CMPE 281

Project Group 7

Final Project Report

Instructor: Dr. Jerry Gao

Team Members:

Rishindra Reddy D.

Varun Mahendra Shah

Mandar Gade

Contents

Abstract

Mobile Sensor Cloud Infrastructure as a Service is a long term which does not make sense the first time someone reads it but when you break it up into its components: Mobile Sensor, Cloud and Infrastructure as a Service it helps to understand the concept. In this project our goal was to provide Mobile Sensors (sensors that are moving while continuously providing a stream of data wirelessly) as a service for hire to users. It can provision and de-provision a mobile sensor to a user 'on-demand'. These user mobile sensors will be shared by all users and these mobiles sensors are emulated in a scalable environment, where new instance is created as and when load increases and instances are deleted when the load decreases.

An important question to ask is why do people need mobile sensor data? It is because in our current environment there is an explosion of IoT (Internet of Things) – where devices of all kinds and categories are connected to the internet to provide additional functionality to them. If someone is looking to make an app which is to utilize IoT, they will have to buy the product, then place sensors on them and set up a wireless network to collect the sensor data. If the product is mobile (on the move) this becomes an extremely expensive affair. This is where we step in, our goal is to provide sensor infrastructure on demand using cloud concept of infrastructure as a service. So user will simply have to sign up, select the sensors they want to use and start getting virtual mobile sensor instance instantly. Once their app is complete, they can add or delete any number of sensors as per their requirement and at any given point of time, they are only billed for the sensors they are currently using i.e. pay-as-you-go model.

Introduction

By using mobile sensor cloud IaaS users can subscribe to virtual sensors and manage them. It also allows users create new sensors as well. Auto scaling feature is provided, as and when demand for a sensor increases, instances are loaded and scaled to match the demand and load balancer evenly distributes the requests to the instances in round robin fashion.

Billing model adopts pay-as-you go model where in a metered component tracks resources used by user and generate a bill according to the usage. The cloud commandments elasticity, load balancing, monitoring, on demand resource acquisition and requisition, pay as you use, multi tenancy and the use of public telephony as an unlimited supply of cheap hardware is what mobile sensor cloud computing is all about.

Outcome of the project was to emulate this cheaply available hardware in mobile devices be it tablets, smartphones or portable communicators - most notably sensors within them as a hire model to customers. Clients are able to hire these sensors from our platform which will support large numbers of users sharing the same resources while also balancing load and managing it equally. It will enable customers to create and monitor sensor data which can be applied to IoT based application development.

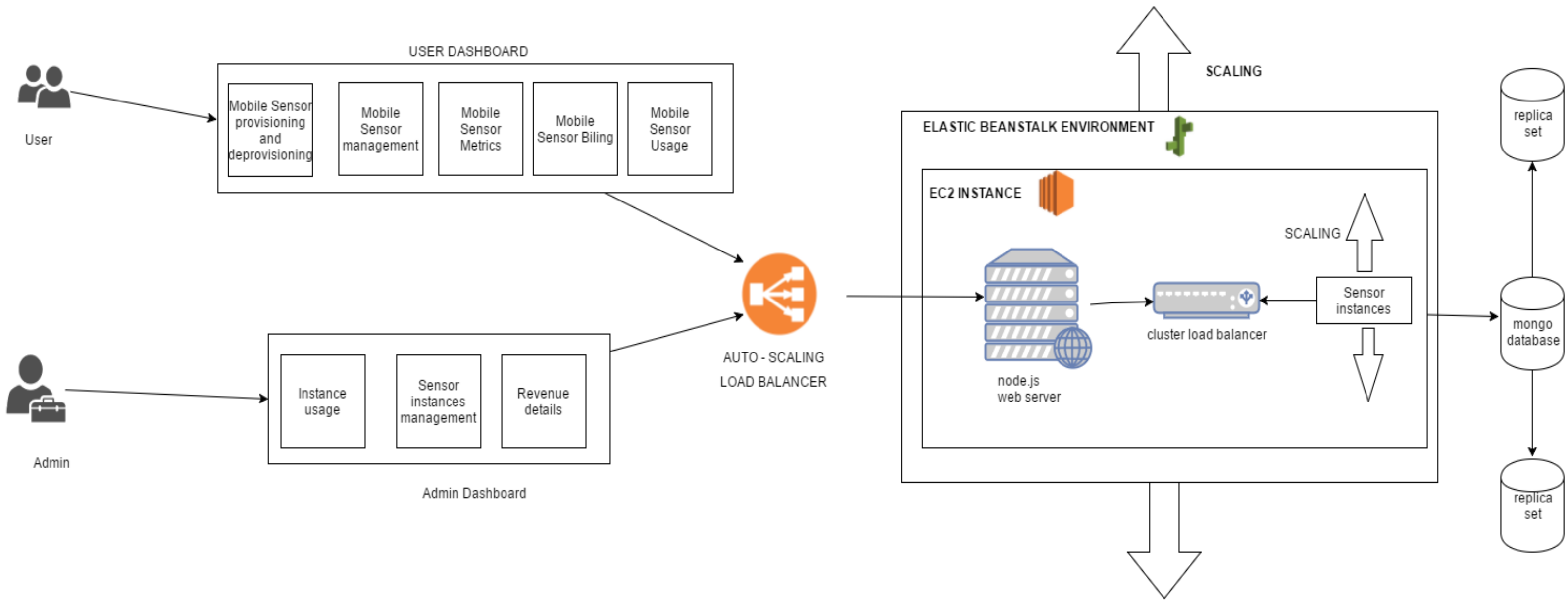
Sensor network cluster creation, emulation, connection and virtualization strategy:

Based on an earlier Survey Report that we made for this course we found out that the two major problems facing the world today are transportation and health. We decided that our infrastructure should help aid research and development aimed at solving these two problems. We created three sensor clusters:

- The Fitbit Cluster consisting of temperature(body), heartbeat, compass(direction) and pulse rate.
- The Car Cluster consisting of compass(direction), speed, engine temperature and current gearshift.
- The Plane Cluster consisting altitude, engine temperature, speed and compass(direction).

Our **assumption** is that we will place these sensor clusters on people/animals – FitBit, Any on road/over water transportation – Car and any aerial transportation – Plane and get sensor data from these clusters. We followed a strategy based on these assumptions where we created a JavaScript that virtualizes and generates sensor cluster at a randomized location which we are then provisioning to users each time they hire a new sensor cluster.

Architecture Diagram



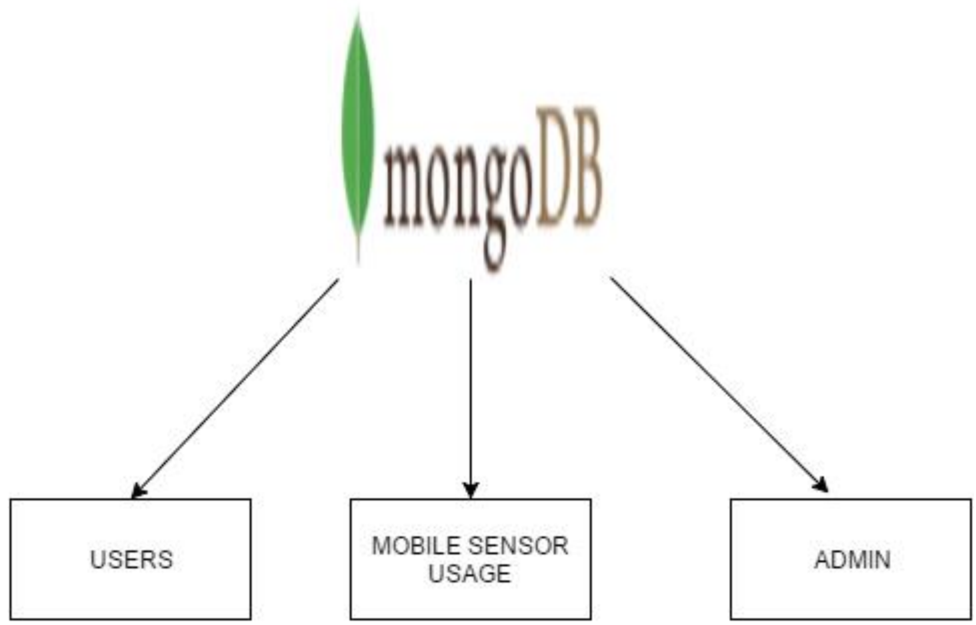
Application we developed was designed to support multi – tenant architecture. Our servers engage with different type of users concurrently. We implemented database replication where multiple servers access a single database. We have configured replication of master database into two replicas of current database for recovery reasons.

We have implemented two different dashboards, one for users and one for application admin. So different users can interact differently with server and manage the mobile sensor instances.

External load balancer is elastic load balancer that we have configured so that if CPU utilization of instance If increases 80 % then the instance is scaled and load balanced. Internal load balancer is cluster load balancer that scales the mobile sensor instances based on number of users accessing the sensor instances.

Internal load balancer scales the instance vertically and external load balancer scales the instance horizontally.

Database design



Collections in database

There are three different collections users, mobile sensor usage and admin collection. Users is a collection of user details. Mobile sensor usage is a collection of user usage details. Admin collection is collection of admin details.


```
{
    "_id": {
        "$oid": "5840b019ae005da03acca5a5"
    },
    "name": "rishi2",
    "email": "rishi2@email.com",
    "password": "Rishi@2",
    "due_bill": "47.93805",
    "fitbit": [
        {
            "start_date": "Sun Dec 04 2016 20:03:41 GMT-0800 (Pacific Standard Time)"
        }
    ],
    "car": [
        {
            "start_date": "Sun Dec 04 2016 20:04:49 GMT-0800 (Pacific Standard Time)"
        }
    ],
    "plane": [
        {
            "start_date": "Thu Dec 01 2016 15:20:48 GMT-0800 (Pacific Standard Time)"
        },
        {
            "start_date": "Sun Dec 04 2016 20:03:43 GMT-0800 (Pacific Standard Time)"
        }
    ]
}
```

Sample User document that would be inserted in user collection of database. It consists of current user contact details, authentication details and his sensor subscriptions.


```
{
    "_id": {
        "$oid": "5840ab374e21674c16dec84e"
    },
    "email": "varun@email.com",
    "sensor_name": "fitbit",
    "start_date": "Thu Dec 01 2016 07:00:11 GMT-0800 (Pacific Standard Time)",
    "end_date": "Thu Dec 01 2016 14:59:03 GMT-0800 (Pacific Standard Time)"
}
```

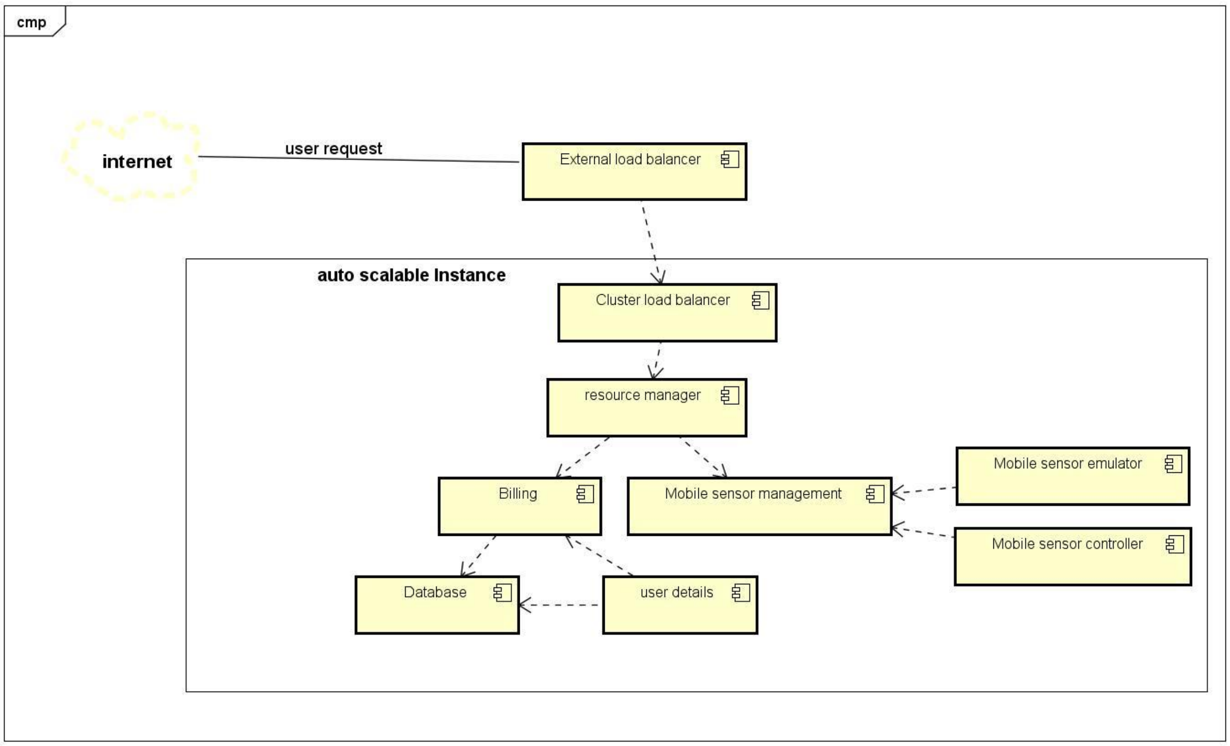

Sample user sensor usage detail document that would be inserted in database. It consists of sensor usage details of mobile sensor. This document is useful to charge the bill as per pay - as – you go terms.

