LAB 10

JAVASCRIPT 3: APIS

What You Will Learn

- Accessing external APIs using JavaScript
- Using advanced language features

Approximate Time

The exercises in this lab should take approximately 90 minutes to complete.

Fundamentals of Web Development, 3rd Ed

Randy Connolly and Ricardo Hoar

Textbook by Pearson
http://www.funwebdev.com

Date Last Revised: Sept 28, 2018

ASYNCHRONOUS CALLS TO EXTERNAL APIS

PREPARING DIRECTORIES

1 If you haven't done so already, create a folder in your personal drive for all the labs for this book.

2 Copy the folder titled `lab10` to your course folder created in step one. This `lab10` folder could be provided by your instructor, or you could clone it from GitHub repo for this lab.

In this lab you will use JavaScript to access external APIs (also referred to as web services). This used to require using JQuery because the native approach was just too complicated. In recent years however, browsers now support the native `fetch` method which we will use here.

Exercise 10.1 — USING FETCH

1 Examine `lab10-ex01.html` and then edit js/`lab10-ex01.js` by adding the following code.

```
window.addEventListener('load', function() {

    // use fetch to request data from an API
    fetch('https://api.github.com/orgs/funwebdev-2nd-ed/repos')
       .then(function (response) {
            console.dir(response);
        });
});
```

Fetch returns a Promise, which is a way to handle asynchronous operations.

2 *Test in browser and then examine the console.*

Notice that the fetch returns a request with a status code and some other headers; the actual return data is hidden in the body and will require some additional work.

3 In your browser, request `https://api.github.com/orgs/funwebdev-2nd-ed/repos`

APIs (web services) simply return data (often JSON but could be XML or other format) via regular HTTP requests.

4 Modify the code as follows and test.

```
window.addEventListener('load', function() {

    // use fetch to request data from an API
    fetch('https://api.github.com/orgs/funwebdev-2nd-ed/repos')
       .then(function (response) {
           return response.json();
        })
        .then(function (data) {
           console.dir(data);
        });
});
```

Notice that the first promise handler now returns the JSON data using the `json()` method of the response. Another promise is then able to extract the data.

5 Modify the URL in the fetch so that it is incorrect and then test.

Not surprisingly, this generates a run-time error. Our fetch needs some additional work to handle errors.

6 Add the following code and test.

```
.then(function (data) {
   console.dir(data);
})
.catch(function (error) {
   console.log(error)
});
```

We still have our run-time error unfortunately. The problem is the second promise still executes regardless of the error.

7 Modify the code as follows and test.

```
.then(function (response) {
     if (response.ok) {
         return response.json();
     } else {
         return Promise.reject({
             status: response.status,
             statusText: response.statusText
           })
     }
})
```

8 Restore the fetch URL to the correct one and test.

Many contemporary JavaScript techniques make use of arrow functions, which is a more concise syntax for defining functions.

Exercise 10.2 — Arrow Functions

1 Examine `lab10-ex02.html` , run in browser, and then examine console.

You will be creating arrow versions of these functions.

2 Edit js/`lab10-ex02.js` by adding the following code and test.

```
/* now define arrow function equivalents */
var f1a = () => { return "function 1a" };
console.log( f1a() );
```

3 Add the following code and test.

```
var f2a = s => { return s };
console.log( f2a('hello f2a') );
```

Notice that you can optionally leave out the () for parameters.

4 Add the following code and test.

```
var f3a = (a,b) => { return "function 3a result=" + (a+b); };
console.log( f3a(4,5) );
```

5 Add the following code and test.

```
var f4a = (a,b) => {
    let s = "function 4a result=";
    let product = a*b;
    let msg = s + product;
    return msg;
};
console.log( f4a(6,7) );
```

6 Add the following code and test.

```
var f5a = (a,b,fn) => {
    let msg = f4(a,b) + fn();
    return msg;
}
console.log( f5a(7,8, () => { return " param f5a"}) );
```

Notice how the arrow syntax makes this last example much cleaner (once you get used to it) in comparison to the regular (non-arrow) version. Indeed, arrow functions are best suited for non-method functions, for instance, functions passed as parameters.

Test Your Knowledge #1

Examine `lab09-test01.html` and then open `lab09-test01.js` in your editor. Display the continents from the API (URL provided in source) in the empty `<ul>`. Be sure to use arrow syntax for the second promise and the catch.

