university of groningen

faculty of mathematics and natural sciences

Web and Cloud Computing 2016

Project: Uber for Bikes

Report for Group 23

Peter Ullrich
S2273942
p.j.ullrich@student.rug.nl

Yannik Beckersjürgen
S2303779
Y.Beckersjurgen@student.rug.nl

Nadia Hartsuiker
S2355809
n.l.hartsuiker@student.rug.nl

Contents

1 Introduction

Uber for Bikes or Ubob is, as the title suggests, an application inspired by the famous Uber application. Ubob allows users to rent bikes in an area of their choosing. Users can see if and where around them bikes are available for rental. With just a few clicks a bike is selected and rented. When the bike is no longer needed it is 'unrented' just as easily.

2 Implemented Functionalities

The following functionalities have been implemented:

For the user

- Creation of an account, enabling users to sign up.
- User Login
- Display of available bikes on a map
- Display of rented bikes on a map distinguishable from unrented bikes
- Rental and unrental of bikes

General functionalities:

- Automated setup using Vagrant
- Distributed servers
- Encapsulation using docker
- Load balancing across servers using HAProxy
- Implementation of (nosql) database to store data. (MongoDB)
- Sharding of database across application servers for fault-tolerance

3 Overview of the Architecture

In the Figure 1, a broad overview of the architecture of the application is given. An instance of the application has several virtual machines running so that, in theory, each could be placed on different hardware. With the current initialisation the application has one load balancing VM and three web-backend VMs running. To remain online the load balancing VM and at least one back-end VM are needed.
The load balancing VM runs both HAProxy and a Consul sever to balance the load across the available servers. Consul Server is used to properly pass ports and addresses and runs also on each web-backend VM. On these backend VMs a Registrator registers the services in Docker containers. These are Ubob, which contains the actual content of the application and MongoDB, the database containing the data of bikes and users.

