Lab Exercise: Creating a Firebase real-time data-store

In a previous lab you used Google App Engine (GAE) to create an online data-store to support mobile client applications. GAE applications require you to use google code (in Python or one of the other supported languages) to handle requests to store data, retrieve data and, if necessary, perform actions that could transform or collect external information. For example, you could arrange it so that requests to your GAE service could gather information from other online services and combine them in some useful way – I've used this method to organize share portfolios, where the share price data has been collected from existing web services: my app simply records share prices and groups them into portfolios for specific users.
In many scenarios, the processing potential of App Engine is not necessary. For example, if you simply wanted retrieve messages sent to a specific group identity (e.g. a chess club) the client app (typically the Javascript in a web-app) could request only messages from the server which had a specific identifier (group="CHESS-CLUB"). To do that, all that would be needed would be a shared store of the data (e.g. messages), and the ability for a client to say which ones are wanted.
The Firebase online database model gives us this, and has the distinct advantage of being much faster in operation than any service based on straightforward web request-response cycles. The firebase.js library that makes and maintains connections to an online data-store does this by using WebSockets to keep a "real-time" connection alive. Rather than the usual HTTP request-response cycle used by a client to gather current data, Firebase provides a PUSH style of notification, where any changes to the data store can be sent directly to all interested clients who are currently connected.
Firebase is now part of Google, which means that it operates of top of the Google web infrastructure. This gives a level of 'respectability' to Firebase data stores – you can assume that Firebase will not disappear in the short term, and that it will be properly resourced; it is a safe prospect for basing an online app on.

Create a Firebase Data Store

1. In a web browser, go to https://www.firebase.com/, have a quick look at the Overview section to get an idea of what you are being offered and then click on Start Building. You will be asked to log in – use a Google (GMail) account if you have one. If you don't already have a Google account, it is worth creating one since this will simplify managing your database in future. Once you log-in you will be asked to allow Firebase to access some parts of your Google account; you need to allow this.
2. You will land on the Firebase Dashboard (headed with "Welcome to Firebase") and a data store will already have been created, called My First App. Your data store will have been given a URL (typically a combination of words and numbers ending in ".firebaseIO.com" – for example, https://resplendent-heat-3736.firebaseIO.com, which is one of my data stores). Figure 1 shows a Firebase dashboard with several data stores.
3. Click on the data URL inside the My First App box (e.g. resplendent-heat-3736.firebaseIO.com) to go directly to the data-store view. Currently this will be an empty data store as shown in figure 2.
4. Firebase data is not stored in tables – it is a NoSQL data manager, which stores its data in nodes as key-value pairs. To create a manageable data-store, it is sensible to

define a node inside the top-level node (your data URL). Move your mouse over the top level node and you will see a green '+' and a red 'x'. The + icon allows you to add a sub-node to a node – click on it and you will be given an edit box where you can add a name and value to your data-store (see figure 3). This is the equivalent of adding a table to an SQL database. Enter the name "messages" and the value 0. The 0 will act as a placeholder until you add some real data to the "messages" node. You must now press the "Add" button to add this node.