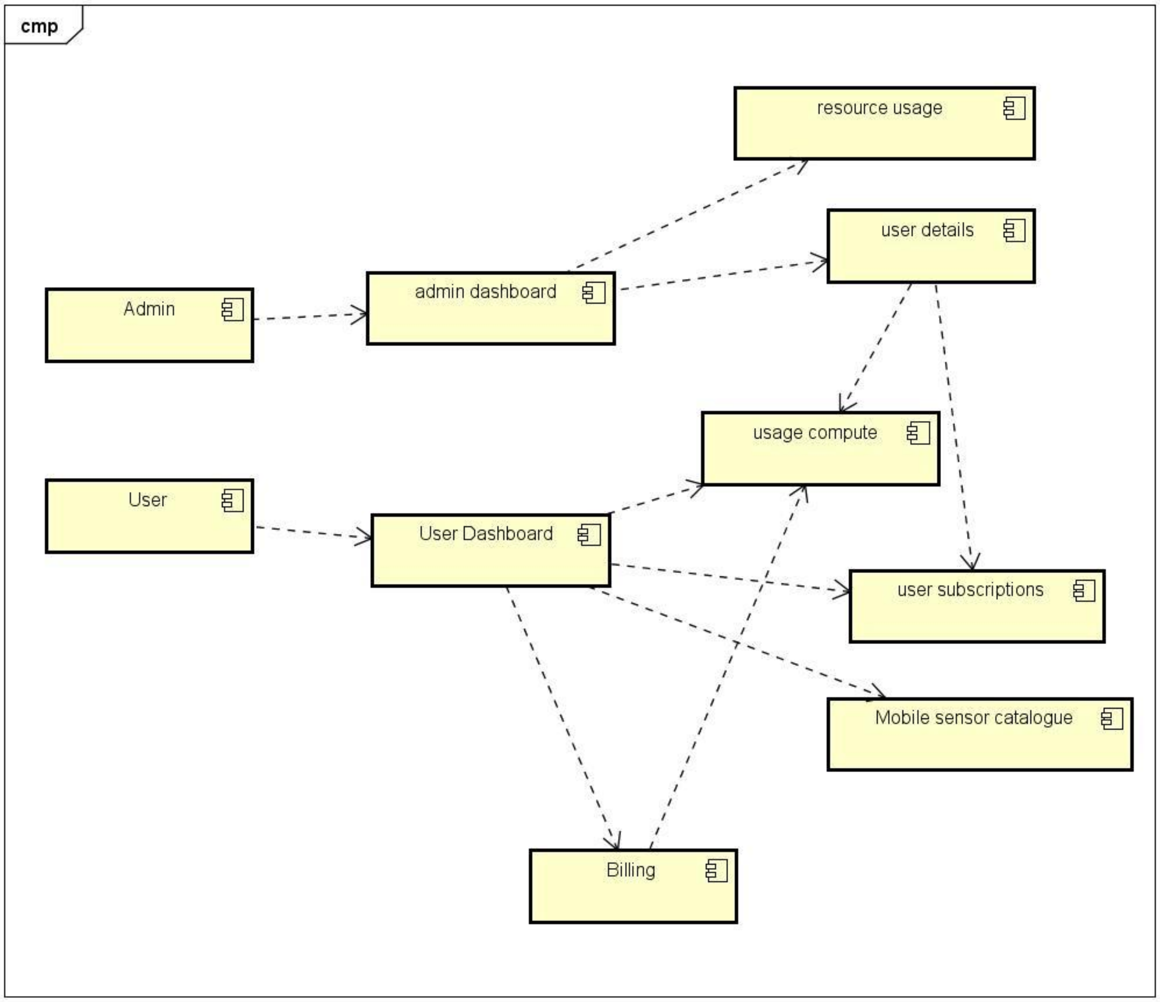

```
{
    "_id": {
        "$oid": "582f9bb1f36d282a957e60c0"
    },
    "email": "admin@miaas.com",
    "password": "admin",



}
```

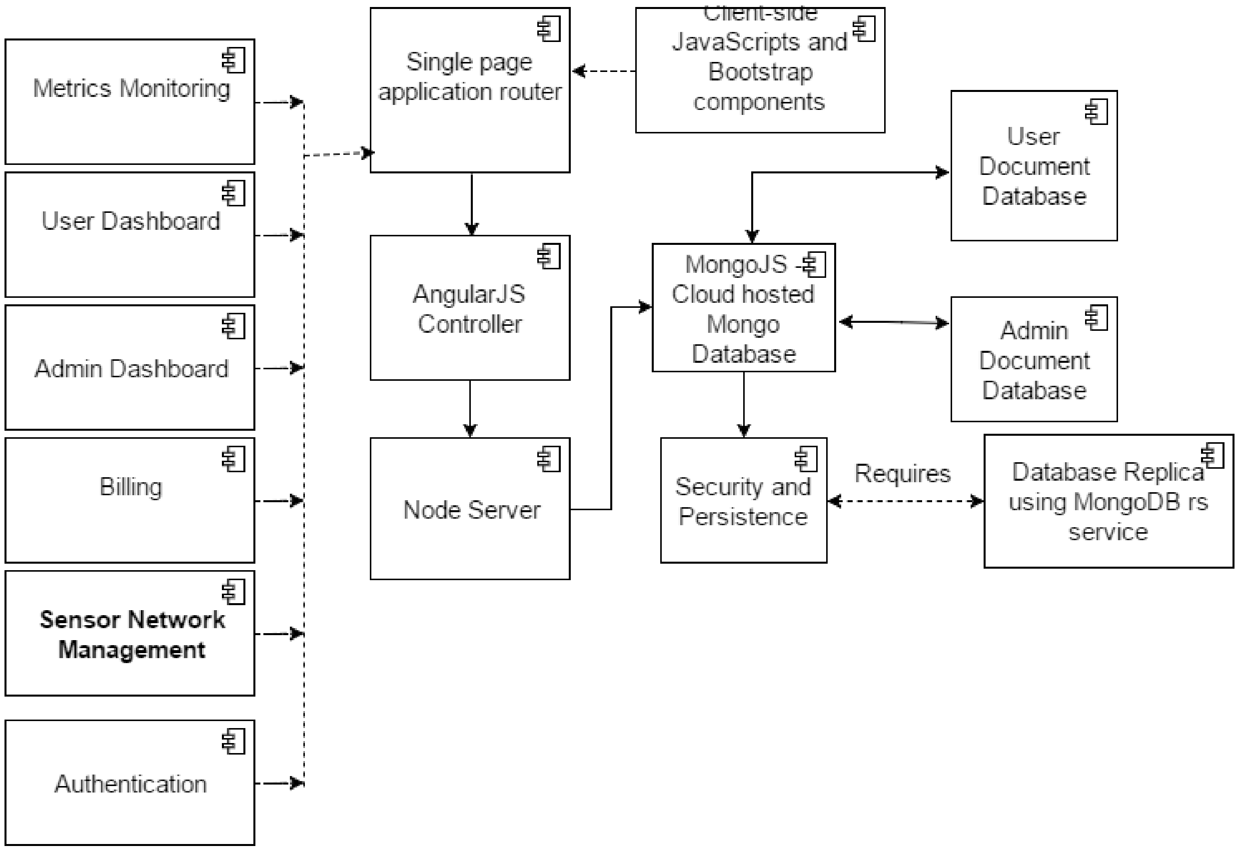

Sample user document of admin details in admin database. It consists of admin authentication and user statistical data.

Other UML Diagrams

Component Diagrams

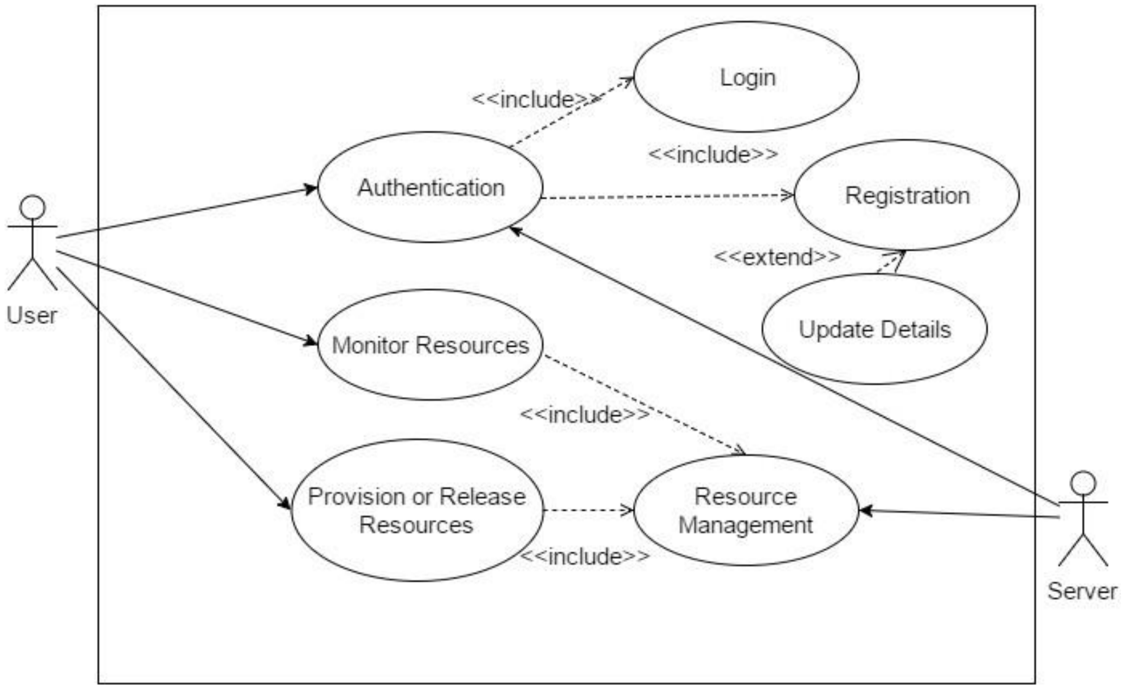



cmp
resource usage
user details
Admin
admin dashboard
usage compute
User
User Dashboard
user subscriptions
Mobile sensor catalogue
Billing

