ACIT 3910 -
Database Administration and Management

Firebase Firestore Database

Introduction:

This lab is broken up into 2 sections. The first section creates a few test databases using Google Firebase's Firestore Database and is a walkthrough of some of the functionality you will need for the second section where you will create a simple prototype web app with buttons, scores and a realtime leaderboard for high scores.

Tasks (as a Backend Developer):

1. Setup a Google Firebase Firestore database
2. Make the buttons on the website store the current color into the database
3. Automatically update the color on the website when the database changes
4. Create a user with a score
5. Update the score with the buttons
6. Store the score to the database
7. Display the winning user (top score)
8. Automatically update the winning user on the website when the database changes

Some Firebase Tutorial Videos:

Get to know Cloud Firestore:
https://www.youtube.com/playlist?list=PLl-K7zZEsYLluG5MCVEzXAQ7ACZBCuZgZ

Firebase Firestore Tutorial #1 - Introduction:
https://www.youtube.com/watch?v=4d-gIPGzmK4

Firebase for SQL Programmers #1:
https://www.youtube.com/watch?v=WacqhiI-g_o

Firebase for SQL Programmers #2:
https://www.youtube.com/watch?v=ran_Ylug7AE

Firebase Firestore Database Documentation:

https://firebase.google.com/docs/firestore

https://firebase.google.com/docs/firestore/query-data/listen

Steps:

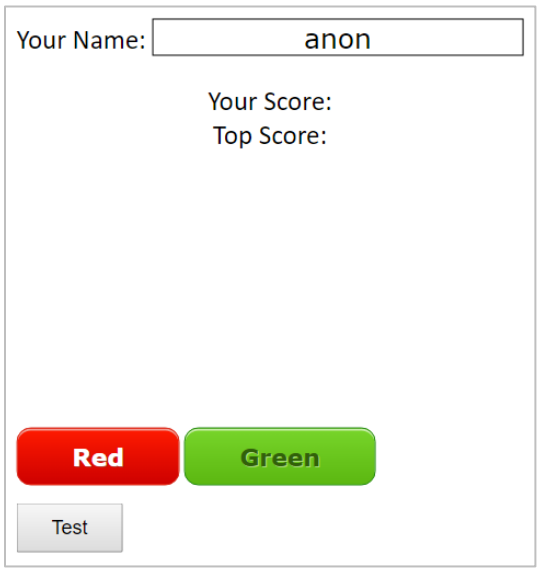
Step 1:
Download the project files from the learning hub (learn.bcit.ca).

Download the following files:

- `index.html`
- `app.js`
- `style.css`

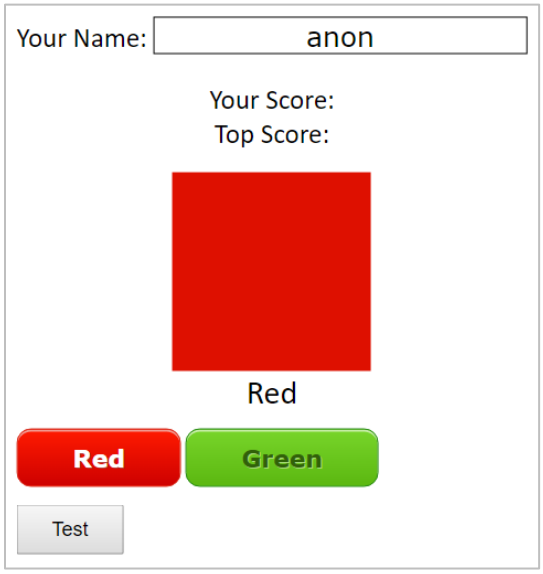
Put them all in the same folder for example `C:\Documents\COMP2930\FirebaseLab\`

Open the `index.html` file in your favourite web browser. You should see the following:

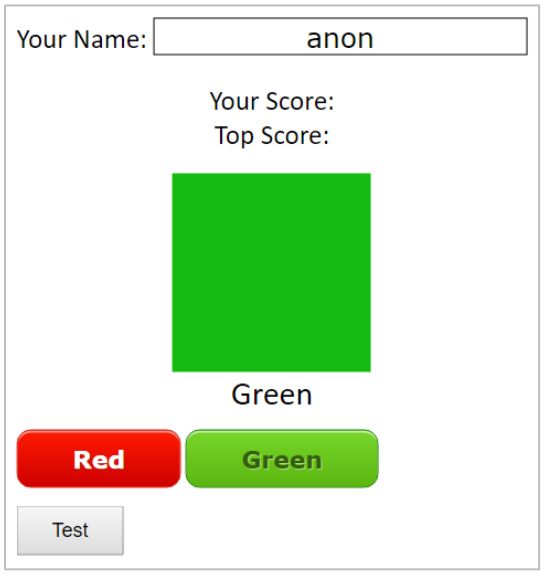


The code right now, if you push the buttons will change the color of the box and the description below to the color corresponding to the button you pressed.

For example, if you press "**Red**" it should look like:



And if you press "**Green**" it should look like:

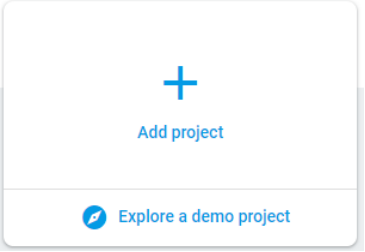


Currently there is no database and no way to sync this color between users. We will change this though!

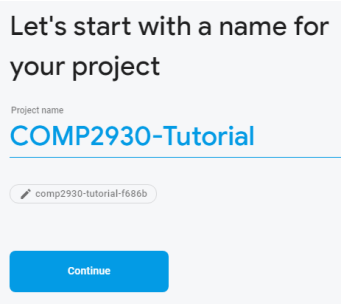
Step 2:
Create a Google Firebase Project and Firestore Database.

Go to https://console.firebase.google.com/ and make sure you are logged in to a google account.

Click on "Add project".

