Lab 11: Async JS - GET

Version 2: RESTful Client App - GET Request to Google Docs

Blog Comments

Table of Content for Implementing Blog Comments:

Lab Introduction

Prerequisites
Must have completed Lab 10: Async JS, as this lab is a direct continuation from it.

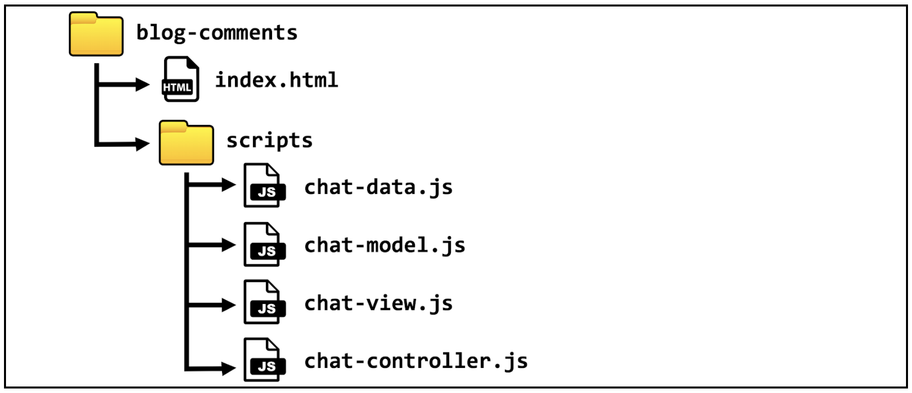
Motivation
Abstract thing we want to achieve, build a browser app (clientend) that sends & receive data from a backend service. Focus only on client-side, thus we'll use a 3rd party backend service to send http requests. Later labs, we will learn to deploy our own backend services.

Goal
BD, What we will achieve implementation wise (practical app)

Learning Objectives
- TBD, individual concepts covered
- API keys to http request from a backend service

Overview
1 Create Google Form for POST Request
2 Define HTML structure (Presentation Prototype)
3 Use async JavaScript to POST to Google DB
4 Publish Google Sheet for GET Request
5 Use async JavaScript to GET from Google DB
6 Response Values display to DOM

Client-side Architecture:
Start this project by making a project folder where all your assets & scripts will be organized.
Create all necessary files and folders as illustrated below.



Iteration 1: Google Docs as Backend DB

'Plan' Phase -- Approach

Goal #1: Create a Google Form to store chat data submissions

Approach: TBD

TBD

Concepts	
Backend Services:	Client (Browser) submits data to backend server to store for data persistence.
Endpoints:	The URL Route & HTTP Verb the server listens on to send/receive HTTPrequests

