# CS4S Advanced Workshop – Modern Technologies – Web

*Building a Student Register with AppEngine & JavaScript*

# Tutorial

## Requirements

To complete this tutorial you will need to install the AppEngine Python SDK, and also have installed the AppEngine Launcher program.

You will also need to download the activity's resources (*"Student Register Resources.zip")* from here: https://goo.gl/Qsih6A and extract them somewhere on your machine.

Some of the instructions are specific to using the computers that were provided in the lab used for the workshop. The tutorial instructs you to use Mozilla Firefox and Notepad++ for developing and testing the app – but any web browser and text editor are likely to work.

## Step 0: Testing the AppEngine Launcher

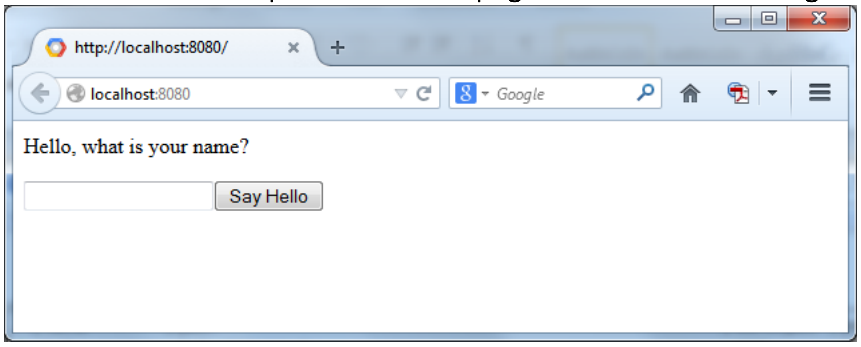
### Step 0.1: Hello World!

Now, before we get started on creating our Student Register let's test that the AppEngine Launcher is working as expected on the lab computers.

Open the AppEngine Launcher through the start menu. Type "App Engine" in the search box to find it quickly, or navigate it by clicking All Programs and selecting "Google App Engine Launcher"

Once the Launcher application has opened, navigate to **File > Add Existing Application**… click *Browse* and navigate to where you extracted the activity's resources. Select the helloapp folder, click *OK,* and then click *Add.*

Now select the helloapp web app in the Launcher. Once you have selected it, the top row in the Launcher's list of apps will become highlighted. Click the green *Run* button in the Launcher with the helloapp row selected. The icon next to the name column should turn yellow for a few seconds, and then turn green. Once it goes green, click the *Browse* button and Firefox should open with a web page that looks something like this:

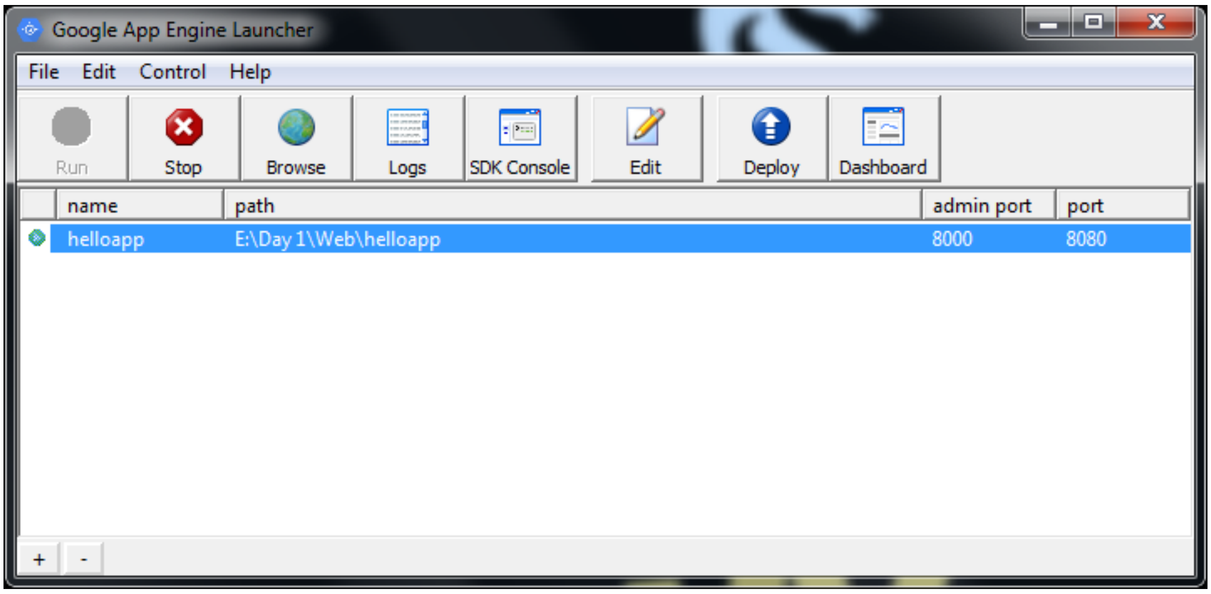


Hello, what is your name?

If you receive a blank page please ask for assistance from the instructor, or click the *Logs* button in the AppEngine Launcher to view any errors that occurred. When you get a web page that looks like the above, type in your (or any) name, and click the Say Hello button to receive a message.

## Step 0.2: The AppEngine Launcher User Interface

Now that we know that AppEngine is working as expected, let's take a closer look at the Launcher User Interface.



| | name | path | admin port | port |
|---|---|---|---|---|
| ● | helloapp | E:\Day 1\Web\helloapp | 8000 | 8080 |

The buttons in the Launcher have the following functions:

### *Run*

Clicking *Run* will start the selected app running. Once it is running, this button is no longer enabled and will only be enabled again once the app has been stopped.

### *Stop*

Clicking *Stop* should make the green icon next to the app name turn black. There seems to be a bug that happens on the lab computers where clicking *Stop* won't work and cause the Launcher to lock up. You may want to avoid clicking this to be safe. If you were to run the Launcher at home on your own computer this is unlikely to happen.

### *Browse*

Clicking *Browse* will open the app in a web browser (e.g. Firefox). This can only be clicked while the app is running.