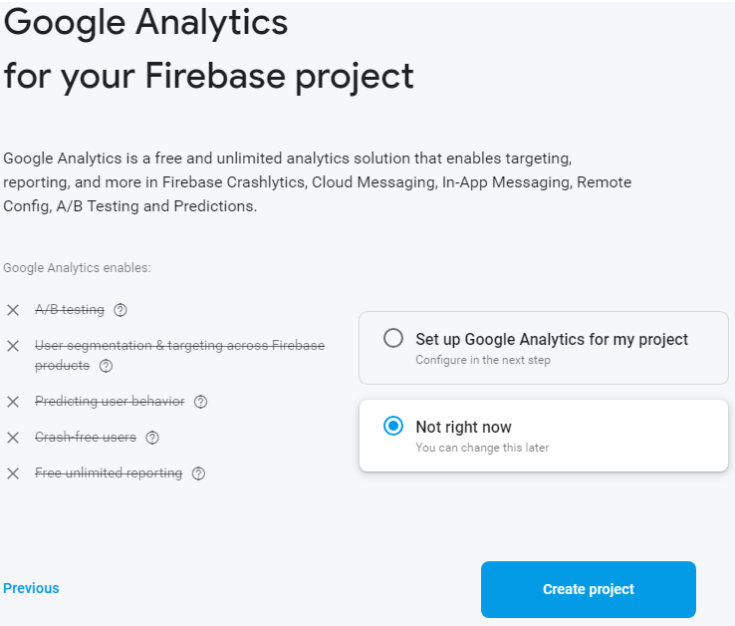


Give your Project a name.



Click "Continue".

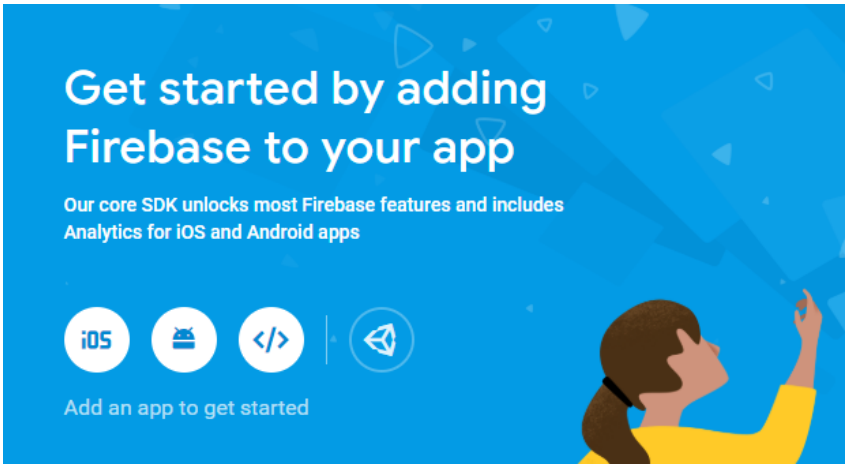
For our project we won't be using Google Analytics so you can skip setting up Google Analytics.

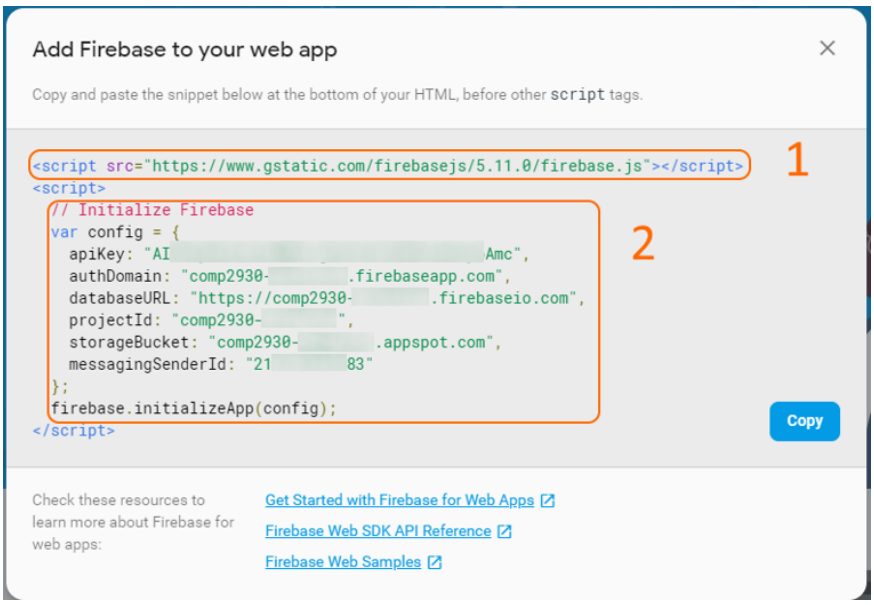


Click "Not right now" and then click "Create project".

Before we create our database, let's capture our project details so that we can connect to it from our web app.

Find the "Web" icon in the "Get started" section of the home screen.





Part 1 is going to go in your `index.html` page in the `<head>` section.
Like so:

```
<head>
    <!--- Put your firebase API js references here: --->
    <script src="https://www.gstatic.com/firebasejs/5.11.0/firebase.js"></script>
    <link rel="stylesheet" href="style.css">
</head>
```

This includes **all of the libraries** required for all possible parts of a Google Firebase project including Database, Storage, Hosting, ML Kit, etc. Some of those things **we don't need** so we are actually going to change the `<script>` tag slightly and add another `<script>` tag like this:

```
<head>
    <!--- Put your firebase API js references here: --->
    <script src="https://www.gstatic.com/firebasejs/5.11.0/firebase-app.js"></script>
    <script src="https://www.gstatic.com/firebasejs/5.10.1/firebase-firestore.js"></script>
    <link rel="stylesheet" href="style.css">
</head>
```

Your 2 script tags should be:

```
<script src="https://www.gstatic.com/firebasejs/8.10.0/firebase-app.js"></script>
<script src="https://www.gstatic.com/firebasejs/8.10.0/firebase-firestore.js"></script>
```

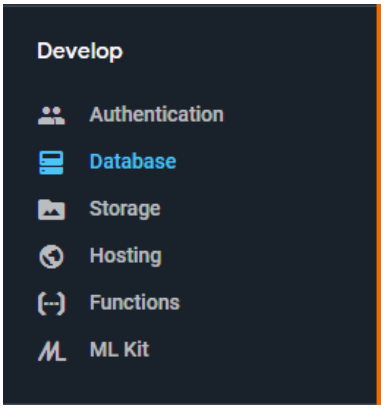
Now we are only including the js files we *actually* need.

Part 2 (the "Initialize Firebase" code) will go in the top of our `app.js` file.

```
// Initialize Firebase
// Get the code provided from Google Firebase's
// Paste the var config = {...}; part here
// Also keep the firebase.initializeApp(...); part
// The <script> </script> portion we will put in the <head> section of our index.html

// Initialize Firebase
var config = {
  apiKey: "AI[redacted]Amc",
  authDomain: "comp2930-[redacted].firebaseapp.com",
  databaseURL: "https://comp2930-[redacted].firebaseio.com",
  projectId: "comp2930-[redacted]",
  storageBucket: "comp2930-[redacted].appspot.com",
  messagingSenderId: "21[redacted]83"
};
firebase.initializeApp(config);
```

Now, let's create a Firestore database.