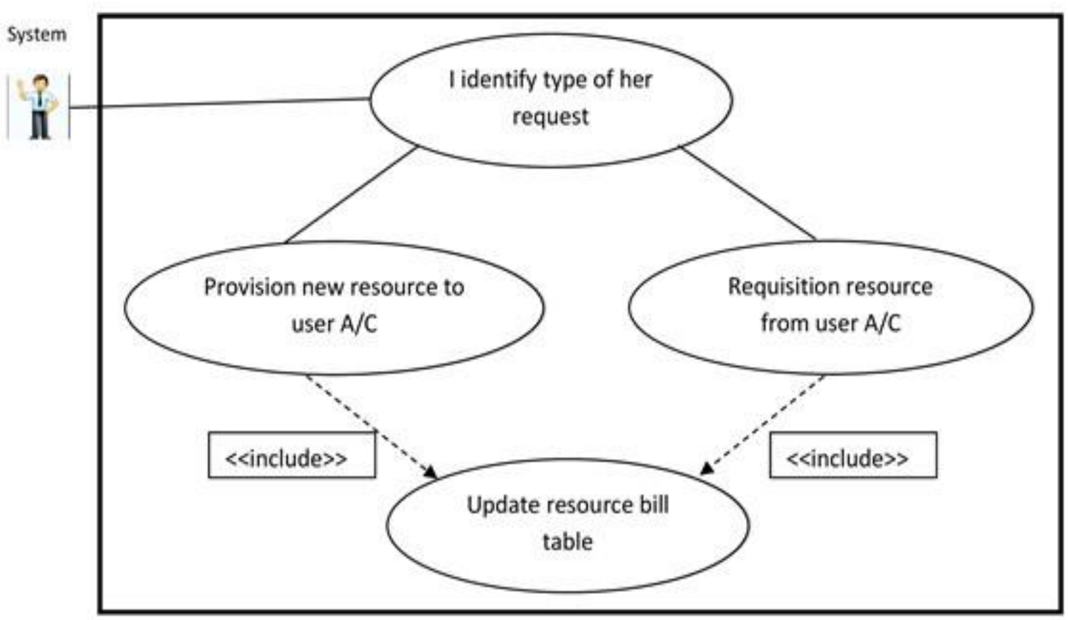


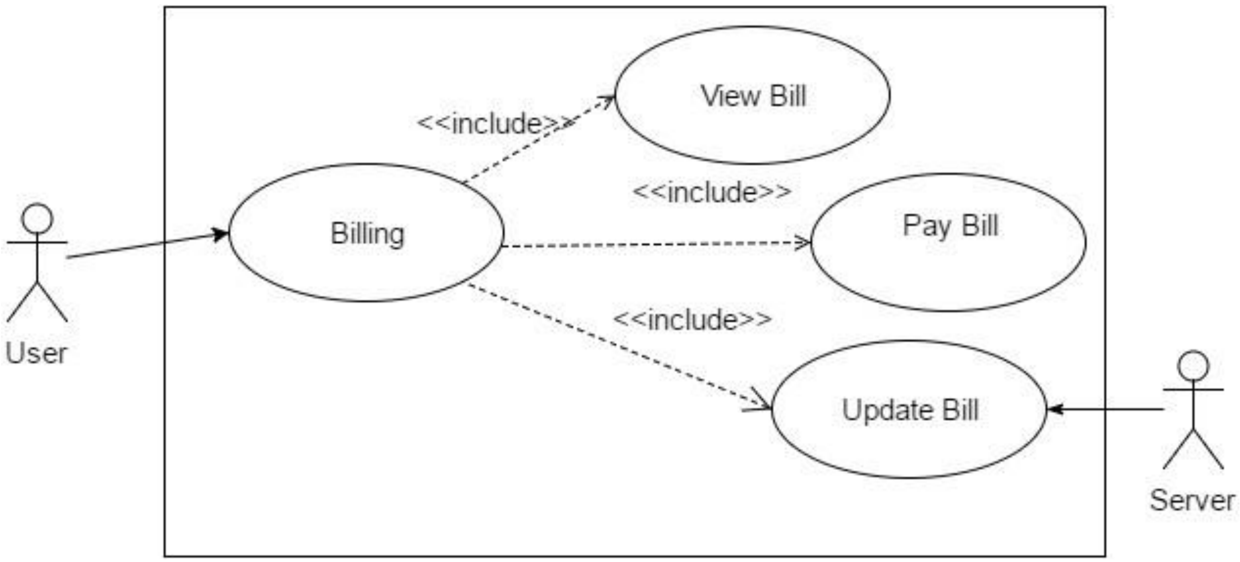
Metrics Monitoring
User Dashboard
Admin Dashboard
Billing
Sensor Network Management
Authentication
Single page application router
Client-side JavaScripts and Bootstrap components
AngularJS Controller
Node Server
MongoJS - Cloud hosted Mongo Database
User Document Database
Admin Document Database
Security and Persistence
Requires
Database Replica using MongoDB rs service

Use Case Diagrams



Resource Management





Sequence Diagram

Sequence diagram too large, can view at:https://goo.gl/xOQYyq

Development

Technology Stack implemented:

- MongoDB
- Express.JS
- AngularJS
- Node.JS
- AWS EC2

Database:

MongoDB NOSQL database selected because all our data operations are giving or receiving JSON data and it is easier to perform CRUD operations on JSON using MongoDB.

Server:

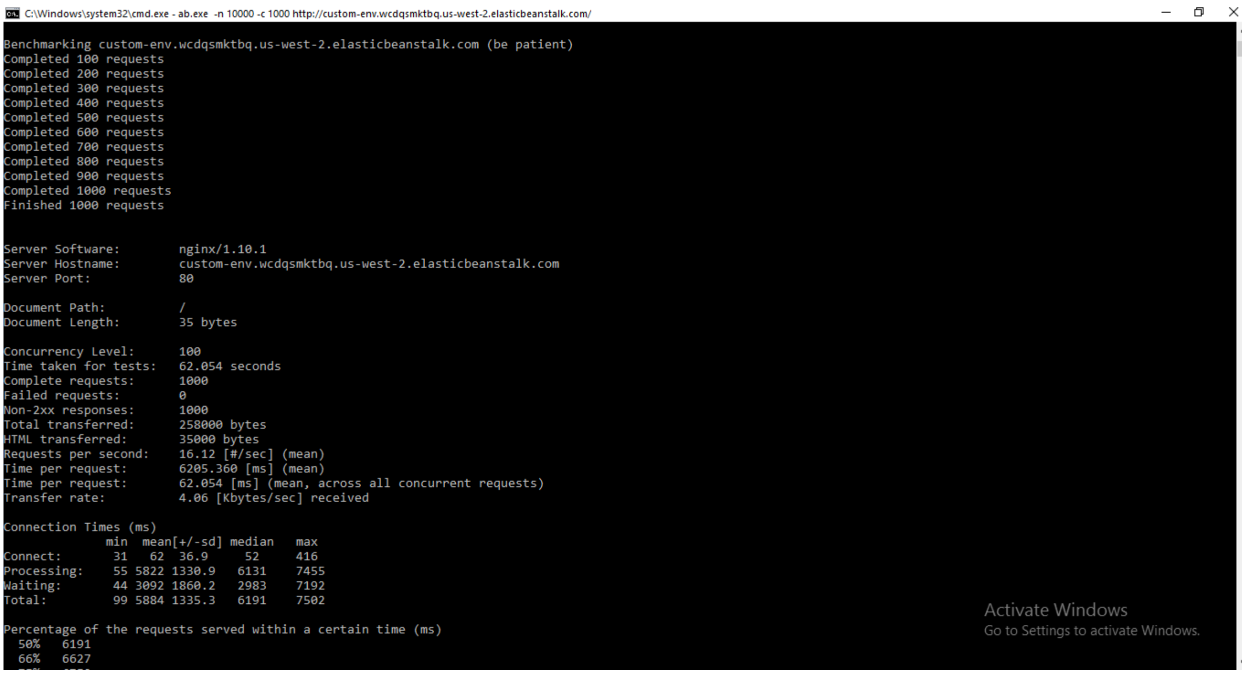
Node.JS Server using Express Framework. Express is the easiest way to handle angular requests and perform operations on a NOSQL database.

Front-End:

Bootstrap Front-End with AngularJS controllers – UI is device independent.

Testing

In testing we have performed the load testing by emulating large number of users to test scalability, availability and performance of the server.



C:\Windows\system32\cmd.exe - ab.exe -n 10000 -c 1000 http://custom-env.wcdqsmktbq.us-west-2.elasticbeanstalk.com/

```
Benchmarking custom-env.wcdqsmktbq.us-west-2.elasticbeanstalk.com (be patient)
Completed 100 requests
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Completed 600 requests
Completed 700 requests
Completed 800 requests
Completed 900 requests
Completed 1000 requests
Finished 1000 requests


Server Software:        nginx/1.10.1
Server Hostname:        custom-env.wcdqsmktbq.us-west-2.elasticbeanstalk.com
Server Port:            80

Document Path:          /
Document Length:        35 bytes

Concurrency Level:      100
Time taken for tests:   62.054 seconds
Complete requests:      1000
Failed requests:        0
Non-2xx responses:      1000
Total transferred:      258000 bytes
HTML transferred:       35000 bytes
Requests per second:    16.12 [#/sec] (mean)
Time per request:       6205.360 [ms] (mean)
Time per request:       62.054 [ms] (mean, across all concurrent requests)
Transfer rate:          4.06 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       31   62  36.9     52     416
Processing:    55 5822 1330.9   6131    7455
Waiting:       44 3092 1860.2   2983    7192
Total:         99 5884 1335.3   6191    7502

Percentage of the requests served within a certain time (ms)
  50%   6191
  66%   6627
```

Activate Windows
Go to Settings to activate Windows.

In above test we ran 1000 requests at 100 concurrency level against our application in the cloud and we observed the performance.

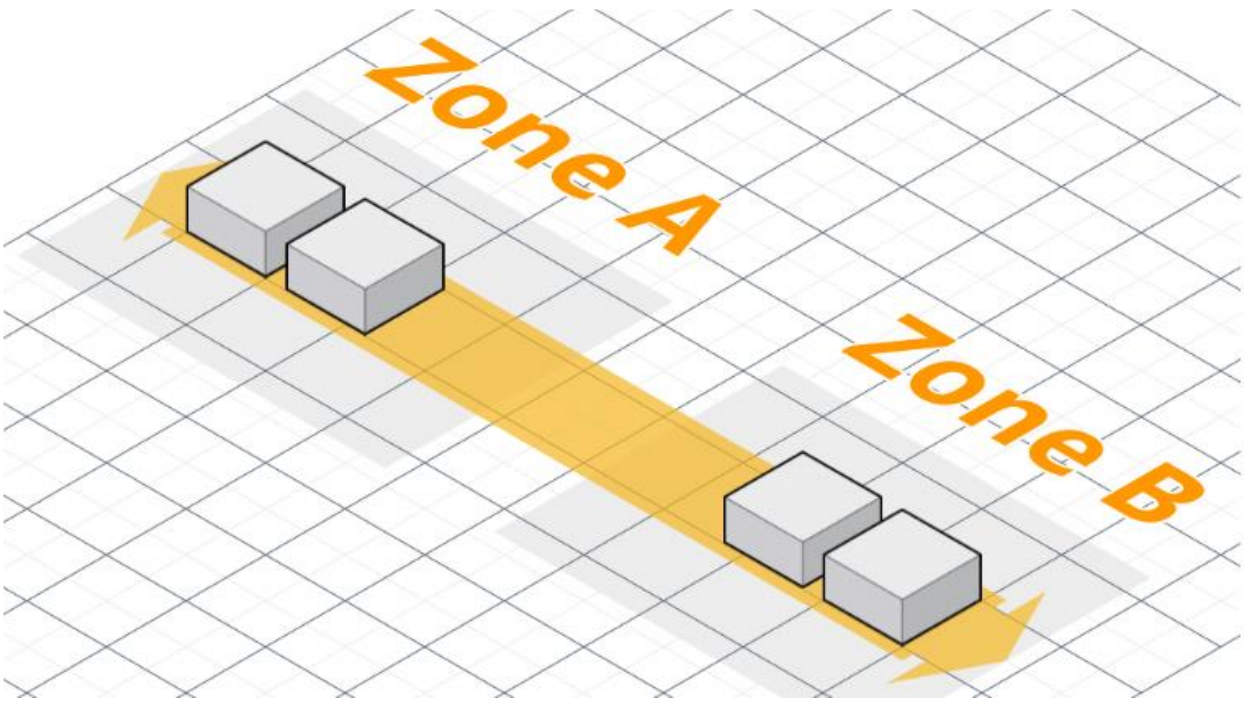
Deployment and Cloud Configuration:

We deployed our application on Amazon AWS elastic beanstalk in Node.js web environment. We configured auto scaling load balancer using AWS elastic load balancer, trigger used for scaling the instances is CPU utilization metric when raises over 80% then new instance was been created.