### *Logs*

This opens a window that will show all the messages that have been logged by AppEngine. It is very useful for debugging (finding errors in your code). If you are getting blank web pages when clicking *Browse*, this is the first place to check to see what has gone wrong.

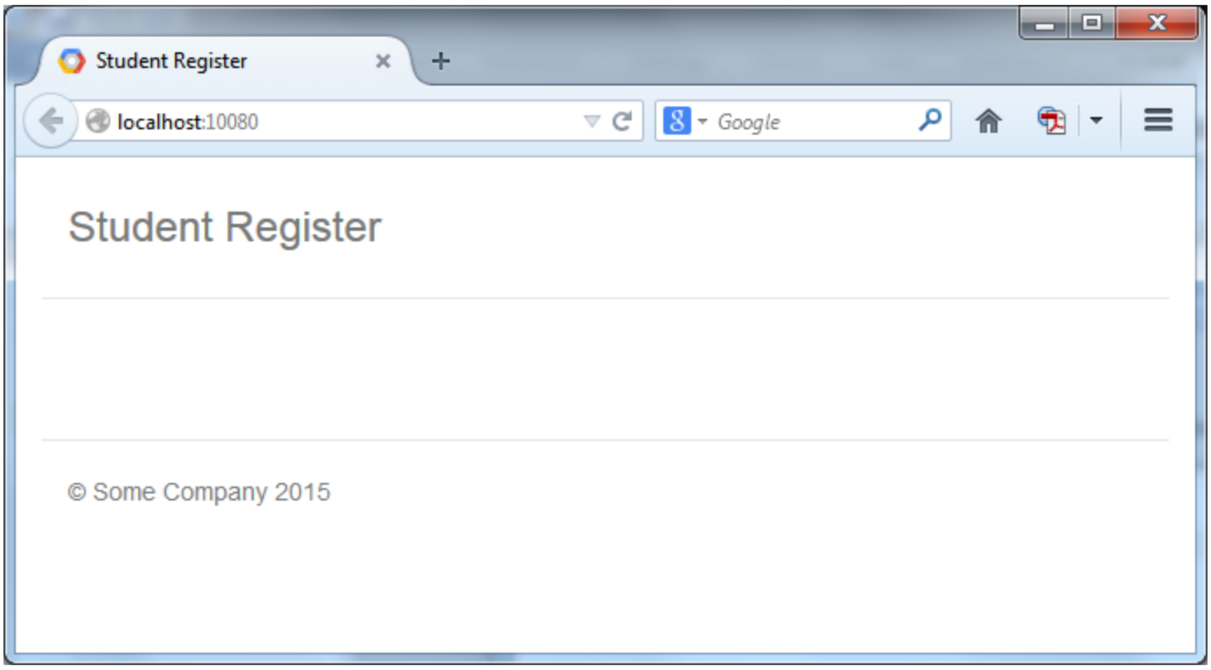
### *SDK Console*
The *SDK Console* allows you to view different information about the selected app, such as the data that has been stored by the app (*the DataStore Viewer*). The options available here and the features it supports are well beyond the scope of this tutorial, and workshop. If you are interested in learning more about these features, they are documented on Google's AppEngine website.

### *Edit*
Clicking *Edit* will open up a file titled **"app.yaml"** in a text editor. This file contains configuration settings for the app. Like the *SDK console* we will learn not much about this file in this workshop, but there is more information about it on the AppEngine site.

### *Deploy & Dashboard*
We won't be using these features of the Launcher today, but these would be useful if you wanted to make an app created in AppEngine available on the Internet.

## Step 0.3 Opening the Project Files
I have included a base project for this tutorial, so we won't be starting completely from scratch. To add the base project to the AppEngine Launcher you will need to navigate to **File > Add Existing Application…** and select the following folder where you extracted the activity's resource files: **studentregisterapp** and click *OK*.

Once you have added the app, selected the studentregisterp app clicked *Run*, and clicked *Browse* you should see something like this:



Student Register

© Some Company 2015

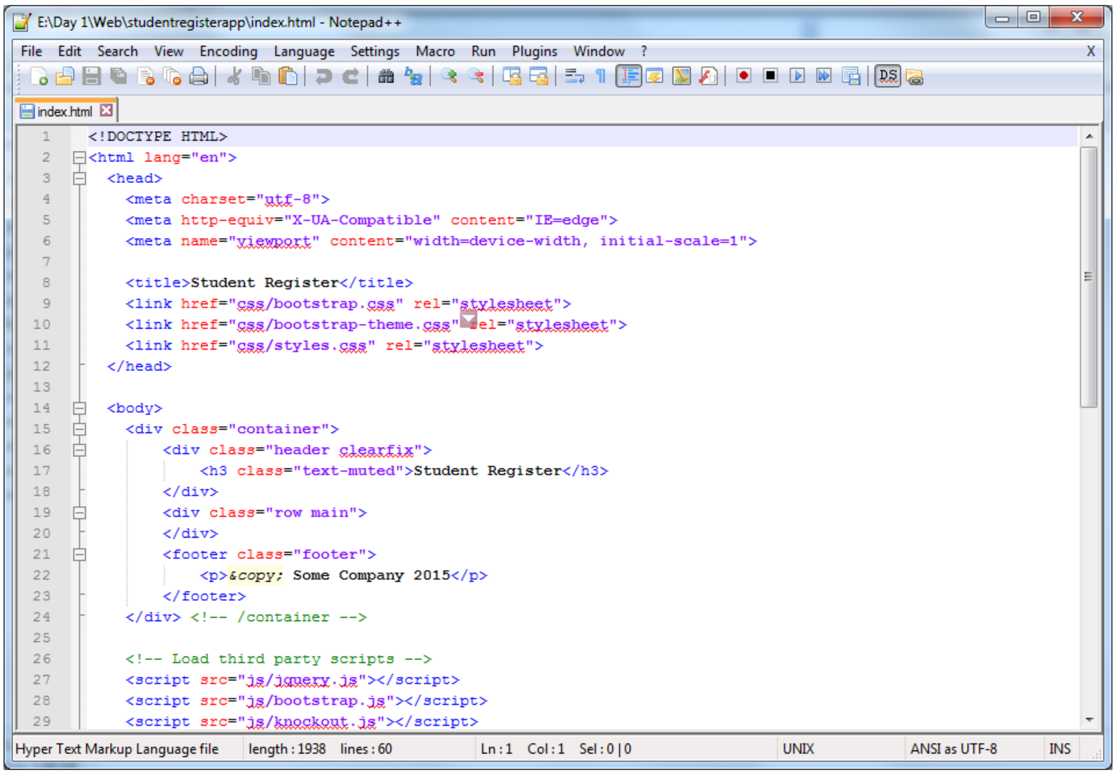
Next, open the folder that contains all of the project files from the Launcher by selecting the app and navigating to **Edit > Open in Explorer** or by using the keyboard shortcut **Ctrl + Shift + E**.