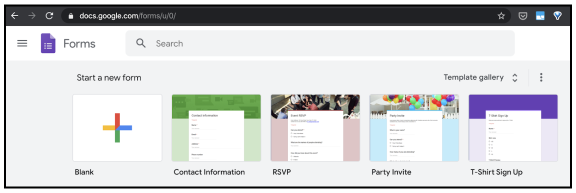
---

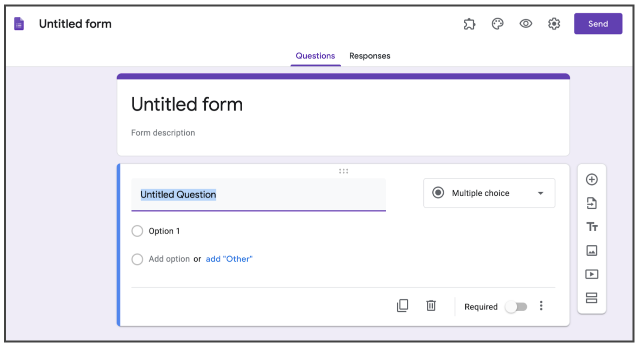
'Do' Phase -- Apply

Step 1: Open Chrome browser & go to URL: forms.google.com

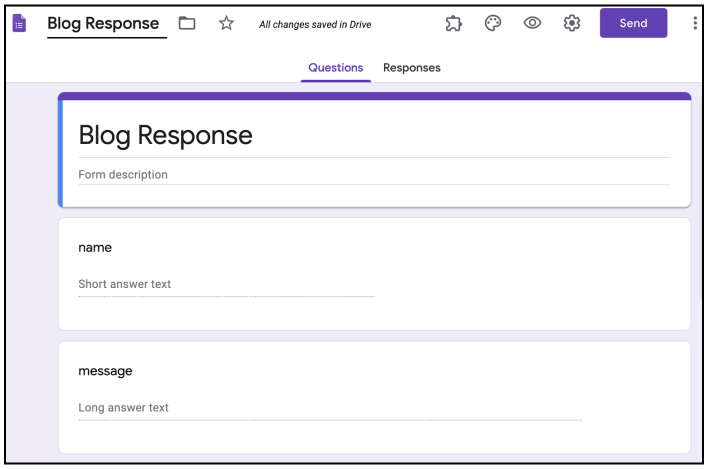
Google Forms - forms.google.com

Step 2: Create a new Blank Google form





Step 3: Fill out the Google form with required input fields

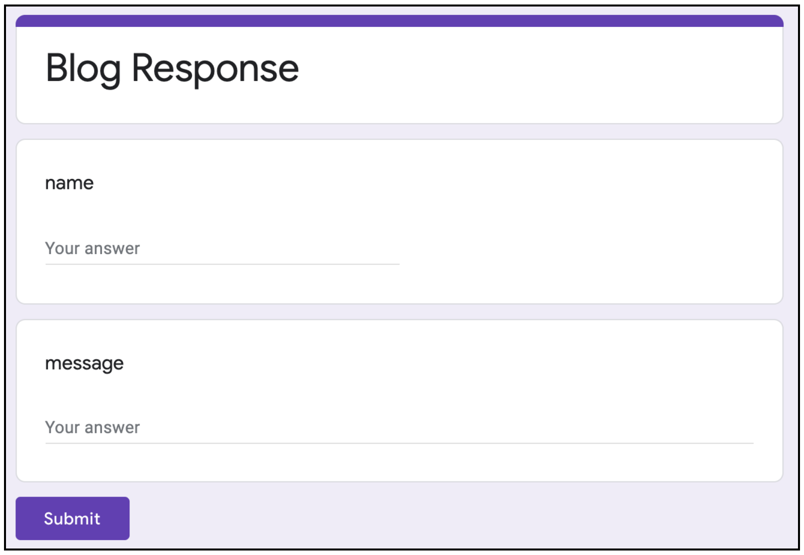


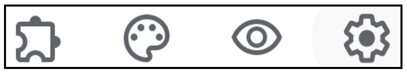
Form Name:
Blog Response

Add Question
label: **name**, *type:* ***Short answer***

Add Question
label: **message**, *type:* ***Paragraph***

Step 4: In Top-Right Menu, Click Preview icon to Publish Form





Preview icon

Step 5: Get Entry Names as Endpoint Targets

Note: This step can be achieved either using the inspect option on each input element or by querying the DOM via JavaScript. This lab uses the later JavaScript approach.

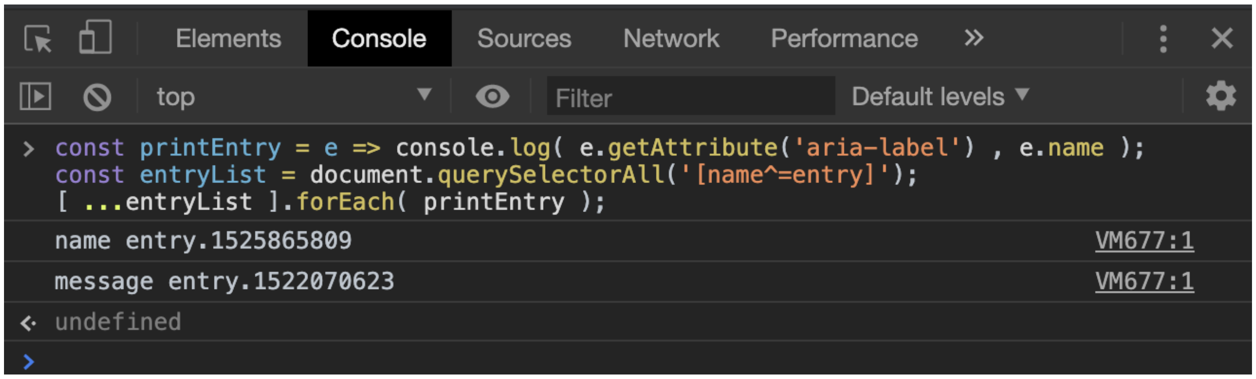
Open console & type in the JavaScript statements:

```
const printEntry = e => console.log( e.getAttribute('aria-label') , e.name );
const entryList = document.querySelectorAll('[name^=entry]');
[ ...entryList ].forEach( printEntry );
```

