LAB 10

JavaScript 3: APIs

What You Will Learn

- Using additional language features of JavaScript
- 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: October 2, 2020

Array Functions

Preparing Directories

1 Copy the folder titled `lab10` to your working area. This `lab10` folder could be provided by your instructor, or you could clone it from GitHub repo for this lab.

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 10.1 — Useful Array Functions

1 Examine `lab10-ex01.html` and then edit `js/lab10-ex01.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 paintings.

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);
```

Test Your Knowledge #1

Using the files `lab10-test01.html` and `lab10-test01.js`, solve the following code problems two ways: first using both regular loops and second using the appropriate array function. Assume your data is the `stocks` array shown below :

```
const stocks = [
  {symbol: "AMZN", name: "Amazon", price: 23.67, units: 59},
  {symbol: "AMT", name: "American Tower", price: 11.22, units: 10},
  {symbol: "CAT", name: "Caterpillar Inc", price: 9.00, units: 100},
  {symbol: "APPL", name: "Amazon", price: 234.00, units: 59},
  {symbol: "AWK", name: "American Water", price: 100.00, units: 10}
];
```

1 Add a new function property named `total` to each symbol object which is equal to `price * units` via a regular for loop and then using the `forEach()` array function. This will require using a function expression.

2 Find the first element whose symbol name is "CAT" (using loop and then using `find()` function).

3 Create a new array that contains stocks whose price is between $0 and $20 (using loop and then using `filter()` function).

4 Create a new array of strings contains `<li>` elements containing the stock `name` property (using loop and then using `map()` function).

5 Sort the array of stocks on symbol using the `sort()` function.

Prototypes, Classes, and Modules

Prototypes are more memory efficient mechanism for adding functions to objects. Instead of adding functions to each object instance, you can add the function once to its `prototype` property.

Exercise 10.2 — Using Prototypes and Classes

1 Edit `js/lab10-ex02.js` by adding the following:

```
// simple luminance (brightness) calculation
Color.prototype.luminance = function () {
   let r = this.rgb[0];
   let g = this.rgb[1];
   let b = this.rgb[2];
   return 0.2126*r + 0.7152*g + 0.0722*b;
}
```

2 Use this function by adding the following at the bottom of your code:

```
// add loop here
colors.forEach( (c) => {
   console.log(c.name + ',' + c.luminance());
});
```

3 Test by viewing `lab10-ex02.html` in browser.

4 Comment out the `Color` function constructor and the `luminance` prototype and replace with the following.

```
class Color {
   constructor(name, hex, rgb) {
      this.name = name;
      this.hex = hex;
      this.rgb = rgb;
   }

   luminance() {
      let r = this.rgb[0];
      let g = this.rgb[1];
      let b = this.rgb[2];
      return 0.2126*r + 0.7152*g + 0.0722*b;
   }
}
```

5 Test. It should work exactly the same.

Classes are just syntactical sugar, meaning that your Color class is simply an alternate way of creating a function constructor with prototype functions.

Exercise 10.3 — Using Modules

1 Edit `js/color.js` and examine. It contains code from previous exercise. Add the following.

```
const getSampleColors = () => sampleColors;

export { Color, getSampleColors };
```

2 Edit `js/lab10-ex03.js` by adding the following:

```
// import required function from your module
import { getSampleColors }  from "./color.js";
// use that function
getSampleColors().forEach( (c) => {
   console.log(c.name + ',' + c.luminance());
});
```

3 Edit `js/lab10-ex03.html` by adding the following. Test.

```
<head>
   <meta charset="utf-8"/>
    <title>Lab 10</title>
    <script src="js/color.js" type="module"></script>
    <script src="js/lab10-ex03.js" type="module"></script>
</head>
```

Notice that modules can only be used within a module. Thus both the module color.js and the file using the module (lab10-ex03.js) must be marked with type=module.

Test Your Knowledge #2

1 You will be modifying a file named `gallery.js`. This is going to be a module that will be used in your other files. It already has some sample data in it.

2 In this module, create a JavaScript class named `GalleryItem` that represents a gallery list item. Its constructor should take two arguments: the gallery name and the gallery id.

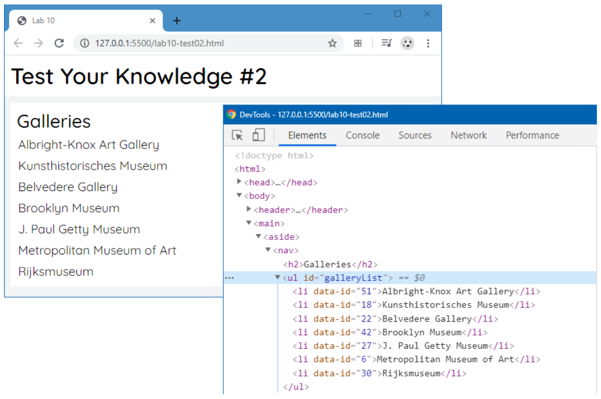
3 Add a method/function to the class named `render()` that returns a DOM element that represents a `<li>` element. The `textContent` of the element should be the gallery name. Add an attribute named `data-id` which is set to the gallery id.

4 Add a function to the module named `getSampleGalleries()` which returns the `galleries` array. Export both `GalleryItem` and `getSampleGalleries` at the end of the module.

5 Modify the file `lab10-test02.js` so that it imports `getSampleGalleries` from the gallery module. Use the `getSampleGalleries()` function to retrieve the sample data.

6 Loop through the sample data. For each `GalleryItem`, add the element returned from its `render()` method to the `<ul>` list using `appendChild()`.

7 Add in the necessary `<script>` tags to `lab10-test02.html`.The result should look similar to that shown in Figure 10.1.


Lab 10
127.0.0.1:5500/lab10-test02.html
DevTools - 127.0.0.1:5500/lab10-test02.html
Elements
Console
Sources
Network
Performance

Test Your Knowledge #2

Galleries

Albright-Knox Art Gallery
Kunsthistorisches Museum
Belvedere Gallery
Brooklyn Museum
J. Paul Getty Museum
Metropolitan Museum of Art
Rijksmuseum

