Cloud Firestore

Cloud Firestore is a flexible, scalable database for mobile, web, and server development from Firebase and Google Cloud. Like Firebase Realtime Database, it keeps your data in sync across client apps through real-time listeners and offers offline support for mobile and web so you can build responsive apps that work regardless of network latency or Internet connectivity. Cloud Firestore also offers seamless integration with other Firebase and Google Cloud products, including Cloud Functions.

Check the **official page** for more information.

Setup

Before starting to use any Firebase extensions, you are required to follow some initial configuration steps.

> NOTE The **SDK** version is only available for the Android, iOS and Web targets; if you're targeting other devices please follow the steps for **REST API**.

- **Create Project** (skip this if you already have a project)
- **Firebase Console** (enabling Firebase Firestore)
- **Platform Setup** (configuring SDKs or REST API)

Functions

The Firestore extension was implemented using a fluent-API (method chaining) approach and the entry point for interacting with the database is using the function **FirebaseFirestore**, which will return a ***<dbReference>***, that can be either a ***<dbCollection>*** or a ***<dbDocument>***.

The following functions are given for working with the Firebase Firestore extension:

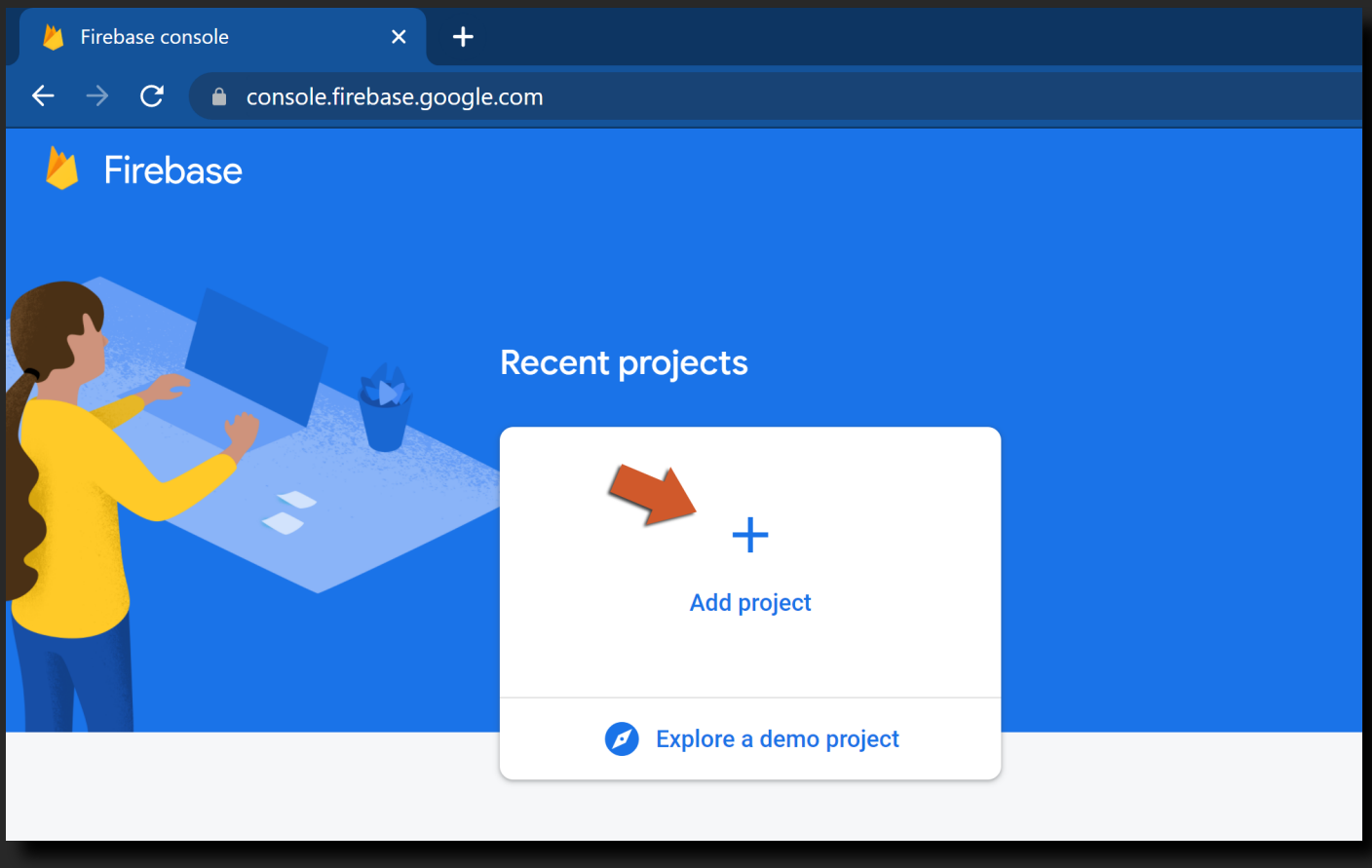
- **FirebaseFirestore** (entry point)
- *<dbReference>*.**Child**
- *<dbReference>*.**Delete**
- *<dbReference>*.**Listener**
- *<dbReference>*.**ListenerRemove**
- *<dbReference>*.**ListenerRemoveAll**
- *<dbReference>*.**Parent**
- *<dbReference>*.**Read**
- *<dbReference>*.**Set**
- *<dbReference>*.**Update**
- *<dbCollection>*.**Query**
    - *The functions given below should only be used for filtering data when querying the database using the* `Query()` *function given above.*
    - *<dbCollection>*.**EndAt**
    - *<dbCollection>*.**Limit**
    - *<dbCollection>*.**OrderBy**
    - *<dbCollection>*.**StartAt**
    - *<dbCollection>*.**Where**
    - *<dbCollection>*.**WhereEqual**

- *<dbCollection>*.WhereGreaterThan
- *<dbCollection>*.WhereGreaterThanOrEqual
- *<dbCollection>*.WhereLessThan
- *<dbCollection>*.WhereLessThanOrEqual
- *<dbCollection>*.WhereNotEqual

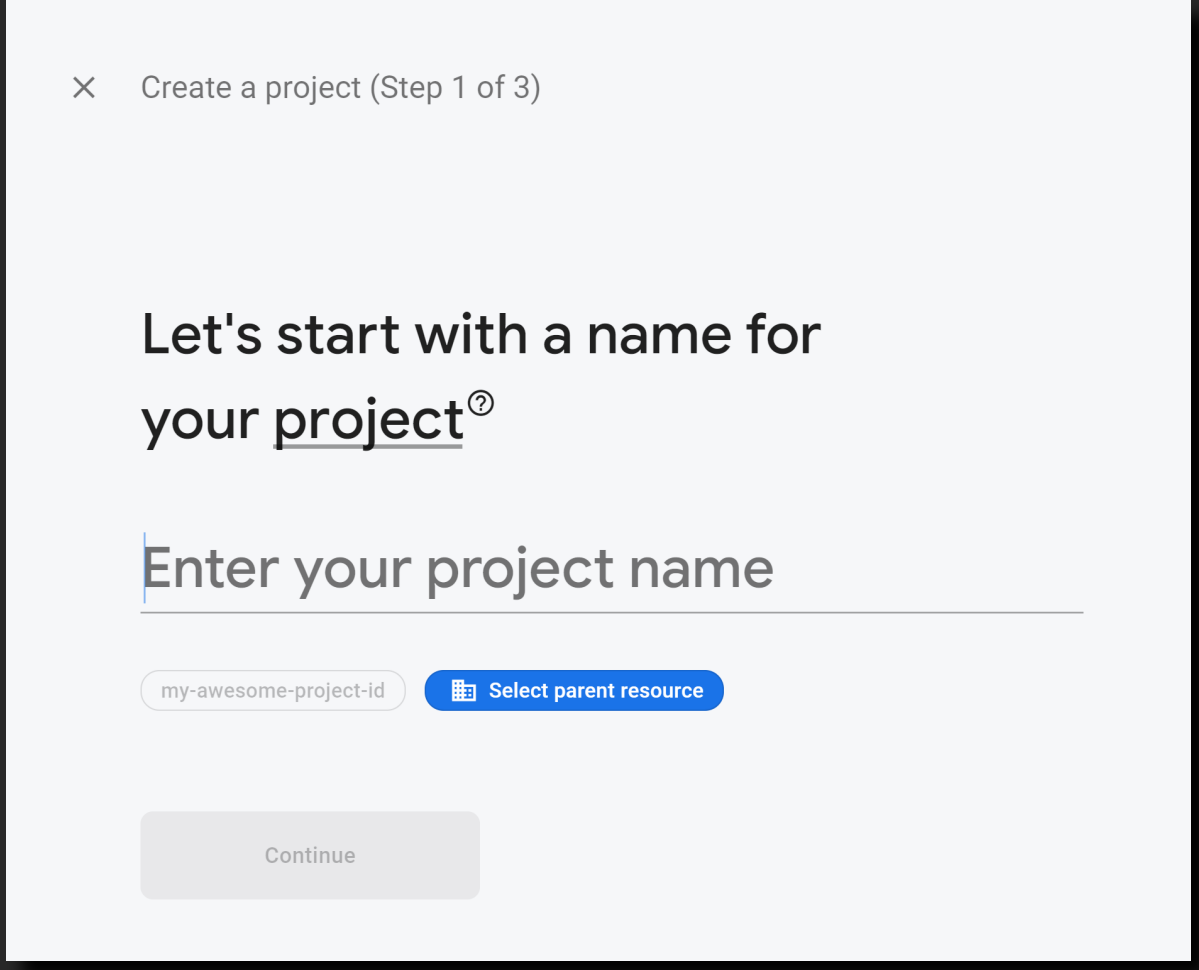
Create Project

Before working with any Firebase functions, you must set up your Firebase project:

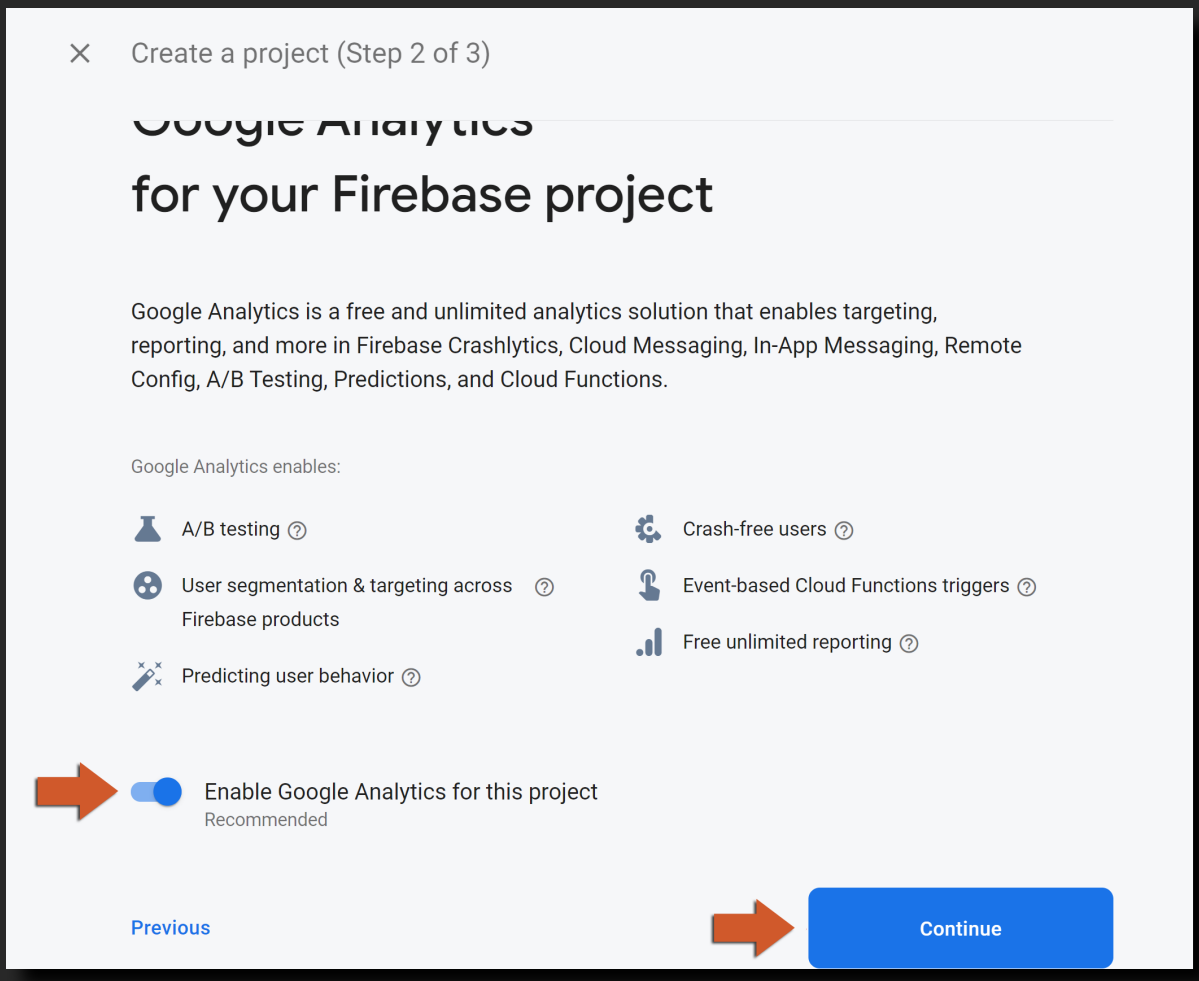
1. Go to the **Firebase Console** web site.
2. Click on **Add Project** to create your new project.



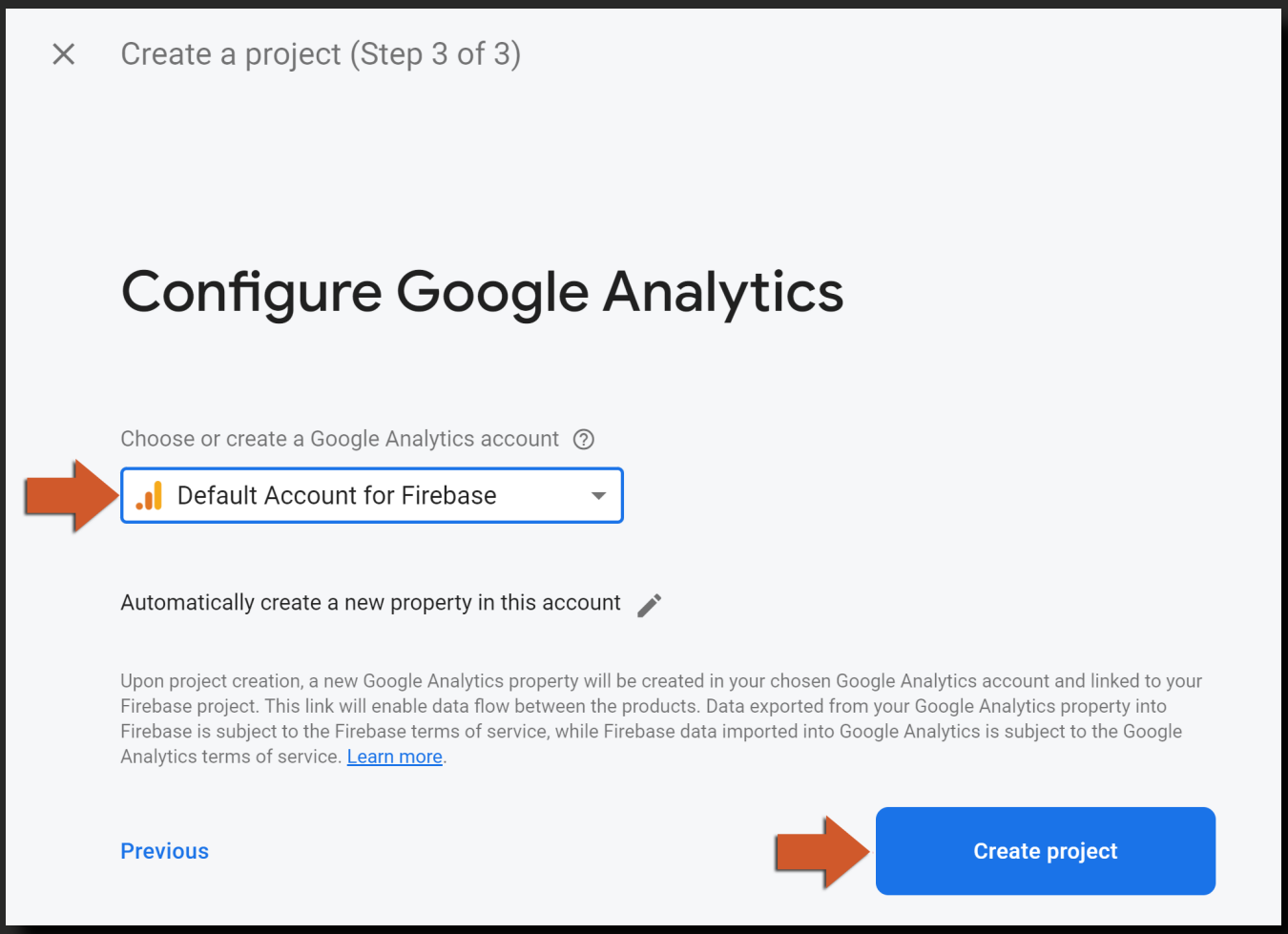
3. Enter a name for your project and click on the **Continue** button.



