

## Lab 06: *http + async*

## Frontend

---

# CONTACT FORM

*HTTP Request: POST + Async JS*

---

*Table of Content for Implementing Contact form:*

<i># of Parts</i>	<i>Duration</i>	<i>Topic</i>	<i>Page</i>
Introduction	5 minutes	<b>Overview for this Lab: Contact Forms</b> Designing version 1 of contact form	1
Goal 1	10 minutes	<b>Google Form as Backend DB</b> Create a Google form with the same input fields.	2
Goal 2	10 minutes	<b>HTML Input Form</b> Implement your contact form in HTML	3
Goal 3	10 minutes	<b>JS DOM sets HTML Form</b> TBD	7
Goal 4	10 minutes	<b>JS encode URL Query String</b> TBD	8
Goal 5	10 minutes	<b>HTTP Request in Asynchronous JS</b> TBD	9
End	5 minutes	<b>Concluding Notes</b> TBD	11

# Lab Introduction

## Prerequisites

First lab in the Async JavaScript sequence. Requires a Google account.

## Motivation

Learn to use HTTP to send requests to a backend service. This lab focuses exclusively on client-side, thus we'll send http requests to a pre-existing backend service.

## Goal

Implement the contact form on your portfolio page to store data to a google form

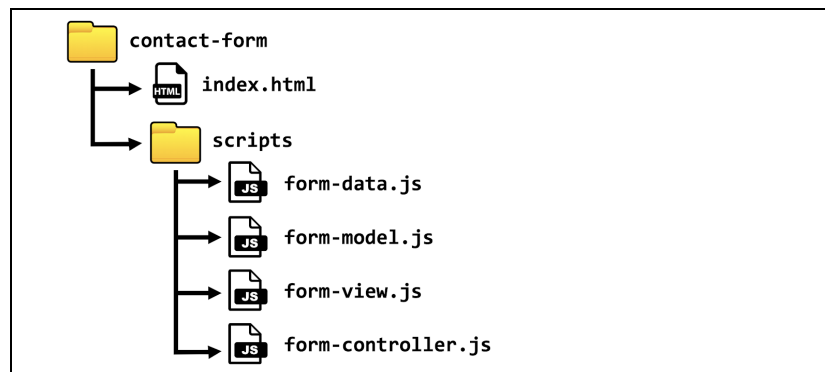
## Learning Objectives

- HTTP Request, POST
- async/await
- fetch method
- promises

## Overview

- 1 Create Google Form for POST Request
- 2 Send Request with HTML Form
- 3 JS DOM sets HTML Form
- 4 JS to encode URL Query String
- 5 HTTP Request in Asynchronous JS

## Project Architecture:



## Iteration 1: Google Docs as Backend DB

### 'Plan' Phase -- Approach

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

**Approach:** Use a pre-existing backend service to store form data

Google provides access to post form data free to all Google accounts.

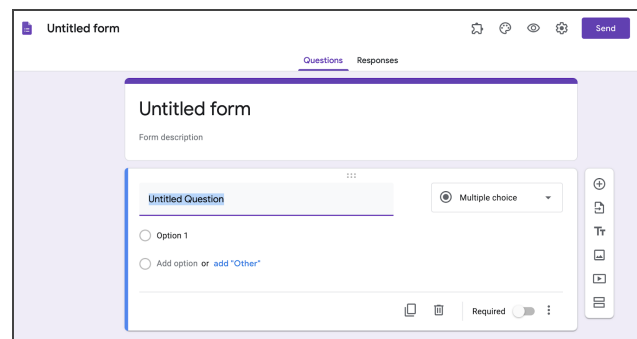
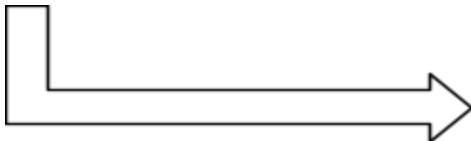
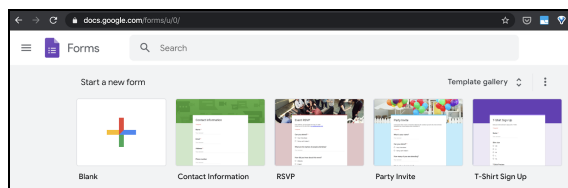
Concepts	
<b>Backend Services:</b>	Client (Browser) submits data to backend server to store for data persistence.
<b>Endpoints:</b>	The URL Route & HTTP Verb the server listens on to send/receive HTTPRequests