```
<!doctype html>
<html>
▸<head>…</head>
▾<body>
  ▸<header>…</header>
  ▾<main>
    ▾<aside>
      ▾<nav>
          <h2>Galleries</h2>
        ▾<ul id="galleryList"> == $0
            <li data-id="51">Albright-Knox Art Gallery</li>
            <li data-id="18">Kunsthistorisches Museum</li>
            <li data-id="22">Belvedere Gallery</li>
            <li data-id="42">Brooklyn Museum</li>
            <li data-id="27">J. Paul Getty Museum</li>
            <li data-id="6">Metropolitan Museum of Art</li>
            <li data-id="30">Rijksmuseum</li>
          </ul>
```

Figure 10.1 – Finished Test Your Knowledge #2

Asynchronous Calls to External APIs

In this section 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 quite complicated. In recent years however, browsers now support the native `fetch` method which we will use here.

Exercise 10.4 — Using Fetch

1 Examine `https://api.github.com/orgs/funwebdev-2nd-ed/repos` in the browser.

Browser APIs are simply web pages that return JSON instead of HTML.

2 Examine `lab10-ex04.html` and then edit `js/lab10-ex04.js` by adding the following code within the supplied `DOMContentLoaded` event handler.

```
let url = 'https://api.github.com/orgs/funwebdev-2nd-ed/repos';

// use fetch to request data from an API
let foo = fetch(url);
console.dir(foo);
```

Fetch returns a Promise, which is a way to handle asynchronous operations. To access the data from the fetch, we have to wait until it is received, which we can do via then().

3 Modify the code as follows and test.

```
let foo = fetch(url);
let bar = foo.then( response => { console.dir(response) });
console.log(bar);
```

When fetch receives a response, it will call the callback function passed to then().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. Notice also that the then() function also returns a promise.

4 Modify the code as follows and test.

```
let url = 'https://api.github.com/orgs/funwebdev-2nd-ed/repos';
fetch(url)
    .then( response => response.json() )
    .then( data  => { console.dir(data); } );
console.log('after the fetch');
```

It is common to chain calls to then() together, thereby eliminating the need for additional variables.

The received data is now being received. Notice also that the data display happens after the other code has executed.

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.

```
fetch(url)
    .then( response => response.json() )
    .then( data  => { console.dir(data); } )
    .catch( err => { console.log(err) });
```

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

7 Modify the first `then()` as follows and test.

```
fetch(url)
    .then( 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.

In the next exercise, you will modify the DOM based on fetched data.

Exercise 10.5 — Fetching then Modifying the DOM

1 Examine the url specified within `lab10-ex05.js` in the browser.

2 Add the following code in `lab10-ex05.js` as follows and then test.

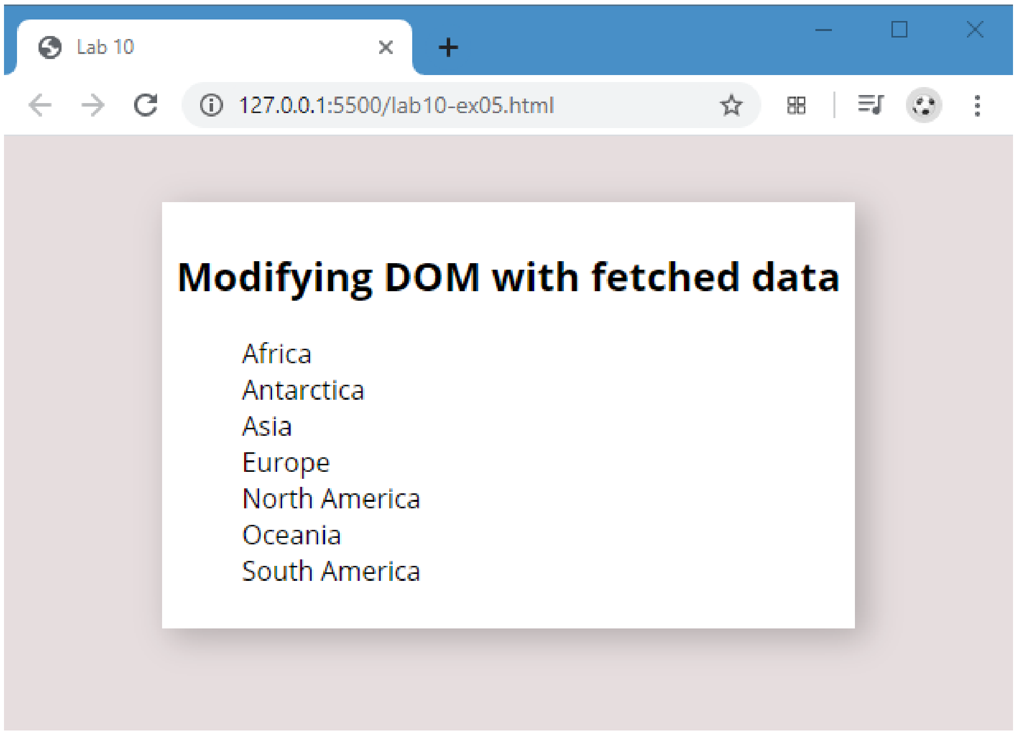
```
// fetch the continents from the api in url
fetch(url)
    .then( (resp) => resp.json() )
    .then( data => { displayContinents(data) } )
    .catch(error => console.error(error));
```

To simplify the code for explanatory purposes, we are cutting out some of the error handling.

3 Add the following function to the code.

```
function displayContinents(continents) {
    const list = document.querySelector('.box ul');

    for (let c of continents) {
        const item = document.createElement('li');
        item.textContent = c.name;
        list.appendChild(item);
    }
}
```

4 Test in browser. The result should look similar to that in Figure 10.2.



Modifying DOM with fetched data

Africa
Antarctica
Asia
Europe
North America
Oceania
South America

Figure 10.2 – Finished Exercise 10.5

5 You are now going to try modifying the code in various ways that won't work. The purpose is to illustrate some common mistakes that students make using fetch.

