Smart Energy System

Thomas van Dongen (s2984318)
Diego Cabo Golvano (s3736555)
Neha Bari Tamboli (s3701417)
Group 5

October 26, 2018

Abstract

We designed a smart energy system consisting of two main components namely, MyHouse and Marketplace. A Smart Energy Systems with regard to the issues of definition, identification of solutions, modelling, and integration of how energy consumption and energy production can be managed for some basic home appliances. The marketplace component consist of how the users will be able to buy and sell energy on a marketplace in order for efficient energy management, while the MyHouse component provides transparency about the users own appliances. The conclusion is that the Smart Energy System concept represents a scientific shift in paradigms away from single-sector thinking to a more coherent energy system.

Introduction

A renewed focus on sustainable energy has created more awareness surrounding the subject of energy consumption, as well as production. Many house owners want to adapt, but find it hard to do this. Because of this, we decided to design a system that helps users achieve their new energy goals. Our application focuses on two main issues:
1: providing transparency in what a household is producing and consuming.
2: providing a marketplace system where house owners can trade any spare energy they have.
In many cases, it is expensive to store a large amount of spare energy. Instead of constantly increasing the energy load that a household can store, we instead want users to sell their spare energy to other users. This is a scalable solution that can eventually create a network where no energy is wasted and where energy will be cheaper as well.

1 Technology Stack

The following technology shows the formation of our architecture.

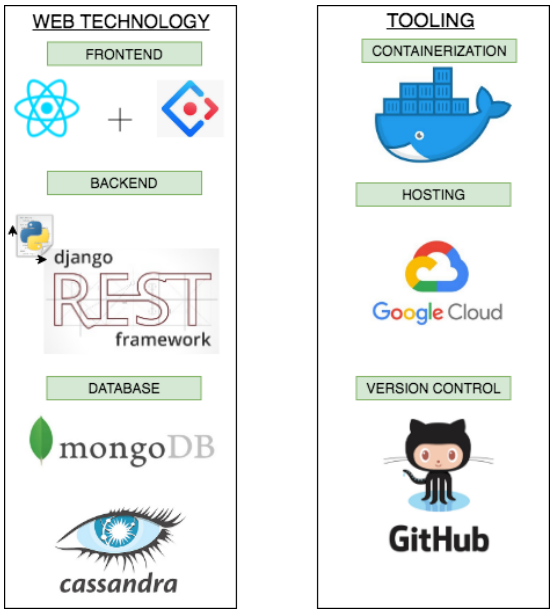

Figure 1: Technology Stack

1.1 Front-end

For our front-end we used React as our JavaScript library. After trying out code examples from multiple libraries, we concluded that React was the best fit for our needs. React is very a simple and lightweight library that only deals with the view layer. One of the biggest pros of React is that it offers a full solution for creating a manageable front-end using a component-based pattern. This makes it easy to create different classes, each with their own state, something that can be quite difficult/unmanageable when using jQuery for example. For our webpage design we used two different UI frameworks: Bootstrap and Ant design. We used Bootstrap for it's usefulness in creating the overall layout of the page (including grids), while we used Ant for some of the more specific features such as form creation and handling. For the styling of our pages we used CSS. The last important technology that we used for our front-end is Axios, a promise-based HHTP client for node that allows us to do asynchronous communication. Internally, it uses XMLHttpRequests, however we wanted to use promises instead of callbacks for all our functions since it is a cleaner solution which also provides a catching mechanism (which can be very important for debugging purposes). Because of these reasons, we decided to use Axios.

1.2 Back-end

For our backend we have used Django as the main framework. Django is an open-source web framework built in Python. Django actually provides everything for the Model-View-Controller pattern, however we were just interested in the controller part and the view part to a lesser extent. One of the main reasons for choosing Django was the large amount of documentation available. Having never worked with a web backend framework before, we concluded that it would be best to choose something with a large amount of help available online. We also liked the fact that Django is built with Python, which is the language we preferred to use for the backend.