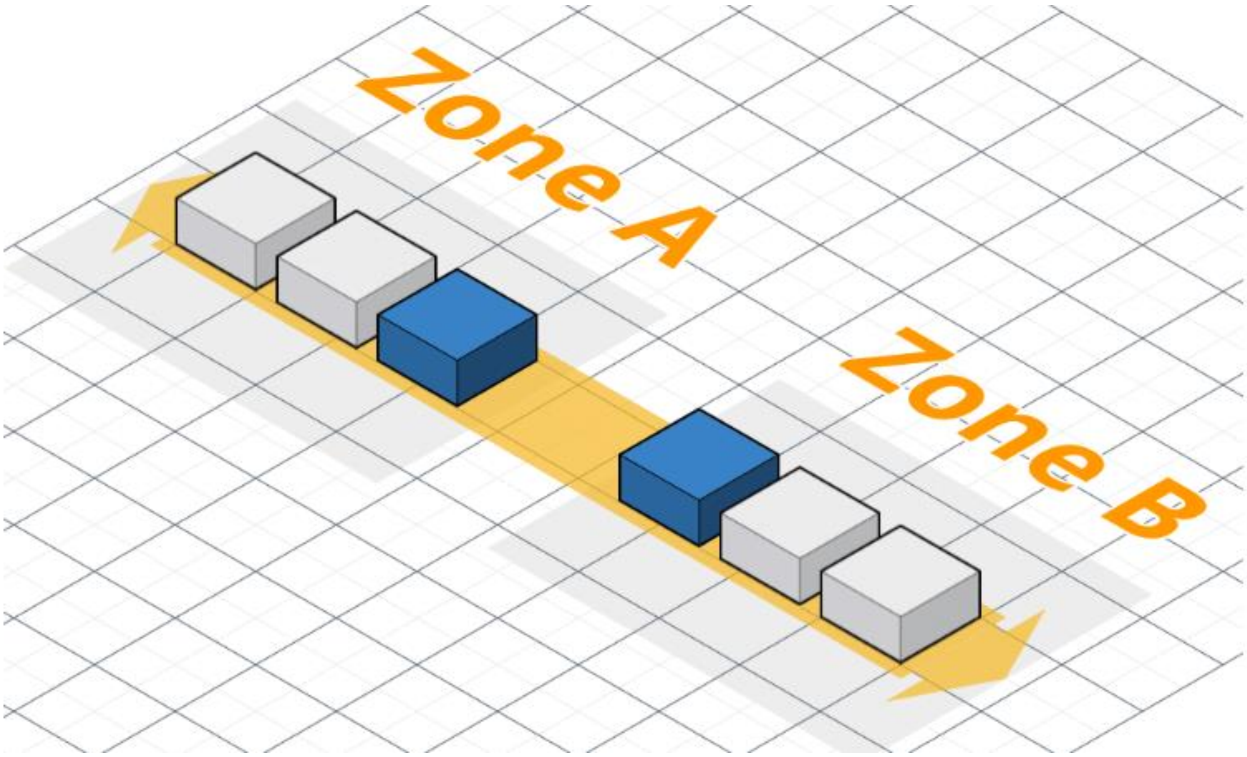
We created zip folder of our application and deployed it on AWS beanstalk and before deploying the application we configured the auto - scaling load balancer on the metric of CPU utilization.

We configured the rolling updates as the procedure for deploying version updates on running instances in a rolling update fashion. This setting ensures that our application has zero – down time even when we are deploying updates on running application concurrently.

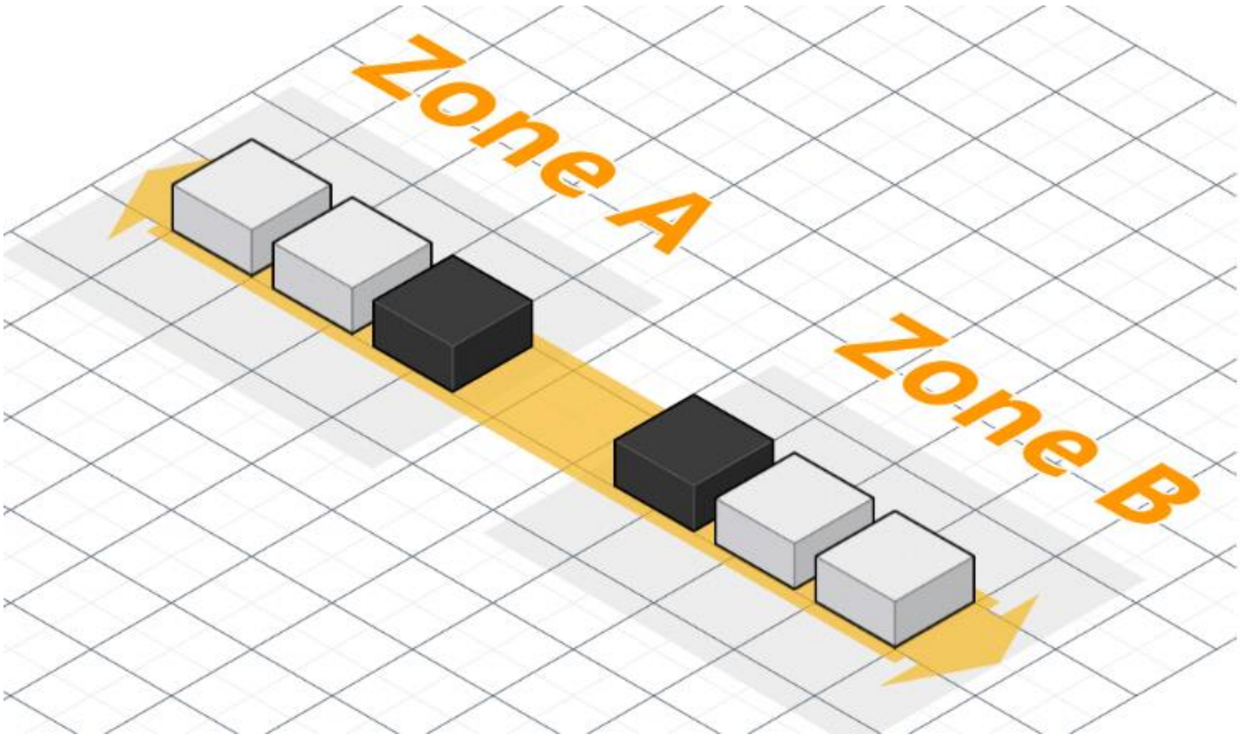
Rolling updates:



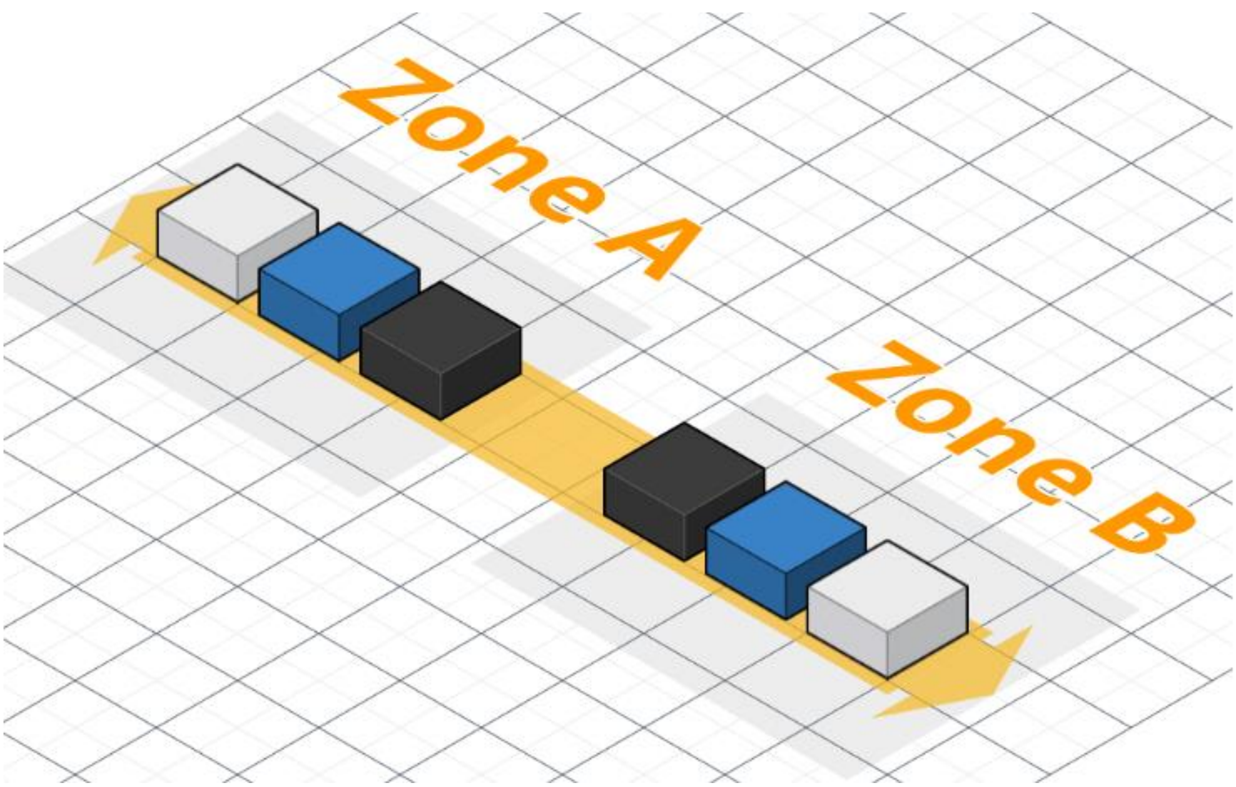
This is our application running on the cloud, we are currently running version 1 of the application concurrently in 4 instances.



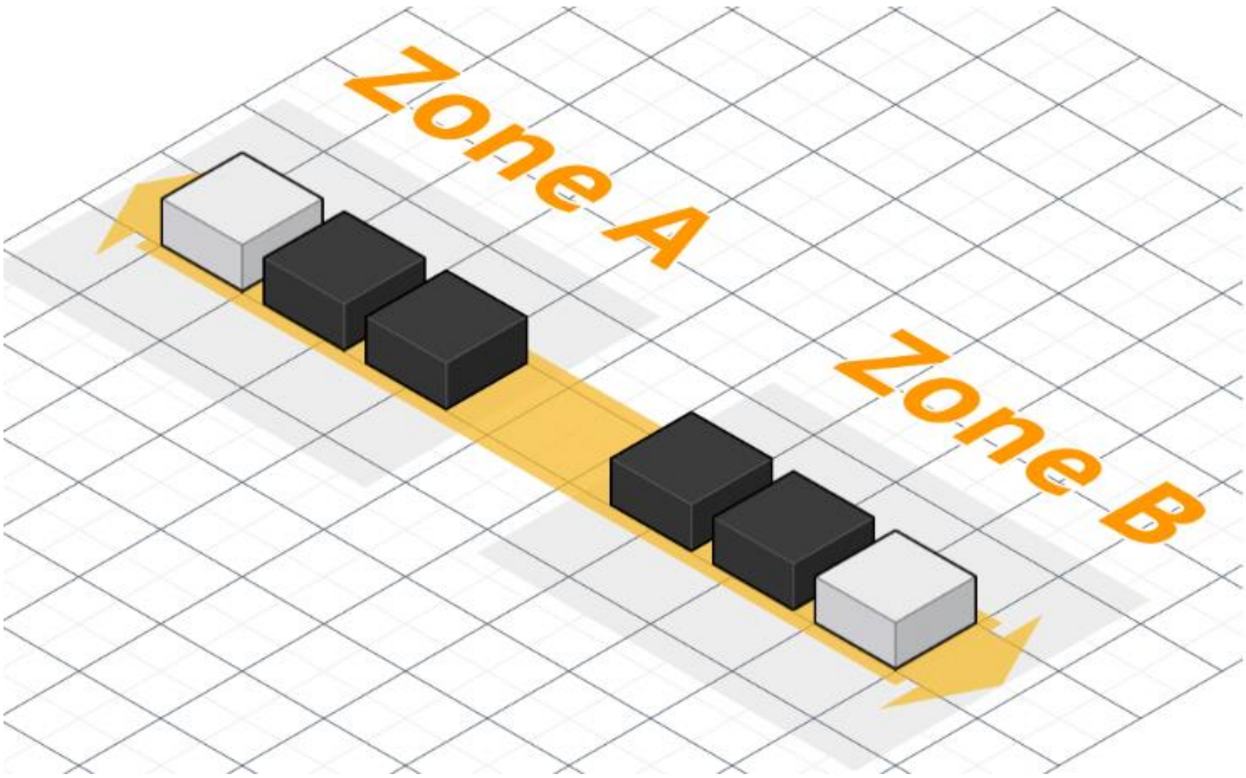
Suppose say we have new version 2 to deploy into the cloud. so now if we follow rolling update method, It launches new instance with version 2 which is currently in update process.