Change your fetch code as follows and test.

```
// this doesn't work ... why not?
let fetchedData;
fetch(url)
    .then( (resp) => resp.json() )
    .then( data => {
       // save fetched data in global variable
        fetchedData = data;
        })
    .catch(error => console.error(error));
// this looks like it should work but it doesn't
displayContinents(fetchedData);
```

Examine the error message in the console. Can you figure out why it didn't work?

6 To help understand the nature of the error, add the following console messages and test.

```
console.log('before fetch');
fetch(url)
    .then( (resp) => resp.json() )
    .then( data => {
       // save fetched data in global variable
        fetchedData = data;
        console.log('got the data');
        })
    .catch(error => console.error(error));

// this looks like it should work but it doesn't
console.log('after fetch');
console.log(fetchedData);
displayContinents(fetchedData);
```

Notice that the message 'after fetch' appears before message 'got the data'. In other words, fetchedData is undefined still. Remember the fetch takes time and is asynchronous.

7 Remove the changes made in steps 5 and 6, and add the following code. Here we are trying to set up click event handlers for the new `<li>` elements added within `displayContinents()`.

```
// this doesn't work either... why not?
const items = document.querySelectorAll('.box ul li');
console.log(items);
for (let li of items) {
    li.addEventListener('click', (e) => {
        alert(e.target.textContent);
    })
}
```

Examine the output in the console. Can you figure out why it didn't work? It's the same problem as the previous one. We are trying to set event handlers for items that don't exist yet (they won't exist until the data is received).

8 To make this work, you will need to move this code to the fetch as shown below.

```
fetch(url)
   .then( (resp) => resp.json() )
   .then( data => {
      // create the list items based on received data
     displayContinents(data);

      // set up event handlers
     const items = document.querySelectorAll('.box ul li');
     for (let li of items) {
         li.addEventListener('click', (e) => {
             alert(e.target.textContent);
         })
     }
  })
  .catch(error => console.error(error));
```

9 Test. It should work correctly now.

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 10.6 — CREATING AN AUTOCOMPLETE BOX

1 Examine `lab10-ex06.html` and then edit `js/lab10-ex06.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);
```

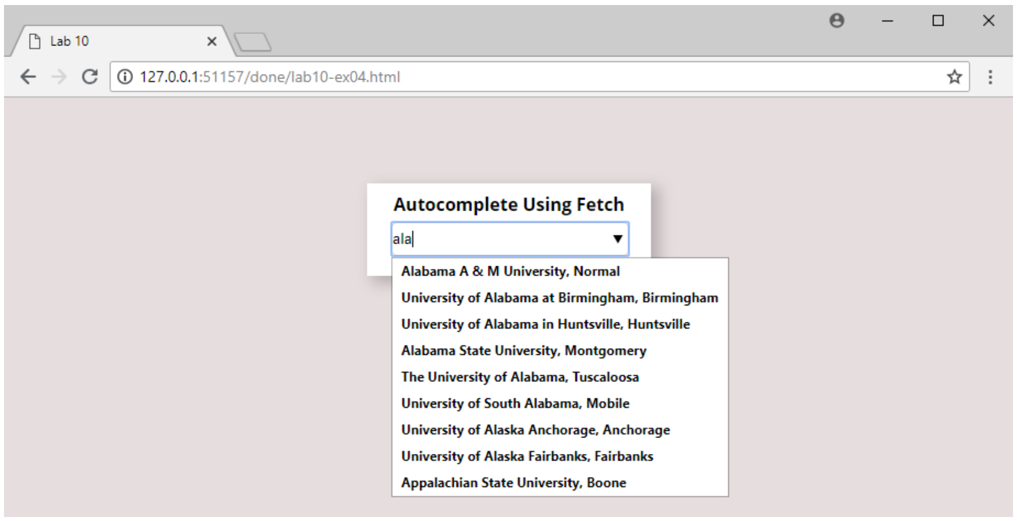
3 Create the `displayMatches()` function as follows:

```
// 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.



Alabama A & M University, Normal
University of Alabama at Birmingham, Birmingham
University of Alabama in Huntsville, Huntsville
Alabama State University, Montgomery
The University of Alabama, Tuscaloosa
University of South Alabama, Mobile
University of Alaska Anchorage, Anchorage
University of Alaska Fairbanks, Fairbanks
Appalachian State University, Boone

Figure 10.3 – Completed AutoComplete Box

Sometimes your fetch will need to send information to the server. In the next exercise, you will use fetch to POST data.

EXERCISE 10.7 — Posting Using Fetch

1 Examine `lab10-ex07.html` in the browser then examine `js/paintings.js` which is a module that contains painting data and has a method for outputting the markup for a single painting. In this exercise, you will be adding functionality to the Add to Favorites buttons.

2 Edit `js/lab10-ex07.js` and add the following function:

```
function showSnackBar(message) {
   const snack = document.querySelector("#snackbar");
   snack.textContent = message;
   snack.classList.add("show");
   setTimeout( () => {
      snack.classList.remove("show");
   }, 3000);
}
```

A snackbar is a brief message that appears at the top or bottom of a screen.

3 Instead of adding individual event handlers for each button, you will event delegation. Add the following code:

```
// set up button handlers here using event delegation
document.querySelector('main').addEventListener('click', (e) =>
{
  if (e.target.nodeName.toLowerCase() == 'button') {
    // retrieve data from button and
    let id = e.target.getAttribute('data-id');
    // get painting object for this button
    let p = paintings.find( p => p.id == id);
  }
}
```

If you inspect the generated HTML you will see that each button has a data-id attribute which contains its id property.