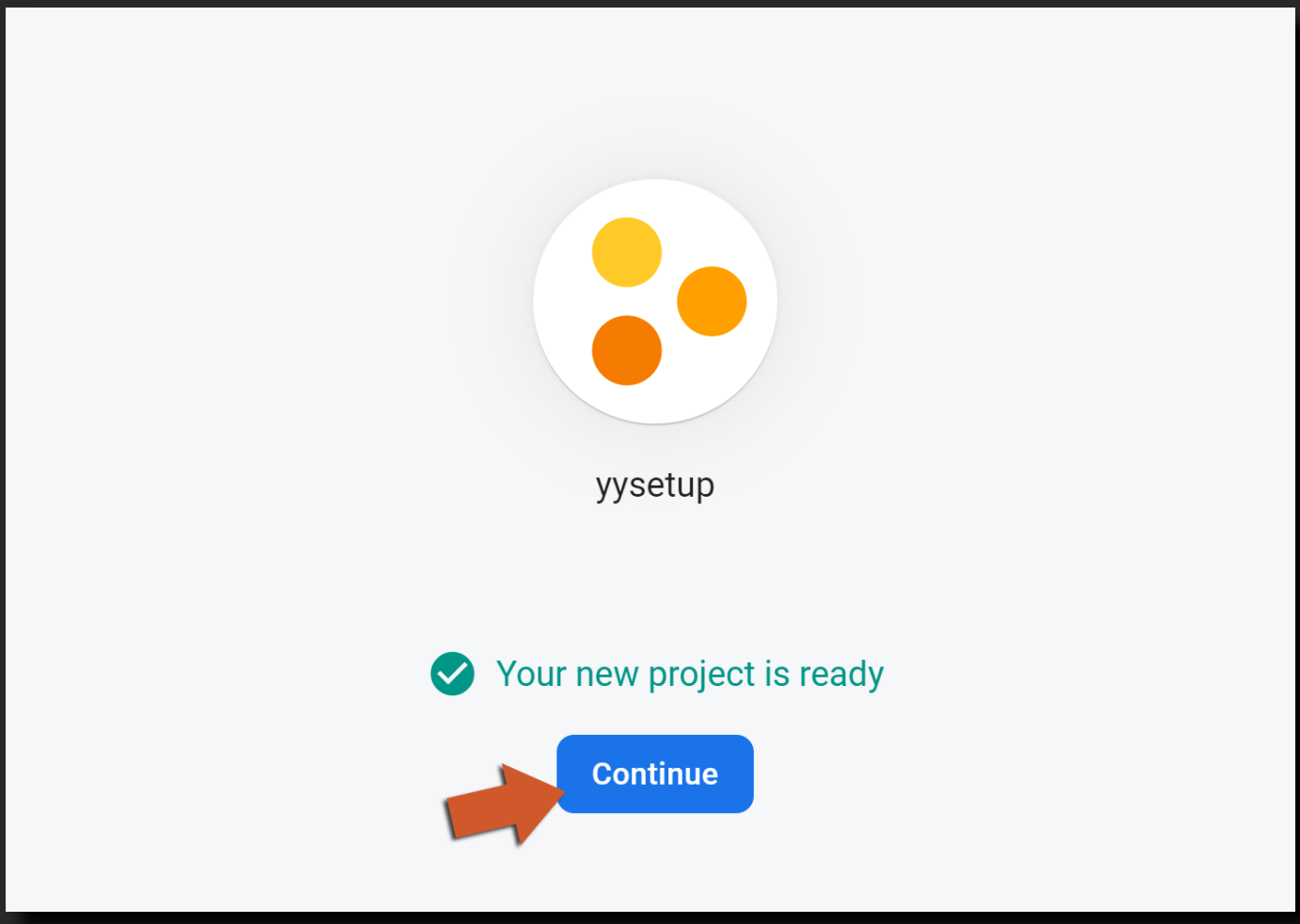
4. On the next page, make sure that **Enable Google Analytics for this project** is enabled and then click the **Continue** button:



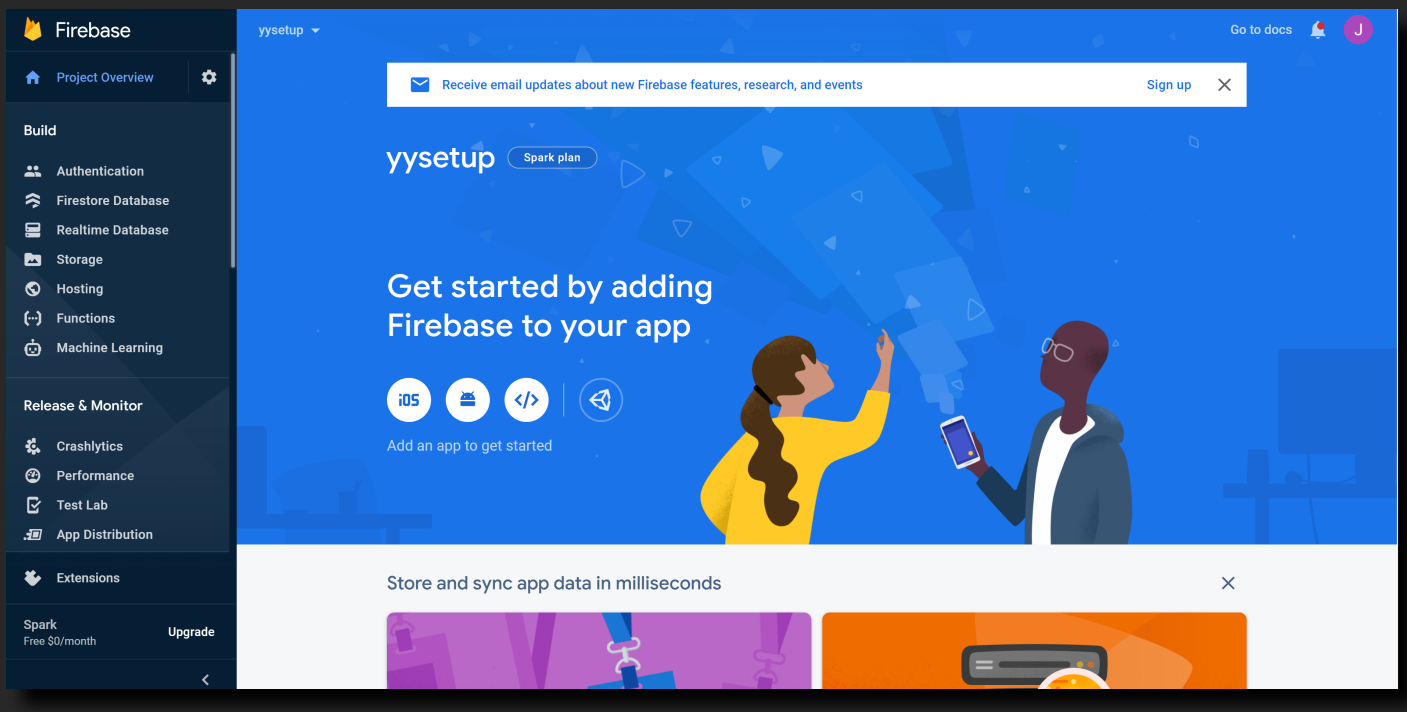
5. Select your account and click the **Create project** button:



6. Wait a moment until you project is created; after a few moments you should see the page shown below:



7. You will now be taken to your new project's home page:

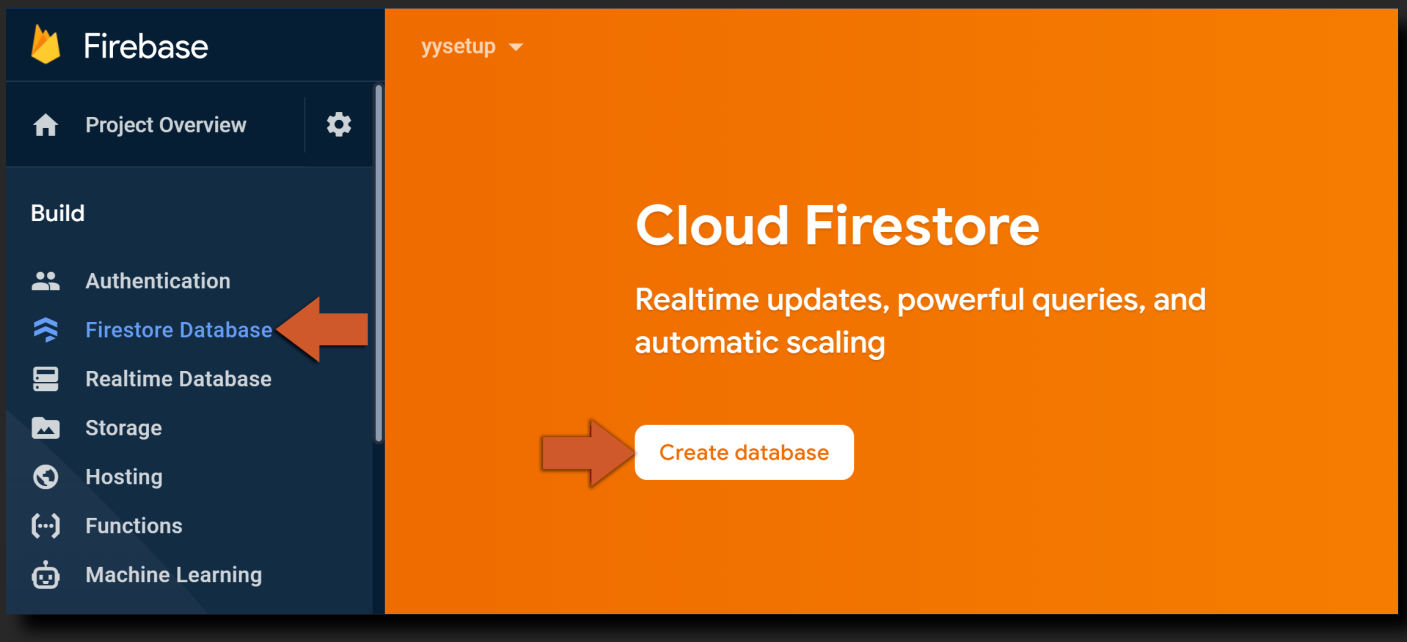


8. Continue your adventure with the Firebase extensions provided for GameMaker!

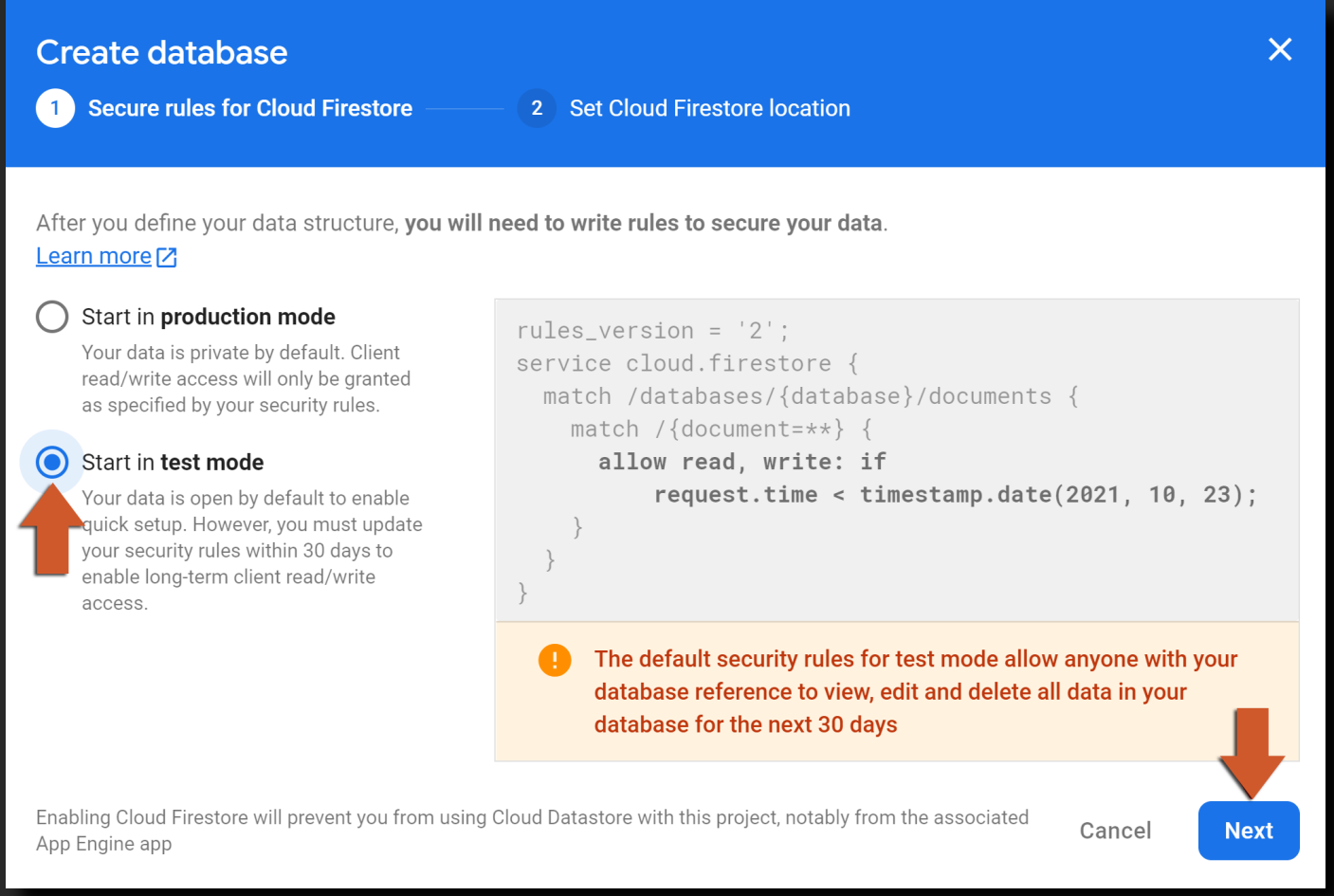
Firebase Console

Before being able to use the Firebase Firestore extension we need to configure a new database to work with in the **Firebase Console**. Follow the steps below to get your first database set up.

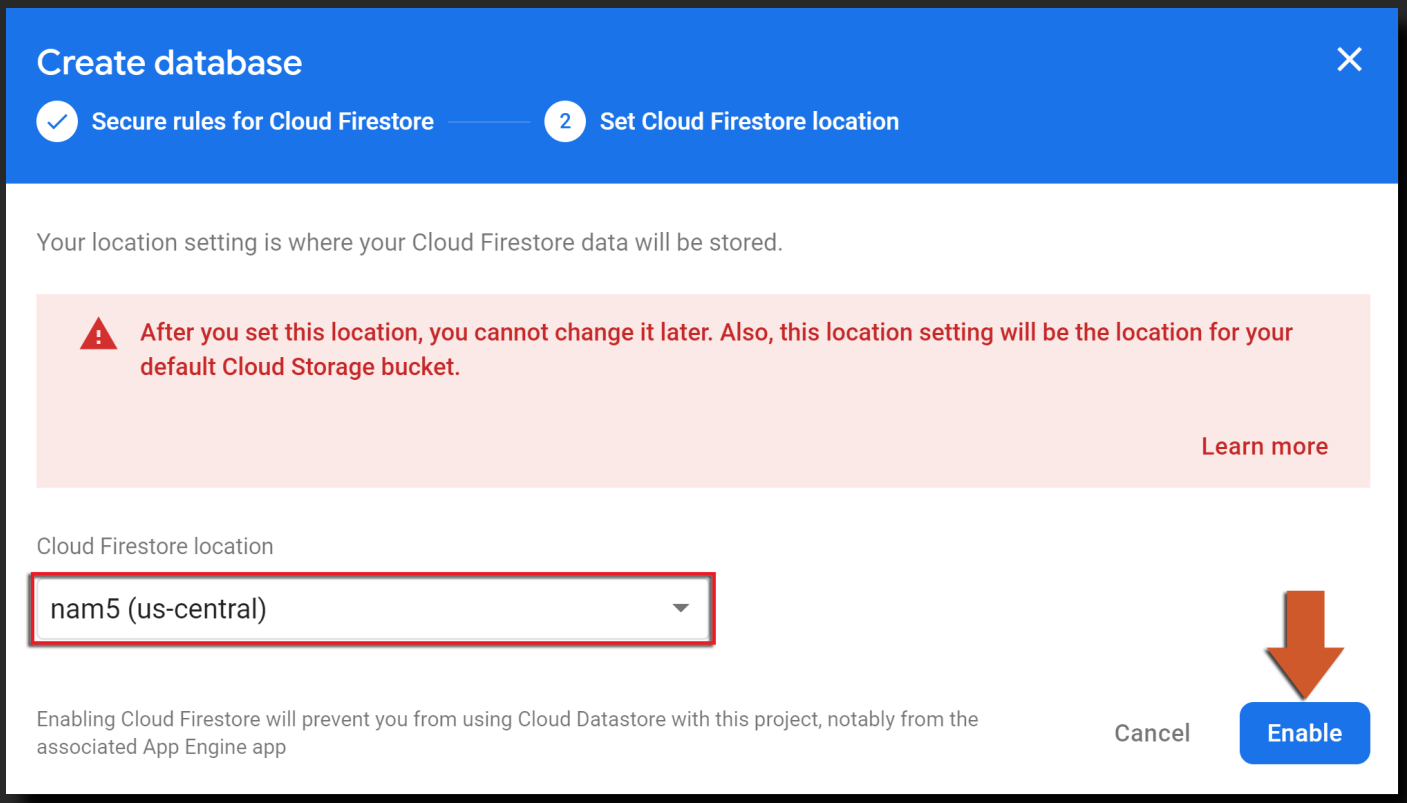
1. On your Firebase project's dashboard, click on **Firestore Database** and then **Create Database:**



2. Select **Start in test mode** (otherwise you will need add your own rules for production mode) and click on **Next**.



3. Select your database server location and click on **Enable**.



4. You are now ready to start using Firebase Firestore extension.

Platform Setup

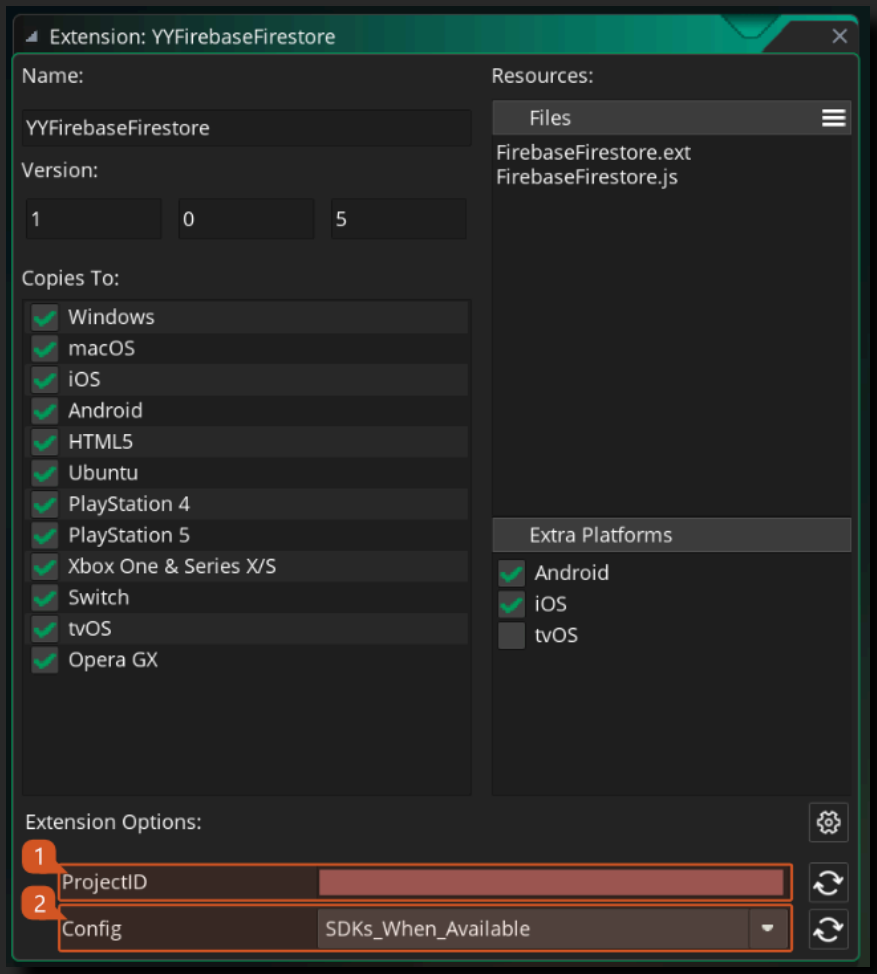
Firebase Firestore implementation uses both SDK (working on **Android**, **iOS** and **Web**) and also REST API that allows it to work on any other platform. In this section we will cover the steps necessary to start using the Firebase Firestore extension in your game.

