p4_Conference_APIs

This project, the fourth in Udacity's Full Stack Web Developer Nanodegree, focuses on using Google Cloud Platform to develop a scalable web app, and it's built with Google App Engine (Python), Google NDB Datastore, and Google Cloud Endpoints.

You can check out the demo web page at https://p4conference.appspot.com. Currently, the demo page does not support all functionality. You can find out all APIs here.

This cloud-based API server lets you organize conferences and sessions, and currently has following functionality:

- User authentication.
- Create, read, update conferences.
- Register/ unregister for conferences.
- Create and read speakers for sessions.
- Create and read sessions for conferences (only the organizer of the conference can create its sessions.)
- Add / remove sessions to user's wishlist.
- Query for sessions and conferences.

Project Detail

Task 1: Database Design

class **Conference**(ndb.Model):
    """ Conference -- Conference object """
    name   = ndb.StringProperty(required=True)
    description   = ndb.StringProperty()
    topics   = ndb.StringProperty(repeated=True)
    city   = ndb.StringProperty()
    startDate   = ndb.DateProperty()
    endDate   = ndb.DateProperty()
    maxAttendees   = ndb.IntegerProperty()
    month   = ndb.IntegerProperty() - is set from the month of startDate.
    organizerUserId   = ndb.StringProperty() - is set from the Id of the login user.
    seatsAvailable   = ndb.IntegerProperty() - is set from maxAttendees and is reduced when someone registers for the conference.

class **Session**(ndb.Model):
    """ Session -- Session object """
    name   = ndb.StringProperty(required=True)
    highlights   = ndb.StringProperty()
    typeOfSession   = ndb.StringProperty()
    date   = ndb.DateTimeProperty()
    startTime   = ndb.DateTimeProperty()
    endTime   = ndb.DateTimeProperty()

```
    location            = ndb.StringProperty()
    speaker             = ndb.KeyProperty(kind='Speaker', required=True)
```

I do not use the *duration* property as the course suggested, because if we store it as a string mixing of hours and minutes without any required standard, we can not use it for further purpose. My *duration* function can calculate and change the duration to the easy readable format. To store *startTime* and *endTime* in database has its advantage while we want to check the integrity of data, for example, a speaker can not have overlapped times of his or her presentation.

The one to many relationship between **Speaker** and **Session** is implemented by *speaker* and *sessions* KeyProperties in Session and Speaker classes.

```
class Speaker(ndb.Model):
    """ Speaker -- Speaker object """
    name          = ndb.StringProperty(required=True)
    phones         = ndb.StringProperty(repeated=True)
    emails        = ndb.StringProperty(repeated=True)
    website        = ndb.StringProperty()
    company         = ndb.StringProperty()
    sessions       = ndb.KeyProperty(kind='Session', repeated=True)

class Profile(ndb.Model):
    """ Profile -- User profile object """
    displayName = ndb.StringProperty()
    mainEmail = ndb.StringProperty()
    teeShirtSize = ndb.StringProperty(default='NOT_SPECIFIED')
    conferenceKeysToAttend = ndb.StringProperty(repeated=True)
    wishlistOfSessionKeys = ndb.StringProperty(repeated=True)
```

conferenceKeysToAttend: A user can register/ unregister for a conference. This list will contain the keys of registered conferences.

wishlistOfSessionKeys: A user can mark/ unmark for a session. This list will contain the keys of marked sessions.

Task 2: Add Sessions to User Wishlist

Users are able to mark some sessions that they are interested in and retrieve their own current wishlist.

- **addSessionToWishlist(SessionKey) –** add/ remove the session to/ from the user's list of sessions they are interested in attending by calling the auxiliary function *_addSessionToWishlist* with the argument set to True or False appropriately.

- **getSessionsInWishlist()** -- query for all the sessions that the user is interested in

Task 3: Work on indexes and queries

Instead of creating some predefined queries like *filterPlayground* method for conference, the *queryConferences* method is implemented to be more flexible. It can have queries with following fields:

NAME, CITY, TOPIC, MONTH, MAX_ATTENDEES, SEATS_AVAILABLE, START_DATE, END_DATE, using these operators: EQ, GT, GTEQ, LT, LTEQ, NE.