4 Your button is going to `POST` the painting id and title to the external API. One way to construct the data the page will be posting is to use the JavaScript `FormData` object. Add the following within the `if` statement added in the previous step.

```
// We will post the painting id and title to favorites
let formBody = new FormData();;
formBody.set("id",p.id);
formBody.set("title",p.title);
```

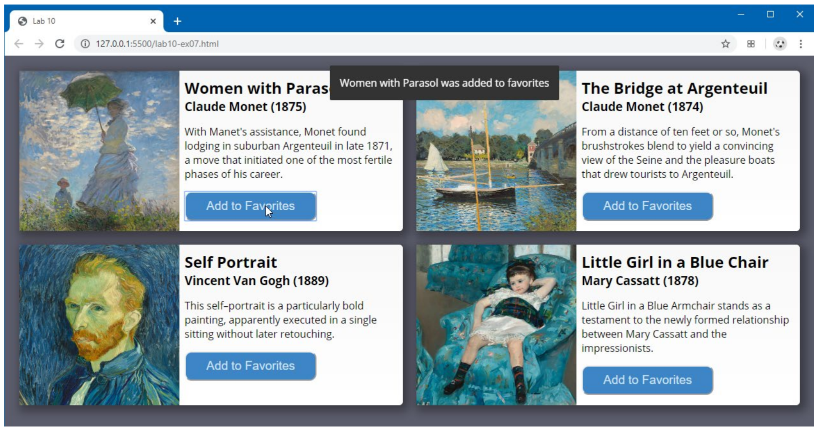
5 In order to have fetch make a POST request, you will need to construct an options parameter that contains the data to post. Add the following after code from the previous step.

```
const opt = {
   method: 'POST',
   body: formBody
};
```

6 Now make your fetch request:

```
// now let's post via fetch
fetch(url, opt)
   .then( resp => resp.json() )
   .then( data => {
      // display notification it was successful
      showSnackBar(`${data.received.title} was added to
favorites`);
      })
   .catch( error => {
      showSnackBar('An error occurred, favorites not
modified');
      });
}
```

7 Test. The result should look similar to that in Figure 10.4.



Women with Parasol was added to favorites

Women with Paras
Claude Monet (1875)
With Manet's assistance, Monet found lodging in suburban Argenteuil in late 1871, a move that initiated one of the most fertile phases of his career.

The Bridge at Argenteuil
Claude Monet (1874)
From a distance of ten feet or so, Monet's brushstrokes blend to yield a convincing view of the Seine and the pleasure boats that drew tourists to Argenteuil.

Self Portrait
Vincent Van Gogh (1889)
This self-portrait is a particularly bold painting, apparently executed in a single sitting without later retouching.

Little Girl in a Blue Chair
Mary Cassatt (1878)
Little Girl in a Blue Armchair stands as a testament to the newly formed relationship between Mary Cassatt and the impressionists.

Figure 10.4 – Completed exercise 10.7

Fetching takes time. A common user interface feature is to supply the user with a loading animation while the data is being fetched. The next exercise will illustrate how this works.

EXERCISE 10.8 — ADDING A LOADING ANIMATION

1 Examine `lab10-ex08.html` in the browser. Notice the empty `<div>` with the `id` of `movieLoading`, which contains the CSS loading animation. This example also fetches a very large data set and might take 20-30 seconds to load.

2 Add the following code just before the fetch to `js/lab10-ex08.js`.

```
// hide form and display loading animation
document.querySelector("form.box").style.display = "none";
document.querySelector("#movieLoading").style.display =
       "block";
```

3 Add the following code within the fetch.

```
fetch(endpoint)
  .then(response => response.json())
  .then(data => {
     // show form and hide animation
      document.querySelector("form.box").style.display =
           "block";
      document.querySelector("#movieLoading").style.display =
           "none";
      movies.push(...data);

  } )
  .catch(error => console.error(error));
```

4 Test. The loading animation should work.

Test Your Knowledge #3

In this exercise, you will be working with nested fetches. You will be modifying `lab10-test03.js` and consuming two APIs. The URLs for the two APIs and the location of the photo image files are included in the JavaScript file. The result will look similar to that shown in Figure 10.5.

1 You will do the first fetch after the user clicks the load button; you will use the fetched country information to populate the <select> list. Be sure to first empty the <select> element (by setting its innerHtml property to ""; otherwise each time the button is clicked, the list size will grow. For each <option> element, set its `value` attribute to the `iso` property of each country object.

Display the first loading animation and hide the <main> element while the fetch is working.

2 When the user selects a country, you will do another fetch to retrieve the photos for that country. This will require adding a querystring to the photo URL that contains the `iso` of the country (which can be obtained from the `value` property of the <select>). The querystring must have the name iso; for instance, to retrieve the photos from Canada, the querystring would be: iso=CA.

The fetched photo data has a filename property that can be appended to the supplied image URL and used for the src attribute of the generated `<img>`. Set the title and alt attributes of the image to the photo's title property.

Display the second loading animation while the second fetch is working. Be sure to also empty the <section> element before adding photos to it.

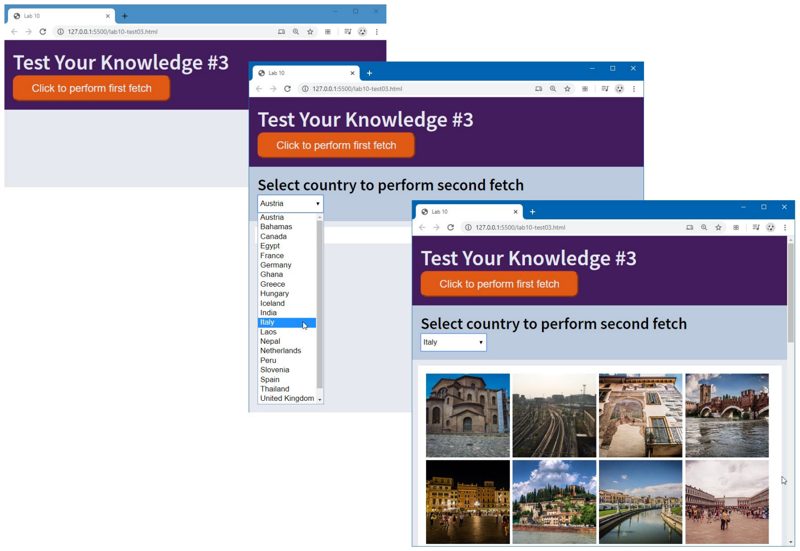


Figure 10.4 – Completed Test Your Knowledge #3

EXERCISE 10.9 — CREATING PROMISES

1 Examine `lab10-ex09.html` in the browser. You will be creating promises to simulate working with asynchronous cloud services.

