Fundamentals of Web Development

Third Edition by Randy Connolly and Ricardo Hoar

Chapter 10

JavaScript 3:

Additional Features

In this chapter you will learn . . .

- Additional language features in JavaScript
- How to asynchronously consume web APIs in JavaScript
- Extend the capabilities of your pages using browser APIs
- Utilize external APIs for mapping and charting

Array Functions

- **forEach()** iterate through an array
- **find()** find the *first* object whose property matches some condition
- **filter()** find all matches whose property matches some condition
- **map()** is similar manner to filter except it creates a new array of the same size whose values have been transformed by the passed function
- **reduce()** reduces an array into a single value
- **sort()** sorts a one-dimensional array

Array forEach()

This function will be called for each element in the array

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

Each element is passed in as an argument to the function.

```
const paintings = [
  {title: "Girl with a Pearl Earring", artist: "Vermeer"},
  {title: "Artist Holding a Thistle", artist: "Durer"},
  {title: "Wheatfield with Crows", artist: "Van Gogh"},
  {title: "Burial at Ornans", artist: "Courbet"},
  {title: "Sunflowers", artist: "Van Gogh"}
];
```

Array find()

One of the more common coding scenarios with an array of objects is to find the *first* object whose property matches some condition. This can be achieved via the **find**() method of the array object, as shown below.

```
const courbet = paintings.find( p => p.artist === 'Courbet' );
console.log(courbet.title); // Burial at Ornans
```

Like the **forEach**() method, the **find**() method is passed a function; this function must return either true (if condition matches) or false (if condition does not match). In the example code above, it returns the results of the conditional check on the artist name.

Array filter()

If you were interested in finding all matches you can use the filter() method, as shown in the following:

```
// vangoghs will be an array containing two painting objects
const vangoghs = paintings.filter(p => p.artist === 'Van Gogh');
```

Since the function passed to the filter simply needs to return a true/false value, you can make use of other functions that return true/false. For instance, you could perform a more sophisticated search using regular expressions.

Array map()

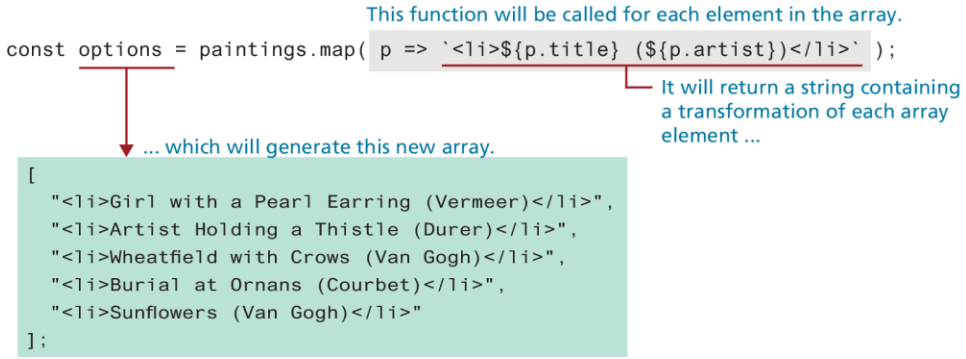
The map() function operates in a similar manner except it creates a new array of the same size but whose values have been transformed by the passed function.

Listing 10.2 shows map using DOM nodes. Figure 10.2 uses strings.

```
// create array of DOM nodes
const options = paintings.map( p => {
    let item = document.createElement("li");
    item.textContent = `${p.title} (${p.artist})`;

    return item;
});
```

LISTING 10.2 Using the map() function



Reduce

The **reduce**() function is used to reduce an array into a single value. Like the other array functions in this section, the **reduce**() function is passed a function that is invoked for each element in the array.

This callback function takes up to four parameters, two of which are required: the previous accumulated value and the current element in the array.

For instance, the following example illustrates how this function can be used to sum the **value** property of each painting object in our sample paintings array:

```
let initial = 0;

const total = paintings.reduce( (prev, p) => prev + p.value, initial);
```

Sort

sort() function sorts in ascending order (after converting to strings if necessary)

```
const names = ['Bob', 'Sue', 'Ann', 'Tom', 'Jill'];
const sortedNames = names.sort();
// sortedNames contains ["Ann", "Bob", "Jill", "Sue", "Tom"]
```

If you need to sort an array of objects based on one of the object properties, you will need to supply the sort() method with a compare function that returns either 0, 1, or −1, depending on whether two values are equal (0), the first value is greater than the second (1), or the first value is less than the second (−1).

Custom sort

```
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;
} );

// more concise version using ternary operator and arrow syntax
const sorted2 = paintings.sort( (a,b) => a.year < b.year? -1: 1 );
```

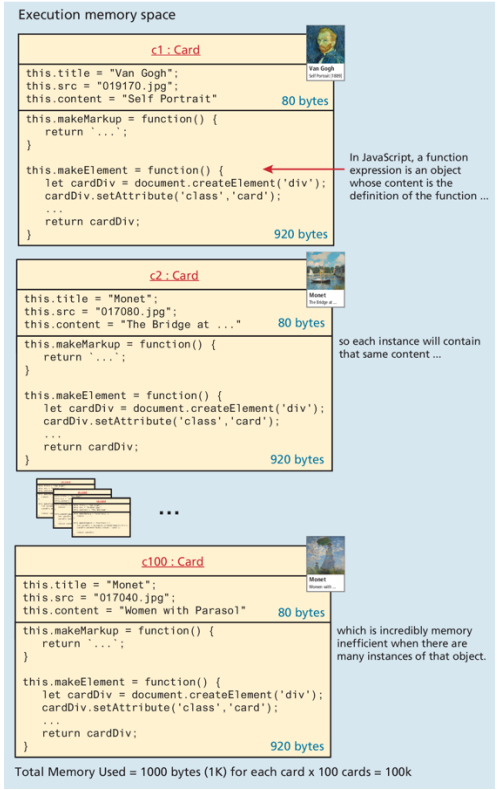
LISTING 10.3 Sorting an array based on the properties of an object

Prototypes, Classes, and Modules

In Chapter 8, you learned how to use constructor functions as an approach for creating multiple instances of objects that need to have the same properties.

While the constructor function is simple to use, it can be an inefficient approach for objects that contain methods since memory must be allocated for each (identical) method.

Prototypes help address this issue.



