**Report overview**

Now we had the hardware set up and the software working, we need a place to display all the results and give users an interface to our platform. We also need a channel for the administrator to monitor and control user access. All these needs can be achieved by a Web application.

In our case, we basically have 3 pages in our website, the first one is real-time dashboard which shows the real time data from different sensor unit. Different units are put into different tags. Second, we have our control panel, which shows all the devices with its description, status and switches. Third, the admin page which shows the login users and the input box for changing user status.

Based on the functions we discussed, we need to figure out the view-model-control for us system and decide tools for each of them.

For Python, there are lots of web frameworks, such as Django, web2py, Flask, Pyramid and App Engine. After doing some research, we decide to use App Engine. The reasons are, first it’s fully compatible to Google service which means we can integrate our “big query” services to it. Second, it has been documented very well with clear examples. Third, it’s scalable and it’s on a higher level, which means Google takes care of the infrastructures for our website such as server configuration and scalability.

After decided the tools and service we are going to use, we need to map the tool and service to our function definitions in order to make the project and coding process clear.

This report is organized as

-Basic Web structure introduction

-Map our desired functions to basic web structure

* Logic mapping
* Login mapping with App Engine

-Implement App Engine and deploy Web application

-Case study: demonstrate how to adjust examples to create desired components and functions

* Embed real-time plot to make a dashboard
* Make forms and exchange data based on it
* User accessible control
* Pass user input to Raspberry Pi(Gateway)

***Basic Web structure***

In a classic view-model-control model, **View** mainly focus on website layout, data display and gets user input from HTML forms. In other words, **View** means what our website looks like, which can be done by using HTML/CSS and some JavaScript. **Model** basically is database of our website. Different framework has different method to connect and process database, for App Engine, it uses Datastore which is a non-SQL database, we discuss its details in following chapter. **Control** takes care of URLs management; this means control decide how to jump between web pages.

You can see, we need to work on page layout (**View**), page connections (**Control**) and data process (**Model**). Furthermore, there are some overlaps between them but it is easy to manage.

***Map functions to basic Web structure under App Engine***

The basic App Engine project provides the similar structure based on the structure we discussed. Here is the basic App Engine project layout.

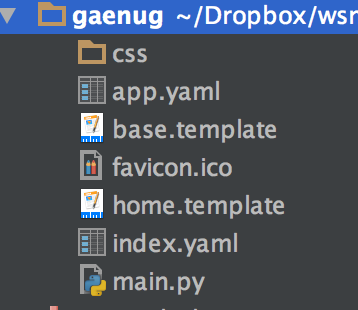


Fig 1 Basic App Engine project layout

We have a ‘app.yaml’ file which is control module, ‘base.template’ which is our basic layout, ‘main.py’ which is our integrate function which process URL jumps, data flow between pages and database.

***Details of each module***

**First, app.yaml**

When you create an App Engine project, there is a file called ‘app.yaml’ which is the configuration file. This file set up the running Python version, scalability settings, URL handlers and libraries.

The URL handlers is the key player for this file (Fig.1). As Fig.1 shows, URLs are assigned to different functions. In our case, we put all the functions in ‘main.py’, since our project is small.

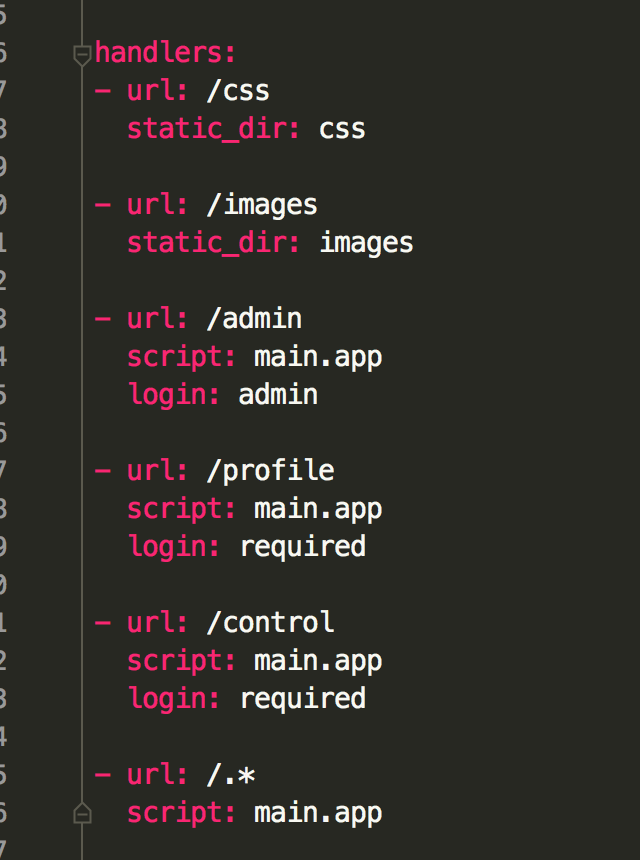


Fig 2 ‘app.yaml’ file layout

In other words, ‘app.yaml’ is our control module. In ‘app.yaml’ file, we also need to define libraries as shown in Fig. 2. There is a library called ‘webapp2’ which helps us manage page jumps.