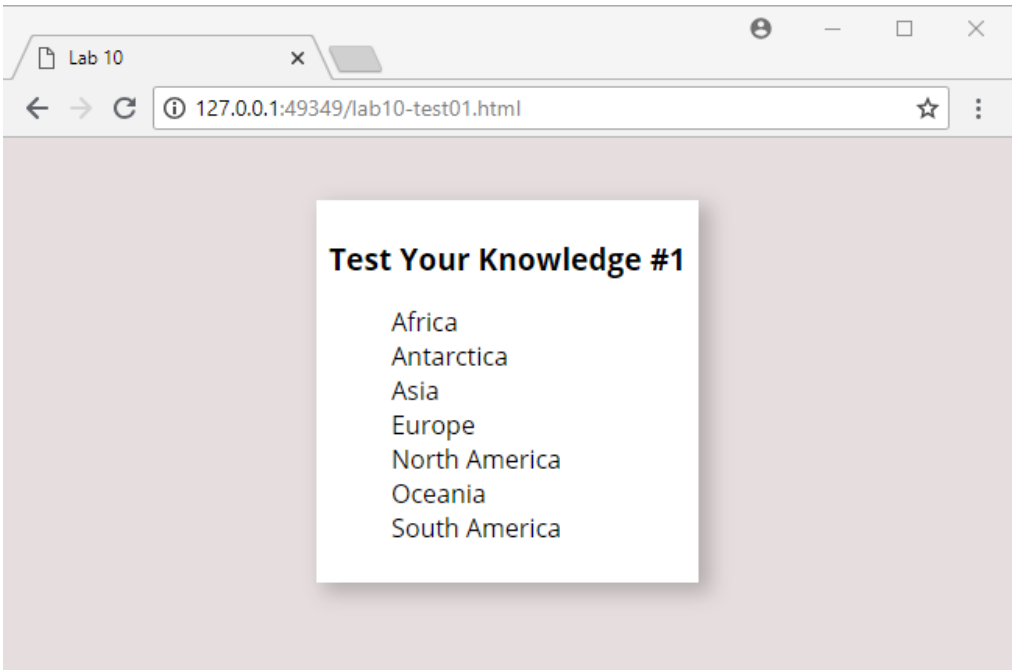

Figure 10.1 – Finished Test Your Knowledge #1

JavaScript provides many powerful iterative functions that take functions as parameters. These can be a bit confusing at first (especially since we will be

using arrow functions) but we will be using them frequently in future labs and chapters.

Exercise 10.3 — Useful Array Functions

1 Examine `lab10-ex03.html` and then edit js/`lab10-ex03.js` by adding the following code.

```
const sortedTitles = titles.sort( );
console.log(sortedTitles);
```

Since titles is a simple array, it can be easily sorted.

2 To sort an array of objects is more complicated. We have to create a compare function; to do so, add the following code (and then test).

```
const sortedPaintingsByYear = paintings.sort( function(a,b) {
    if (a.year < b.year)
        return -1;
    else if (a.year > b.year)
        return 1;
    else
        return 0;
} );
console.log(sortedPaintingsByYear);
```

We can make this even more concise.

3 Comment out the code from the previous step and enter the following, more concise version of the same using the ternary operator and arrow functions:

```
const sortedPaintingsByYear = paintings.sort( (a,b)  => {
    return a.year < b.year ? -1 : 1;
} );
```

4 To create a new array that is a subset of an existing array, you can use the `filter()` method. Add the following and test.

```
const nineteenthCentury = paintings.filter(painting =>
                  (painting.year >= 1800 && painting.year < 1900));
console.log(nineteenthCentury);
```

Notice that you supplied the conditional test that must be true for a painting to be included in the new array.

5 To find the first element in an array that matches a criteria, you can use the `find()` method. Add the following and test.

```
const manet = paintings.find( painting => painting.artist == "Manet");
console.log(manet);
```

Notice it finds only the first painting by Manet.

6 Finally, this function approach can also be used for iteration via the `forEach()` method of arrays. Add the following and test.

```
paintings.forEach( p => console.log(p.title + ' by ' + p.artist));
```

7 You can perform more powerful filtering and finding by using regular expressions. Add the following and test.

```
// first define regular expression pattern ('i' = case insensitive)
```

```
const re = new RegExp('with','i');
const regexEx = paintings.filter( p => p.title.match(re) );
console.log(regexEx);
```

This uses regular expressions to search for all paintings whose title contains the word 'with'.

8 Finally, a powerful new addition to JavaScript is the spread operator (... or the ellipse), which can be used to expand an array or object. This sounds pretty confusing, so let's look at some examples. Try entering the following and test.

```
// using the spread operator
const moreTitles = ["Balcony", "Sunflowers", ...titles];
console.log(moreTitles);
```

This has added the `titles` array to the end of the `moreTitles` array.