Code Explanation:

- Line 1: define printEntry function that displays a HTML label and name
- Line 2: regex query on DOM to select all elements where name attribute contains 'entry'
- Line 3: destructure list into Array to use forEach function to print each item

Example output in Console



```
> const printEntry = e => console.log( e.getAttribute('aria-label') , e.name );
  const entryList = document.querySelectorAll('[name^=entry]');
  [ ...entryList ].forEach( printEntry );
  name entry.1525865809                                               VM677:1
  message entry.1522070623                                            VM677:1
<· undefined
>
```

Note: The entry values will be different for every form.

Entry Endpoints

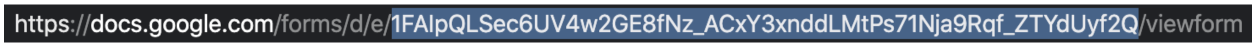
These entry names will be the endpoints that your HTML inputs will have to send data to in order to store it within Google Docs.

Step 6: Get Form ID as Endpoint URL

The form id is different for every google form. This step may be achieved in two ways: use the url address bar or query the DOM via JavaScript. Both approaches are shown below..

Approach 1: Use Browser Address bar

Copy form id from browser address bar as highlighted.



```
https://docs.google.com/forms/d/e/1FAIpQLSec6UV4w2GE8fNz_ACxY3xnddLMtPs71Nja9Rqf_ZTYdUyf2Q/viewform
```

Approach 2: Use DOM in Console

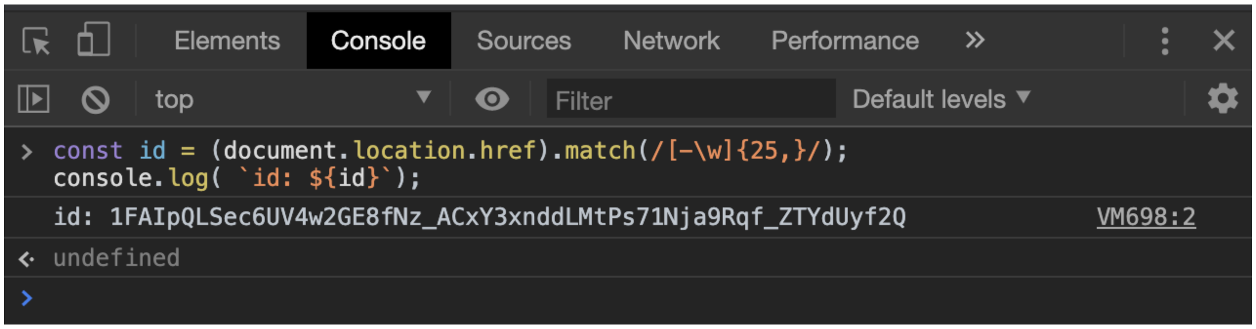
Open console & type in the JavaScript statements:

```
const id = (document.location.href).match(/[-\w]{25,}/);
console.log( `id: ${id}`);
```

Code Explanation:

- *Line 1:* regex match (i.e. string width of 25+) on DOM's href to isolate id
- *Line 2:* display id to console

Example output in Console



```
> const id = (document.location.href).match(/[-\w]{25,}/);
  console.log( `id: ${id}`);
  id: 1FAIpQLSec6UV4w2GE8fNz_ACxY3xnddLMtPs71Nja9Rqf_ZTYdUyf2Q          VM698:2
<· undefined
>
```

URL Endpoints

This id defines the url path that the HTML request must be routed to post data to the endpoints within Google Docs.