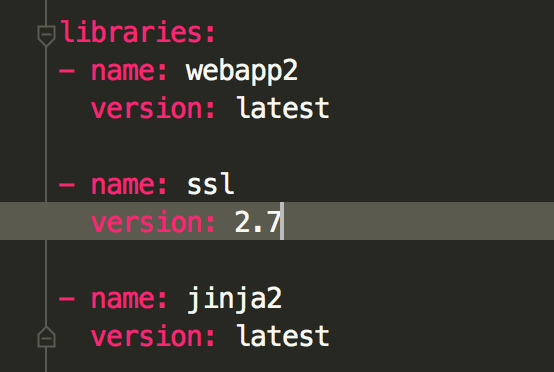


Fig 3 Libraries in ‘app.yaml’

Here’s a link of example of app.yaml. <https://cloud.google.com/appengine/docs/python/getting-started/handling-user-input-form>

We mentioned that ‘app.yaml’ link URLs to different python functions, these functions help us process data flow between pages and database, at the meantime it manage the connection between different URLs. Main.py is brain of each page.

**Second, main.py**

Here is a glance of the main.py.



Fig 4 main.py function layout.

The function displayed generates a ‘hello world’ page, but it demonstrates the idea of main.py and app.yaml very well.

The process between ‘app.yaml’ and ‘main.py’ is simple. When you put the ULR of our website, the browser brings you to the home page. Normally the URL ends up with ‘/’, since we declare ‘/’ as connector to ‘main.py’ in our ‘app.yaml’ file, and our browser goes to main.py to process. Browser finds out the webapp2 functions, and finds out that ‘/’ is equal to a class called ‘MainHandler’. The browser goes to the ‘MainHandler’, and it figures out that there’s only a “get” method. This “get” method only gives browser a ‘hello world’ sentence to print. The browser will print it and end the process.

All the requests on our website follows the same pattern. Browser checks up the ending of URL and compares it against ‘app.yaml’, then browser goes to specific python function. In the python function, browser finds out the relevant python class and processes it. After this process, the browser will display the certain information or sent form to our server. If the class only have a “get” method, the browser only displays information based on the template and the information we provide to it. If there are “get” and “post” method, the browser will display information as long as the page is called and send information to sever when some event is triggered such as a button is clicked. Below, we give you an example of a class has two methods.

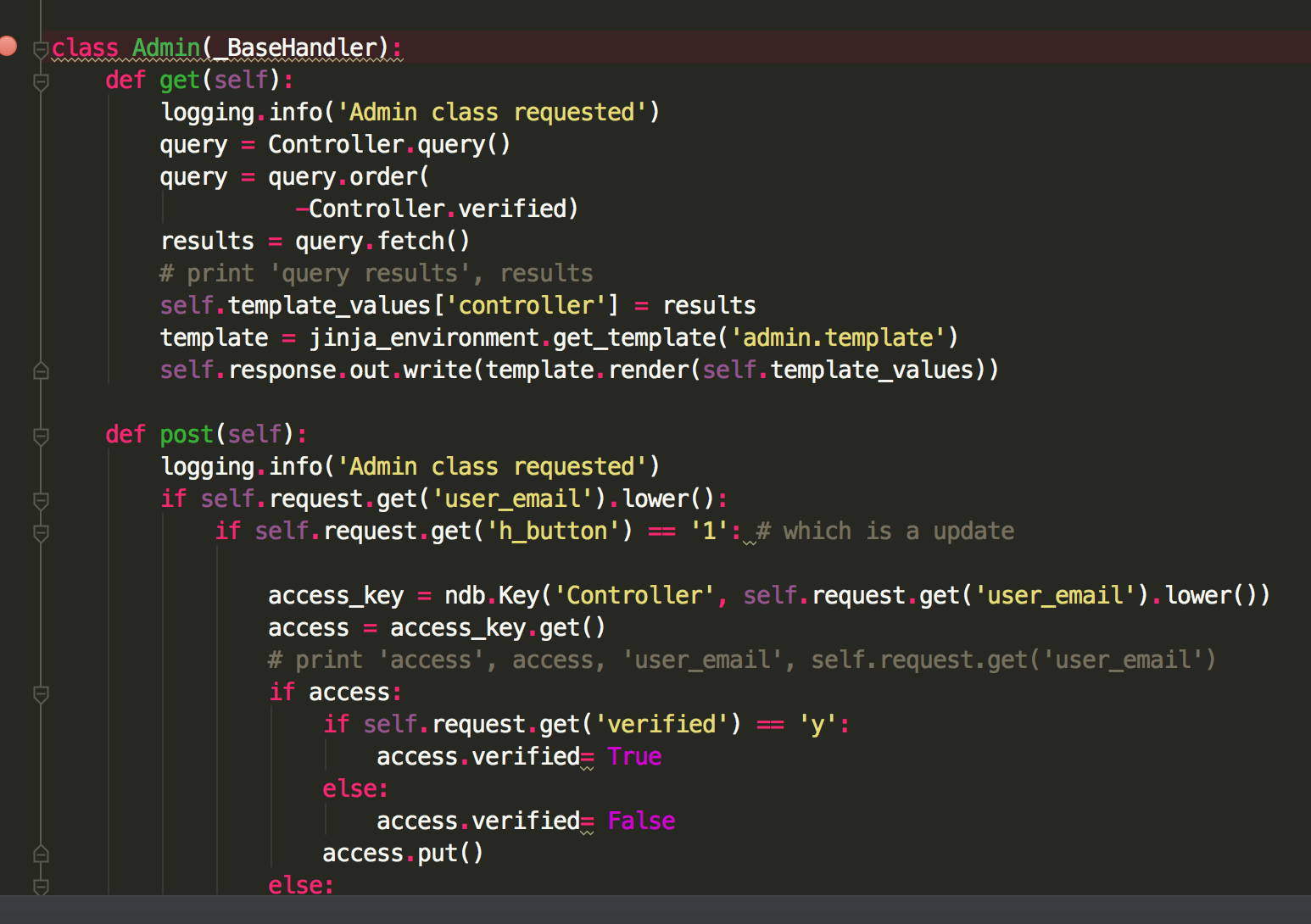


Fig 5 Class has get and post method.

