18

How to work with Ajax

This chapter shows how to use Ajax to update a web page without loading a new web page into the browser. Since Ajax is commonly used to get data from web services, this chapter also shows how to work with two web services that provide data that you can use for development and testing.

Introduction to Ajax

This chapter begins by describing how Ajax works. Then, it describes two data formats commonly used with Ajax.

How Ajax works

Ajax (*Asynchronous JavaScript and XML*) allows a web browser to update a web page with data from a web server without reloading the entire page. Figure 18-1 begins by showing Google's Auto Suggest feature because it's an example of a typical Ajax application. As you type the start of a search entry, Google uses Ajax to get the terms and links of items that match the characters that you have typed so far. Ajax does this without refreshing the page so the user doesn't experience any delays. This is sometimes called a "partial page refresh."

Modern browsers provide two ways to send an Ajax request to the web server and to process the data in the response that's returned from the server: the *XMLHttpRequest (XHR) object* and the *Fetch API*. This chapter describes both of these techniques.

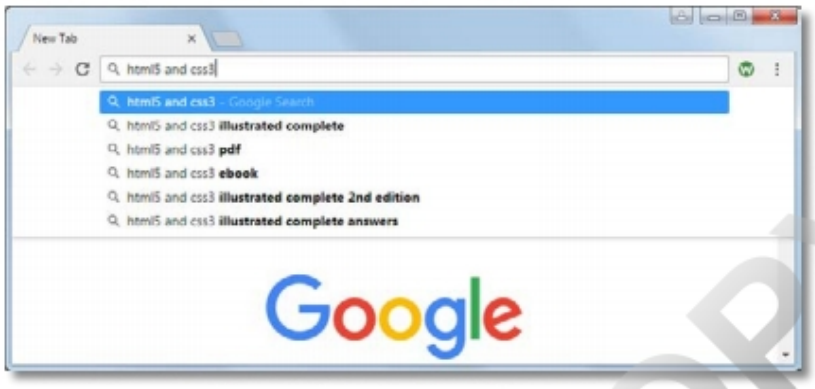
When working with Ajax, the browser uses JavaScript to send the request, parse the returned data, and modify the DOM so the page reflects the returned data. In many cases, a request includes data that tells the server what data to return.

On the web server, an application or script that's written in a server-side language like PHP typically returns the data that is requested. Often, these applications or scripts are part of a *web service* that provides an *Application Programming Interface* (*API*) that developers can use to get data from a website. Then, you can use that data to enhance your own web pages. For instance, many of the biggest and most popular websites provide APIs that let you make Ajax requests for data from their sites.

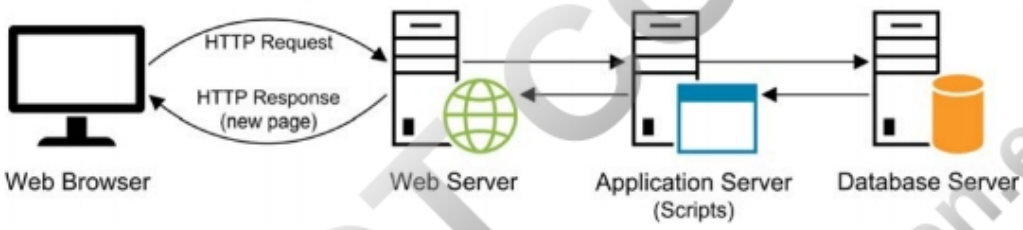
The two diagrams in this figure show how a normal HTTP request compares to an Ajax request. For a normal HTTP request, the browser makes an HTTP request for an entire page, the server returns an HTTP response for the page, and the browser loads the entire page. In contrast, with an Ajax request, the browser uses JavaScript to send an asynchronous request to the server, the server returns the requested data in the response, and the web page uses JavaScript to update the DOM with the new data. As a result, the browser doesn't need to reload the entire page.

Because Ajax is so powerful, it's used by many of the biggest and most popular websites. For instance, when you post a comment to Facebook, the comment just appears. And when you drag within a Google Map, the map is automatically adjusted. In neither case is the entire page reloaded.

