University of Groningen

Web and Cloud Computing

Smart Energy System Project Report

Group 30

Satyanarayan Nayak, Swastik
s.nayak.1@student.rug.nl
S4151968

Palayiparambil Mathew, Anil
a.palayiparambil.mathew@student.rug.nl
S4056167

Baskaran, Siddharth
s.baskaran.1@student.rug.nl
S3922782

October 29, 2019

Contents

1 Introduction

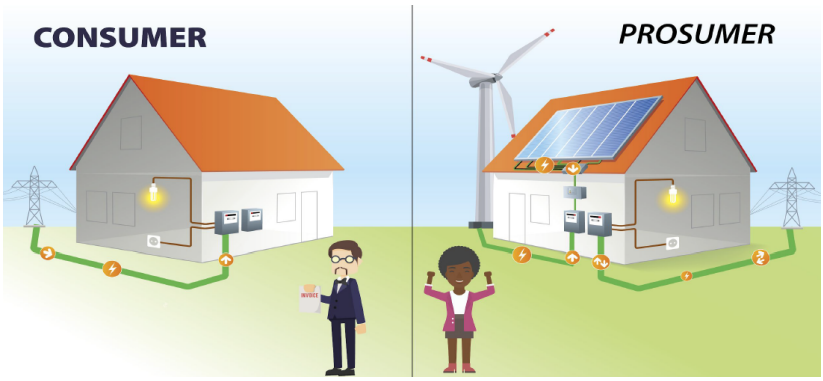

Figure 1: Prosumer & Consumer difference

The transition towards more sustainable energy transforms the passive consumers of energy into active prosumers that both consume and produce energy. some consumer produces clean energy by harvesting Solar or Wind energy, often times the excess energy harvested will be lost without a reliable storage, to avoid this loss of valuable energy the consumer is given the option to trade excess energy back to the electrical substation for which he will be remitted in the form of money and deduction from his energy bill. Consumers who generate and trade energy are called prosumers.

The Web Application developed will allow consumers, prosumers and the authorities in-charge of electrical supply to interact with one another, visualize their energy report and make sound decisions. The authorities and consumers will be able to visualize his energy consumption and the overall energy consumption and energy production by all the users. The prosumer will have and additional access to his energy generation charts, which will allow him to make sound decision on trading his excess energy.