**Third, template files**

In Fig.1, the class one defines “get” method and the method only prints a line on the page. This is not what we want. Normally, we want far more complex format and layouts. All these diversity formats and layouts are defined at template files. These template files are connected to each page at the page class. As shown in Fig.5, in the get method, we pass the data to ‘admin.template’ to display. Figure 6 shows how the template looks like.

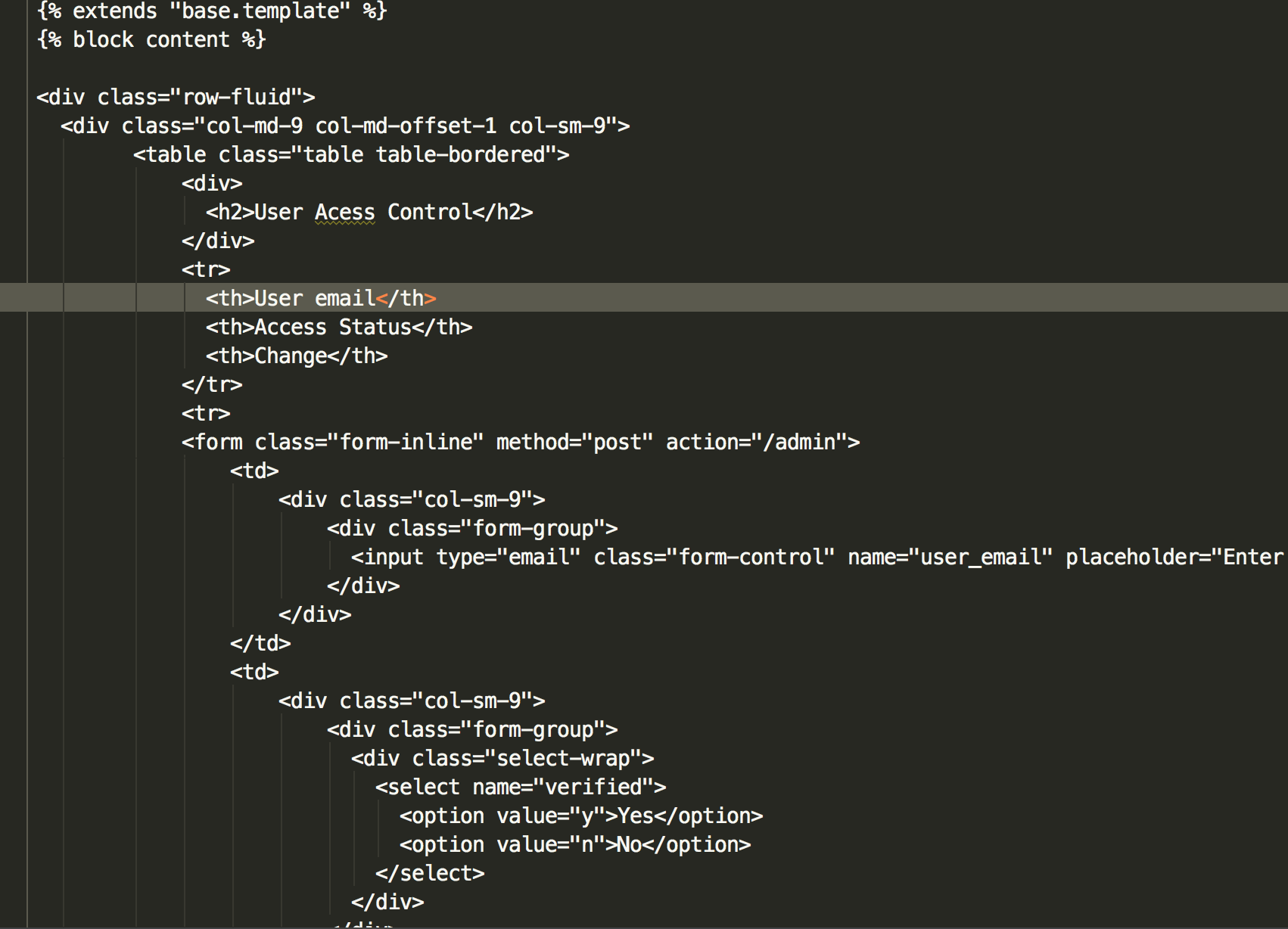


Fig 6 Template file layout

You can see that we use the HTML tags to layout our content, we use form tags to exchange data between page and server.

In order to make our page genetic we use

{% extends "base.template" %}  
{% block content %}

to inherit from the ‘base.template’ which defines the header, footer and JavaScript libraries.

So far, we figure out the View and Control module in our website and now let us move to the database/model module of our website.

**Finally, database -- Datastore in App Engine**

As mentioned, we are using the Datastore as database in our Web App. We need to introduce Datastore a little bit before we use it. Datastore is a Non-SQL database which means that it differs itself from SQL database in the way it describes relationships between data objects. Here’s a high-level comparison between Datastore and relational database concepts:

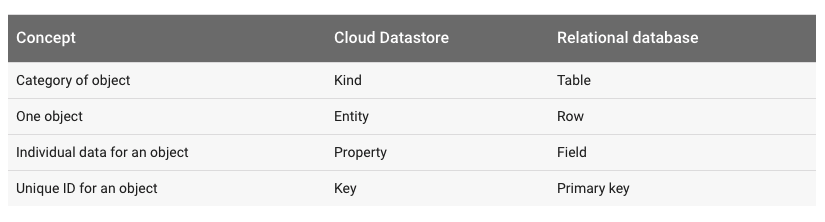


Fig 7 Comparison of Datastore and relational database concepts

These differences lead Datastore to a more intuitive way to store application data. In the view of implementation, every input can be viewed as an entity with properties and keys. Every entity can be queried by using language like SQL.

