4200 and see a sample home page made by angular cli.

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

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
```

to install all dependent libraries for

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. Therefore we need to change the output path option in the file angular.json

```
...
    "outputPath": "target/classes/static",
    ...
```

Now every time the Angular Project is built it will be compiled directly to the path where spring boot serves static files. When we now start the spring boot application we will get the Angular Project 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
```

Interface RandomNumber

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

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.

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

```
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.

```
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 now create a seperate Module to manage all depedencies for Material design.

```
ng generate module material
```

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

```
 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 [4]

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

We wil use Ngx-Charts Module to display the statistics as a graph. [5]

and we install the module with the following command.

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

Now we can start creating the logig within the Main Component. first we declatre the Variables which we are going to use.

```
randomNumber: RandomNumber = new RandNumberImpl();
statistics: PodStatistic[] = [new PodStatisticImpl()];
multi: Series[];
```

In the constructor, we provide our two Services as Private.

next we create the Methods which fetches the data from the service.

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

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

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

[5] Documentation can be found here `https://pixinvent.com/apex-angular-4-bootstrap-admin-template/documentation/documentation-charts-ngx.html`

```
getHistory(){
  this.statistic_service.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 within ngOnInit we call only buttonClick().
As a last thing in this file we need to add some configuration Variables.

```
// Table
displayedColumns: string[] = ['id','counter'];
dataSource: any;

// 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;
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 Numbers</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
    >
```

```
16
17 <mat-card>
18   <mat-card-header>
19     Statistics Table
20   </mat-card-header>
21   <mat-card-content >
22     <table mat-table [dataSource]="dataSource" class="mat-elevation
     -z8">
23       <ng-container matColumnDef="id">
24         <th mat-header-cell *matHeaderCellDef>ID</th>
25         <td mat-cell *matCellDef="let element">{{element.id}}</td>
26       </ng-container>
27       <ng-container matColumnDef="counter">
28         <th mat-header-cell *matHeaderCellDef>Counter</th>
29         <td mat-cell *matCellDef="let element">{{element.counter}}<
     /td>
30       </ng-container>
31       <tr mat-header-row *matHeaderRowDef="displayedColumns"></tr>
32       <tr mat-row *matRowDef="let row; columns: displayedColumns;">
     </tr>
33     </table>
34   </mat-card-content>
35 </mat-card>
36
37     <ngx-charts-line-chart
38     [view]="view"
39     [scheme]="colorScheme"
40     [results]="multi"
41     [gradient]="gradient"
42     [xAxis]="showXAxis"
43     [yAxis]="showYAxis"
44     [legend]="showLegend"
45     [showXAxisLabel]="showXAxisLabel"
46     [showYAxisLabel]="showYAxisLabel"
47     [xAxisLabel]="xAxisLabel"
48     [yAxisLabel]="yAxisLabel"
49     [autoScale]="autoScale"
50     [timeline]="timeline"
51     (select)="onSelect($event)">
52     ></ngx-charts-line-chart>
53
54
55 </section>
```

As next thing we can finally go through the Server side Part of the Frontend, which is from our Spring Boot Part. 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:

```
private Long podStatID;
private String id;
private Long timeStamp;
private Long counter;
```

Fields from RandomNumber:

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

Series:

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

SeriesItem:

```
private Long name;
private Long value;
```

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

To fetch all the Data we need to declare in the file RestService.java all the methods which communicate to the Statistic Service and to the Middle Tier Service.

As we want to inject this Service within the Main Application we need to annotate this Class with @Service.

Fields from RestService:

```
private Map<String, String> env = System.getenv();
private final String statURL = env.get("STAT_URL");
private final String randNumURL = env.get("RAND_NUM_URL");
private final String historyURL = env.get("HISTORY_URL");
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 Configuration.

Now we declare the methods.

```
public List<PodStat> getStatistics() {
 PodStat[] tmpStats = restTemplate.getForObject(statURL, PodStat[].
    class);
 return Arrays.asList(tmpStats);
}

public RandomNumber getRandomNumber() {
 return restTemplate.getForObject(randNumURL, RandomNumber.class);
}

public List<Series> getHistory() {
```

```
 Series[] series = restTemplate.getForObject(historyURL, Series[].
    class);
 return Arrays.asList(series);
}
```

With this we are finished with the Rest Service File.

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.

We need one dependency which we need to inject with @Autowired and this is our previously declared RestService;

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

The last thing we need to do is to declare all the 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(){
 return restService.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.

```
FROM node:10-alpine as frontbuilder

COPY . .

RUN npm install

RUN npm run ng build -- --prod

FROM maven:3.5-jdk-8 AS builder

COPY --from=frontbuilder /pom.xml /app/pom.xml

COPY --from=frontbuilder /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
```

```
20
21 ENV SPRING_DATASOURCE_URL=jdbc:mysql://rand-gen-database/
       rand_numbers
22
23 ENV SPRING_DATASOURCE_USERNAME=random_user
24
25 ENV SPRING_DATASOURCE_PASSWORD=random_password
26
27 ENV RANDOM_GENERATOR_URL=rand-gen-app
28
29 CMD java -jar /opt/middle-tier.jar
```

3.6 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. It is mostly important for the build from our Frontend, 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.

Create Random Generator within the root directory of the Random Generator.

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

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

3.7 running with Docker

Create a Network

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

Start the Database Container

```
1 docker run -d --name rand-gen-database \
2 -e MYSQL_ROOT_PASSWORD='password' \
3 -e MYSQL_USER='random_user' \
4 -e MYSQL_PASSWORD='random_password' \
5 -e MYSQL_DATABASE='rand_numbers' \
6 --network rand-gen-network \
7 mysql:5.6
```

Start the Random Generator App

```
1 docker run -d --name=rand-gen-app \
2 --network rand-gen-network \
3 rand-gen-image
```

Start the middle tier Application

```
1 docker run -d \
2 --name middle-tier \
3 --network rand-gen-network \
4 middle-tier
```

Starting the Stat Tier Application

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

Start the Frontend Service

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

4 Installing Kubernetes

5 Deploy Services to Kubernetes