After looking at the functionality that Django offered, we found out that there are some great libraries extending the power of Django. The most important one was the Django REST framework, which provides Django with full REST functionality. We also made use of additional libraries for connecting our back-end to the databases. We used Djongo for MongoDB and Django-Cassandra-Engine for Cassandra. These two packages made it very simple to populate the database with objects using a Django-style model's definition.

1.3 Databases

We use Cassandra and MongoDB as our two main no-SQL databases. Both are highly scalable databases, a must-have when designing web applications that are used by thousands of users. Also, both databases can be easily and efficiently replicated across multiple nodes in order maintain availability in the event of outages or planned maintenance events. Besides those general reasons that can make one choose a no-SQL database over a regular SQL database, we have chosen specifically these 2 for the following reasons:

- **MongoDB** for storing user's personal data. Some of mongo's main features and reasons to chose this database for our project are: consistency, high availability, flexibility (due to document-type DB and no rigid schema) and big community support due to large share of users.
- **Cassandra** for storing user's time-series data, such as consumption/production rates, or a specific appliance's consumption. Cassandra is a noSQL database optimized for high write throughput, so it's the ideal database to store this kind of data.

2 Application Architecture

A diagram of our web application's final architecture is presented in Figure 2. After several trials and errors exploring multiple possibilities (localhost, Amazon AWS) we finally deployed our application within Google Cloud Platform. Several reasons triggered the decision of deploying our application in Google Cloud:

- The hardware requirements for having some level of fault tolerance in our application (replicated databases and several containers of front-end and back-end) were too high to have everything running on a single machine.
- Kubernetes is completely integrated with Google Cloud in their Google Kubernetes Engine (**GKE**). Google is behind the creation of Kubernetes so there is a good amount of documentation about it.
- Google Cloud offers $300 when signing up with a new account. That gave us some freedom to experiment with their engine and eventually we were able to deploy our application for free.

We create and manage our application with GKE. We use 3 virtual machines for deploying the Kubernetes cluster. A MongoDB replica set is created using the default docker image from Docker Hub. A mongo-sidecar docker container takes care of creating and connecting the 3 mongo nodes. Cassandra is deployed using Google's marketplace application "Cassandra Cluster". Finally, the back-end and front-end are deployed with 3 replicas and a LoadBalancer for each. Additionally, all of the activity of the application can be monitored by the application Stackdriver.

