PRATIK BHUJADE

● Part 1 --- Method Overview

This project 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 eg. - 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 eg. - 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 eg. - 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 nab.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 3 models, first is the **TaskBoard** model, second is the Task model and third is the User model. The TaskBoard model has the following properties - **taskBoardName, taskBoardTask**. As mentioned above this class also extends the **ndb.Model** class. This model has used only **StringProperty()** along with repeated = True argument. This argument states that the column can now store a list of objects of a particular type. In this case **StringProperty(repeated = True)** says that the column will store a list of String objects. In TaskBoard model only StringProperty() is used because in this entity I am only storing the name of the task board and key of the tasks created under the task board.

The Task model is more complex as it uses different types of properties. The Task model also extends the ndb.Model class and has the following properties - **taskDescription, taskDueDate, taskFlag, taskUser, taskFinishDate, taskBoard**. This model has used various properties like **StringProperty(), IntegerProperty(), BooleanProperty()** and **DateProperty()**. In the Task model **taskUser, taskBoard** and **taskDescription** are represented by the **StringProperty(), taskDueDate** and **taskFinishDate** are represented by the **DateProperty()** and for representing **taskFlag, BooleanProperty()** is used.

The User model is used to represent the logged in user. This model also extends the ndb.Model class and has the following properties -

user_email, created_boards, added_boards. This model has used only StringProperty(). In this model the user's email and

the keys of task boards which the user has created and joined are both stored separately.

● Part 3 --- UI Design

Home page

On this page the user is initially greeted with a login link. Without logging in the user cannot access the rest of the functionality of the system. After logging in the user is presented with the links to create board, view boards and logout.

Create Task board page

On this page the user is presented with a navigation menu which the user can use to navigate to other available functionalities and an input box and a submit button. On the input box the user is presented a placeholder showing that this box for entering the name of the task board. From there the user can create a new task board.

Task board list page

On this page the user is presented with a list of all the task boards the user has joined or created. These names are hyperlinks which allow the user to view the board details and perform other operations when clicked. On this page a navigation menu is also shown. Also an “Owner” message is also shown behind the names of the task boards which the user has created.

Task board details page

On this page the user is presented with the various operations the user can perform on the task board. First the name of the task board is displayed below that a dropdown is shown which has the names of all the users who have previously logged into the system, from there a user can

be selected and added to the task board. Once the user is added to the board the name of the user removed from the list of dropdown. Below this section the create task section is present. In this section the user can create and assign task to users. In the dropdown of this section only the names of those users is displayed who are already added to the task board. Below this a table showing the details of the on going tasks on the task board is displayed. This includes details like the task title, task owner, due date and a complete check box which when selected can be used to end the tasks which are shown. Here checkbox is used so that multiple tasks can be selected and ended at the same time. If a user who has on going tasks is removed from the board then in the table that on going task is highlighted in red until that task is assigned to a new user. The task detail page is accessed by clicking the name of the task which is a hyperlink. On this page data regarding the total active tasks, total tasks etc is also displayed. Also in the navigation menu the “Edit board” option is shown only to the user who created the board.

Edit board page

Form this page the user can change the name of the board and also from this page remove the users from the board. The page

has a dropdown of users that are added to the board. Also the page has a delete button using which the board can be deleted.

Edit task page

Form this page task details like task owner, due date can be edited. Task name and task board can’t be edited because this can cause ambiguity. Also task can be deleted from this page.