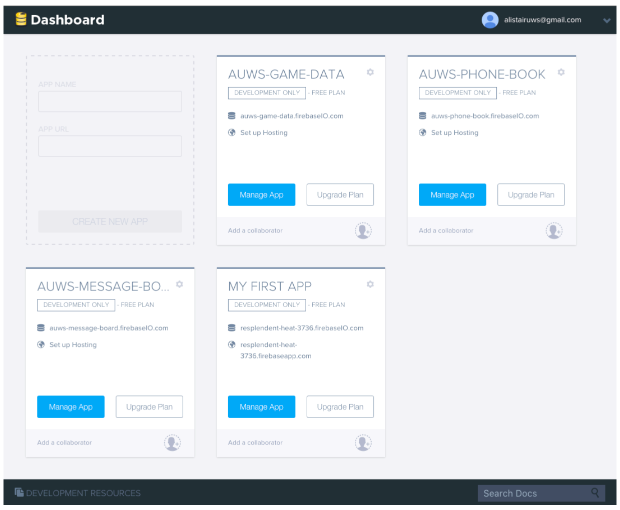
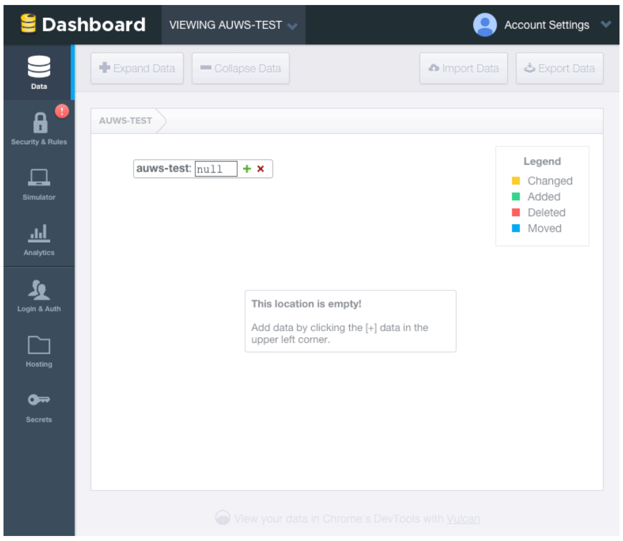


Figure 1: A firebase dashboard with four apps



This location is empty!

Add data by clicking the [+] data in the upper left corner.

Figure 2: The dashboard for a new Firebase data-store

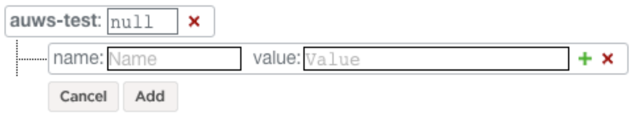


Figure 3: Adding a node to the top level of the data-store.

5. At this stage you have a fully configured data-store with (the NoSQL equivalent of) a single table. Select and copy the URL currently in the browser (this is the location of your entre data-store). It would also be sensible to keep the dashboard page open in a browser.

Create A Web-App

Now that we have a data store to work with, we can write some quick & dirty™ code to work with it.

6. Using WebStorm or some alternative HTML-aware editor (e.g. Brackets), create a new web-app project somewhere in your home directory or on a memory stick and add an HTML file called “index.html” to it. In the first iteration, you will have a single file containing HTML mark-up and Javascript code, although you should move the JS code into a separate file linked to the HTML file once you have the app working.
7. Add the HTML mark-up in listing 1 to your HTML file.

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
    <meta name="viewport" content="width=device-width, initial-scale=1"/>
    <script src="https://cdn.firebase.com/js/client/2.4.0/firebase.js"></script>
</head>
<body>
    <h1>Test Data-Store</h1>
    <label for="name">Name:</label><input type="text" id="name"/><br/>
    <label for="message">Message:</label><br/>
    <textarea id="message" cols="60" rows="10"></textarea><br/>
    <button id="post">Post Message</button><br/>
    <ul id="messages"></ul>
</body>
<script>
</script>
</html>
```

Listing 1: HTML Mark-up for the Firebase test app (index.html)

If you serve this app to a web browser (in WebStorm, right-click on the HTML file name in the project area, and select “Open in Browser”), you will get some idea of how the app will appear. It’s not very inspiring for a mobile app, but once we’ve added the code to make the connection to the Firebase data store, you can deal with that by adding, e.g. the jQuery Mobile framework. Your app should appear as shown in figure 4.

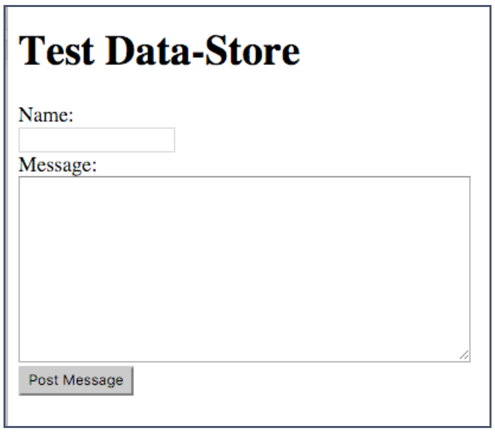


Figure 4: The test app in a browser

We now need to add code to access the Firebase data-store.

8. The first requirement is to create a link to the Firebase store, which will need a new Firebase object. Once we have this object, we can also define an events to react to changes in the data-store over time (which is Firebase’s main selling point). See listing 2 for the JS code needed for this.