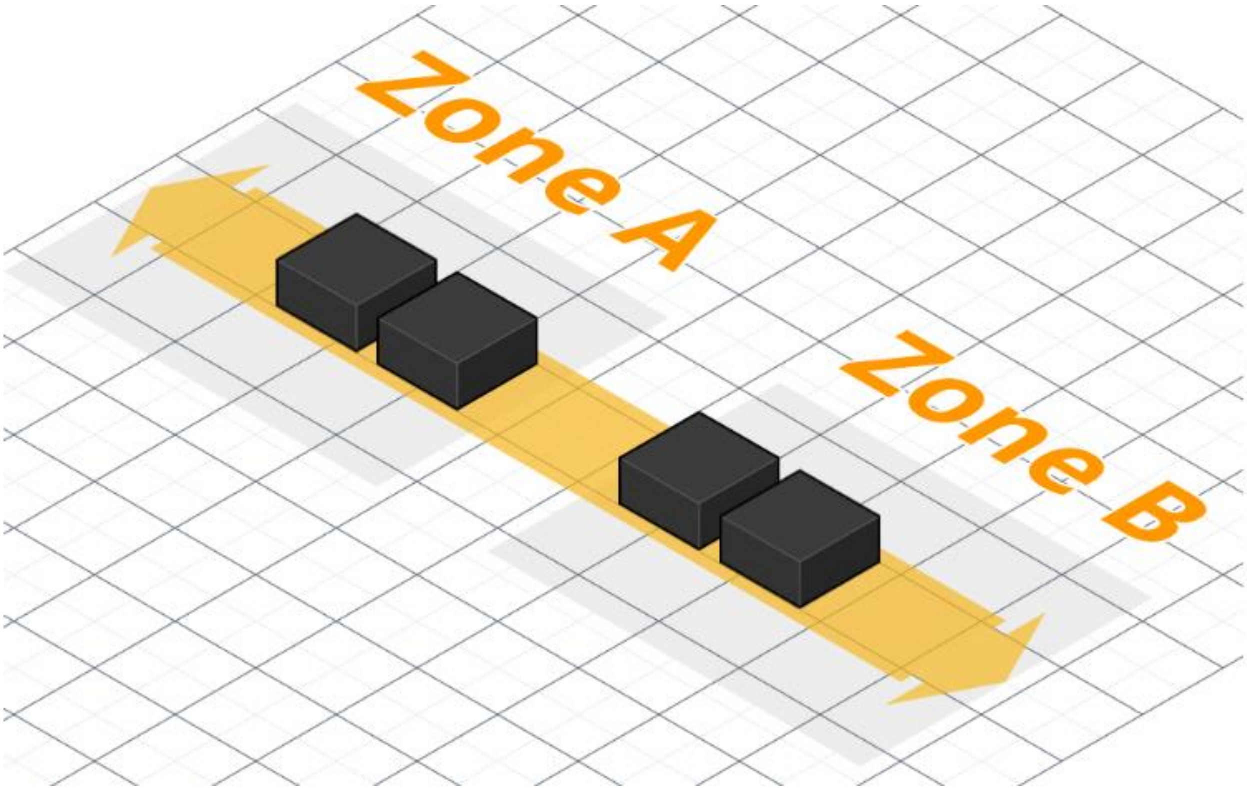
Once the update is completed the new version of our update is running on new instances. Now at this moment we have six instances with 4 of version 1 and 2 of version 2.



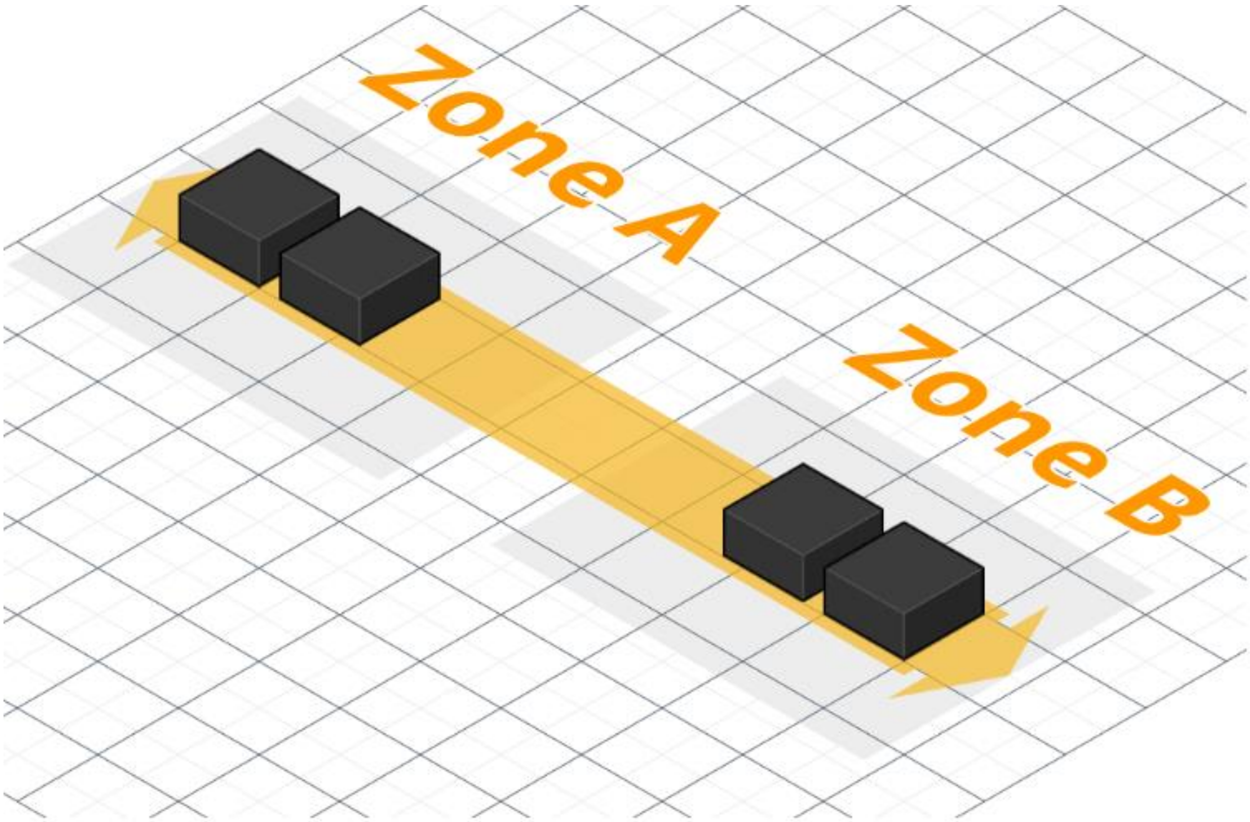
Now the running instances of version 1 are updated to version 2 one by one. So now two instances of version 1 are currently updated to version 2 and are in process.



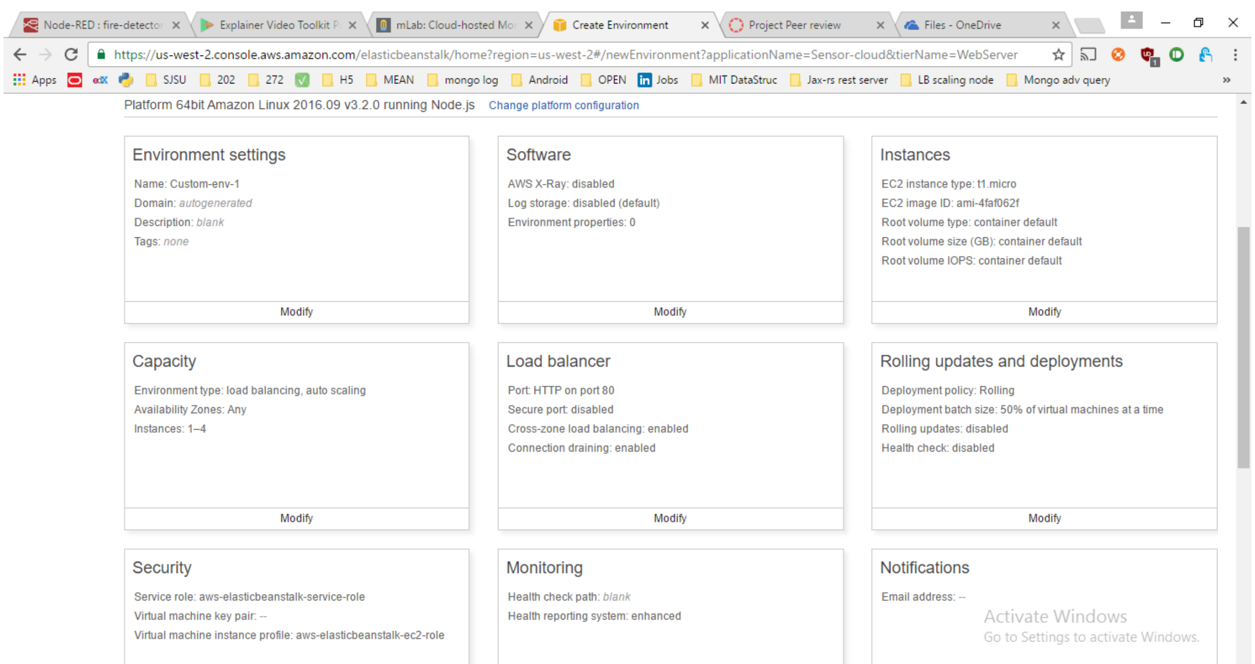
Now we have 4 instances of version 2 and two instances of version 1, which are running concurrently.



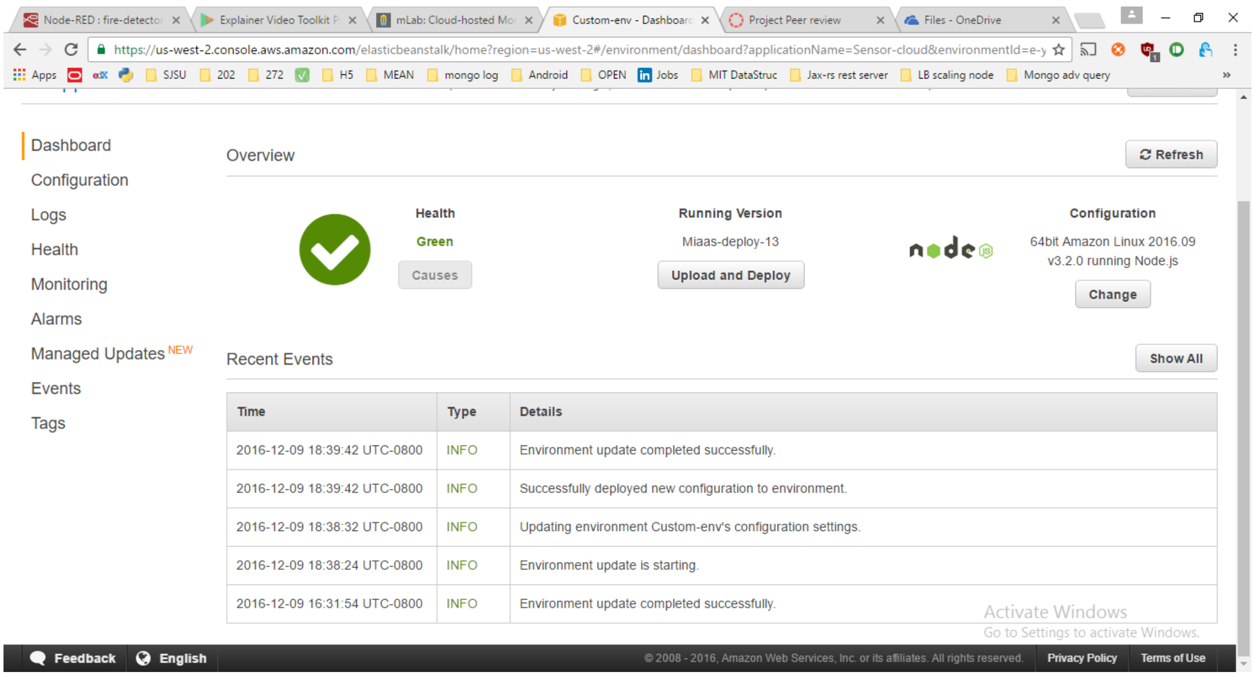
Two instances of older version are removed and now we have all our running instances with version 2 and zero downtime.



While deploying configurations are made in rolling updates and load balancing along with the capacity of the number of instances limit.