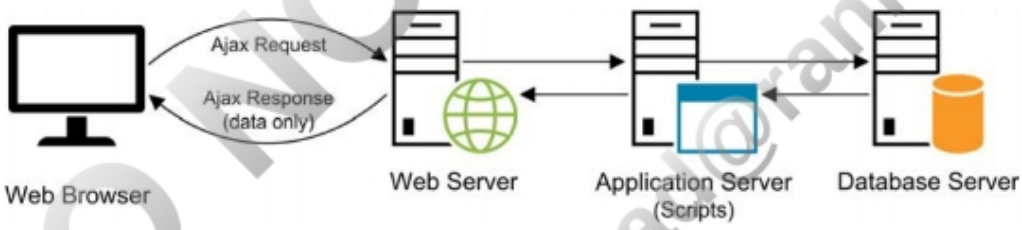
Google's Auto Suggest is an Ajax application



How a normal HTTP request is processed



How an Ajax request is processed



Description

- Unlike normal HTTP requests, *Ajax* (*Asynchronous JavaScript and XML*) requests update a page with data from a web server without needing to reload the entire page. This is sometimes known as a "partial page refresh."
- When working with Ajax, JavaScript sends the request, processes the response, and updates the DOM with the new data. As a result, the browser doesn't need to reload the entire page.
- To send an Ajax request, JavaScript can use a browser object known as the *XMLHttpRequest (XHR) object*, or it can use the *Fetch API*.
- Ajax requests are often made to *web services* that provide *Application Programming Interfaces* (*APIs*) that developers can use to get data from a website.
- Many popular websites provide APIs that let you use Ajax to get data from their sites.
- Many popular websites use Ajax to improve the way they function.

Figure 18-1 How Ajax works

Two common data formats for Ajax

Figure 18-2 presents the two most common data formats for Ajax applications: XML and JSON. Ajax was originally designed to be used with *XML* (*Extensible Markup Language*). That's why XML is part of the Ajax name.

XML is a format that works well for exchanging data across the Internet. In addition, XML is a markup language that works much like HTML. As a result, it's easy for programmers who have experience with HTML to understand. Because of that, XML was commonly used in the early days of Ajax.

Today, *JSON* (*JavaScript Object Notation*) is the most popular format for working with Ajax. JSON, pronounced "Jason", is similar to XML in that it's a format that works well for exchanging data across the Internet. However, it's less verbose than XML.

In this figure, the XML and JSON examples store the same data. However, the JSON example uses fewer characters. This makes it easier for humans to read the JSON. In addition, the JSON uses less memory when it's sent from the server to the client.

Most server-side languages provide functions for encoding data into JSON. For example, PHP provides the json_encode() function. That makes it easy for web services that run on a server to return JSON in Ajax responses. Then, on the client side, JavaScript provides functions for parsing the JSON that's returned from the client into a JavaScript object. For example, the json() function described later in this chapter parses the JSON that's returned in an Ajax response into a native JavaScript object.

Although XML and JSON are the two most popular data formats for working with Ajax, they aren't the only ones that are supported. If you want, it's possible to use plain text or other formats such as HTML, YAML, or CSV.

Two common data formats for Ajax

Format	Description	File extension
XML	Extensible Markup Language	xml
JSON	JavaScript Object Notation	json

XML data

```
<?xml version="1.0" encoding="utf-8"?>
<management>
    <teammember>
        <name>Agnes</name>
        <title>Vice President of Accounting</title>
        <bio>With over 14 years of public accounting ... </bio>
    </teammember>
    <teammember>
        <name>Wilbur</name>
        <title>Founder and CEO</title>
        <bio>While Wilbur is the founder and CEO ... </bio>
    </teammember>
</management>
```

JSON data

```
{"teammembers":[
    {
        "name":"Agnes",
        "title":"Vice President of Accounting",
        "bio":"With over 14 years of public accounting... "
    },
    {
        "name":"Wilbur",
        "title":"Founder and CEO",
        "bio":"While Wilbur is the founder and CEO ... "
    }
]}
```

Description

- The two most common data formats for working with Ajax are XML and JSON.
- Both *XML* (*Extensible Markup Language*) and *JSON* (*JavaScript Object Notation*) are formats that use text to store and transmit data.
- Most server-side languages provide methods for encoding data into JSON.
- JavaScript provides methods for parsing the JSON that's returned from a web service into a JavaScript object.
- JSON is less verbose than XML, so it uses less memory when being sent from the server to the client.