We will be editing files using Notepad++ today. If you have tried to open the files in the project directory you may have noticed that the HTML files will open in Firefox, and the Python (.py) files will cause a command prompt to open, but not do anything else.

We want to open these files in Notepad++, so firstly right click one of the HTML files (import.html or index.html) and go to **Open With > Choose Default Program…** Click *Browse* in the bottom left corner of the dialog, and navigate to *C:\Program Files (x86)\Notepad++*, select the notepad++ application and click *Open.* Make sure the checkbox labelled *“Always use the selected program to open this kind of file”* is checked, and click *OK.*

Double clicking on the HTML files should now open them in Notepad++. When opening Notepad++ a dialog with the message “Please wait while windows configures” may appear. If it does, just let it finish, and Notepad++ will open after this has completed.

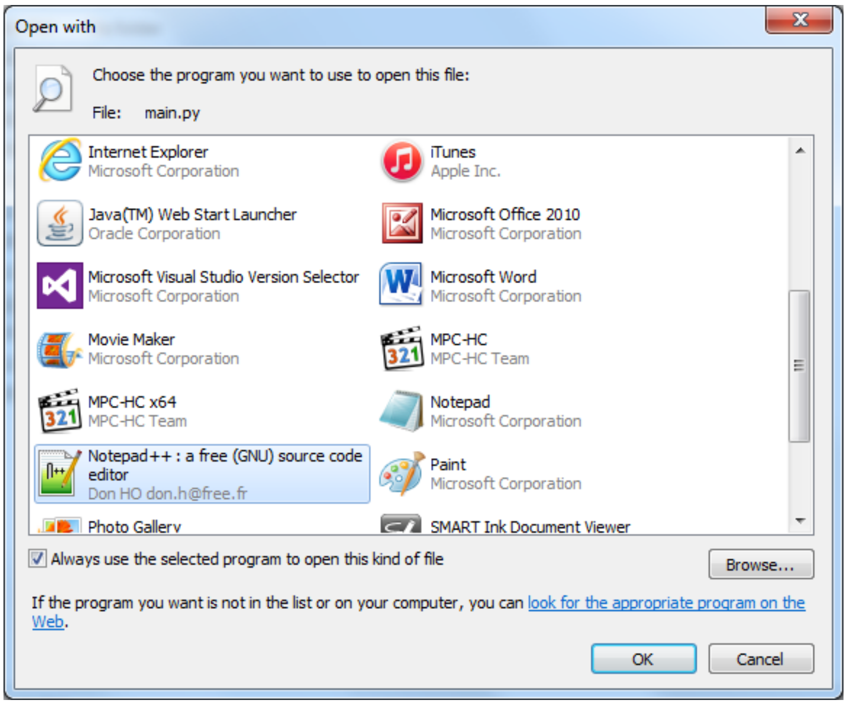
Once notepad++ has opened the *index.html* file, you should see something like this:


E:\Day 1\Web\studentregisterapp\index.html - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
index.html

```
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <title>Student Register</title>
    <link href="css/bootstrap.css" rel="stylesheet">
    <link href="css/bootstrap-theme.css" rel="stylesheet">
    <link href="css/styles.css" rel="stylesheet">
  </head>

  <body>
    <div class="container">
        <div class="header clearfix">
            <h3 class="text-muted">Student Register</h3>
        </div>
        <div class="row main">
        </div>
        <footer class="footer">
            <p>&copy; Some Company 2015</p>
        </footer>
    </div> <!-- /container -->

    <!-- Load third party scripts -->
    <script src="js/jquery.js"></script>
    <script src="js/bootstrap.js"></script>
    <script src="js/knockout.js"></script>
```

Hyper Text Markup Language file   length : 1938   lines : 60   Ln : 1   Col : 1   Sel : 0 | 0   UNIX   ANSI as UTF-8   INS

Now, do the same for the Python files, note that these are the files that have "PY" in the Type column, not "Compiled Python". Instead of having to click Browse and navigate to the Notepad++, you should now be able to select it in the *Other Programs* section pictured below. Once you have the notepad++ application selected, click *OK*.


Open with

Choose the program you want to use to open this file:

File: main.py

Internet Explorer
Microsoft Corporation

iTunes
Apple Inc.

Java(TM) Web Start Launcher
Oracle Corporation

Microsoft Office 2010
Microsoft Corporation

Microsoft Visual Studio Version Selector
Microsoft Corporation

Microsoft Word
Microsoft Corporation

Movie Maker
Microsoft Corporation

MPC-HC
MPC-HC Team

MPC-HC x64
MPC-HC Team

Notepad
Microsoft Corporation

Notepad++ : a free (GNU) source code editor
Don HO don.h@free.fr

Paint
Microsoft Corporation

Photo Gallery

SMART Ink Document Viewer

[✓] Always use the selected program to open this kind of file

If the program you want is not in the list or on your computer, you can look for the appropriate program on the Web.

Now, let's get starting on our Student Register app.

## Part 1: Client Side

The first part of this tutorial involves the creation of the Client. The Client in the case of this app is the index.html, which will make requests to the Server (covered in Part 2).

