

06 - Web APIs Technology Spike

COMP290 – Large Scale and Open Source Software Development
Dickinson College

Name:

Introduction:

In the prior activities you have learned how to structure a page using some basic HTML functionality including form elements and tables (03). You have used Vue.js and JavaScript to make a page dynamic by binding the content of some of its HTML elements to data in the Vue instance (04). Most recently, you used JavaScript functions and Vue directives to respond to user events (e.g. button clicks, key presses, etc). The code executed in response to these events modified the data in the Vue instance and thus, via Vue's data binding, altered how the page was rendered (05).

Thus, far all of the data that we have used has been hard coded into the HTML or JavaScript. In this activity you will learn about and gain experience using Application Programming Interfaces (APIs). These interfaces provide a way for JavaScript code in your page to request data from a web based service. Here specifically you'll be using the farmOS API to fetch and incorporate live data from the FarmData2 database into your report.

It is not required, but if you would like a good general introduction to the idea of APIs and what they do you can watch the video *What is an API? Introduction to Application Programming Interfaces with Google Maps APIs!* With Katherine from BlondieBytes:

- <https://www.youtube.com/watch?v=T74OdSCBJfw> (9:27)
 - Note that since this video was produced Google has begun requiring an API Key (a way to authenticating the user) to access its services. So, the examples won't run as they are shown. But this video still provides a good conceptual overview of APIs and how they work.

The farmOS API:

An API defines a set of *endpoints*, which are URLs that can be used to exchange information with a web service. For example, `/farm.json` is an endpoint in the farmOS API. Making a request to that endpoint on an farmOS server will provide some basic information about the farm.



1. Ensure that your FarmData2 instance is running and that you have logged in as manager1 (or 2) or worker1 (or 2,3,4,5). Then enter the following URL into your browser to make a request to the `/farm.json` endpoint:

`http://localhost/farm.json`

You should see a lot of lines with lots of `{ }` and `:` in them. If that is not what you see, ensure that your FarmData2 instance is running and that you are using a browser in your developer environment.


Copy and paste the first few lines of the response that you received here.

The Hoppscotch API Tool:

While you can interact with APIs through your browser, as you just did, its not very pretty or easy to read. Thus, developers will usually use specialized tools that make it much easier and more convenient. We'll use the *Hoppscotch application* to interact with and learn about APIs and the farmOS API in particular. Hoppscotch will allow you to manually send API requests to the server and inspect the responses that it returns. Doing this manually is an excellent way to learn about APIs. But it is also an essential technique for designing and testing calls to APIs that you will eventually integrate into code in an application. You'll see that once an API call has been designed and tested using Hoppscotch, it is relatively easy to translate it into JavaScript code that makes the requests and processes the results.

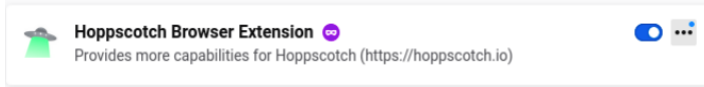
2. Visit the Hoppscotch page: <https://hoppscotch.io>.

3. You'll learn more about Hoppscotch and APIs in a minute, but first you'll need to install the Hoppscotch browser extension. To do so:

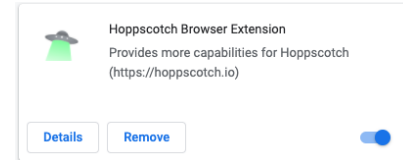
- Click on the "interceptor" icon () at the bottom left.
- Choose "Browser extension"
- Click on your browser "Firefox" or "Chrome"
 - This will take you to the to the Firefox Add-Ons site or the Chrome Web Store where you can install the extension.
- Click the "Add to..." button on the page that appears to install the extension.
- Reload the Hoppscotch page.

4. Find the "Add-Ons" (Firefox) or "Extensions" (Chrome) in your browser and confirm that you have the Hoppscotch Browser Extension installed. Be sure that you see one of the boxes below depending upon your browser before continuing:





Firefox



Chrome

5. You can now use Hoppscotch to make a request to the `/farm.json` endpoint. Enter the URL `http://localhost/farm.json` into the text field next to the word “GET” and click the “Send” button. When the server responds, text should appear in the “Response Body” area of the page and some green text should appear next to the “Status” label. If you do not see the response body or the status check to make sure FarmData2 is running, that you have logged in, that you correctly installed the Hoppscotch browser extension and that you correctly entered the endpoint URL.