9 To make sense of this, let's implement that same functionality in the old fashioned way:

```
const moreTitlesOldFashioned = ["Balcony", "Sunflowers"];
for (let t of titles) {
    moreTitlesOldFashioned.push(t);
}
console.log(moreTitlesOldFashioned);
```

As you can see, using the spread operator simplifies our code (well, that is for you to decide; it certainly uses less code).

10 A common task with arrays is to create a new array based on data within another array. For instance, enter the following and test.

```
console.log('--- create new simple array from existing array of objects');
const artists1 = [];
for (let painting of paintings) {
    artists1.push(painting.artist);
}
console.log(artists1);
```

This creates a simple array of artist names based on artist field in your array of painting objects.

11 A more concise way of doing this is via the `map()` function, show below. Try adding this code and test.

```
const artists2 = paintings.map( function (painting) {
    return painting.artist;
});
console.log(artists2);
```

12 A more concise way of doing this is via arrow syntax, show below. Try adding this code and test.

```
const artists3 = paintings.map( painting => painting.artist );
console.log(artists3);
```

One of the early common uses of asynchronous API calls was to create autocomplete lists. As the user types in content in a text box, a dynamic list is displayed showing matches based on the letters being typed by the user. The next example provides an example.

EXERCISE 10.4 — CREATING AN AUTOCOMPLETE BOX

1 Examine `lab10-ex04.html` and then edit js/`lab10-ex04.js` by adding the following code after the endpoint url definition:

```
const endpoint = ...

// begin with an empty universities array
const universities = [];

// fetch from API will populate this empty array using spread operator
fetch(endpoint)
  .then(response => response.json())
  .then(data => universities.push(...data))
  .catch(error => console.error(error));
```

2 The next step is to define keyboard event handler for the input element:

```
// now set up keyboard event handlers
const searchBox = document.querySelector('.search');
const suggestions = document.querySelector('#filterList');
searchBox.addEventListener('keyup', displayMatches);
```

3 Create the `displayMatches()` function as follows:

```
// handler for keyboard input
function displayMatches() {
    // don't start matching until user has typed in two letters
    if (this.value.length >= 2) {
       const matches = findMatches(this.value, universities);

       // first remove all existing options from list
       suggestions.innerHTML = "";

       // now add current suggestions to <datalist>
       matches.forEach(univ => {
          var option = document.createElement('option');
          option.textContent = univ.name + ', ' + univ.city;
          suggestions.appendChild(option);
       });
    }
}
```

4 Finally create the `findMatches()` function as follows:

```
// uses filter and regular expression to create list of matching
universities
function findMatches(wordToMatch, universities) {
   return universities.filter(obj => {
       const regex = new RegExp(wordToMatch, 'gi');
       return obj.name.match(regex);
   });
}
```

5 Test in browser. Try typing in the first three letters of 'Alaska'. Notice how it matches any university that contains the letter 'ala' in the name. As you type more, the list should shrink.

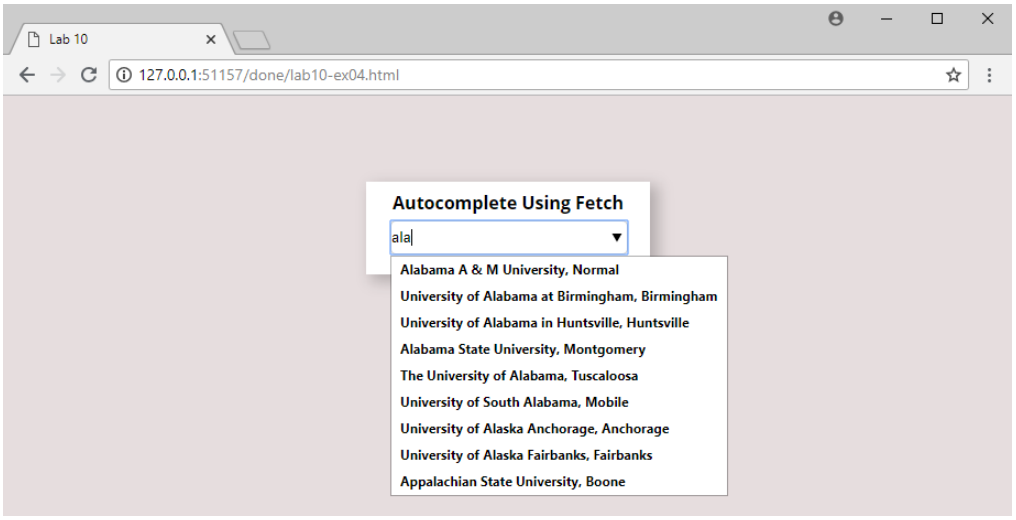

Figure 10.2 – Completed AutoComplete Box

Browser APIs

In the last section, you learned how to use the `fetch()` method to access data from external APIs. In this section, you will instead make use of the browser APIs (often also called Web APIs). These are APIs available to JavaScript developers that are provided by the browser.