## Step 1.0: Student Register Template

Before we get into writing some JavaScript & HTML, let's take a brief look at the files we have in the project directory at the moment. You may notice that there are a lot of files and folders in the *studentregisterapp* project directory. We will only need to make changes to a couple of these files for this tutorial, but the purposes of these files are explained below.

### Project Files

#### *HTML Files*

The **index.html** file will be the main page of the Student Register. It will contain the HTML for the layout of the main page, as well as the JavaScript code needed to add and remove Students from the Register. We will make most of our changes in this file.

The **import.html** file will be a page for uploading a data file with a list of students; this won't be needed until Part 2 of this tutorial.

### *CSS Directory*
**bootstrap.css** and **bootstrap-theme.css** are a set of CSS files that are part of the Bootstrap front-end framework (developed by Twitter). Bootstrap includes styles for buttons, fonts and different sized screens. It's useful for building web applications, because it means you don't have to make your own graphics or write your own CSS to make a nice looking website. There are many alternative front-end frameworks to Bootstrap, but because of its popularity there is a large amount of tutorials and resources available for it.

**styles.css** has some custom styling for the "Student Register" header, and content. If you'd like to make adjustments to the website's styling, this is the best place to do this.

### *JS Directory*
**knockout.js** is a JavaScript library called KnockoutJS. Its purpose is to simplify the development of web applications that have dynamic user interfaces (like our Student Register).

**knockout-mapping.js** is a plugin for KnockoutJS that simplifies the conversion of JSON objects to KnockoutJS View Models. It will be used when retrieving our Student data from the AppEngine DataStore in Part 2 of this tutorial.

**jquery.js** is a JavaScript library called jQuery. Its purpose is to simplify actions like making AJAX requests in JavaScript. It is used on many web sites, and like Bootstrap there is a wealth of resources and tutorials available if you would like to learn more about it.

**bootstrap.js** is a script that provides features such as modal dialogs and animations for tabs. We won't use any of these features in the tutorial, but they could be used to make some nice future additions to our Student Register if you were to extend this tutorial.

**xml2json.js** is a script that controls conversion between JSON and XML data (and vice versa), this isn't required until part 2, and is only used once.

### *Data Directory*
This directory contains the **data.json** file, which has a list of Students in the JSON data format. We will use this list of students to populate the Student Register in the first part of this tutorial, before moving this data into an AppEngine DataStore.

### *Configuration File*
The **app.yaml** file contains the configuration for our web application. The instructions for Part 2 will contain more information about this file; you can ignore it for now.

### *Rest Directory & .py files*
You can ignore these for now; these won't be needed until Part 2 of this tutorial.

### *Code Directory*
All of the snippets in this tutorial are in this directory, organised into folders for both parts of the tutorial and each of their steps. For steps in this tutorial that have a lot of code you may want to copy and paste these into your code, rather than type it all out yourself.

### A Closer Look at Index.html

Now let's take a closer look at the index web page's HTML. If you are familiar with web development you will probably recognise most of the document structure and tags. If you are new to KnockoutJS or JavaScript this section will likely be unfamiliar to you:

```
var viewModel = function() {
    var self = this;
    self.students = ko.mapping.fromJS([]);
    self.loadStudents = function() {

        // Create a Student in code and add it to the list of Students
        var student = {};
        student["first_name"] = "Alan";
        student["last_name"] = "Turing";
        student["address"] = "1 Bletchley Park Parade";
        student["phone_number"] = "2306 1912";

        // Need to convert the JavaScript object to a Knockout Observable
        var observableStudent = ko.mapping.fromJS(student)
        self.students.push(observableStudent);
    };
};

var vm = new viewModel();
vm.loadStudents();
ko.applyBindings(vm);
```

In this code we are creating a Knockout ViewModel, which will be bound to our user interface. This ViewModel has an array of Student observables (which is created as an empty array), and a function (loadStudents) for loading all the Students into this array. For now the loadStudents function creates a student named "Alan Turing" and adds him to the Students array.

In the last 3 lines, an instance of the ViewModel is created, the loadStudents function called, and the ViewModel is then bound to the HTML page. This means that once we display the Students on the HTML page any change made to them through the HTML page will be reflected in our JavaScript code, and vice versa.

## Displaying the List of Students

Now we start adding HTML to the index.html file to display the table of Students.

### Creating the Table

Our first step will be adding a table to display all of the Students in our student register. All of the code that needs to be added to this project is in the **studentregisterapp\code\** directory where you extracted the activity's resources. These can be directly copied and pasted into the appropriate sections of the index.html file, but in some cases you may wish to type them out yourself.

### *student_table.html*

The code to add the table is as follows. This should be placed between the *<div class=" row main">* tag and its corresponding *</div>* tag. Note that notepad++ may not retain the indentation of the code being copied and pasted, so you may have to adjust this by selecting the pasted text and using the **Tab** or **Shift + Tab** keys to correct the indentation.

```
<table class="table table-striped table-bordered">
    <thead>
        <tr>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Address</th>
            <th>Phone Number</th>
            <th></th>
        </tr>
    </thead>
    <tbody data-bind="foreach: students">
        <tr>
            <td data-bind="text: first_name"></td>
            <td data-bind="text: last_name"></td>
            <td data-bind="text: address"></td>
            <td data-bind="text: phone_number"></td>
            <td>
                <!-- Delete Button will go here later on. -->
            </td>
        </tr>
    </tbody>
</table>
```

The classes used in the *<table>* tag are bootstrap CSS classes, which give the table a nicer appearance than the default HTML tables.

The *data-bind* attribute in the *<tbody>* tag is used to set up a binding from the HTML page to our ViewModel. This means that there will be a row in our table for each of the students in our list.

The *data-bind* attributes in the *<td>* tags are bindings for properties on the Students themselves. For example the **"text: first_name"** means that the first column in our table will display a Student's first name.

If you save your index.html file, and refresh the Student Register page, you should see a table with Alan Turing listed as a Student, along with his address and phone number. If you closed your web browser earlier, make sure to reopen the page by clicking the AppEngine Launcher's *Browse* button.

For now you can ignore the text about the Delete Button in the HTML file, we will add this later on in the tutorial.