2 Modify `lab10-ex09.js` by adding the following:

```
function saveInCloud(filename) {
   // some sample filenames to simulate existing files
   const existingFiles = ['cats.jpg', 'family.jpg',
           'afewdogs.jpg', 'farm.jpg'];

   return new Promise( (resolve, reject) => {
      console.log('saving to cloud ...');
     // to simulate time delay in working with external
     // service, do the next steps after a time delay
      setTimeout( () => {
         // just have a made-up AWS url for now
         let cloudURL = `http://bucket.s3-aws-
region.amazonaws.com/makebelieve/${filename}`;
         // see if passed filename exists
         if ( existingFiles.some( file => file == filename) )
            resolve(cloudURL)
         else
            reject( new Error(`${filename} does not exist`));
      }, 2000);

   });
}
```

3 Use this function by adding the following:

```
function simplePromise(filename) {
   saveInCloud(filename)
      .then( url => {
         console.warn(`Cloud URL for ${filename} is ${url}`);
      })
      .catch(err => {
         console.warn(err);
      });
}
```

4 Test. Be sure to have the console visible before clicking buttons.

Right now the last two buttons don't do anything. The next steps will chain several promises together.

5 Add the following functions:

```
function tagImageContent(url) {
   return new Promise ( (resolve, reject) => {
      console.log(`Using ML to tag ${url} ...`);
      // list of sample tags
      const tags = ['fun', 'animal', 'cute'];
      // for now simply resolve after a 3 second delay
      setTimeout( () => resolve(tags), 3000);
   });
}
function createThumbnail(url) {
   return new Promise( (resolve, reject) => {
      console.log(`Creating thumbnail of ${url} ...`);
      // simply resolve after a second with a fake CDN url
      const thumbURL =
         "https://res.cloudinary.com/myapp/totallyfake.jpg"
      setTimeout( () => resolve(thumbURL), 1000);
   } );
}
```

6 Use these functions by adding the following:

```
function chainedPromises(filename) {
   saveInCloud(filename)
      .then( url => {
         console.warn(`Cloud URL for ${filename} is ${url}`);
         return tagImageContent(url)
      })
      .then( tags => {
         console.warn('Returned tags=' + tags);
         return createThumbnail('???');
      })
      .then ( url => {
         console.warn('Thumbnail url=' + url);
      })
      .catch(err => {
         console.warn(err);
      });
}
```

Why do the first two then() callbacks return another function? Remember that the then() function should always return a Promise. We want these two actions (create thumbnail and create thumbnail) to happen after the saveInCloud, so they are chained together.

7 Test the chained button.

We still have one problem. The createThumbnail() function needs the cloud URL returned from the resolved saveInCloud. We can solve this simply by saving the value in a variable outside the function.

8 Add the following and test.

```
function chainedPromises(filename) {
   let cloudURL;
   saveInCloud(filename)
      .then( url => {
         console.warn(`Cloud URL for ${filename} is ${url}`);
         cloudURL = url;
         return tagImageContent(url);
      })
      .then( tags => {
         console.warn('Returned tags=' + tags);
         return createThumbnail(cloudURL);
      })
      .then ( url => {
         console.warn('Thumbnail url=' + url);
      })
      .catch(err => {
         console.warn(err);
      });
}
```

9 Both `tagImageContent` and `createThumbnail` could potentially happen at the same time. To do so, edit the code as follows and test using the last button.

```
function simultaneousPromises(filename) {
  saveInCloud(filename)
    .then( url => {
       console.warn(`Cloud URL for ${filename} is ${url}`);
       Promise.all([ tagImageContent(url),
                     createThumbnail(url) ])
            .then(resolves => {
               // do something with array of responses
                const [tags, thumbURL] = resolves;
                console.warn(`Returned tags=${tags}
                                   thumbnail=${thumbURL}`);
            });
     })
     .catch(err => {
         console.warn(err);
     });
}
```

EXERCISE 10.10 — ASYNC AND AWAIT

1 Test `lab10-ex10.html` in the browser. You will be refactoring the fetches used in this exercise to use `async … await` instead.

2 In `lab10-ex10.js` comment out the existing `displayCountries()` and create a new one as follows (you can copy and paste most of this code from existing function).

```
function displayCountries() {
   const countries = getCountryData();

   filters.style.display = "block";
   countries.forEach( c => {
      const opt = document.createElement('option');
      opt.setAttribute('value',c.iso );
      opt.textContent = c.name;
      select1.appendChild(opt);
   });
   select1.addEventListener('input', displayCities);
}
```

3 Create the following new function:

```
async function getCountryData() {
   const response = await fetch(countryAPI);
   const data = await response.json();
   return data;
}
```

4 Test. It won't work.

Examine the console to see the error message. The problem is that countries is null.

5 Add in the following console messages and test.

```
function displayCountries() {
   console.log('before getCountryData');
   const countries = getCountryData();
   console.log('after getCountryData');

   filters.style.display = "block";
   countries.forEach( c => {
      const opt = document.createElement('option');
      opt.setAttribute('value',c.iso );
      opt.textContent = c.name;
      select1.appendChild(opt);
   });
   select1.addEventListener('input', displayCities);
}

async function getCountryData() {
   const response = await fetch(countryAPI);
   console.log('after first await');
   const data = await response.json();
   console.log('after second await');
   return data;
}
```

It is important to remember that the async function is asynchronous; the waiting only happens within the async function itself. That is, the next line after the call to getCountryData will get called after the await line in it is called, but before it is satisfied.