Figure 18-2 Two common data formats for Ajax

The JSON Placeholder API

Ajax requests typically get data from a web service. That's why figure 18-3 begins by presenting some information about a web service named JSON Placeholder. This web service provides an API that accesses fake data that's in JSON format. This data mimics the kind of data that's typically returned by real web services. Using an API like this allows you to practice making Ajax calls without having to set up accounts or worry about other implementation details.

The table in this figure describes the type of data you can request from the JSON Placeholder web service. This data includes information about users, blog posts, comments, and so on. Most of the data is related to other data. For example, comments are related to a blog post, and photos are related to an album.

To make a GET request with the JSON Placeholder API, you use the URL for the web service with the desired resource added to the end. For example, to get data for all users, you use the URL like the one shown in the first example. This returns the JSON data shown in the second example. However, if you want to get the data for a single user, you can add the id for the user to the end of the URL like this:

```
https://jsonplaceholder.typicode.com/users/1
```

You can use this API to simulate other kinds of HTTP requests such as POST, PUT, and DELETE requests. This chapter doesn't show how to make those types of requests, but the documentation for this web service does.

The JSON Placeholder web service

https://jsonplaceholder.typicode.com

The fake data that's available from this web service

Resource	Description
/users	10 users with data such as name, username, and email address.
/posts	100 blog posts with each one related to a specific user.
/comments	500 comments with each one related to a specific blog post.
/albums	100 photo albums with each one related to a specific user.
/photos	5000 simple photos of various colors with each one related to a specific album. This includes one photo that's 600x600 pixels and a thumbnail that's 150x150 pixels.
/todos	200 tasks with each one related to a specific user.

A URL that returns data

```
https://jsonplaceholder.typicode.com/users
```

Some of the JSON that's returned

```
[
  {
    "id": 1,
    "name": "Leanne Graham",
    "username": "Bret",
    "email": "Sincere@april.biz",
    ...
    }
  },
  ...
  {
    "id": 10,
    "name": "Clementina DuBuque",
    "username": "Moriah.Stanton",
    "email": "Rey.Padberg@karina.biz",
    ...
    }
  }
]
```

Description

- The JSON Placeholder web service provides an API that accesses fake data that's in JSON format. This data mimics the kind of data that's typically returned by real web services.
- Using the JSON Placeholder API allows you to practice making Ajax calls without having to set up accounts or worry about other implementation details.
- Much of the data returned by the JSON Placeholder API uses Latin text that's been used in printing and typesetting since the 1500s.

Figure 18-3 The JSON Placeholder API

How to make a single Ajax request

This chapter continues by showing two techniques for making a single Ajax request. First, this chapter shows how to use the XMLHttpRequest object. Then, it shows how to use the Fetch API. Although using the XMLHttpRequest object was common in the early days of Ajax, it has some drawbacks that have led to the rise of the newer Fetch API.

How to use the XMLHttpRequest object

Figure 18-4 presents some of the *members* (methods, properties, and events) of the XMLHttpRequest object. To start, every Ajax request uses the open() and send() methods. The open() method opens a connection for a request. Its first parameter specifies whether the request is a GET or POST request, and the second parameter provides the URL for the request. In a production application, the URL typically specifies a web service.

After the open() method opens a connection, the send() method sends the request. If necessary, the first parameter of this method can specify any data that's sent to the server along with the request. Typically, the server uses this data to filter the data that it returns.

The responseType property specifies the format for the data that's returned by the server, and the response property gets that data in the specified format. Then, the readyState property indicates the state of the request, and the status property provides the status code that's returned by the server.

The two events determine what happens when the Ajax request completes. The onreadystatechange event can be used to processes the data that's returned, and the onerror event can be used to handle any errors that might occur.