## Reading the Students from a Data File

Instead of adding the student in the code, we will now load the list of students from the *student-list.json* file in the *data* directory in your project files. To do this we will use a jQuery function to make an AJAX request.

### *load_students_from_file.js*

**Note:** If this file doesn't open in Notepad++, follow the same steps as we did when setting up the opening of HTML and python files in Notepad++.

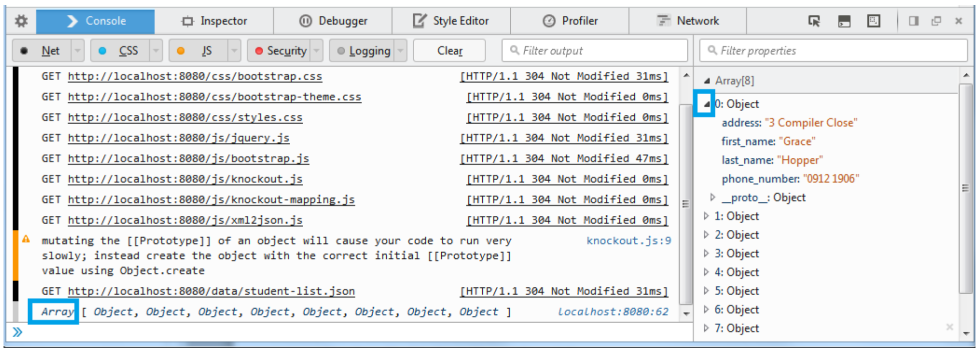
Back in the *index.html* file, use the code in the "*load_students_from_file.js*" file to replace the code from "self.loadStudents = function() {" to its corresponding end ("};").

```
self.loadStudents = function() {
    $.getJSON("/data/student-list.json", function(studentList) {
        console.log(studentList);
        ko.mapping.fromJS(studentList, self.students);
    })
};
```

The $.getJSON is a jQuery function for retrieving JSON data from a given URL. The URL we are retrieving data from is a JSON file (titled *student-list.json*) that is in the *data* directory of our project.

Save and refresh the page and you should see a list of 9 Students in your Student Register.

In Firefox, open up the developer tools (**Ctrl + Shift + I**), and refresh the page.



```
GET http://localhost:8080/css/bootstrap.css            [HTTP/1.1 304 Not Modified 31ms]
GET http://localhost:8080/css/bootstrap-theme.css       [HTTP/1.1 304 Not Modified 0ms]
GET http://localhost:8080/css/styles.css                [HTTP/1.1 304 Not Modified 0ms]
GET http://localhost:8080/js/jquery.js                 [HTTP/1.1 304 Not Modified 31ms]
GET http://localhost:8080/js/bootstrap.js              [HTTP/1.1 304 Not Modified 47ms]
GET http://localhost:8080/js/knockout.js                [HTTP/1.1 304 Not Modified 0ms]
GET http://localhost:8080/js/knockout-mapping.js        [HTTP/1.1 304 Not Modified 0ms]
GET http://localhost:8080/js/xml2json.js                [HTTP/1.1 304 Not Modified 0ms]
mutating the [[Prototype]] of an object will cause your code to run very     knockout.js:9
slowly; instead create the object with the correct initial [[Prototype]]
value using Object.create
GET http://localhost:8080/data/student-list.json       [HTTP/1.1 304 Not Modified 31ms]
Array [ Object, Object, Object, Object, Object, Object, Object, Object ]    localhost:8080:62
```

```
Array[8]
  0: Object
    address: "3 Compiler Close"
    first_name: "Grace"
    last_name: "Hopper"
    phone_number: "0912 1906"
    __proto__: Object
  1: Object
  2: Object
  3: Object
  4: Object
  5: Object
  6: Object
  7: Object
```

The console shows all of the requests that have been made by the browser. In our code above the "*console.log(studentList)*" line means that the data we retrieved via the AJAX request will be written to the console. We can view details about the retrieved data by clicking the Array button and then by expanding the arrows on the right side of the screen, as highlighted by the blue squares above.

The code "*ko.mapping.fromJS(studentList, self.students);*" turns the retrieved data into what is called a Knockout Observable Array. This means that any changes made to the list of Students through code will be automatically be updated on the HTML page. For example if we add a Student to our View Model's list of Students, then a new row will be added to the table on the web page.

Next add a new student to the Student Register by adding a new Student to the *student-list.json* file in the data directory. An example of how to do this is shown below:

```
  {
    "first_name":"Jeanette",
    "last_name":"Wing",
    "address":"8 Research Road",
    "phone_number":"2222 1000"
  },
  {
    "first_name":"New",
    "last_name":"Student",
    "address":"123 Fake St",
    "phone_number":"0101 0101"
  }
]
```

Once you have added a student, saved the *student-list.json* file and refreshed your page you should see the new student in the table of Students. This isn't a very practical way to add a student to the register though, is it? Let's add a form for adding a new student instead.

## Creating a New Student Form

We will now add a form to add a new Student to the Register. This form will allow you to enter a student's first name, last name, phone number and address, and click a button to add them to the Register.

### *add_student_form.html*

Next add the *add_student_form.html* code after the table (On the line under the *</table>* tag*)* in the *index.html* file. It is recommended to copy and paste this code, as there's lots of tags and text in it.

There is lots of HTML in the *add_student_form.html* file, but there are only a few parts of this that we will focus on for this tutorial. The *row*, *col*, *form-group* and *form-control* CSS classes are bootstrap classes that are used to organise the layout of the Add Student form.

```
<label for="firstNameTextBox">First Name:</label>
<input type="text" class="form-control" id="firstNameTextBox" data-bind="value: inputFirstName">
```

In the HTML above, the *"data-bind='value: inputFirstName'>"* line sets up a binding so that the *inputFirstName* field on our ViewModel will be whatever is typed into the textbox with the label *"First Name*". This field, and the other input fields, haven't been added to our ViewModel yet - we will do that in the next step.

```
<button class="btn btn-success pull-right" data-bind="click: addStudent">
    Add Student
</button>
```

The data-bind attribute *"click: addStudent"* sets up a binding so that when the *"Add Student"* button is clicked, the *addStuden*t function on the ViewModel will be called. We will add this function soon.