***Summary*: Form Endpoint IDs** (*Note: These values vary for every form*)

id	1FAIpQLSec6UV4w2GE8fNz_ACxY3xnddLMtPs71Nja9Rqf_ZTYdUyf2Q
name	entry.1525865809
message	entry.1522070623

Iteration 2: HTML Elements for Chat App

'Plan' Phase -- Approach

Goal #2: Define HTML file with the necessary presentation data

Approach: **HTML 'Dummy' Prototype of the Chat App**

Start the client-end with the HTML presentation elements that this project requires. The HTML serves as a Proof of Concept model of the app. Each iteration hereafter focuses on wiring new interactivity to these HTML components

'Do' Phase -- Apply

HTML Steps

`index.html` *[Frontend]* must have same set of inputs as required by the Google form *[Backend]*

index.html → `<body></body>`

```
<ul id='chat-list'></ul>
<div id='chat-form'>
   <div>
        <input type="text" id="name" placeholder="name">
   </div>
   <div>
        <textarea rows="4" id="message" placeholder="message"></textarea>
   </div>
   <button id="submit"> Submit </button>
</div>
<script src='scripts/chat-data.js'></script>
<script src='scripts/chat-model.js'></script>
<script src='scripts/chat-view.js'></script>
<script src='scripts/chat-controller.js'></script>
```

'Test' Phase -- Assess

Open `index.html` in browser & ensure all HTML elements render & no errors report in console.

Iteration 3: Async JS to POST to Google DB

'Plan' Phase -- Approach

Goal #3: Use Asynchronous JavaScript to POST data to backend service

Approach: **JavaScript to send a HTTP POST Request to backend service.**

This iteration handles instantiating HTTP POST Requests with JavaScript and sent with a fetch request performed within an async/await function.

Concepts	
Request	https://developer.mozilla.org/en-US/docs/Web/API/Request
fetch	https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/fetch
async	https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function
await	https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await

---

'Do' Phase -- Apply

JavaScript Steps

1. ***chat-data***: defines all data values required for performing chat communications
2. ***chat-controller***: defines operations that sets up event listeners
3. ***chat-mode***l: defines all operations to construct & transmit a request to backend service
4. ***chat-view***: defines operations that update the DOM

Step 1: Data: defines the data values from the Google Form & the URL to send POST request

chat-data.js

```
//Data: POST REQUEST KEYS
const formId = "1FAIpQLSec6UV4w2GE8fNz_ACxY3xnddLMtPs71Nja9Rqf_ZTYdUyf2Q";
const name = "entry.1525865809";
const message = "entry.1522070623";

//Data: URL REQUEST
const urlPOST=`https://docs.google.com/forms/d/e/${formId}/formResponse`;
```

Note: formId, name, message will vary for every google form

Step 2: Controller: defines the event listener for submit button

chat-controller.js

```
const initControllers = function(){
  const submitButton = document.getElementById('submit');
  submitButton.addEventListener('click', submitEvent);
}

const submitEvent = function(){
  const formData = new Object();
  formData[name] = document.getElementById('name').value;
  formData[message] = document.getElementById('message').value;

  postToGoogleDB(formData);
}

initControllers();   //Must be last line of code
```

Step 2: Model: defines all operations to construct & transmit a request to backend service

chat-model.js

```
const postToGoogleDB = function(data){
  const urlEncoded = encodeURL( urlPOST, data);
  const request = initRequest('POST', urlEncoded, 'no-cors');
  sendRequest(request)
      .then( responseEventPOST );
}

const encodeURL = function(path, params){
  const url = new URL(path);
  for (let key in params){
      url.searchParams.set( key, params[key] );
  }
  return url;
}

const initRequest = function(verb, url, mode='cors'){
  const init = new Object();
  init.method = verb;
  init.mode = mode;
  return  new Request(url, init);
}

const sendRequest = async function(request) {
   const response = await fetch(request);
   return response;
}

const responseEventPOST = response => console.log('POST Success',response)
```

'Test' Phase -- Assess

Open `index.html` in browser & submit a message. Verify that messages POST to the Google form.