Select your target platform below and follow the simple steps to get your project up and running:

- **Android Setup** (once per project)
- **iOS Setup** (once per project)
- **Web Setup** (once per project)
- **REST API Setup**

Advanced Configuration

Firebase Firestore by default uses SDKs on **Android**, **iOS** and **Web** targets and uses REST API on all other exports, but you can change this behavior (ie.: forcing REST API to be used even on SDK enabled platforms) by changing the extension options value inside extension window (2).

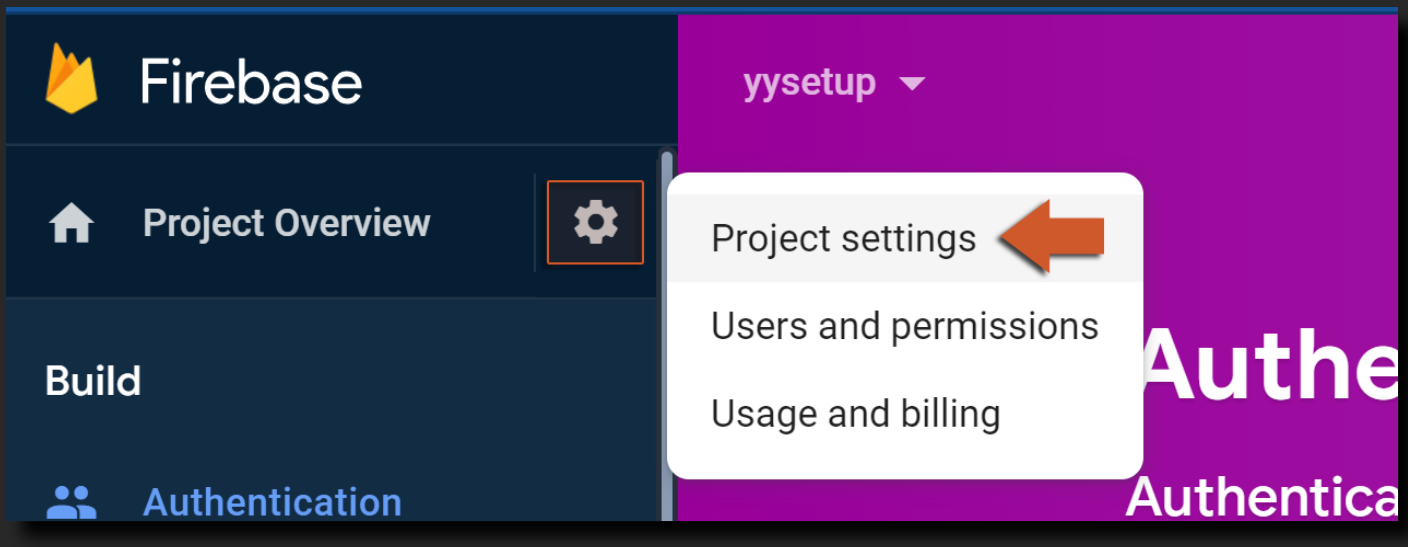

Extension: YYFirebaseFirestore
Name:
YYFirebaseFirestore
Version:
1
0
5
Copies To:
Windows
macOS
iOS
Android
HTML5
Ubuntu
PlayStation 4
PlayStation 5
Xbox One & Series X/S
Switch
tvOS
Opera GX
Resources:
Files
FirebaseFirestore.ext
FirebaseFirestore.js
Extra Platforms
Android
iOS
tvOS
Extension Options:
1
ProjectID
2
Config
SDKs_When_Available

Android Setup

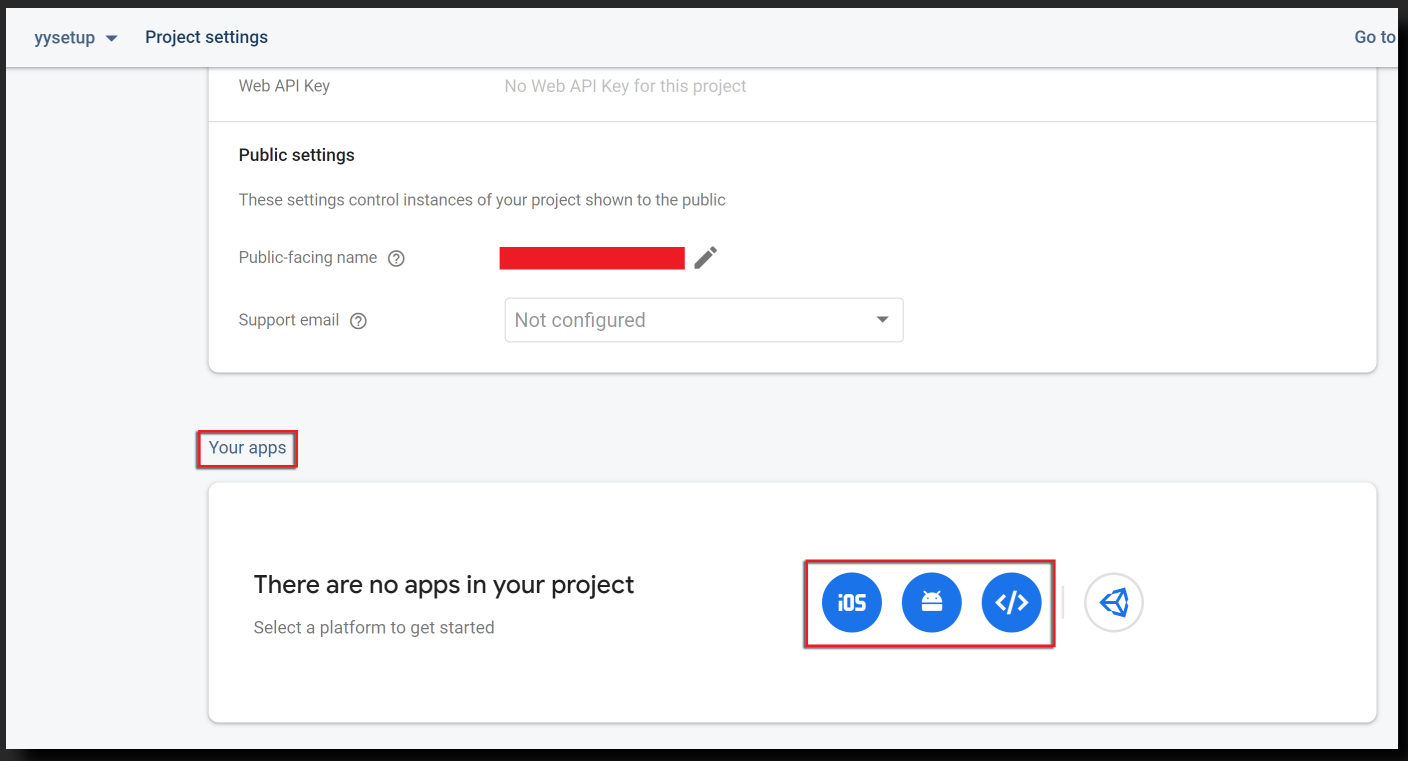
This setup is necessary for all the Firebase modules using Android and needs to be done once per project, and basically involves importing the **google-services.json** file into your project.

IMPORTANT Please refer to **this Helpdesk article** for instructions on setting up an Android project.

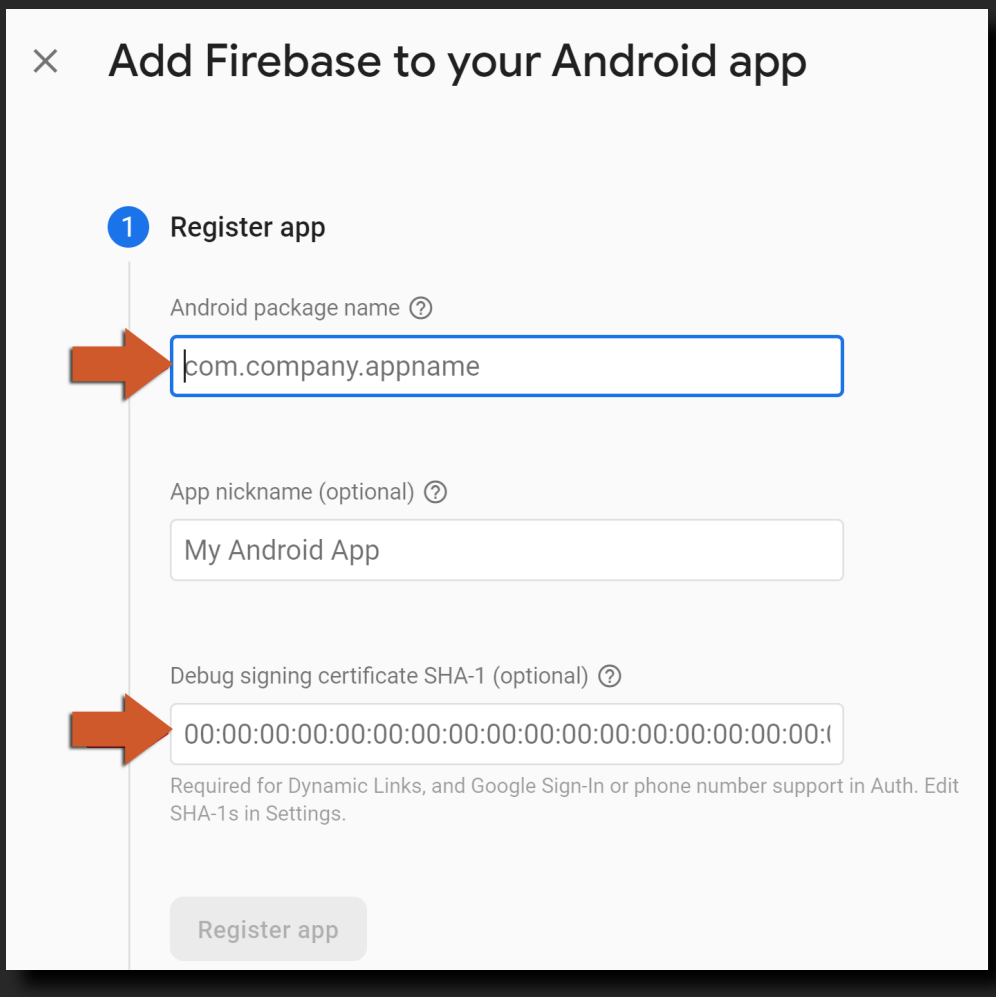
1. Click the **Settings** icon (next to Project Overview) and select **Project settings:**



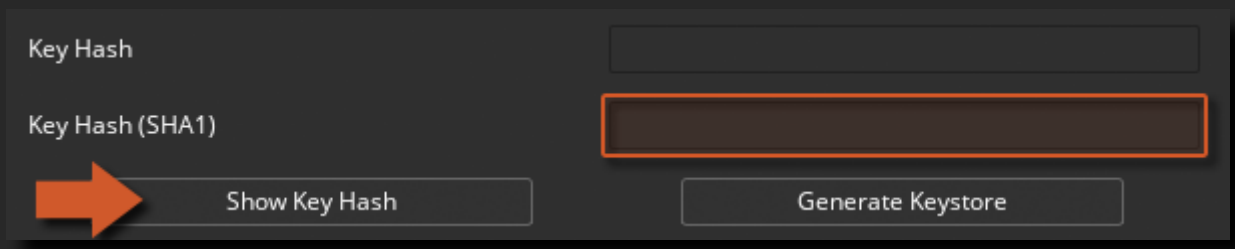
2. Now go to the **Your apps** section and click on the **Android** button:



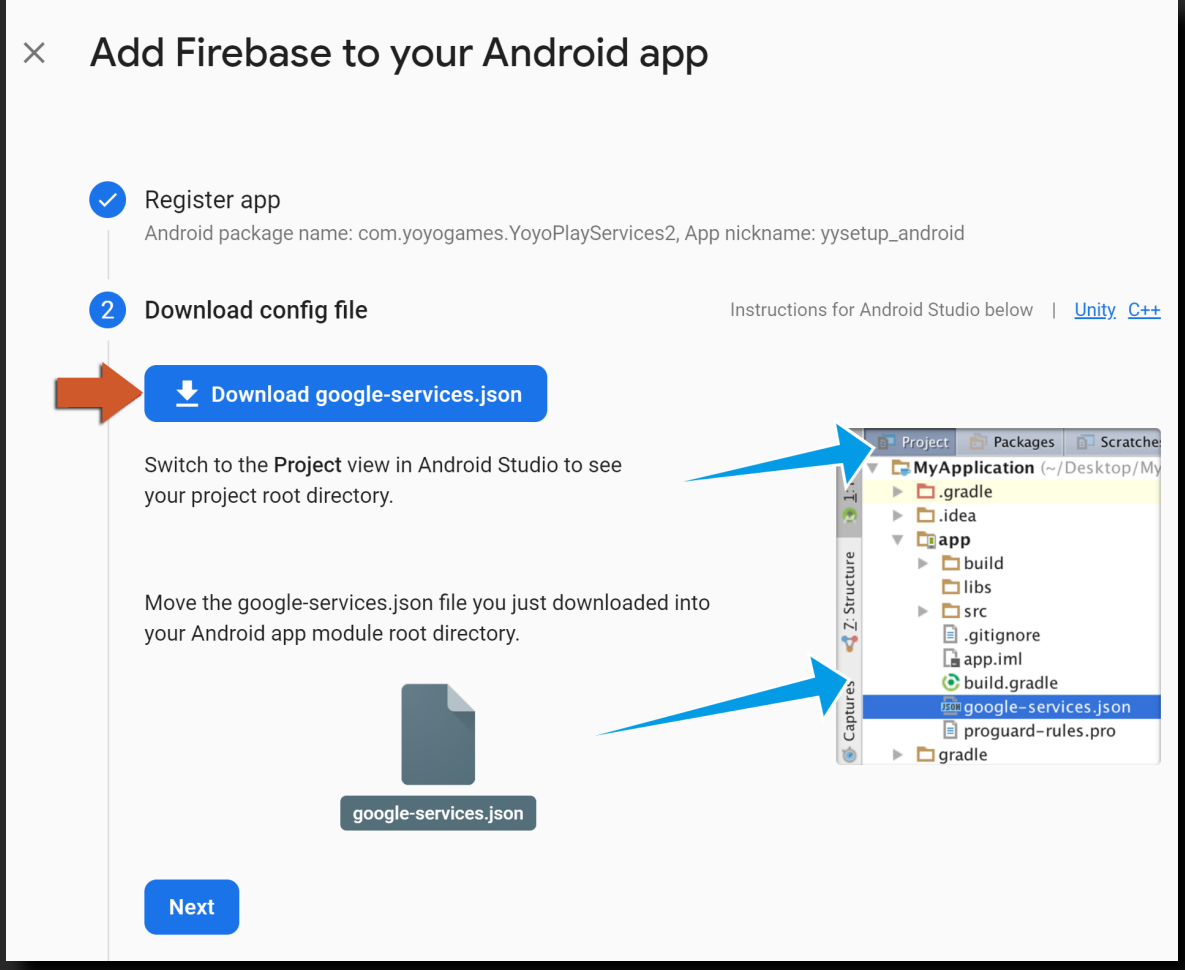
3. On this screen you need enter your **Package name** (required), **App nickname** (optional) and **Debug signing certificate SHA-1** (required if you are using Firebase Authentication).



You can get your package name from the **Android Game Options**, and your **Debug signing certificate SHA-1** from the **Android Preferences** (under Keystore):