6. When the server responds to a request Hoppscotch displays some information about the response:

a. It shows the *response status code* returned by the server. This code indicates if the request succeeded or failed (e.g. you’ve probably seen the infamous *404 Not Found* status before). What response status code did the farmOS server return in response to your request to the `/farm.json` endpoint?

b. It also shows the size (in bytes) of the response that was received. What was the size of the response returned in response to your request to the `/farm.json` endpoint?

API Requests and JSON:

Hoppscotch also displays the “Response Body,” which will contain the actual information that you are interested in. This is the same information you saw in your browser earlier, but it is now displayed in a nicely formatted and easier to read way. The farmOS API will provide its responses in *Java Script Object Notation (JSON)*. This JSON format should look reasonably familiar because it is the same format that we used in activity 05 to define JavaScript objects.

It is not required, but if you’d like to review or learn more about JSON format you can find more information using the links below:

- *An introduction to JSON* by Raivat Shah:
 - <https://towardsdatascience.com/an-introduction-to-json-c9acb464f43e>



- *JavaScript Object Initializers* in the MDN pages
 - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer

7. Just like a JavaScript object, JSON is made up of *properties* and *values*. Use the content of the “Response Body” from the request to the `/farm.json` in Hoppscotch to answer the following questions:

a. What is the *value* of the *farm property*?

b. What is the name of the *property* whose *value* tells us which version of the farmOS API that generated the response?

c. The property named “user” provides information about the user that made the request. Its value is an object. Copy and paste the JSON for that object here.

d. Give a reference to the value that indicates the name of the user that made the request. Hint: You’ll need to use “dot notation” to specify the sequence of property names that must be followed to get to the name.

e. Give a reference to the value that indicates the name of the language that is being used.

Adding Another Sub-Tab:

Now let’s extend the Harvest Report spike to make an API request and then does something with the response.



7. Synchronize the main branch of your local and origin FarmData2 repositories with the upstream and merge any changes to main into your feature branch (refer to past Activities if necessary). List here any files in the main branch that were changed. If there have been no changes to the main branch indicate that instead.

8. Make sure you have your feature branch checked out. Add another new sub-tab named API1 to the FD2 School tab. Have the contents of this new tab be provided by the file api1.html. Make a copy of your vue2.html file into the api1.html file. Don't forget to clear the Drupal cache when you are done. The result should be that you now have an API1 sub-tab that is (for now) identical to your Vue2 sub-tab. You'll be working on the API1 tab throughout this activity.

9. Commit your changes to your feature branch with a meaningful commit message and push it to your origin. Recall that this also updates your Draft Pull Request.

Adding to the Harvest Report - Spike 1:

As you saw above, the response from the `/farm.json` endpoint provides information about the farm including its name, the user that is logged in and the language being used. Those values are currently hard coded in our sample Harvest Report. We instead want to request them from the `/farm.json` endpoint. It will take a few steps to get there.

10. As a first step, bind the values for "Farm", "User" and "Language" in the report to data in the Vue instance. Set the initial values of your Vue properties to be empty strings. Reminder hint: Add data properties for these values and then use the "double mustache" in the report.

11. Add code to the click handler for the "Generate Report Button" that sets the values of the Vue properties you created in question #10 to "Sample Farm", "manager1" and "English". Now when the button is clicked, it will set the values in the Vue instance and the data binding will cause them to be displayed. The next step will be to actually get the values from an API request instead of hard coding them.

12. Commit your changes to your feature branch with a meaningful commit message and push it to your origin. Recall that this also updates your Draft Pull Request.

Making an API Call with Axios:

FarmData2 uses the *Axios* JavaScript library to make API requests from the farmOS API. In general, you won't need to use Axios directly because FarmData2 provides a convenience methods that hide some of the details. But because it is good to know, we'll make our first API request using Axios directly. After that, you'll learn more about some of the FarmData2



convenience methods and how to use them. Everything you need to know about Axios should be here, but if you'd like another resource at a similar level of detail you can check out *Using Axios to Consume APIs*:

- <https://vuejs.org/v2/cookbook/using-axios-to-consume-apis.html>

The basic format for an API request using Axios will be:

```
axios.get('api/endpoint/goes/here') // this line sends the request.
.then((response) => {
  // A:
  // Code here executes when the response is received.
  // response is an Object created from the "Response Body" JSON.
})
.catch((error) => {
  // B:
  // Code here executes if an error occurs processing the request.
  // error is an Object describing the error that occurred.
})
// C:
// Code here runs immediately after the request is sent,
// which will be before the code in then or catch.
```

13. Answer the following questions based on the above example to be sure you've picked up on the most relevant details.

a. Does the code in block A or B run when a successful response is received from the server?

b. Does the code in block A or B run if the server is unable to process the request?

c. Does the code in block C run before or after the request is sent?

d. Will the code in block C run before or after the code in block A or B?

e. Does the code in block C run before or after the response is received?



