05 - 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 (02). 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 (03). 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 (04).

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 web 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 API provided by farmOS to fetch and incorporate live data from the FarmData2 sample database into your report.

It is not required viewing, 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 authenticate 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:

A web 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 a farmOS server will provide some basic information about the farm.

1. Ensure that your FarmData2 developer environment is up and running and that you have logged into FarmData2 as manager1 (or 2) or worker1 (or 2,3,4,5). Then start Firefox and enter the following URL to make a request to the `/farm.json` endpoint on the `fd2_farmdata2` server.

```
http://fd2_farmdata2/farm.json
```

When the content loads, click on the "Raw Data" view. 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 the FarmData2 developer environment.

Copy and paste the first two lines of the response that you received here. These lines should show the name of the farm and the name of the user that you are logged in as.

The Hoppscotch API Tool:

While you can interact with APIs through your browser, as you just did, it's not very pretty or easy to read. A few of the Firefox features will make this information a little easier to read (e.g. the "JSON" or "Pretty Print" views). However, most developers will use other specialized tools that make it much easier and more convenient to interact with APIs during testing and development.

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.

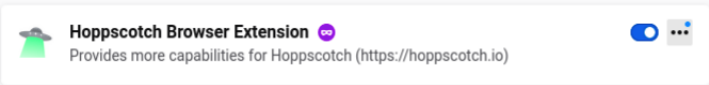
2. Using the Firefox browser in the FarmData2 developer environment, visit the Hoppscotch page:

- https://hoppscotch.io.

3. You'll learn more about Hoppscotch and APIs in a minute. But to make Hoppscotch fully functional 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 "Firefox"
    - This will take you to the to the Firefox Add-Ons site where you can install the extension.
- Click the "Add to Firefox " button on the page that appears to install the extension.
    - You should now see a little Hoppscotch icon ( ) at the top right of the Firefox window.
- Return to the Hoppscotch tab and reload the page.

4. Find "Add-ons and themes" on the Hamburger menu ( ) in Firefox. Then click the "Plug In" icon ( ) on the left. You should then be able to confirm that the Hoppscotch Browser Extension is installed and enabled as shown below:



If you do not see the Hoppscotch Browser Extension installed and enabled, return to #3 and try again. You will not be able to proceed without having this extension installed.

5. You can now use Hoppscotch to make a request to the `/farm.json` endpoint. Enter the URL below into the text field next to the word "GET" in Hoppscotach and click the "Send" button:

```
http://fd2_farmdata2/farm.json
```

When the `fd2_farmdata2` 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 as a worker as a manager, that you correctly installed the Hoppscotch browser extension and that you correctly entered the endpoint URL above.

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

    a. The text next to "Status" 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 `fd2_farmdata2` server return in response to your request to the `/farm.json` endpoint?

b. The size (in bytes) of the response that was received is also displayed next to the "Size" label. What was the size of the response returned in response to your request to the `/farm.json` endpoint?

API Requests and JSON:

Most of the information we are really interested in is in the "Response Body." This is the same information you saw in Firefox earlier, but it is now displayed in a nicely formatted and easier to read way.

The responses received from the farmOS API will be in *JavaScript Object Notation (JSON)*. This JSON format should look reasonably familiar because it is the same format that we used in activity 04 to define JavaScript objects.

It is not required, but if you'd like to review or learn more about JSON the links below will be helpful:

- *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 generated the response?

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

d. Give a reference using *dot notation* to the value that indicates the name of the user that made the request. Hint: Use dot notation to specify the sequence of property names that must be followed to get to the user’s name (i.e. the name that you logged into FarmData2 with).

e. Give a reference using dot notation to the value that indicates the name of the language that is being used. Hint: You will need to use three property names separated by dots for this one.

Synchronizing with the Upstream:

8. Some time has passed since you created your fork and clone of the upstream FD2School-FarmData2 repository. It is possible that there have been updates to the upstream since you did so. So, as when working on any fork and clone it is important to synchronize your main branch with the upstream so that you have all of the recent changes.

a. Synchronize your local and origin FarmData2 repositories and your feature branch with the upstream. The steps you will need to do this include are:

1. Switch to the `main` branch.
2. Pull the `main` branch from the upstream.
3. Push the `main` branch to your origin.

If you don’t remember the commands for this, you can refer back to the previous activity where you will have listed them.

b. Now because your `main` branch has been updated, you will need to merge those changes into the feature branch that you created in the prior activity. To merge the `main` branch into your feature branch:

1. Switch to your feature branch from the prior assignment (`<name>-04-HTML`).
2. Merge the `main` branch into your feature branch.