Iteration 4: Publish GoogleSheet for GET Request

'Plan' Phase -- Approach

Goal #4: Publish a Google Sheets from the Google Form to access the chat data

Approach: Google Sheet API provides HTTP GET access to sheet data

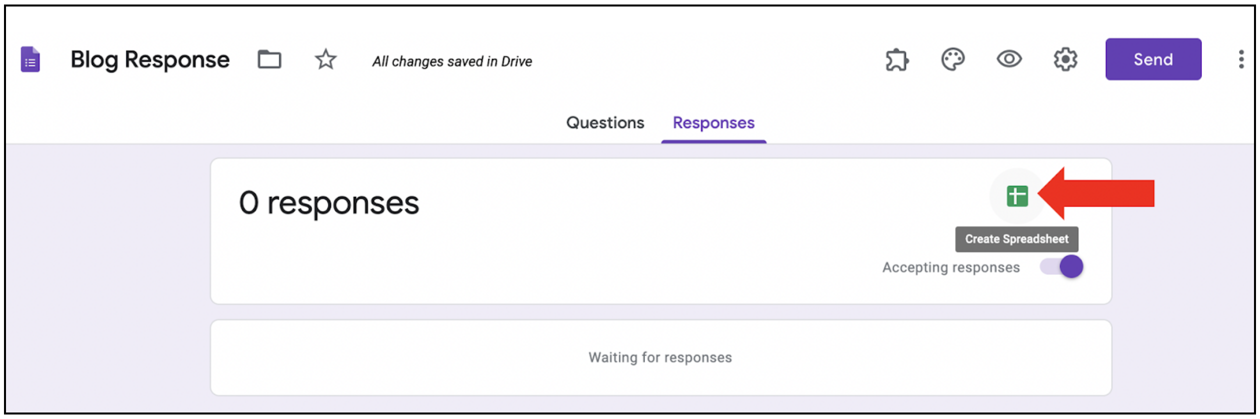
TBD

Concepts	
HTTP GET	GET is a request to retrieve data from a backend service
API Key	REST Requests from a server may require authentication such as tokens or API keys