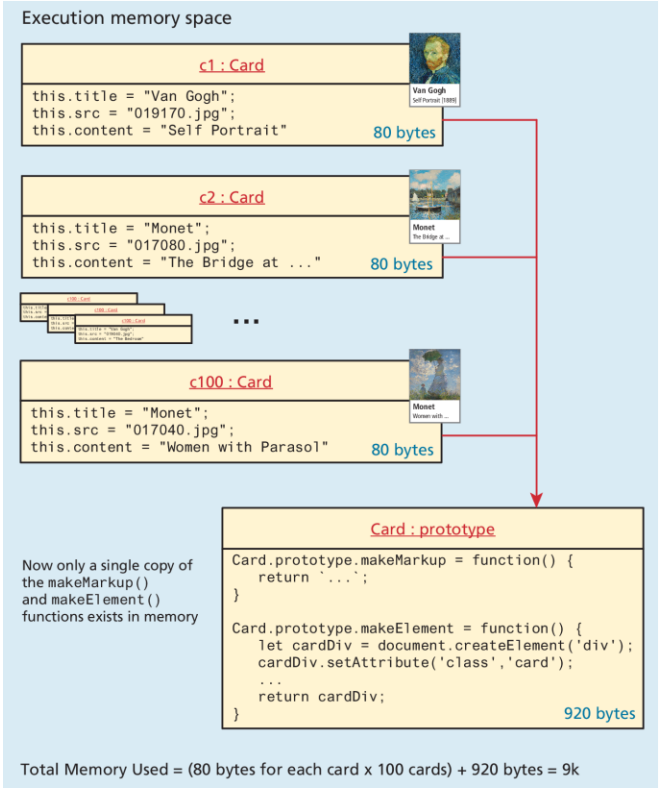
Prototypes

Prototypes make JavaScript behave more like an object-oriented language.

Every function object has a **prototype** property, which is initially an empty object.

The prototype properties are defined once for all instances of an object created with the new keyword from a constructor function.

This approach is far superior because it defines the method only once, no matter how many instances are created.



Using a prototype

```
function Card(title, src, content) {
    this.title = title;
    this.src = src;
    this.content = content;
}

Card.prototype.makeMarkup = function() {
    return `<div class="card">
            <img src="${this.src}" alt="${this.title}" >
            <div>
                <h4>${this.title}</h4>
                <p>${this.content}</p>
            </div>
            </div>`;
};
```

```
Card.prototype.makeElement = function() {
    let cardDiv = document.createElement('div');
     ....
    cardDiv.appendChild(div);
    return cardDiv;
};

// You use prototype functions as if they were declared
in the object
const container =
document.querySelector("#container");
const c1 = new Card("Van Gogh", "019170.jpg", "Self
Portrait");
container.appendChild( c1.makeElement() );
```

LISTING 10.5 Using a prototype

Using Prototypes to Extend Other Objects

Prototypes also enable you to extend existing objects (including built-in objects) by adding to their prototypes. Imagine a method added to the String object that allows you to count instances of a character.

```
String.prototype.countChars = function (c) {
        let count=0;
        for (let i=0;i<this.length;i++) {
                if (this.charAt(i) == c)
                        count++;
        }
        return count;
}
```

```
const msg = "HELLO WORLD";
console.log(msg + " has" +
msg.countChars("L") + " letter L's");
```

LISTING 10.6 Extending a built-in object using the prototype

Classes

- A **class** provides an alternate syntax for a function constructor and the extension of it via its prototype. In reality, they are merely “syntactical sugar” for JavaScript’s prototype approach to inheritance
- While the class syntax provides a familiar alternate syntax for working with functions, the developer community has not universally adopted it
- Regardless of these concerns, the React framework, which has become one of the most widely adopted frameworks in the past several years (and which is covered in Chapter 11), does use JavaScript class syntax, so it is likely that as a JavaScript developer you will encounter this syntax more and more moving forward.

Using a class

```
class Card {
   // constructor replaces the function constructor
   constructor(title, src, content) {
   this.title = title;
   this.src = src;
   this.content = content;
   }
   // class methods replace prototypes
   makeMarkup() {
   return `<div class="card">
         <img src="${this.src}" alt="${this.title}" >
         <div>
            <h4>${this.title}</h4>
            <p>${this.content}</p>
         </div>
         </div>`;
   };
   // notice the function property shorthand syntax
   makeElement() {
      let cardDiv = document.createElement('div');
      ...
      return cardDiv;
   }
}

// Use the class
const container =
document.querySelector("#container");
const c1 = new Card("Van Gogh", "images/019170.jpg",
"Self Portrait");
container.append( c1.makeElement() );
```

LISTING 10.7 Implementing Listing 10.5 (slide 13) using class syntax

Extending classes and more

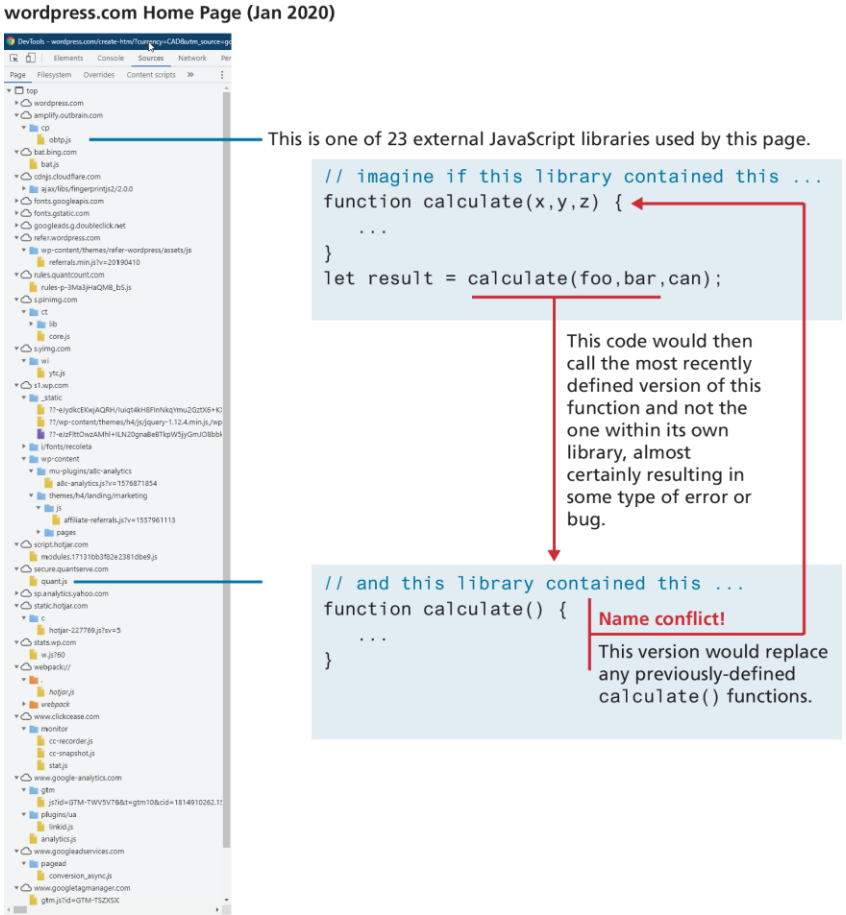
There are additional syntactical features of classes in JavaScript, including getters/ setters and static functions that we are not covering.