If you save and refresh your file, you should see the Add Student form – but it will not work yet. We now have to add some more JavaScript code.

### *input_properties.js*

Now we will add the properties to the ViewModel which are bound to the text boxes on the Add Student form. This code goes in the javascript, underneath this line: *"self.students = ko.mapping.fromJS([]);"*

```
self.inputFirstName = ko.observable();
self.inputLastName = ko.observable();
self.inputAddress = ko.observable();
self.inputPhoneNo = ko.observable();
```

This means that when the text box for the Student's first name is typed into, the property on the ViewModel in JavaScript will automatically be updated. We will use these when creating new Students and adding them to the list of Students.

### *add_student_click.js*

The next step is adding code for adding the student to the list. This code should be inserted after the following line in index.html: *"self.students = ko.mapping.fromJS([]);"*

```
self.addStudent = function() {
    var newStudent = {};
    newStudent["first_name"] = self.inputFirstName();
    newStudent["last_name"] = self.inputLastName();
    newStudent["address"] = self.inputAddress();
    newStudent["phone_number"] = self.inputPhoneNo();

    //Convert the new student to Observable object and add it to the list of Students
    var observableStudent = ko.mapping.fromJS(newStudent);
    self.students.push(observableStudent);

    //Clear the input fields
    self.inputFirstName("");
    self.inputLastName("");
    self.inputAddress("");
    self.inputPhoneNo("");
};
```

In this code we:

1. Create a new JavaScript object representing the new Student
2. Use the inputs from the Add Student form to "fill in" the first_name, last_name, address and phone_number information for the added student.
3. Map the JavaScript object to a Knockout Observable
4. Add the new Student to the ViewModel's list of Students
5. Clear all of the input properties, which will also update the text boxes on the main page.

Save your changes to the *index.html* page, refresh the page, and try to add a new Student. The table should update with the newly added Student after clicking the *Add Student* button. If you refresh the page you will notice that the Students you added will disappear. This is expected - we will address this in Part 2 of this tutorial.

## Deleting a Student

Now we will add a way of deleting Students from the Student Register.

### *delete_student_button.html*

This html code should replace the comment in one of the *<td>* tags, that says *<!-- Delete Button will go here later on. -->.*

```
<button class="btn btn-default" data-bind="click: $root.deleteStudent">
    Delete
</button>
```

You may notice the *$root* in the "*click*" binding. This means that the click will be bound to the "*deleteStudent*" function in our ViewModel. If it was just "*deleteStudent*" without the *$root*, Knockout would look for the binding on our Student object, and as it is not set up for each Student this would cause errors.

### *delete_student_click.js*

This code adds the function for deleting Students, as you will see we don't have to add much code; Knockout takes care of removing the appropriate from the Students list (and the table). This code goes after the *"self.students = ko.mapping.fromJS([]);"* line in *index.html*.

```
self.deleteStudent = function(student) {
    self.students.remove(student);
};
```

Save the *index.html* file, and refresh the page. You should now be able to delete Students from the register. If you delete one of the original 9 students, and reload the page you will notice that they haven't really been deleted.

This is because the Student data is not being persisted; the Student data only exists until the page reloads, when the data from the *student-list.json* file is loaded again. There are many options for persistence of this data, but today we will use an AppEngine DataStore to store the Student data.

## Part 2: Server Side

In this part of the tutorial there is an explanation of how the server-side of the Student Register has been set up. Then after this explanation there are instructions on how to change our HTML to retrieve from and save data in the AppEngine DataStore.

### What is AppEngine DataStore?

The AppEngine DataStore is a database that is built into the AppEngine SDK. It allows you to define models (also referred to as *entities*) that can be persisted. DataStore is a schema-less NoSQL datastore, and quite different to traditional relational databases you may be familiar with, like MySQL and Oracle. This tutorial does not much cover much about the DataStore and its advantages/disadvantages when compared to other options, there is documentation about it available here: https://cloud.google.com/appengine/docs/python/datastore/

When running locally, the data is stored on the machine you are developing on as an sqlite file. If you were to Deploy an app on the Internet, the data is stored "in the cloud".

### Defining Models

Models are the objects that are stored in the DataStore. These objects can have different properties, for example StringProperty properties for text like Names and Addresses. There are other property types available for AppEngine models. For example, if you were to add an Age property to the Student model, this could be an IntegerProperty.

```
class Student(db.Model):
  first_name = db.StringProperty()
  last_name = db.StringProperty()
  address = db.StringProperty()
  phone_number = db.StringProperty()
```

### Importing our Data into the Student Register Data Store

Our first step will be to import the Students in our *student-list.json* file into the DataStore. The code to do this has already been written, and is in the same file as our Student model (model.py).

```
def create_from_file(file_content):
  student_list = json.loads(file_content)
  count = 0
  for student in student_list:
    first_name = student["first_name"]
    last_name = student["last_name"]
    address = student["address"]
    phone_number = student["phone_number"]
    new_student = Student(first_name=first_name, last_name=last_name, address=address, phone_number=phone_number)
    new_student.put()
    count += 1
  return count
```

In the code above, the contents of our *student-list.json* file is loaded into a list of Python dictionary objects. We then loop through each Student in the list, create a Student Model and save it to the DataStore. Python has an inbuilt JSON module, which makes the parsing of JSON files very easy.

