Vlad Predovic

CS 496

Fall 2015

Final Project

http://web.engr.oregonstate.edu/~predoviv/CS496FinalVideo.mp4

LINK TO MY APP VIDEO

Contents

An explanation of the URI structure used to access resources

URI structure and HTTP verbs:

http://proj2-1095.appspot.com/studyProcedures

GET: Request all existing procedure entities

POST: Provide data for the creation of new entities

http://proj2-1095.appspot.com/TimeStudies

GET: Request all existing Time Study entries along with associated procedure keys.

POST: Provide data for the creation of a new TimeStudy. Optionally procedure keys to become associated with said TimeStudy

PUT: Provide data to add procedures to an existing TimeStudy.

DELETE: Used to delete existing time studies. Deletes all existing TimeStudy instances of the study as well.

http://proj2-1095.appspot.com/TSinstance

GET: Request all existing instances of Time Studies. Additionally, if used from a mobile app, only **instances created by the current user** will be displayed.

POST: Provide data for the creation of a new instance of an existing TimeStudy. If the TimeStudy does not exist, instance will not be created. When the new instance is created, new procedure instances are created as well, completing the implementation a new 'TimeStudy sheet' This allows the user to form multiple studies on the same topic.

http://proj2-1095.appspot.com/UserAuth

POST: Used for user verification or registration

Note on helpful web application:

GET procedures are also in place to display existing entries on a web page as well as the mobile app interface. At this point in the application's development a user could easily benefit from having all the

entities displayed on a computer screen as opposed to a small mobile device. As of right now, a user-friendly interface for scanning and searching through entities has not been developed for the mobile application. Therefore it is beneficial to be able to scan through a list of entries, especially under Time Studies, in order to access the associated instances and procedures.

Constraints met for a Restful API:

Statelessness: In the body of any request made all the parameters and data are sent to the server needed to generate a response.

Client-server: Concerns of client and server are separated. The server is able to interface with multiple clients, while the client does not have any need to see the inner-workings of the server. Most of this is achieved through use of the google-app engine.

Stateless: No sessions, all input by the client has all information provided to do the necessary tasks. Changes in resources are saved, but cannot be tied to the client.

Layered System: Client has no need to understand the different layers of the system. All functions interact with server to complete client's task without his knowledge of the interaction.

Uniform Interface: The client can only request JSON representations at the moment and all representations on the API live site are in text/html format. Although caching was attempted (see timestudy.py) it does not have a use for my server use and the no-cache default setting is the most to my advantage as up to date data is very important. Future use would be setting a cache to final product images for my TimeStudy NDB model.

Explanation of Entity Relations

TimeStudy: Declaration of existing studies, the associated procedure keys, and a picture for visualization of the product being studied. All associated procedures have an iteration number of 0. Procedures with an iteration number of 0 are considered the root procedure, from which all other iterations are copied (*see TSInstance)*.

StudyProcedures:

The timed procedures that when combined in a specific manner make up a study. These have an iteration number and timestamp of creation that correspond directly with the TSInstance they are associated with.

```
12  class TimeStudy(ndb.Model):
13      name = ndb.StringProperty(required = True)
14      procedures = ndb.KeyProperty(repeated = True)
15      endFigure = ndb.BlobKeyProperty()
16      #dictionary overrides
17      def to_dict(self):
18          d = super(TimeStudy, self).to_dict()
19          d['procedures'] = [m.id() for m in d['procedures']]
20          return d
21
22  class TSInstance(ndb.Model):
23      TSname = ndb.StringProperty(required = True)
24      user = ndb.StringProperty(required = True)
25      procedures = ndb.KeyProperty(repeated = True)
26      datetimeStarted = ndb.DateTimeProperty(auto_now_add = True)
27      iteration =  ndb.IntegerProperty(required = False)
28      location = ndb.StringProperty(required = False)
29
30  class StudyProcedures(ndb.Model):
31      Pname = ndb.StringProperty(required = True)
32      startTime = ndb.TimeProperty(auto_now = True)
33      endTime = ndb.TimeProperty(auto_now = True)
34      datetimeStarted = ndb.DateTimeProperty(auto_now_add = True)
35      iteration =  ndb.IntegerProperty(required = True)  #The TSIns
36
37
38  class appUsers(ndb.Model):
39      userName = ndb.StringProperty(required = True)
40      passWord = ndb.StringProperty(required = True)
41
```

TSInstance:

The real-time instance of a TimeStudy. Each instance is a copy of the main TimeStudy object with unique StudyProcedures associated directly to it through the creation timestamp and iteration. The timestamp is identical because both instance and procedure entities are inserted into the database at the same time. The location attribute is automatically updated when you run the instance using the native phone applications. The idea behind this is that it will be important for employees to know where the data is coming from when conducting studies in multi-factory companies.

appUsers:

Simple verification for the user. Registering and logging in is done through the initial view on the mobile application. Each user has a set of unique TimeStudy instances which he has ran and can view privately. The *user* property of the TSInstance class is a mirror of the *userName* property of the appUsers class. Every userName must be unique.

User-Verification and Implementation:

The account system allows for registration or login from the mobile device. When the user enters registration information, the userName is compared with the database for any matches, If none are found, the account is stored with the given userName and a cryptographically hashed representation of the password. An example is shown below.

```
newUser = db_models.appUsers()
Uname = self.request.get('userName', default_value = None)
Upass = hashlib.md5("self.request.get('passWord', default_value = None)").hexdigest()
choice = self.request.get('choice', default_value = None)
existCheck = db_models.appUsers.query(db_models.appUsers.userName == Uname).fetch()
logCheck = db_models.appUsers.query(db_models.appUsers.userName == Uname and db_models
```

```
appUsers(key=Key('appUsers', 5655612935372800), passWord=u'9b1ed222715617a001b920d8b8e502f5', userName=u'test')
```

When a user logs in, the input is compared with the database. If no match is found for the username and password, the user is rejected. Otherwise, the user is presented with a link to enter the application.