Write an “AppEngine Standard” App

Description

For this project you are going to implement and deploy a basic AppEngine (Standard, not Flex) application that keeps track of (and counts down to) upcoming events. It will have the following characteristics (note that if you are adventurous, you can use whatever language or serverless provider you like, but you might have less help):

- Events are displayed in ascending order of “closeness to today”, ignoring non-repeating events in the past.
- Events stored in Cloud Datastore (via ndb if you’re using Python)
    - name
    - (partial - missing year means repeats yearly for bonus points) date
- The main page is all static, with JavaScript[1] doing dynamic requests.[2]
- Users can add new events from the web page (and optionally delete them).[3]

You may use any language you like. Python is a solid and popular choice, but language will not be dictated in this course. Pick what you are comfortable working with, change later if you don’t like it.

---

[1] You **can** use a framework like Angular or React if you want, but 1) you’ll be on your own, and 2) you **cannot** use a framework in a way that blurs the distinction between server and client, as many try to do. Keep that part crisp. Everything else is up to you.

[2] This is not an arbitrary requirement. First, it’s how many apps are done these days, and second, it paves the way for some important stuff when we augment this in later assignments.

[3] We need to have some way to mutate our data, after all! And we’ll be talking about some of the ramifications of using POST to modify potentially sensitive data and how to get around them.

Grading Rubric

Grading labs is always tricky business. Every lab has a minimum standard that, if you meet it, it will guarantee at least a B- on that lab. Other points fit on top of that standard. If the minimum standard is not met, the grade assigned is discretionary based on how much was completed.

Minimum Requirements

- All HTML and JavaScript is served statically (not generated from a template).
- User can
    - Access the web page and immediately see stored events with ETAs
    - Add a new event from that page and see it appear in the events list
- GET /events returns JSON containing event information
- POST /event accepts JSON for an event, pushes it into the database, and returns a response indicating success or error, with the new event's ID.
- Code: all code is original work and free of debug logic.

It is possible to implement these in slightly different ways. Especially on this first lab, there will be some flexibility allowed in implementation (e.g., how errors are returned, and what additional data comes in the response for a POST or DELETE request).

Meeting these minimum requirements guarantees a score of 80 points on the lab.

Additional Points

Additional points can be earned, up to a maximum of 100:

- 5 pts: Code well-documented (e.g., if using Python you use docstrings to specify parameters and return values) and clean
- 5 pts: DELETE /event/<event_id> deletes an event from the database or returns an error (including the interface to trigger it)
- 5 pts: Yearless dates work, showing time to next occurrence of a matching date (e.g., 03-01 means "every March 1st")
- 5 pts: ETA values change every second so they're counting down constantly.
- 5 pts: Automatic DELETE trigger for dates in the past (like a cron job that checks for old dates and deletes them)

In other words, you don't have to do all of these things to get full credit, and if you decide to do them all, it reduces the risk of partial credit keeping you from a score of 100.

The Siren's Serverless Call

Why AppEngine, and not something like AWS Lambda or Google Cloud Functions, or something else entirely? Because

- Google AppEngine Standard includes storage.
- Free tier stays free longer and with heavier use.
- Auth is included (for later labs) with minimal fuss.
- It's a good thing to know how to use in general.

Lambda-style serverless[4] approaches are relatively new, and somewhat overhyped - there is a cost to austerity, and it often shows up in the form of increased developer complexity, particularly at first while you figure out how things fit together (for example, you need to separately provision storage/pub-sub/etc., and that is a research project all its own).

> If you are motivated and capable, I won't stop you from using another technology. Just be prepared to do a little extra work with a little less help if you do. I will accept a project developed on any underlying PaaS or even IaaS so long as your code implements the required characteristics. If you do decide to go your own way and find it to be better in any way, please share what you did: future classes may benefit.

There are several moving parts in this lab, so it's going to be important to start early. The basic steps you'll need to take are

1. Install the AppEngine SDK.
2. Start a new project (with an app.yaml file) the exports necessary static resources.
3. Write an index.html that gets JSON events, displayed with computed countdowns.
4. Add a submission form for new events to the index.html file.
5. Create a GET routine to produce event JSON when asked.
6. Create a POST routine to create new events when asked.
7. Optional: create a way to delete events.
8. Deploy your app and test it out.

Fortunately, none of these steps is difficult on its own, but they do have to be done mostly in order, so it's best to get as many early steps out of the way as possible, as soon as you can, so that you can focus on getting the communication working between the browser and the server. That's where the real work is, particularly if this is at all new to you.

[4] AppEngine is also "serverless" because you don't think about things in terms of servers; you think of them in terms of events coming into your functions. AppEngine happens to be just enough richer that the barrier to entry is far lower than with functional approaches.

To help with those new to all of this, a few of these tasks are broken out below. If you have done something like this before, already know enough HTML and JavaScript to be dangerous, and don't mind digging into online tutorials, you can ignore the rest of the content here.