Google provides a python library to help us talk to Datastore, furthermore Datastore is integrated in App Engine then we don’t even need to connect it to our website manually. We just need to use python library operate directly. Here’s an example of using library to initialize the Datastore.

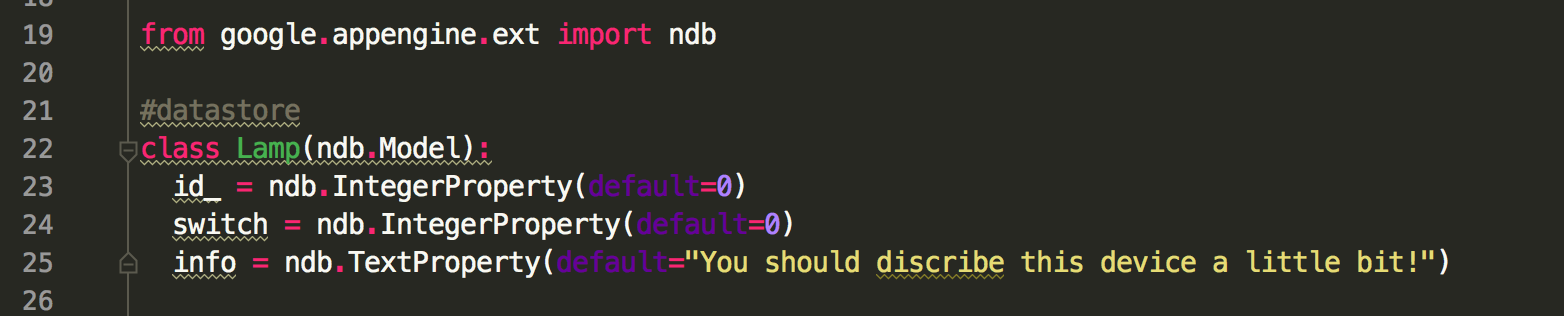


Fig 8 Use ‘ndb’ library to operate the Datastore

In this example, line 19 imports the library. Line 22 defines a class which can be viewed as defining a table. You should inherit the ndb.Model in order to do so. In this class, we define the properties of each entity. We define the key in page functions which will be showed in the following chapters.

Now we have walked through the web app backend logic, file layout and mapped the functions to specific file in App Engine. We will provide a model of this project and you can modify it to make you own web app. In other word, you can make you own web by modifying the templates we provide, and here’s a link from Google that gives you more details related to each parts we’ve talked about.

<https://cloud.google.com/appengine/docs/python/getting-started/handling-user-input-form>

***App Engine implementation***

We move to implementation of our web app. In this part, we focus on:

* API installation
* App Engine initialization
* Web deployment

**First, API installation**

For our project, we need to use Google Cloud Platform SDK again, which we have used for our big query project. You can follow the same steps to set it up. Here’s a link of installation.

<https://cloud.google.com/sdk/docs/>

**Second, API initialization**

In this step, we need to use our GCP account to create a project for our web engine which means you need to go to GCP account page to create a project and enable the App Engine for this project.

<https://cloud.google.com/appengine/docs/python/quickstart>

**Third, Web Deployment**

For deployment, we can deploy our web app locally for debug then move it to GCP. Here we recommend to use App Engine launcher instead of use command directly since the launcher gives you a GUI interface which is more friendly for beginners. Here’s the link for download.

<https://cloud.google.com/appengine/docs/python/download>

After you install it, it will look like this.