### 'Do' Phase -- Apply

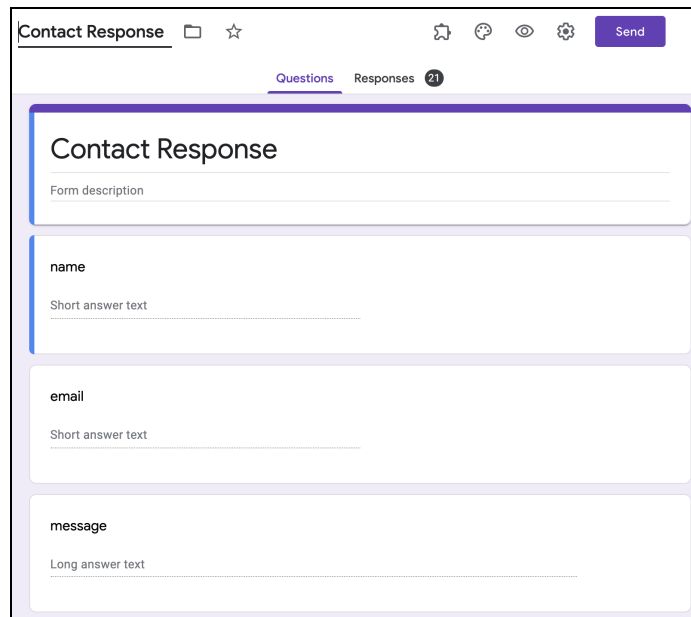
**Step 1:** Open Chrome browser & go to URL: [forms.google.com](https://forms.google.com)



**Step 2:** Create a new Blank Google form



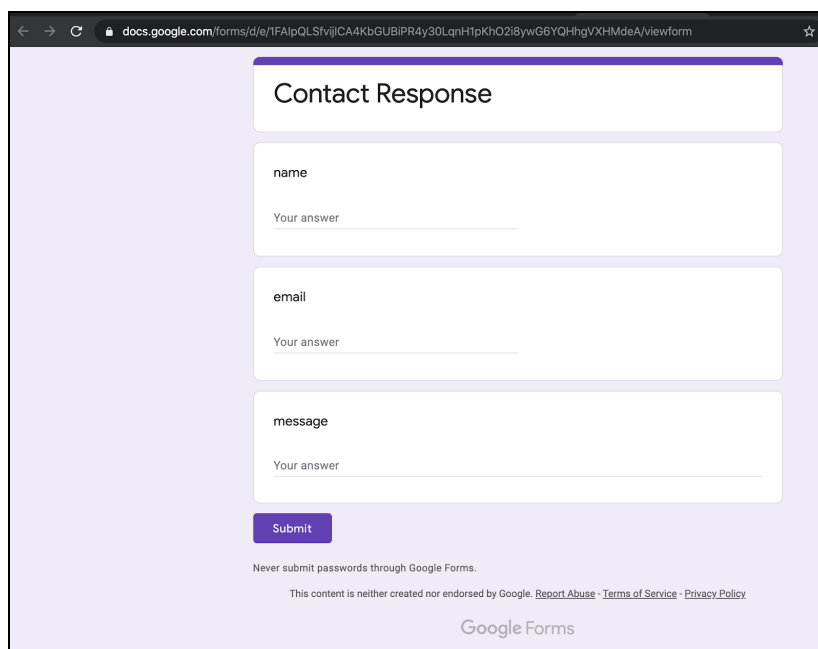
### Step 3: Fill out the Google form with required input fields



The screenshot shows the Google Forms editor interface for a form titled "Contact Response". The form has a description field and three question fields: "name" (Short answer text), "email" (Short answer text), and "message" (Long answer text). The "Responses" tab is selected, showing 21 responses. The "Send" button is visible in the top right corner.