```
// Note – put this code inside the <script>...</script> tag...
var fb = new Firebase("https://auws-game-data.firebaseio.com"),
    fbMessages = fb.child("/messages");
    messages_area = document.getElementById("messages");
if (fb) {
    // This gets a reference to the 'location" node.
    var fbMessages = fb.child("/messages");
    // Now we can install event handlers for nodes added, changed and removed.
    fbMessages.on('child_added', function(snapshot){
        var data = snapshot.val();
        console.dir({'added': data});
    });
}
```

Listing 2: Creating a reference to Firebase data and attaching an event handler to it.

An events-handler has been attached to the Firebase object. It will react to any additions to the “messages” node (think “messages table”).

- .on(‘child_added’) will signal the application code if a new sub-node is added to the “/messages” node
- .on(‘value’) will signal the application code passing the complete node to it. This event is more useful to collect a complete set of data from a data-store

These events will provide us with all of the information needed to keep track of messages posted by ANY users of the app – not just changes added locally. We now need to add some code to let us add messages posted by users. Note that some variables have been added to the top of the JS code in listing 2.

- **fb** is our Firebase object – this connects us to the Firebase data store
- **fbMessages** is a reference to a sub-node of the data-store
- **messages_area** is a reference to the <div> element in the HTML mark-up that we will use to display messages. We will add code to the event handler to display the list of messages as they change over time.

Currently, the only browser response to the ‘child_added’ event is that the resulting data will be displayed on the developers’ console; we’ll fix that later. First, we need a very simple function to post a new message to the data store:

```
function postMessage(name, message) {
    fbMessages.push({
        name: name,
        message: message,
        timestamp: Firebase.ServerValue.TIMESTAMP
    });
}
```

Listing 3: Code to post a message to the Firebase data-store

We can attach an event to the ‘post’ button in the user-interface to call this function:

```
document.getElementById('post').addEventListener('click', function(){
    var nameElement = document.getElementById('name'),
        messageElement = document.getElementById('message'),
        name = nameElement.value,
        message = messageElement.value;
    if(name && message) {
        postMessage(name, message);
    }
    messageElement.value = "";
});
```

Listing 4: Posting a message from the U-I

At this stage, you should refresh the app, enter a name, then a message and then press the Post Message button. Open a console window (Ctrl+Shift+I) and you should see that an Object is displayed – click on the ▶ Object arrow to see its contents. If you return to the Firebase Dashboard, you should see that a new child has been added to the 'messages' node and that it contains the message and info pushed to it.
To make our app display messages posted, we will need to add some code to the fbMessages event handler so that posted messages will show up on the page:

```
function showMessage(key, data){
    messageList.appendChild(formatMessage(key, data));
}

function formatMessage(key, data) {
    var li = document.createElement("li");
    li.innerHTML = "<h3>" + data.message + "</h3><p>Posted by:" +
                            data.name + "</p>";
    li.setAttribute("id", key);
    return li;
}
```

Listing 5: Code to update messages on the web page.

Finally, you need to call showMessage() in the messages event-handler and amend the handler to pass key and data values into it. The two values passed as parameters are both accessible from the snapshot in the event handler:

```
fbMessages.on('child_added', function (snapshot) {
    var data = snapshot.val(),
        key = snapshot.key();
    console.dir({'added': data});
    messages[key] = data;
    showMessage(key, data);
});
```

Listing 6: Calling showMessage() from the event handler

While this app is as yet fairly primitive, it still gives us access to a shared data store and incorporates some quite complex behaviour – for example, the app will collect messages posted by ***any*** current users, and it will also recover from a loss of connection and send a message when a connection is re-established. You can read a fuller explanation of Firebase data storage in the appendix at the end of this lab sheet.

Adding Data Filtering

Let's assume that you only want to retrieve new messages sent to a message board. Firebase has a number of functions defined to allow filtering of this type (see Appendix 2 for fuller details). Alter the code from listing 6 as follows:

```
var now = (new Date()).valueOf();
fbMessages.orderByChild('timestamp').startAt(now)
                    .on('child_added', function (snapshot) {
    var data = snapshot.val(),
        key = snapshot.key();
    console.dir({'added': data});
    messages[key] = data;
    showMessage(key, data);
});
```

Listing 7: Filtering the data that a 'child_added' event will request

The intent of the code in listing 7 is to access only new messages added to the server. We start by calculating a date/time value that equates to 'now'. Javascript Date objects are based on the same Unix epoch-based date system, and we can get a suitable date/time value by using the .valueOf() method of the Date type. This is simply a number equating to the number of milliseconds since midnight on the 1st of January, 1970.

Further Filtering

The Firebase.startAt() method is used to give the start of a range, and will work for numbers or strings. To force the event to work for the timestamp values that Firebase provides, we use Firebase.orderByChild() to arrange messages into timestamp order. With the combination of .orderByChild('timestamp') and .startAt(now), Firebase will return only objects which are time-stamped with a value that is more than the 'now' value. A similar combination of .orderByChild() and .endAt() can be used to indicate the end of a range.
Changing the 'messages' event handler definition as shown in Listing 7 will mean that the list of messages will start empty, and only show new messages. We can arrange to add a limited number of messages to the app at start-up, as shown in Listing 8. Add the following event handler inside the **if(fb){…}** block, immediately after the 'child_added' handler:

```
fbMessages.limitToLast(5).once('value', function(snapshot){
    snapshot.forEach(function(data){
        showMessage(data.key(), data.val());
    });
});
```

Listing 8: Picking up a specified number of messages

Note that the function used to set up this event handler is .once(), not the usual .on() function. Firebase uses this to create a single-shot event handler, that will be disabled once it has done its job. The .limitToLast() function in listing 8 lets us say how many messages to get from the end of a collection (i.e. the last N messages). A similar, .limitToFirst() function, can be used to pick up items from the start of a collection (i.e the first N messages).

Exercises

1. By using a combination of .startAt() and .endAt(), you can arrange to pick up only messages posted by people with names in a given range – for example:
   ```
   fbMessages.orderByChild("name").startAt("E")
             .endAt("M").once('value', function(snapshot){
       snapshot.forEach(function(data){
           showMessage(data.key(), data.val());
       });
   });
   ```
   will pick up only messages posted by people with initials "E" through "L". Set this up for your database app, and provide a couple of input boxes where the user can indicate the starting and ending initial letters.
2. You can work out Firebase timestamp values using Javascript dates – e.g.
   ```
   var start = (new Date("2016-03-21")).valueOf(),
       end = (new Date("2016-03-26")).valueOf();
   ```
   Using a pair of <input type="date"> fields, allow the user to select the range of dates that messages should be picked up from and to. You can post messages for a specific

date using the .valueOf() date variables in the messages instead of timestamps to test the feature.

3. You can use the snapshot value in an event to filter words out of messages – for example:

