Write an “AppEngine Standard” App

[**Description**](#_y3t8gkwrrudo) **1**

[**Grading Rubric**](#_8gkkdn4cyz9w) **2**

[Minimum Requirements](#_u7dsxh35efco) 2

[Additional Points](#_lnvzh94j8nln) 2

[**The Siren’s Serverless Call**](#_h2x2gs5vdd7k) **3**

[**Setting Up**](#_hxpy3h10gdez) **4**

[**Getting Structured Data Into a Static Page**](#_d3z1dkiyp2ze) **5**

[Serving Static Content and Dynamic Data Structures](#_de1t0m2y6rji) 7

[**Cloud Datastore**](#_weuh29o6w75b) **8**

[**Hopefully Helpful Tidbits**](#_83l9g5arik42) **9**

[Cloud Datastore from Python](#_1d0kpe2d0o3m) 9

[Flask and JSON](#_barve1t8brx7) 10

[Date Handling](#_5y3njl9ijxt7) 10

[Form Submission](#_fvmygz4yzijg) 11

# 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 “soonest first” order, ignoring non-repeating events in the past.
* Events are stored in Cloud Datastore
  + name
  + date (bonus: missing year means “repeats yearly”)
* The main page is all static, with JavaScript[[1]](#footnote-0) doing dynamic requests.[[2]](#footnote-1)
* Users can add new events from the web page (and optionally delete them).[[3]](#footnote-2)

|  |
| --- |
| You may use any language you like. Python 3 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. |

# Grading Rubric

Grading labs is always tricky business. Every lab has a minimum standard that, if you meet 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 dynamically on the server).
* A 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 in this lab, there will be some flexibility allowed in the 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 is well-documented and clean (e.g., if using Python you use docstrings to specify parameters and return values)
* 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.
* The 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]](#footnote-3) approaches are still more cumbersome to set up than I would like, and are 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—that is the sole reason I don’t push AWS Lambda instead of AppEngine: setting things up is more manual there). Setting these things up also requires a solid mental model of how web technologies fit together in a real system, but that’s *exactly what we’re trying to learn right now*, so we want to not get ahead of ourselves.

|  |
| --- |
| 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 FaaS, PaaS, or even IaaS[[5]](#footnote-4) 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 (index.html is one such static resource, scripts, images, style sheets, etc. are, too).
3. Write a simple “Hello World” application and run it locally.
   1. This will involve authenticating using the “gcloud” command line and doing some configuration there. Find a tutorial online that can help with this.
4. Deploy it and make sure you can access it on the web.
   1. This will also involve using the “gcloud” command.
5. Write an index.html that gets JSON events, displayed with computed countdowns.
6. Add a submission form for new events to the index.html file.
7. Create a GET routine to return event JSON when asked.
8. Create a POST routine to create new events when asked.
9. Optional: create a way to delete events in a DELETE routine.
10. Deploy your app and test it out (actually, do this every time you want to test something).

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 interesting work is, particularly if this is at all new to you.

To help with those new to all of this, a few of these tasks are broken out below. This will be familiar if you have done the previous lab.

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.

# Setting Up

This section assumes you are using AppEngine Standard[[6]](#footnote-5) with the default settings (Cloud Datastore, Python 3, Flask).

The steps you will complete for *project* setup are basically these:

* Go to <https://console.cloud.google.com> and create a new Project.
* Use the upper-left menu to get to the [AppEngine](https://console.cloud.google.com/appengine) page.
* If you’re in your new project, it will show you a “Create Application” button. Click that.
* Choose a region. We’re in the us-east-1 region, so that will give the best latency from JHU.
* I choose “Python”, “Standard” on the next page, but you can pick another language if you like. I would **not** choose “Flexible” unless you really know what you are doing.
* Note that by default the documentation it leads you to is for Python 2. The Python 3 documentation can be found here: <https://cloud.google.com/appengine/docs/standard/python3/>
* Follow instructions to download the cloud SDK and get it set up.
* Do a “gcloud auth login” so that deployment can work, and your dev server can hit the database (the local development datastore is apparently deprecated).

Eventually (maybe now!) you will want to go to your project’s “service accounts” and generate a JSON key file for the default appengine service account. You can then point to that file in the [GOOGLE\_APPLICATION\_CREDENTIALS](https://cloud.google.com/docs/authentication/production) environment variable to make things work from your local system. I use a small bash script to point to that when running my local Flask app so that it doesn’t pollute my system with random junk. For example:

|  |
| --- |
| #!/bin/bash  export GOOGLE\_APPLICATION\_CREDENTIALS="$HOME/.config/gcloud/appengine-token.json" export GAE\_ENV="localdev" python main.py |

Once you are up and running, you will need to create your HTML file that you want served. You can do this in a couple of ways:

1. Static: you can set up your app.yaml handler for “/” to be a static directory, and place an “index.html” file in the directory you specify, or
2. Dynamic: you can write a little Python/Flask application that serves up the index.html file and specify that as the script.

There are numerous “Hello, World!” AppEngine tutorials online. I would encourage you to look at these to get started.