2 Architecture

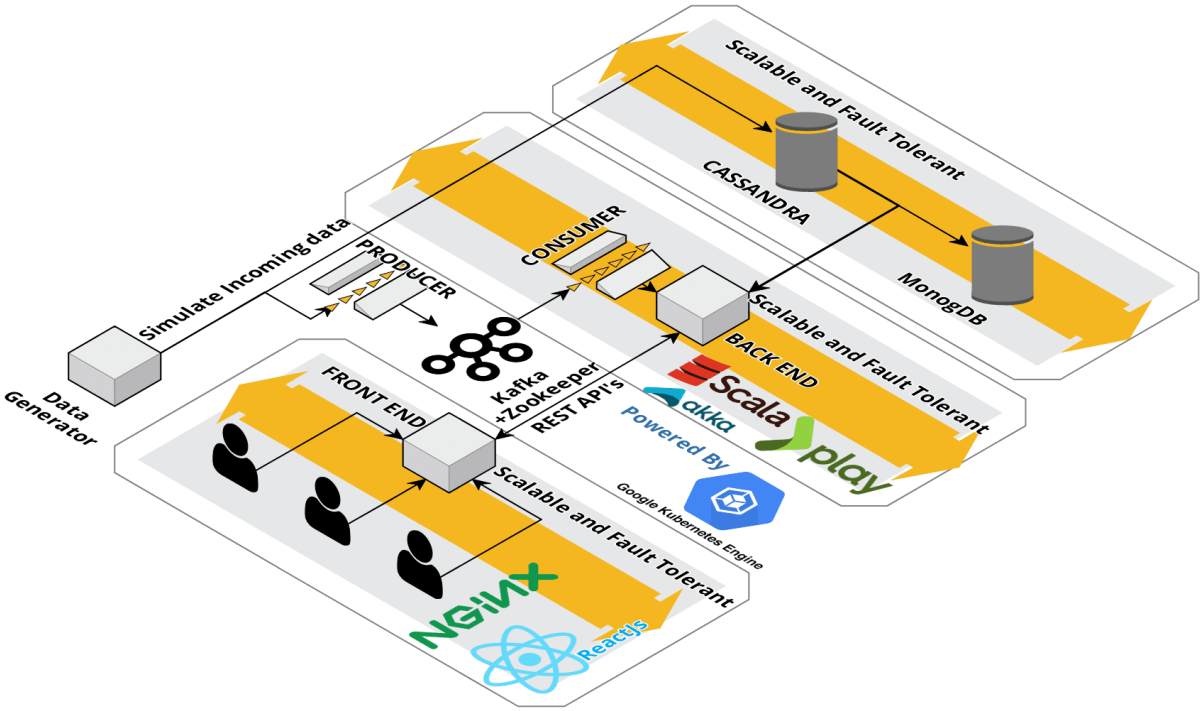

Figure 2: Smart Energy System Architecture

Figure 2 shows the architecture of the application. The users are able to access the frontend, which uses RESTful API's to communicate with the backend. The backend handles the incoming requests and handles it accordingly in a **asynchronous** fashion. User information is stored in MongoDB and time series information of the energy consumption across various applications is stored in Cassandra database with their timestamps. The backend is allowed to only read the Cassandra database and a direct write is forbidden. The Data-Generator module simulates the sensor data and dumps the sensor information to Cassandra. In the real world application this Data generator would be the source that will handle the incoming sensor data stream, this modularization was performed to decouple backend and input acquisition.

The Kafka stream is consumed directly by the backend and is a general purpose channel for message transfer. The module is generalized to handle any topic, the frontend can request the backend with a topic name to which the backend will return the messages that is present in the kafka channel. The Data generator simulates the kafka stream with the total energy consumption and generation for all the users. The data generator can be swapped with a different component that outputs relevant kafka topic.

The main Key Drivers for our project are:

- **Scalability :** Systems ability to handle gracefully a growing amount of work. Our system provides horizontal scaling by increasing the number of replication sets and vertical scaling is done by increasing the CPU and memory which is done using the cloud.
- **Maintainability :** It is the highest concern because the system has to enable reusability and maintainability. Our team has proceeded to develop our application in a modular pattern. We are using containerization technology to separate all our layers to reduce the complexity and increase the readability of our code. We have mainly three layers(Frontend, Backend and Database) and all the related components are developed in their respective layers. With this approach, in the future we will be able to change any layer and it will require minimum changes.
- **Fault tolerance :** The degree to which a system, product or component continues to function properly in the event of failure in some of its components. With the use of docker containers and Kubernetes orchestration, we are able to handle fault tolerance. If a pod fails, our application will still remain running as we have the instance running in multiple pods and thus its fault tolerant.