```
fbMessages.on('child_added', function(snapshot){
    var msg = snapshot.val(), mess;
    if(msg.message.indexOf("rudeword") > -1){
        msg.message.replace("rudeword", "********");
        fbMessages.child(snapshot.key()).set(msg); // Rewrite message
        showMessage(snapshot.key(), msg);
    }
});
```

   As the webmaster of a message board, this would be something you would need to do. Using an array of "pseudo-rude" words (e.g. stopWords=["shoat", "clurking", "phunt", "bar-steward", "pemp-slider"]), add code to your app that will automatically replace any of the stop-words that turn up in a message with asterisks. For a complete solution, you would need to replace multiple rude words in a single message ("you clurking phunt"), which will take a bit more effort.

Protecting Firebase Data

In this application, we we are storing messages posted by users. If we needed to keep track of users, the best approach would be to require them to log-in to the database. In Firebase this is a good idea anyway, since allowing unfettered access to anyone who wants it will almost certainly result in the database being destroyed by malice or idiocy.
Every Firebase data store has the capability of maintaining a collection of users, and providing secure access to them automatically. Once User-Login and Authentication has been set up in the dashboard, all the developer has to do is to provide functions to register a new user and log-in an existing one. The Firebase instance object will keep track of logged-in users and you can set up rules to deny access (read and/or write) to a user that is not logged in.
Adding authentication to the Firebase data-app you have been building is described in the lab supplement (Security in Firebase Applications) on Moodle.

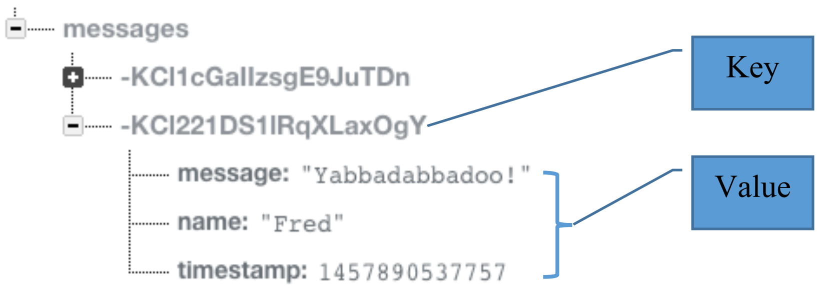
Appendix 1: Firebase storage – some explanation

Firebase stores an entire database as a key-value structure in JSON format. The firebase.js library that your app uses takes care of Javascript objects, converting their data into a JSON format automatically, and returning Javascript objects in the various event handlers as a "snapshot" of data (i.e. the data of a node identified by a particular key, as it was when the event was initiated).
Every Firebase node has a key:value pair. If you do not provide a key (i.e. when you use the .push() method to store a new data element), Firebase automatically generates one. The options for adding data to a store are therefore:

- **<data-node-URL>.push("some new data");**
  Firebase will generate a random key to associate with the data added. Since this key is random, you have no way of predicting its value, but the .push() operation returns the value of the key and the "child_added" event provides it a a property of the snapshot. By .push()ing data to a node, you allow for many data items to exist at the same node. For example:

```
fbMessages.push({“message”: “Yabbadabbadoo!”, “name”: “Fred”,
                           “timestamp”: 1457890537757});
```

The above .push() operation would add a new node to the specified path:



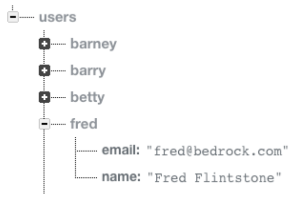
```
messages
  -KCl1cGallzsgE9JuTDn
  -KCl221DS1IRqXLaxOgY
    message: "Yabbadabbadoo!"
    name: "Fred"
    timestamp: 1457890537757
```

Note the random key (-KCI221DS1IRqXLaxOgY) that this data has been assigned to. The timestamp value (1457890537757) is a Unix date/time value, and normally would not be posted in this way – Firebase.ServerValue.TIMESTAMP provides a ‘current’ value which is set at the server.

- **<data-node-URL>.child(“/<a key>”).set(“the associated data”);**
  If you provide a specific key, you need to be sure that it is unique, since if any other data in the Firebase store had the same key, its data would be replaced by the new data. For that reason, you should only use a set operation to store data under a key you know to be unique - for example, a user’s name can make a good key provided you did not allow two users with the same name:

```
    fb.child(“/users/fred”).set({“name”: “Fred Flintstone”,
                           “email”: “fred@bedrock.com”});
```

  Note in the above example, the name “fred” has been used as a key. Fred’s email address would be a better key, since email addresses are guaranteed to be unique to each user. Using the above code would either add a new node under the key “users/fred” or would change the content of “users/fred” to the new values if the node already existed:



```
users
  barney
  barry
  betty
  fred
    email: "fred@bedrock.com"
    name: "Fred Flintstone"
```

  Note that all the user nodes have their content (object data) stored under specific key names like “fred”, “barney” etc.

In short, Firebase stores object data under specified keys, but if a set of objects are to exist at the same level, as would be the case for an array of data in Javascript, a .push() operation can be used to make sure that each object has a distinct key.

Appendix 2: Filtering Messages from the Data Store

Using the methods we’ve looked at so far, you can limit the messages that an .on(‘child_added’) event will deliver from the server by specifying the node you want to get messages from. For example, if a data-store was organized so that all data for a particular user belonging was stored under a specific node (e.g. “users/fred”), only that data would be delivered by an event handler defined as:

```
fb.child('/users/fred').on('child_added', function(sn){...}
```

However, it is much more likely that you will want to retrieve records within a specific node filtered on some more specific criteria. For example, all of the messages sent to a message board since some specific date or time, or all users with an email address at a specific domain. In that case, you need to apply query-style parameters to an event handler. Firebase provides a number of options to modify an .on() event handler. The query processing requires that you request data in a specific order, using an .orderByXXX() request. The ordering options are:

```
.orderByValue( ).on('child_added', function() {…});
.orderByKey( ).on('child_added', function() {…});
.orderByChild(<child-node>).on('child_added', function() {…});
.orderByPriority( ).on('child_added', function() {…});
```

Of these, the most useful is .orderByChild( ) since that equates to a WHERE clause in a SQL style database. For example, the following event handler would request child nodes from the "messages' node in alphabetical order of the name of the poster:

```
fbMessages.orderByChild("name").on('child_added', function(snap) {
    console.dir(snap.val());
});
```

Now that we have created an ordering for results retrieved from the data store, we can apply a filter to that to extract only specific items of data. Filtering can involve the following options:

```
.limitToFirst(<number>).on('child_added', function( ){…});
.limitToLast(<number>).on('child_added', function() {…});
.startAt(<value>).on('child_added', function( ){…});
.endAt(<value>).on('child_added', function( ){…});
.equalTo(<value>).on('child_added', function( ){…});
```

For example, to retrieve only messages posted by "fred", we can use:

```
fbMessages.orderByChild("name").equalTo("fred").on('child_added', function(snap){…});
```

The limitToFirst(N) and limitToLast(N) filters can work on their own (i.e. they don't need an orderBy...() clause) and will simply retrieve the first or last N child nodes – e.g.:

```
fbMessages.limitToLast(5).on('value', function(snapshot){ … });
```

Will retrieve the most recent 5 messages posted. The .startAt(<value>) and .endAt(<value>) filters let you retrieve a range of nodes based on a specific child-node ordering. For example:

```
fbMessages.orderByChild("name").startAt("barney")
                 .endAt("betty").on('value', function(snapshot){…});
```

Will retrieve only messages posted by "barney", "barbara", "bethany", "betty" (if these were people who had posted messages). It would not return messages posted by people whose name came before "barney" or after "betty".