Extending classes is one advanced feature worth noting. The **extends** keyword lets a class inherit the properties and methods of another class as with class-based programming languages such as Java or C#

```
class AnimatedCard extends Card {
    constructor(title, src, content, effect) {
    super(title, src, content)
    this.effect=effect;
}
```

Name Conflicts

As shown in Figure 10.5, complex contemporary JavaScript applications might contain hundreds of literals defined in dozens of .js files, so some way of preventing name conflicts becomes especially important



Modules

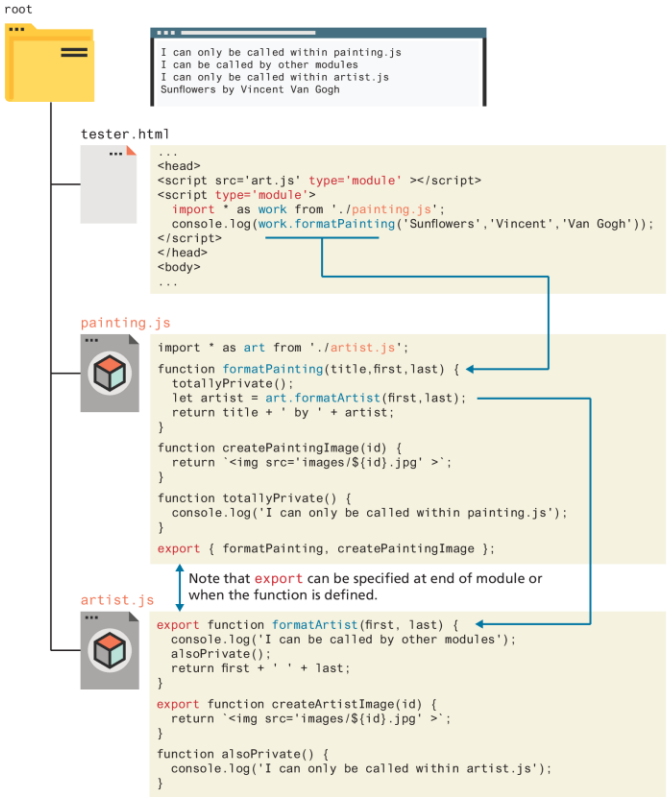
An ES6 **module** is simply a file that contains JavaScript. Unlike a regular JavaScript external file, literals defined within the module are scoped to that module.

You do have to tell the browser that a JavaScript file is a module and not just a regular external JavaScript file within the <script> element. This is achieved via the type attribute as shown in the following:

```
<script src="art.js" type="module"></script>
```

Within a module, any literals are private to that module. To make content in the module file available to other scripts outside the module, you have to make use of the **export** keyword.

Visualizing Modules in JavaScript



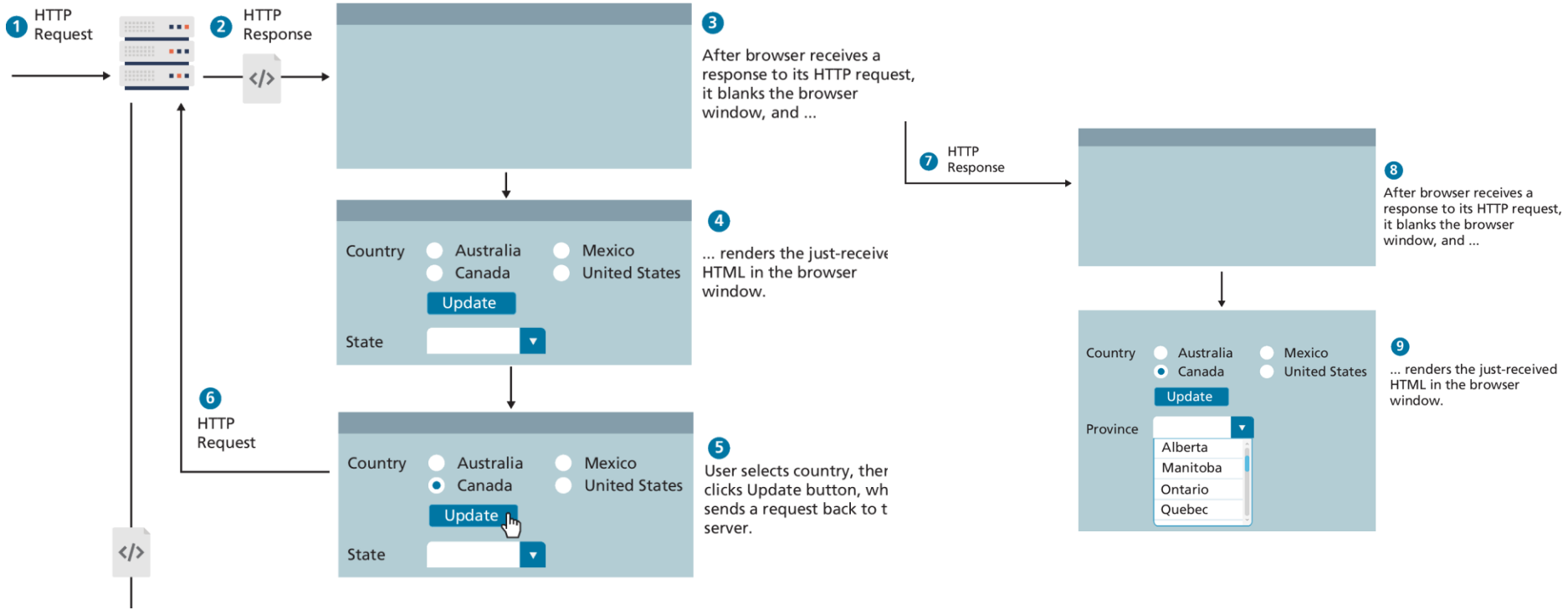
Asynchronous Coding with JavaScript

asynchronous code is code that is doing (or seemingly doing) multiple things at once. In multi-tasking operating systems, asynchronous execution is often achieved via **threads**: each thread can do only one task at a time, but the operating system switches between threads