# 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 an *index.html* file (and potentially some .js files, if you want to split those out) that never changes and is downloaded all at once when a user comes to your site. After that, DOM events will trigger your embedded JavaScript code and get data in the background, using it to change how your page looks.

A useful technique for getting started when there’s a lot to do is to assume that some of the work is already done. Let’s do that. Let’s assume that the server is already built, and all we’re doing is building the web site.

Specifically, let’s assume for just a moment that we **already have** a service that has an endpoint ***/events***. We can issue an HTTP GET request to it using code like this[[7]](#footnote-6):

|  |
| --- |
| <html> <head> <title>My Lovely One-Page App</title> <script> function reqJSON(method, url, data) {  return new Promise((resolve, reject) => {  let xhr = new XMLHttpRequest();  xhr.open(method, 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.response});  }  };  xhr.onerror = () => {  reject({status: xhr.status, data: xhr.response});  };  xhr.send(data);  }); }  document.addEventListener('DOMContentLoaded', () => {  reqJSON('GET', '/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 *reqJSON* 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). If you aren’t familiar with how promises work, the main thing to know is that you pass a function into .then and .catch that gets called if things are successful or fail, respectively.

In the example above, we “unpack” the object passed into the “then” handler, getting its “status” and “data” values into two variables. We could also have done something like this:

|  |
| --- |
| reqJSON('GET', '/events').then(obj => {  // do stuff with obj.status and obj.data })*;* |

You can see in the long listing 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

* Is not terribly efficient,
* Doesn’t allow for customization,
* Isn’t templatized, and therefore isn’t common (nor a best practice), and
* Doesn’t compute the time until the event, but
* 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; that’s essentially how a one-page app works.

Note that once the *reqJSON* function is written, it can just be reused wherever you need it, including to issue POST and DELETE requests. 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.

### 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 specified in JSON |
| POST | /delete | (optional) delete an existing event (figure out how to identify it!) |

If we leave out the optional 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. The server also needs to return the contents of *index.html* when asked for the root “/” or “/index.html” endpoint.

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

# Cloud Datastore

You will want to store your events in a database of sorts so that you can recall them between stateless HTTP requests. Why a database? Because with platforms as a service (and functions as a service), *you don’t know how many copies of your code are running, and where*. That means you can’t rely on your server code being able to access files: you might hit a server on machine 1 with one request, then on machine 2 with a second request. If you just assume you can mutate files as your storage mechanism, you’ll be pretty severely disappointed. Thus, we use a centralized database.

For our *static* files, we can think of them as *part of the server*, so they’re okay. If you have static templates that you fill out and return, those are part of your deployment, so those end up on *all* of the machines where your service is running. It’s data that you *mutate* that you have to keep centralized in a database.

In the free-tier AppEngine environment, that means using Cloud Data Store. If you are using Python 3[[8]](#footnote-7), you will access your data via the [Cloud Datastore client library](https://cloud.google.com/appengine/docs/standard/python3/using-cloud-datastore). If you are using other languages, there are similar libraries available that you will need to learn.

Note that there is not, for Python 3, any “local” development data store. If you use Python 2, you can use ndb to access the datastore, and that **does** have a local development environment, but it’s an older runtime. It’s up to you what you use; we’ll focus on Python 3.

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.

# Hopefully Helpful Tidbits

Here are a few bits of important advice for the lab. They’re a bit sparse because a big part of the point of this class is to exercise our “look things up and figure things out” muscles, so it’s expected that you will use the resources available to you, including each other. Just don’t plagiarize: write your own code!

## Cloud Datastore from Python

First, make sure every entity you store has a parent key. That makes it much more likely that, if you push a new value, it comes back when you immediately query it again. Without a parent key, the entity might take some time getting there, which makes weird things happen when you submit events and then expect them to come right back. The following sets up a datastore client and a parent “root” key that can be used as the parent of all entities:

|  |
| --- |
| from google.cloud import datastore  app = Flask(\_\_name\_\_)  DS = datastore.Client() EVENT = 'Event' *# Name of the event table, can be anything you like.* ROOT = DS.key('Entities', 'root') *# Name of root key, can be anything.*  def put\_event(name, date\_str):  entity = datastore.Entity(key=DS.key(EVENT, parent=ROOT))  entity.update({'name': name, 'date': date\_str})  DS.put(entity) |

If you want to *query (read)* things, you will set the “ancestor” to be the root key (instead of “parent”) in the query or fetch statement.

|  |
| --- |
| for val in DS.query(kind=EVENT, ancestor=ROOT).fetch():  # do stuff with each value in the events table. |

Note that all of this will will access *the production database online*. One way to avoid data clashes between production and development is to choose the parent key based on the “GAE\_ENV” environment variable so that you can store test data separately from “running on AppEngine” data. This is up to you and 100% optional, but you should at least be aware of its existence. Here’s an example of the code.

|  |
| --- |
| if os.getenv('GAE\_ENV', '').startswith('standard'):  ROOT = DS.Key('Entities', 'root') else:  ROOT = DS.Key('Entities', 'dev') |

## Flask and JSON

When using Flask, you’ll be providing endpoints that handle and produce JSON. Flask provides the *jsonify* library that is really useful. You can also pull JSON from *request.json* when needed. Look at the Flask documentation, run through a tutorial or two, and it should be relatively straightforward.