3. Resolve any conflicts that arise (If you are just doing the FD2School activities there should not be any).

If you don't remember the commands for this, you can refer back to the previous activity where you will have listed them.

c. Your work on this assignment builds from what you did in the prior assignment. So you now need to:

1. Switch to your prior feature branch `<name>-04-Vue2` (if you are not there already).
2. Create a new feature branch named `<name>-05-API` from your prior feature branch.
3. Switch to your new feature branch.

You will do all of your work for this activity in this feature branch.

If you don't remember the commands for this, you can refer back to the previous activity where you will have listed them.

Adding a New FarmData2 School Sub-tab:

9. Make sure you have your feature branch checked out and then add another new sub-tab named **API** to the FD2 School tab. Have the contents of this new tab be contained in the directory `api` with the page content provided by the file `api.html`. Make a copy of your `vue2.html` file into a file named `api/api.html`. Don't forget to clear the Drupal cache when you are done. The result should be that you now have four sub-tabs in the FD2 School tab: HTML, Vue1, Vue2 and API. For now, your Vue2 and API tabs will be exactly the same. You'll be modifying and extending the API tab throughout this activity.

10. Commit your changes to your feature branch with a meaningful commit message that describes what you have done and push it to your origin.

11. On GitHub create a *Draft Pull Request* for your new feature branch to the upstream FD2School-FarmData2 repository. Be sure to link your PR to the issue for this activity by including a line like the following in the body of your PR:

Addresses #??

Replace the ?? by the issue number in the FD2School-FarmData2 issue tracker for this assignment.

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 these values from the `/farm.json` endpoint and display the data it provides in the page.

It is going to take us a few steps to get there. The following activates will guide you through a way to make these changes incrementally. Working incrementally is always a good idea. It allows you to test each change you make to ensure it is working prior to going on. By working in this way, you'll know that if something breaks or stops working, you'll have a pretty good idea that it came from the small incremental change that you just made. This can save you a lot of time in debugging.

12. As a first step, modify your `api.html` file so that the values for "Farm", "User" and "Language" in the report are bound to properties in the `Vue` instance. Set the initial values of your new Vue properties to be empty strings.

- Hint: Add properties for these values to the `data` object in the `Vue` instance and then use the double mustache notation to render them in the report.
- Hint2: You can check that what you have done by examining and modifying your new properties using the Vue DevTools.

13. 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.

14. Add code to the click handler for the "Generate Report" button that sets the values of the properties you created to "Sample Farm", "manager1" and "English".

- Hint: You added a function to the `methods` property in your `Vue` instance in the previous homework and connected it to the `@click` handler for the "Generate Report" button. Modify that function!

Now when the button is clicked, it will set the values in the `Vue` instance and the Vue data binding will cause them to be rendered in the page. The next step (in just a bit) will be to get the these values from an API request instead of hard coding them in your function.

15. 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. Once you are fully immersed in FarmData2 you probably won't need to use Axios directly because FarmData2 provides convenience methods that hide some of the details. However, knowing how Axios, and more generally *JavaScript Promises* work, will be helpful in the future. So, to get a feel for these things, we will make our first API request directly using Axios instead. After that, you'll learn more about some of the FarmData2 convenience methods and how to use

them. You'll see later that the coding pattern for making an Axios request is very similar to the coding pattern that you'll use with the FarmData2 convenience methods. So, what you learn here will be helpful to you as you continue on with FarmData2 development.

Everything you need to know about Axios should be in this assignment, 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 code pattern for an API request using Axios will be:

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

16. 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.

The Promise allows the request to occur *asynchronously.* This means, that while the request is being transmitted, processed, rather than just waiting idly for the response the program will continue executing the code at point C. Then later, when the response is actually received the code in block A (in the `then` function) will execute. Or, if an error occurs with the request (e.g. the API endpoint doesn't exist, or the server is down) then the code in block B (in the `catch` function) will execute.

In JavaScript terminology we say that the Promise is *resolved* when the requested action completes and the code in the `then` function executes. If there is an error, and the code in the `catch` function executes, we say the Promise is *rejected*.

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

There is a good bit more to JavaScript Promises, but that should be sufficient to get you going. Thus, 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 provides a definitive guide to *JavaScript Promises* that is a lot more detailed and gives a very complete picture of what they are and how they work:

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

Adding to the Harvest Report - Spike 2:

Now that we know a little bit about making Axios API requests and about JavaScript Promises, we can request the information about the farm (name, user, language) from the farmOS API and use it to fill in the data in the report.