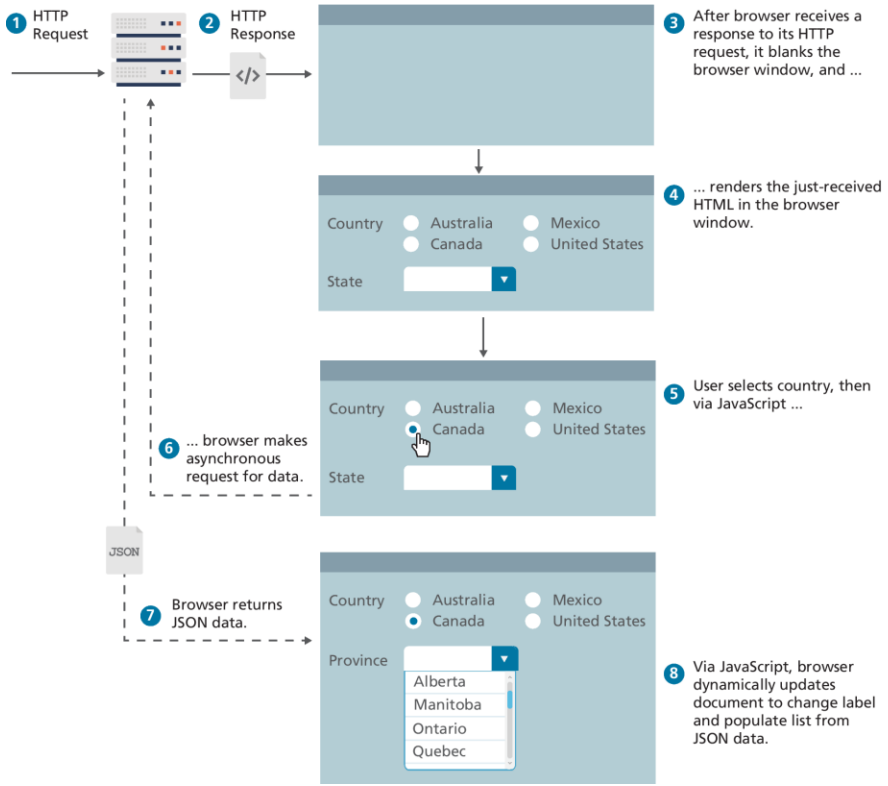
Many contemporary web sites make use of asynchronous JavaScript data requests of Web APIs, thereby allowing a page to be dynamically updated without requiring additional HTTP requests.

A **web API** is simply a web resource that returns data instead of HTML, CSS, JavaScript, or images

Normal HTTP request–response loop



Asynchronous data requests

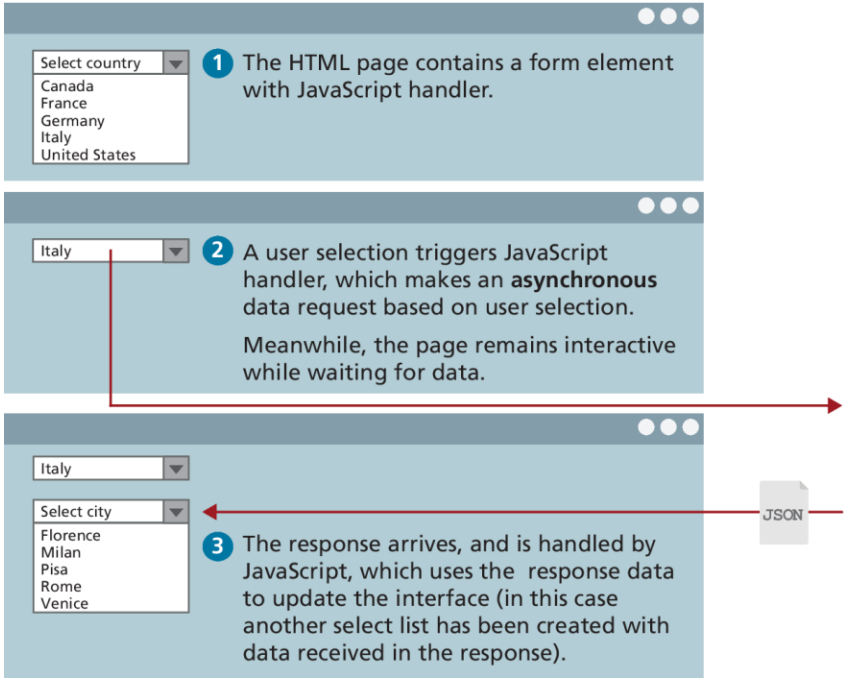


Fetching Data from a Web API

To illustrate fetch, consider the scenario of a page containing a <select> element.

When the user selects from the country list, the page makes an asynchronous request to retrieve a list of cities for that country.

```
let cities = fetch('/api/cities.php?country=italy');
```



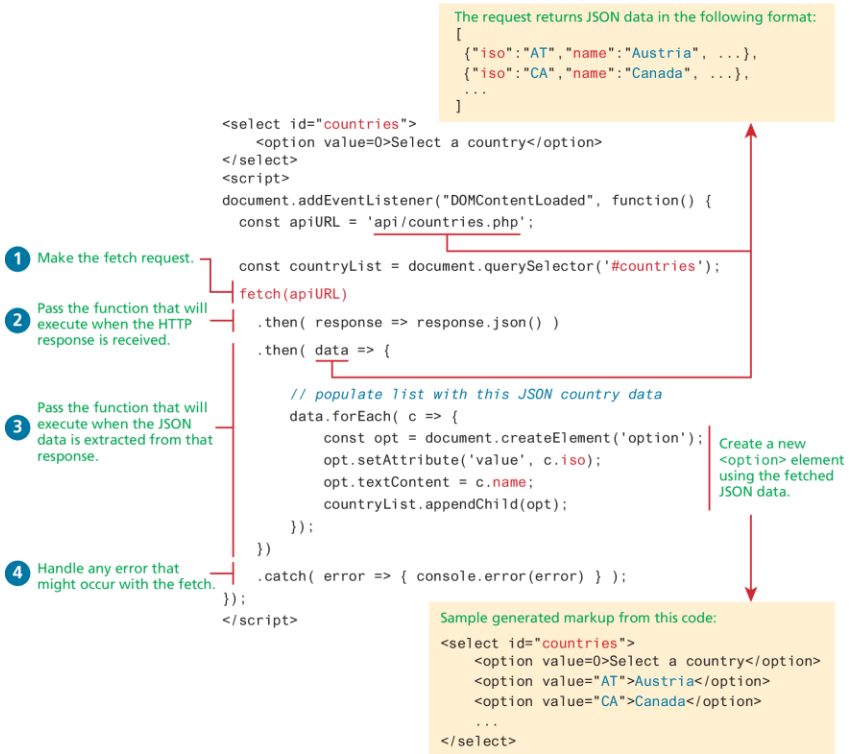
Fetching Data from a Web API (ii)

So what does cities contain after this call? You might expect it to contain the requested JSON data. But it doesn't. Why? Because it will take time for this service to execute and respond to our request.

What the above fetch will return instead is a special Promise object.

Promises in JavaScript are usually handled by invoking two methods: then() for responding to the successful arrival of data, and catch() for responding to an unsuccessful arrival.

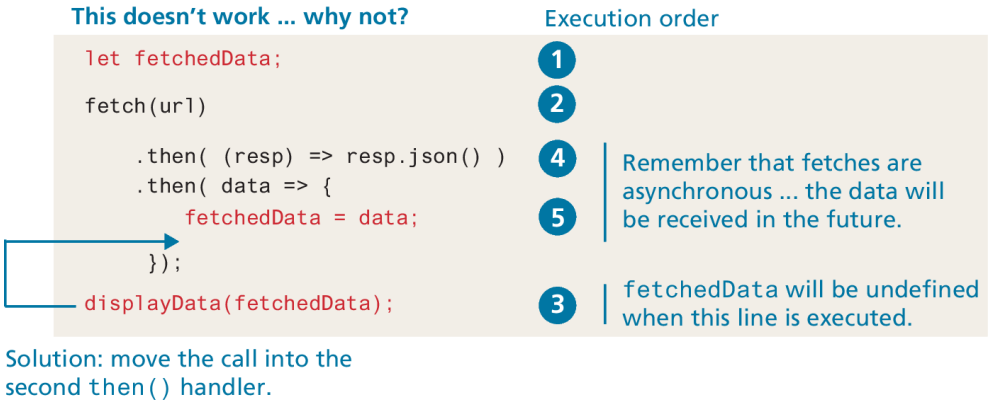
Example of asynchronous request using fetch



Common Mistakes with Fetch

Students often struggle at first with using fetch and often commit some version of the mistake shown in Figure 10.14.