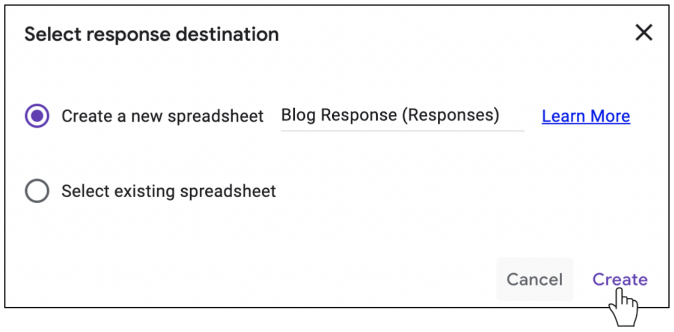
'Do' Phase -- Apply

Step 1: In Google Form Responses, select Create Spreadsheet

Screenshot of Google forms - Edit view



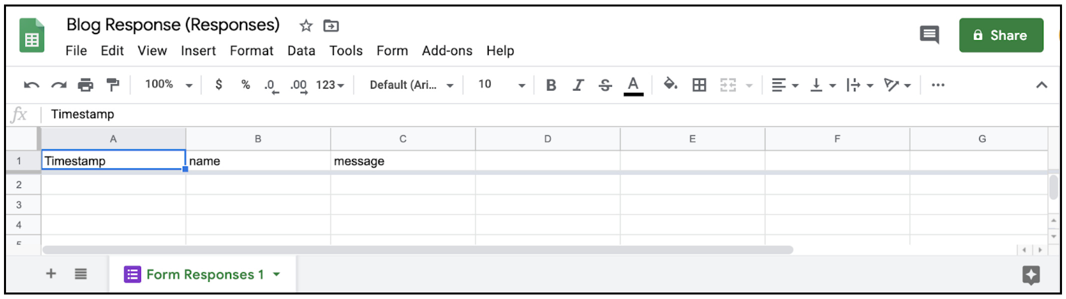
Step 2: Click Create



Select response destination

Create a new spreadsheet Blog Response (Responses) Learn More

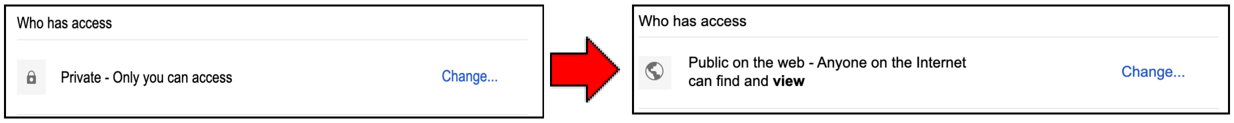
Select existing spreadsheet



Blog Response (Responses)

	A	B	C	D	E	F	G
1	Timestamp	name	message				
2							
3							
4							

Step 3: Share → Advanced → Who has access



Who has access

Private - Only you can access

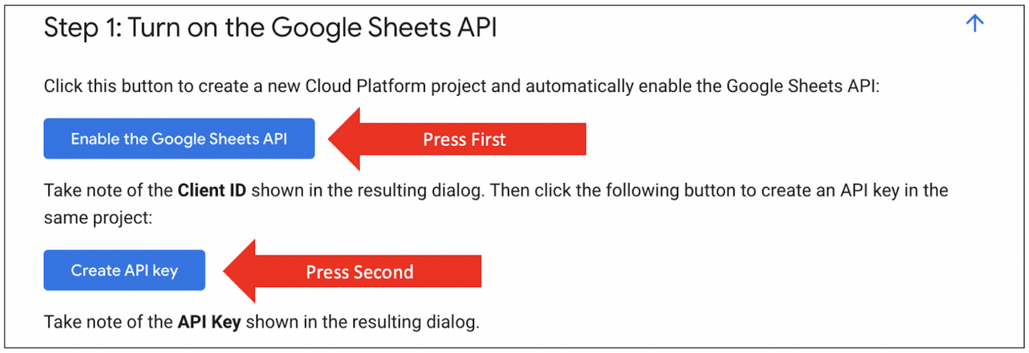
Who has access

Public on the web - Anyone on the Internet can find and **view**

*Change Access from Private to **Public on the Web.***

Step 4: Turn on the Google Sheet API & get an API key

Goto *Step 1* of https://developers.google.com/sheets/api/quickstart/js



Step 1: Turn on the Google Sheets API

Click this button to create a new Cloud Platform project and automatically enable the Google Sheets API:

Take note of the **Client ID** shown in the resulting dialog. Then click the following button to create an API key in the same project:

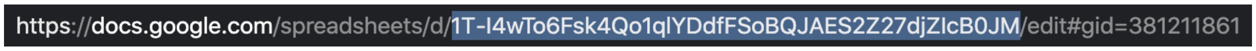
Take note of the **API Key** shown in the resulting dialog.

Note: Step First: Click Done, Step Second: Copy your API key for the next iteration

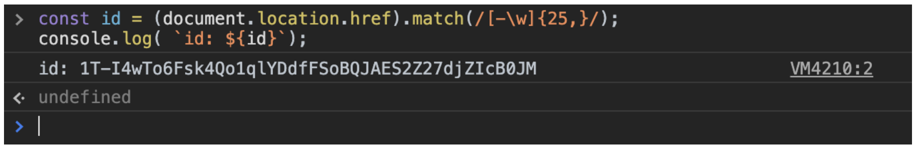
'Test' Phase -- Assess

Go to Google Sheet API page for GET test a query with your Sheet ID
https://developers.google.com/sheets/api/reference/rest/v4/spreadsheets/get