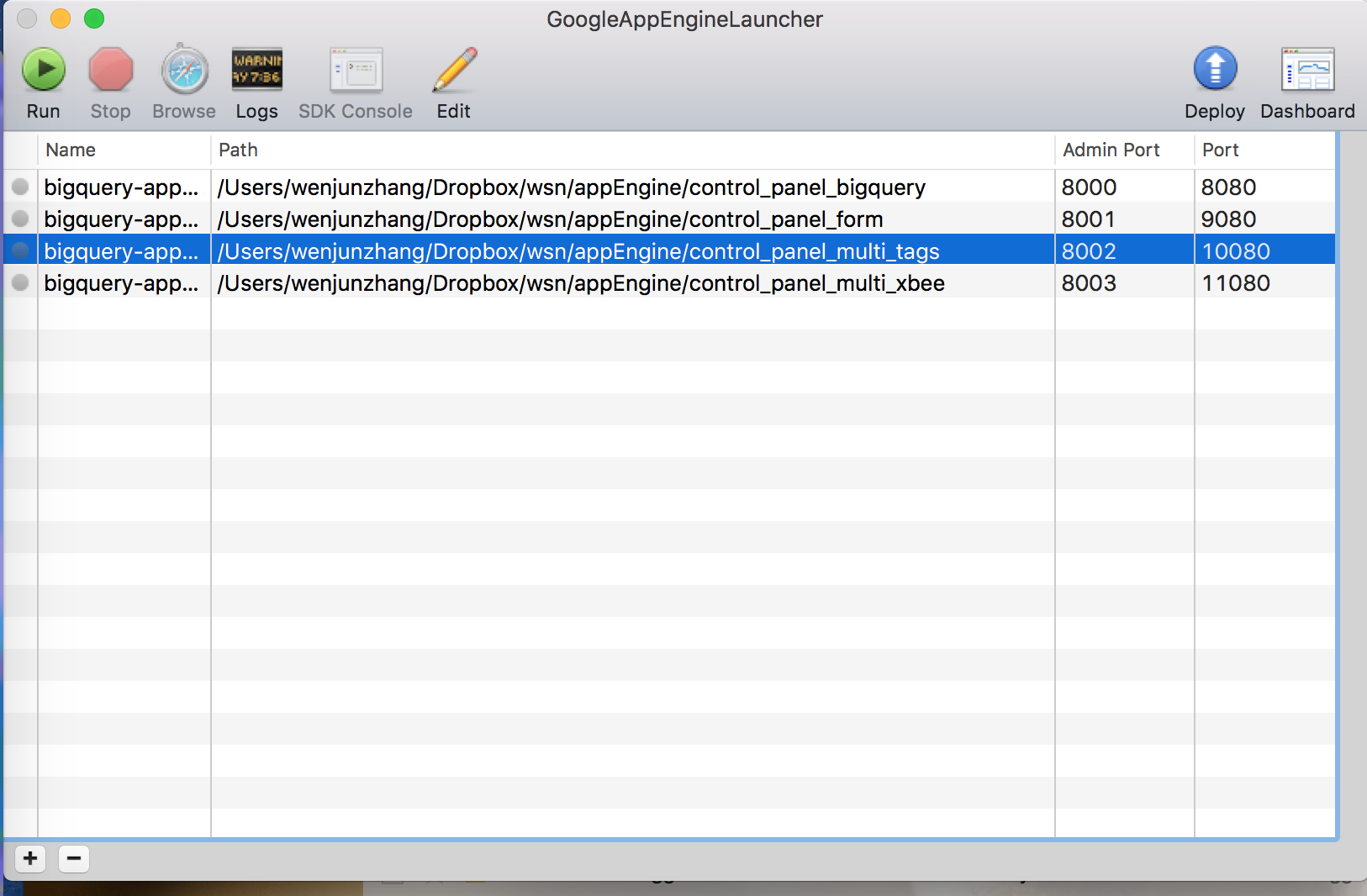


Fig 9 App Engine Launcher

You can add our example by simply click the ‘+’ button at the left down corner and navigate to you the example direction. After you add your file to launcher, you are ready to deploy your web app locally. You just click the ‘run’ button. If you pass the debug, click the ‘Browser’ button. It launches the default browser and shows the website you built. You can also check up the error logs if it occurs, and you can click the ‘Log’ button to retrieve it.

If all the local tests go well, you can deploy your website on Google Cloud. You just need to click the ‘Deploy’ button at App Engine Launcher. After your fill up your Google account information, it pops up the project information for you to choose up. You need to use the same project ID you applied at step 2. If every things goes well, you already have your own website running. Normally your website link is like this:

<https://spheric-mission-143823.appspot.com/>

the first part of this URL is your project ID, except that the ‘.appspot.com’ is the same for every app engine project. If you have your own domain name, you can change this URL to it by reset it in your Cloud account.

Here is another way to deploy the website, which is command line based. First, you need to use terminal with ‘gcloud init’ to set up your credential information.

For this step, you need to initialize your credential information as the one you use to apply for App Engine.

If you finish these cloud account setup, you can use this command line to deploy your application.

appcfg.py -A **spheric-mission-143823** -V v4-0 update .

Of course, you need to be under your application file and you need to replace the bold words with your own project ID. The underline part is the version of your application. You can use it control your application flow, while you need to set it up in your Google cloud account page.

***Case study***

Now, you should have your first App running and next I will use a case study to show you how to modify the code to make sure that it fits our application. Like how to embed the real-time plot, how to make forms and fill out forms with data from Datastore.

* Embed real-time plot to make a dashboard
* Make forms and exchange data base on it
* User accessible control
* Pass user input to Raspberry Pi(Gateway)

**First, embed real-time plot to make a dashboard**

We have been using the Plotly for real-time plot, and Plotly provides HTML embed method to us. We can simply use it on our website. In order to do so, we just need to inherit a template from our ‘base.template’ and insert the HTML embed object into it. Here’s the steps to do so.

First, create a template file called ‘usb\_unit.template’

Put it in the same direction as the ‘base.template’

Second, inherit from ‘base.template’

This will give each page the same header and footer style. In order to do so, put codes below at the top of your ‘usb\_unit.template’

{% extends "base.template" %}

{% endblock %}

Fig 10 Code example: inherit html template

Third, Get the embed object from Plotly

Go to the real-time graph and there is an embed button looks like this:

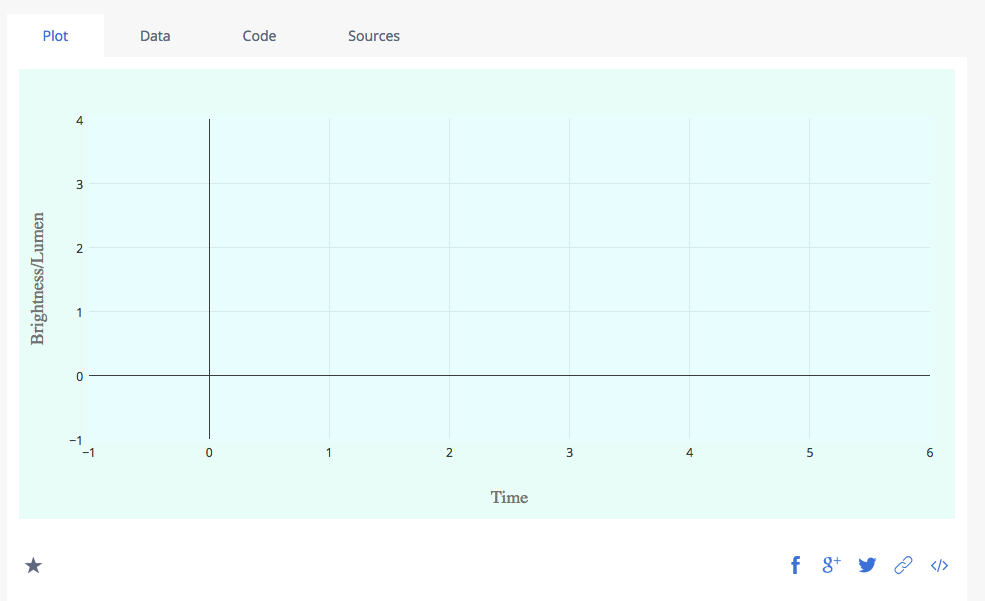


Fig 11 Plotly plot embed method example

If you click the html embed button , it gives you a piece of HTML code, which is the embed code you need for your page, copy it.

