WEB AND CLOUD COMPUTING 2017

Project: Uber for Bikes

Group 18

Authors
Thom Carretero Seinhorst
Alexandru Tatarov
Anastasia Serebryannikova

November 4, 2017

1 Introduction

In this project we developed a single page web application "Uber for bikes". The final goal was to create something similar to the Uber functionality for bikes.

The application represents a map on which the positions of the bikes are shown. These bikes are used by people for transportation. The users can also see the table with the address information about the bikes. The coordinates of the bikes are streamed from the database and then transformed into addresses using the Google Maps API. The user can book a bike by pressing the corresponding button. When a bike is taken, its corresponding point on the map disappears (because it is no longer available) for all users except the user who reserved it. For that user the marker will be updated every 10 seconds in order to help the user reach the end-point easier. When the trip is finished the bike appears back on the map.

2 Architecture

In Figure 1, a broad overview of the architecture of the application is given. There are four main services that we implemented: a client, an Nginx reverse proxy, a webserver (denoted as Play) and a MongoDB database.

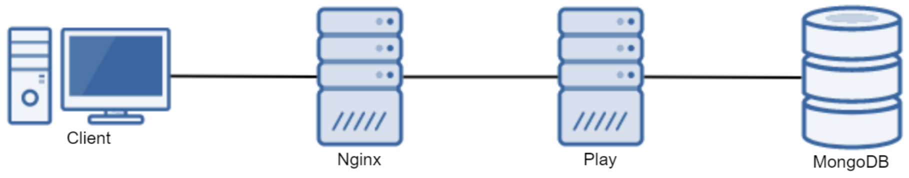


Figure 1: Overview of the architecture of Uber for Bikes

For our webserver we have written the main component in Scala. Scala is a functional programming language supporting high scalability. We used the Play framework which fits our purposes very well as it is designed for developing web applications. As well as this, it ensures the integration with the database through the corresponding driver.

For our project we used the MongoDB database with a ReactiveMongo driver. ReactiveMongo is a Scala driver that provides fully non-blocking and asynchronous I/O operations. ReactiveMongo is designed to avoid any kind of blocking request. Every operation returns immediately, freeing the running thread and resuming execution when it is over.

In the frond-end, AngularJS and Bootstrap are used for providing a nice user interface. The UI serves a simple purpose: renting and unrenting the bikes. The main view of the UI is a Google Maps view. It is currently set to the location of Groningen, but could eventually be used to show the region where the user resides.

In our project we use an external third party service. The service of Google is needed to enable the conversion between a textual representation of a location and the corresponding geographical representation of the GPS system. This conversion is done via the Google Maps API.

2.1 Docker swarm

We deploy our application using the docker swarm mode. Docker swarm is a tool that provides high scalability and fault-tolerance with minimum effort. With Docker swarm, it is possible to run applications on a cluster of virtual or physical machines. If one of the nodes or one of the containers fails, the other keep functioning, thus the whole system remains stable. The scalability can be controlled easily as well. For every service, there are two modes available, global and replicated. If the replicated mode is used, we can define the number of replicas for every service. After that, the scheduler distributes them among the available nodes of the cluster. If the global mode is used, every node in the cluster runs exactly one instance of the service. In our application, we use the global mode for all the services except one (see 3.1), which means that if we run our application on several (virtual) machines, each machine will always have exactly one instance of every service running on it.

3 Design decisions

3.1 Database

We used MongoDB to provide the database service. We chose to use this database as it is easy to implement and easy to use. We have a MongoDB database with 1 collection, Bikes. The Bikes collection stores all bike locations and their availability. There should be a second database collection, Users, for storing the name, passwords and a list of rented bikes. In fact, we have implemented most of the back-end and front-end functionality to provide this but due to the lack of time we weren't able to connect the front-end implementation with the back-end one.

The database entry for a bike looks as represented in Figure 2. Every bike has an automatically generated id, a pair of coordinates and a boolean availability status. The ids are unique for all the instances.