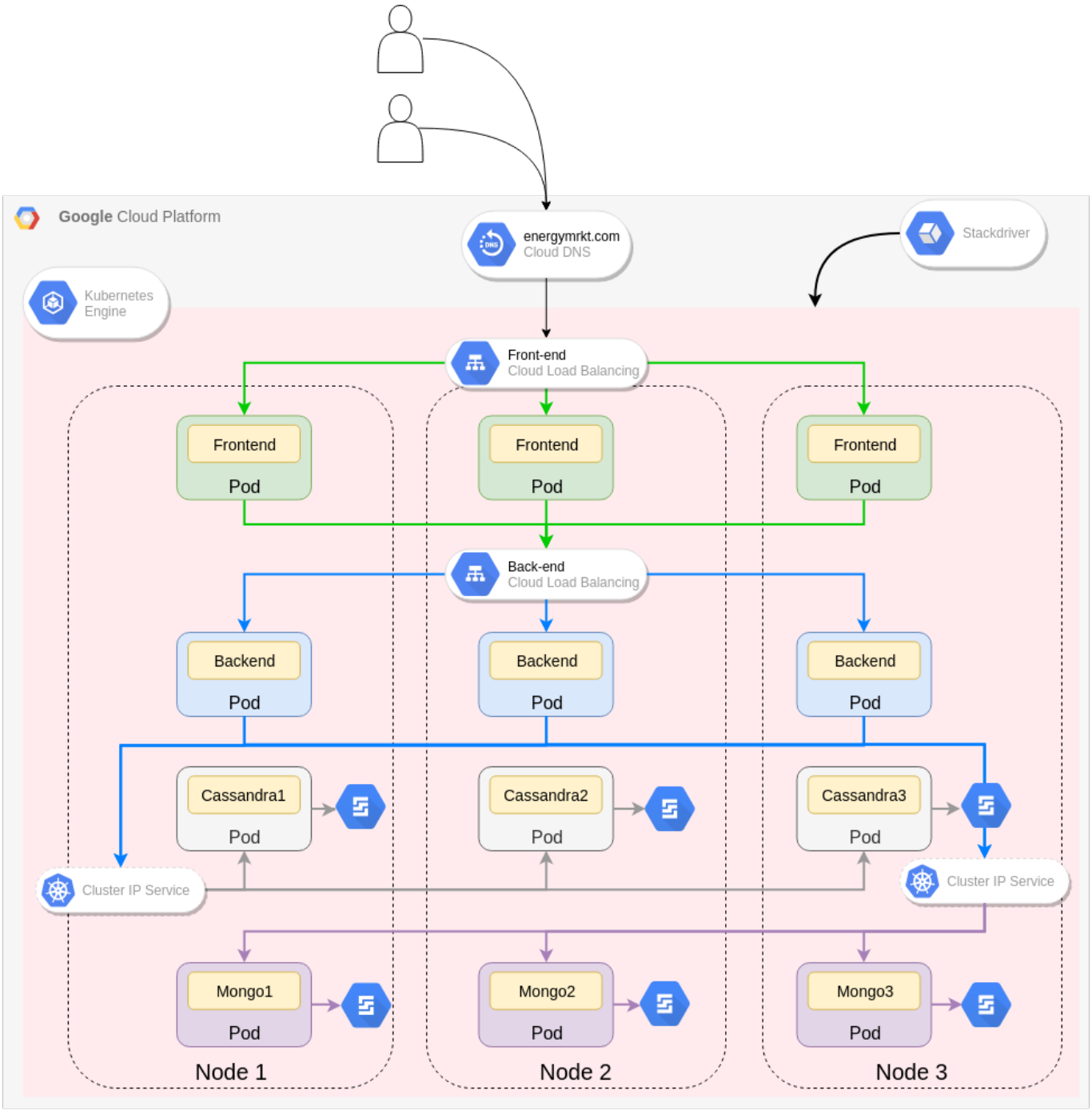

Figure 2: Web App Architecture

2.1 Front-end Architecture

The front end architecture is designed using React and there are various components that work together. at the base, there is an App.js file that is the first file that is being called for the front end. In here we define our router which is used to make it a single page application. This wraps the entire application. Then, we define our layout which wraps all of our components. The base layout includes anything that is shown on every page, in our case the navigation bar and the footer. We make heavy use of Redux, a library that is used to handle the application state. We mainly use it to check whether a user is actually logged in for the MyHouse and Marketplace pages using an isAuthenticated call which checks whether there is a user login token in the current session.

2.1.1 User interface

- Navigation bar: the navigation bar includes links to all our different components. It also creates a drop-down menu when the screen size becomes too small, making the application mobile friendly.

- Footer: a simple footer that is displayed on every page, showing some basic contact information and mock links to social media platforms.
- Home page: our home page is the basic landing page for our application. It provides links to our two main modules: MyHouse and Marketplace.
- MyHouse: the MyHouse page provides users with all their information. It includes information about their money, battery, energy consumption, energy production and appliances. Users can update their appliances, after which the consumption and production rates are automatically updated. This creates a dynamic environment where a user can input what appliances they own and can automatically see what their energy rates are.
- Marketplace: the marketplace is the main component of our application. Here, users can buy offers form other users, or create their own users. Architecturally this is also the most complex page of the application. The marketplace displays OfferListView, which is the main wrapper for two components: Offer and OfferForm. Offer is the view for a single offer. This offers the functionality to buy the offer, giving the user who created the offer the allocated money and subtracting the money from the buyer, while also updating the energy that is being bought and sold. The OfferForm offers the functionality to create custom offers, giving the user the option to choose a price and offer an amount of their energy.
- Login and Signup: the login and signup pages for users. They send the user data to the store.

2.1.2 Authentication

A critical part of the front-end includes the store, in which all functionality for user authentication is provided. The store contains two versions of auth.js.