Forth, Embed plot

After your copied the code from Plotly, then paste it in your ‘usb\_unit.template’ and it should between the inherit code.

Now you have embedded plot in your page. Next you need to assign this page to a URL address.

Fifth, Assign page to URL

There are 2 ways to do it, first assign a URL to your page(template) in ‘app.yaml’ then create a python script for it to process your page. Second way to do it is that assign the URL in the main.py file and create python class in main.py to process your page.

In this case, we use the second way due to our website is not very large.

To do so, first navigate to main.py file, and add the URL, for example if you want your page URL is domain + ‘xbeepage’, you need to add the third line in this wbeapp2 function. The part after ‘/xbeepage’ is the class of your page which will have get/post method to handle user inputs.

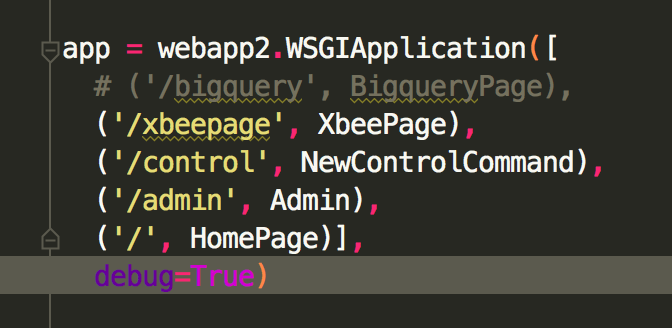


Fig 12 Code example: assign URL to template

Here’s an example of handling template in class

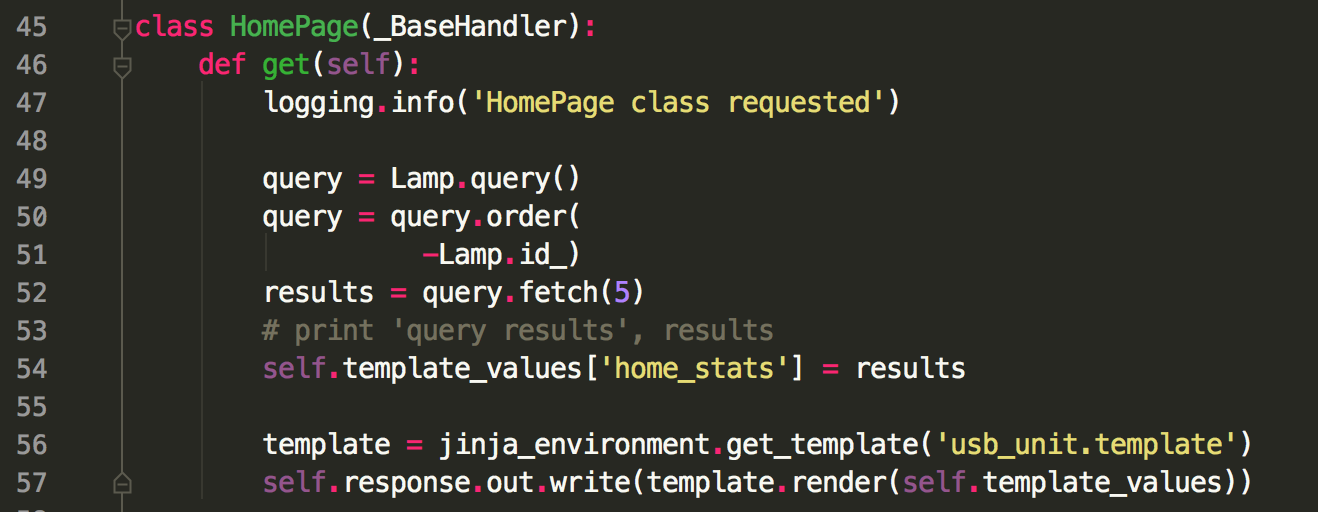


Fig 13 Code example: Render template

Line 56 and 57 direct users to the specific page of the URL. In our case, we assign the home URL to this template.

Now you can test your code. If you want more pages, you just repeat these steps.

**Second, make forms and exchange data with it**

Exchange between users and server there are lots of ways. In our case, we use the most common and simple way, html form.

In order to use form, we need to:

1. Declare a form in page (template)
2. Address form in Python by variable name which is defined in template

**Declare a form in page(template)**

HTML is tag based language, which means we can declare form using tag like

<form class="form-inline" method="post" action="/control">

In this line, the form is the tag declare. Class defines the style of this form, and action refers the URLs of this form.

In this form, we also need input tag to specify the input variable which uses in class for data exchange.

<input type="text" class="form-control" name="id" placeholder="1">

Above is an example of input tag, the name attribute is ‘id’ which is the variable name that can be addressed in Python class.

Now, in our page, we have form and each file of the form has a variable value. If we want to use the data or post data to this form, we need to process in Python class.