In recent years, the amount of programmatic control available to the JavaScript developer has grown tremendously. You can now, for instance, retrieve location information, access synthesized voices, recognize and transcribe speech, and persist data content in the browser's own local storage.

In the next exercise you will work with one of these APIs, the Web Storage API, which provides mechanisms for preserving non-essential state across requests and even across sessions. Local Storage is simply a dictionary of strings that lasts until removed from the browser. Session storage is also a dictionary of strings but only lasts as long as the session.

EXERCISE 10.5 — USING THE STORAGE API

1 Some of the functionality has already been implemented in the starting files. Examine `lab10-ex05.html` in the browser and then examine in the editor. Try defining a color scheme by clicking on the color selectors and then clicking the Add to Scheme Collection button. This should add a series of appropriate colored boxes in the top part of the page. You can add multiple schemes if you wish (see Figure 10.3)

To allow this exercise to focus on the Storage API, most of the functionality on this page has already been implemented in JavaScript. This JavaScript makes use of several techniques covered earlier in this chapter.

2 Try clicking on one of the Test Color Scheme links. It should take you to a page that displays an alert that says "Colors not set yet". Click Ok and then click on the Return to Scheme Collection link. **Your color schemes should have disappeared.**

This shouldn't be a surprise: the page is using JavaScript to dynamically add content to the page; so if we leave the page, then the page reverts to its initial state. This exercise is about using local storage in the browser to preserve state.

3 Edit js/`lab10-ex05.js` by modifying the following code (don't test yet).

```
window.addEventListener('load', function () {
    // define a few variables referencing key interface elements
    const colorChoosers = document.querySelectorAll('input[type=color]');
    const colorLabels = document.querySelectorAll('fieldset span');
    const schemePreviews =
        document.querySelectorAll('.scheme-group .scheme');
    const schemeGroup = document.querySelector('article.scheme-group');

    // holds collection of user-created schemes ...
    let schemeCollection = retrieveStorage();
```

In the starting code the scheme collection is simply an empty array. In the next step we will modify this method.

4 Modify the following code (don't test until instructed later).

```
// retrieve from storage or return empty array if doesn't exist
function retrieveStorage() {
    return JSON.parse(localStorage.getItem('schemes')) || [];
}
```

As the comment indicates, this function will either return whatever string is stored in localstorage for the specified key (in this case the key is named 'schemes'). If the key value doesn't exist, then it returns an empty array instead.

The `JSON.parse()` method turns the JSON string (remember that Local Storage can only contain strings) into a JavaScript object.

5 Modify the following code.

```
// update storage with revised collection
function updateStorage() {
    localStorage.setItem('schemes', JSON.stringify(schemeCollection));
}
```

This code saves the array of schemes in local storage. Since Local Storage can only save strings, you have to use the `JSON.stringify()` method to convert the array of scheme objects into a JSON string.

6 Modify the following code.

```
// removes collection from storage
function removeStorage() {
    localStorage.removeItem('schemes');
}
```

Once something is added into local storage, it stays until user removes it (by clearing browser history) or by programmatically removing it as shown in this step.

7 Add the following code to the `setupAddSchemeHandler()` method:

```
// add scheme to collection array
schemeCollection.push(scheme);
// update storage with revised collection
updateStorage();
// tell scheme collection to update its display
updateSchemePreviews();
```

When the user clicks the Add to Scheme Collection button, then the method creates a new scheme object populated from the values in the color selectors. It then adds/pushes this new object to the scheme collection array. The new line we added calls your revised update method which adds (or replaces if it already exists) the collection to local storage and then redraws the preview sections.

8 Now test by repeating steps 1 and 2. The Test Color Scheme page, which reads the scheme collection from local storage, should show a larger preview of the colors in the scheme. When you return to the main page via the Return to Scheme Collection link, the previously constructed schemes should still be there.

9 Close the browser window or tab used for your testing. Restart the browser or create a new tab and navigate to the `lab10-ex05.html` page. **The schemes should still be there.**

10 Add the following code to the `setupRemoveAllHandler()` method:

```
// defines handler for Remove All Schemes button
function setupRemoveAllHandler() {
    document.querySelector('button#btnRemoveAll')
                    .addEventListener('click', (e) => {
        // empty the scheme collection
        schemeCollection = [];
        // update local storage and update preview display
        removeStorage();
        updateSchemePreviews();
    });
}
```

11 Test the Remove All Schemes button. Your schemes should now be gone. Try repeating step 9 (close browser and then re-request).