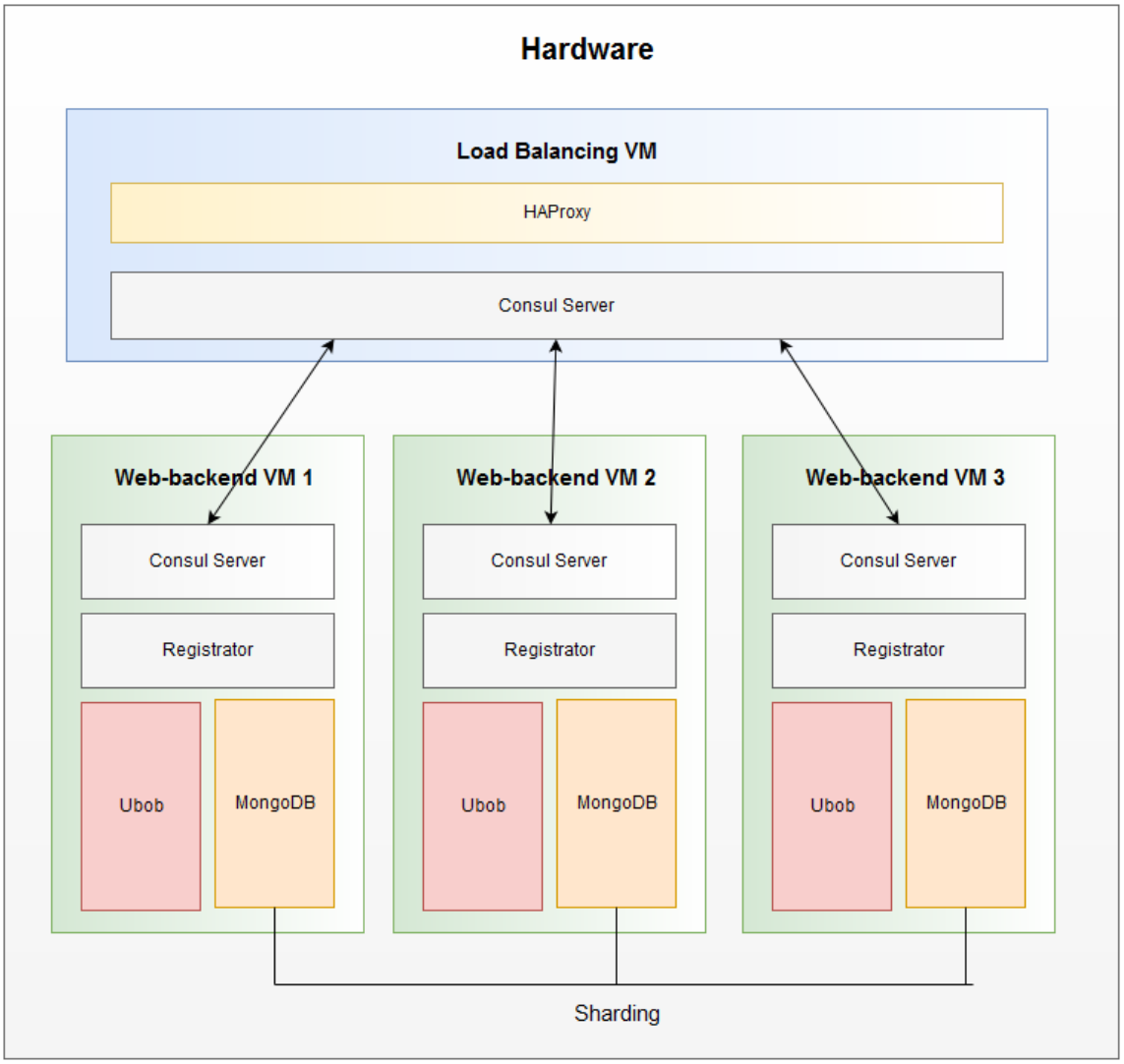

Figure 1: Broad overview of the architecture of Uber for Bikes

.

The Figure 2 gives a broad overview of the architecture of Ubob. This contains the UI which takes care of interaction with the user. For communication with the backend a Websocket and a Servlet are used. The Servlet is used for the Login and Register interaction and the Websocket for all other user interactions like renting and 'unrenting' a bike and updating a bike object or the user rented bikes list. Both Servlet and Websocket communicate with the DB controller which communicates with MongoDB through a MongoClient using a local port. The Ubob application and the MongoDB are both Docker containers linked together to avoid making common ports publicly available.