To work with these events, you assign *callback functions* to them. This is similar to assigning an event handler function by passing it to the addEventListener() method. Callback functions are essential to asynchronous JavaScript. That's because asynchronous code can't return a value or throw an exception the way synchronous code can. Instead, you must define a callback function that runs when the return value is ready or the error occurs.

The example in this figure shows how to use the XMLHttpRequest object to request data from the JSON Placeholder API. To start, this code creates a new XMLHttpRequest object. Then, it assigns a string of "json" to the responseType property to specify that the response should be in JSON format.

After setting the response type, this code assigns a callback function to the onreadystatechange event. This callback function begins by checking whether the readyState property is 4 (DONE) and the status property is 200 (SUCCESS). If so, it displays the JSON data in the response in the console. In a more realistic application, this code would parse the response data and update the DOM. For now, though, this simple code shows how the XMLHttpRequest object works.

After assigning the first callback function, this code assigns a callback function to the onerror event. This callback function displays an error message in the console. Again, though, in a more realistic application it would typically log the error or update the DOM to notify the user of the error.

Common members of the XMLHttpRequest object

Method	Description
`open(method, url)`	Opens a connection for a request. The parameters let you set the method to GET or POST and set the URL for the request.
`send([data])`	Starts the request. This method can include data that gets sent with the request. This method must be called after a request connection has been opened.
Property	**Description**
`responseType`	A string that indicates the format of data the response contains. Common values are "text", "json", "document", and "blob".
`response`	The content that's returned from the server. The format of the response depends on the value of the responseType property.
`readyState`	A numeric value that indicates the state of the current request: 0 is UNSENT, 1 is OPENED, 2 is HEADERS_RECEIVED, 3 is LOADING, and 4 is DONE.
`status`	The status code returned from the server in numeric format. Common values include 200 for SUCCESS and 404 for NOT FOUND.
Event	**Description**
`onreadystatechange`	An event that occurs when the state of the request changes.
`onerror`	An event that occurs when an error occurs.

Code that uses the XMLHttpRequest object to get and display user data

```
const xhr = new XMLHttpRequest();
xhr.responseType = "json";

xhr.onreadystatechange = () => {
    if (xhr.readyState == 4 && xhr.status == 200) {
        console.log(xhr.response);
    }
};
xhr.onerror = e => console.log(e.message);

xhr.open("GET", "https://jsonplaceholder.typicode.com/users");
xhr.send();
```

The data displayed in the console

```
▼(10) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}]
  ▶0: {id: 1, name: "Leanne Graham", username: "Bret", email: "Sincere@april.biz", address: {…}, …}
  ▶1: {id: 2, name: "Ervin Howell", username: "Antonette", email: "Shanna@melissa.tv", address: {…}, …}
  ▶2: {id: 3, name: "Clementine Bauch", username: "Samantha", email: "Nathan@yesenia.net", address: {…}, …}
  ▶3: {id: 4, name: "Patricia Lebsack", username: "Karianne", email: "Julianne.OConner@kory.org", address: {…}…
```

Description

- A *member* of an object is one of its methods, properties, or events.
- To handle the events raised by the XMLHttpRequest object, you can assign a *callback function* to the event. Then, the callback function runs when the event occurs. Callback functions are essential to asynchronous JavaScript.

Figure 18-4 How to make an Ajax request using the XMLHttpRequest object

At this point, the code has set up the callback functions for the request, but it hasn't sent the request. To do that, this code uses the open() method to open the connection for the request. This statement sets the method for the request to GET and provides the URL for the JSON data. Then, the last statement uses the send() method to send the request.

How to use the Fetch API

The Fetch API provides methods and objects for making Ajax requests. For new development, it's generally considered a best practice to use the Fetch API, not the older XMLHttpRequest object.

The first table in figure 18-5 summarizes the fetch() method of the Fetch API. It accepts the URL for a GET request and returns a Promise object, which represents the eventual return value of the asynchronous request. In this case, that eventual return value is a Response object, which represents an HTTP response.