12 Change the `updateStorage()`, `retrieveStorage()`, and `removeStorage()` methods to use `sessionStorage` instead of `localStorage`. You will also have to change the one reference to `localStorage` in `lab10-ex05-tester.js` to `sessionStorage` as well.

With sessionStorage, the values only last as long as the browser session.

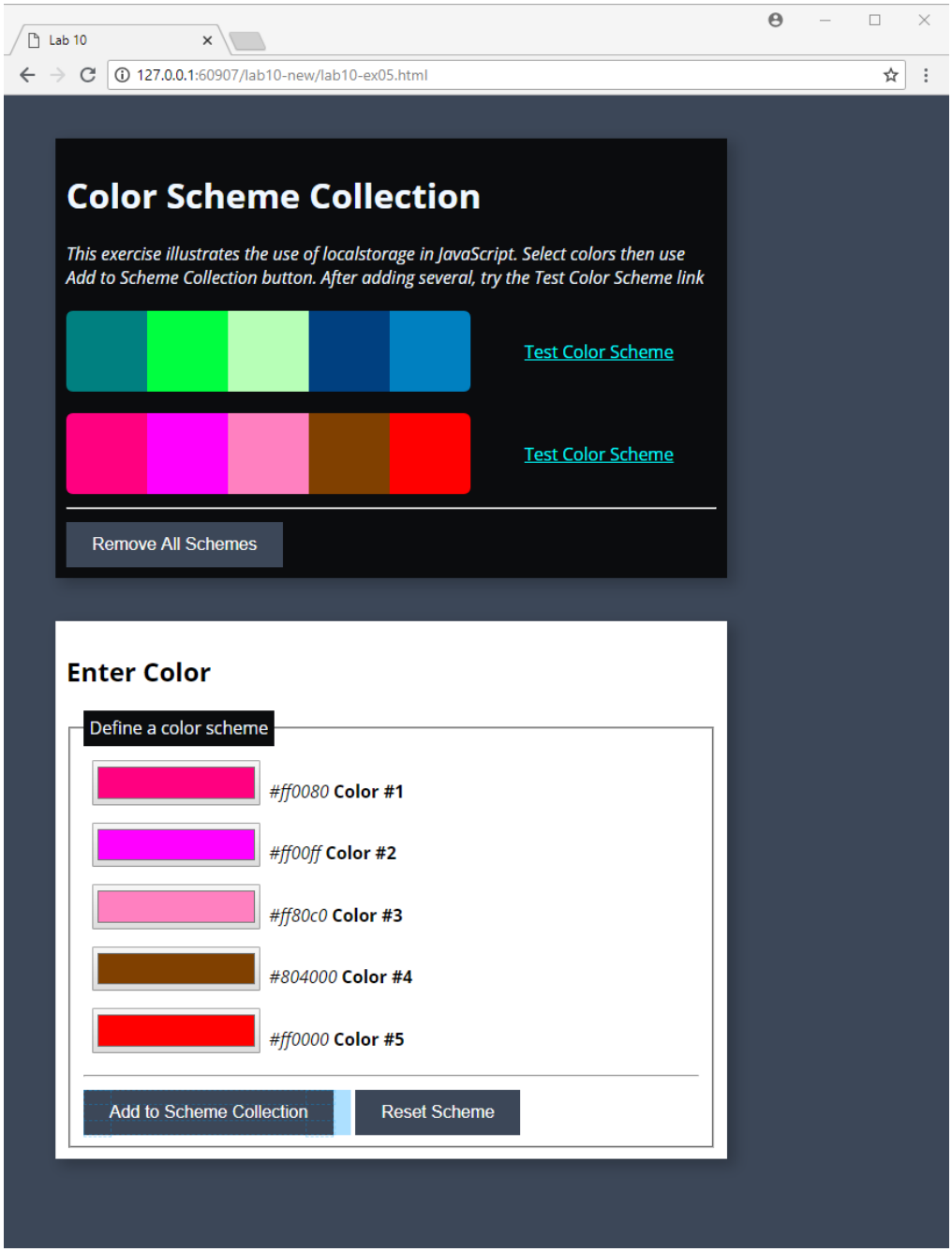

Figure 10.3 – Exercise 10.5

EXERCISE 10.6 — USING SPEECH SYNTHESIS

1 Examine `lab10-ex06.html` and then edit js/`lab10-ex06.js` by adding the following code. Save then test (you will need to have your device's sound turned on).

```
window.addEventListener('load', function () {
    const utterance = new SpeechSynthesisUtterance('Hey there buddy');
    speechSynthesis.speak(utterance);
});
```

The browser should speak the text passed to the SpeechSynthesisUtterance constructor.