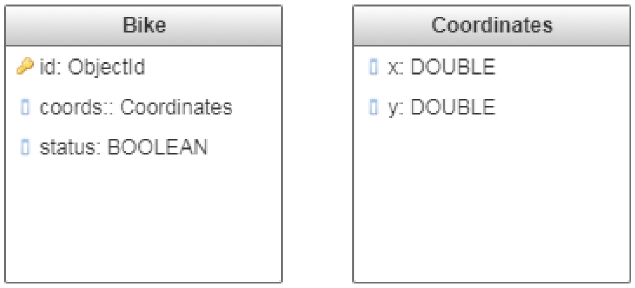


Figure 2: Bike model

In the docker swarm mode, the Mongo replica set is used to enable sharing of the data between the instances of MongoDB on different hosts. Replica sets provide redundancy and increase the availability of the data. All write operations are performed to the primary node and then the data is replicated to the secondary nodes, as shown in Figure 3. When communicating with the database, the driver is responsible for detecting the primary node. ReactiveMongo does not need any additional configuration for that and works perfectly both when there is a single instance of the database and a replica set.

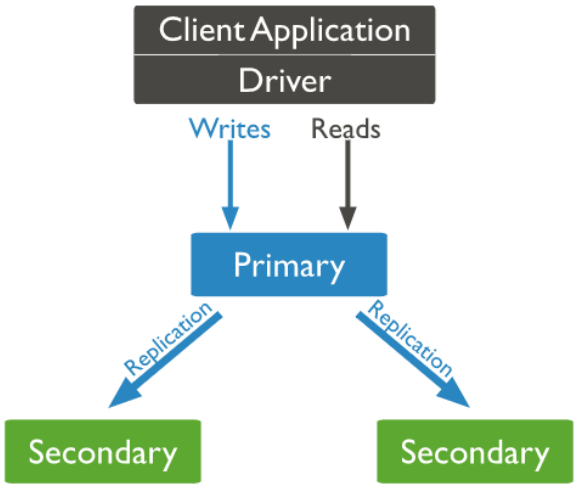
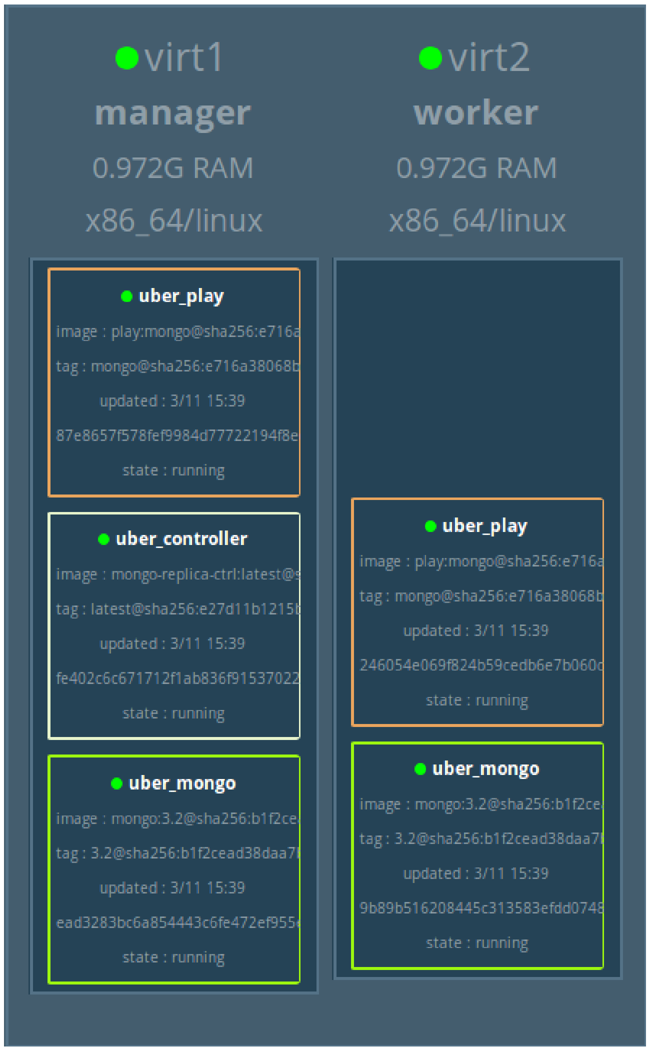