2.1 Frontend

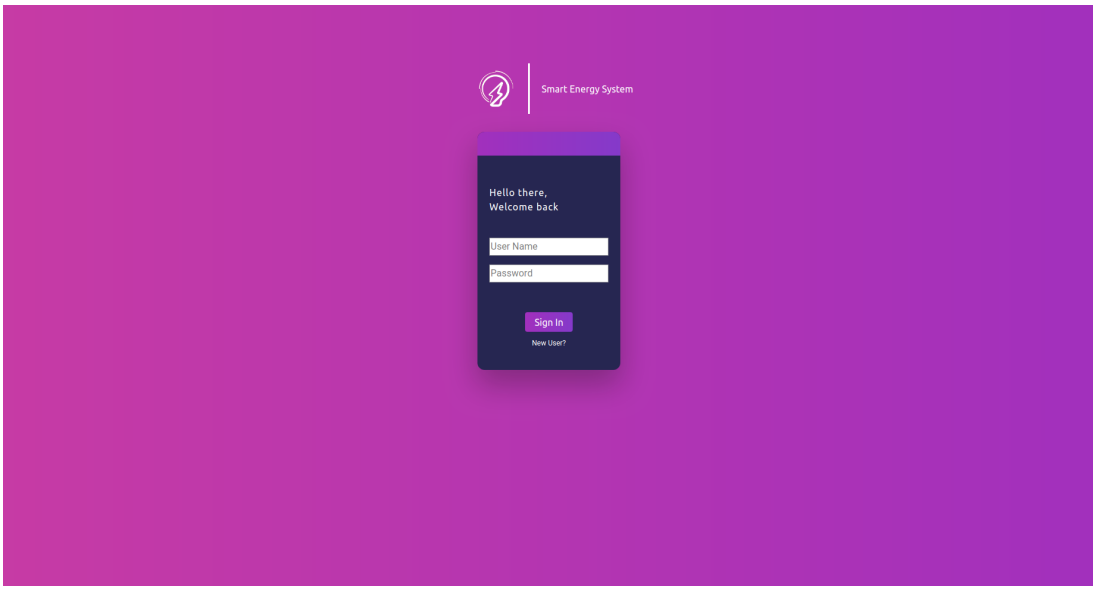

Figure 3: Smart Energy System - Login

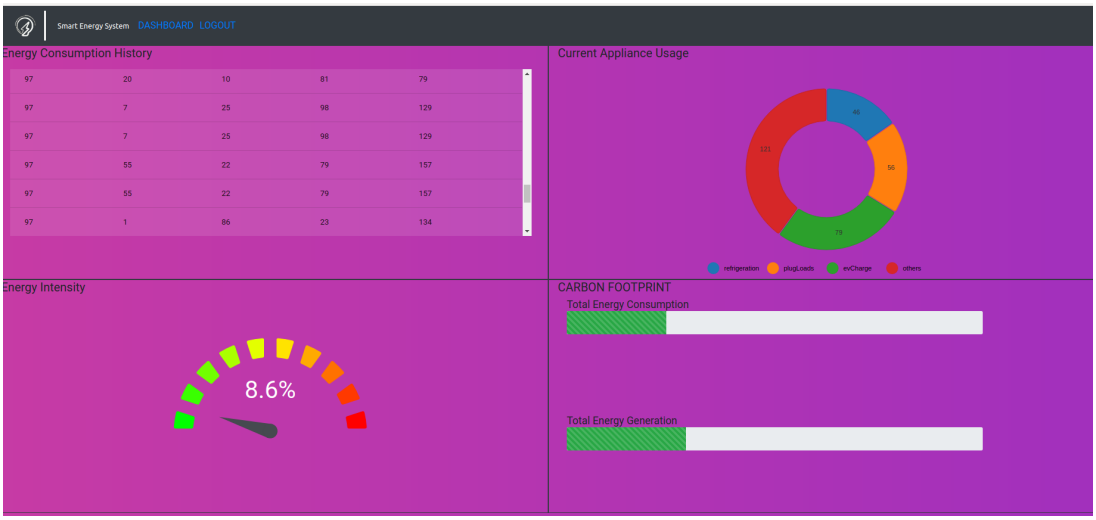

Figure 4: Smart Energy System - Dashboard

For our frontend, our team has planned to go with **React version 16.10.2** which is the leader among the various SPA technologies like Angular and Vue. Moreover, with the use of React we are able to satisfy our main key driver which is a scalable and lightweight solution because React is Javascript library rather than a framework. It uses Virtual DOM which makes the page load blazingly fast.