- Form Name:  
**Contact Response**
- Add Question  
label: **name**, type: **Short answer**
- Add Question  
label: **email**, type: **Short answer**
- Add Question  
label: **message**, type: **Paragraph**

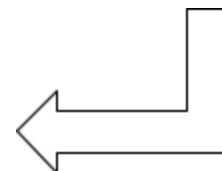
### Step 4: In Top-Right Menu, Click Preview icon to Publish Form



The screenshot shows the published version of the "Contact Response" form. It features the same three input fields: "name", "email", and "message", each with a "Your answer" label. A "Submit" button is at the bottom. The Google Forms logo and footer text are visible at the bottom of the 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.*

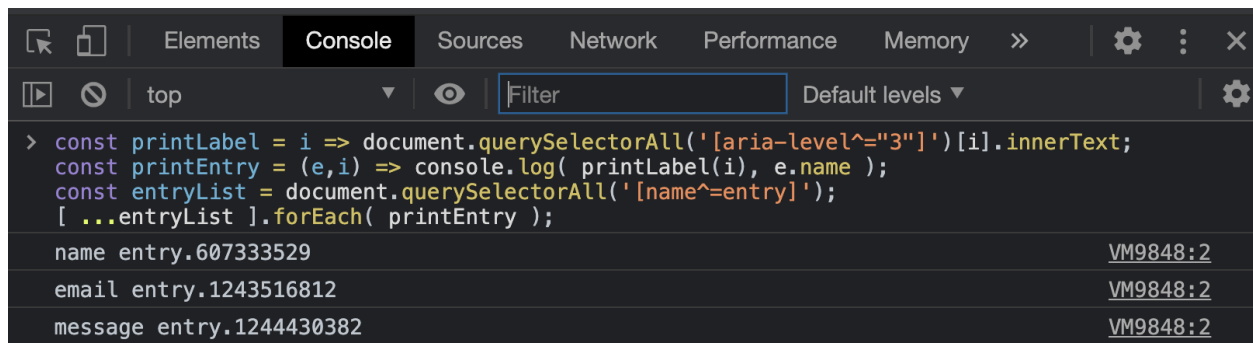
Open console & type in the JavaScript statements:

```
const printLabel = i => document.querySelectorAll('[aria-level^="3"]')[i].innerText;  
const printEntry = (e,i) => console.log( printLabel(i), e.name );  
const entryList = document.querySelectorAll('[name^=entry]');  
[ ...entryList ].forEach( printEntry );
```

Code Explanation:

- Line 1: define printLabel function that selects the aria label & returns its text
- Line 2: define printEntry function that displays a HTML label and name
- Line 3: regex query on DOM to select all elements where name attribute contains 'entry'
- Line 4: destructure list into Array to use forEach function to print each item

*Example output in Console*



The screenshot shows a web browser's developer console with the 'Console' tab selected. The console displays the following JavaScript code and its output:

```
> const printLabel = i => document.querySelectorAll('[aria-level^="3"]')[i].innerText;  
const printEntry = (e,i) => console.log( printLabel(i), e.name );  
const entryList = document.querySelectorAll('[name^=entry]');  
[ ...entryList ].forEach( printEntry );
```

The output consists of three lines, each showing a label and a name:

```
name entry.607333529 VM9848:2  
email entry.1243516812 VM9848:2  
message entry.1244430382 VM9848:2
```

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

## Entry Endpoints

These entry names are the endpoints that your HTML inputs use to send data to Google Forms in order to store it on their servers.

## 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/1FAIpQLSfvijICA4KbGUBiPR4y30LqnH1pKhO2i8ywG6YQHhgVXHMdeA/viewform>