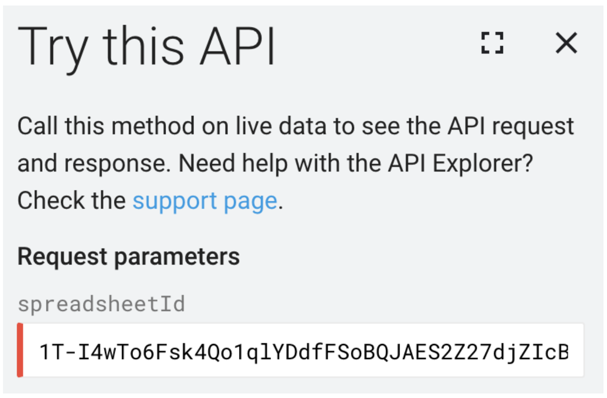
Note: Sheet ID is different than the Form ID.


https://docs.google.com/spreadsheets/d/1T-I4wTo6Fsk4Qo1qlYDdfFSoBQJAES2Z27djZIcB0JM/edit#gid=381211861

Note: Get the Sheet ID from the URL Address bar


```
> const id = (document.location.href).match(/[-\w]{25,}/);
  console.log( `id: ${id}`);
  id: 1T-I4wTo6Fsk4Qo1qlYDdfFSoBQJAES2Z27djZIcB0JM                VM4210:2
<· undefined
> |
```

Note: Alternatively, use can use console code to get Sheet ID, same as Form ID

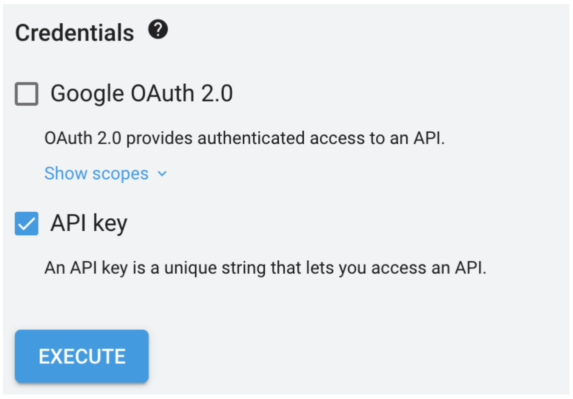

Try this API

Call this method on live data to see the API request and response. Need help with the API Explorer? Check the support page.

Request parameters

spreadsheetId

1T-I4wTo6Fsk4Qo1qlYDdfFSoBQJAES2Z27djZIcB


Credentials

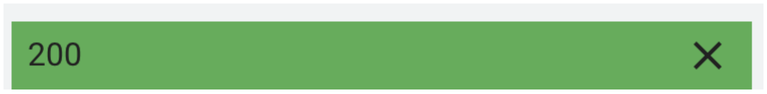
☐ Google OAuth 2.0

OAuth 2.0 provides authenticated access to an API.

Show scopes

✓ API key

An API key is a unique string that lets you access an API.


200

Iteration 5: Client's GET Request to Google DB

'Plan' Phase -- Approach

Goal #5: Client performs a GET Request to Google Sheets & handle HTTP Response

Approach: **Use async fetch method to manage GET Requests to Google API**

TBD

Concepts	
HTTP GET	GET is a request to retrieve data from a backend service
API Key	REST Requests from a server may require authentication such as tokens or API keys

---

'Do' Phase -- Apply

JavaScript Steps

1. ***chat-data***: Refactor to include GET Requests data
2. ***chat-mode***l: Create a function to get data from the 'Google Database'
3. ***chat-mode***l: Create a callback function in the event of a GET response from backend service
4. ***chat-mode***l: Refactor the callback function for POST response to invoke a GET Request

Step 1: Data: Initialize variables to store keys, ids, & URL paths to server's GET endpoints

chat-data.js

```
//Data: POST REQUESTS
const formId = "1FAIpQLSdOWILDHlIwUzmjH1VQMg9T7zqiXmwz3VW6mwYfKqWPNCFCkA";
const name = "entry.1864339201";
const message = "entry.931995720";
const urlPOST=`https://docs.google.com/forms/d/e/${formId}/formResponse`;

//Data: GET REQUESTS
let sheetId = '1T-I4wTo6Fsk4Qo1qlYDdfFSoBQJAES2Z27djZIcB0JM';
const key = "AIzaSyBYlTNZJsj4HXHpzs99ToTqGF9k9qhjEG8";
const sheet = encodeURIComponent('Form Responses 1');
const urlGET=`https://sheets.googleapis.com/v4/spreadsheets/${sheetId}/values/${sheet}?key=${key}`;
```