JavaScript Promises:

The call to the Axios get method above returns a JavaScript *Promise* object. Promise objects are used when a requested action will take an uncertain amount of time to complete. For example, when we make an API request to a server somewhere on the internet it is unknown how long it will take for that server to respond. Rather than waiting for the response, using a Promise allows the request to occur *asynchronously* while the program continues on by executing the code at point C. Then later, when the requested action completes (e.g. when the response is received) the code at point A in the `then` method (or at point B in the `catch` method will execute). In JavaScript terminology we say that the Promise is *resolved* when the requested action completes and the code in `then` method executes. If there is an error, and the code in the `catch` method executes, we say the Promise is *rejected*.

14. Answer the following questions based on the above paragraph to be sure you've picked up on the most relevant details.

a. When is a Promise useful? Give an example.

b. When is a Promise resolved?

c. Which block of code A, B or C executes when a Promise is resolved?

d. When is a Promise rejected?

e. Which block of code A, B or C executes when a Promise is rejected?



There is a good bit more to Promises, but that should be sufficient to get us going. It is not required reading, but if you'd like to learn more about Promises, Pangara provides a nice introduction in the blog post "An Introduction to Understanding JavaScript Promises":

- <https://medium.com/@PangaraWorld/an-introduction-to-understanding-javascript-promises-37eff85b2b08>

If you want more detail than that the JavaScript Tutorial page on "The Definitive Guide to the JavaScript Promises" is a lot more detailed but gives a complete picture:

- <https://www.javascripttutorial.net/es6/javascript-promises/>

Adding to the Harvest Report - Spike 2:

Now that we know a little bit about Axios API requests and JavaScript Promises, let's request the information about the farm from the farmOS API and use it to fill in the data in the report.

15. Add a call to the `axios.get` function to the handler that is invoked when the "Generate Report" button is clicked. This call to `get` should:

- Request data from the `/farm.json` endpoint.
- Place the following lines in the `then` (i.e. at point A).
 - `console.log("Got Response")`
 - `console.log(response)`
- Place the following lines in the `catch` (i.e. at point B).
 - `console.log("Error Occurred")`
 - `console.log(error)`
- Place the existing code in your click handler after the `catch` (i.e. at point C).

16. Ensure that FarmData2 is running and that you are logged in. Open the DevTools and reload the FarmData2 page. Click on the "Generate Report" button.

a. What appears in the DevTools console?

b. Explore the response object that is printed in the console. What *property* of that object contains the farm name username and language information?

17. Replace the endpoint in the call to `axios.get` with `/fern.json` (note the misspelling) and reload the page. What output appears in the console when you click the “Generate Report” button now? Why?

18. Fix the mistake we introduced in question #17. Then edit your click handler so that the farm name, user and language are populated from the response that was returned instead of being hard coded. Hints: (1) Think about when you know the response has been received. That is where you need to set the Vue data values. (2) The object representing the response is named `response`, use dot notation to access the appropriate properties (see #16b). The name of the farm can be accessed with `response.data.name`. (3) Try logging out and back in as a different user to check that the user value changes.

19. Commit your changes to your feature branch with a meaningful commit message and push it to your origin. Recall that this also updates your Draft Pull Request.

The FarmOSAPI Library:

We will now turn to populating the Crop and Area dropdowns with the actual list of crops and areas fetched from the FarmData2 database instead of hard coding them. To get these lists we’ll need to make API requests to the farmOS API to ask for them. However, instead of using Axios calls directly we will now begin using some of the convenience functions that FarmData2 defines for us.

20. Open the following file from your FarmData2 repository in a web browser:

- `farmdata2_modules/fd2_tabs/resources/doc/index.html`

The list on the right of that page includes all of the functions provided by the FarmOSAPI library. What are the names of the four functions that deal with crops and areas?

21. Click on the `getIDToCropMap` function to see more detailed documentation for this function. What does the documentation say that this function does? Hint: It’s the first sentence of the description.