For the frontend, we have implemented two main modules which is **Login** and **Dashboard**. The Login module provide the user an interface to login and sign-up. The first step that the user needs to perform is to sign-up in which the user name and password are input text and password fields that will be sent to the backend and inserted into MongoDB. The Login screen gives an interface for the existing users to enter into the application using the user name and password. The overview or the dashboard modules gives a holistic view of the energy data for that user. We have used interactive charts like bar, pie and speedometer charts in order to represent the user information. The data for the charts are received using web sockets (Kafka with a topic and normal) which fetches values from Cassandra database.

We have also performed basic testing for our React front end application using **Jest, Enzyme, Chai and Nock**. The below tests cases were executed:

- **Render Component without crashing :** This checks if the component is loaded on the screen without the application itself crashing.
- **Check API call response :** We check if the API response matches the expected result. In this case we checked if the authentication is executed without any error

Figure 5 provides an overview of the unit test cases executed on our application.

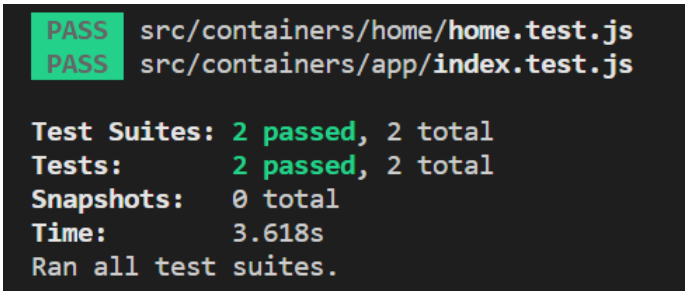

Figure 5: Smart Energy System Front end - Test result

2.2 Backend

The Backend code is predominantly built on Scala, Play2Framework, Akka and Kafka. Scala is consise, eliminates the need to write boilerplate code, supports functional and object oriented design all the while inheriting the advantages of a JVM, scala is flexible and results in a overall shorter development turnaround time. Play2Framework is based on a lightweight, stateless, web-friendly architecture. It is built on Akka, Play provides predictable, asynchronous and

minimal resource consumption (CPU, memory, threads) for highly-scalable applications [1]. Play2Framework's dependency injection was used to create singleton objects, and the framework allows the developer to recompile the code during an execution which is desireable. Play supports reactive mongo that provided self contained encapsulation to quickly deploy the mongoDB functionalities. Its is robust and its ability to recover from exceptions is highly regarded to create a fault tolerant backend application.

MongoDB for User information: The user is allowed to register himself on the login page. The backend will store this user information will be stored in the MongoDB. To avoid inserting duplicates into the Database, we have implemented a logic that generates a unique id based on the user name, the user is forbidden to change his user name due to this functionality.

AUTHENTICATION: The application uses a basic authentication of password. On an Authentication request from the frontend the backend will fetch the user's password stored in the database and will perform a simple authentication.

Cassandra for Sensor data: The time series data from the sensor's are being stored in the Cassandra. The module which dumps the sensory data into Cassandra should be handled externally to promote modularization, currently Data Generator is a module in this project which handles this input dump into Cassandra. The data present in the Cassandra has a timestamp which indicates the arrival time of the data, the backend queries the Cassandra database based on the user information. The backend sends the sensory data into 2 Websocket channels, one where the user's current energy consumption will be published, and another where the user's energy consumption for a given time frame will be published.

Kafka support: Kafka support is implemented and can be readily used any future requirement. The implementation is a generic kafka consumer that read messages from any requested topic. Currently the kafka implementation in the application is responsible for streaming the total energy consumption and total energy generated by all the users, the kafka messages are sent using a Websocket. KafkaUtils has been setup for any out of the box kafka requirements.

Configuration: The module utilizes the play2frameworks default configuration potential. Provisions have been made to control the application through external environment variables. for example: The configuration for Cassandra Host is declared as follows,

```
cassandra.hosts = "localhost"
cassandra.hosts = ${?CASSANDRA_HOSTS}
```

We can override the **cassandra.host** value by setting a **CASSANDRA_HOSTS**. This environment variable. This is especially useful as it eliminates the need to redeploy the entire application if any configuration change is required.