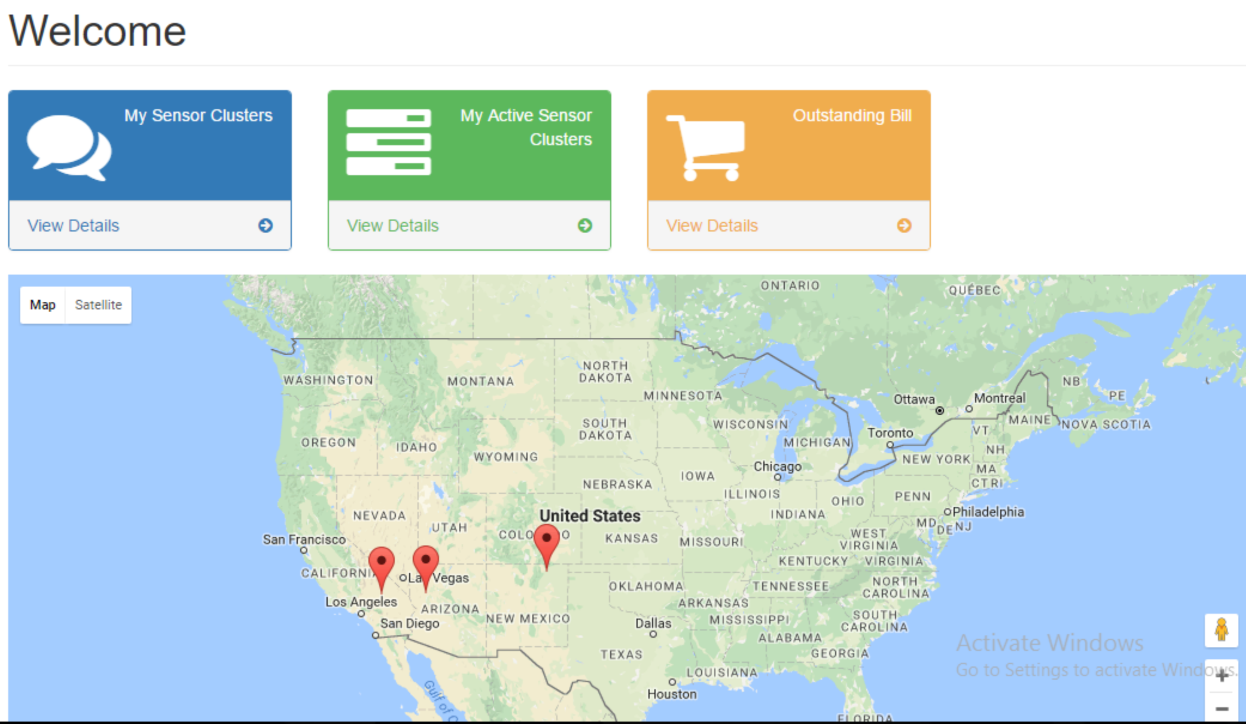
Now after successfully deploying your application into AWS beanstalk with proper configurations, the dashboard consists of following information



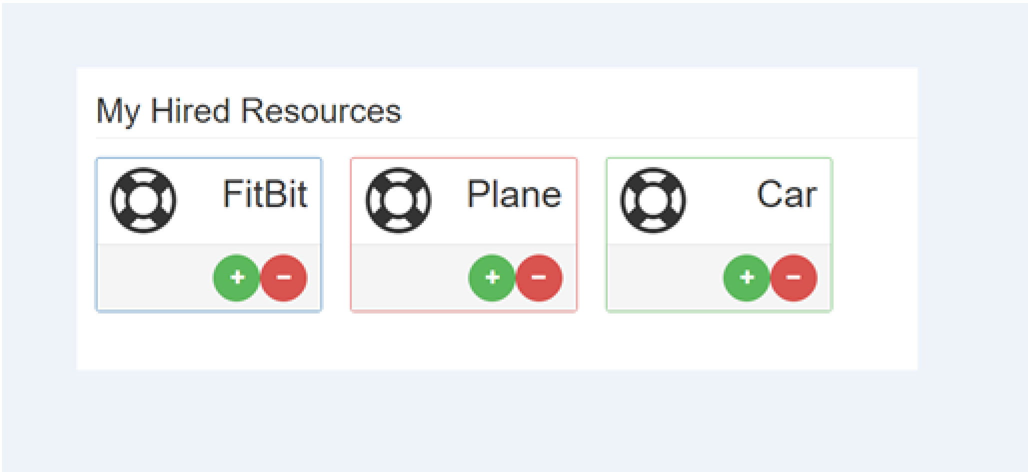
Implementation Screenshots:

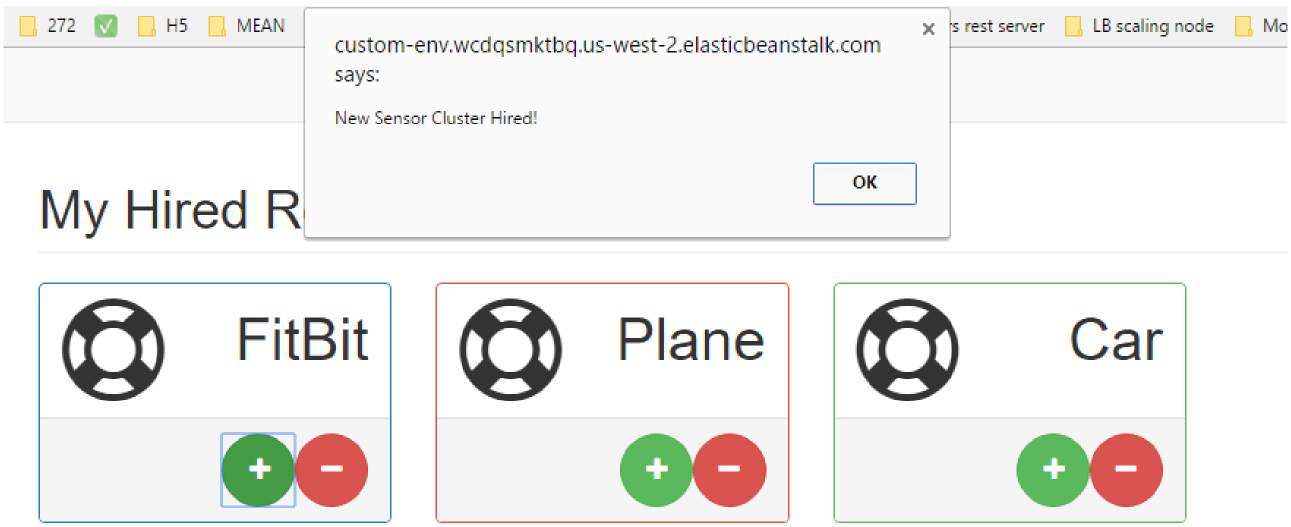
Our Landing Page. It has two login buttons, the ‘LOGIN’ in upper right corner for User Login and the ‘C’ on its right hand side for admin login. We have deliberately chosen to make the admin login page hidden by making the button small and difficult to notice.

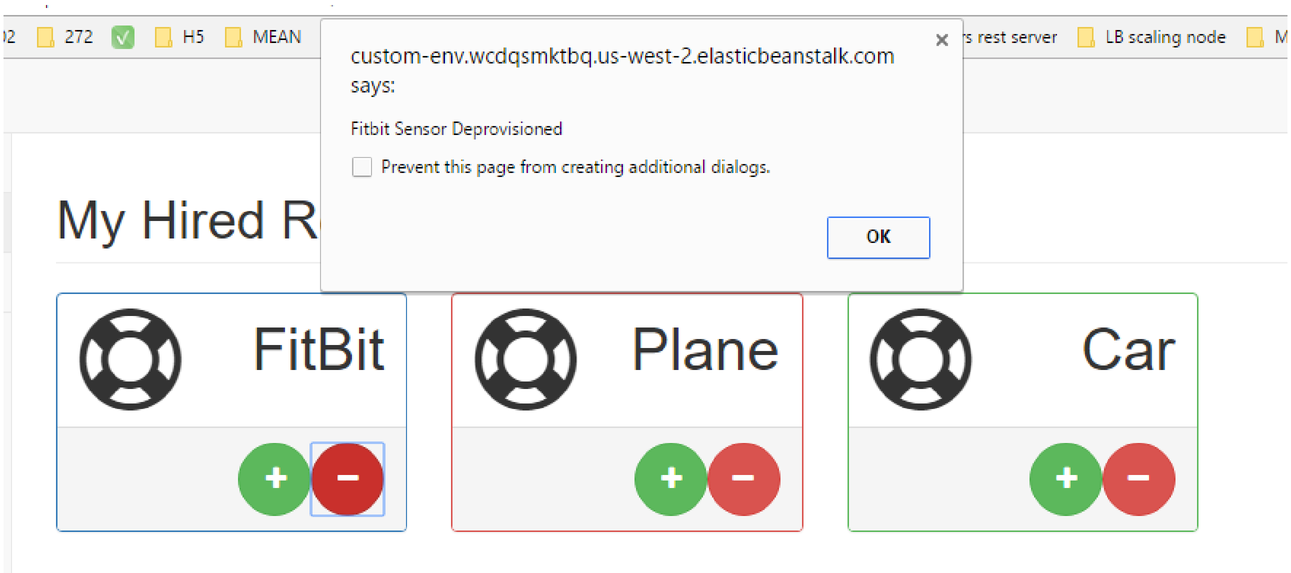
The User Dashboard from where a user can control all the aspects of his account details, pay his bills, view sensor cluster metrics and hire or de provision new sensors. On the map you can see the locations of the current virtual sensor clusters user has hired.



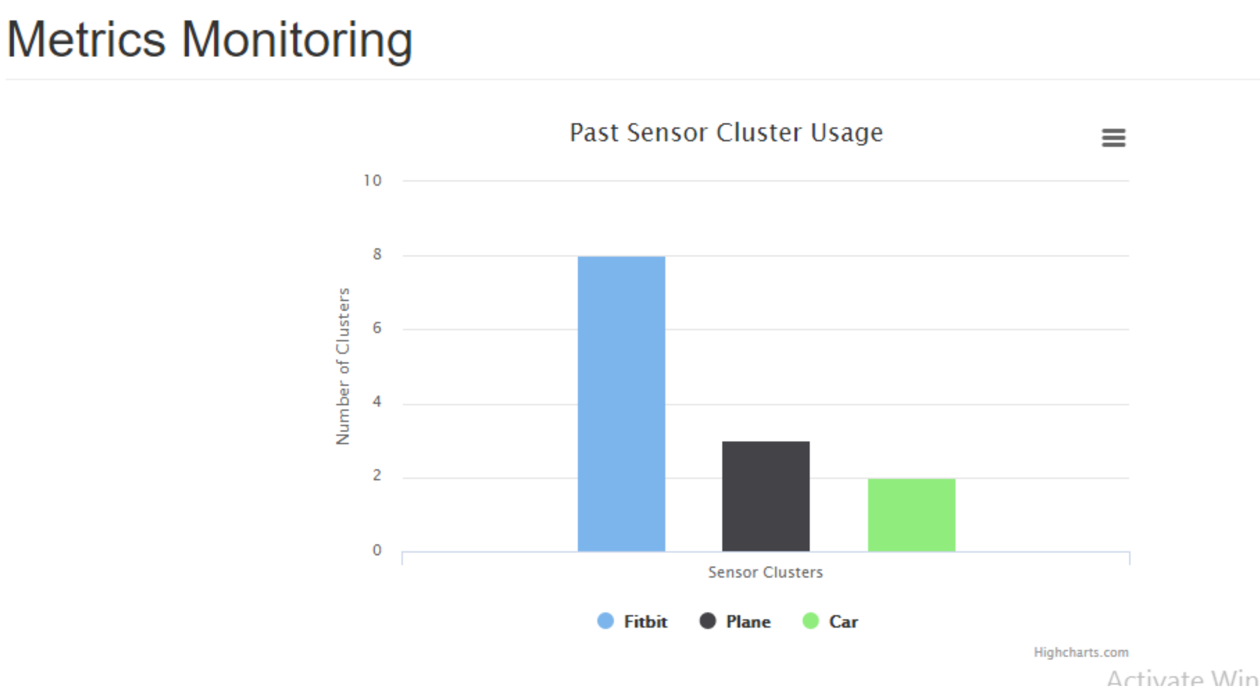
Below screenshots display how users can add and delete sensors:

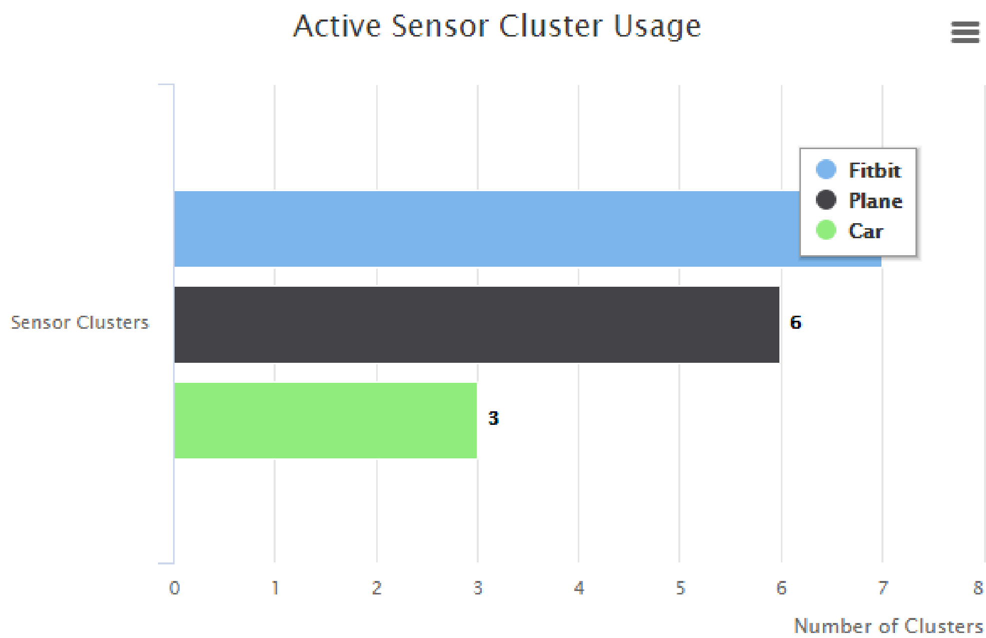


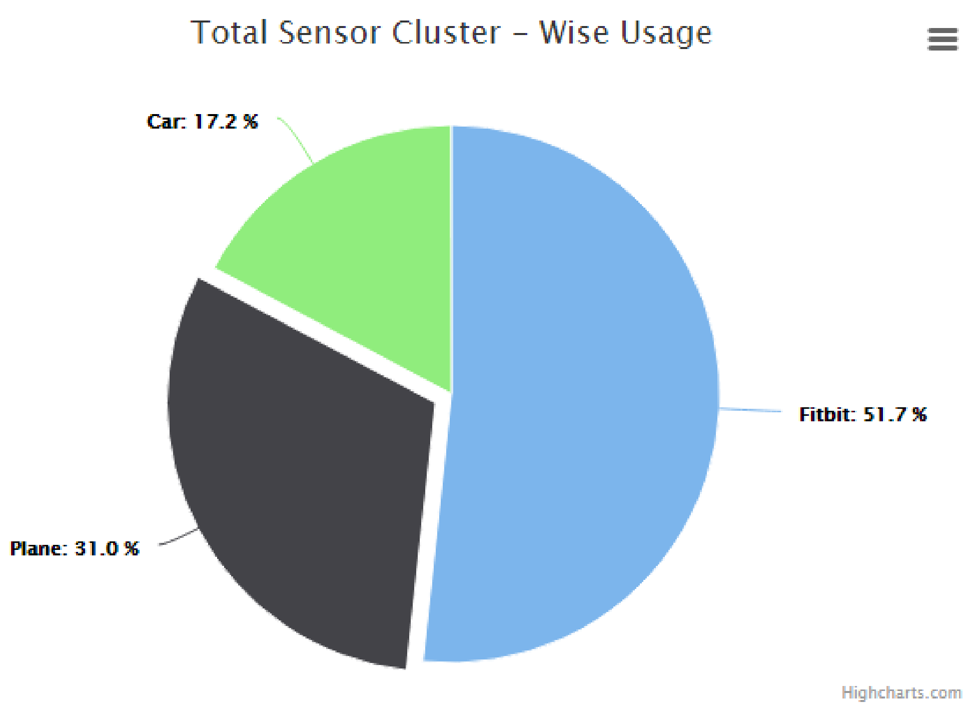




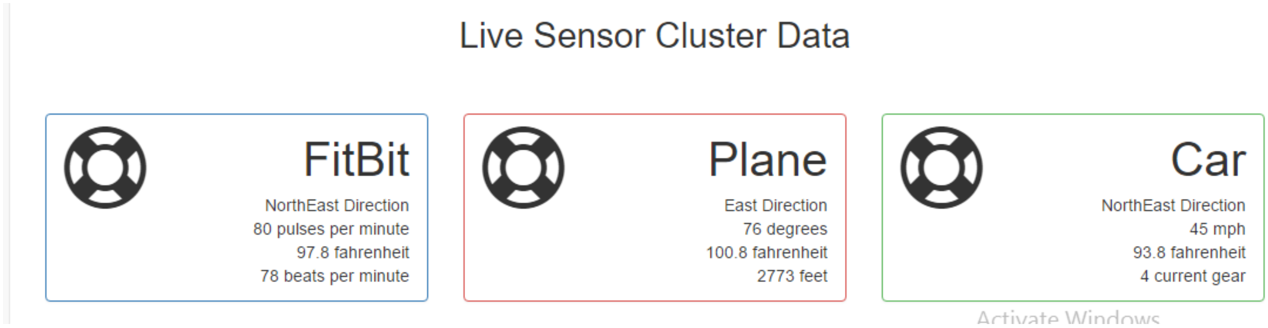
Now we show the metrics page showing the various metrics a user has access to



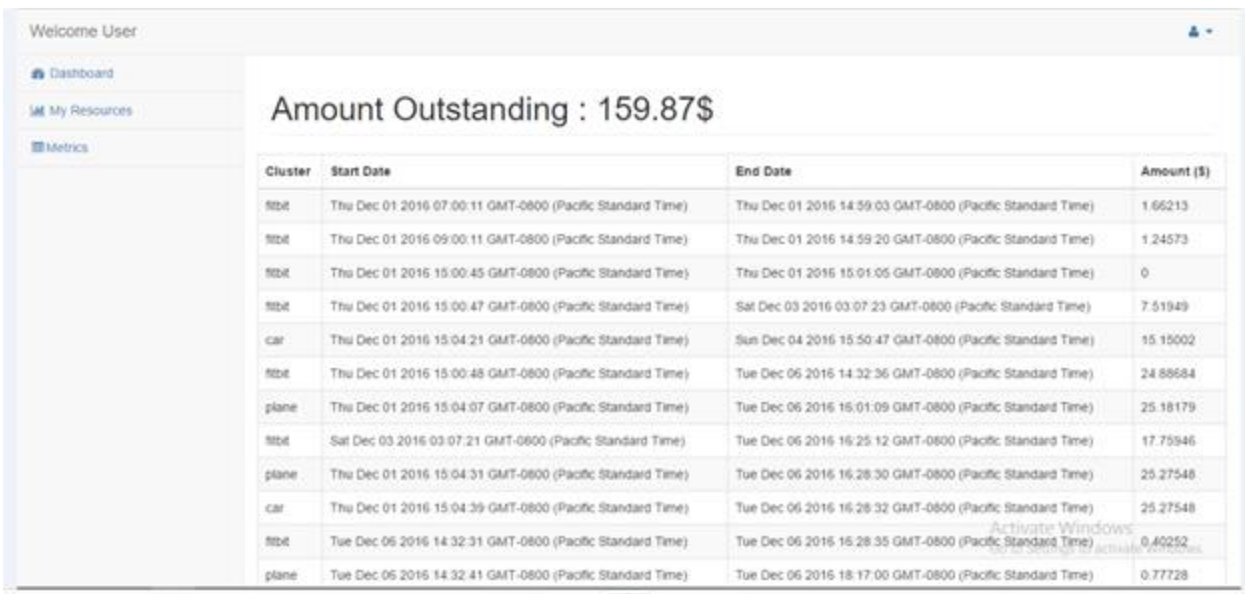

Active Sensor Cluster Usage
Fitbit
Plane
Car
Sensor Clusters
6
3
0
1
2
3
4
5
6
7
8
Number of Clusters


Total Sensor Cluster - Wise Usage
Car: 17.2 %
Fitbit: 51.7 %
Plane: 31.0 %
Highcharts.com

Sensor data is updating at 5 second intervals as shown below.