18. Add the coding pattern for Axios requests as shown above to the JavaScript function that handles clicks on the "Generate Report" button.

Fill in the coding pattern so that:

- It requests data from the `/farm.json` endpoint.
    - Note: It is not necessary to include the `http://fd2_farmdata2` part here. The endpoint should just be `/farm.json`.
- Block A contains the lines:
    - `console.log("Got Response")`
    - `console.log(response)`
- Block B contains the lines:
    - `console.log("Error Occurred")`
    - `console.log(error)`
- Block C contains all of the prior code that you had in your function.

Note that the console.log statements that you added will print information to the Console tab in the DevTools in Firefox. Thus, console.log provides is a very useful tool for incremental

development and testing. For example, here we don't yet attempt to process the response. Instead, we just print a message and the response (or error) to the console. This allows us to be sure we have received the response correctly before trying to use it. Then, once we know we are receiving the response correctly we will go on to add code that processes the response.

19. Reload your API sub-tab in Firefox and open the Console tab in the DevTools. Then, click on the "Generate Report" button. As you know, this will run the JavaScript function that you have connected to the button using `@click`. That code now makes a request to the `/farm.json` endpoint and processes the response (or error) that it receives.

a. Look at the Console tab in the DevTools. You should see some indication that you have successfully received a response. How can you tell from the Console that your request was successful?

b. Your code printed the response `Object` to the Console. The Console allows you to explore this `Object` by clicking on the little triangles (▸). When you open the response `Object` you should see its properties (`config`, `data`, `headers`, `request`). Which of those properties contains the information that we are interested in for our page (i.e. the name of the farm, the username and the language)?

20. Now let's check to see that our error handler works correctly too. Replace the endpoint in the call to `axios.get` with `/fern.json`. Note that the farmOS API does not have an endpoint named `fern.json` so this should generate an error. Reload the page and click the "Generate Report" button. How can you tell from the Console that your request resulted in an error?

21. Fix the mistake you introduced in question #20 and ensure that you get a successful response again.

22. Edit your click handler so that the farm name, user name and language properties have their values set from the response that was returned instead of being hard coded.

- Hint: Modify the code in Block A of the coding pattern!
- Hint2: The information we want is in the `data` property of the response `Object` (see #19b). So, to get the name of the farm you would use `response.data.name`. The

user name and language properties can be accessed similarly, but they require a few more dots and property names. You can use the triangles in the printed object to find the right sequence of properties and dots. Sometimes it is easier to find the property you want in Hoppscotch. If you do that, just preface the property reference that you find with `response.data` when you use it in your code.

- Hint3: To fully test your code you should also log out of FarmData2 and log back in as a different user (e.g. manager2, or worker4) to ensure that the username changes.

23. Once you verify that your request is working properly and that the values for the farm name, username and language are displayed in the page, remove the `console.log` statements in the `then` (block A). They were there for incremental development and testing and should not be a part of the final product.

24. 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. Each of these convenience functions uses Axios calls internally to request the data from the farmOS API. They just make it easier for us.

25. All of the FarmData2 convenience functions have documentation that we can use to learn about them. Before we can look at this documentation, we need to generate it. The instructions for generating the documentation are contained in the "Generating Documentation" section of the `README.md` file in the `farmdata2` directory.

Generate the documentation for the FarmData2. What command did you use?

26. The same `README.md` file tells you how to access the documentation using your browser. Open the documentation in Firefox.

The list on the right of the page that you open lists all of the functions provided by the FarmOSAPI library under the "Global" heading. What are the names of the four functions that deal with crops and areas?

27. Click on the `getIDToCropMap` function to see more detailed documentation for this function.

a. What does the documentation say that this function does? Hint: It's the first sentence of the description.

b. What does this function return? Hint: See the text under the "Returns" heading.

JavaScript Maps:

The function you looked at in question #27, and a number of others, return a Promise that yields a Map when it resolves.

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 number. For example, each crop has both a name and an id number called the `tid`. Often when working with data in FarmData2 we will have an id number but need the associated name, or vice versa. These maps provide us with an easy way to translate between id numbers and names. For right now though we will use them an easy way to get a list of the crop names and area names.

28. Revisit the documentation for the `getIDToCropMap` 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 that appears in the API sub-tab. Later you'll adapt what you do here to populate the Area dropdown.

When adding the farm information (i.e. 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.

29. Add the following `created()` function to the `Vue` instance in your `api.html` page. Insert it at the top level in the `Vue` instance - i.e. at the same level as the `data` property and `methods` property. Do not place it inside the `data` property.