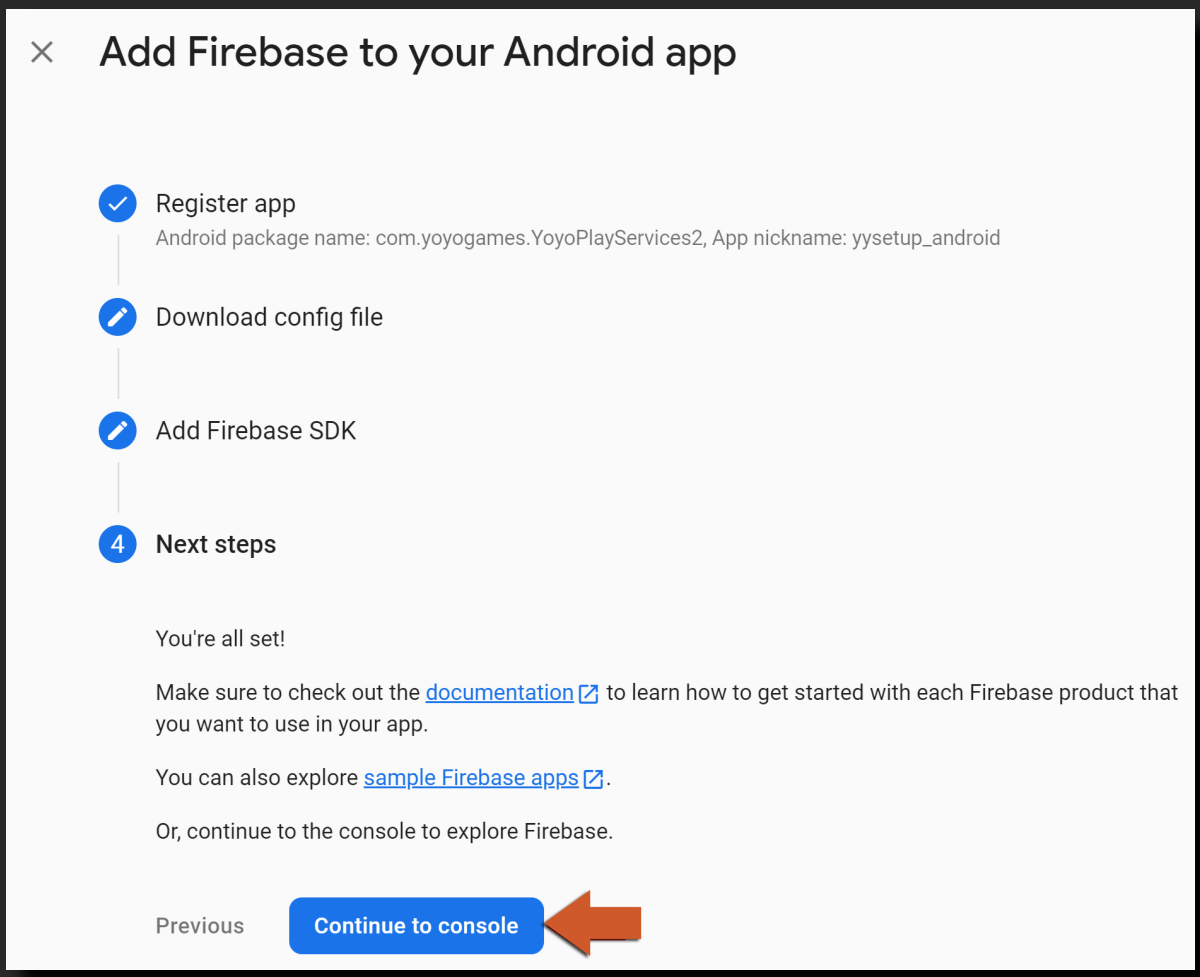


4. Click on **Download google-services.json** (make sure to save this file as we will need it in subsequent steps).

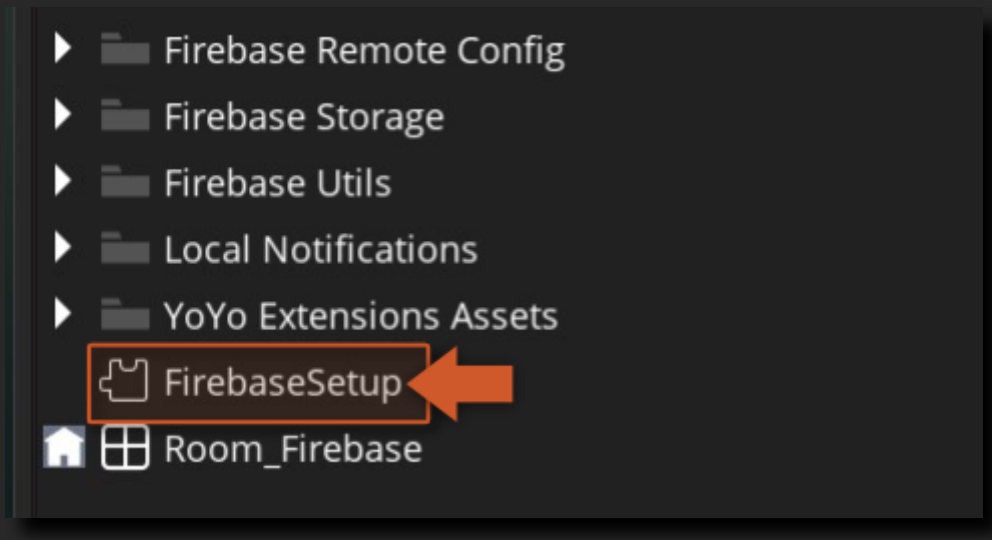


5. Ignore this screen, as this is already done in the extension.

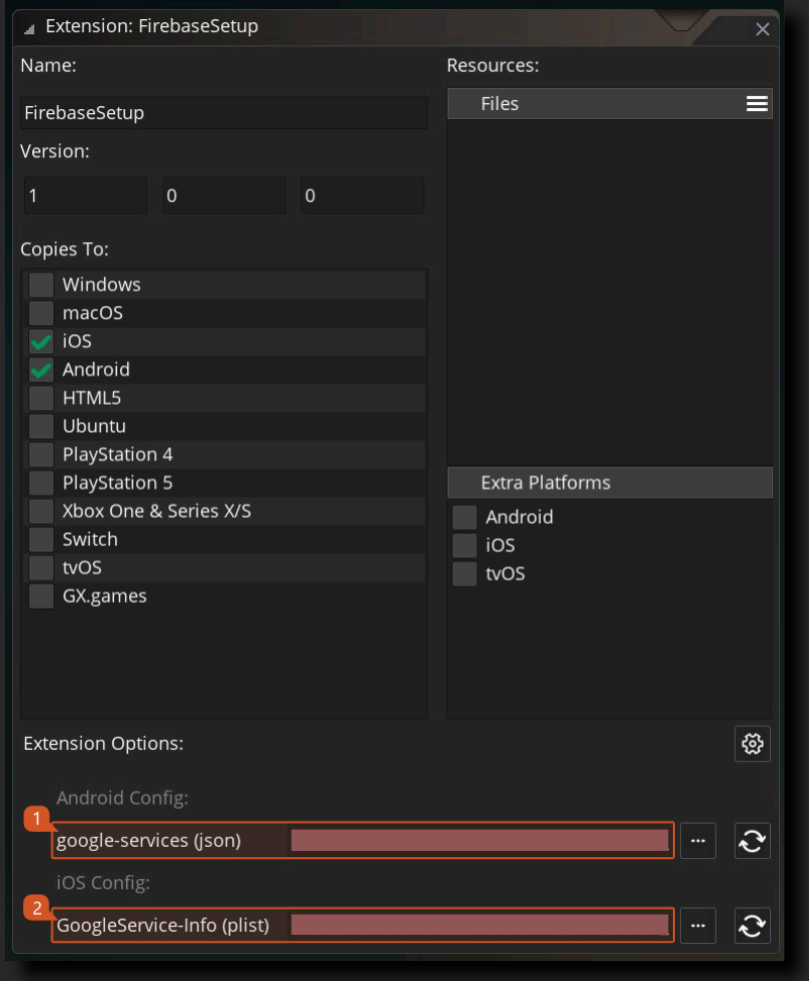
6. Click on the **Continue to console** button.



7. Now go into GameMaker, double click the extension **FirebaseSetup** asset.



8. In the extension panel just fill in the paths for the correct files (Android and/or iOS).



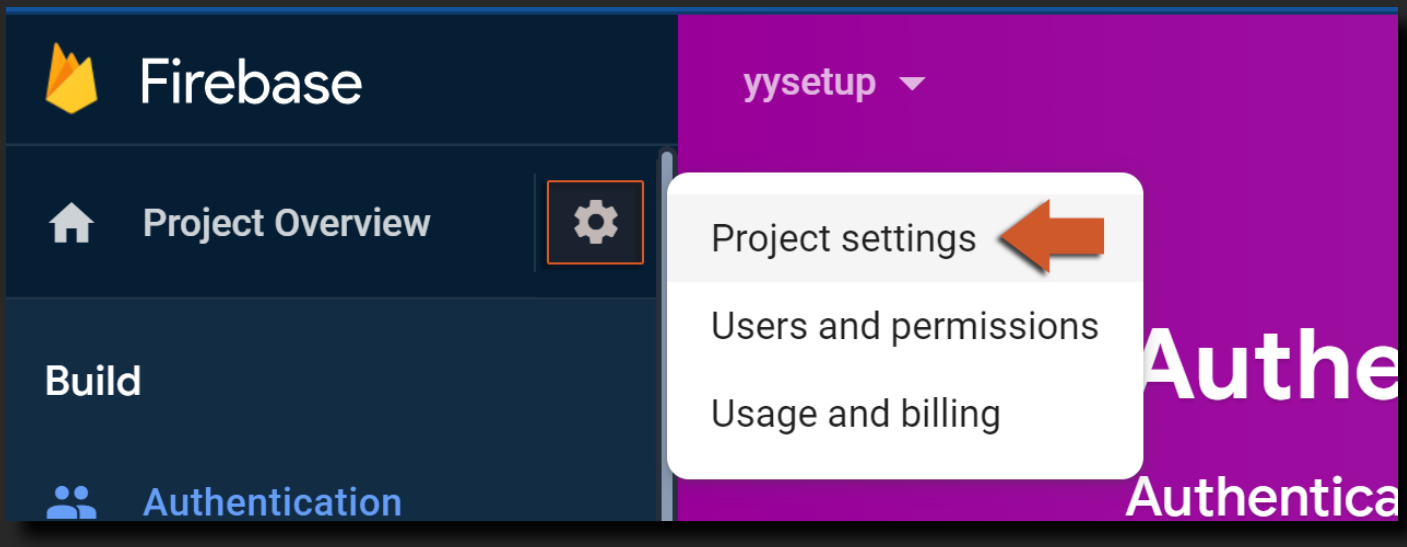
9. You have now finished the main setup for all Firebase Android modules!

iOS Setup

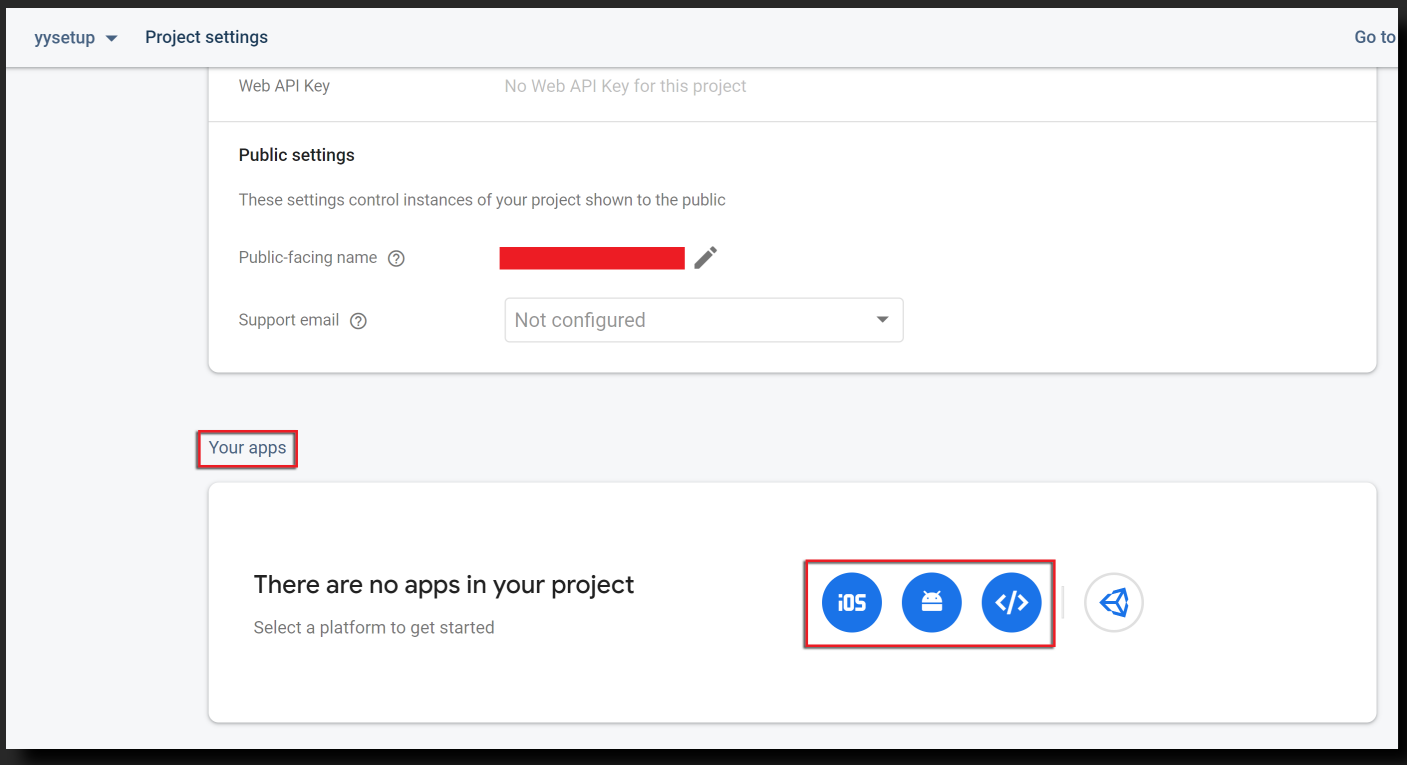
This setup is necessary for all the Firebase modules using iOS and needs to be done once per project, and basically involves importing the **GoogleServices-Info.plist** file into your project.

IMPORTANT Please refer to **this Helpdesk article** for instructions on setting up an iOS project.

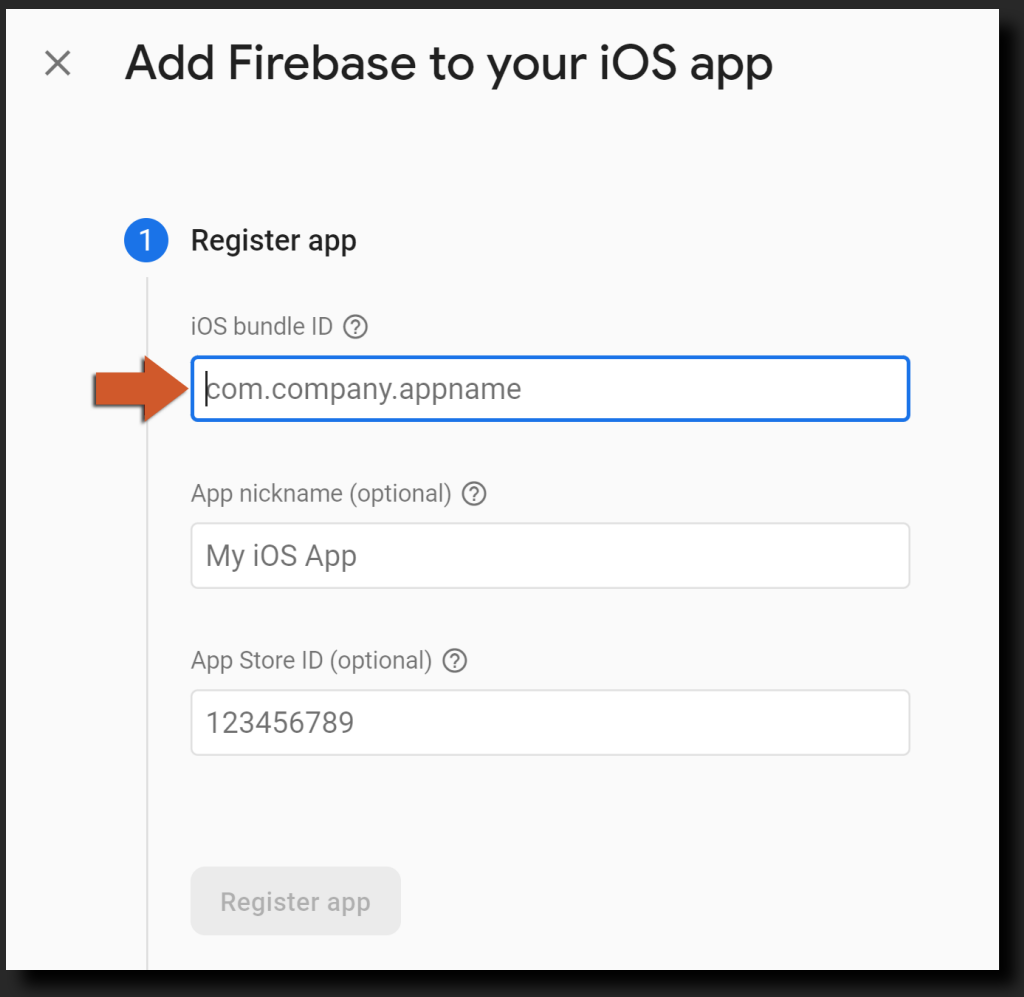
1. Click the **Settings** icon (next to Project Overview) and select **Project settings:**



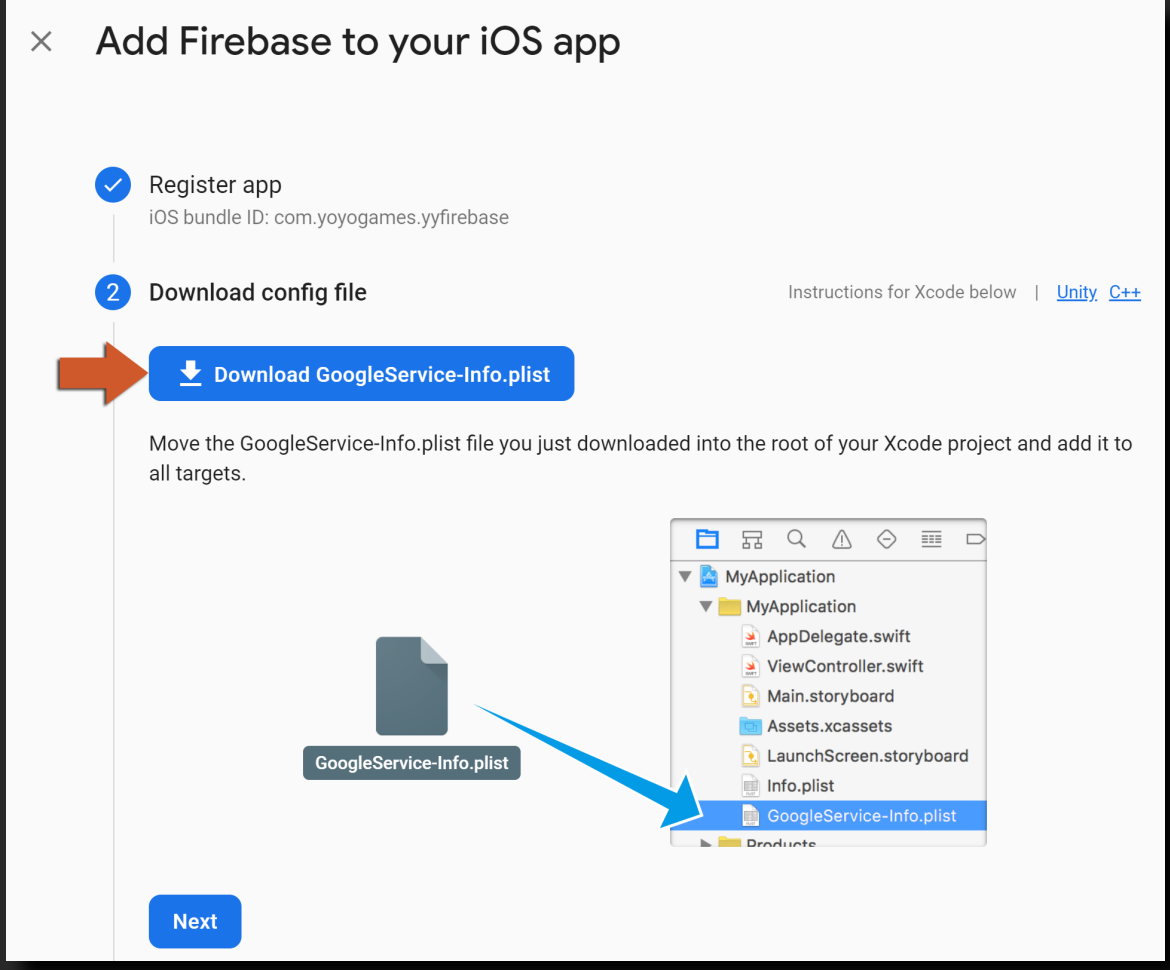
2. Now go to the **Your apps** section and click on the **iOS** button:



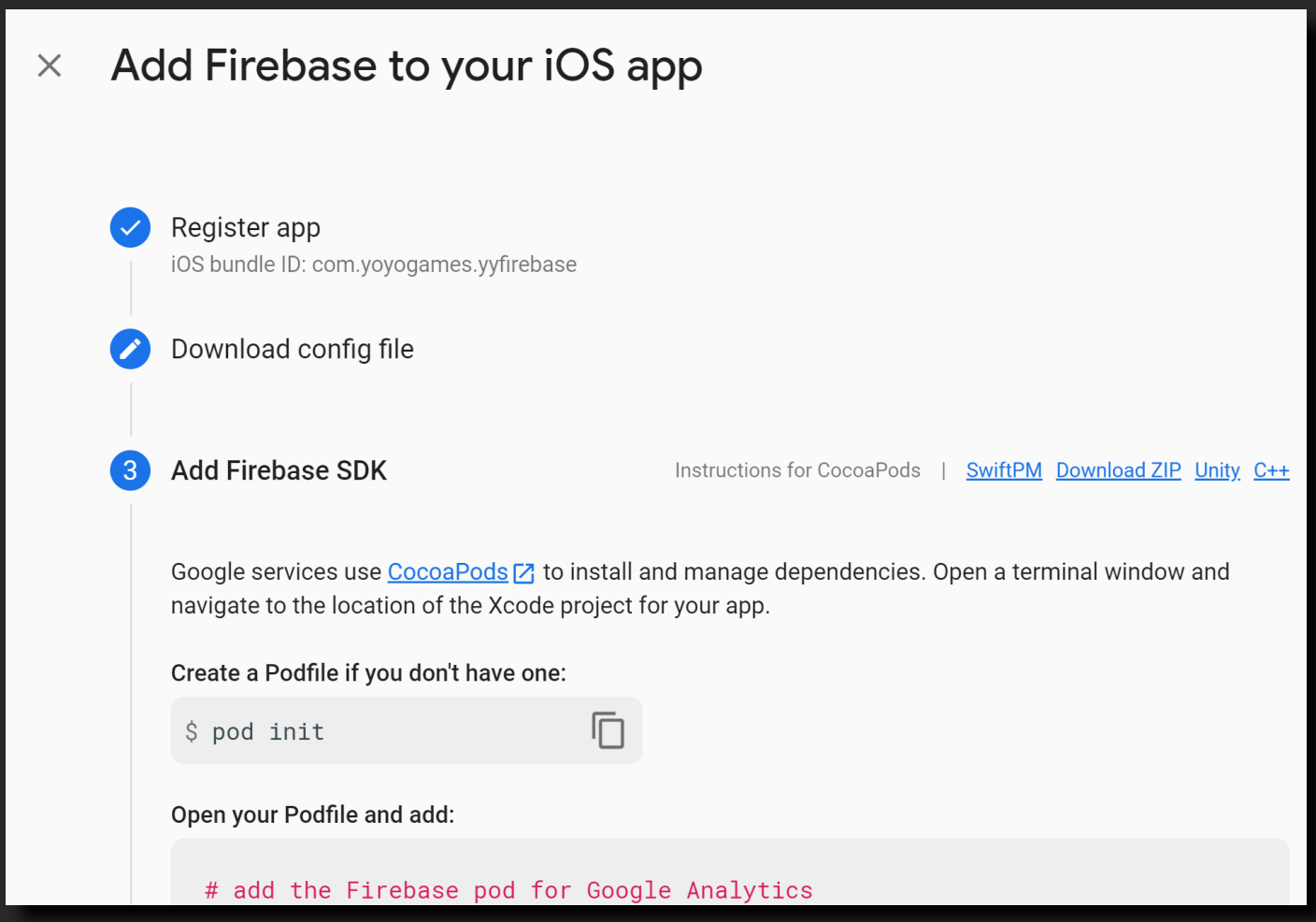
3. Fill the form with your **iOS Bundle ID**, **App nickname** and **AppStore ID** (last two are optional).