Note: formId, name, message will vary for every google form

Step 2: Model: Method responsible for managing GET requests to Google database

chat-model.js

```
const getFromGoogleDB = function(){
  const request = initRequest('GET', urlGET);
  sendRequest(request)
      .then( responseEventGET );
}
```

Step 3: Model: Callback for GET response event must be async since we need to wait for the data to be delivered to the client before it can be displayed.

chat-model.js

```
const responseEventGET = async function (response){
   const sheetData = await response.json();
   console.log(sheetData);
}
```

Step 4: Model: Refactor existing method, In response to a POST Request, the client should perform a GET request to get all the blog comments along with the new post.

chat-model.js

```
const responseEventPOST = response => getFromGoogleDB();
```

'Test' Phase -- Assess

Open `index.html` in browser & submit a message. Verify that a GET response prints to console.

Iteration 6: Response Values display to DOM

'Plan' Phase -- Approach

Goal #6: Display the GET Response Data to the Comments List in HTML via DOM

Approach: **Implement chat-view to take Response values & format to HTML list items**

Concepts	
Destructure	Array destructuring allows for-loop to assign a row of values into named variables

---

'Do' Phase -- Apply

JavaScript Steps

1. ***chat-model***: Refactor callback for a GET response to invoke view's show comments function
2. ***chat-view***: Initialize all views, which requires GET request from Google DB for chat data
3. ***chat-view***: Create a 'show comments' function that manages displaying all the comment data
4. ***chat-view***: Create a 'add comment' function that adds a single comment to the HTML list
5. ***chat-view***: Create a 'clear chat' function that empties the HTML list before updating it.

Step 1: Model: Refactor GET response callback to send sheet data to a method in the view to display

chat-model.js

```
const responseEventGET = async function (response){
   const sheetData = await response.json();
   showComments(sheetData);
}
```

Step 2: View: Initializes all view components, must perform a GET request to Google DB to get data

chat-view.js

```
const initViews = function(){
   getFromGoogleDB();
}


initViews(); //last line of view
```

Step 3: View: Show comments manages iterating through all comment data, parsing it, & pass to helper

chat-view.js

```
const showComments = function(sheetData){
   clearChat();
   for (let row of sheetData.values ){
       addComment(row);
   }
}
```

Step 4: View: Helper function, converts data into an HTML element & adds it to the DOM as list item

chat-view.js

```
const addComment = function(row){
   const [time, name, message] = row;
   const chatList = document.getElementById('chat-list');
   chatList.innerHTML += `<li>${time} ${name} ${message}</li>`;

}
```

Step 5: View: Helper function, clears the HTML list before rewriting comments to it.

chat-view.js

```
const clearChat = function(){
   const chatList = document.getElementById('chat-list');
   chatList.innerHTML = '';
}
```

'Test' Phase -- Assess

Open `index.html` in browser & submit a message. Verify that comments display within the Browser viewport.

Concluding Notes

Chaining Async Callbacks

TBD

Backend services: web server vs data store

TBD

Final Comments

You can curate your comments by editing the spreadsheet entries in google sheets. Any changes made to the google spreadsheet will be reflected in the client view. Must refresh the page to see other user comments, this is typical to a Blog Comment system. To adapt this into a real-time chatroom, use setInterval function on to repeatedly invoke initViews every couple of seconds.

Future Improvements

- Use Bootstrap to better style your comments
- No authorization system is not a secure comment system
- No capability for user to edit comments or reply either
- Google Documents is very limited as a backend service
- Host your own backend service provides much more options & customizations

Lab Submission

Compress your project folder into a zip file and submit on Moodle.

Additional Upgrades: Convert to Realtime Chat

chat-view.js

```
setInterval(initViews, 1000) //last line of views code
```