Figure 3: Replica sets in MongoDB

As opposed to the other services that are deployed in the docker swarm mode, MongoDB needs some additional configuration. Docker cannot take care of initializing the replica set and making sure that the data is correctly replicated among different nodes. This means that we have to manage the replica set by ourselves. If the replica set is configured manually there is a high risk that if one of the MongoDB instances goes down and is then recreated again, its IP address will be different and the replica set will need to be reconfigured (the new IP address of the node should be added and the old one removed). This may cause the read and write operations to fail, which is not what we want. So we have found a way to deal with this problem. We have a separate container (called *controller*) that maintains the replica set up and running and updates the configuration whenever necessary. This container does not need replication and is deployed only once on the manager node. So the backend, when deployed in the swarm mode, looks as represented in Figure 4.



```
virt1
manager
0.972G RAM
x86_64/linux

uber_play
image : play:mongo@sha256:e716a
tag : mongo@sha256:e716a38068b
updated : 3/11 15:39
87e8657f578fef9984d77722194f8e
state : running

uber_controller
image : mongo-replica-ctrl:latest@s
tag : latest@sha256:e27d11b1215b
updated : 3/11 15:39
fe402c6c671712f1ab836f91537022
state : running

uber_mongo
image : mongo:3.2@sha256:b1f2ce
tag : 3.2@sha256:b1f2cead38daa7b
updated : 3/11 15:39
ead3283bc6a854443c6fe472ef955
state : running

virt2
worker
0.972G RAM
x86_64/linux

uber_play
image : play:mongo@sha256:e716a
tag : mongo@sha256:e716a38068b
updated : 3/11 15:39
246054e069f824b59cedb6e7b060c
state : running

uber_mongo
image : mongo:3.2@sha256:b1f2ce
tag : 3.2@sha256:b1f2cead38daa7b
updated : 3/11 15:39
9b89b516208445c313583efdd0748
state : running
```

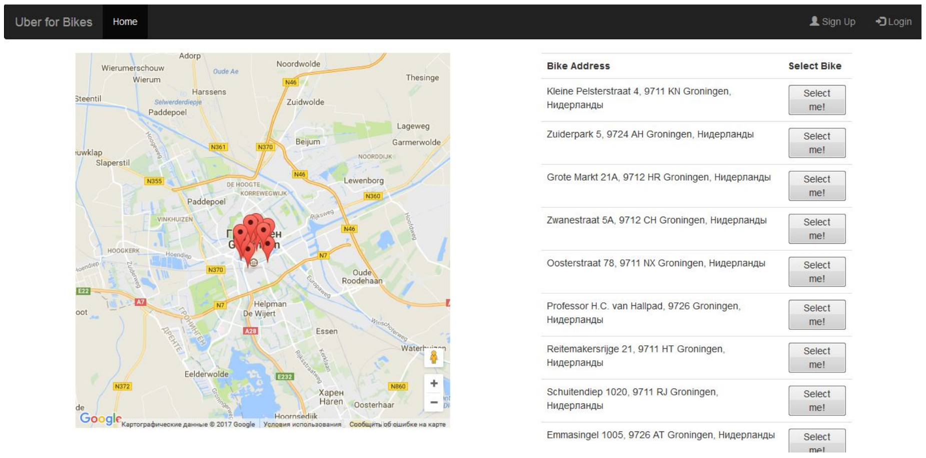
Figure 4: MongoDB replication in docker swarm mode

3.2 Webserver

We chose Play as the framework for the back-end development of our application mostly because it supports asynchronous I/O. There were two main causes for choosing Scala: the first is the opportunity to learn a new programming language and the second is professor's incentive to use Scala for back-end development. In the end we succeeded in building a working back-end. We tried to benefit as much as possible from the completed projects we found on the Internet. They contributed a lot to our understanding of how everything needs to be done and to the development of the final product. During the development we had difficulties connecting different services (back-end, front-end, database) to each other as well as using WebSockets for live data streaming.