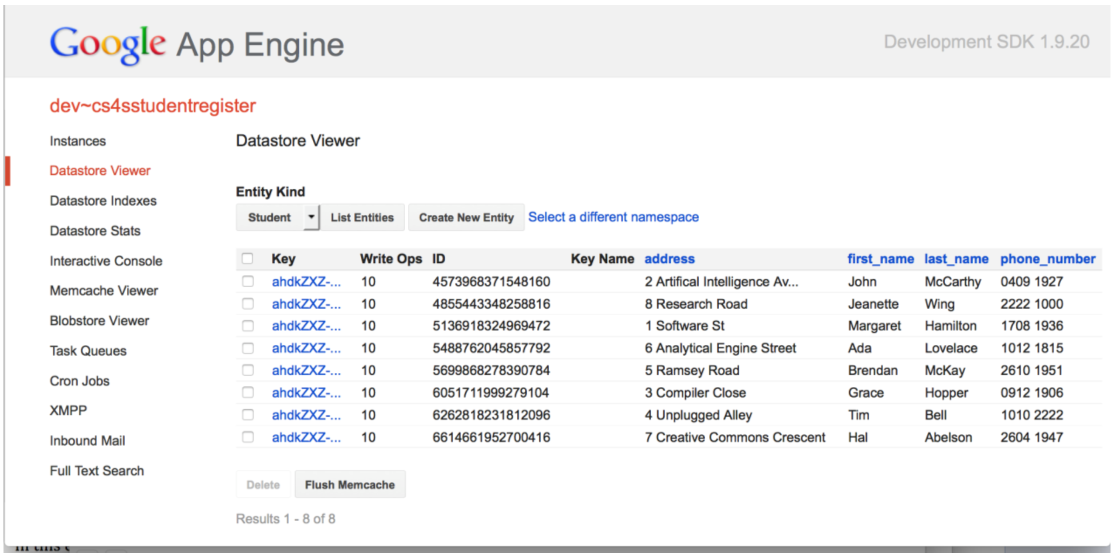
You don't have to worry about this code; we won't be making any changes to it in this tutorial. To import the students from our data file first navigate to */import* in your browser, by typing /import after *localhost:9080* in the address bar of your web browser and hitting enter.

On the import web page, click *Browse* and navigate to to where you extracted the activity's resources, to the directory **/studentregisterapp/data**, select the **student-list.json** file, click *Open* and once the Open File Dialog closes, click the *Upload* button.

After clicking the *Upload* button you should see a web page with the following text: "8 students added to the Register." (Or if not 8, however many students are in your *student-list.json* file).

One of the handy features of the AppEngine Launcher is the SDK Console. You can view the imported Students using the DataStore Viewer. To do this, click *SDK Console* in the AppEngine Launcher, and select *DataStore Viewer* in the navigation menu on the left.

You should see a result like the following:



| Key | Write Ops | ID | Key Name | address | first_name | last_name | phone_number |
|---|---|---|---|---|---|---|---|
| ahdkZXZ-... | 10 | 4573968371548160 | | 2 Artifical Intelligence Av... | John | McCarthy | 0409 1927 |
| ahdkZXZ-... | 10 | 4855443348258816 | | 8 Research Road | Jeanette | Wing | 2222 1000 |
| ahdkZXZ-... | 10 | 5136918324969472 | | 1 Software St | Margaret | Hamilton | 1708 1936 |
| ahdkZXZ-... | 10 | 5488762045857792 | | 6 Analytical Engine Street | Ada | Lovelace | 1012 1815 |
| ahdkZXZ-... | 10 | 5699868278390784 | | 5 Ramsey Road | Brendan | McKay | 2610 1951 |
| ahdkZXZ-... | 10 | 6051711999279104 | | 3 Compiler Close | Grace | Hopper | 0912 1906 |
| ahdkZXZ-... | 10 | 6262818231812096 | | 4 Unplugged Alley | Tim | Bell | 1010 2222 |
| ahdkZXZ-... | 10 | 6614661952700416 | | 7 Creative Commons Crescent | Hal | Abelson | 2604 1947 |

Results 1 - 8 of 8

You may notice that you can create new Students and delete existing Students through the DataStore Viewer. However this involves synchronous requests, rather than asynchronous requests, which makes it less "slick" than our Student Register.

AppEngine includes a web application framework called *"webapp2",* which is used for setting up request handlers. The following code is from the *main.py* file, which sets up our web app.

```
app = webapp2.WSGIApplication([
    ('/', MainHandler),
    ('/import', ImportHandler),
    ('/api/.*', rest.Dispatcher)
], debug=True)
```

In this code different handlers are set up for our application. For example, this means that when you navigate to *localhost:9080/import* in your browser, the application will run the code in the ImportHandler request handler. If you look at the ImportHandler code in the main.py file you will notice that there is a function called *get()* and a function called *post()*. Depending on the type of request (GET or POST) one of these functions will run when a request to */import* is made.

This tutorial does not go into further detail about setting up request handlers or doing more advanced work with these because of time constraints. However there is a lot that you can use webapp2 for. There are many alternative options for web application frameworks in Python, which can also be used with AppEngine, for example Django and Flask.

## Creating a REST API

For our Student Register, we want to be able to do the following:

- Create new Students in the DataStore
- Retrieve a list of all the Students in the DataStore
- Delete Students in the DataStore

These operations are often referred to as CRUD (Create, Read, Update and Delete). We won't be doing any Update operations in this tutorial, but this project could be extended to include this.

There are many approaches to implementing CRUD operations, but luckily for us there is a drop-in solution for AppEngine apps that requires little coding and configuration. This solution is a project called *appengine-rest-server* which will handle most of the work for us. More information is available at: https://github.com/jahlborn/appengine-rest-server

Using *appengine-rest-server* will create what is called a REST API for our defined model (Student). A REST API has URLs for different CRUD operations. For example, performing a GET request to */api/Student* will list all the Students in our Register in XML format (the Read operation). Additionally, performing a POST request to */api/Student* would create a new Student in the DataStore.

**Note:** using *appengine-rest-server* is appropriate for prototype apps and examples, but if you were to build a more complex app it may not be appropriate. If you were storing sensitive information in your app that was to be made available on the Internet it would not be appropriate to use this project without adding some extra security checks so your data would be safe.

The code to register the *appengine-rest-server* has already been set up in *main.py*, but here is a quick explanation of the process of how it's set up.

```
import model
import rest
```

Firstly, we import the rest module, which is in the *rest* directory of the project. We also import the *model* module, which is the *model.py* file. The *model.py* file contains our definition of the Student model, which needs to be registered with the *rest.Dispatcher* to set up our REST API.

```
app = webapp2.WSGIApplication([
    ('/', MainHandler),
    ('/import', ImportHandler),
    ('/api/.*', rest.Dispatcher)
], debug=True)
```

We register the *rest.Dispatcher* as a request handler for all requests made with the pattern *'/api/.*'*

```
rest.Dispatcher.base_url = '/api'
rest.Dispatcher.add_models_from_module(model)
```

We register the models (in this case the Student) with the rest dispatcher, so that it will allow us to do CRUD operations with this model.