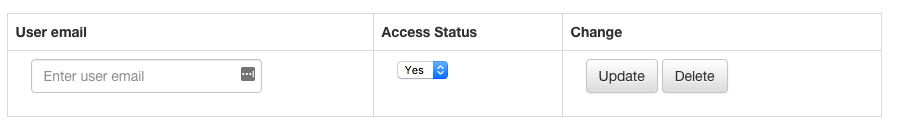


Fig 14 HTML form example

Here the button will trigger the URL actions.

**Address form in Python by variable name defined in template**

Here’s the example of the class code of the form:



Fig 15 Code example: Form operations

You can see we use the variable name defined in HTML template, and process actions and data from it.

You can modify this code to meet your design. Keep in mind that, the form is a round way trip, so you need to take care both template and Python class related to it.

**Third, user accessible control**

In this part, it’s all about the Datastore. We have already initialized the database with Datastore. Now let’s start to use it, i.e., query, insert or delete.

**Query**

We use the ‘ndb’ library to work with Datastore. The ndb library gives us a simple way to query the database in Datastore.

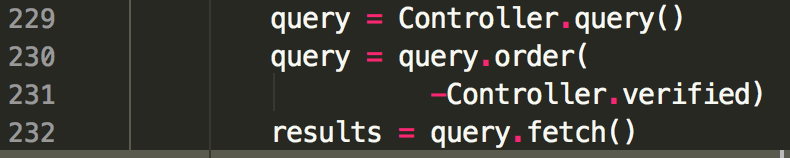


Fig 16 Code example: Query on Datastore

If you want query by keys, you just need put the key into function as argument. In this sample code, you get the result in a variable called results which pass to template and are showed in page.

**Insert or delete**

Let’s see the first code example.



Fig 17 Code example: Insert or delete on Datastore

For insert or delete, first we want to make sure whether there is an entity existing or not. If so, we can delete and we can only update instead of inserting to it.

Line 243 is the key to look up the Datastore. The keyword ‘Controller’ identify which entity we are looking for. ‘*self.request.get(‘user\_email’).lower()*’ is the keywork against the entity. In this case, it is the email address of user\_email.

The result of this query is the entity class which can be used for more details. The line 244 is an example that shows if the entity exists or not. If so, we can change the attribute of the entity and you need to call the put() method to commit this to the datastore.

The same idea applies in delete entity showed in line 262 to 264.

Here, we also show that how to control the user access. We just define the attribute of the Controller entity. This way can also be used for updating entities.

Here’s a link for using ndb, library.

<https://cloud.google.com/appengine/docs/python/ndb/>

**Fourth, pass user input to Raspberry Pi**

We have a control panel page in our web app. It grabs user inputs, while it’s not enough. We need to pass this information to the Raspberry Pi. The structure is like,

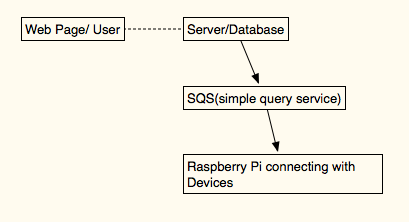


Fig 18 High level structure between Web and Raspberry Pi

To transform the information between server and Raspberry Pi, we use a service called SQS which basically is a global variable running on Amazon server and is shared by users as long as the request has the approved access keys.

Here are the steps.

**First, apply Amazon account and get API key**

Amazon gives 12 months’ free tier which is sufficient to our project. Then go to your account and enable the SQS service.

It gives you tip related to apply api\_key. You should keep this key safe since this key has the highest correlation to your account. This API key is shared between your server and Raspberry Pi. This is link and the unique key on Internet for Amazon to reach out and serve you.

**Second, set up SQS service on server**

We need to install the AWS python client on our server. In this case, we need to download the source code called ‘BOTO’ and copy this file to our web root direction.

After install ‘BOTO’, we can use it as a normal library and here’s the sample code.

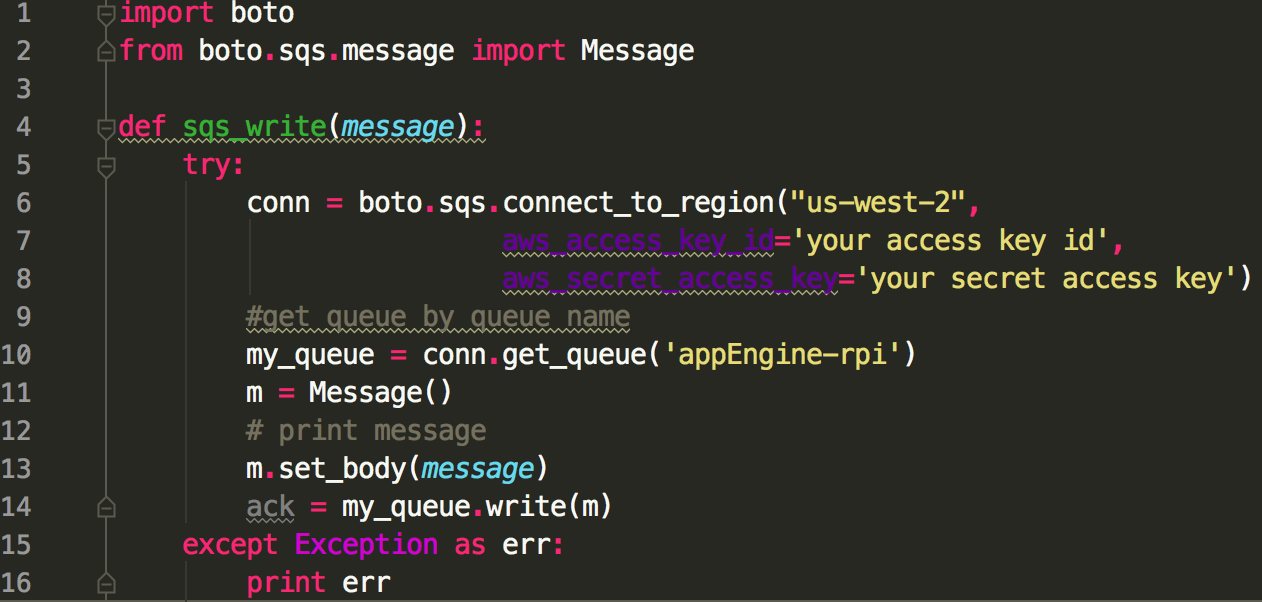


Fig 19 Code example: SQS write function on server

As you can see, we simply put the message to the queue called ‘appEngine-rpi’. On the side of Raspberry Pi, we need a read function that points to the same queue, which can be done on the other way around.