API calls: The playframework api calls are listed in the routes file and the calls to the backend are defined in the following fashion:

Type	Route uri	Controller
GET	/kafka_Actor_stream	controllers.StreamController.akkaActorSocket

The api calls can be summarized as
$< back-end-hostname-or-ip >:< portnumber > /kafka\_Actor\_stream?topic = testtopic$

Scalability and Fault tolerance: The module allows for horizontal scalability, the deployment is orchestrated by Kubernetes and the backend service is run as a Loadbalancer. The backend utilizes kubernetes DNS couples with service name calls to access other container services, this eliminates the possibility of backend bottle-necking the application and to allow scaling of its environment. Play2framework provides out of the box fault tolerance, in addtion to this the kubernetes Auto healing characteristics makes the backend robust.

2.3 Kafka with Zookeeper

We used kafka to stream messages to the backend like the global information (total energy consumption and generation for all the users). Kafka has good performance and has excellent fault tolerance capabilities. The Kafka service relies on Zookeeper for replication and management. The kafka implementation also provisions a generic api call to the backend that can read from any requested kafka topic within the scope of the kubernetes cluster.

[1] https://www.playframework.com/

2.4 Cassandra

The Apache Cassandra database is the right choice when you need scalability and high availability without compromising performance. Linear scalability and proven fault-tolerance on commodity hardware or cloud infrastructure make it the perfect platform for mission-critical data. Cassandra's support for replicating across multiple datacenters is best-in-class, providing lower latency for your users and the peace of mind of knowing that you can survive regional outages [2].
Cassandra performs exceptionally well with timeseries data, the sensory data generated is a timeseries data and can maximize the capabilities of Cassandra. In the application we use Cassandra to store the Energy consumption information captured by the sensors in the appliances registered. The backend queries the Cassandra for the current energy usage of the user logged in and the historical consumption of the user.
Cassandra is schema oriented database and to not limit our sensory data to a specific structure we are storing them as key-value pairs, where the key is the Identification of the user with timestamp and the value is a json string which can be readily parsed.

2.5 MongoDB

MongoDB is a general purpose, document-based, distributed database built for modern application developers and for the cloud era[3]. MongoDB does not maintain a schema of the tables and can be used to store the user information which requires constant changes as per our requirements. The user information is not a timeseries data and would not utilize the full potential of Cassandra. In the application we use the MongoDB to only store the user information and is queried when user related operations like CRUD and authentication is required. MongoDB is schemaless, document oriented, easy to scale out, no complex join or deep nested queries, faster reads due to internal windowing. MongoDB is recommended to be used for user data management requirements [4].

2.6 Data Generator

The Data generator is a module that supplies that simulates the sensory data input stream and also populates the kafka topics with inputs to consume. The Data generator requires a MongoDB, Cassandra and Kafka to be up and running for it to start populating the channels. The data generator queries the mongoDB to any new Users and generates a series of sensor data at internvals of 3second and populates the Cassandra and kafka with the sensory data and total energy statistics of all the users respectively. This helps us simulate real world scenarios and test our application with respect to features, scalability and fault tolerance.

3 Deployment

3.1 Google Cloud Platform

We have decided to use **Google Cloud Platform** primarily as we have used Kubernetes container-orchestration system. As Kubernetes is originally designed by Google and Google cloud having a better dashboard interface for Kubernetes made our cloud choice pretty simple. However, we have tried deploying our solution on other cloud service providers like Microsoft Azure and Amazon Web Services. In the case of Azure Kubernetes Service(AKS), we deployed the Kubernetes solution but had to use the Kubernetes native UI to view the deployments, services, pods, ingress and configMap for our application.