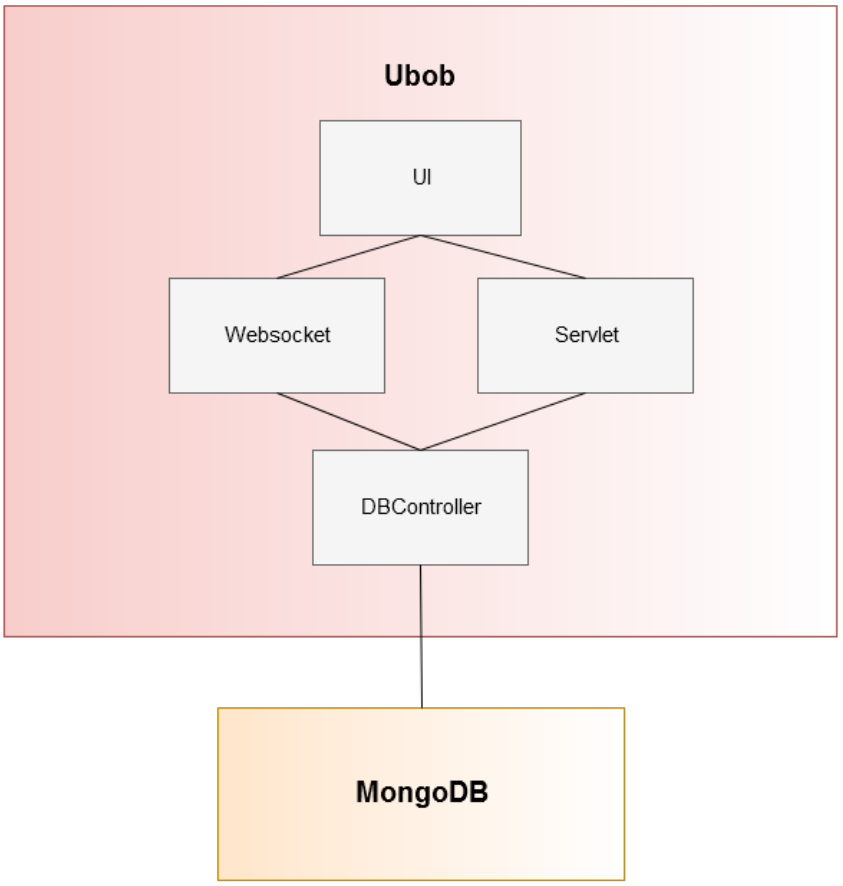

Figure 2: Overview of the internal architecture of Ubob

.

4 Design Aspects and used Technologies

In this section different design aspects are discussed. For each design aspect used technologies and design decisions are discussed.

5 Initialisation

The initialisation, the process of starting the application and making it available online has been automated using Vagrant. Vagrant creates the images of the virtual machines, installs docker and its containers and runs the SHELL scripts, which we use to configure each VM.

On execution, Vagrant creates a Load Balancing VM, which gets provisioned with Docker, Consul and HAProxy. The consul instance on this VM acts as a leader to which subsequent consul instances will connect. Next, Vagrant creates a number of application virtual machines, which also get provisioned with Docker. Docker in turn installs a consul Docker container, which connects to the leader consul instance on the Load Balancing VM. After deploying the consul instance, a Registrator Docker container is deployed, which connects to the Docker socket in order to be informed about further Docker container deployments. Subsequently, our Ubob application and MongoDB 3.3 are pulled from the Docker hub and deployed as well. The Ubob application gets linked to the MongoDB instance via Docker so that the internal ports are not exposed to the public.

6 Service discovery

To keep track of our application virtual machines and their Ubob and MongoDB instances, we use Consul. This helps us to easily use HAProxy for load balancing as these two applications can automatically be connected using e.g. consul-template. Consul-template allows us to retrieve the current server instances running a specific service (in our case Ubob) and to add these servers to HAProxy. HAProxy can then easily balance the load from incoming requests along these servers.

For administrative purposes, we have enabled the statistics of HAProxy, which show the connected servers and the current load balancing. This data is available via the prepared UI of HAProxy running on a certain port on the load balancing virtual machine (Port :1936). Additionally, we also enabled the UI of Consul in order to keep track of the registered services and the servers running Consul. This UI can also be accessed through a dedicated port (Port :8500/ui)

7 Load Balancing

As discussed before, for load-balancing a separate Virtual machine is created. This VM runs HAProxy. HAProxy offers the implementation of several load balancing algorithms. Given the fact that the Ubob application is relatively small in terms of users, hardware and traffic, a layer 4 load balancing algorithm has been chosen. This is the simplest way to balance network traffic to multiple servers. The user accesses the load balancer which forwards the request to one of the application servers. The chosen server will respond directly to the user's request. Servers all contain the same content.

The algorithm used to select servers is roundrobin. This is the simplest and default algorithm HAproxy supports. Roundrobin selects for each request a server at random. Given the level of complexity of our application, this algorithm is sufficient.

In its basic implementation roundrobin does not suffice to keep users logged in. In the basic implementation user sessions cannot be maintained because each request (e.g. rent/unrent a bike or login) is send to a different server at random. This can be solved in two different ways. The first is the use of cookies. The second the implementation of so-called 'sticky sessions'. This means that, once a user has performed a login action, all server requests from that user are done to the same server. Sticky sessions are significantly easier to implement than the use of cookies. The drawback of sticky sessions compared to cookies is that once the connection with the selected server is lost, the user is no longer logged in. This means that if the server to which a user is assigned crashes, the user will have to log in again to another server to continue their activities. We did not consider this drawback to have a significant influence on the performance of our application. Because of the limited time frame we have chosen for the implementation of sticky sessions.