6 Refactor the code as follows by replacing it with the following. Test.

```
async function displayCountries() {
   const response = await fetch(countryAPI);
   const countries = await response.json();

   filters.style.display = "block";
   countries.forEach( c => {
      const opt = document.createElement('option');
      opt.setAttribute('value',c.iso );
      opt.textContent = c.name;
      select1.appendChild(opt);
   });
   select1.addEventListener('input', displayCities);
}
```

7 What about the missing `.catch()`? You can instead use normal JavaScript `try…catch` instead. Add the following and test.

```
async function displayCountries() {
   try {
      const response = await fetch(countryAPI);
      const countries = await response.json();

      filters.style.display = "block";
      countries.forEach( c => {
         const opt = document.createElement('option');
         opt.setAttribute('value',c.iso );
         opt.textContent = c.name;
         select1.appendChild(opt);
      });
      select1.addEventListener('input', displayCities);
   }
   catch (err) {
      console.error(err);
   }
}
```

8 You can also use async with anonymous functions. Copy the function code from step 6 and paste it in the modification shown below. It should work as well.

```
button.addEventListener('click', async () => {
   const response = await fetch(countryAPI);
   const countries = await response.json();

   filters.style.display = "block";
   countries.forEach( c => {
      const opt = document.createElement('option');
      opt.setAttribute('value',c.iso );
      opt.textContent = c.name;
      select1.appendChild(opt);
   });
   select1.addEventListener('input', displayCities);
});
```

9 Modify displayCities to use async…await.

Test Your Knowledge #4

In this exercise, you will extend Test Your Knowledge #3. As you can see in Figure 10.5, this exercise will contain four <select> elements, which will be populated from four APIs. You will also use async…await.

1 Examine lab10-test04.js and test out the URLs of the four APIs included in the starting code.

2 These API fetches are unrelated so they can happen at the same time using Promise.all(). Use this method in conjunction with async…await. Once the data is retrieved, sort each data set using the name or lastName (for users) property.

3 Populate the four select lists using the following properties from the retrieved data:

Continents: code (value attribute of each option), name (textContent of each option)

Countries: iso (value attribute of each option), name (textContent of each option)

Cities: id (value attribute of each option), name (textContent of each option)

Users: id (value attribute of each option), lastName (textContent of each option)

4 Add event handlers for the input event of each select. When the user chooses an item from one of the lists, then use the photo API to retrieve photos for the specified continent, country, city, or user. You will supply a different querystring parameter based on which filter criteria to use. For instance:

```
images.php?continent=NA
images.php?iso=CA
images.php?city=252920
images.php?user=2
```

5 Once you have retrieved the images, display them in the same manner as in Test Your Knowledge #3.

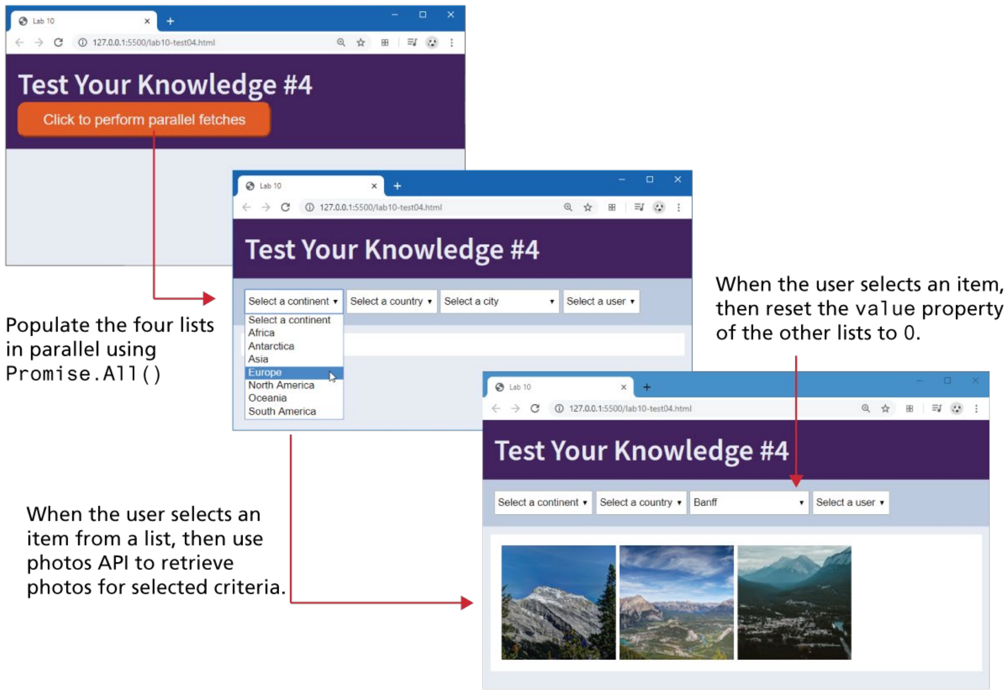

Test Your Knowledge #4
Click to perform parallel fetches
Populate the four lists in parallel using Promise.All()
Select a continent
Select a country
Select a city
Select a user
Africa
Antarctica
Asia
Europe
North America
Oceania
South America
When the user selects an item, then reset the value property of the other lists to 0.
When the user selects an item from a list, then use photos API to retrieve photos for selected criteria.
Banff

Figure 10.5 – Completed Test Your Knowledge #4

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 10.11 — USING THE STORAGE API

1 Some of the functionality has already been implemented in the starting files. Examine `lab10-ex11.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 10.6)