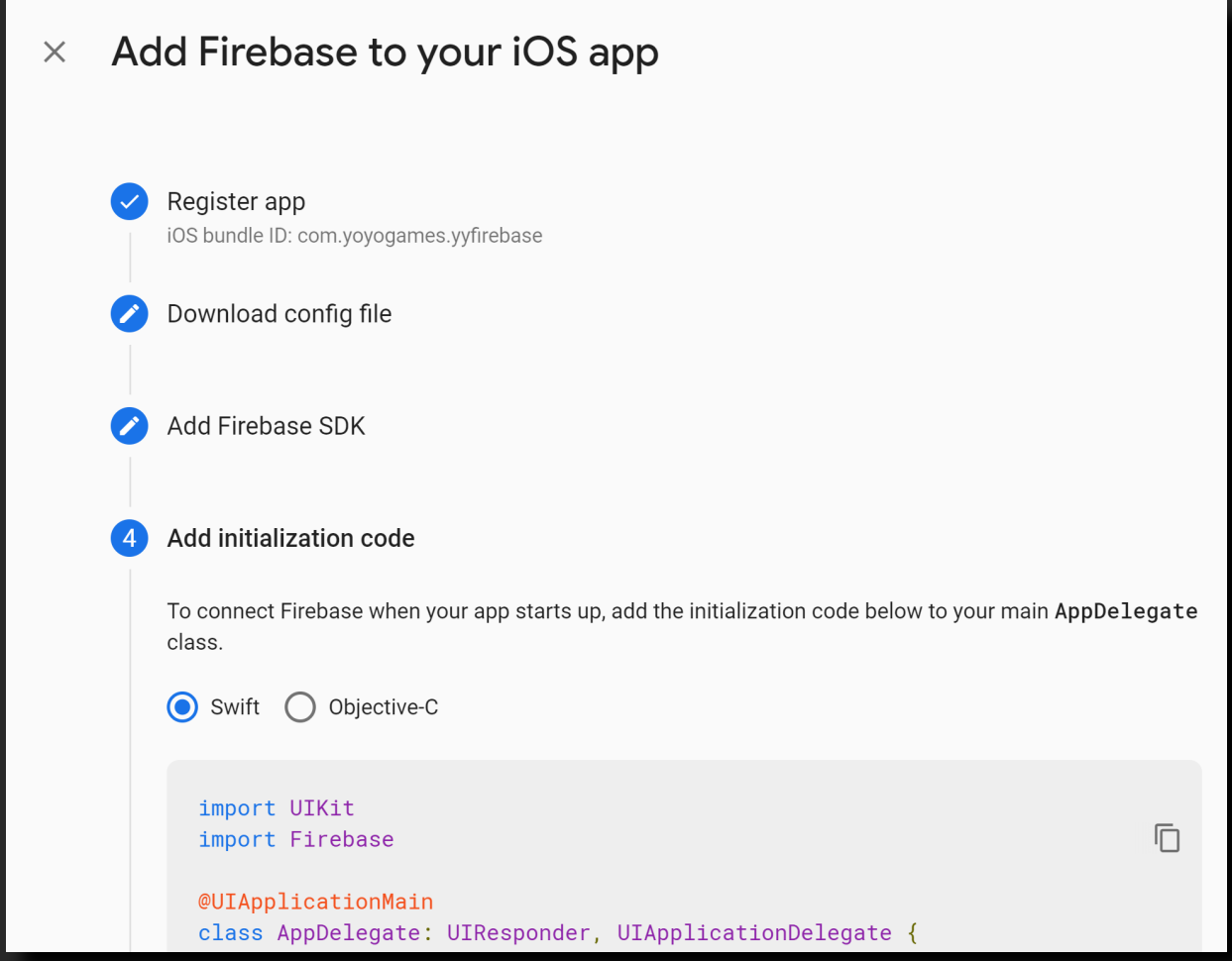
4. Click on **Download GoogleService-info.plist** (make sure to save this file as we will need it in subsequent steps).



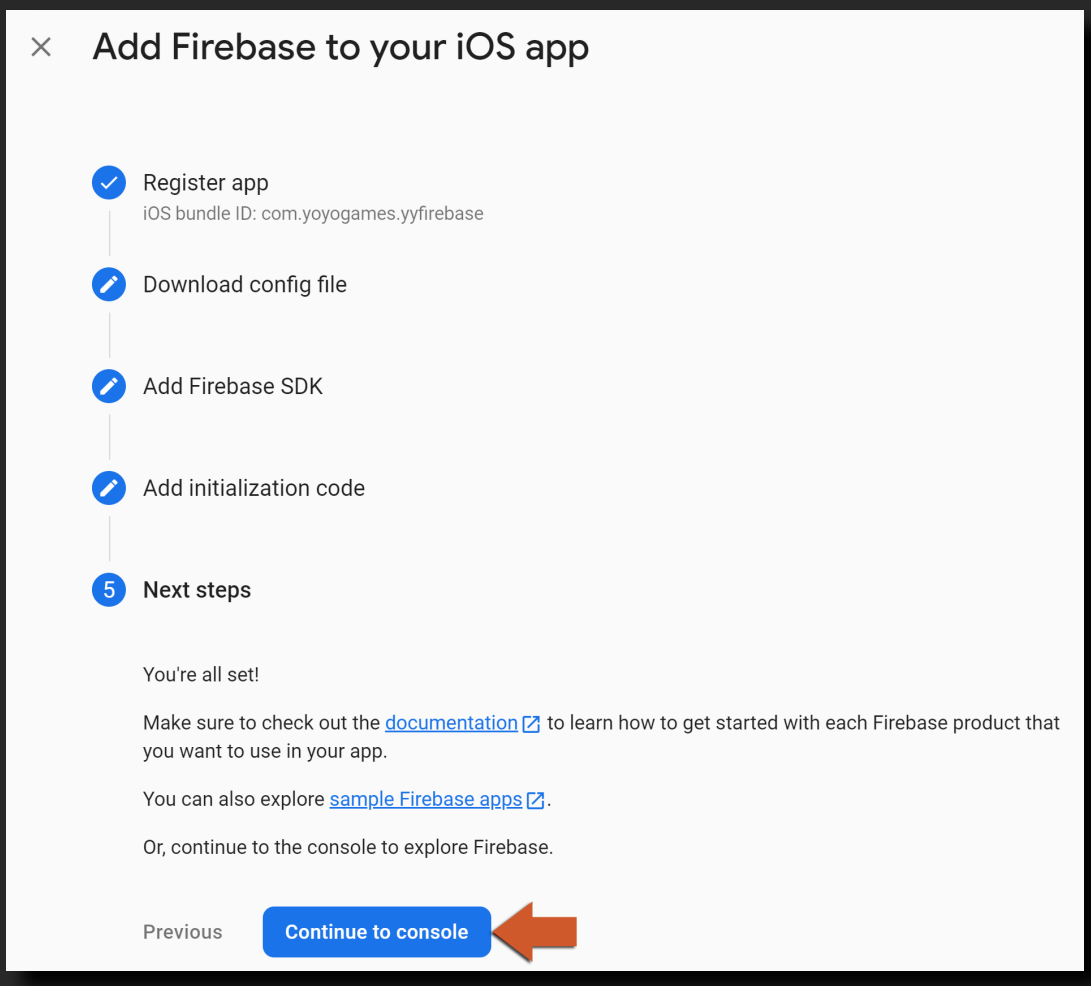
5. Ignore this screen, as this is already done in the extension.



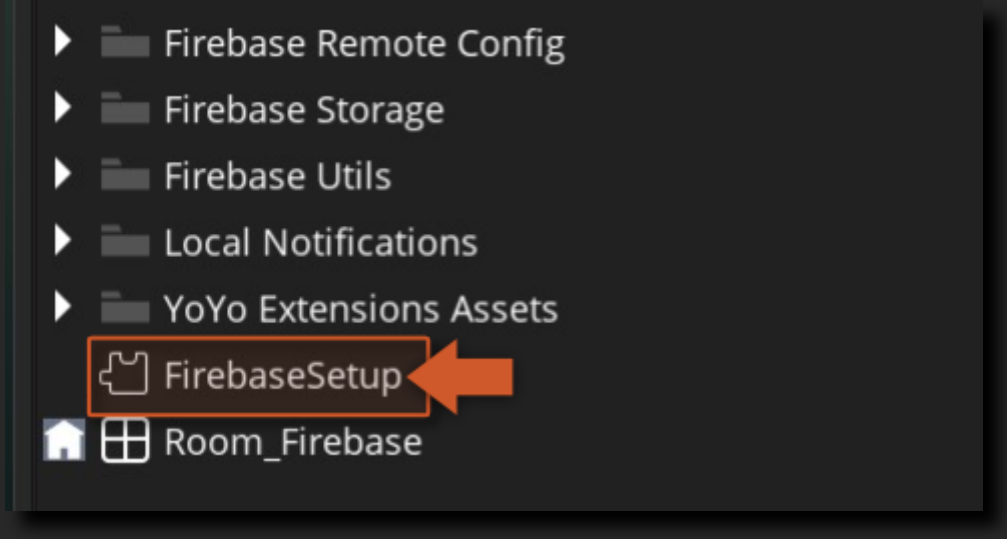
6. Ignore this screen as well, as this is also done in the extension.



7. Click on the **Continue to console** button:



8. Now go into GameMaker, double click the extension **FirebaseSetup** asset.



9. In the extension panel just fill in the paths for the correct files (Android and/or iOS).

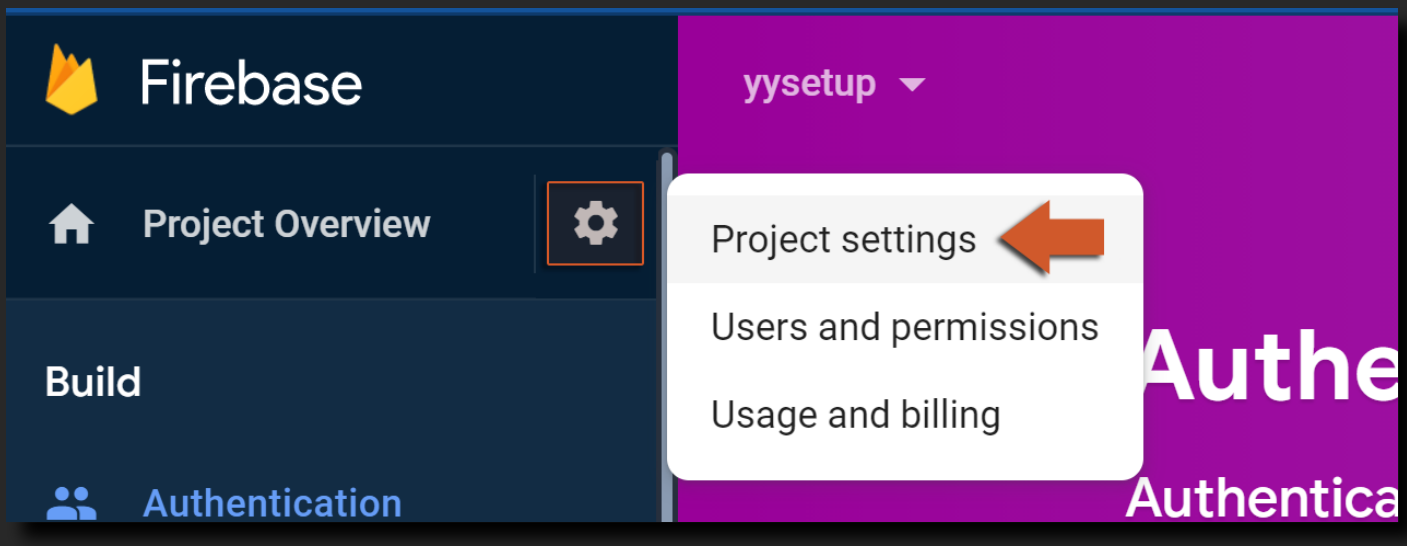
10. Make sure to set up **CocoaPods** for your project *unless* you are only using the REST API in an extension (if one is provided -- not all extensions provide a REST API) or the Firebase Cloud Functions extension (which only uses a REST API).

11. You have now finished the main setup for all Firebase iOS modules!

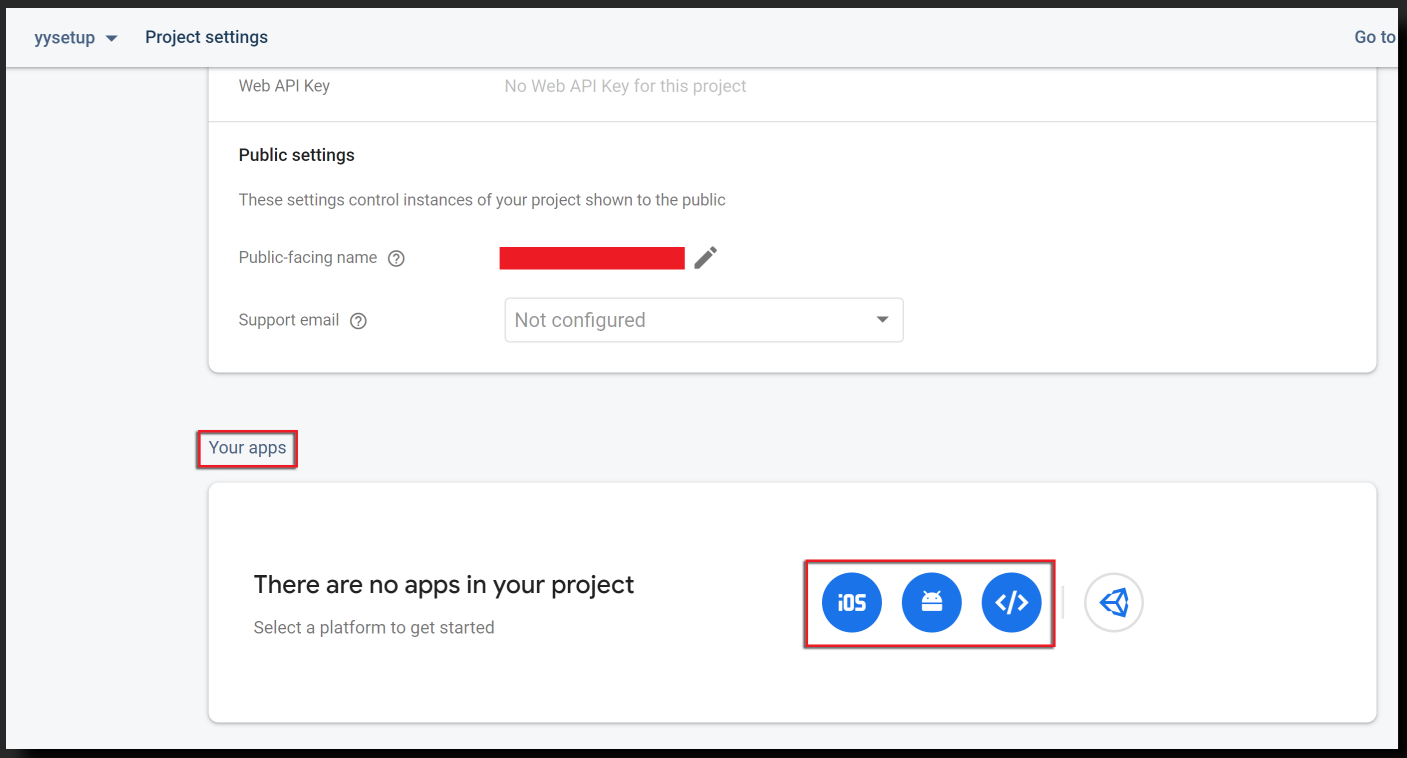
Web Setup

This setup is necessary for all the Firebase modules using Web export and needs to be done once per project, and basically involves adding Firebase libraries and your Firebase values to the **index.html** file in your project.

