PRATIK BHUJADE

● Part 1 --- Method Overview

This application was made using **Google's app engine** and **python 2.7** The Google app engine is included in the Google Cloud SDK which allows us to develop applications which can be deployed to Google Cloud. This project also utilised bootstrap for styling the webpages. **Bootstrap** is a **CSS** framework which is used to develop responsive webpages. Bootstrap includes templates for all the basic **HTML** elements like buttons, text boxes, checkboxes etc. **Jinja2** was also used in this project. Jinja2 is a python based web template engine which is used to generate HTML content which is then returned back to the user over **HTTP.**

In this project classes are used to specify and implement different functionalities. Each class has specific functions and the code required to implement a specific functionality. All classes in this project extends webapp2's **RequestHandler** class in order to handle HTTP requests. This class contains the application logic to handle a HTTP request. This class can be thought of as a Controller in **MVC,** i.e these classes processes the HTTP requests and generate an appropriate response and return it to the browser.

The defined classes work by calling a method as per the corresponding HTTP request so that the server can process the request. For instance - Suppose I want to perform search operation and I click on the search link, initially as the page is loaded a **GET** request will be sent to the server which will be processed by the Search class. Upon reaching the server, Search class's **get()** will be called as the incoming request from

the browser is GET request. If the incoming request was a **POST** request then **post()** would have been called. The called method will process the request and return a response to the browser which will then be displayed to the user. The methods defined in the classes take an argument self. This tells that this method can only be called via an instance of the class.

For mapping the urls meaning which class should be referred when a particular url is hit, we define an object of **WSGIApplication**. The class WSGIApplication contains all the code required to route requests to particular class. When defining the WSGIApplication object we specify a list which maps urls and classes. WSGIApplication object will map url requests to classes and then depending on the type of request appropriate method of that class will be called.

In this project I have structured the code in different files. All the classes representing the functionalities are specified in the main.py file. This file contains all the classes used by the application. Similarly for mapping the urls a file called **urls.py** is used, meaning in this file the WSGIApplication object is defined. The file **app.yaml** is used to define all the runtimes and libraries which are required by the application. The google app engine uses this file to know about the dependencies and libraries required by the application. Another file called constants.py is used which stores all the constants which will be required by the application. This file contains the object of jinja2.Environment class. This object tells the app engine where to get the HTML templates which will be used by the application.

In the file **main.py** all the logic required by the application is written. This logic is defined inside classes all of which extend the **webapp2's RequestHandler** class. As per the requirement appropriate functions are defined inside the classes mapped to particular type of HTTP requests. Inside these functions a lot of built-in functions are used for implementing the required logic. For instance - Suppose I need to check whether a user is logged into the system or not for that purpose the app engine's users class has a function called **"get_current_user()"** which returns the user object of the currently logged in user and if no user is

logged in then the function returns **None.** This can be then be used to check if a user is logged into the system or any other functionality which may require the user object of the logged in user can use this. The application uses jinja2's **get_template()** method to get the HTML template which needs to be rendered. This template is then returned as a response to the browser. If required, data can also be passed to the jinja template by passing a dictionary containing all the keys and values which will be rendered in the template. This dictionary is passed as argument in the template's **render()** method. For specifying the type of response the server must return to the browser I have defined **response.headers**. By specifying the **response.headers['Content-Type']** to 'text/html' it is being specified to the server that the function will return a response in text or HTML format.

For retrieving data from HTTP responses the get() method of the request class is used. This function will retrieve the data received in the HTTP request. For instance - Suppose the user submits a form, all the data input by the user is sent to the server for processing. This data is sent in key-value pairs. By using the get() function and specifying the key as argument we can retrieve the associated value. This value can then be used for processing as required.

For retrieving data from the datastore I have used the **query()** method. With this method data can be queried using multiple fields and parameters which is a complex task on a **NoSQL** database like datastore. The query() method can take arguments or run without arguments. As arguments we pass the fields we want to compare and if no arguments are passed then it will return the entire list of objects of that model class. The query() allows us to perform various SQL like operations like AND, OR, IN etc.

For building login and logout system I have used app engine's built-in authentication system. To implement this functionality users class's **create_login_url()** method is used to generate the url required for logging into the system. Similarly for logout **create_logout_url()** method is used to generate the url required for logging out of the system.

● Part 2 --- Models and Data Structures

This application uses datastore to store data. Datastore is a NoSQL database which is offered on the google cloud platform. Datastore works on the key value pair system.

In this project all the models are stored inside a folder called "models". Inside this folder there are python files which represent tables in a database. For allowing the datastore to recognise the files as datastore models the model classes must extend the ndb.Model class. This allows the datastore to store and retrieve our classes. In our model class we define all the properties of the data models.

In this project I have used 2 models, first is the ElectricVehicle model and second is the Review model. The ElectricVehicle model has the following properties - vehicleName, vehicleManufacturer
, year, batterySize, wltpRange, cost, power, averageRating. As mentioned above this class also extends the ndb.Model class. This model has used various properties like **StringProperty()**, IntegerProperty() and **FloatProperty()**. These define the type of data a particular column will have. In ElectricVehicle model vehicleName and vehicleManufacturer uses StringProperty(), year, batterySize, wltpRange, power use IntegerProperty() and cost, averageRating use FloatProperty(). No structuredProperty was used in this project because that level of binding is not required in this project.

As we have seen in ElectricVehicle mode, the Review model also same basic properties and an additional property called **DateTimeProperty()** which is used to present date and time. The Review model also extends the ndb.Model class and has the following properties - user, vehicle, review, rating, date. This model has used various properties like StringProperty(), IntegerProperty() and DateTimeProperty(). In the Review model user, vehicle and review are represented by the StringProperty(), rating is represented by IntegerProperty() and for representing date, DateTimeProperty() is used.

The project uses another model called the users model which is built into the app engine. This model is used to represent the logged in users and it stores the email field. This built in model is what enables us to use the built in authentication service provided by the google app engine. The users model has built-in methods like **get_current_user(), create_logout_url(), create_login_url()** etc which further help in the development.

- ## Part 3 --- UI Design

Home page

On this page 2 options are displayed to the user, first is to login and second is to search. These are the only two functionalities which are available to a user if he/she is not logged in. From these depending on the preference if the user wants to just search for a vehicle he/she can do so without logging in or if they want to do some other operation they can click on the login button and login and perform more operations like add vehicle, compare vehicle, review vehicles etc.

Search page

On this page the user is presented with all the possible filters they can use to search for vehicles. The user can combine any number and combination of filters as they want can get results. The string values like vehicle name and manufacturer have a single input box but numerical values have 2 different input boxes one for low value and one for high. This enables a user to enter range between which the user can get results. And having the ability to combine any number and combination of properties allows for more detailed filtering.

Edit page

On this page the user can edit vehicle details. In the form the user is presented with the current values of the vehicle so that the user can know the current values and make changes accordingly this enables the

user to make changes without having to recheck the values somewhere else and quickly make changes as desired.

Compare page

On this page the user can select vehicles from a list of checkboxes and pick the vehicles which they want to compare. The vehicle names are presented in a single line as checkboxes so that the user can select any number of checkboxes and compare as desired. Also selected vehicles values are presented in a tabular format in such a way that the vehicle has higher value for a particular property is highlighted in green and the lowest one is highlighted in red while the values which lie in between and presented in white. For instance - If 3 vehicles are selected for comparison then the highest battery range will be green, the second highest will be white and the lowest one will be red.