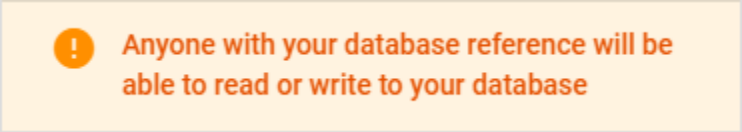
Create your Firestore Database by clicking "Database" in the "Develop" section on the left navigation bar.



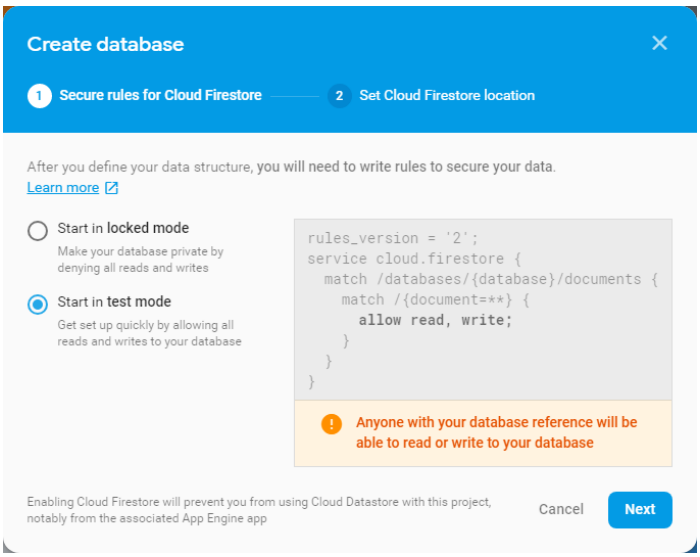
Find the section that says "Firestore Database" and click "Create database".

You can put the database in "test mode" so we don't run into issues with permissions.

We would definitely want to revisit this *if* we were creating a **production ready** application.



Currently **anyone** with our API key can access and re-write *all* of our data in our database!!



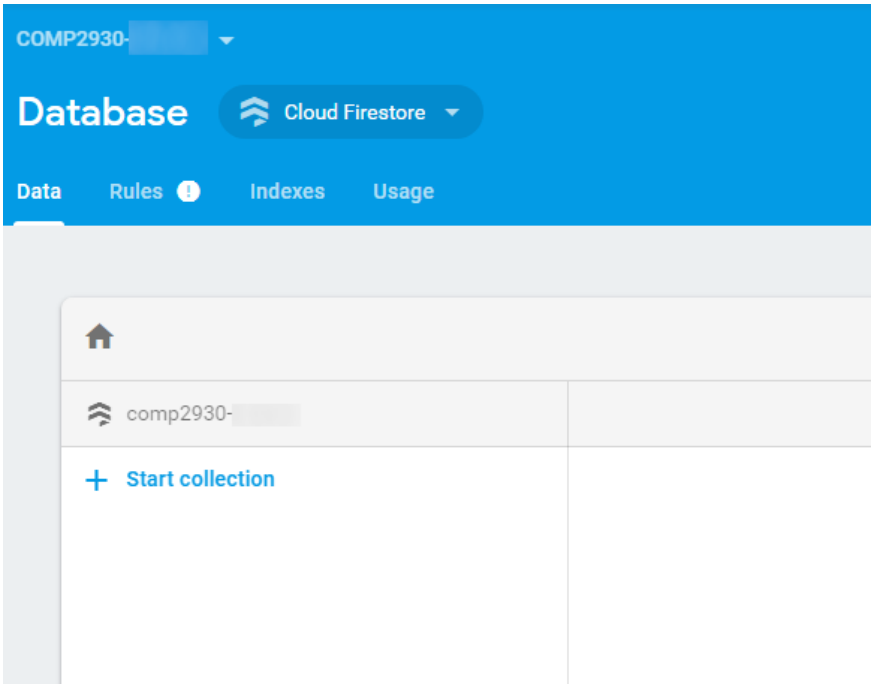
Click "Next".

Next it will ask you about which location to use. For our purposes it doesn't matter so just pick the default.
Click "Done".

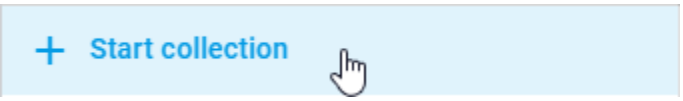
Let me introduce you to your database!

Right now, it's not that exciting… there's noting in it.

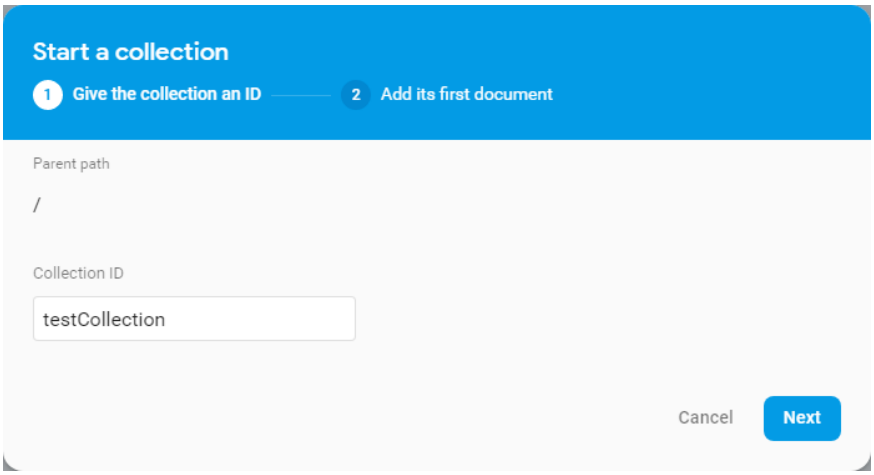
To add data to our database, let's create a new collection.



Click "Start collection".