If it has all been set up correctly, navigate to *localhost:9080/api/Student* in your browser and you should see a list of Students in XML format. If it doesn't work make sure the URL is */api/Student* (uppercase S), and **not** */api/student* (lower-case s).

Now that we have the Server Side set up, we will now add code to our client (our index.html file) to call the Server (AppEngine / Python) to allow us to perform CRUD operations on the AppEngine DataStore.

## Reading Students from the DataStore

Now, instead of reading the Students from the data file we have, we will read them from our DataStore.

### *load_students_from_api.js*

Replace the *"self.loadStudents"* function in index.html with the following code.

```
self.loadStudents = function() {
    $.getJSON("/api/Student", function(studentList) {
        console.log(studentList);
        if (studentList["list"]["Student"]) {
            ko.mapping.fromJS(studentList["list"]["Student"], self.students);
        }
    })
};
```

We are still using jQuery’s *getJSON* function, but we are now making a request to our REST API instead of our JSON file. We then check if the data that has come back contains any data, and then map the results to the ViewModel’s Students array.

Once you have saved and refreshed you should see the table of Students like before. If you had added any Students through the DataStore Viewer then they should also be in the table.

## Adding a Student to the DataStore

Now we will add code to create new students in the DataStore. Note that we still need to add the Student to the ViewModel’s Array of Students so that the table is updated with the new student.

### *add_student_ajax.js*

Before adding the add_student_ajax.js code, remove the following lines from index.html:

```
//Convert the new student to Observable object and add it to the list of Students
var observableStudent = ko.mapping.fromJS(newStudent);
self.students.push(observableStudent);
```

Once they have been removed, add the *add_student_ajax.js* code after the *“newStudent["phone_number"] = self.inputPhoneNo();”* line.

There is a lot of code in this file, so it has been broken down into steps:

```
//Convert the New Student JavaScript object to XML
var x2js = new X2JS();
var xmlData = x2js.json2xml_str({ "Student" : newStudent });
```

We first convert the Student JavaScript object into XML. This is done because the *appengine-rest-server* only seems to accept XML data for creating new entities in the DataStore. We use *x2js* here; this is a library that handles the conversion of JSON to XML (and vice versa).

```
$.ajax({
    url: '/api/Student',
    data: xmlData,
    type: 'POST',
    contentType: "text/xml",
    dataType: "text",
    success: function(data) {
        //The response from the POST will be the key of the added Student record
        newStudent["key"] = data;

        //Convert the new student to Observable object and add it to the list of Students
        var observableStudent = ko.mapping.fromJS(newStudent);
        self.students.push(observableStudent);
    }
});
```

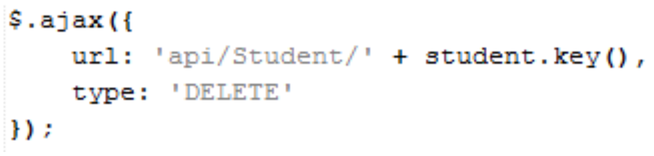
We then perform an AJAX POST request to */api/Student* URL, sending the new Student data as XML. The success function is code that runs if the request was successful. The data in response is the key of the newly added Student. We update the newStudent object with this key, as it will be needed if we try and delete the Student. Then, we convert the JavaScript object to a Knockout observable as we did previously, and add it the ViewModel's list of students so that the HTML table updates correctly.

## Deleting a Student in the DataStore

Now, all we have left to do is add code to perform deletion in the DataStore.

### *delete_student_ajax.js*

Add the following code after the line *"self.students.remove(student);"* in the *deleteStudent* function.



```
$.ajax({
    url: 'api/Student/' + student.key(),
    type: 'DELETE'
});
```

Note that this uses the student's Key property (*student.key()).* This key value is included in the response to the request to*'/api/Student*. Although it isn't displayed in the table, it is available on the underlying Student object.

Save and refresh and you should now be able to delete Students for good!

….And you're done! Good work on completing this tutorial! Hopefully you have learned about some new tools and concepts. This tutorial covered a lot of material, but not in a lot of depth. If you would like to find out more about some of the material covered in this tutorial, the following links could be useful for further research.

## Useful Links

### *AppEngine Documentation*

https://cloud.google.com/appengine/docs

### *jQuery Documentation*

https://jquery.com/

### *Google App Engine*

http://tutorialzine.com/2011/01/getting-started-with-google-app-engine/

Note that this tutorial is from 2011, and uses Python 2.5. However, the tutorial is very comprehensive and a very good example of how to implement a small web application in AppEngine.

### *Twitter Bootstrap: Getting Started*

http://getbootstrap.com/getting-started/

### *Knockout JS Live Examples*

http://knockoutjs.com/examples/

## Knockout JS Mapping Documentation

http://knockoutjs.com/documentation/plugins-mapping.html

## Possible Extensions

If you finish early, or would like to expand on this tutorial, the following are some extensions that you could investigate and implement.

If you want to make changes to one of the Students in the Student Register, you would have to delete them, and then create a new Student with the changed details. How could you change the code to allow the editing of the Students in the table? Would this require changes to both the Client Side (HTML/JS) and the Server Side (AppEngine/Python)?

Say that you wanted to include a register for the Courses that Students are enrolled in, how would you do this using AppEngine's DataStore?

## Questions

The following are some questions to consider:

Could a school use this app to keep track of their students' information? What other information and pages might be required for a Student Register application at a real school?

Should the information be secured in some way? If you were using real student's information could you publish this application on the Internet? How could you secure this information to ensure that only people who are authorised could view it?

What is another application that you could make using a REST API by following similar steps to this tutorial? For example, could you make an application that has a list of your favourite recipes, or a to-do list?