[2] http://cassandra.apache.org/
[3] https://www.mongodb.com/
[4] https://www.tutorialspoint.com/mongodb/mongodb$_{a}dvantages.htm$

3.2 Kubernetes

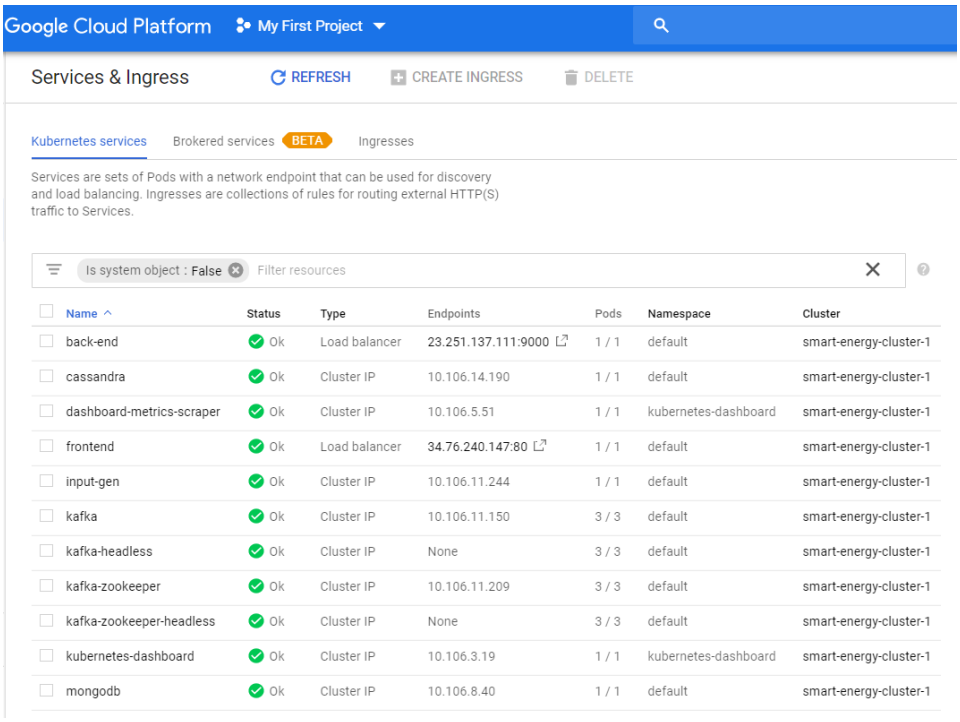

Figure 6: Google Kubernetes Engine - Deployed Services

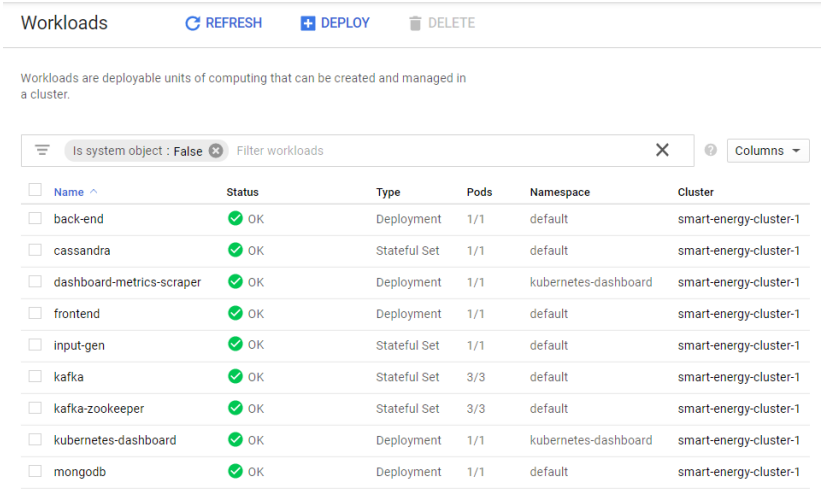

Figure 7: Google Kubernetes Engine - Pods

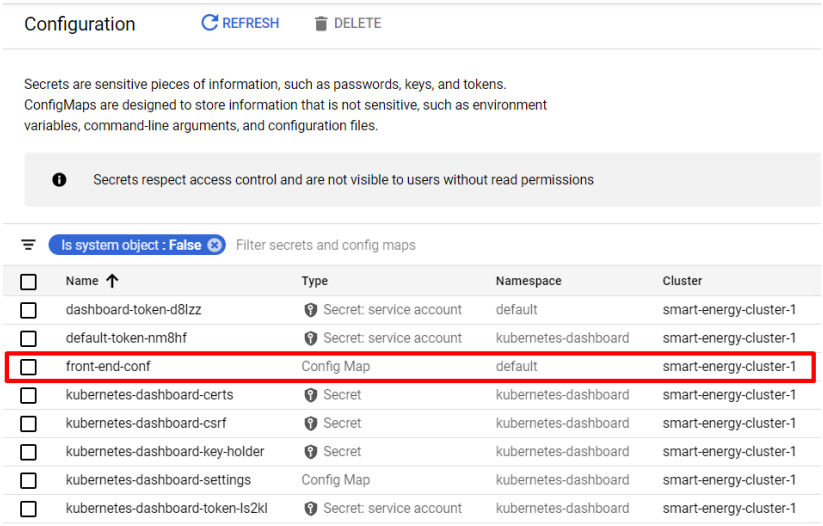

Figure 8: Google Kubernetes Engine - Configuration

Our team has planned to deploy our application using **Google Kubernetes Engine**(GKE) which is a service provided by Google Cloud. The advantage of using the GKE is that Google provides its own dashboard in order to handle all kubernetes related activities. In other clouds, the GUI for kubernetes is handled using the Kubectl UI dashboard. Using the dashboard provided by Google we are able to handle easily vertical and horizontal scaling of our application.

Fault tolerance and scalability of our application is completely handled by Kubernetes service. When deploying any instance we provided the replication sets so kubernetes service ensures