Now, the question is what’s the message we put into our queue and how to build a message.

Message can be in any formats as long as anyone who is using the queue can understand the format. In our case, we build our message as follows.

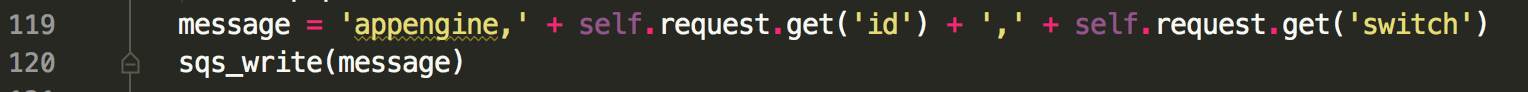


Fig 20 Code example: SQS message

In this sample, we identify the device by “id” and control it by “switch”.

**Third, set up SQS on Raspberry Pi**

Let us see the code sample on Raspberry Pi,

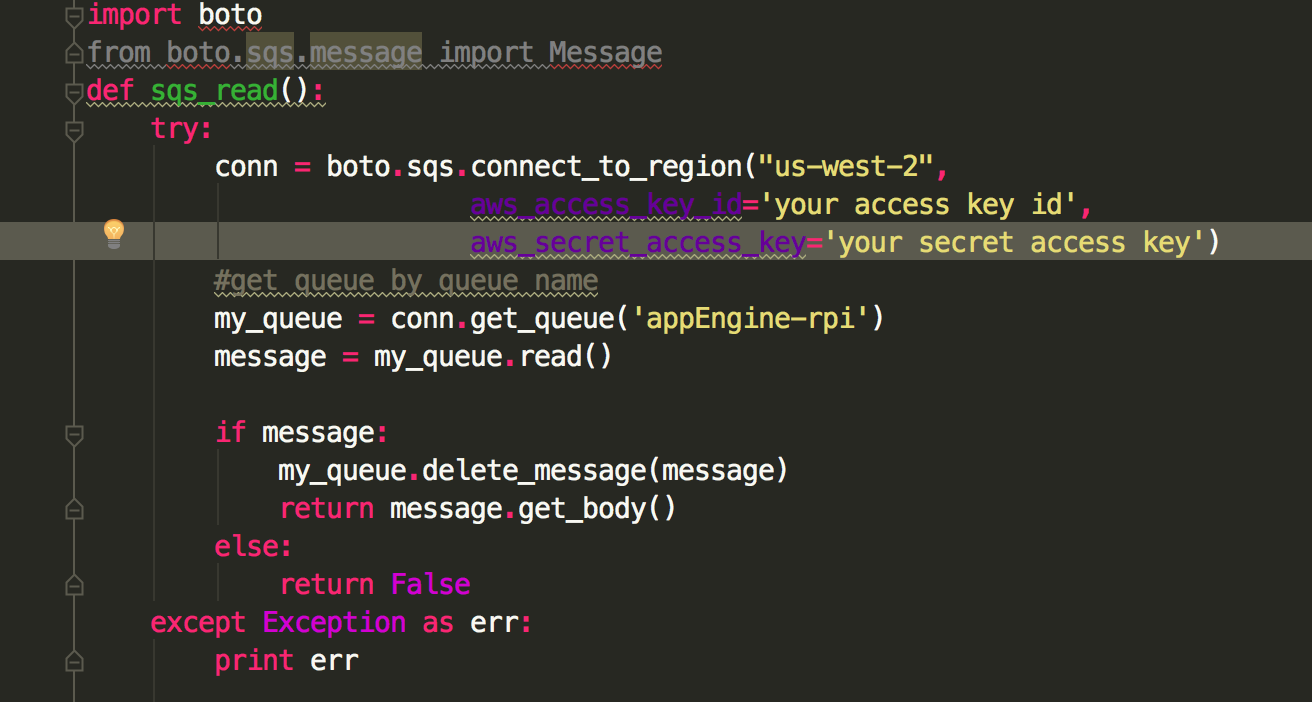


Fig 21 Code example: SQS read function on Raspberry Pi

This code simply uses the API information to point to the SQS queue and grab data/message from it.

After we get these information, Raspberry Pi parses the device id and acts based on the switch information.

That’s all we need to do to control Raspberry Pi from our website. If you want to make these more dynamic, you just need to modify these sample functions.

***Conclusion***

In this report, we illustrated the basic structure of a web application and mapped these ideas into Google App Engine. After that, we used case studies to demonstrate the details between the web page, server and Raspberry Pi.

This report illustrates the basic ideas about how to pick up available techniques from big suppliers and put them together to obtain our desired project, since we truly believe that in the era of IoT, developing is more about integrating ideas, products and solutions than do all of them by yourself.

As you can see, there are a lot of important aspects we haven’t dig deeply enough, such as security, network management and data intelligence. We will tackle them one by one in following reports, while the next one I will try is data analyze, more specifically real-time data analysis.