3.3 Client

Angular is a framework for building Web single-page applications and Bootstrap is an easy to use front-end web framework for designing web applications. These are the main reasons why we chose to use these frameworks. Other advantage of using Angular is its highly readable and comprehensive code. It was an important factor because we wanted something that wouldl take as little time to learn as possible. We created the project using Angular Cli and used the corresponding components to create the navigation bar, the map component, and two (login and registration) forms (unfortunately not seen in the final layout). Even if it is not linked to the framework itself, we found the large community of developers using Angular extremely useful as online answers helped us solve most encountered problems.



Bike Address	Select Bike
Kleine Pelsterstraat 4, 9711 KN Groningen, Нидерланды	Select me!
Zuiderpark 5, 9724 AH Groningen, Нидерланды	Select me!
Grote Markt 21A, 9712 HR Groningen, Нидерланды	Select me!
Zwanestraat 5A, 9712 CH Groningen, Нидерланды	Select me!
Oosterstraat 78, 9711 NX Groningen, Нидерланды	Select me!
Professor H.C. van Hallpad, 9726 Groningen, Нидерланды	Select me!
Reitemakersrijge 21, 9711 HT Groningen, Нидерланды	Select me!
Schuitendiep 1020, 9711 RJ Groningen, Нидерланды	Select me!
Emmasingel 1005, 9726 AT Groningen, Нидерланды	Select me!

Figure 5: Application layout

4 Discussion

In this chapter we review and reflect on the project.

At the beginning of the project we started with developing an UI. After that we created an nginx webserver and implemented a backend service using Play framework, but we struggled a lot to provide the connection from the front-end to the back-end. It took us quite a lot of time to figure it out how exactly to connect all the parts together. We ended up using Rest API to connect the front-end to the back-end. We also tried to implement WebSockets. At the end we finally managed to successfully retrieve the data from the database and to present it in the UI. As already mentioned, we chose to use Angular as a framework for our UI.

It took us some time to set up the back-end with Play and Scala. Since the programming language Scala was new to all of us, we had to put some time and effort in understanding how Scala works.

4.1 Angular

Angular proved to be the easiest component of the project. However not everything was straightforward. One tricky part was using Ajax calls to retrieve information about available bikes and displaying it. We needed a little more time than expected to use the values from *.subscribe()* in some other function.

4.2 Scala

Scala was completely new to all of us. We were lucky enough to find a project that provided some similar functionalities. Using that project as a guideline, we implemented a part of our own back-end functionality that communicates with front-end and the database. The biggest issue we encountered at using Scala was implementing WebSockets because we could not find many examples online.

4.3 Docker

Docker appeared to be an extremely nice tool for developing applications. With Docker, it has become very easy to separate the infrastructure decisions from the internal application issues. Docker was pretty easy to learn and quite easy to use. The documentation appears to be more or less comprehensive and Docker itself seems to be very user-friendly. The main issues that we had in this part were mainly connected to deploying certain containers and figuring out the correct settings for them. For example, sbt failed to dockerize our Play application, so we had to discover an alternative way to do that. Replicating MongoDB instances and importing data in the database also caused some problems. However, the docker logic overall seemed to be very straightforward and did not cause any issues.

Docker swarm mode that we used in our application appears to be extremely useful in providing fault tolerance and its logic is easy to understand. It enables fault tolerance in a very intuitive way by replicating the services among different nodes in the cluster or even within a single node. Getting acquainted with Docker definitely was one of the most valuable and beneficial tasks during this course.

4.4 Future work

Probably the most important feature that is missing is the live streaming of data. It was supposed that bike locations will change every 10 or 30 seconds in correspondence to the real ones. Another important part of the project that is missing is the login and register part for users of the application. By saying that it is missing we mean that it is not functional. The form components have been created, but they do nothing when *Submit* button is pressed. Users should be able to create an account and to login. When a user is logged in, he can rent a bike and see his history of rented bikes. This history can vary from just the day and the coordinates/address of the start and end points to a map that shows all the intermediary points recorded using live data streaming. Another missing part is the history of the bike. Due to the difficulties we encountered there was not enough time to implement this correctly. Finally, some stylistic upgrades can be done. However this is not important in the context of the functionality of the project.