A Promise object (or *promise*) has three states. When a Promise object is first created, it's *pending*. When the request returns its value, the promise is *fulfilled*. However, if an error occurs during the request, the promise is *rejected*.

There are two more terms you need to understand to work with Promise objects. First, a promise that is no longer pending is considered *settled*. This is true whether the promise is fulfilled or rejected. Second, a promise can be *resolved* without being fulfilled. For example, if a promise returns another promise, the original promise is resolved, even though the requested data isn't returned yet and so the promise isn't fulfilled.

The tables in this figure present the methods that you can use to make an Ajax request with the Fetch API. Because each of these methods returns a Promise object, they can be chained with each other. Of these methods, the then() method is the most difficult to understand because it accepts a callback function that executes when the promise is resolved. The parameter that's passed to this callback function is the eventual return value of that promise. The catch() method works similarly, but it executes when the promise is rejected.

The example below the tables shows how to use these methods. To start, this code passes a URL for a GET request to the fetch() method, which returns a promise that resolves when the status and headers of the HTTP response are received. However, at this point, the data of the HTTP response isn't received yet. So, this first promise is resolved because it has returned a Response object, but it isn't fulfilled, because the requested data isn't received yet.

The first then() method accepts a callback that executes when the promise from the fetch() method resolves. This callback accepts the Response object and calls its json() method, which returns a promise that resolves to a JavaScript object that's created from the JSON stored in the response. At that point, the original promise is fulfilled.

The second then() method accepts a callback that executes when the promise from the json() method resolves. This callback accepts a JavaScript object named json and displays that object in the console.

The catch() method accepts a callback that executes when the promise is rejected. This callback function displays an error message in the console.

One method of the Fetch API

Method	Description
`fetch(url)`	Makes an asynchronous GET request to the specified URL. Returns a Promise object that eventually returns a Response object.

Two methods of the Promise object

Method	Description
`then(callback)`	Registers the callback function to execute when the promise is resolved. The callback function receives a single parameter, which is the eventual return value of the asynchronous request. Returns a Promise object.
`catch(callback)`	Registers the callback function to execute when the promise is rejected. The callback function receives a single parameter, which is usually an Error object. Returns a Promise object.

One method of the Response object

Method	Description
`json()`	Returns a Promise object that eventually resolves to a JavaScript object that's created from the JSON that's returned by the asynchronous request.

Code that uses the Fetch API to get and display user data

```
fetch("https://jsonplaceholder.typicode.com/users")
    .then( response => response.json() )
    .then( json => console.log(json) )
    .catch( e => console.log(e.message) );
```