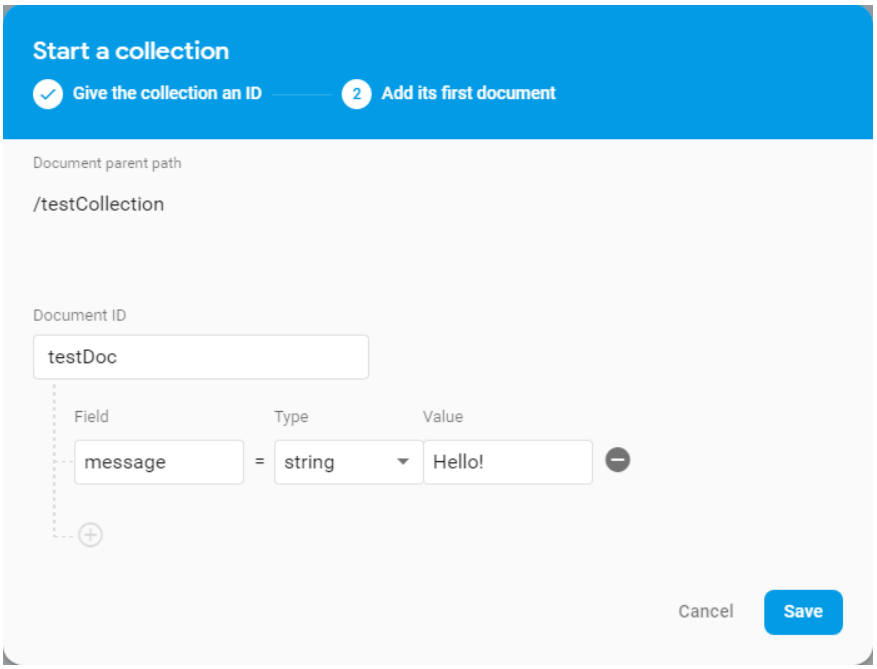
Give your collection a name - call it `testCollection`.



Firestore likes to initialize the collection by creating a document in it.

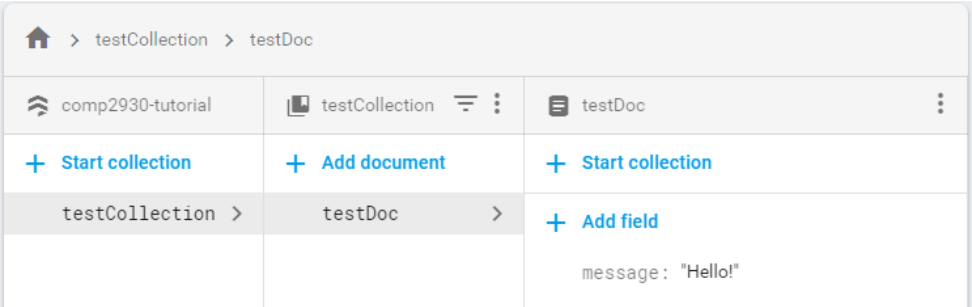
Create a document called `testDoc`

In your document create a field of type `string` called `message` with a value of `"Hello!"`.



Click "Save".

Now you should be able to see you first collection called `testCollection` and your first document called `testDoc` like this:



Step 3:
Listen for realtime updates and set the test message div text to match, when the database changes.

Add the following code after the firestore configuration and initialization:

```
const testMessage = document.querySelector('#testMessage');
const db = firebase.firestore();
const testDBRef = db.collection('testCollection').doc('testDoc');

testDBRef.onSnapshot(doc => {
  let message = doc.data().message;
  testMessage.innerHTML = message;
});
```

In the first line:
`const testMessage = document.querySelector('#testMessage');`
we create a reference to our `div` with id `testMessage` (`<div id="#testMessage">`) so we can modify the text within it later.

Next we create a reference to the database called `db`:
`const db = firebase.firestore();`
And a database reference to our specific collection and document called `testDBRef`:
`const testDBRef = db.collection('testCollection').doc('testDoc');`
This specifies the collection with `.collection('testCollection')` and the document `.doc('testDoc')` to access from within our firebase firestore database (using the `db` database reference created in the line above).

The remainder of the code will listen for updates in the `testDoc` document inside the `testCollection` collection.

The `.onSnapshot(…)` sets up the callback function that will be called every time there is a change to any part of the `testDoc` document. Our callback function takes in a single parameter called `doc` which we can use to extract the data and the message parameter like this: `doc.data().message`.

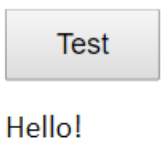
The last line in the callback function:
`testMessage.innerHTML = message;`
sets the text within the `div` with id `testMessage` to be whatever we got back from the data after it was changed.

Your code should look something like this:

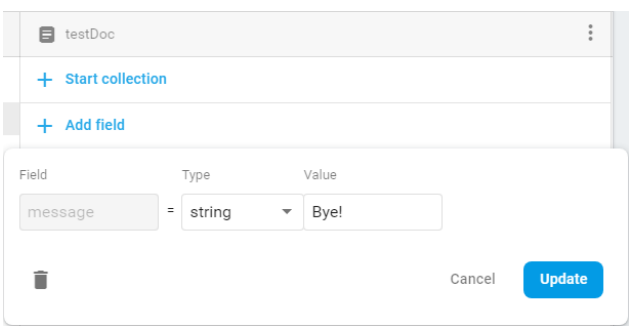
```
firebase.initializeApp(config);

const testMessage = document.querySelector('#testMessage');
const testDBRef = firebase.firestore().collection('testCollection').doc('testDoc');

testDBRef.onSnapshot(doc => {
  let message = doc.data().message;
  console.log(message);
  testMessage.innerHTML = message;
});
```

Test it out by going back to your browser and refreshing `index.html`.

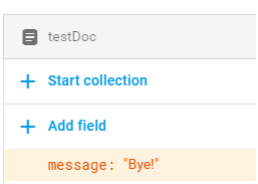
You should see the message `"Hello!"` below the Test button like this:



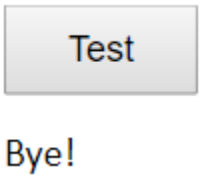
Keep your browser open with `index.html` but switch to your firebase console and change the value of `message` in `testDoc` to `"Bye!"`.



Click "Update".



Switch back to `index.html` and you should see the message automatically switch to `"Bye!"` ***without*** having to refresh the page!! How awesome is that?



Step 4:
Update the `message`'s values in the `testDoc` when Test button is pressed.

We now have a way to see the message currently in the database, but we don't have any way to change its value from within the website. Let's add an event listener to the Test button and update the value in the database.

Here is some code to change the value of the `message` string inside the `testDoc`:

```
const testButton = document.querySelector('.myButtonTest');

testButton.addEventListener('click', (e) => {
  e.preventDefault();
  e.stopPropagation();

  var JSONobj = {};
  JSONobj.message = "Yes!";
  testDBRef.update(JSONobj);
});
```

Let's break this down and understand it in smaller pieces.