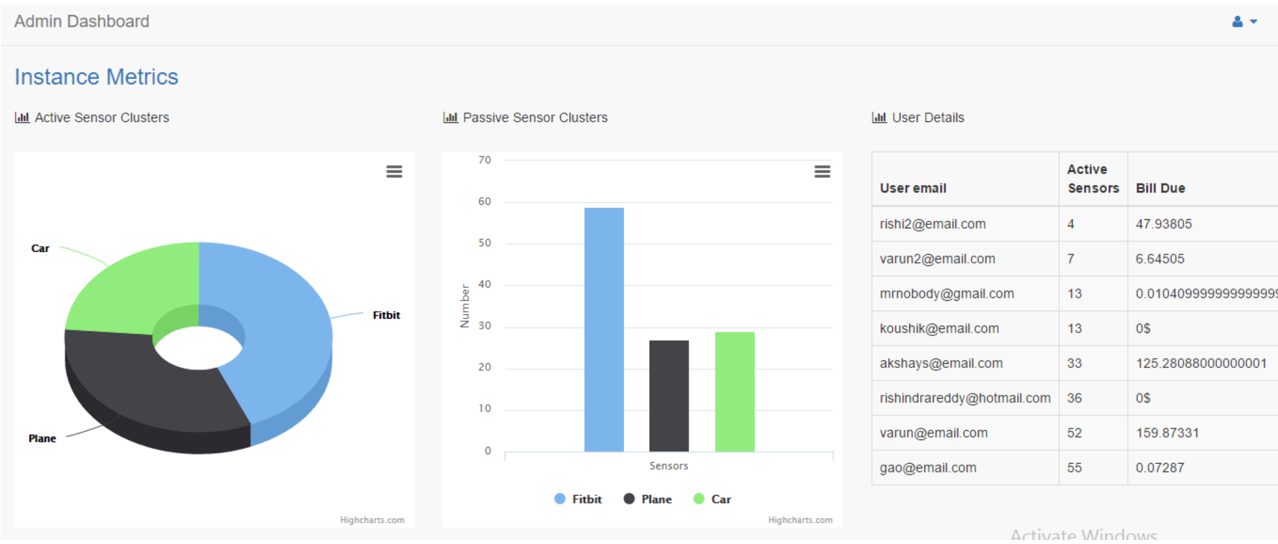
The user bill page

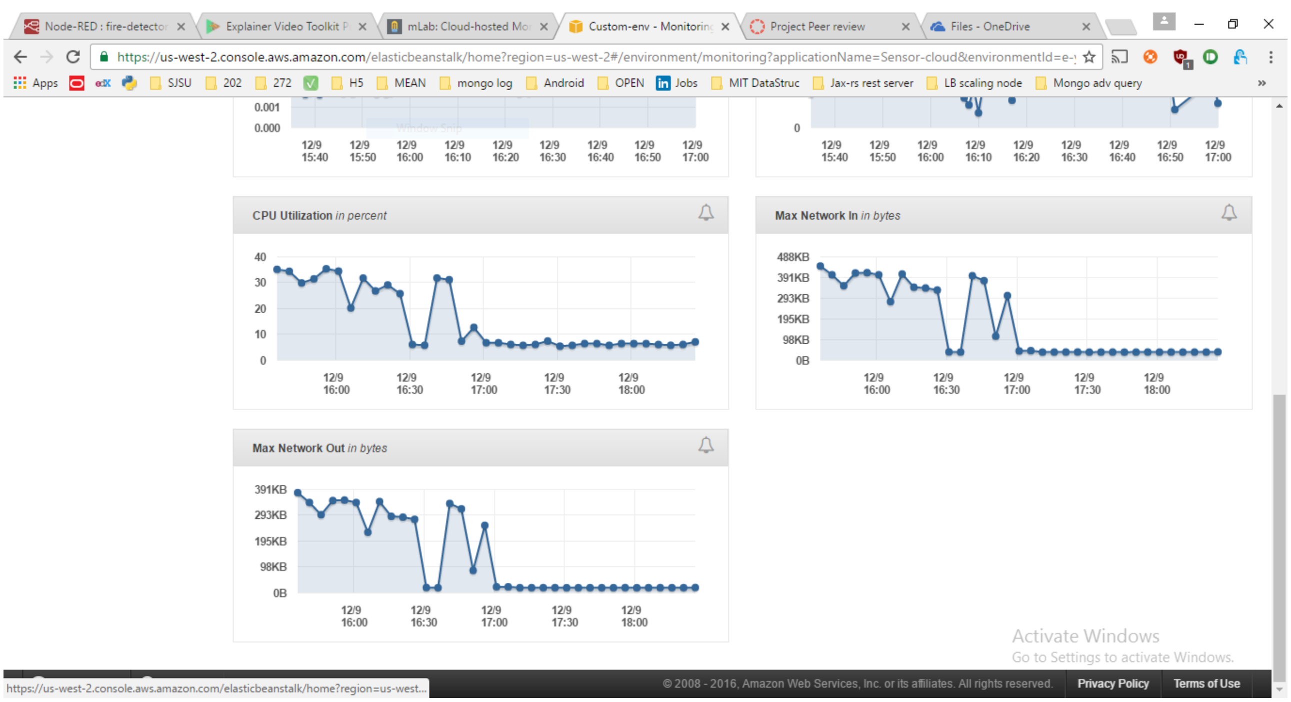


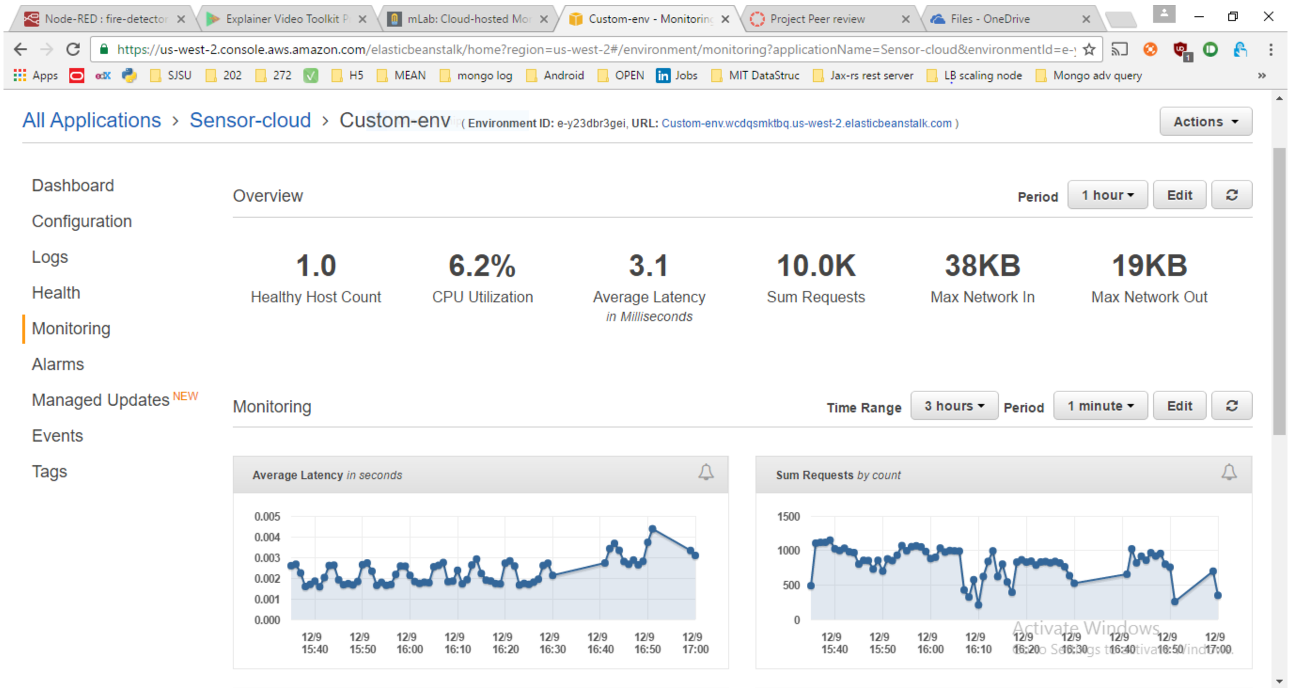
Cluster	Start Date	End Date	Amount ($)
fitbit	Thu Dec 01 2016 07:00:11 GMT-0800 (Pacific Standard Time)	Thu Dec 01 2016 14:59:03 GMT-0800 (Pacific Standard Time)	1.66213
fitbit	Thu Dec 01 2016 09:00:11 GMT-0800 (Pacific Standard Time)	Thu Dec 01 2016 14:59:20 GMT-0800 (Pacific Standard Time)	1.24573
fitbit	Thu Dec 01 2016 15:00:45 GMT-0800 (Pacific Standard Time)	Thu Dec 01 2016 15:01:05 GMT-0800 (Pacific Standard Time)	0
fitbit	Thu Dec 01 2016 15:00:47 GMT-0800 (Pacific Standard Time)	Sat Dec 03 2016 03:07:23 GMT-0800 (Pacific Standard Time)	7.51949
car	Thu Dec 01 2016 15:04:21 GMT-0800 (Pacific Standard Time)	Sun Dec 04 2016 15:50:47 GMT-0800 (Pacific Standard Time)	15.15002
fitbit	Thu Dec 01 2016 15:00:48 GMT-0800 (Pacific Standard Time)	Tue Dec 06 2016 14:32:36 GMT-0800 (Pacific Standard Time)	24.88684
plane	Thu Dec 01 2016 15:04:07 GMT-0800 (Pacific Standard Time)	Tue Dec 06 2016 16:01:09 GMT-0800 (Pacific Standard Time)	25.18179
fitbit	Sat Dec 03 2016 03:07:21 GMT-0800 (Pacific Standard Time)	Tue Dec 06 2016 16:25:12 GMT-0800 (Pacific Standard Time)	17.75946
plane	Thu Dec 01 2016 15:04:31 GMT-0800 (Pacific Standard Time)	Tue Dec 06 2016 16:28:30 GMT-0800 (Pacific Standard Time)	25.27548
car	Thu Dec 01 2016 15:04:39 GMT-0800 (Pacific Standard Time)	Tue Dec 06 2016 16:28:32 GMT-0800 (Pacific Standard Time)	25.27548
fitbit	Tue Dec 06 2016 14:32:31 GMT-0800 (Pacific Standard Time)	Tue Dec 06 2016 16:28:35 GMT-0800 (Pacific Standard Time)	0.40252
plane	Tue Dec 06 2016 14:32:41 GMT-0800 (Pacific Standard Time)	Tue Dec 06 2016 18:17:00 GMT-0800 (Pacific Standard Time)	0.77728

Now we display the admin dashboard

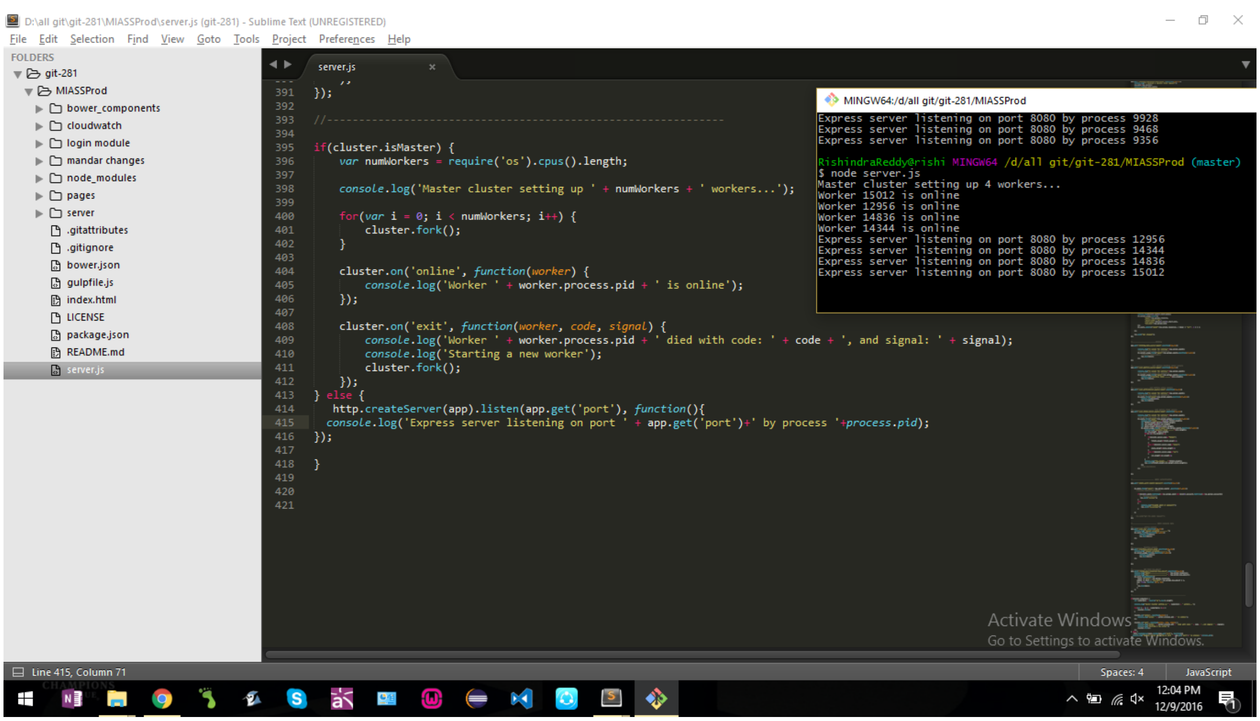
From here the admin can view different metrics as shown in the screen shots and also start/stop the services of users.







Below you can see screenshot of our internal load balancer. We have built a cluster load balancer in front of our mobile sensor instances which are emulated inside the server. This load balancer always maintains high availability of mobile sensor instances. In clusters there are always child processes that work to ensure that requests of the users are resolved quickly by forking themselves and as and when demand for resource is less these worker processes are moved to rest state.



JavaScripts for Virtualizing Sensor Cluster Data:

```
$(function() {

    var cdir = ['North','East','West','South','NorthEast','NorthWest','SouthEast','SouthWest'];
    var CDData = cdir[Math.floor(Math.random() * cdir.length)];
    var CSData = Math.floor(Math.random() * (10 - 120 + 1)) + 120;
    var CTData = Math.floor(Math.random() * (100.8 - 91.8 + 1)) + 91.8;
    var CGData = Math.floor(Math.random() * (1 - 6 + 1)) + 6;

    document.getElementById("CarDirection").innerHTML = CDData + ' Direction';
    document.getElementById("CarSpeed").innerHTML = CSData + ' mph';
    document.getElementById("CarTemp").innerHTML = CTData + ' fahrenheit';
    document.getElementById("CarGear").innerHTML = CGData + ' current gear';
});
```

```
$(function() {

    var dir = ['North','East','West','South','NorthEast','NorthWest','SouthEast','SouthWest'];
    var CData = dir[Math.floor(Math.random() * dir.length)];
    var PData = Math.floor(Math.random() * (100 - 50 + 1)) + 50;
    var TData = Math.floor(Math.random() * (100.8 - 91.8 + 1)) + 91.8;
    var HData = Math.floor(Math.random() * (60.00 - 100.00 + 1)) + 100.00;

    document.getElementById("Compass").innerHTML = CData + ' Direction';
    document.getElementById("Pulse").innerHTML = PData + ' pulses per minute';
    document.getElementById("Temp").innerHTML = TData + ' fahrenheit';
    document.getElementById("Beat").innerHTML = HData + ' beats per minute';
});
```

```
$(function() {

    var pdir = ['North','East','West','South','NorthEast','NorthWest','SouthEast','SouthWest'];
    var PDData = pdir[Math.floor(Math.random() * pdir.length)];
    var PAData = Math.floor(Math.random() * (100 - 50 + 1)) + 50;
    var PTData = Math.floor(Math.random() * (100.8 - 91.8 + 1)) + 91.8;
    var PALData = Math.floor(Math.random() * (60 - 10000 + 1)) + 10000;

    document.getElementById("PlaneDirection").innerHTML = PDData + ' Direction';
    document.getElementById("PlaneAngle").innerHTML = PAData + ' degrees';
    document.getElementById("PlaneTemp").innerHTML = PTData + ' fahrenheit';
    document.getElementById("PlaneAltitude").innerHTML = PALData + ' feet';
});
```

End