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-ex11.js` by modifying the following code (don't test yet).

```
document.addEventListener("DOMContentLoaded", 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-ex11.html` page. **The schemes should still be there.**

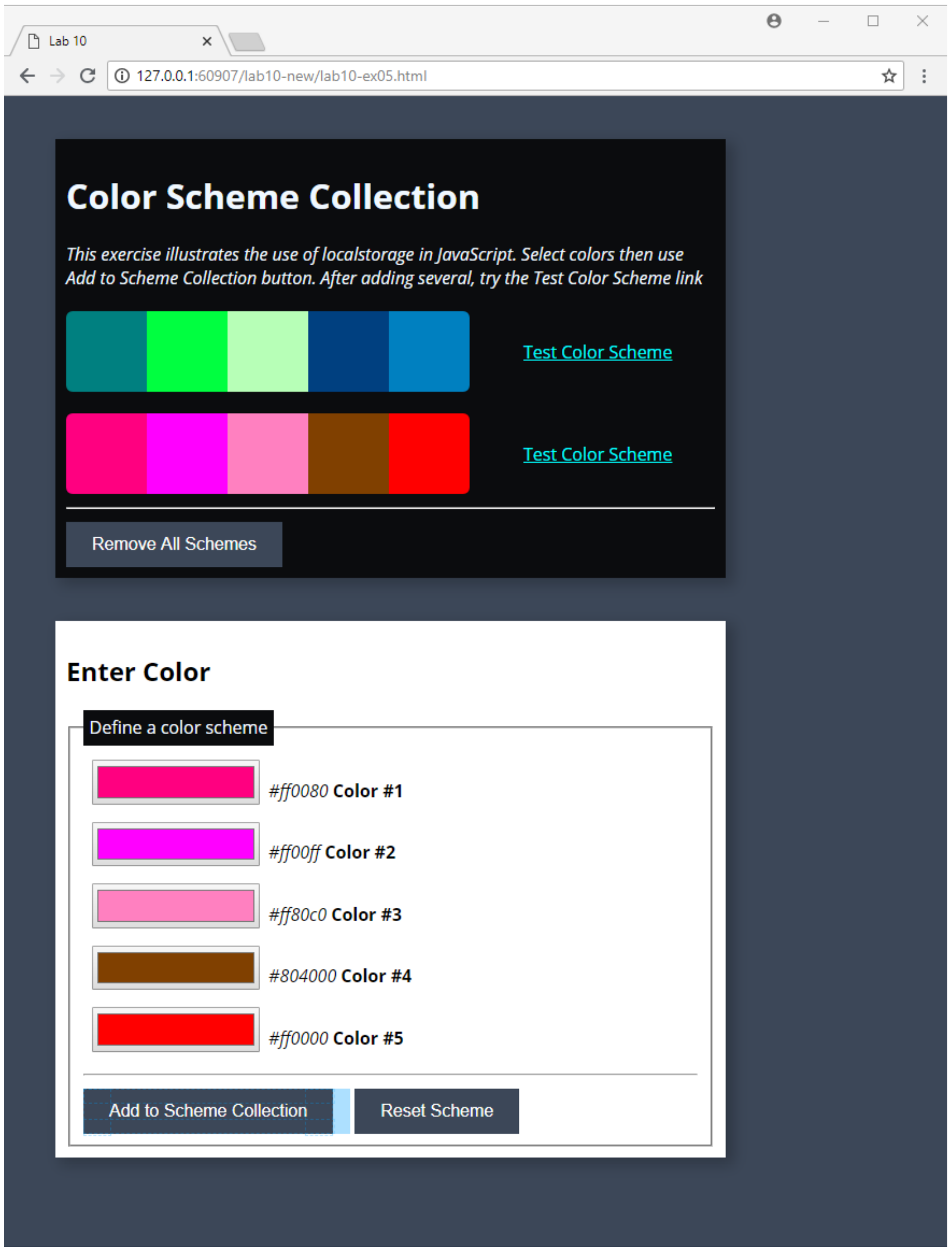
10 Add the following code to the `setupRemoveAllHandler()` method:

```
// 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-ex11-tester.js` to `sessionStorage` as well.

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



Color Scheme Collection

This exercise illustrates the use of localstorage in JavaScript. Select colors then use Add to Scheme Collection button. After adding several, try the Test Color Scheme link

Test Color Scheme

Test Color Scheme

Remove All Schemes

Enter Color

Define a color scheme

#ff0080 **Color #1**

#ff00ff **Color #2**

#ff80c0 **Color #3**

#804000 **Color #4**

#ff0000 **Color #5**

Add to Scheme Collection   Reset Scheme

Figure 10.6 – Completed Exercise 10.11

EXERCISE 10.12 — USING SPEECH SYNTHESIS

1 Examine `lab10-ex12.html` and then edit `js/lab10-ex12.js` by adding the following code. Save then test by clicking on the Speak button (you will need to have your device's sound turned on).

```
document.addEventListener("DOMContentLoaded", function () {
   document.querySelector('#speak').addEventListener('click',
      (e) => {
         const utterance = new SpeechSynthesisUtterance
                                ('Hey there buddy');
         speechSynthesis.speak(utterance);
   });
});
```

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

2 Comment out the code 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', ()=>{
   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', () => {
   let select = document.querySelector("#voices");
    // get array of 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. Currently, this is listed as experimental **and the voice code listed here only works in Chrome.***

5 Now add the handler for the Speak button within the voiceschanged handler, then test.

```
// now add event handler for speak button
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 === selectedVoice);
  utterance.rate = document.querySelector('#rate').value;
  utterance.pitch = document.querySelector('#pitch').value;
  // all ready, make it speak
  window.speechSynthesis.speak(utterance);
});
```

It should work (in Chrome)!

External APIs

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 10.13 — 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-ex13.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-ex13.css` by adding the following:

```
#map {
    height: 600px;
}
```