The line:
`const testButton = document.querySelector('.myButtonTest');`
creates a reference to Test button. This makes it a bit easier to reference the button later when we want to add an event listener to it.

Next comes the event listener with `testButton.addEventListener(…)`. This creates a callback function which gets called when ever the Test button is pressed. The event type for a button press is `'click'` and specified with the first parameter set to `'click'`.

Inside the event callback the first thing we do is:
`e.preventDefault();`
and
`e.stopPropagation();`
this prevents any default button and any parent events that might get fired.
A form button will go to a new page and submit the form contents. In our case we don't want this, so we use `e.preventDefault()` and `e.stopPropagation()` to stop this from happening.

Next we create a temporary JSON object to reflect the changes we want to see in the `message` string.

```
var JSONobj = {};
```

and set the `message`'s value to "Yes!".

```
JSONobj.message = "Yes!";
```

Now we can update the database to with what we have in our temporary JSON object.

```
testDBRef.update(JSONobj);
```

Remember that `testDBRef` points to our `testDoc` in our `testCollection` from our previous reference declaration.

Here we are using the `.update(…)` method which *will only change* the `message` string within our document.

We could also use the `.set(…)` method here as well, however, it *would replace all of the document* with just `message = "Yes!"`. If there were any other key/value pairs in our JSON object, they would be **deleted**.

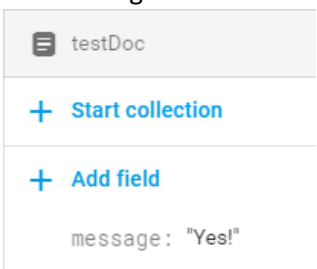
Your code should look something like this:

```
const testButton = document.querySelector('.myButtonTest');

testButton.addEventListener('click', (e) => {
  e.stopPropagation();

  var JSONobj = {};
  JSONobj.message = "Yes!";
  testDBRef.update(JSONobj);
});
```

Test it by going back to index.html refreshing the page and clicking the button. You should see the message on the page change to "Yes!".

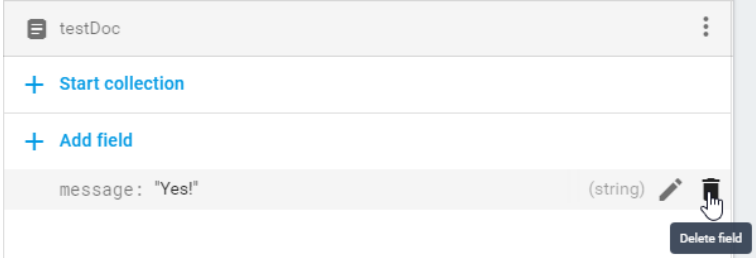
Yes!

The message in the database changes as well!

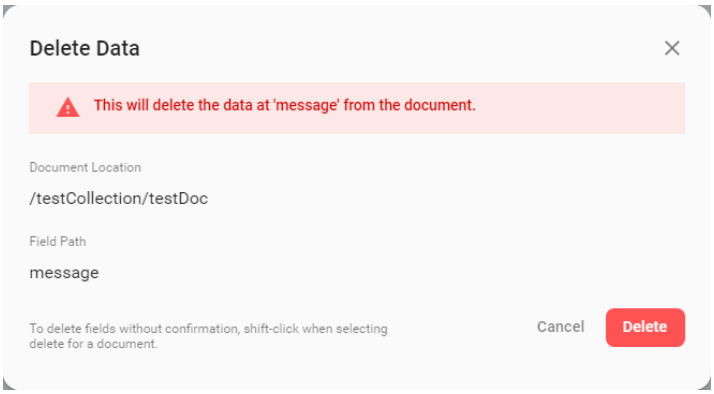


Step 5:
Modify your `testDoc` document with a slightly more complicated document.

First let's delete the `message` field in the `testDoc` document.

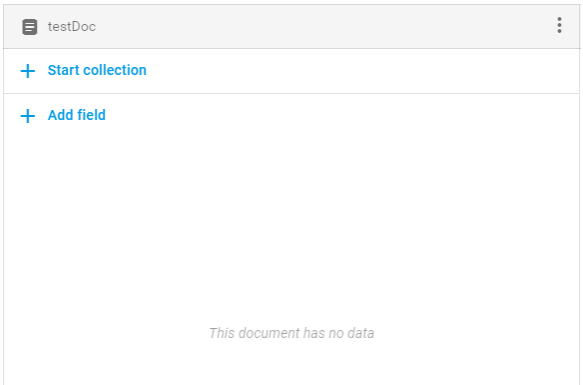


Click the "Delete Field" trash icon next to the `message` field.

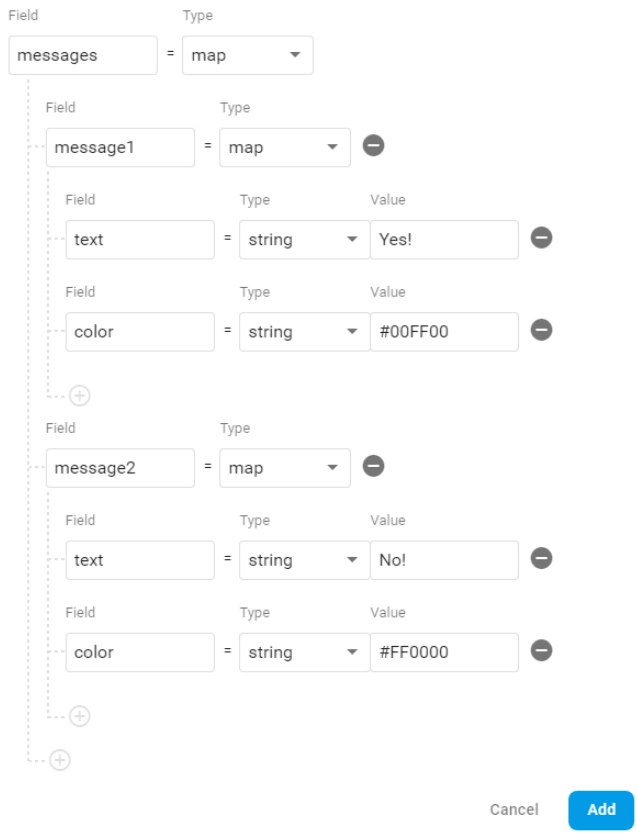


Click "Delete" to confirm.