that our instance will have the required running instance every time. We have exposed our Front-end and backend service static ips using the load balancers type of deployment and finally with the help of ingress the routing rules that govern how external users access services running in a Kubernetes cluster.

3.3 Kafka

Both Kafka and ZooKeeper are deployed in cluster mode with 3 replicas. A minimum replication of 3 is required to satisfy the Leader election algorithm. Kafka relies on ZooKeeper for replication and management. Kafka deployed in our application is for a general purpose streamer.

3.4 Databases

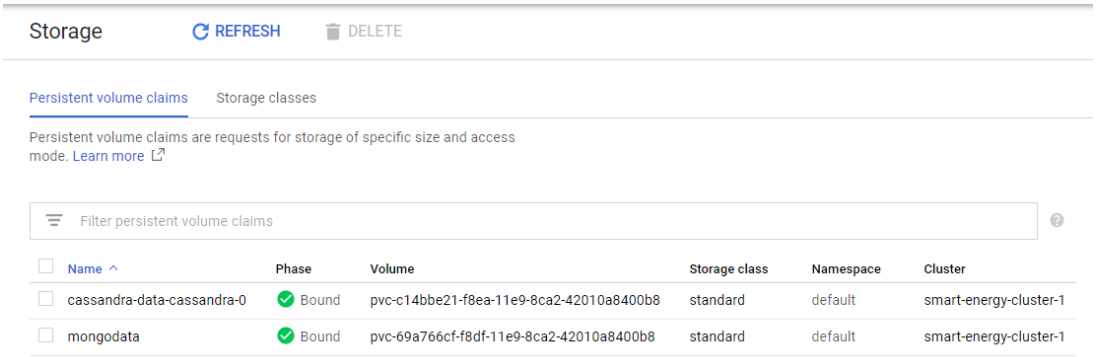

Figure 9: Google Kubernetes Engine - Storage

We are using two NoSQL databases for our application which are MongoDB and Cassandra. MongoDB is used to store user name and password details for the users. While Cassandra stores time series data that is generated using the data generator. We are using 3 replications for each MongoDB and Cassandra. Figure 9 shows the persistent volumes for MongoDB and Cassandra.