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-ex13.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) );
```

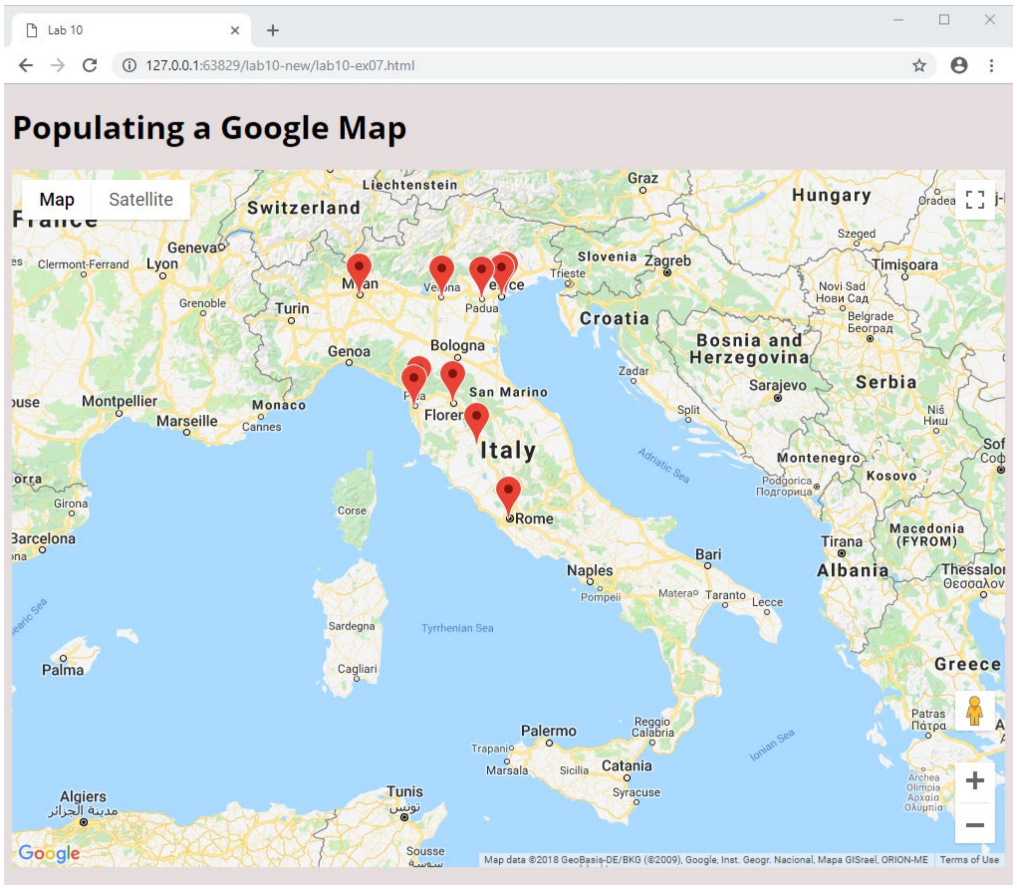


Figure 10.1 – Exercise 10.13

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 #5

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 data is received, you will have to programmatically change the CSS `display` property from `none` to `block`.

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.

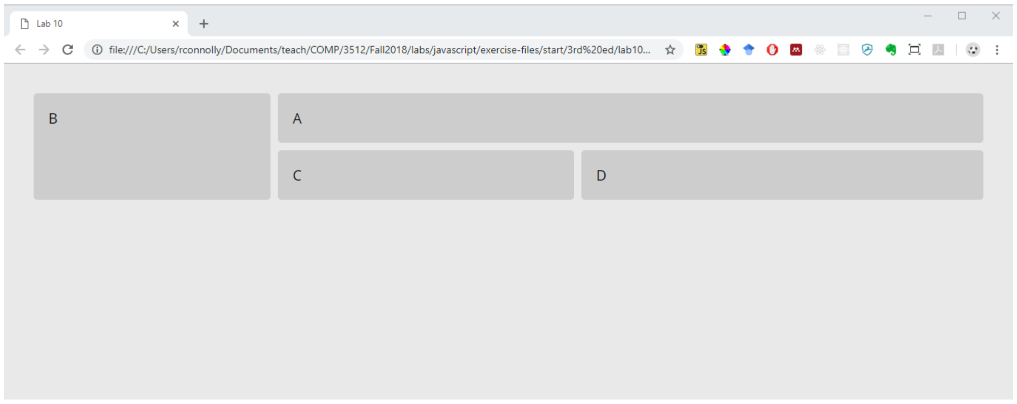
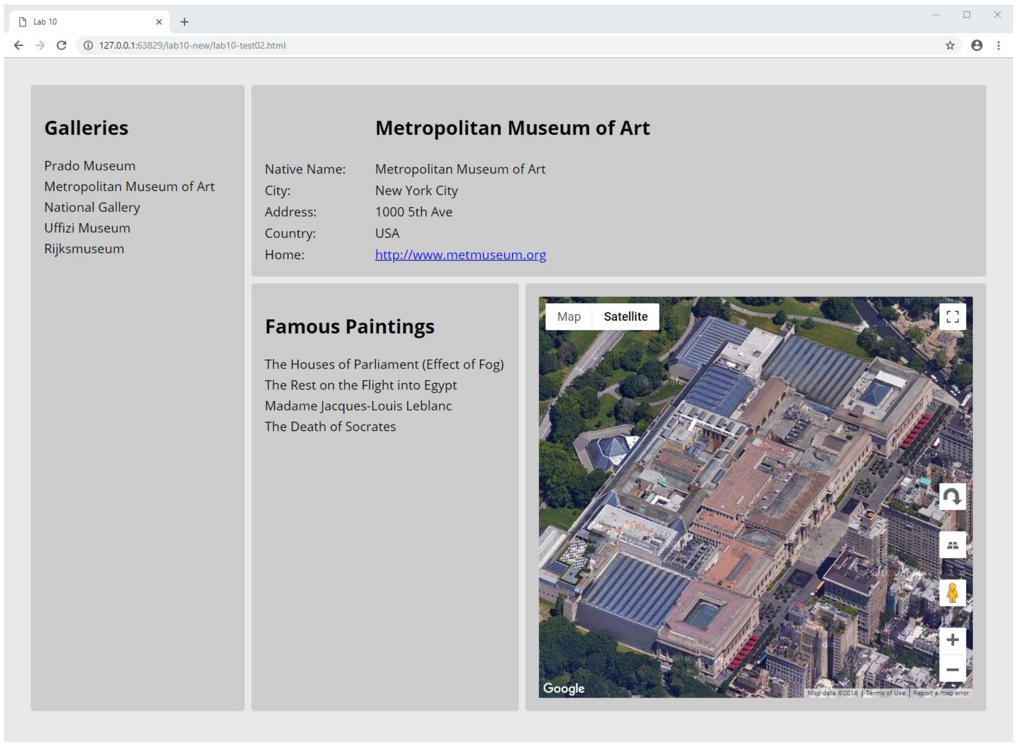

Lab 10
file:///C:/Users/rconnolly/Documents/teach/COMP/3512/Fall2018/labs/javascript/exercise-files/start/3rd%20ed/lab10...
B
A
C
D

Figure 10.2 – Layout of lab10-test05 at start



Galleries

Prado Museum
Metropolitan Museum of Art
National Gallery
Uffizi Museum
Rijksmuseum

Metropolitan Museum of Art

Native Name:	Metropolitan Museum of Art
City:	New York City
Address:	1000 5th Ave
Country:	USA
Home:	http://www.metmuseum.org

Famous Paintings

The Houses of Parliament (Effect of Fog)
The Rest on the Flight into Egypt
Madame Jacques-Louis Leblanc
The Death of Socrates

Figure 10.3 – Finished lab10-test05