Multiple nested fetches can be problematic,



Cross-Origin Resource Sharing

Modern browsers prevent cross-origin requests by default (Chapter 16), so sharing content legitimately between two domains becomes harder.

Cross-origin resource sharing (CORS) is a mechanism that uses new HTTP headers in the HTML5 standard that allows a JavaScript application in one **origin** (i.e., a protocol, domain, and port) to access resources from a different origin.

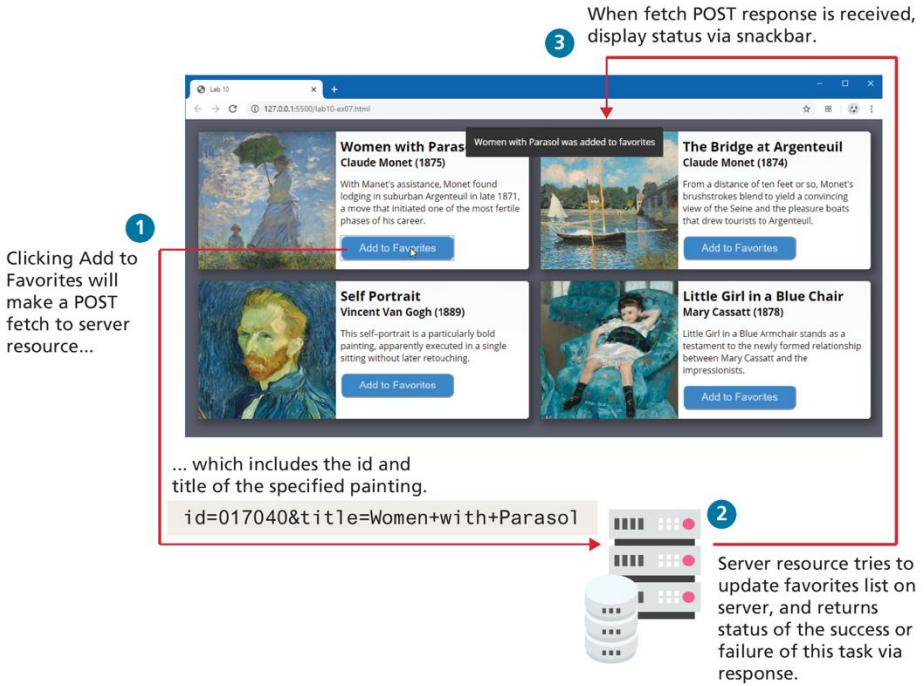
If an API site wants to allow any domain to access its content through JavaScript, it would add the following header to all of its responses:

Access-Control-Allow-Origin: *

Fetching Using Other HTTP Methods

By default, fetch uses the HTTP GET method. There are times when you will instead want to use POST, or even PUT or DELETE

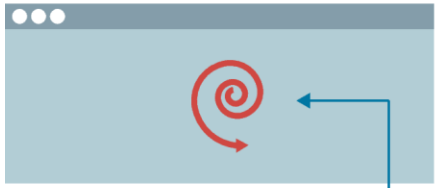
For instance, imagine you wanted to add an item to a favorites list or to a shopping cart in an asynchronous manner. This would typically require sending data to the server, so a POST fetch makes the most sense.



Adding a Loading Animation

Fetching takes time. A common user interface feature is to supply the user with a loading animation while the data is being fetched.

Simply show or hide an element that contains either an animated GIF, or, even better, CSS that uses animation



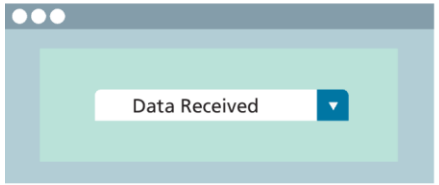
```
<div id="load"></div>
<div id="box">
   ...
</div>
```

A CSS animation is defined for this id

This div will eventually display fetched data

1 Before fetch hide content box and show loading div.

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

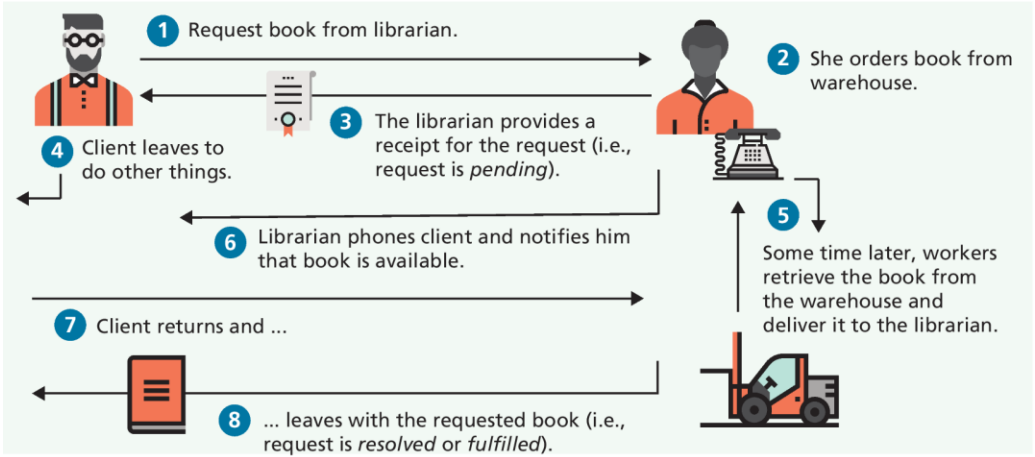


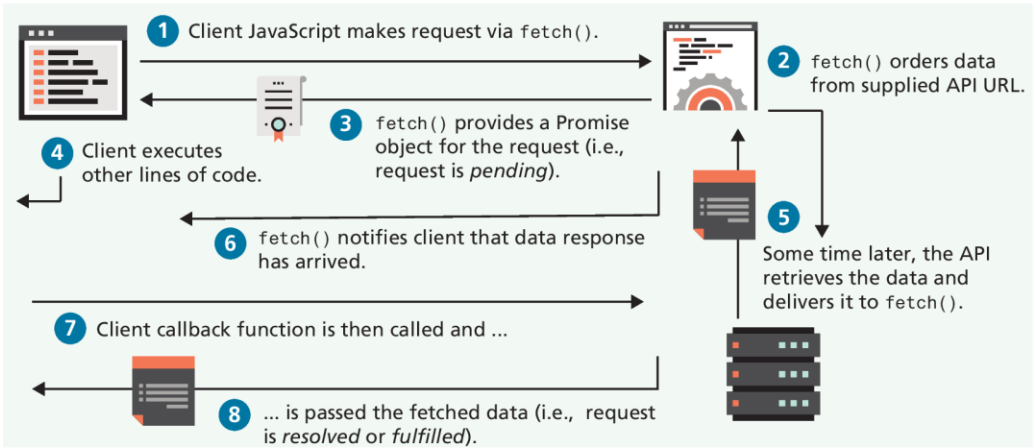
2 Once fetch data is received, hide animation and show box.

```
fetch(endpoint)
  .then(response => response.json())
  .then(data => {
     load.style.display = "none";
     box.style.display = "block";
     // do other stuff with data
     ...
});
```