The data displayed in the console

Description

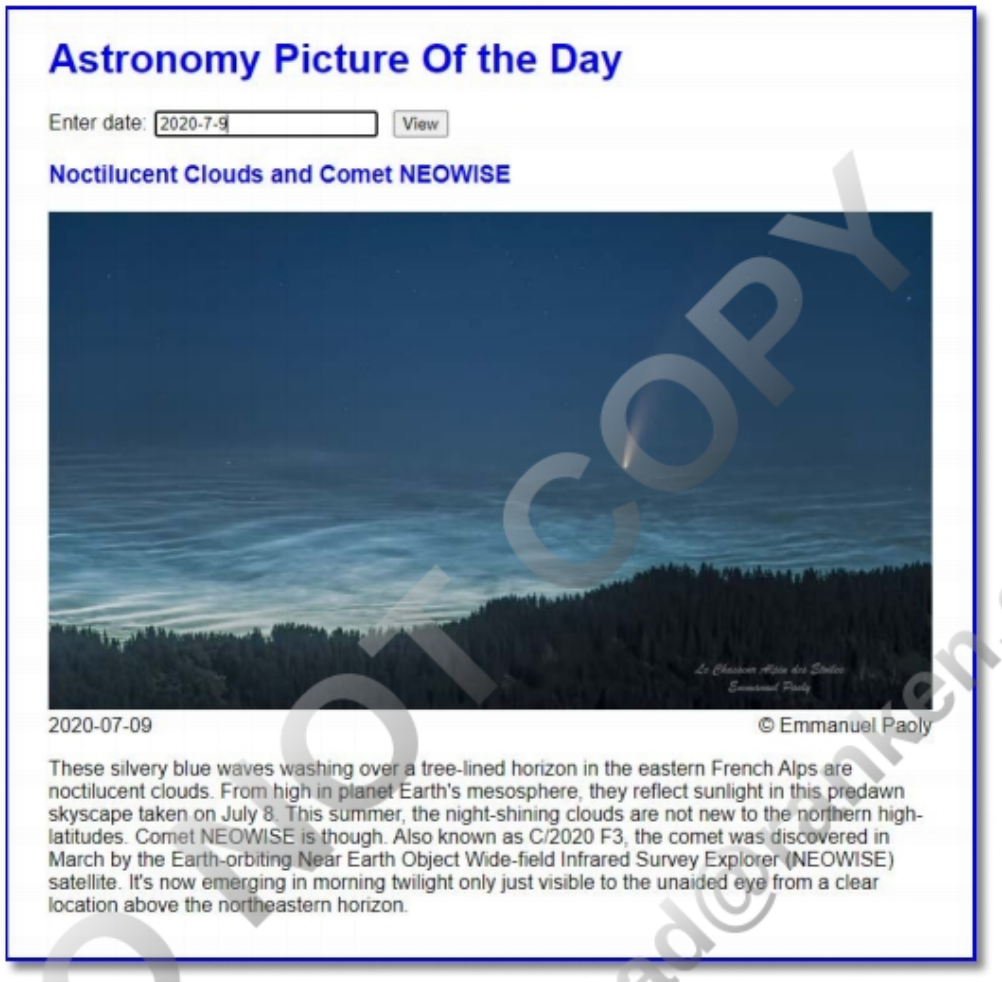
- For new development, it's generally considered a best practice to use the Fetch API, not the older XMLHttpRequest object.

Figure 18-5 How to make an Ajax request with the Fetch API

The Astronomy Picture Of the Day application

Figure 18-6 presents an Astronomy Picture of the Day (APOD) application that displays an image or video for any date between June 16, 1995 and today. In addition, it displays some related data such as title, date, copyright, and explanation. To make this possible, the APOD application gets its data from the API that's available from NASA's APOD web service.

The Astronomy Picture Of the Day application



Description

- The Astronomy Picture of the Day (APOD) application allows the user to enter a date and click the View button. Then, it displays an image or video for the specified date. In addition, it displays some related data such as title, date, copyright, and explanation.
- The APOD application gets its data from the API that's available from NASA's APOD web service.
- When it loads, the APOD application displays the current date in the "Enter date" text box.

Figure 18-6 The Astronomy Picture Of the Day application (part 1)

The HTML and CSS

Part 2 of figure 18-6 begins by presenting some of the HTML and CSS for the application. To start, the HTML displays the title of the application as well as a text box and a View button that allow the user to enter a date. Here, the text box has an id of "date", and the View button has an id of "view_button". Below the View button, this HTML includes a div element with an id of "display". This is where the application displays the picture and its related data if the Ajax request is fulfilled. It's also where the application displays any error messages.

The CSS aligns the label and input elements. In addition, it provides the classes named error and right. The application uses the error class to display error messages in red, and it uses the right class to align the copyright data with the right side of its container element.

The NASA APOD API

After the CSS, this figure shows the URL for NASA's APOD API. To use this API, you need to supply an api_key parameter as described in the table and shown by the example URL.

When you're first getting started, the api_key parameter can specify a value of DEMO_KEY. However, this key only supports 30 requests per hour or 50 requests per day. As a result, if you need to make more requests than that, you should get your own API key. To do that, you can visit this URL:

https://api.nasa.gov/

This URL also provides information about using the APOD API and other similar NASA APIs.

If you don't supply a date parameter, the APOD API returns the JSON data for the current date. However, the APOD application presented in this figure allows the user to specify a date. As a result, it must supply a date parameter as shown by the example URL.