Firestore will confirm that your `testDoc` is empty:

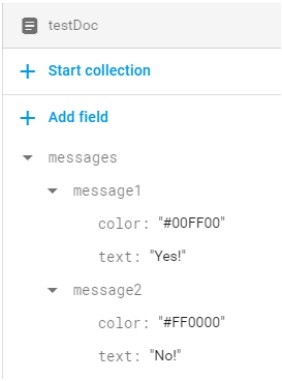


Click on "Add field" and create the following structure:



Click "Add".

Your firestore `testDoc` document should look like this:



Step 6:
Update our website to match the new firestore database data structure.

We'll take both the message's `text` field and display `messages.message1.text` in the `div` with id `testMessage` and display `messages.message2.text` in the `div` with id `testMessage2`. Also we'll change the color of the text in each `div` to match the `color` field in each message object.

Add a reference to the `div` with id `testMessage2`:
```
const testMessages = document.querySelector('#testMessage2');
```

Replace the code for the realtime update callback function to this:

```
testDBRef.onSnapshot(doc => {
  let message = doc.data().messages.message1.text;
  let message2 = doc.data().messages.message2.text;
  let color = doc.data().messages.message1.color;
  let color2 = doc.data().messages.message2.color;

  testMessage.innerHTML = message;
  testMessage2.innerHTML = message2;
  testMessage.style.color = color;
  testMessage2.style.color = color2;
});
```

Now instead of getting the single message from `doc.data().message` we need to dig deeper into our JSON object to find the `text` field of `message1` inside the `messages` object like this:
```
  let message = doc.data().messages.message1.text;
```

We also want to retrieve the 2nd message's text:
```
  let message2 = doc.data().messages.message2.text;
```

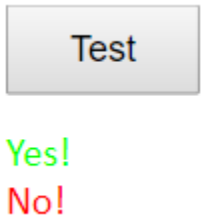
Also grab the associated colors for each of the messages:
```
  let color = doc.data().messages.message1.color;
  let color2 = doc.data().messages.message2.color;
```

Now we can set the text for each of the `div`s and set the colors of the text using the `style.color` property:
```
  testMessage.innerHTML = message;
  testMessage2.innerHTML = message2;
  testMessage.style.color = color;
  testMessage2.style.color = color2;
```

Your code should look something like this:

```
21  const testMessage = document.querySelector('#testMessage');
22  const testMessages = document.querySelector('#testMessage2');
23  const testDBRef = firebase.firestore().collection('testCollection').doc('testDoc');
24
25  testDBRef.onSnapshot(doc => {
26    let message = doc.data().messages.message1.text;
27    let message2 = doc.data().messages.message2.text;
28    let color = doc.data().messages.message1.color;
29    let color2 = doc.data().messages.message2.color;
30
31    testMessage.innerHTML = message;
32    testMessage2.innerHTML = message2;
33    testMessage.style.color = color;
34    testMessage2.style.color = color2;
35  });
```

And your page should now look like this:



This fixes the realtime update callback with the new data structure, but our button pressed event will still be broken unless we fix it.

To fix the Test button, we want to update `messages.message1.text` to `"Maybe"` and change `messages.message1.color` to `#0000FF`.
We do **not** want to change `messages.message2` at all.

Replace the code for the test button event listener to this:

```
testButton.addEventListener('click', (e) => {
  e.preventDefault();
  e.stopPropagation();

  var JSONobj = {};
  var messageObj = {};
  messageObj.text = "Maybe";
  messageObj.color = "#0000FF";
  JSONobj = {
    "messages.message1" : messageObj
  };
  testDBRef.update(JSONobj);
});
```

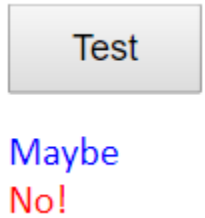
Using this code we create a `messageObj` that contains what we want to change in the `messages.message1` portion. By specifying `messages.message1` as the key (using firebase's special dot notation) and `messageObj` as the value in the `JSONobj` for the `.update(..)` method, we ensure that only `messages.message1` is changed.
The rest of the `testDoc` document that contains `messages.message2` is left **unchanged**.

For more information on the difference between .set(…) and .update(…) and the dot notation for nested objects read the firebase documentation here:
https://firebase.google.com/docs/firestore/manage-data/add-data#update_fields_in_nested_objects

Your code should look something like this:

```
testButton.addEventListener('click', (e) => {
  e.stopPropagation();

  var JSONobj = {};
  var messageObj = {};
  messageObj.text = "Maybe";
  messageObj.color = "#0000FF";
  JSONobj = {
    "messages.message1" : messageObj
  };
  testDBRef.update(JSONobj);
});
```

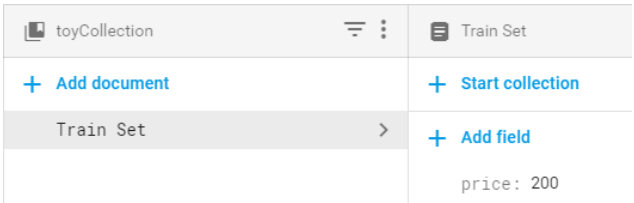
And if you push the Test button on your `index.html` page (after refreshing it) should look like this:



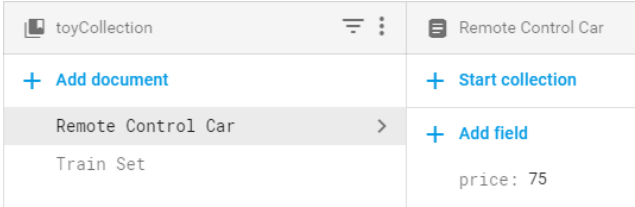
Step 7:
Create a new collection for toys.

Create a new collection called `toyCollection`.

Create the `Train Set` document inside the `toyCollection` collection.
Give the `Train Set` document a single field of type `number` with the value of `200`.



Create another document called `Remote Control Car` inside the `toyCollection` collection. Give the `Remote Control Car` document a single field of type `number` with the value of `75`.



Step 8:
Create a realtime listener for the most expensive toy.

Just like we can add realtime listeners on a single document, we can also add realtime listeners to collections. What's more, we can filter and sort the results we get back from our collection to only listen to specific *parts* of our collection.

If we wanted to listen to any change in any document within our collection we could use the `onSnapshot(…)` function to attach a listener callback function like this:

```
db.collection('toyCollection')
.onSnapshot(querySnapshot => {
});
```

The `querySnapshot` parameter will contain a list of each document in our collection and we could use a `forEach` callback to iterate through the documents like this:

```
db.collection('toyCollection')
.onSnapshot(querySnapshot => {
  querySnapshot.forEach(doc => {
  });
});
```

If we are only interested in some of the documents in our collection we can use the `.where(…)` function to filter out certain documents.