Similarly, *queryConferences* method is for querying sessions with the following fields:

NAME, SPEAKER, TYPE_OF_SESSION, DATE, START_TIME, END_TIME, LOCATION, using these operators: EQ, GT, GTEQ, LT, LTEQ, NE.

Create indexes

Indexes are created automatically in app.yaml file when some new types of query are made. However, the Datastore rejects some combinations of filters and orders. If an inequality filter is used, the first sort order must specify the same property as the filter. Also, Datastore do not allow more than one inequality filter. So there are two auxiliary methods for querying sessions and conferences:

_testAndFormatFilters:
. Check if the inequality filtering exists, the sessions/ conferences objects must be ordered by that property before ordering by name.
. Reject the second inequality filter for another property.

_setFilters:
. Implement some necessary changes of types for property comparison (key, date, int)
. Equal filter for date type must be changed into two inequal filters ,
e.g. date1==date2 is equivalent to (date1 >= date2) and (date1 < date2 + 24 hours)

Two additional queries

```
def additionalQuery1(self, request):
    """ Query for spare time intervals for a given speaker in a given month of a year. """

        This is useful for future scheduling for that speaker.

def additionalQuery2(self, request):
    """ Query for all the sessions of the seat available conferences in a given period of time . """

        Having some spare time, a user might want to know if he or she can choose some session to join.
```

Solve the following query related problem

“Let’s say that you don't like workshops and you don't like sessions after 7 pm. How would you handle a query for all non-workshop sessions before 7 pm? What is the problem for implementing this query? What ways to solve it did you think of?”

The solution is equivalent to:
q = Session.query(ndb.AND((Session.startTime < request.startTime),
(Session.typeOfSession != request.typeOfSession)))

but there are more than one inequality filter for more than one property,
which is rejected by Datastore. So we have to solve by python code:

```
sessionsTime = Session.query(Session.startTime < request.startTime)

sessions = []
for session in sessionsTime:
    if session.typeOfSession != request.typeOfSession:
        sessions.append(session)
```

In general, we can avoid the query with more than one inequality filter by modifying the data model. We can do that whenever we assume some query may be common.

For example, all the sessions started after 7pm are late sessions, we add the lateSession property to the class Session, and we update the lateSession property to 'True' whenever we create a late session. Then the query will be:

```
if request.startTime == datetime.strptime("19:00:00", "%H:%M:%S"):
    sessions = Session.query(
        ndb.AND(
            Session.lateSession == True,
            Session.typeOfSession != request.typeOfSession
        )
    )
```

See the method: def **queryProblem**(self, request)

Task 4: Add a Task

The featured speaker is the one who speaks for more than one session in the same conference.

Whenever we create a new session, we add a task setFeaturedSpeaker to the task queue because this task is not required to be done immediately. This task then check if the speaker of that session has another sessions in that conference, if so the speaker will be added to memcache with the key FEATURED_SPEAKER.

Products

- App Engine

Language

- Python

APIs

- Google Cloud Endpoints

Setup Instructions

1. Register in Google Developer Console**,** create a new project.

2. In Credentials menu, create new Client ID and set the redirect URIs like that:

   https://**your_projectID**.appspot.com/oauth2callback
   http://localhost:8080/oauth2callback

3. Clone or download this project.

4. Update the value of **application** in app.yaml to your**_projectID**.

5. Update the value of **WEB_CLIENT_ID** in settings.py to the **Client ID**.

6. Update the value of **CLIENT_ID** in static/js/app.js to the **Client ID**.

7. Run at local:

   Run: "**dep_appserver.py .**" in your working folder in your terminal.
   Go to:
   localhost:8080/ (conference app with some limited UI)
   localhost:8000/ (admin server with Datastore Viewer)
   localhost:8080/_ah/api/explorer/ (APIs)

8. Deploy on Google server:

   Run: "**appcfg.py --oauth2 update .**" in your working folder in your terminal.
   Go to:
   https://**your_projectID**.appspot.com/ (conference app with some limited UI).

   https://apis-explorer.appspot.com/apis-explorer/?
   base=https://**your_projectID**.appspot.com/_ah/api#p/conference/v1/ (APIs)

   You can see the database in Google Developers Console (go to Storage/ Cloud DataStore/ Query).

In either cases, you can add new entities to the database manually or via the APIs functions in the program, and you can see all of them in Datastore Viewer.

- End -