Getting Structured Data Into a Static Page

What we're building is known in industry as a "one-page app". It's a very simple one, but it has all of the right characteristics: it's a single static page (with its JavaScript dependencies), and it updates by making background requests and rewriting itself as needed.

That's how this assignment will work. You'll create index.html and potentially some .js files that never change and are downloaded all at once when a user comes to your site. After that, DOM events will trigger your JavaScript code and potentially cause background requests to produce data that changes how your page looks. Let's assume for just a moment that we **already have** a service that has an /events endpoint. We can issue an HTTP GET request to it using code like this (the *getJSON* function is provided here as a convenience if you want it):

```
<html>
<head>
<title>My Lovely One-Page App</title>
<script>
function getJSON(url) {
  return new Promise((resolve, reject) => {
    let xhr = new XMLHttpRequest();
    xhr.open('GET', url);
    xhr.responseType = 'json';
    xhr.onload = () => {
      if (xhr.status >= 200 && xhr.status < 300) {
        resolve({status: xhr.status, data: xhr.response});
      } else {
        reject({status: xhr.status, data: xhr.responseText});
      }
    };
    xhr.onerror = () => {
      reject({status: xhr.status, data: xhr.responseText});
    };
    xhr.send();
  });
}

document.addEventListener('DOMContentLoaded', () => {
```

```
  getJSON('/events')
  .then(({status, data}) => {
    // Use the *data* argument to change what we see on the page.
    // It will look something like this:
    // {
    //   "events": [
    //     {"name": "Grandma's Birthday", "date": "08-05"},
    //     {"name": "Independence Day", "date": "07-04"}
    //   ]
    // }

    // There are better ways, but this is illustrative of the concept:
    let html = '';
    for (let event of data.events) {
      html += `${event.name}: ${event.date}<br>`;
    }
    document.getElementById('events').innerHTML = html;
  })
  .catch(({status, data}) => {
    // Display an error.
    document.getElementById('events').innerHTML = 'ERROR: ' +
      JSON.stringify(data);
  });
});
</script>
</head>
<body>
<div id="events"></div>
</body>
</html>
```

When *getJSON* is called, it returns a promise that either resolves to a status and returned data (in the *.then* handler) or is rejected with a status and data (in the *.catch* handler).

You can see above how the status and data are unpacked in both *then* and *catch*, and how they are displayed in the browser window. The way these are displayed and handled

1. Is not terribly efficient,
2. Doesn't allow for customization,
3. Isn't templatized, and therefore isn't common (nor a best practice), and
4. Doesn't compute the time until the event, but
5. Is straightforward to understand and illustrates the important stuff.

In other words, no, you probably wouldn't set *innerHTML* in your own code, and yes, you will need to do some computations on the date, but this is good enough as an example of the fetch-and-display concept in a static web app. Note that nowhere is the server sending down HTML that *changes based on the request*. Instead it always sends down the *same* HTML and JavaScript, and then those go ask for more things that dynamically alter the page contents. That's what we're after.

Note that once the *getjSON* function is written, it can just be reused. You may need to tweak it a bit, or copy it for mutating actions, but that big chunk of space it takes up won't be repeated for every little thing you do. I wrote it using Promises because that makes it work with async, in case anyone cares. There are definitely simpler implementations that just involve passing functions around.

Serving Static Content and Dynamic Data Structures

Your server will need to be implemented using AppEngine, as mentioned earlier. You can choose any language you like for this. My favorite is Go, for multiple reasons, but it's more common to start with Python.

Whatever language you use, you need to minimally implement the following endpoints. For purely static files, you can set up *app.yaml* to serve them for you and skip the code part.

GET	/	produce index.html contents
GET	/events	produce event JSON
POST	/event	add a new event (sends JSON)
POST	/delete	(optional) delete an existing event (figure out how to identify it!)

If we leave out the optional (for now) deletion endpoint, that leaves a minimum of two functions you need to write in the server: one for getting a list of events, and one for adding an event.

Find a suitable tutorial for AppEngine that is in your language, and it will be sure to cover at least those two things.

Cloud Data Store

You will want to store your events in a database of sorts so that you can recall them between stateless HTTP requests. That will mean, in the free-tier AppEngine environment, using Cloud Data Store. If you are using Python, you will define and access your models via ndb. If you are using other languages, there are similar libraries available that you will need to learn.

Structure your database however you want. This is a small project, so do it however you like, but think about what would happen if you suddenly had to store thousands of events. Would you do it the same way, or would you change it? If you would change it, it might be a good idea to decide either on the limits you would impose on the app's functionality, or on how you would migrate from a less scalable approach to a more scalable approach.

When you add an event, store it in the data store. The order doesn't really matter, since part of the assignment is to sort by event "closeness" to today anyway.