Note again that one of the points of this course is to exercise your “look things up and figure things out” muscles, so the snippets here are purposefully a little sparse on information. Use whatever information sources you can, including each other, just write your own code!

## Date Handling

Assume that dates are in YYYY-MM-DD or MM-DD format. JavaScript can parse dates in that format using *new Date(datestr)*, but it is actually **not recommended** to use that mechanism (in the official documentation!) because it always assumes UTC if the timezone is not specified. That’s rather annoying and can be quite confusing. So don’t use the obvious thing. Sorry.

Instead, here are a couple of options:

* Use a regular expression (works well, maybe a little more complicated), or
* Split on hyphens and use the numbers in each part that comes back.

I opted for the second one for this example. Here’s a YYYY-MM-DD example:

|  |
| --- |
| const [y, m, d] = datestr.split('-'); const date = new Date(+y, m-1, +d); |

Note that this only really works properly because “07” is the same as “7” in octal, and “08” is obviously not octal so it is interpreted as decimal. Therefore, it *does work* for all dates we care about, since we are guaranteed to not go above two digits, and once we get to two digits we never have a leading zero.

*That said*, if you try to do this with years far in the past, it will *not work properly* because it will treat something like ‘0100’ not as year 100 AD, but as year 64 AD. Octal is fun.

A more correct approach to getting integer values from JavaScript would be this:

|  |
| --- |
| function parseDate(datestr) {  const [y, m, d] = datestr.split('-');  return new Date(Number.parseInt(y), Number.parseInt(m)-1, Number.parseInt(d)); } |

That gets you a time set to midnight of the given day (00:00:00, the very beginning of that day).

To get today’s date and the current time, you can just do *const now = new Date()*.

Note that, if you have to do sorting, you can easily treat a JavaScript date as a number by using the unary + operator. So, if you have the string “500”, you can make it act like a number by saying +”500”, for example.

With dates, you can do this to get the number of seconds from now until a given “date”:

|  |
| --- |
| let seconds = Math.floor((+date - new Date()) / 1000); |

The *new Date()* part doesn’t need a leading + because the subtraction operator also coerces the right side into a number.

Note that once you have this “distance in seconds”, you also know whether the date is in the future or the past by looking at the sign of the result.

Think about how, given the number of total seconds, you might create a nice countdown timer display with days, hours, minutes, and seconds remaining. You can use *Math.floor* as shown above, and you can also make use of the modulus operator %.

## Form Submission

To create form elements that allow you to create a new event, here is one approach. Note carefully the ***onsubmit*** attribute. Also note that newlines get converted to spaces; you need to use HTML tags to indicate a new line or paragraph if you want that:

|  |
| --- |
| <form onsubmit="return false">  Event name: <input type="text" id="nameInput">  date: <input type="text" id="dateInput">  <button onclick="createEvent()">create</button> </form> |

This will produce a form that you can fill out to create an element. You'll need to write the `createEvent` function, which will find the `nameInput` and `dateInput` nodes, get their values, package them into JSON, and POST to /event.

To find the date node and get its text, you might have code somewhere like this, in JavaScript:

|  |
| --- |
| const dateStr = document.getElementById('dateInput').value |

Again, note that the *onsubmit* for the form is important. Without that, your page will always be reloaded when the button is pressed, which makes debugging super hard (that bit me when I was writing up this example, and trying to figure \*that\* out while tired was... less successful than I would have liked).

That reminds me, you can make debugging easier even with “Preserve Log” checked in the network tab in the Chrome developer tools; that at least allows you to see what happened across requests.

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. [↑](#footnote-ref-0)
2. This is not an arbitrary requirement. First, it’s the way that many apps are done these days, and second, it paves the way for some important stuff when we augment this in later assignments. [↑](#footnote-ref-1)
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. [↑](#footnote-ref-2)
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. [↑](#footnote-ref-3)
5. Functions, Platform, or Infrastructure as a service. Lambda is a FaaS, AppEngine is a PaaS, and a virtual server that you have total control over would be an IaaS. [↑](#footnote-ref-4)
6. You can, as mentioned earlier, use AWS Lambda or other technologies, but the setup there is nontrivial (involving S3 for static storage, an API gateway, CORS-enabled web pages, and several IAM roles to manage, with various attached policies on top of it all), so if you want to do that, you should assume you are mostly on your own.

   I can help you with just about any technology you choose, but my main focus will be on helping people who are just starting out, which means I’ll be expecting most folks to use the path of least resistance, and that’s Google AppEngine at the moment. [↑](#footnote-ref-5)
7. The *reqJSON* function is provided here as a convenience if you want it. I use a promise-based implementation here, but that’s not required. If you know how to use the *XMLHttpRequest* you can just use it directly, for example. Or you can use the *fetch* API. [↑](#footnote-ref-6)
8. Python 2 users will often use a library called “ndb”. [↑](#footnote-ref-7)