Promises

A **Promise** is a placeholder for a value that we don't have now but will arrive later. Eventually, that promise will be completed and we will receive the data, or it won't, and we will get an error instea,





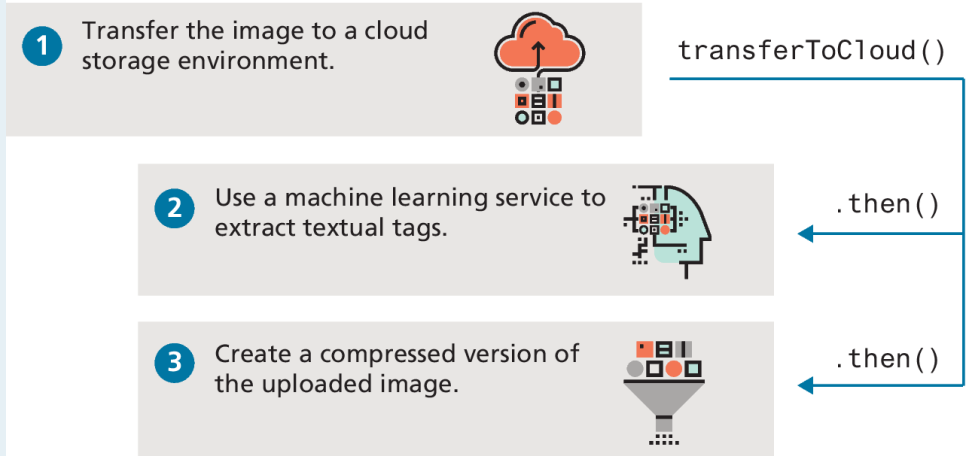
Creating a Promise

Creating a promise is quite simple: you simply instantiate a Promise object.
For promises to make some sense, we must first understand that the handler function passed to the Promise constructor must take two parameters: a **resolve**() function and a **reject**() function.

```
const promiseObj = new Promise( (resolve, reject) => {
 if (someCondition)
    resolve(someValue);
 else
    reject(someMessage);
 });
promiseObj
    .then( someValue => {
        // success, promise was achieved!
    })
    .catch( someMessage => {
        // oh no, promise was not satisfied!!
    });
```

A Promise example

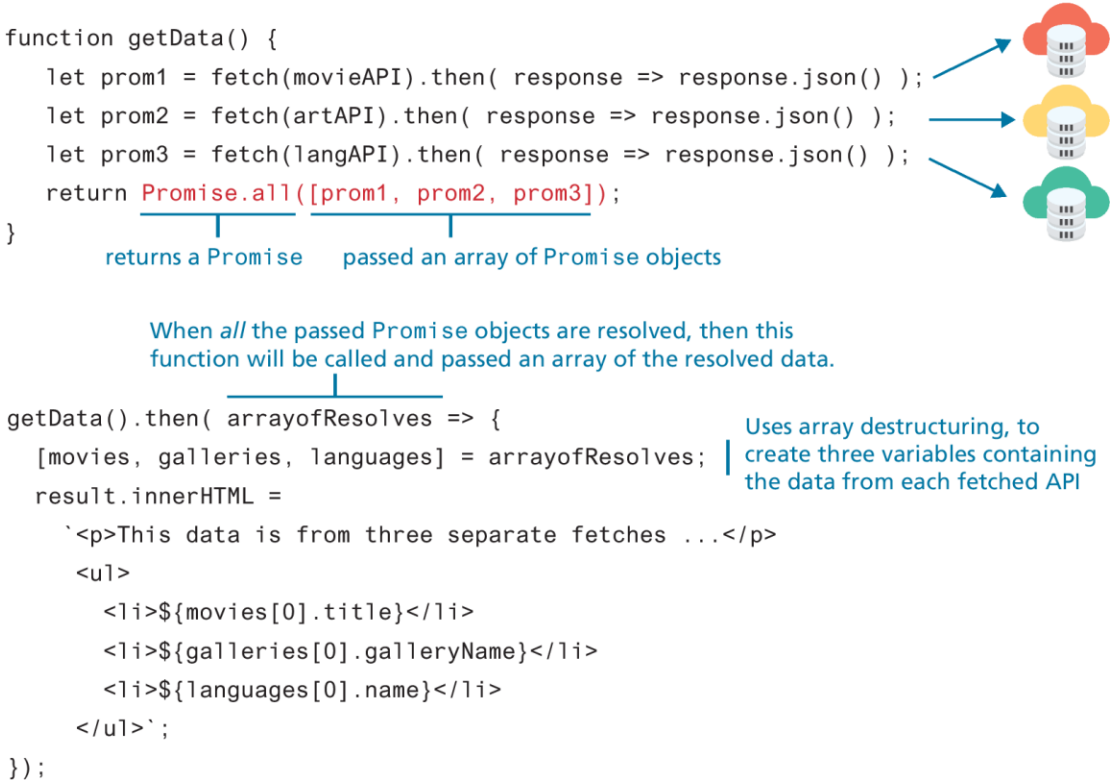
```
// promisified version of the transfer task
function transferToCloud(filename) {
    return new Promise( (resolve, reject) => {
        // just have a made-up AWS url for now
        let cloudURL ="http://.../makebelieve.jpg";
        // if passed filename exists then upload ...
        if ( existsOnServer(filename) ) {
            performTransfer(filename, cloudURL);
            resolve(cloudURL);
        } else {
            reject( new Error('filename does not exist'));
        }
    });
}
// use this function
transferToCloud(file)
.then( url => extractTags(url) )
.then( url => compressImage(url) )
.catch( err => logThisError(err) );
```



LISTING 10.10 Creating Promises

Executing multiple Promises in parallel

For executing multiple promises, you can make use of the **Promise.all()** method, which returns a single Promise when a group of Promise objects have been resolved.



Async and Await

ES7 introduced the **async...await** keywords that can both simplify the coding and even eliminate the typical nesting structure of typical asynchronous coding.

Recall this sample line from the earlier section on fetch?

```
let obj = fetch(url);
```

Content is contained in the variable obj is a **Promise**; the then() method of the Promise object needs to be called and passed a callback function that will use the data from the *promisified* function fetch.