2 Comment out these two lines you just added.

3 Add the following code, which displays the voices available to your browser and operating system. Test in browser and examine the console.

```
window.speechSynthesis.addEventListener('voiceschanged', function () {
    let voices = this.getVoices();
    voices.forEach( (voice) => { console.log(voice.name + " - " +
                                    voice.lang)});
});
```

4 Modify the event handler so it populates the Select A Voice <select> list with the available English voices and test.

```
window.speechSynthesis.addEventListener('voiceschanged', function () {
    let select = document.querySelector("#voices");
    // get array of all available voices for this computer+browser
    let voices = this.getVoices();
    // filter them so only have the english voices
    let englishVoices = voices.filter( voice =>
                                voice.lang.includes('en'));
    // populate the list only the first time in
    if (select.childElementCount == 1) {
        englishVoices.forEach( (voice) => {
            let opt = document.createElement("option");
            opt.setAttribute("value", voice.name)
            opt.appendChild( document.createTextNode(voice.name + ' ['
                                + voice.lang + ']'));
            select.appendChild(opt);
        });
    }
});
```

You should now have a list populated with available voices. You will likely see a different list of voices on a Mac running Safari, than on a Windows machine running Chrome.

5 Now add the handler for the Speak button as follows then test.

```
// now add event handler for speak buttonm
document.querySelector('#speak').addEventListener('click', (e) => {
    e.preventDefault();
    // get the text to say and the voice options from form
    let message = document.querySelector('textarea').value;
    let selectedVoice = document.querySelector('#voices').value;
    // create utterance and give it text to speak
    let utterance = new SpeechSynthesisUtterance(message);
    // set the speech options (voice, rate, pitch)
    utterance.voice = englishVoices.find(voice =>
                          voice.name === utterance);
    utterance.rate = document.querySelector('#rate').value;
    utterance.pitch = document.querySelector('#pitch').value;
    // all ready, make it speak
    window.speechSynthesis.speak(utterance);
});
```

It should work!

GOOGLE MAPS JAVASCRIPT API

Not every external API is simply a service for requesting data. The Google Maps JavaScript API is a prominent example of an API which consists almost entirely of an external `<script>` library; using the API thus involves using its objects' properties and methods.

Many external APIs require you to get some type of API key from the owners of the API. This key is then used with your requests.

The process of getting a Google Maps JavaScript API key changed in the summer 2018. It is possible that when you work on this lab, the process will have changed again. At the time of writing, you will need to first create a Google Cloud Platform account via https://cloud.google.com/free/.

EXERCISE 10.7 — WORKING WITH GOOGLE MAPS

1 You will need to join Google Cloud Platform and then get a Google API key for this project. The process for this can be found at:

`https://developers.google.com/maps/documentation/javascript/get-api-key`

It may take a few hours for you to receive this API key. The process of creating a key can be a bit complicated, and changed in the summer 2018.

2 Once you have a Google API key, then edit `lab10-ex07.html` by adding the following:

```
<h1>Populating a Google Map</h1>
<div id="map"></div>
```

Each Google Map must have an empty `<div>` element into which the map will be inserted by Google's

3 Continue editing the html by adding the following to the end of the body.

```
<script
  src="https://maps.googleapis.com/maps/api/js?key=YOUR_API_HERE&
callback=initMap"></script>
```

Notice that you will have to replace YOUR_API_HERE with your actual API, which you should find in your Google Console.

Notice as well the callback parameter in the URL. This is the name of the function to be called when the API is ready to draw your map. You will define this function below.

4 Edit `css/lab10-ex07.css` by adding the following:

```
#map {
    height: 700px;
}
```

Perhaps the most common mistake one makes with Google Maps is to forget to set a non-zero height for the element which will contain the map. If you forget this step, then the map will not appear!

5 Edit `js/lab10-ex07.js` by adding the following:

```
var map;

function initMap() {
    map = new google.maps.Map(document.getElementById('map'), {
      center: {lat: 41.89474, lng: 12.4839},
      zoom: 6
    });
}
```

The `initMap()` function is quite straightforward. It uses the `Map` constructor and passes it the node that will house the map, the latitude and longitude of the center point of the map, and the zoom setting (a value between 1-20).

6 Test. It should work.

7 Modify the map options as follows (case is important), then test.

```
function initMap() {
    map = new google.maps.Map(document.getElementById('map'), {
      center: {lat: 41.89474, lng: 12.4839},
      mapTypeId: 'satellite',
      zoom: 18
    });
}
```

8 Add the following call to the same method after creating the map:

```
map.setTilt(45);
```

9 Return the `initMap()` function back to how it was in Step 5. Then add the following method:

```
function createMarker(map, latitude, longitude, city) {
    let imageLatLong = {lat:  latitude, lng: longitude };
    let marker = new google.maps.Marker({
        position: imageLatLong,
        title: city,
        map: map
    });
}
```

This function will be used to add markers to our map.

