Bidding System

Web & Cloud Computing

Ruben Kip & Hatim Alsayahani

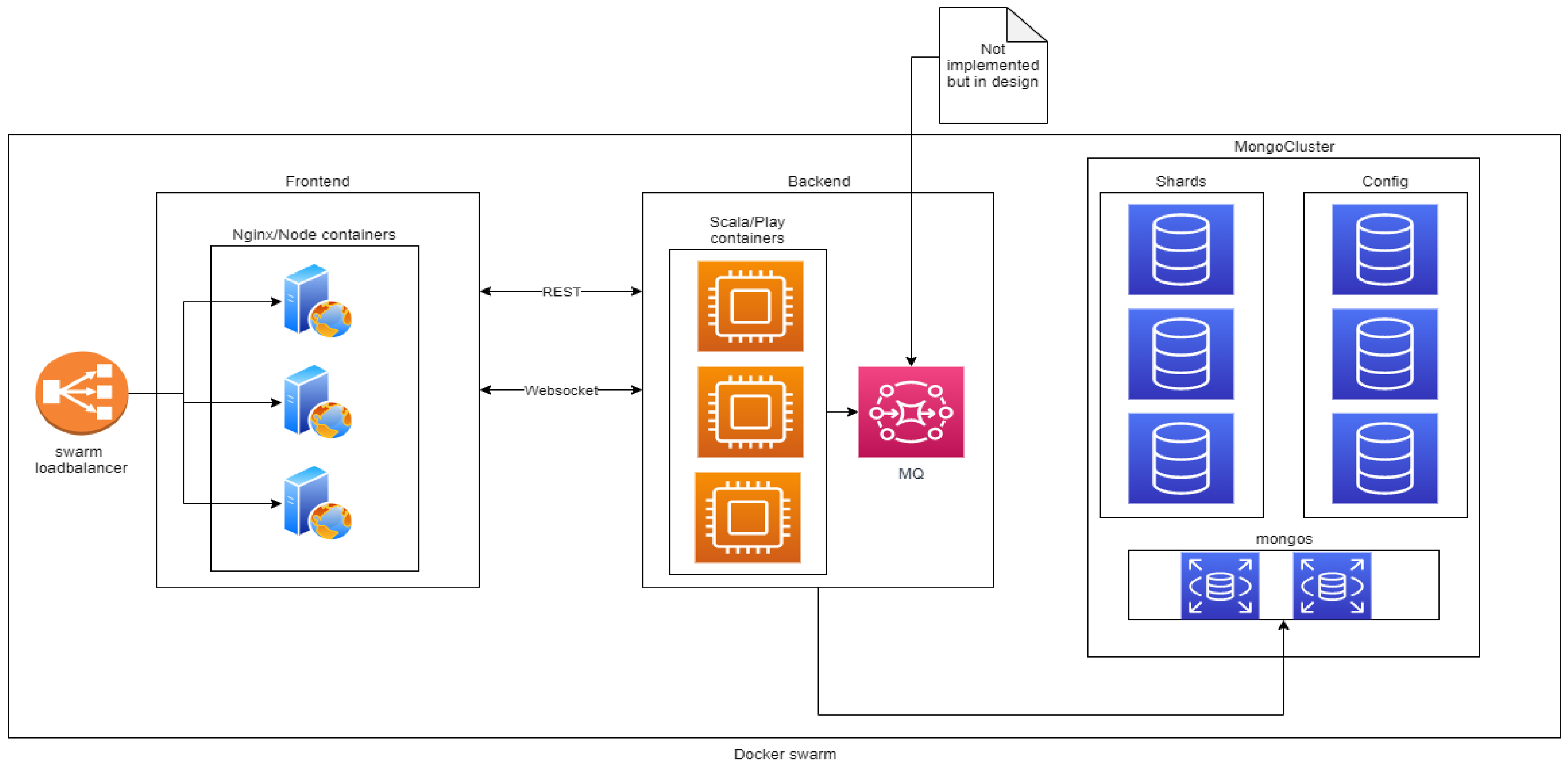
Introduction

The Bidding system project is an auction system for items. Items can be created, which will then a show on the webpage, allowing users to create bids on specific products.
In the following sections we will discuss the architecture of our implementation, and then we will further move onto discussing our choices for design and our technology stack for the front-end, back-end, the database, message queues, websockets and the fault tolerant mechanisms.

Architecture

We use a 3 tier architecture of the system see figure 1. There are 3 replicas of the Frontend running. The frontend is sending all requests to itself, which are then proxied by Nginx to the backend. Docker swarm handles loadbalancing of the frontend servers.
There are 3 replicas of the Backend application, these handle incoming requests (API/Websocket) from frontend and handle any application logic.
Connected to the backend is a MongoDB cluster. This cluster consists of a replica set of 3 shards. A replica set of 3 config instances. And 2 mongos instances for accessing the cluster.

Side note: Although not fully implemented the original design also has a MQ and Cassandra DB. Originally we wanted to have a MQ to share status of created bids over the backend replicas. The initialized source of bids for a newly connected user should come from the Cassandra DB because this is good for time series as it is high write efficient.


Not implemented but in design
Frontend
Nginx/Node containers
swarm loadbalancer
REST
Websocket
Backend
Scala/Play containers
MQ
MongoCluster
Shards
Config
mongos
Docker swarm

Technology stack + Reasoning

Front-end

Each replica is a **Nginx** webserver with **Node js** for Angular. We chose **Angular** as it supports the SPA requirement very well and is easy to understand for beginners. We choose Nginx because it is high performance webserver to host the Angular on application on but we also use it because we can hide the backend more by proxying requests to the backend. We chose to use regular **websockets** because the part of our application which shows bids requires instant updates when other people create bids. Because we did not want to start from scratch some styling is custom made but most of it is from **Bootstrap**.

Backend

Each replica uses the **Scala** because it supports async requests very well with Futures and promises. The **Play framework** is then a logical choice to use as it is a very popular Scala web framework, by us used to handle incoming requests.

There are 3 backend replicas for fault tolerance, for now all 3 replicas act the same so the other replicas can continue the work. But if in the future we require a master/slave structure backend the number 3 overs us a chance to ensure failure and implement a voting between the remainder instances for new master.

MongoDB

We choice **mongoDB** as it offers a very easy way to create new products without a scheme, which then can be easily migrated in the future. Furthemore we also chose it as it is also fault tolerant. We sharded our MongoDB so the database can handle bigger amounts of data.
The mongoDB replica set has 3 shards and 3 config servers so MongoDB can figure out which instance has failed with the other 2 instances if there is a failure. Furthermore for sharding we also need more then 2 instances else sharding would not make sense as your trying to grow vertically. The Mongos instances is just there to access the cluster so only needs 2 instances as both do the same job anyway so in case of failure we have a spare so not to create a bottleneck.

Scalability and Fault tolerance

For containerization we use **Docker**, it is very popular and very well documented.
Because i did not have a credit card we did not choose for a cloud service, and without a cloud service setting up Kubernetes would have cost too much time. This is why we use **Docker swarm** for orchestration, ran on 2 local VM's. A plus is that Docker swarm also handles accessing and loadbalancing of the frontend replicas. A con is that Docker swarm orchestrates for less containers then Kubernetes, so if in the future the system grows this might become an issue. However for the current system this will be far enough.