Example:

```
db.collection('toyCollection').where('price','==',75)
.onSnapshot(querySnapshot => {
  querySnapshot.forEach(doc => {
  });
});
```

Firestore query reference: https://firebase.google.com/docs/firestore/query-data/queries

If we care about the order in which the documents are returned, we can use the `.orderBy(…)` function to sort the results either ascending or descending. This will use a compare the value of the key specified in the order by and sort each of the documents.

Example:
```
db.collection('toyCollection').orderBy("price")
.onSnapshot(querySnapshot => {
  querySnapshot.forEach(doc => {
  });
});
```

If we want to limit the number of documents returned we can use the `.limit(…)` function.

Example:
```
db.collection('toyCollection').limit(1)
.onSnapshot(querySnapshot => {
  querySnapshot.forEach(doc => {
  });
});
```

In the above example `.limit(1)` would ensure that only 1 document was returned from the `toyCollection`.

Firestore ordering and limits reference:
https://firebase.google.com/docs/firestore/query-data/order-limit-data

Keep in mind that many of the firestore collection functions mentioned above (the `.where(…)`, `.orderBy(…)` and `.limit(…)` functions) can be combined to retrieve just the documents you are interested in.

Example:
```
db.collection('toyCollection').where('price','==',75).limit(1)
.onSnapshot(querySnapshot => {
  querySnapshot.forEach(doc => {
  });
});
```

The above example would return only 1 document that had a `price` equal to 75.

Use the following code to create a realtime listener to display the most expensive toy.

```
const mostExpensive = document.querySelector('#mostExpensive');

db.collection('toyCollection').orderBy('price', "desc").limit(1)
.onSnapshot(querySnapshot => {
  mostExpensive.innerHTML = '';
  querySnapshot.forEach(doc => {
    mostExpensive.innerHTML += 'The most expensive toy is: '+doc.id+'
$'+doc.data().price;
  });
});
```

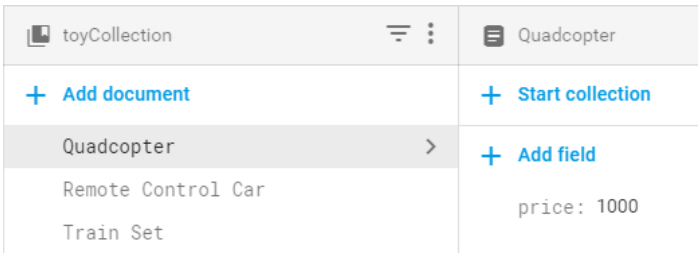
This will query the `toyCollection` collection and return the only the first document when ordering by the price descending (most expensive first). It will replace the text of the `div` with id `mostExpensive` with the name of the toy (using `doc.id`) and its `price`.

You should see this on the website below the Test button:

The most expensive toy is: Train Set $200

Go ahead and add a new toy to the collection - Quadcopter with a price of 1000.

Your `toyCollection` collection should now look like this:



And the website should automatically update to show this:

The most expensive toy is: Quadcopter $1000

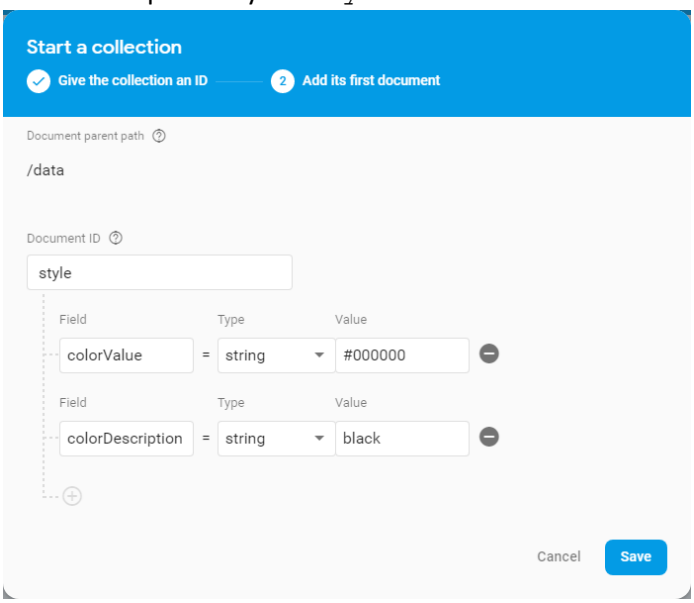
Step 9:
Store the color of the button that the user pressed to the database.

If the user presses the "**Red**" button we want to store **Red** to the database, and if the user presses "**Green**" we will store **Green** to the database.

Create a collection called `data`.

Create a document called `style`.

Create a map inside your `style` document with the following key/value pairs:



Using the `data` collection and the `style` document:

Implement the storing of **Red** to the database when the **Red** button is pressed.

Use "`#de1000`" as the `colorValue`.
Use "`Red`" as the `colorDescription`.

Implement the storing of **Green** to the database when the **Green** button is pressed.

Use "`#15ba10`" as the `colorValue`.
Use "`Green`" as the `colorDescription`.

Step 10:
Examine the code that has been used for the **Red** and **Green** buttons and implement an **Orange** button.

The stylesheet should already include the code for styling an orange button.

Add the HTML required to create an **Orange** button (use the naming convention found in the `style.css` file).

Add the required JavaScript code to make the **Orange** button functional. It **must**, change the color on the page and in the database (just like the **Red** and **Green** buttons did).

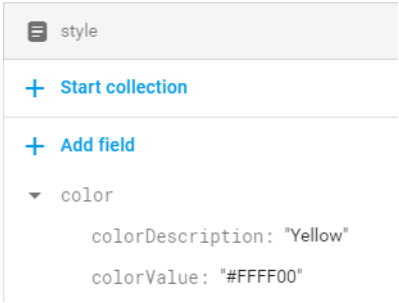
Use "`#de7610`" as the `colorValue`.
Use "`Orange`" as the `colorDescription`.

Step 11:
Respond to realtime updates from changes to the database.

Our goal is to make sure that any time the database changes (and a new color is put in our color JSON object part) that we update our website. This could happen if another user browses to the page and clicks a button or if we change the value directly in the database from within the google firebase administration page.

You should now have a site that responds automatically to whenever the color in the database changes!

Test it out by changing the database:
Use "`#ffff00`" as the `colorValue`.
Use "`Yellow`" as the `colorDescription`.



And you should see the website automatically update (without having to refresh the page) to this:

Yellow

Step 12:
Create a score variable.