10 Now add a call to this function in `initMap()` after creating the map:

```
createMarker( map, 41.89474, 12.4839, "Rome" );
```

11 Comment out this call. You will now get a list of cities from an external API and display a marker on the map for each one. Your starting code has a URL assigned to the variable called `endpoint`. Copy the URL in the string and view in browser to see the structure of the JSON data returned by the API.

12 Add the following code after the endpoint declaration and test.

```
const endpoint = ...

// to simplify no error handling for initial response
fetch(endpoint)
    .then( response => response.json() )
    .then( (data) => {
       // data received, now add cities to map
       data.forEach( (city) => {
           createMarker( map, city.Latitude, city.Longitude,
                         city.AsciiName );
       });
    })
    .catch( error => console.log(error) );
```

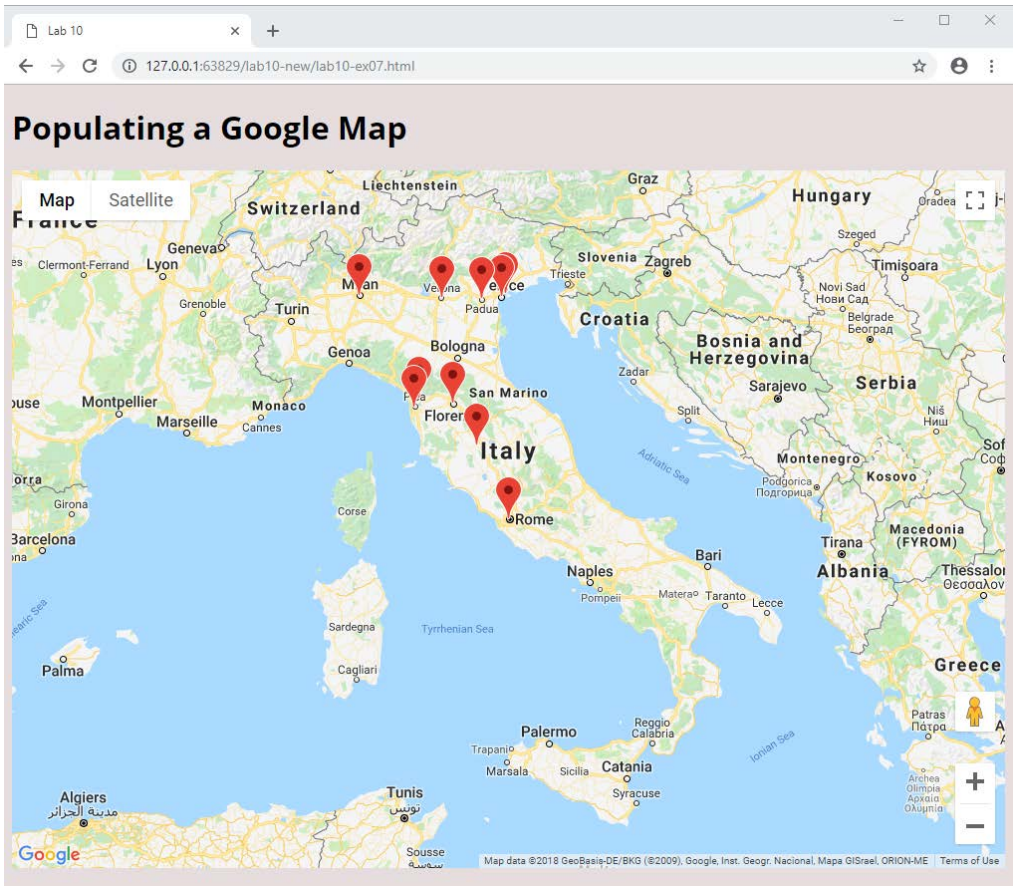

Figure 10.4 – Exercise 10.6

Note: Be careful with your Google Map API key! If others gained access to it and used it for their own site, you potentially would be financially responsible for their usage.

TEST YOUR KNOWLEDGE #2

In this exercise, you will use `fetch()` to retrieve a JSON list of galleries: for each gallery, there is information about it, its location, and several of its famous paintings. You have been provided with the starting files and some initial CSS but you will have to do the rest.

1. In area B (Figure 10.5), you will need to display a list of galleries from the fetched data. When the user clicks on a gallery name in B, see steps 2, 3.