### Approach 2: Use DOM in Console

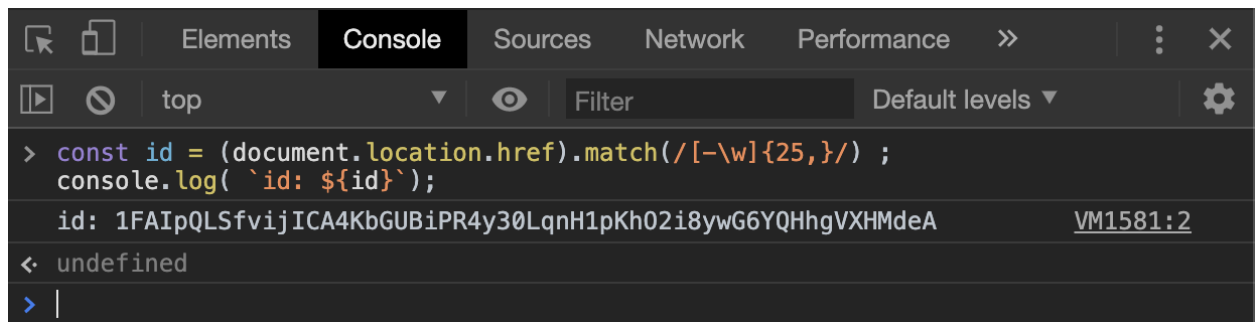
Open console & type in the JavaScript statements:

```
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



## 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	1FAIpQLSfvijICA4KbGUBiPR4y30LqnH1pKhO2i8ywG6YQHhgVXHMdeA
name	entry.2046953029
email	entry.916687864
message	entry.764058979

## Iteration 2: Send Request with HTML Form

### 'Plan' Phase -- Approach

**Goal #2:** Use the HTML form element to submit client data to a backend service

#### Approach: HTML Form, (HTML Only)

We'll use HTML to submit data to the backend service, i.e. Google Docs. HTML has `<form>` tags that can submit requests to a HTTP server. A HTTP request requires an **action** (*URL Endpoint*) & **method** (*HTTP verb: GET, POST, PUT, DELETE*). Note: Posting data to a Google spreadsheet uses a POST method

Concepts	
<b>Frontend service</b>	UI/UX. The browser maintains views & controls that the user interacts with
<b>Submit Event</b>	User triggers a request to a web server triggered by a submit event on the form.
<b>HTTP Request</b>	HTTP is protocol that browsers use to send request to Web servers
<b>HTML <code>&lt;form&gt;</code> tag</b>	HTML form can trigger a submit event to send HTTP request to web server

### '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>`

```
<form action="https://docs.google.com/forms/d/e/$id/formResponse" method="POST">
  <div>
    <input type="text" name="$entry1" placeholder="Name">
  </div>
  <div>
    <input type="email" name="$entry2" placeholder="Email">
  </div>
  <div>
    <textarea rows="4" name="$entry3" placeholder="Message"></textarea>
  </div>
  <button type="submit"> Submit </button>
</form>
```

*\*Note: Replace \$id, \$entry1, \$entry2, \$entry3 with the values for your Google Form from prior step*

### 'Test' Phase -- Assess

Open the `index.html` in a browser, then input data and submit. The page should redirect to a Google response page & the submitted data should appear within the Google spreadsheet. Disadvantages of this approach is that it offers no control in resolving server responses & everything is hardcoded to the HTML.



## Iteration 3: JS DOM sets HTML Form

### 'Plan' Phase -- Approach

**Goal #3:** Use Async JS to submit users data to backend service

### Approach: JavaScript to set Attributes for HTML Form

We'll use JavaScript to submit data to the backend service, i.e. Google Docs. JavaScript. This approach is similar to the prior, except that the form's attributes will be set within JavaScript. The HTTP request is performed by the HTML Form element's submit event.

Concepts	
<b>Form Attributes</b>	To submit a HTTP request, a HTML form must have: action (URL), method (HTTP Verb), and child values must define the name attribute with a unique label
<b>Submit Listener</b>	A 'submit' event listener may be set on a form triggered by a child button element

### 'Do' Phase -- Apply

### HTML Steps

**Step 1:** Create form with child inputs & submit button. Script tags to import external JS files

*index.html* → `<body></body>`

```
<form id='contact-form'>
  <div>
    <input type="text" id="name" placeholder="Name">
  </div>
  <div>
    <input type="email" id="email" placeholder="Email">
  </div>
  <div>
    <textarea rows="4" id="message" placeholder="Message"></textarea>
  </div>
  <button id="submit"> Submit </button>
</form>