Create a variable in your `app.js` called `score`. Set its initial value to `0`.

Step 13:
Add code to your **Red**, **Green** and **Orange** buttons to update the `score` value.

The **Red** button should **subtract 5 points**.

The **Green** button should **add 5 points**.

The **Orange** button should **reset** the score **to 0 points**.

Update the value of `span` with id `my-score` with the current score.

Example (after pressing the **Green** button twice):

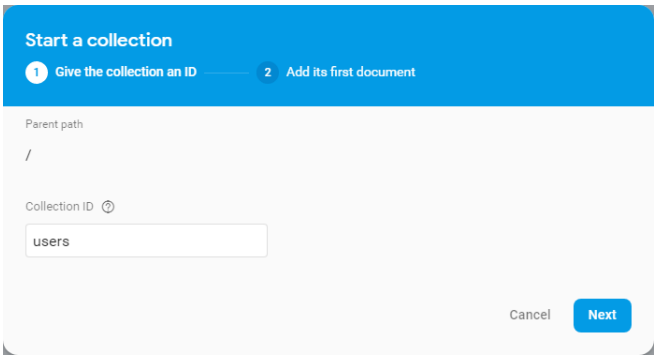
Your Score: 10
Top Score:

Example 2 (after pressing the **Orange** button and then the **Red** button):

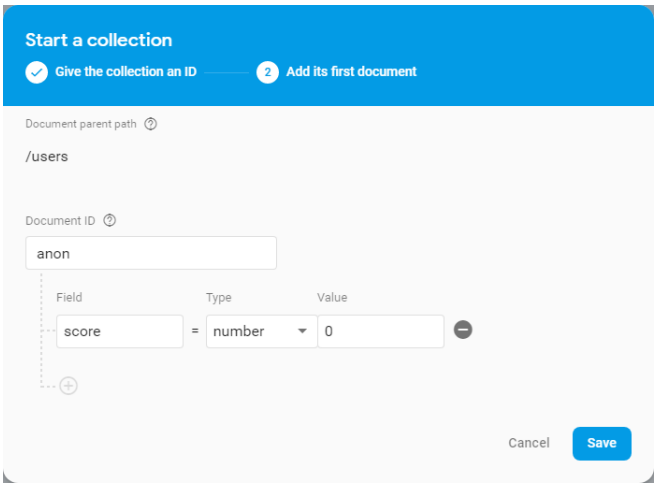
Your Score: -5
Top Score:

Step 14:
Create a firestore collection to store a list of users who have played the game and their current score.

Create a new collection called `users`.



Create a document called `anon` with a field called `score` (type: number) with a value of `0`.



Step 15:
Use the current value of the textbox as the document name inside the `users` collection.

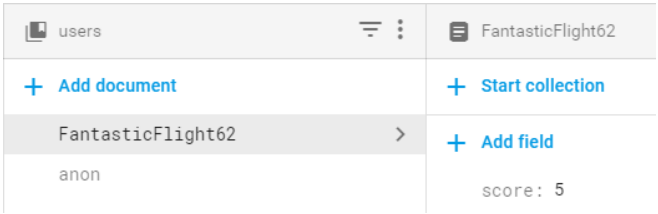
When the **Red**, **Green** and **Orange** buttons are pressed and the score is updated, the score should be updated in the database as well. Use the input field with id `username` as the document name in the `users` collection.

For example, if the website has FantasticFlight62 as the username in the textbox like this:

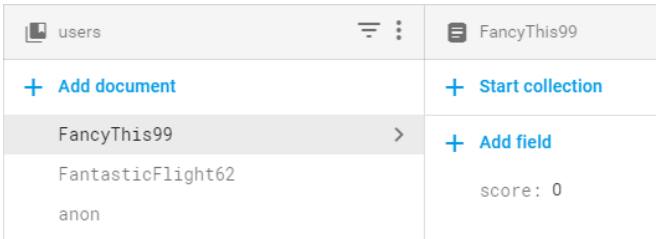
Your Name: FantasticFlight62

Your Score: 5

and the **Green** button is pressed, the score should be 5 and stored to the firestore database like this:



If later, I decide to change my username to FancyThis99 and press the **Orange** button the firestore database should look like this:



Step 16:
Add the leaderboard (the top score).

Create a realtime listener for the user with the top score. Display the top score and who currently holds the highest score in the `span` with id `top-score`.

Your Name: FantasticFlight62

Your Score: 20
Top Score: 20 by FantasticFlight62

Currently the scores for the players are:

FantasticFlight62: score = **20** ← **Leader (highest score)**
FancyThis99: score = 10
anon: score = 15

If FancyThis99 gets 15 more points, they'll be in the lead with 25 points and should show up as the leader in the leaderboard:

Your Name: FantasticFlight62

Your Score: 20
Top Score: 25 by FancyThis99

Now the scores for the players are:

FantasticFlight62: score = 20
FancyThis99: score = **25** ← **Leader (highest score)**
anon: score = 15

Marking Criteria:

Criteria	Marks

Proper implementation of the **Red**, **Green** and **Orange** buttons that will change the color shown on the page **and** will update the Firestore database (collection `data` and document `style`) with the values:
Red:
`color.colorDescription = "Red"`
`color.colorValue = "#de1000"`
Green:
`color.colorDescription = "Green"`
`color.colorValue = "#15ba10"`
Orange:
`color.colorDescription = "Orange"`
`color.colorValue = "#de7610"`
 	
Any change to the color value and/or description in the database ***must also*** automatically update the website's color and text values.
	4 marks

Red, **Green** and **Orange** buttons properly adjust the score for the current user. • **Red** subtracts 5 points. • **Green** adds 5 points. • **Orange** resets to 0 points. Use the text input field as the user's name and store their current score in the firestore database. Store the each of the user's scores in a collection called `users` with each of the user's name as the document name.	**4 marks**
Implement the leaderboard to show the top scoring user. Any change to the user's scores in the database ***must*** automatically update the website's top scoring user (without refreshing the page). Ex: if someone new get the highest score, their score and name should show as the highest score on the website.	**2 marks**
Note: You will receive a **mark of 0** if your code doesn't compile (i.e. too many errors to run properly) or if your database is not connected properly!	
Total:	**10 marks**

Submission Requirements:

Submission:	File name:
A Single Zip file containing: • HTML File (index.html) • Javascript Application (app.js) • CSS Styles (style.css)	code.zip

Note: There are no changes required to the style.css file.

IMPORTANT! Your code will be tested to see if it works! Please ensure that the firebase firestore database has read and write permissions and is configured properly.