- actions/auth.js: a file that contains all the methods that the front-end uses when a user makes any authentication calls. These incluse authLogin,authSignup, logout as the main functions and authStart, authSucces and authFail as dispatch functions. We use sessions to store data for the user session. There are four values being stored in the session, mainly: token, expirationDate, username and id. The token is the unique token that is dispatched when a user logs in. This is used to make sure that the user stays logged in, even when leaving the website. The experiatonDate is then used to automatically log out the user after one hour of inactivity. There is also a small file called actionTypes.js which just includes constants that are used for printing with Redux DevTools, a handy chrome plugin that prints calls being made using Redux.
- reducers/auth.js: this file contains reducers which specify how our application's state should change when one of the actions is sent to the store. For example, when authSucces is called, the states token is set to the new login token.

2.2 Back-end Architecture

As mentioned previously the back-end is created in Django framework along with its components.

- In the Admin component we basically import the models and then simply register them using the "admin.site.register". From here on we further access the other components.
- The Model component, The model component is used to model the database and how it will be stored in the database. There are 5 models created that will update the data in the database dynamically. Namely as, Money,Battery,Household,offer,Energy. The details are explained in Database Architecture.
- The URL component the serialized views for the data and its operations(view,update...) and sets a path for those serialized objects for the front-end.
- The view component contains the serialized JSON objects which are sent to front-end basically via URL.

2.3 Database Architecture

As mentioned before, we are using two different noSQL databases: MongoDB and Cassandra. In order to obtain a good degree of fault tolerance, we replicated both databases across 3 different nodes. Each database cluster is managed by Kubernetes, as defined in the configuration files. If a pod dies, another one will be created immediately. The Cassandra cluster was created using the Kubernetes App from Google's Marketplace. MongoDB replica set was created by us. We used a YAML configuration file to specify the replica set's StatefulSet deployment configuration (like number of replicas, persistent volume claim, Mongo's side-car container...). Both are connected to the back-end using DNS (this is where the service comes into play) so if a pod suddenly dies, the newly created pod would be accessible from the back-end with no interaction from our side.

As to which objects are stored in which database, we provide a brief list below:

- Django's default data: MongoDB is the default database for Django, so all information that Django stores will be saved there (users, tokens...)
- Household: This is where we save a household's data. Currently consists of: user_id, username, money, and battery. This objects will not change much or not at all during time, hence we save this in MongoDB.

- Offer: We store the offers made in the marketplace. Each offer consists of: user_id, username, amount of energy to sell and total price. We store offers in MongoDB since we don't expect an extremely large amount of offers going in and out.
- Energyrates: This is where we store all the energy information for a specific user. This includes the owned appliances, as well as the production and consumption rates (which are based on the appliances). The appliances consist of (number of): stoves, lights, household_appliances, home_entertainment, solar_panels and windmills. We store this in Cassandra since the rates will go up and down every second and it is useful to store old data (to see if you are using less energy than last month for example).

3 Fault tolerance

We have taken different approaches to make our application fault tolerant. The most important are listed below:

- We have our front-end and back-end fully separated. This enables the the front-end to keep running even if the back-end dies.
- We deploy our application on Google Cloud. This enables us to have our application always running, without having to use our own hardware. Additionally, cloud providers usually have a very small downtime. This means that if the servers were to fail, it should only be for a few hours.
- We use load balancers for the front-end and back-end
- DNS is used for connecting to the databases, not hardcoded IPs.
- We use an orchestrator. By using Kubernetes we were able to accomplish several things.
  - Creating a 3 VM cluster, which protects our application against hardware failures in a specific machines.
  - Create 3 instances for the front-end and 3 for the back-end, avoiding an application failure if one of them goes down.
  - Create a replicated database. If a database node fails, the data is replicated along the cluster, avoiding loss of data. It would still also be accessible for reads
  - Kubernetes orchestrates all the containers. If, for example, one of the back-end containers dies, a new one will automatically be started.

4 Future upgrades

We were not able to implement a live-simulation element into our application. Our initial goal was to generate data real-time based on the users appliances and use websockets to continuously send this data to a graph displayed in the front-end. This could be a future upgrade.