<script src='scripts/form-model.js'></script>
<script src='scripts/form-controller.js'></script>
```

Code Explained:

- `<form>`, `<input>`, `<textarea>` needs an `id` to be accessed by JavaScript.
- `<form>` is the HTML element that triggers a submit event
- `<form>` must be the parent to all `<input>` elements that contain values to be submitted

## JavaScript Steps

**Step 1: Model Logic:** Initialize data & getPath function to return path string

*form-model.js*

```
const formId = "1FAIpQLSfvijICA4KbGUBiPR4y30LqnH1pKh02i8ywG6YQHhgVXHMdeA";
const entry1 = "entry.2046953029";
const entry2 = "entry.916687864";
const entry3 = "entry.764058979";

const getPath = formId => `https://docs.google.com/forms/d/e/${ formId }/formResponse`;
```

**Step 2: Controller Logic:** Add 'submit' listener to the HTML form via the DOM

*form-controller.js*

```
//Initializes all controllers
const initControllers = function(){
  const contactForm = document.getElementById('contact-form');
  contactForm.addEventListener('submit', submitEvent);
}

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

**Step 3: Controller Logic:** Define the behavior on a submit event. *i.e. set the form's attributes*

*form-controller.js*

```
//Submit event -- set HTTP attributes for the contact form
const submitEvent = function(){
  document.getElementById('name').name = entry1;
  document.getElementById('email').name = entry2;
  document.getElementById('message').name = entry3;

  const contactForm = document.getElementById('contact-form');
  contactForm.action = getPath(formId);
  contactForm.method = "POST";
}
```

Code Explained:

- A HTML form needs both a method & action to submit a request to a web server. Any data intended to be included with the submit must have a name attribute.

## 'Test' Phase -- Assess

Open index.html in a browser, input values and submit. The page should redirect to a Google response page & the submitted data should appear within the Google spreadsheet. Disadvantages of this approach is that it offers no control in resolving server responses.

## Iteration 4: JS to encode URL Query String

### 'Plan' Phase -- Approach

**Goal #4:** Use JS to define the action Query String that encodes data for server endpoint

### Approach: JavaScript to encode data as Query String sent by HTML Form

This iteration handles encoding data into the Query String sent by the HTML form. The objective is to start constructing the HTTP Request without depending on the form's builtin logic. With this approach, we decide what data to include in the request, but depend on the form's submit to trigger the HTTP request.

#### Concepts

<b>Query String</b>	Read for more info: <a href="https://en.wikipedia.org/wiki/Query_string">https://en.wikipedia.org/wiki/Query_string</a>
<b>URL object</b>	JavaScript's built-in URL object used to manage URLs and Query Strings
<b>encodeURIComponent</b>	JS method that encodes text into URI -- Try & see how data transmits in a URL

### 'Do' Phase -- Apply

#### HTML Steps

None. No refactoring to the HTML document in this iteration.

#### JavaScript Steps

1. Refactor submitEvent to get input values from DOM & pass them to function in the model.
2. Define sendToGoogleForm function that manages operations of sending data to a Google Form
3. Define getURL function that converts a path into a URL & encodes data into a Query String
4. Define initRequest function that initializes HTTP Request with an action (URL) & method (POST)

**Step 1: Controller:** Create a Key-Value pair to associate each entry name with its input value.

Send that object to a method in the model that will send it to Google Forms.

*form-controller.js → refactor*

```
const submitEvent = function(){
  const formData = new Object();
  formData[entry1] = document.getElementById('name').value;
  formData[entry2] = document.getElementById('email').value;
  formData[entry3] = document.getElementById('message').value;

  postToGoogleDB(formData);
}
```

**Step 2: Model:** Use form id to get path, use path & data to get URL with Query String. Use URL and 'POST' (HTTP verb) to initialize a HTTP request. Requires 3 helper functions.

*form-model.js*

```
const postToGoogleDB = function(data){  
  const path = getPath(formId);  
  const url = getURL( path, data)  
  initRequest('POST', url);  
}
```

**Step 3: Model:** *Helper function*, Instantiate URL object & set data to its searchParams attribute

*form-model.js*

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

**Step 4: Model:** *Helper function*, Initializes a request. Uses the form to do so. Sets form action with URL string & sets form method with the HTTP verb. Both passed in as parameters.