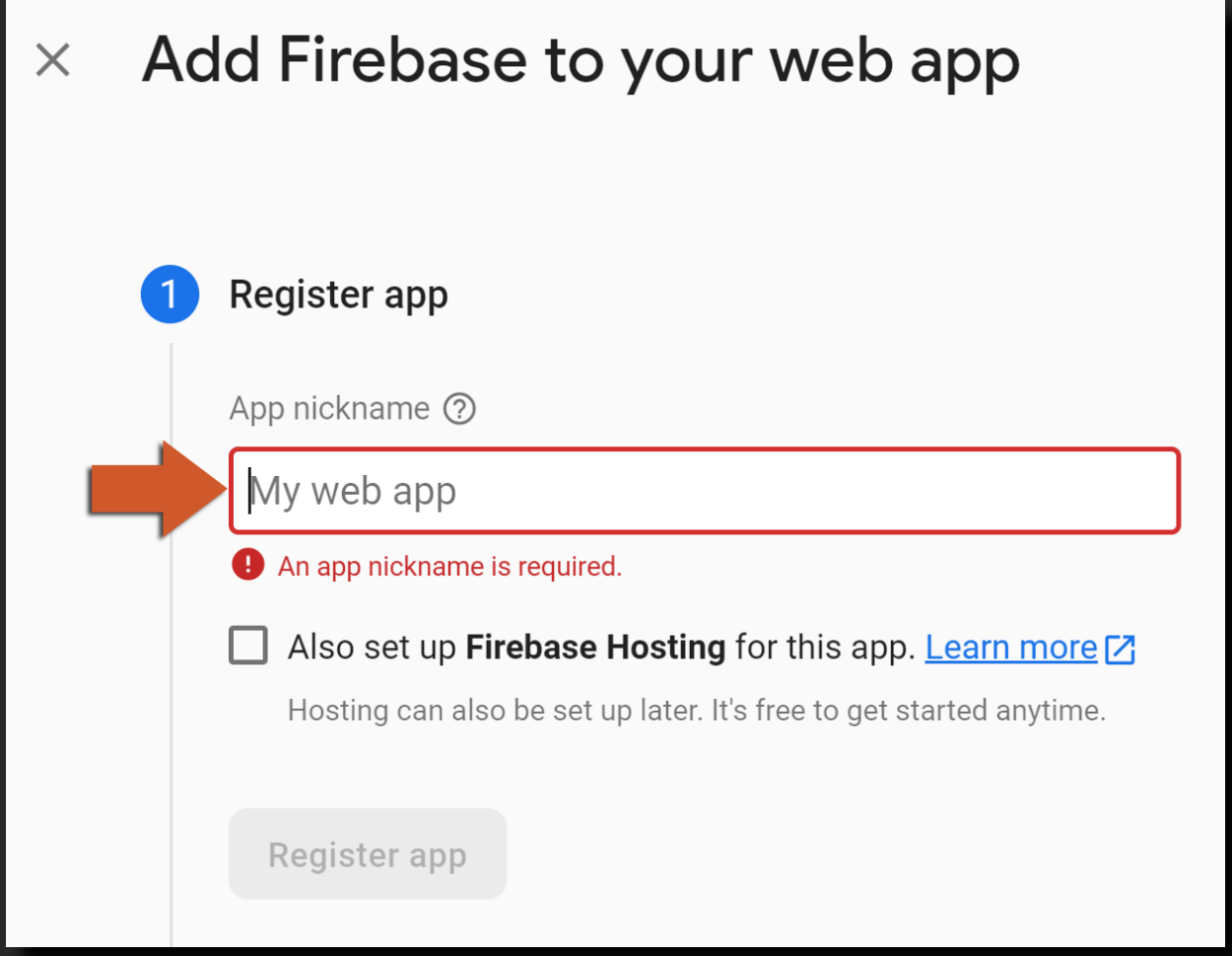
1. Click the **Settings** icon (next to Project Overview) and then **Project settings:**



2. Now go to the **Your apps** section and click on the **Web** (</>) button:



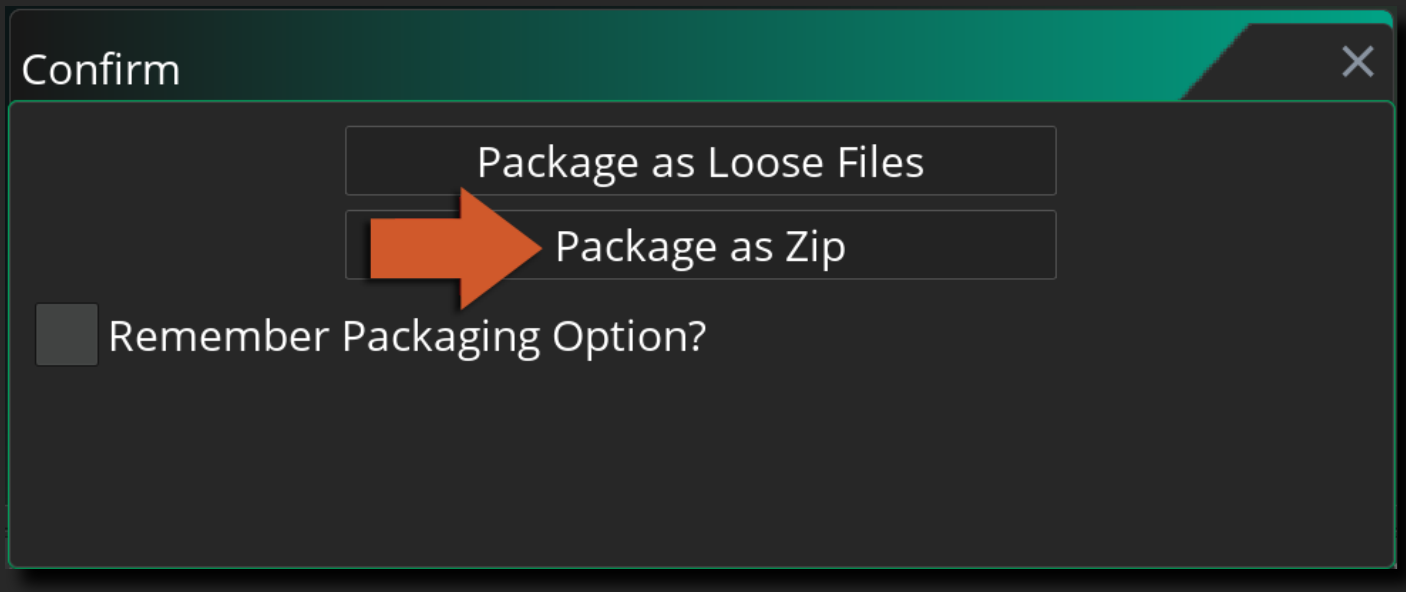
3. Enter your App nickname (required):



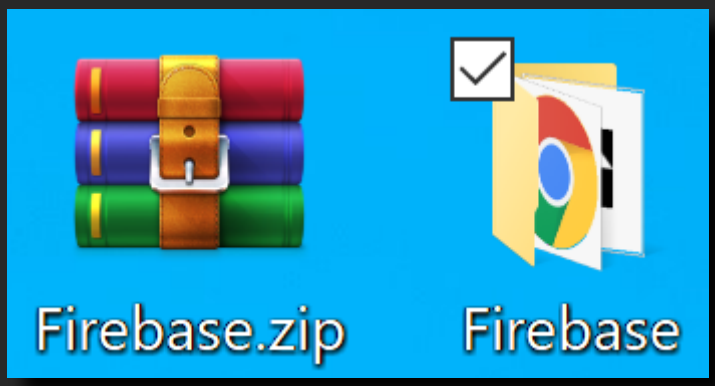
4. On this screen, just copy the **firebaseConfig** struct:

5. Now go back into GameMaker Studio 2 and **build** your project.

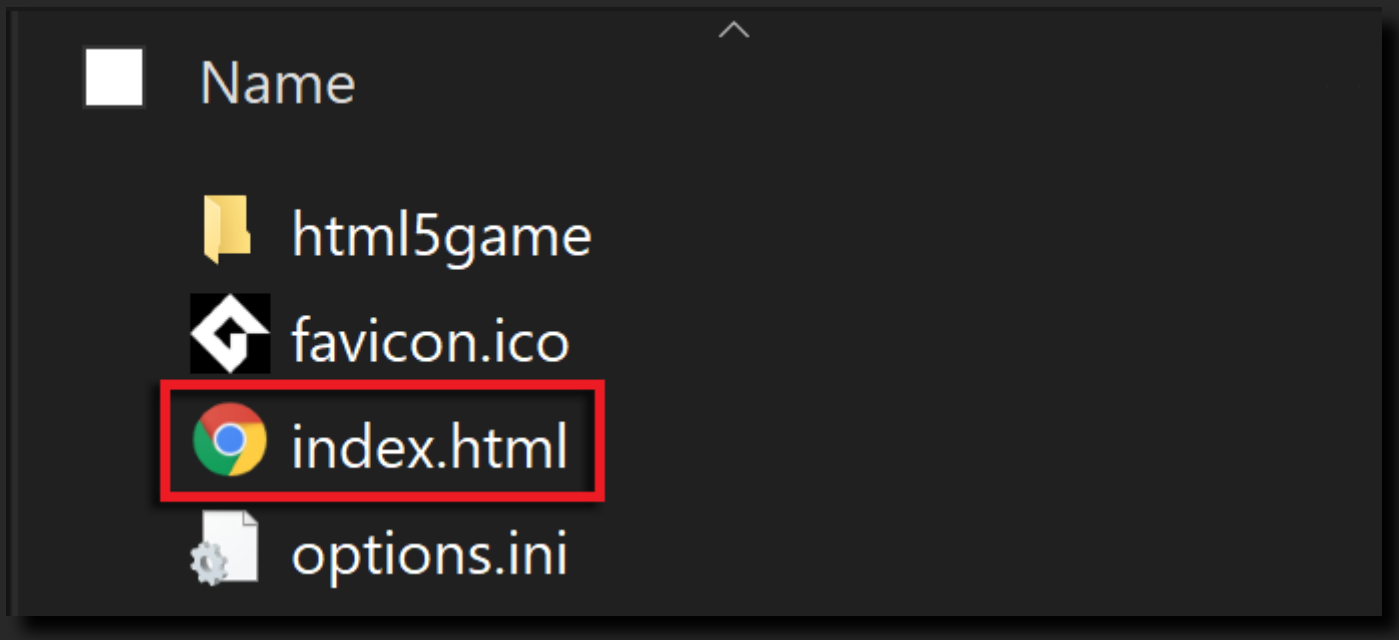
6. Choose the **Package As Zip** option:



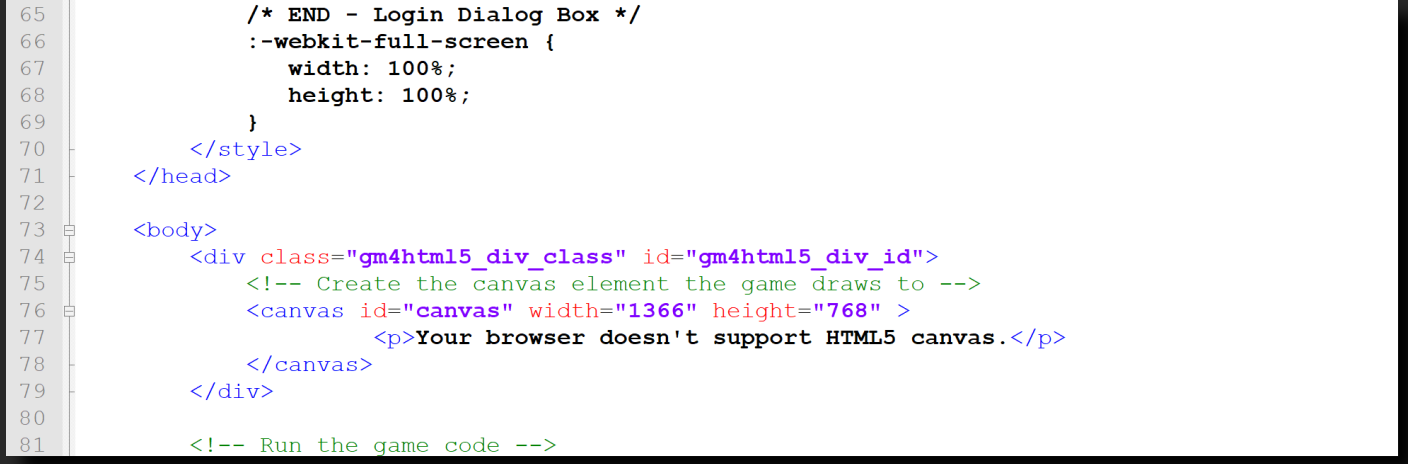
7. Locate the created package and **extract it** into a folder.



8. Open the extracted folder and look for an **index.html** file.



9. Open the **index.html** file in Notepad++ or Visual Studio Code (or any other text editor you prefer).



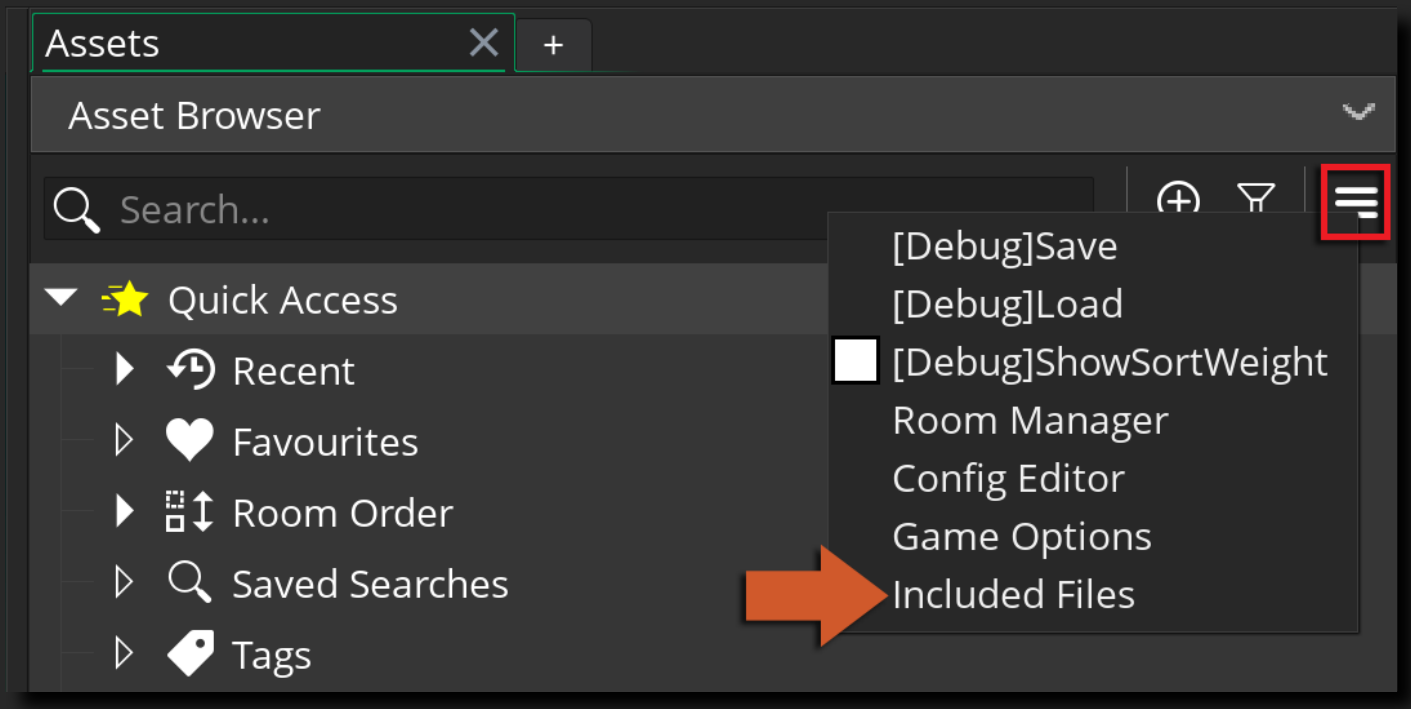
10. Now we need to add the following code between the `</head>` and `<body>` tags (line 72 in the **html.index** image above):

```
<script src="https://www.gstatic.com/firebasejs/8.9.1/firebase-
app.js"></script>
<script src="https://www.gstatic.com/firebasejs/8.9.1/firebase-
analytics.js"></script>
<script src="https://www.gstatic.com/firebasejs/8.9.1/firebase-
auth.js"></script>
<script src="https://www.gstatic.com/firebasejs/8.9.1/firebase-
database.js"></script>
<script src="https://www.gstatic.com/firebasejs/8.9.1/firebase-
firestore.js"></script>
<script src="https://www.gstatic.com/firebasejs/8.9.1/firebase-remote-
config.js"></script>

<script>
  const firebaseConfig = {
    apiKey: "",
    authDomain: "",
    databaseURL: "",
    projectId: "",
    storageBucket: "",
    messagingSenderId: "",
    appId: "",
    measurementId: ""
  };
  firebase.initializeApp(firebaseConfig);

</script>
```

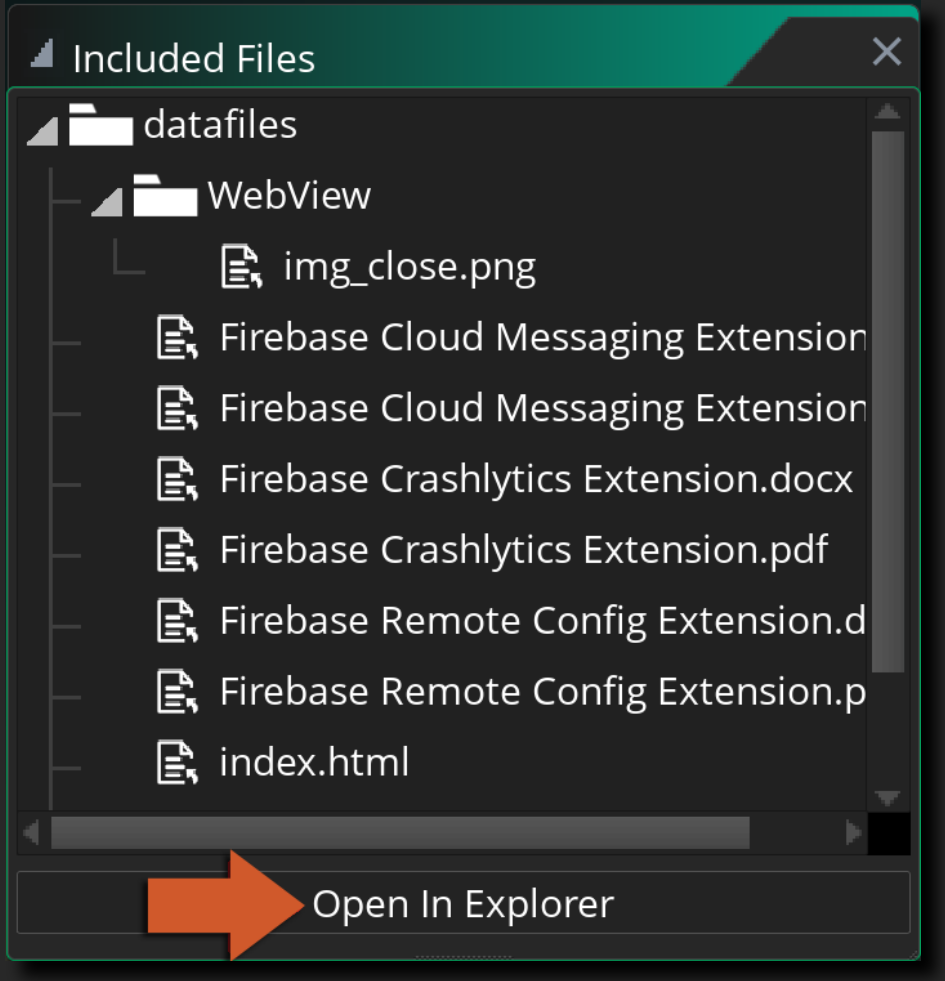
11. Replace the **const firebaseConfig** part with the one copied in step 4:

```
                height: 100%;
            }
        </style>
    </head>

<!-- Firebase -->
<script src="https://www.gstatic.com/firebasejs/8.9.1/firebase-app.js"></script>
<script src="https://www.gstatic.com/firebasejs/8.9.1/firebase-analytics.js"></script>
<script src="https://www.gstatic.com/firebasejs/8.9.1/firebase-auth.js"></script>
<script src="https://www.gstatic.com/firebasejs/8.9.1/firebase-database.js"></script>
<script src="https://www.gstatic.com/firebasejs/8.9.1/firebase-firestore.js"></script>
<script src="https://www.gstatic.com/firebasejs/8.9.1/firebase-remote-config.js"></script>


<script>

  // Your web app's Firebase configuration
  // For Firebase JS SDK v7.20.0 and later, measurementId is optional
  const firebaseConfig = {
    apiKey: "[redacted]",
    authDomain: "[redacted]",
    databaseURL: "[redacted]",
    projectId: "[redacted]",
    storageBucket: "[redacted]",
    messagingSenderId: "[redacted]",
    appId: "[redacted]",
    measurementId: "[redacted]"
  };

  firebase.initializeApp(firebaseConfig);

</script>



    <body>
        <div class="gm4html5_div_class" id="gm4html5_div_id">
```