Async and Await (ii)

The **await** keyword provides exactly that functionality, namely, the ability to treat asynchronous functions that return Promise objects as if they were synchronous.

```
let obj = await fetch(url);
```

Now, obj will contain whatever the resolve() function of the fetch() returns, which in this case is the response from the fetch. Notice that no callback function is necessary!

There is an important limitation with using the **await** keyword: it *must* occur within a function prefaced with the **async** keyword

Using 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**

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. Table 10.1 lists several of the more important browser APIs.

Web Storage API

The **Web Storage API** provides a mechanism for preserving non-essential state across requests and even across sessions. It comes in two varieties:

- **localStorage** is a dictionary of strings that lasts until removed from the browser.
- sessionStorage is also a dictionary of strings but only lasts as long as the browsing session.

To add a string to either involves calling the **setItem**() method of the **localStorage** or **sessionStorage** objects.

To retrieve a value from either simply requires using the **getItem**() method.

Web Storage API (ii)

The **Web Storage API** provides a mechanism for preserving non-essential state across requests and even across sessions. It comes in two varieties:

- **localStorage** is a dictionary of strings that lasts until removed from the browser.
- sessionStorage is also a dictionary of strings but only lasts as long as the browsing session.

To add a string to either involves calling the **setItem**() method of the **localStorage** or **sessionStorage** objects.

To retrieve a value from either simply requires using the **getItem**() method.

Web Speech API

The Web Speech API provides a mechanism for turning text into speech (sounds) and for turning speech (microphone input) into text.

To verbalize a string of text, you can simply make use of the **SpeechSynthesisUtterance** and **speechSynthesis** objects:

```
const utterance = new SpeechSynthesisUtterance('Hello world');
speechSynthesis.speak(utterance);
```

Some browsers provide different voices: for instance, U.S. male, U.S. female, U.K. male, etc. You can also adjust the speed and pitch of the speech.

GeoLocation

The **Geolocation API** provides a way for JavaScript to obtain the user's location (accuracy/availability dependent on permission and device)

```
if (navigator.geolocation) {
   navigator.geolocation.getCurrentPosition(hav
   eLocation,geoError);
} else {
// geolocation not supported or accepted
...
}
```

```
function haveLocation(position) {
   const latitude = position.coords.latitude;
   const longitude = position.coords.longitude;
   const altitude = position.coords.altitude;
   const accuracy = position.coords.accuracy;
   // now do something with this information
   ...
}
function geoError(error) { ... }
```

LISTING 10.13 Sample GeoLocation API usage

Using External APIs

An **external API** refers to objects with events and properties that perform a specific task that you can use in your pages.

Unlike browser APIs, these external APIs are **not** built into the browser but are external JavaScript libraries that need to be downloaded or referenced and added to a page via a <script> tag.

In this section, we will look at two of the most popular ones: the Google Maps API and the plotly API.

Google Maps

The Google Maps code used in the 1st edition of this book (2014) no longer worked by the time of the second edition (2017). That code no longer works now at the time of writing (2020). Hopefully, when you use this edition, the Google Maps code still works—but it might not!

- The point here is that external APIs are an externality, meaning that you have no control over them and that change over time should be expected with them.
- Use the latest Documentation from the API.

Overview | Maps JavaScript API | Google Developers

Google Maps (ii)

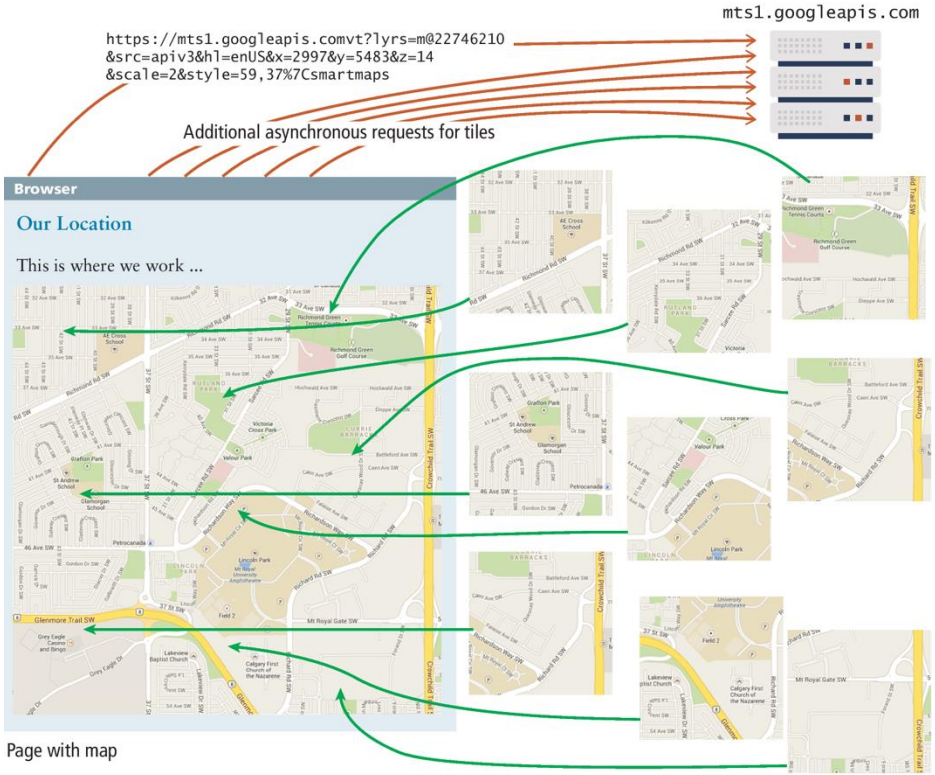
Notice that the API is made available to your page by referencing it in a **<script>** tag.

Our **initMap**() function creates a map object via the **google.maps.Map()** function constructor, which is defined within the library downloaded in our <script> element.

```
<head>
…
<style>
    #map {
        height: 500px;
    }
</style>
<script>
function initMap() {
    const map = new
    google.maps.Map(document.querySel
ector('#map'), {
    center: {lat: 51.011179,
                lng: -114.132866},
                zoom: 14
    });
}
</script>
<script
src="https://maps.googleapis.com/maps
/api/js?key=YOUR-API-
KEY&callback=initMap" async defer>
</script>
</head>
<body>
    Populating a Google Map
    <div id="map"></div>
</body>
</html>
```

LISTING 10.14 Webpage to output map centered on a location

Google Maps at work



Charting with Plotly.js

Charting is a common need for many websites. This section makes use of Plotly, which is open-source and available in a variety of other languages besides JavaScript.