   We recommend creating a function that is passed the array of galleries and which then does the following tasks:

   Loop through the galleries and add a `<li>` element to `<ul id=galleryList>` whose `textContent` is the `nameEn` property of each gallery object

   Loop through the newly created `<li>` elements and assign the same click event handler to each of them. This handler should change the visibility of the A and C sections, and display the gallery details (step 2), display list of paintings (step 4), and display the map showing the gallery (step 3). Since each of these requires multiple lines of programming, each of these steps should be encapsulated in its own function.

2. In area A, display the gallery name, link, and its location (city, address, country) in the provided elements.

3. In area D, display a google map showing the gallery in satellite view, zoom level 18. The latitude and longitude of the gallery can be found in the `location` object.

4. In area C, display a list of paintings in that gallery. When you click on the painting, use the speech synthesizer to say the text in the description field.

The final result should look similar to that shown in Figure 10.5.

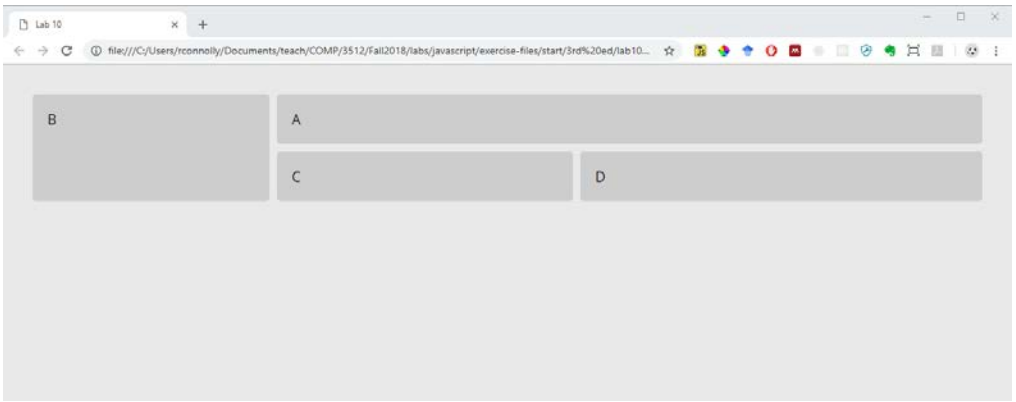

Figure 10.5 – Layout of lab10-test02 at start

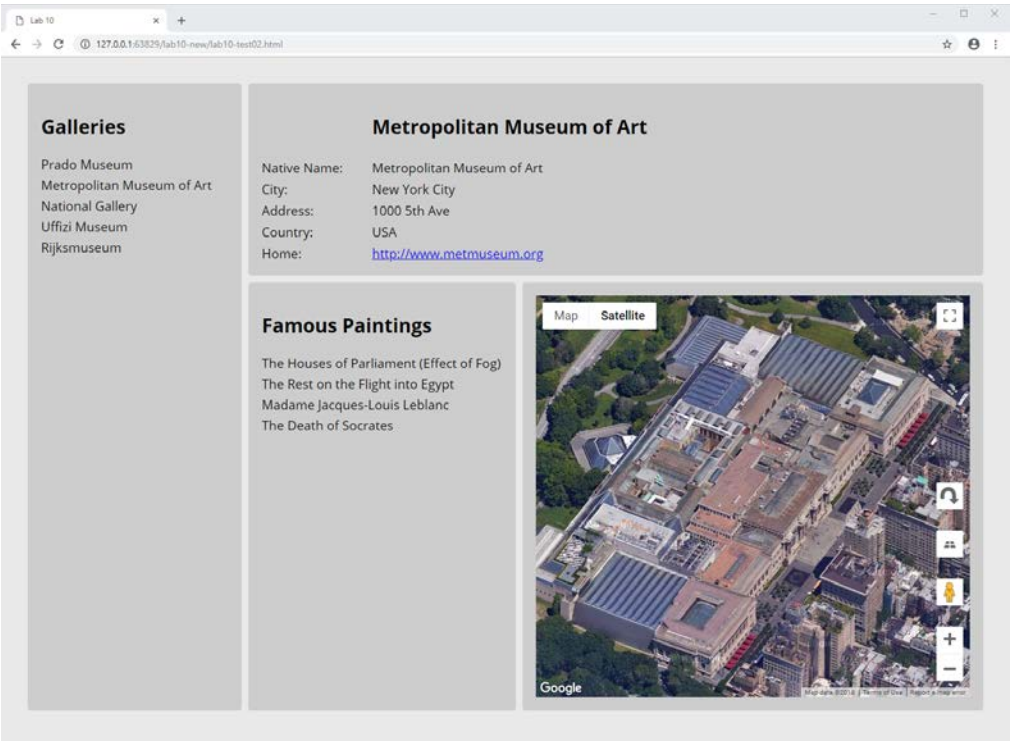

Figure 10.6 – Finished lab10-test02