*form-model.js*

```
const initRequest = function(verb, url){  
  const contactForm = document.getElementById('contact-form');  
  contactForm.action = url.toString();  
  contactForm.method = verb;  
}
```

---

## 'Test' Phase -- Assess

Open index.html in a browser, input values and submit. The page should redirect to a Google response page & the submitted data should appear within the Google spreadsheet. Disadvantages of this approach is that it offers no control in resolving server responses.

## Iteration 5: HTTP Request in Asynchronous JS

### 'Plan' Phase -- Approach

**Goal #5:** Use Asynchronous JavaScript to submit data to backend service

**Approach:** JavaScript to instantiate & send HTTP Request to backend service.

This iteration handles instantiating HTTP Requests exclusively in JavaScript. No HTML form is required or utilized to trigger a submit. A JavaScript fetch request is performed within an async/await function.

Concepts	
<b>Request</b>	<a href="https://developer.mozilla.org/en-US/docs/Web/API/Request">https://developer.mozilla.org/en-US/docs/Web/API/Request</a>
<b>fetch</b>	<a href="https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/fetch">https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/fetch</a>
<b>async</b>	<a href="https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function">https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function</a>
<b>await</b>	<a href="https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await">https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await</a>

### 'Do' Phase -- Apply

#### HTML Steps

Refactor <form> tags into <div> tags, as it will not be used for submitting the HTTP request.

*index.html*

```
<div id='contact-form'>
  <div>
    <input type="text" id="name" placeholder="Name">
  </div>
  <div>
    <input type="email" id="email" placeholder="Email">
  </div>
  <div>
    <textarea rows="4" id="message" placeholder="Message"></textarea>
  </div>
  <button id="submit"> Submit </button>
</div>

<script src='scripts/form-model.js'></script>
<script src='scripts/form-controller.js'></script>
```

## JavaScript Steps

1. Refactor sendToGoogleForm to sendRequest, then on response event trigger a callback
2. Async function, get the Request & fetch it to the server, returns a Response object from server
3. Instantiates a Request object with URL and HTTP Verb
4. Callback function for a Response event, alerts a success message
5. Refactor event listener to button element instead of a form element

**Step 1: Model:** Refactor to invoke an asynchronous function to create & send HTTP Request

*form-model.js*

```
const postToGoogleDB = function(data) {  
  const path = getPath(formId);  
  const url = getURL(path, data);  
  sendRequest('POST', url)  
    .then(responseEvent);  
}
```

**Step 2: Model:** *Helper function*, Asynchronous function sends a Request to server with a fetch & returns a Response object

*form-model.js*

```
const sendRequest = async function(verb, url) {  
  const request = initRequest(verb, url);  
  const response = await fetch(request);  
  return response;  
}
```

**Step 3: Model:** *Helper function*, Instantiates a request object with URL object & HTTP method

*form-model.js*

```
const initRequest = function(verb, url) {  
  const init = new Object();  
  init.method = verb;  
  init.mode = 'no-cors';  
  return new Request(url, init);  
}
```

**Step 4: Model:** *Helper function*, Given a Response object, renders a success message.

*form-model.js*

```
const responseEvent = response => alert('Success!');
```

**Step 5: Controller:** Refactor to get button element with id 'submit' & to listen for 'click' event

*form-controller.js*

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

---

## **'Test' Phase -- Assess**

Open index.html in a browser, input values and submit. A popup success message should display. The advantage of this approach is that we fully control the request/response interactions between the Browser & web server.

## Concluding Notes

### Asynchronous JavaScript

Asynchronous execution of code is required for the purpose of sending/receiving data between a client & a server. This is because the client will have to wait for the server to receive the request & send a response, before any instruction requiring that request may be processed.

### HTTP Request / Response

HTTP is the protocol for clients to send a request to a web server whereby the server sends back a response. A request contains a route (URL) & verb: POST, PUT, GET, DELETE. A response has a code: 200, 300, 400, etc.

### Future Improvements

- This lab, we implemented POST request to send data to Google
- Next lab, we'll implement GET request to get data from Google to show user
- Google forms may be configured to send email notification on new submissions

### Lab Submission

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