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

Reload your API sub-tab and confirm that your `created` lifecycle hook was executed. How did you confirm that the `created` lifecycle hook was executed?

30. Now you know how to run code when a `Vue` instance is created (i.e. when the page containing it is loaded). So let's change what happens in `created()` so that it fetches the `Map` from crop id to crop name and saves it so that we can use it.

a. Add a new variable to your Vue `data` that will eventually hold the map. Initialize that variable to an empty Map object (e.g. `idToCropMap: new Map()`).

b. The documentation for the `getIDToCropMap` includes an "Example" section that contains a code pattern for how to use this function. This pattern should look familiar because it is very similar to how we called `Axios.get` earlier. That is because just like `Axios.get`, the `getCropToIDMap` function returns a Promise.

Adapt the code pattern from the FarmOSAPI documentation for the `getIDToCropMap` function to request the map in your `created` lifecycle hook and assign `theMap` that is returned to the new variable that you added to your Vue data in step a. Don't forget the `this`!

c. It is good practice to ensure that something happens if an error occurs during an API request. Otherwise, if errors will occur silently and 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 an error message will be printed to the Console.

31. Now reload the page and use the Vue DevTools to inspect the Map that you saved into the `idToCropMap` property in the `data` of your `Vue` instance. You can click the little triangles to inspect the Map contents.

Sometimes the Vue DevTools do not pick up changes to the data property right away. If you do not see the idToCropMap property in the data try clicking the reload icon (↻) in the top right of the Vue DevTools. If that doesn't work try closing the DevTools, reloading the page and then reopening the DevTools and the Vue DevTools tab.

Use the information in the Vue DevTools to answer the following questions:

a. What are the key and value that appear at entry 0 in the Map?

b. What is the key that maps to BOKCHOY? Hint: Just open a few more of the Map entries until you find BOKCHOY.

Vue Computed Properties:

It might be tempting now to just take the values from the ID-to-crop Map and use them to replace your hard-coded array of crops. While that would work, a more dynamic and the preferred way to do this in Vue is with a *computed property*.

Find and watch the *Computed Properties (3:08)* video in the free *Vue.js Fundamentals* course, then use the information in the video to answer the following questions.

- https://vueschool.io/courses/vuejs-fundamentals

32. What does the video say that computed properties let us do? Hint: It's right at the start of the video.

33. What does the video say that a computed property should not do? Hint: Its around when the `reversedItems()` computed property is being added.

34. Computed properties often seem a lot like methods, but they have a different purpose. The video gives a good rule of thumb to help with the decision of whether to use a method or a computed property. Hint: This is toward the end after the example code is complete.

a. When does the video say that you should use a method?

b. When does the video say that you should use a computed property?

Adding to the Harvest Report - Spike 4:

The id to crop name Map contains the data we need to populate the crop dropdown. However, it needs to be transformed from a Map into an array of just the crop names in order to be used with the drop down.

Based on the previous question, using a computed property would be the right approach here. In the video the list of items in the shopping list was transformed by reversing it. Here, we'll do something similar to transform the Map into an array containing just the crop names (i.e. the values). You may find it helpful to match the following steps up to the steps taken in the video.

35. Add a `computed` property to your `Vue` instance just above the `methods` property (e.g. 0:25 in the video). Hint: Don't forget the commas between your `data` property and the `computed` property and also between your `computed` property and the `methods` property.

36. Add a function to the `computed` property that we will use to transform the id to crop Map into an array containing just the crop names. Be sure to pick a good descriptive name for the function. Because a computed property returns a thing rather than performing an action, it is common practice to use a noun rather than a verb for their name. For example, here you might use `cropNames` rather than `getCropNames`.

37. The function you added in the previous question will now need to create and return an array containing just the crop names (i.e. the values) from the Map. Given 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 computed property function so that it creates and returns an array of the crop names.

- Hint: The search string *javascript map values to array* gave pretty good results.

38. Now, change the `v-for` that generates the `<option>` elements for the Crop dropdown so that it uses your new computed property instead of the hard coded array of crops names. At the same time, remove the hard coded array of crop names from your `Vue` instance, as it is no longer needed.

39. Reload your API sub-tab in Firefox and confirm that the Crop dropdown now contains the full list of crops. There should be a lot of them!

40. 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 5:

41. **Optional Challenge:** Now that you have populated the Crops dropdown from the farmOS API you can also populate the Areas dropdown in a similar way. 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 convenience function to use and to see an example of its use.

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.

42. When your assignment is complete, convert your draft Pull Request to a regular 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.