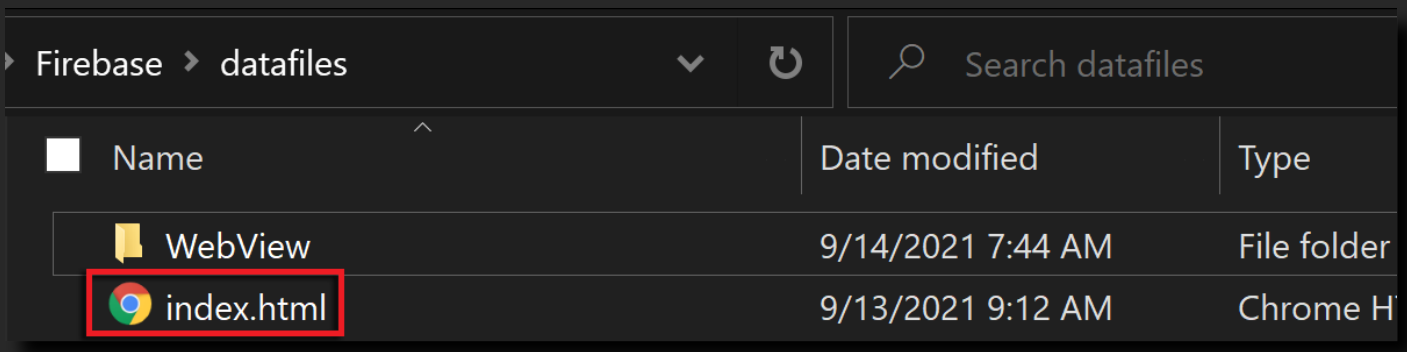
12. Go back into GameMaker and open your **Included Files** folder.



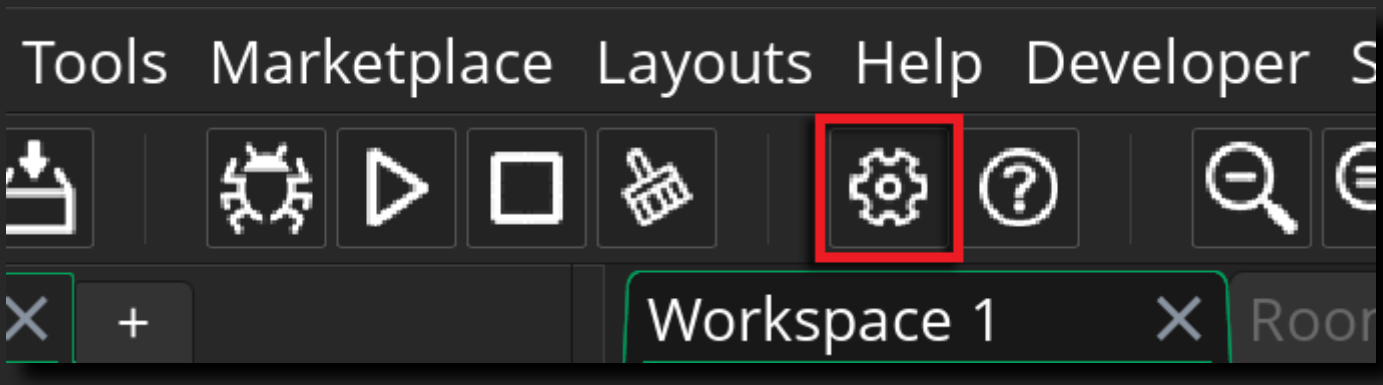
13. Press the **Open in Explorer** button:



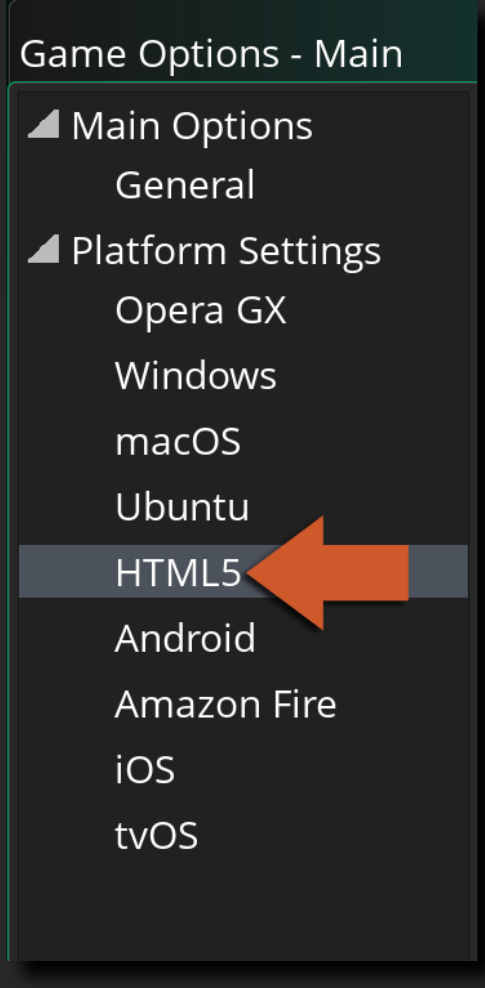
14. Place your **index.html** file inside the folder that opens (/datafiles).



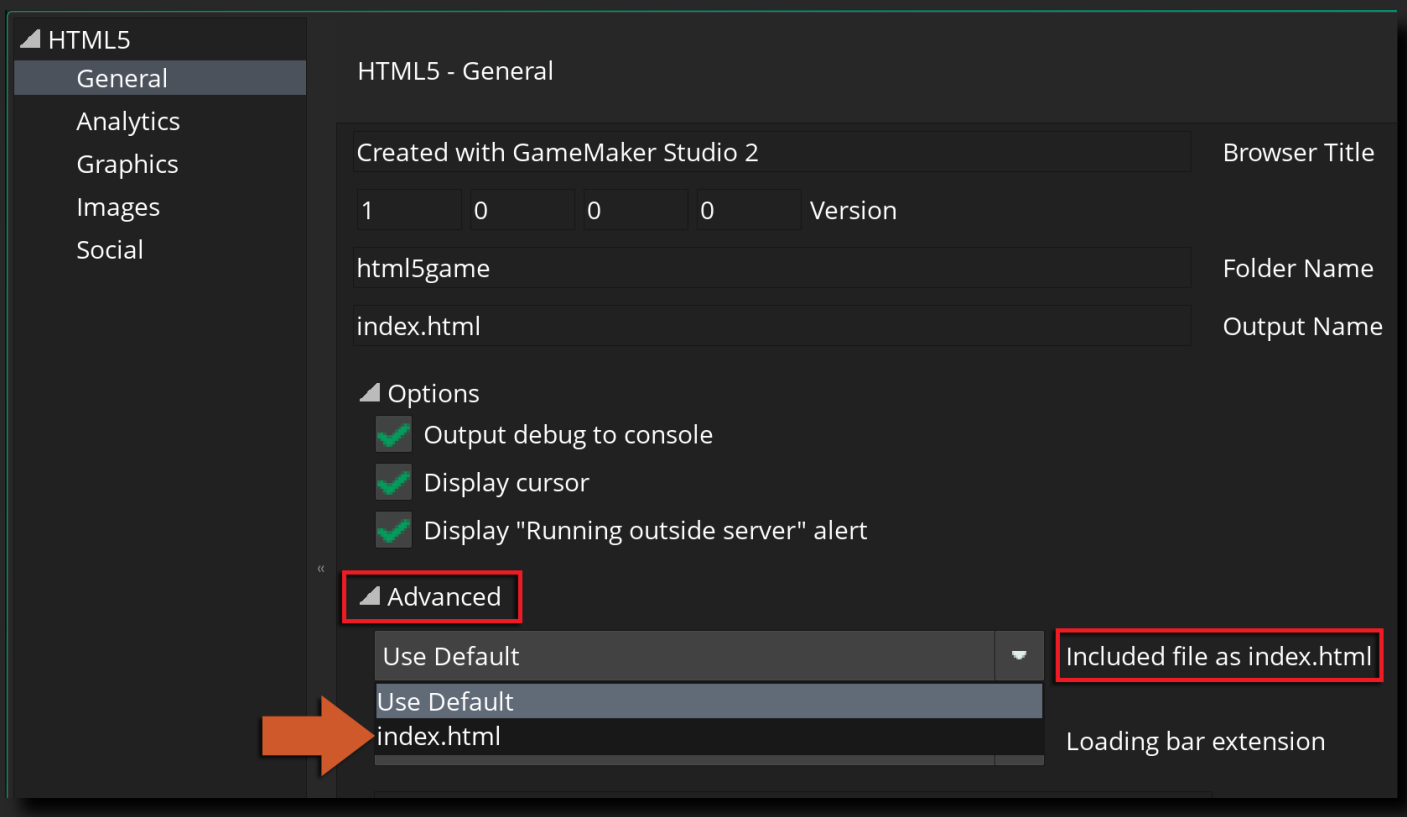
15. Back in GameMaker, click on the **Game Options button.**



16. Go into the **HTML5** platform settings



17. In the **Advanced** section go to the "Include file as index.html" dropdown and select the **index.html** option (this is the file we have just added to the included files).

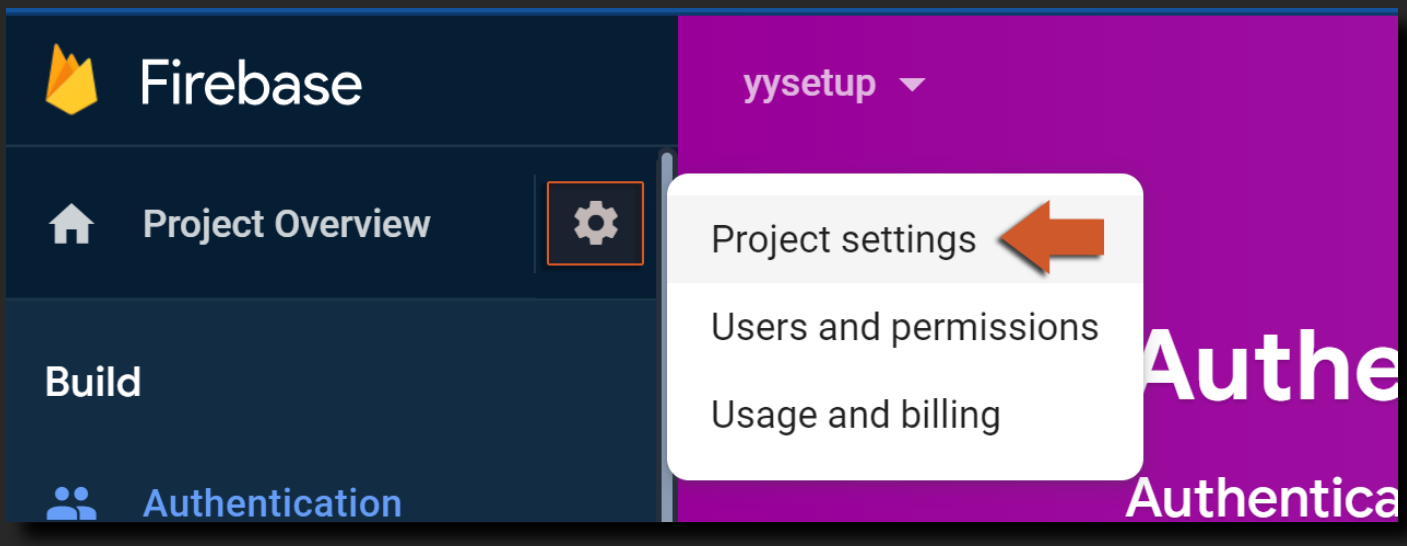


18. Press **Apply** and the main setup for all Firebase Web modules is finished!

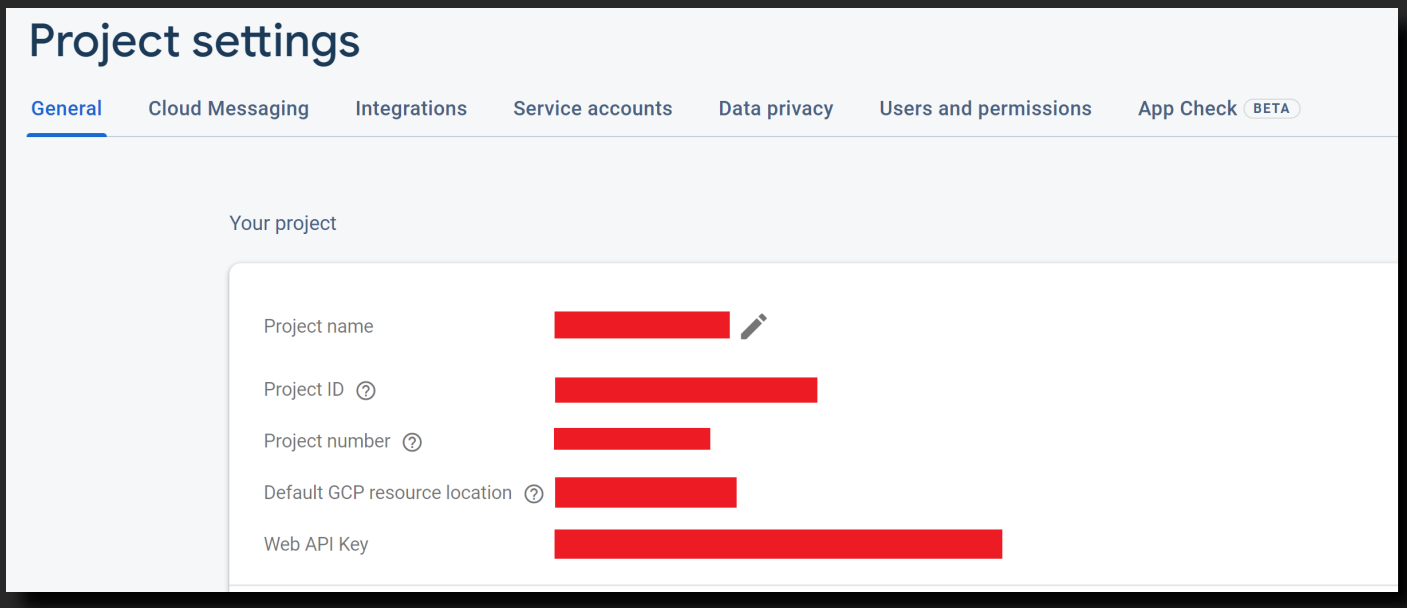
REST API Setup

This setup is necessary for syncing the Firebase console with the REST API implementation of the extension.

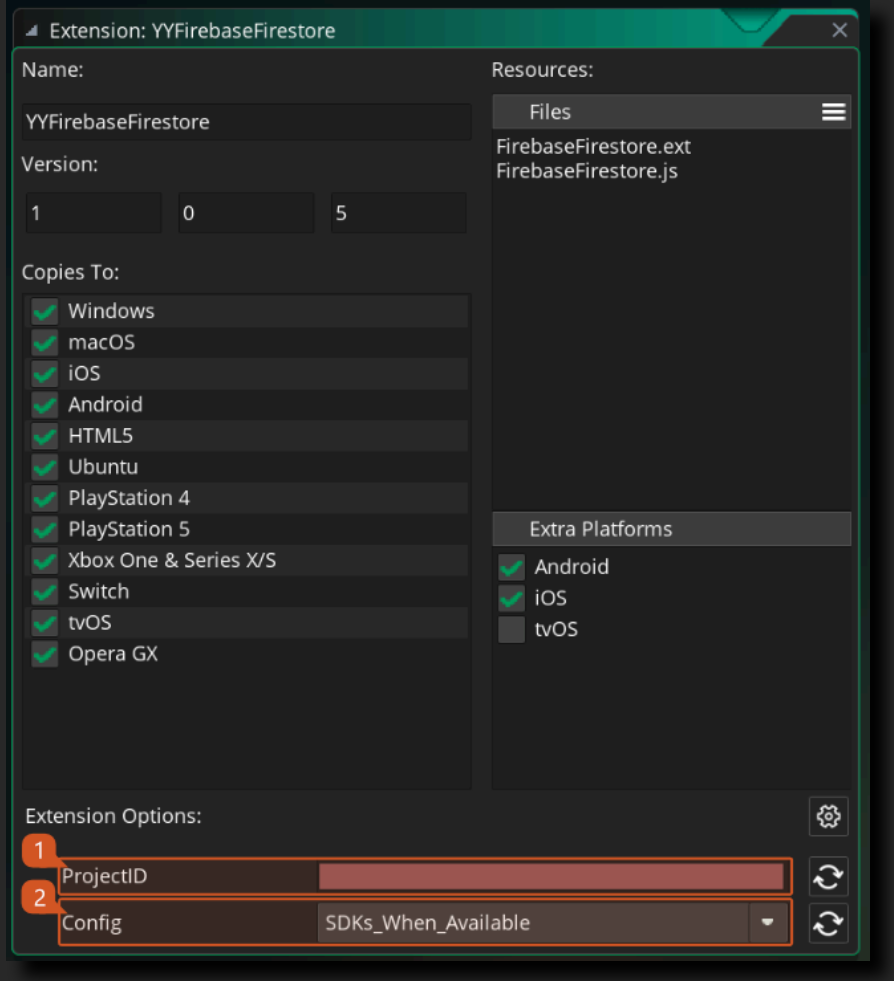
1. On your Firebase console, click on the **Settings** icon (next to **Project Overview**) and then on **Project settings**.



2. Copy the **Project ID**.



3. In your GameMaker Studio project, open the **YYFirebaseFirestore** extension window using the asset browser. And fill in the **Project ID** information (1) also make sure you select the REST API mode (2) to make sure it works on all the available platforms.