To evaluate the performance of the combination of roundrobin and sticky sessions once the application is in use and the traffic increases, HAproxy offers an UI that gives an overview of the way the load distributed over the different servers. This can be used to provide feedback to administrators and, if needed, to improve the load balancing.

8 UI

The UI serves a simple purpose: Renting and unrenting bikes. A user can register to our service with a username and a password. If the user is already registered, the Login view can be used. The register and login requests are handled by a Servlet on the server-side, because we can send back an http-response to such a request which we are not able to do using a websocket. 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. The Google Map shows all available bikes and, given that the user is logged in and has at least one bike rented, the rented bikes of the user in green. The rented bikes are also listed and can be unrented on the left.

A user can rent a bike by clicking on an available bike and clicking the rent button. This will send a notification to the back-end via a Websocket that the bike with the given bike UUID was rented or unrented by the user with the given user UUID. The back-end will update the availability of the bike and the bikes rented list of the user in the database and send back an updated version of the bike object and the bikes rented list to the front-end. AngularJS then updates the information displayed based on these updates. The rent function is only available if the user is logged in. Once a bike is rented it will vanish from the Google Maps of other users as we use AngularJS for live updates.

To have both applications, the front-end webserver and back-end Java application, running in a single instance, we use **Glassfish 4.1.1**. This allows us to easily deploy a single Web Application Archive (WAR) which contains all functionality needed. This automatisation of the deployment is especially helpful as we develop our front-end in HTML, CSS, and Javascript whereas the back-end is coded in Java 7. To connect such different applications normally takes a lot of time and effort, but is automatised using Glassfish.

9 Database

As a database we use MongoDB Version 3.3. We chose to use this database as it is easy to implement and to use using a Java development environment. We have a MongoDB database with 2 collections, **Users** and **Bikes**. **Users** stores the usernames, passwords and a list of the bikes rented by the user. The **Bikes** collection stores all bike locations and their availability.

10 Fault tolerance

Fault tolerance is implemented by creating several servers each containing the same docker containers. Ubob that enables user interaction with the back-end and MongoDB containing stored data of users and bikes. The instances of MongoDB are synchronised. Two of the three created back-end VMs can crash without severe consequences for the user experience. If a user has a sticky session (see 7 Load Balancing) at a server that crashes, the user will have to log in again before continuing their activities. There is only one load distribution VM, if this VM crashes, the application crashes as well.

11 Testing

We use a few JUnit tests to evaluate the functionality of the MongoDB connection. These tests are not automatically executed but can be used manually for debugging. Furthermore, we conducted a fair amount of monkey testing in order to make the UI of our application fool-proof.

12 Evaluation

We started our project by developing an UI. In the beginning we used an Nginx webserver, but struggled to connect it with any Java classes. Following a recommendation of a teaching assistant we switched to Glassfish 4.1.1 in order to have an easy connection between front-end and back-end. It took us some time to figure out how exactly to connect these parts but eventually we managed to retrieve data from the database and to present it in the UI. We chose to use AngularJS as a framework for our UI as it simplifies tremendously the dynamic visualisation of our data. However, also here the implementation was not easy especially as we received no input through the lectures or the teaching assistants on how to implement this framework. Once AngularJS was running we implemented a Servlet in order to be able to receive async http-requests on our server-side. The implementation hereof was rather easy just as the implementation of a Websocket to send the bike objects from the database to the UI. Eventually, we figured out how to put all these frameworks together and went on to distribute our systems over multiple virtual machines (VMs). This lead to tremendous difficulties as we had no clue how to use Docker together with Vagrant together with Consul together with a Registrator together with HAProxy. At the point when we hand in this report we still have not figure out how to exactly put these pieces together.

In general the course lead to connect many different frameworks and technologies together, but gave us almost no input on how to do so especially as the lectures were not helpful at all.