JavaScript Maps:

All of the functions you identified in question #20, and a number of others get a Map. A Map in JavaScript is a data structure that maps a *key* to an associated *value*. So, it is the same thing as a Dictionary in Python or a Map/HashMap in Java. These maps are particularly useful in FarmData2 as nearly everything has both a name and an id. For example, each user, crop, area, log and asset has both a name and an id. Often when working with data in FarmData2 we will have an id but need the associated name, or vice versa. So, having easy access to these maps simplifies that process.

22. Revisit the documentation for the `getIDToCropMap` function. In the Map that is created by this function:

a. What is the key in the Map created by this function?

b. What is the value in the Map created by this function?

c. Would this Map be most useful for converting crop IDs to crop names? Or crop names to crop IDs? Why?

Adding to the Harvest Report - Spike 3:

Our goal in this spike is to get a list of crops from the farmOS API and use them to populate the Crop dropdown. Later you'll adapt what you do here to populate the Area dropdown.

When adding the farm information (e.g. name, user, language) we made the API request in the click handler for the "Generate Report" button. Thus, the user action of clicking the button was what initiated the API request. In the case of the Crops (and later the Areas) we want the API request to occur as soon as the page is loaded, without requiring any action by the user. Vue provides the `created ()` *lifecycle hook* for this purpose.

The `created ()` lifecycle hook is a function, that when added to a Vue instance, is invoked immediately after the Vue instance is created. Thus, the `created ()` lifecycle hook provides an opportunity to execute code automatically when the page is loaded.



23. Add the following `created()` function to the Vue instance just below your data property. Hint: Don't forget the comma after the closing `}` on our data property!

```
created() {  
  console.log("HarvestReport created!")  
},
```

What happens when you reload the page now?

24. Now you know how to run code when a Vue instance is created (i.e. when the page containing it is loaded). Let's change what happens in `created()` so that it fetches the Map from crop name to crop ID. Use the example from the FarmOSAPI documentation for the `getIDToCropMap` function. Use a `console.log` statement in the `then` clause to print the Map object to the console. When this works the console should display the Map. Clicking on the little triangles you can expand the sections and see the entries in the map, which show the keys and the values, as seen below:

```
▼ Map(111) ⓘ  
  ▼ [[Entries]]  
    ▼ [0 .. 99]  
      ▶ 0: {"102" => "ARUGULA"}  
      ▶ 1: {"142" => "ASPARAGUS"}  
      ▶ 2: {"129" => "BEAN"}  
      ▶ 3: {"130" => "BEAN-DRY"}
```

Open the "Entries" of the Map object to answer the following questions:

a. What is the ID of BROCCOLI?

b. Which crop has the ID 105?

25. As you saw in the last question, the Map contains the names of all of the crops as its values. So, what's left to do here is to use those values to populate the Crop dropdown.

a. Once you have a Map, there is a relatively simple one-line JavaScript statement that will get all of the values from a Map as an array. Use your favorite search engine to find this

statement and then modify your `created ()` function so that the Crop dropdown will contain all of the crops.

b. Optional: In some browsers the list of crops will appear in alphabetical order, only because that is the order that the `getIDToCropMap` placed them into the Map. But the MDN Documentation describing the “Key Order” for the Map class says that “it's best not to rely on property order.”

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map

So as not to rely on the order of the key properties and thus to ensure that the crops are always alphabetical the array should be sorted. Use your favorite search engine to find out how to sort an array in JavaScript. Use what you find in the `created ()` function so that the Crop dropdown will always be sorted alphabetically.

26. Optional: It is good practice to ensure that something happens if an error occurs during an API request. Otherwise debugging can be extremely difficult. So, if you haven't already, add a `console.log` statement to the `catch` so that if an error occurs the error message will be printed to the console.

27. Commit your changes to your feature branch with a meaningful commit message and push it to your origin. Recall that this also updates your Draft Pull Request.

28. You will notice that it takes a few moments after the page loads for the Crops dropdown to populate with all of the crops. Briefly explain why you think that delay occurs? Hint: Think about the API request and the Promise that is being used.

Adding to the Harvest Report - Spike 4:

29. Now that you have populated the Crops dropdown from the farmOS API you need to populate the Areas dropdown. Adapt what you have done for Crops code so that the Areas dropdown is also populated when the page is loaded. Hint: Refer to the documentation for FarmOSAPI to find the right function to use and to see an example of its use.

30. Commit your changes to your feature branch with a meaningful commit message and push it to your origin. Recall that this also updates your Draft Pull Request.



Optional: To help us improve and scope these activities for future semesters please consider providing the following feedback.

a. Approximately how much time did you spend on this activity outside of class time?

b. Please comment on any particular challenges you faced in completing this activity.