4. The extension should now work on all your REST API exports!

FirebaseFirestore

This function is the entry point for database interactions using the Firebase Firestore extension. It returns a *<dbReference>* that can further be modified and configured using chained methods.

NOTE Firebase Firestore implements a fluent-API meaning almost all functions return **self** allowing them to be chained and for you to build complex database interactions while keeping the code clean and easy to follow. Entries in the database are structs and are referred as *<dbReference>* in this documentation (they can be either a *<dbDocument>* or a *<dbCollection>* depending on the path). These, under the hood, are structs that point to a specific path in the database (they are NOT the actual entry).

Syntax:

```
FirebaseFirestore(path)
```

Argument	Type	Description
path	string	The path reference to document or collection OPTIONAL

Returns:

```
dbReference Struct
```

Example:

```
var data = { name: "Hero", level: 99 };
var json = json_stringify(data);

FirebaseFirestore("playersCollection/heroDocument").Set(json);
```

The code sample above will create a data structure (`data`) that will be stored inside the database. This data needs to be converted to string using the json_stringify function. Afterwards it creates a *<dbReference>* to the path `"playersCollection/heroDocument"` and calls the Set method on it (setting the database path to the specified value).

<dbReference>.Child

The method appends the given relative path to the current path of a *<dbReference>* struct.

Syntax:

```
<dbReference>.Child(path)
```

Argument	Type	Description
path	string	The relative child path.

Returns:

```
dbReference Struct
```

Example:

```
var rootPath = "database";
var dbReference = FirebaseFirestore(rootPath).Child("players/hero");
```

The code sample above will create a *<dbReference>* to the `"database"` path and afterwards appends the relative path `"player/hero"` to it, meaning the resulting reference (`dbReference`) now points to the `"database/player/hero"` path.

<dbReference>.Delete

This method deletes the current document reference and returns a listener identifier.

This is an asynchronous function that will trigger the **Async Social** event when the task is finished.

IMPORTANT This function will only work on *dbDocuments*; if you pass in a *dbCollection* it won't be deleted.

Syntax:

```
<dbReference>.Delete()
```

Returns:

```
Real (Asynchronous Listener ID)
```

Triggers:

```
Asynchronous Social Event
```

Key	Type	Description
type	string	The string `"FirebaseFirestore_Document_Delete"`
listener	real	The asynchronous listener ID.
path	bool	The path of the database reference.
status	real	The HTTP status response code (see reference)
errorMessage	string	The readable error message OPTIONAL

Example:

```
listenerId = FirebaseFirestore("myCollection/oldDoc").Delete()
```

In the code above we first get the reference to a database path (`"myCollection/oldDoc"`) and then delete that entry. The function call will then return a listener ID (`listenerId`) that can be used inside an **Async Social** event.

```
if (async_load[? "type"] == "FirebaseFirestore_Document_Delete")
{
    if (async_load[? "status"] == 200)
    {
        show_debug_message("Document was deleted successfully.");
    }
    else
    {
        var errorMessage = async_load[?"errorMessage"]
    }
}
```

The code above matches the response against the correct event **type**, providing a success message if **status** is valid.

<dbReference>.Listener

The method listens for changes to a given document in the database. If you need to stop listening to changes you can use ListenerRemove for a specific database reference or ListenerRemoveAll to remove all listeners.

This is an asynchronous function that will trigger the **Async Social** event each time there is a change to the document.

IMPORTANT When you no longer need to listen to changes make sure to remove the listeners; not doing so can lead to memory leaks.

Syntax:

```
<dbReference>.Listener()
```

Returns:

```
Real (Asynchronous Listener ID)
```

Triggers:

```
Asynchronous Social Event
```

Key	Type	Description
type	string	The string `"FirebaseFirestore_Document_Listener"` or `"FirebaseFirestore_Collection_Listener"`
listener	real	The asynchronous listener ID.
path	string	The path of the database reference.
status	real	The HTTP status response code (see reference)
value	string	A JSON formatted string of either a **struct** (document) or **array** (collection)
errorMessage	string	The readable error message OPTIONAL

Example:

```
listenerId = FirebaseFirestore("myCollection/newDoc").Listener();
```

The code above starts listening for updates on the document (`"myCollection/newDoc"`). If this reference gets updated, set or deleted an Async event will be triggered. The function call will return a listener ID (`listenerId`) that can be used inside an **Async Social** event.

```
if (async_load[? "type"] == "FirebaseFirestore_Document_Listener")
{
    if (async_load[? "status"] == 200)
    {
        show_debug_message("Changes were made to this document");
    }
    else
    {
        var errorMessage = async_load[? "errorMessage"]
    }
}
```

The code above matches the response against the correct event **type** and logs when changes are made to the database reference being listened to.

<dbReference>.ListenerRemove

This function removes a previously created listener (that was created using the function Listener).

TIP If you wish to remove all created listeners use the function ListenerRemoveAll instead.

Syntax:

```
.ListenerRemove(listenerId)
```

Argument	Type	Description
listenerId	real	The listener ID to be removed.

Returns:

```
N/A
```

Example:

```
listenerId = FirebaseFirestore("myCollection/myDoc").Listener()

// Some time later

FirebaseFirestore().ListenerRemove(listenerId)
```

The code sample above starts by creating a listener to a database reference path (using the Listener function) which returns a listener ID (`listenerId`); at a later stage it removes that listener.

<dbReference>.ListenerRemoveAll

This function removes all previously created listeners (that were created using the function Listener).

> TIP If you wish to remove a specific listener use the function ListenerRemove instead.

Syntax:

```
<dbReference>.ListenerRemoveAll()
```

Returns:

```
N/A
```

Example:

```
listenerId1 = FirebaseFirestore("myCollection/myDoc").Listener()
listenerId2 = FirebaseFirestore("myCollection/otherDoc").Listener()
listenerId3 = FirebaseFirestore("myCollection").Listener()

// Some time later
FirebaseFirestore().ListenerRemoveAll()
```

The code above creates three listeners (using the Listener function) and after some time it removes them all.

<dbReference>.Parent

This method goes up the hierarchy to the parent path of the current *<dbReference>* struct.

Syntax:

```
<dbReference>.Parent()
```

Returns:

```
dbReference Struct
```

Example:

```
var dbPath = "database/player/hero";
var dbReference = FirebaseFirestore(dbPath).Parent();
```

The code above creates a *<dbReference>* to the `"database/player/hero"` path and afterwards goes up to the parent path meaning the resulting reference (`dbReference`) now points to the `"database/player"` path.

<dbReference>.Read

This method reads the current database reference and returns a listener identifier.

This is an asynchronous function that will trigger the **Async Social** event when the task is finished.

Syntax:

```
<dbReference>.Read()
```

Returns:

```
Real (Asynchronous Listener ID)
```

Triggers:

```
Asynchronous Social Event
```

Key	Type	Description
type	string	The string `"FirebaseFirestore_Document_Read"` or `"FirebaseFirestore_Collection_Read"`
listener	real	The asynchronous listener ID.
path	string	The path of the database reference.
status	real	The HTTP status response code (see reference)
value	string	A JSON formatted string of either a **struct** (document) or **array** (collection)
errorMessage	string	The readable error message OPTIONAL

Example:

```
listenerId = FirebaseFirestore("myCollection/myDoc").Read()
```

The code above first gets the reference to a database document (`"myCollection/oldDoc"`) and then reads that entry. The function call will then return a listener ID (`listenerId`) that can be used inside an **Async Social** event.

```
if (async_load[? "type"] == "FirebaseFirestore_Document_Read")
{
    if (async_load[? "status"] == 200)
    {
        show_debug_message("Document data is: " + async_load[? "value"]);
    }
    else
    {
        var errorMessage = async_load[?"errorMessage"]
    }
}
```

The code above matches the response against the correct event **type**, providing the document **data** if the task succeeded.

<dbReference>.Set

This method creates or overwrites a document on the database. If *<dbReference>* is a collections then a document will be automatically created with an auto-generated name. If you wish to update an existing document instead of replacing it use **Update** instead. This function returns a listener identifier.

This is an asynchronous function that will trigger the **Async Social** event when the task is finished.

Syntax:

```
<dbReference>.Set(json)
```

Argument	Type	Description
json	string	A JSON formatted string to fill the document with.

Returns:

```
Real (Asynchronous Listener ID)
```

Triggers:

```
Asynchronous Social Event
```

Key	Type	Description
type	string	The string `"FirebaseFirestore_Document_Set"` or `"FirebaseFirestore_Collection_Add"`
listener	real	The asynchronous listener ID.
path	string	The path of the database reference.
status	real	The HTTP status response code (see **reference**)
errorMessage	string	The readable error message OPTIONAL

Example:

```
var json = json_stringify({ name: "Hero", level: 100 });
listenerId = FirebaseFirestore("myCollection/newDoc").Set(json);
```

The code above creates a JSON formatted string of a struct (using the **json_stringify** function) and then sets the database document (`"myCollection/newDoc"`) to the new string. The function call will then return a listener ID (`listenerId`) that can be used inside an **Async Social** event.

```
if (async_load[? "type"] == "FirebaseFirestore_Document_Set")
{
    if (async_load[? "status"] == 200)
    {
        show_debug_message("Set() function call succeeded!");
    }
    else
    {
        var errorMessage = async_load[? "errorMessage"]
    }
}
```

The code above matches the response against the correct event **type** and logs the success of the task.

<dbReference>.Update

This method updates the given document on the database, without deleting any omitted keys. This function returns a listener identifier.

This is an asynchronous function that will trigger the **Async Social** event when the task is finished.

NOTE This function will only work on *dbDocuments*; if you pass in a *dbCollection* it won't be updated.

Syntax:

```
<dbReference>.Update(json)
```

Argument	Type	Description
json	string	A JSON formatted string to update the document with.

Returns:

```
Real (Asynchronous Listener ID)
```

Triggers:

```
Asynchronous Social Event
```

Key	Type	Description
type	string	The string `"FirebaseFirestore_Document_Update"`
listener	real	The asynchronous listener ID.
path	string	The path of the database reference.
status	real	The HTTP status response code (see reference)
errorMessage	string	The readable error message OPTIONAL

Example:

```
var json = json_stringify({ level: 200 });
listenerId = FirebaseFirestore("myCollection/newDoc").Update(json);
```

The code above creates a JSON formatted string of a struct (using the **json_stringify** function) and then updates the document (`"myCollection/newDoc"`) with the new key/value pairs (omitted keys will not be deleted). The function call will then return a listener ID (`listenerId`) that can be used inside an **Async Social** event.

```
if (async_load[? "type"] == "FirebaseFirestore_Document_Update")
{
    if (async_load[? "status"] == 200)
    {
        show_debug_message(".Update() function call succeeded!");
    }
    else
    {
        var errorMessage = async_load[? "errorMessage"]
    }
}
```

The code above matches the response against the correct event **type** and logs the success of the task.

<dbReference>.Query

This method executes a query and returns a listener identifier.

This is an asynchronous function that will trigger the **Async Social** event when the task is finished.

Syntax:

```
<dbReference>.Query()
```

Returns:

```
Real (Asynchronous Listener ID)
```

Triggers:

```
Asynchronous Social Event
```

Key	Type	Description
type	string	The string `"FirebaseFirestore_Collection_Query"`
listener	real	The asynchronous listener ID.
path	string	The path of the database reference.
status	real	The HTTP status response code (see reference)
value	string	A JSON formatted string of an **array**
errorMessage	string	The readable error message OPTIONAL

Example:

```
listenerId = FirebaseFirestore("myCollection").OrderBy("Points",
"ASCENDING").Limit(10).Query();
```

The code above performs a query (using the **Query** function) on the database reference `"myCollection"` ordered by the `"Points"` field in an ascending order and returns the first `10` elements (using the **Limit** function). The function call will then return a listener ID (`listenerId`) that can be used inside an **Async Social** event.

```
if (async_load[? "type"] == "FirebaseFirestore_Collection_Query")
{
    if (async_load[? "status"] == 200)
    {
        show_debug_message(async_load[? "value"]);
    }
    else
    {
        var errorMessage = async_load[? "errorMessage"];
    }
}
```

The code above matches the response against the correct event **type** and logs the results of the performed query.

<dbReference>.EndAt

This method is a filter that should be used when preforming a Query that is ordered by a specific field. The filter assures that the queried documents stop at a specific value (relative to the field specified in OrderBy).

Syntax:

```
<dbReference>.EndAt(value)
```

Argument	Type	Description
value	real/string	The value to stop the query at.

Returns:

```
dbReference Struct
```

Example:

```
listenerId = FirebaseFirestore("myCollection").OrderBy("Points",
"ASCENDING").EndAt(99).Query();
```

The code above performs a query (using the Query function) on the database reference `"myCollection"` ordered by the `"Points"` field in an ascending order (using the OrderBy function) ending the query results when the `"Points"` reach the value `99`.

<dbReference>.Limit

This method is a filter that should be used when preforming a Query that will limit the number of queried documents.

Syntax:

```
<dbReference>.Limit(value)
```

Argument	Type	Description
value	real	The maximum number of documents to return.

Returns:

```
dbReference Struct
```

Example:

```
listenerId = FirebaseFirestore("myCollection").OrderBy("Points",
"ASCENDING").Limit(10).Query();
```

The code above performs a query (using the Query function) on the database reference `"myCollection"` ordered by the `"Points"` field in an ascending order (using the OrderBy function) and returns the first `10` elements.

<dbReference>.OrderBy

This method is a filter that should be used when preforming a Query that will return documents ordered by a specific field.

NOTE This function does not ensure that the data you receive from your query will be sorted, however any other filter functions being called after this will see the data sorted in the correct order. For example, if you call this function to sort a collection in an **ascending order** and then call `.Limit(5)`, you will get the first 5 items from that **sorted list** however those 5 items may not get to you in the same order.

Syntax:

```
<dbReference>.OrderBy(path, direction)
```

Argument	Type	Description
path	string	The path to the field to be used for sorting/ordering
direction	string	The direction to order by `"ASCENDING"` or `"DESCENDING"`

Returns:

```
dbReference Struct
```

Example:

```
listenerId = FirebaseFirestore("myCollection").OrderBy("Points",
"ASCENDING").Limit(10).Query();
```

The code above performs a query (using the Query function) on the database reference `"myCollection"` ordered by the `"Points"` field in an ascending order and returns the first `10` elements (using the Limit function).

<dbReference>.StartAt

This method is a filter that should be used when preforming a Query that is ordered by a specific field. The filter assures that the queried documents start at a specific value (relative to the field specified in OrderBy).

Syntax:

```
<dbReference>.StartAt(value)
```

Argument	Type	Description
value	real/string	The value to start the query at.

Returns:

```
dbReference Struct
```

Example:

```
listenerId = FirebaseFirestore("myCollection").OrderBy("Points",
"ASCENDING").StartAt(10).Query();
```

The code above performs a query (using the Query function) on the database reference `"myCollection"` ordered by the `"Points"` field in an ascending order (using the OrderBy function) starting the query results when the `"Points"` fields is higher then `10`.

<dbReference>.Where

This method is a filter that should be used when preforming a Query and allows you to perform comparison operations on a field's values.

Syntax:

```
<dbReference>.Where(path, operation, value)
```

Argument	Type	Description
path	string	The path to field.
operation	string	Any one of the following operations (as strings): `"=="`, `"!="`, `""`, `">"`, `">="`, `"="`
value	real/ string	The value used for the comparison.

Returns:

```
dbReference Struct
```

Example:

```
listenerId = FirebaseFirestore("myCollection").Where("Points", ">=", 1000).Query();
```

The code above performs a query (using the Query function) on the database reference `"myCollection"` returning every element where the `"Points"` field is more than or equal to `1000`.

<dbReference>.WhereEqual

This method is a filter that can be used when preforming a Query and selects elements where the field's `path` value is equal to `value`.

Syntax:

```
<dbReference>.WhereEqual(path, value)
```

Argument	Type	Description
path	string	The path to field.
value	real/string	The value used for the comparison.

Returns:

```
dbReference Struct
```

Example:

```
listenerId = FirebaseFirestore("myCollection").WhereEqual("Points", 1000).Query();
```

The code above performs a query (using the Query function) on the database reference `"myCollection"` returning every element where the `"Points"` field is equal to `1000`.

<dbReference>.WhereGreaterThan

This method is a filter that can be used when preforming a Query and selects elements where the field's `path` value is greater than `value`.

Syntax:

```
<dbReference>.WhereGreaterThan(path, value)
```

Argument	Type	Description
path	string	The path to field.
value	real/string	The value used for the comparison.

Returns:

```
dbReference Struct
```

Example:

```
listenerId =
FirebaseFirestore("myCollection").WhereGreaterThan("Points", 1000).Query();
```

The code above performs a query (using the Query function) on the database reference `"myCollection"` returning every element where the `"Points"` field is greater than `1000`.

<dbReference>.WhereGreaterThanOrEqual

This method is a filter that can be used when preforming a Query and selects elements where the field's `path` value is greater than or equal to `value`.

Syntax:

```
<dbReference>.WhereGreaterThanOrEqual(path, value)
```

Argument	Type	Description
path	string	The path to field.
value	real/string	The value used for the comparison.

Returns:

```
dbReference Struct
```

Example:

```
listenerId =
FirebaseFirestore("myCollection").WhereGreaterThanOrEqual("Points", 1000).Query();
```

The code above performs a query (using the Query function) on the database reference `"myCollection"` returning every element where the `"Points"` field is greater than or equal to `1000`.

<dbReference>.WhereLessThan

This method is a filter that can be used when preforming a Query and selects elements where the field's `path` value is less than `value`.

Syntax:

```
<dbReference>.WhereLessThan(path, value)
```

Argument	Type	Description
path	string	The path to field.
value	real/string	The value used for the comparison.

Returns:

```
dbReference Struct
```

Example:

```
listenerId = FirebaseFirestore("myCollection").WhereLessThan("Points", 1000).Query();
```

The code above performs a query (using the Query function) on the database reference `"myCollection"` returning every element where the `"Points"` field is less than `1000`.

<dbReference>.WhereLessThanOrEqual

This method is a filter that can be used when preforming a Query and selects elements where the field's `path` value is less than or equal to `value`.

Syntax:

```
<dbReference>.WhereLessThanOrEqual(path, value)
```

Argument	Type	Description
path	string	The path to field.
value	real/string	The value used for the comparison.

Returns:

```
dbReference Struct
```

Example:

```
listenerId =
FirebaseFirestore("myCollection").WhereLessThanOrEqual("Points", 1000).Query();
```

The code above performs a query (using the Query function) on the database reference `"myCollection"` returning every element where the `"Points"` field is less than or equal to `1000`.

<dbReference>.WhereNotEqual

This method is a filter that can be used when preforming a Query and selects elements where the field's `path` value is not equal to `value`.

Syntax:

```
<dbReference>.WhereNotEqual(path, value)
```

Argument	Type	Description
path	string	The path to field.
value	real/string	The value used for the comparison.

Returns:

```
dbReference Struct
```

Example:

```
listenerId = FirebaseFirestore("myCollection").WhereNotEqual("Points", 1000).Query();
```

The code above performs a query (using the Query function) on the database reference `"myCollection"` returning every element where the `"Points"` field is not equal to `1000`.