Creating a simple chart is quite straightforward. Simply include the library, add an empty element that will contain the chart, and then make use of the newPlot() method, as shown in the following slide:

Charting with Plotly.js (ii)

```
<script>window.addEventListener("load", function() {
        const data = [
                                { x: [4,5,6,7,8,9,10,11],
                                 y: [23,25,13,15,10,13,17,20]
                                }
                        ];
        const layout = { title:'Simple Line Chart' };
        const options = { responsive: true };
        Plotly.newPlot("chartDiv", data, layout, options);
});
</script>
<script src="https://cdn.plot.ly/plotly-latest.min.js"></script>
<div id="chartDiv"></div>
```

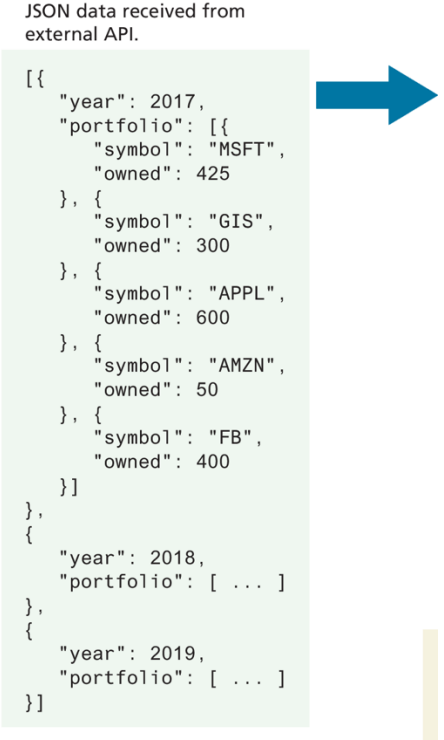
Plotly.js display a simple line chart

```
window.addEventListener("load", async () => {
   const url =
'https://www.randyconnolly.com/funwebdev/3rd/api/
                stocks/sample-portfolio.json';
   try {
      let data = await fetch(url)
      .then(async response => await response.json() );
         generateChart( transformDataForCharting(data) );
      }
   catch (err) { console.error(err) }

   function transformDataForCharting(data) {…}
   function generateChart(portfolioData) {…}
});
```

LISTING 10.15 Displaying a chart

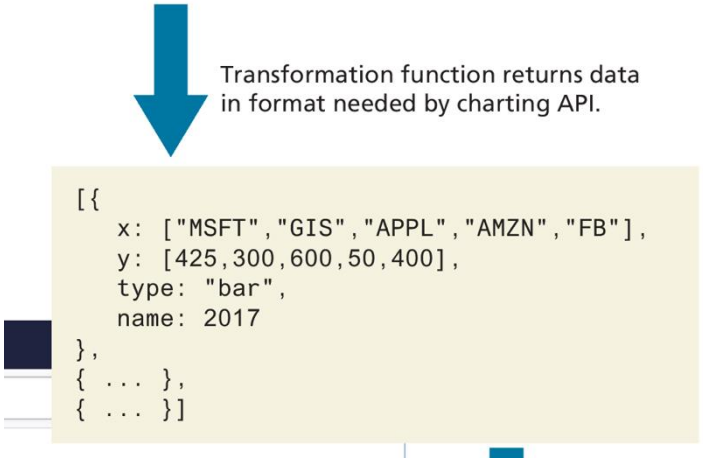
JSON data received from external API.



```
[{
   "year": 2017,
   "portfolio": [{
      "symbol": "MSFT",
      "owned": 425
   }, {
      "symbol": "GIS",
      "owned": 300
   }, {
      "symbol": "APPL",
      "owned": 600
   }, {
      "symbol": "AMZN",
      "owned": 50
   }, {
      "symbol": "FB",
      "owned": 400
   }]
},
{
   "year": 2018,
   "portfolio": [ ... ]
},
{
   "year": 2019,
   "portfolio": [ ... ]
}]
```

Plotly.js display a simple line chart (ii)

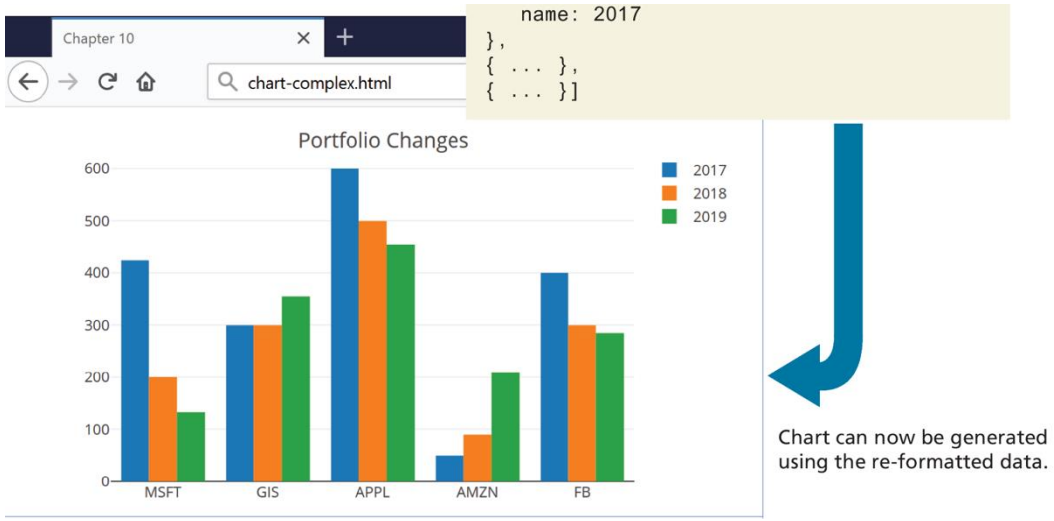
```
function transformDataForCharting(data) {
        const portfolioData = [];
        data.forEach((s) => {
                let trace = {};
                trace.x = [];
                trace.y = [];
                trace.type = 'bar';
                trace.name = s.year;
                for (let p of s.portfolio) {
                        trace.x.push(p.symbol);
                        trace.y.push(p.owned);
                }
        portfolioData.push(trace);
        });
        return portfolioData;
}
```



LISTING 10.15 Displaying a chart

Plotly.js display a simple line chart (iii)

```
/* generate the chart */
function generateChart(portfolioData) {
        const layout = {
        title: 'Portfolio Changes',
        barmode: 'group'
        };
        const options = {
        responsive: true
        };
        Plotly.newPlot("chartDiv",
        portfolioData, layout, options);
}
```



LISTING 10.15 Displaying a chart

Key Terms

async . . . await

asynchronous code

browser API

card

class

cross-origin resource

sharing (CORS)

external API

fetch()

forEach()

find()

filter()

Geolocation API

localStorage

map()

module

origin

offline first

Progressive Web Applications (PWA)

prototype

promise

service workers

threads

TypeScript

web API

Web Storage API

Copyright

This work is protected by United States copyright laws and is provided solely for the use of instructors in teaching their courses and assessing student learning. Dissemination or sale of any part of this work (including on the World Wide Web) will destroy the integrity of the work and is not permitted. The work and materials from it should never be made available to students except by instructors using the accompanying text in their classes. All recipients of this work are expected to abide by these restrictions and to honor the intended pedagogical purposes and the needs of other instructors who rely on these materials.