The HTML

```
<body>
    <main>
        <h1>Astronomy Picture Of the Day</h1>
        <div>
            <label for="date">Enter date:</label>
            <input type="text" name="date" id="date">
            <input type="button" id="view_button" value="View">
        </div>
        <div id="display"></div>
    </main>
    <script src="https://code.jquery.com/jquery-3.4.1.min.js"></script>
    <script src="apod.js"></script>
</body>
```

Some of the CSS

```
div {
    margin-bottom: 1em;
}
label {
    display: inline-block;
    width: 5em;
}
input {
    margin-right: 0.5em;
}
.error {
    color: red;
}
.right {
    float: right;
}
```

The URL for NASA's Astronomy Picture of the Day (APOD) API

https://api.nasa.gov/planetary/apod

Parameters

Parameter	Description
api_key	The API key for the web service. You can specify a value of DEMO_KEY to explore the API. However, this key only supports 30 requests per IP address per hour or 50 requests per day. As a result, you should sign up for your own API key if you plan to use the API extensively.
date	The date of the APOD data to retrieve. This date must be in YYYY-MM-DD format. If you don't specify this parameter, the API retrieves the data for today's date.

Example URL

```
https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY&date=2020-07-22
```

Figure 18-6 The Astronomy Picture Of the Day application (part 2)

The JavaScript

Part 3 of figure 18-6 shows the JavaScript for the APOD application. To start, this JavaScript defines a helper function named getDateString(). This function accepts a Date object and returns a corresponding date string in the YYYY-MM-DD format that's required by the APOD API.

After the getDateString() function, this code defines a function named displayPicture(). This function displays the JSON data that's returned by an Ajax call to the APOD API. To start, this code defines a variable to store the HTML that's displayed. Then, it checks whether the data object that's passed to this function has an error property. If so, an error occurred. As a result, this code notifies the user by setting the HTML to a span element that displays the message that's stored in the error property.

After checking the error property, this code checks whether the data object has a code property. If so, the request succeeded but there's a problem with the data. As a result, the JavaScript notifies the user by setting the HTML to a span element that displays the message that's stored in the code property.

If the data object doesn't contain an error property or a code property, the Ajax request succeeded. In that case, the JavaScript gets data from the data object to set the HTML so it displays the title for the image or video, the image or video itself, the date, the copyright (if one exists), and the explanation.

After the if statement, the last statement in this function displays the HTML in the div element that has the id of "display". If the HTML contains an error message, this displays the error message. Otherwise, it displays the image or video and its related data.

After the displayPicture() function, this code defines a function named displayError(). This function accepts an Error object and sets the HTML to a span element that displays the message that's stored in the Error object. Then, it displays this span element in the div element that has the id of "display".

The JavaScript

```
"use strict";

// returns date string in YYYY-MM-DD format
const getDateString = date =>
    `${date.getFullYear()}-${date.getMonth() + 1}-${date.getDate()}`;

const displayPicture = data => {
    let html = "";
    if(data.error) {       // error - display message
        html += `<span class="error">${data.error.message}</span>`;
    }
    else if (data.code) {  // problem - display message
        html += `<span class="error">${data.msg}</span>`;
    }
    else {                 // success - display image/video data
        html += `<h3>${data.title}</h3>`;
        const width = 700;
        switch (data.media_type) {
            case "image":
                html += `<img src="${data.url}" width="${width}"
                        alt="NASA photo">`;
                break;
            case "video":
                html += `<iframe src="${data.url}"
                         frameborder="0"
                         allowfullscreen></iframe>`;
                break;
            default:
                html += `<img src="images/notavailable.png"
                        width="${width}" alt="NASA photo">`;
        }

        // date and copyright
        html += `<div>${data.date}`;
        if (data.copyright) {
            html += `<span class="right">&copy; ${data.copyright}</span>`;
        }
        html += "</div>";

        // explanation
        html += `<p>${data.explanation}</p>`;
    }

    // display HTML
    $("#display").html(html);
};

const displayError = error => {
    let html = `<span class="error">${error.message}</span>`;
    $("#display").html(html);
};
```

Figure 18-6 The Astronomy Picture Of the Day application (part 3)

Part 4 of figure 18-6 shows the ready() event handler for the application. To start, this event handler gets today's date and converts it to a date string in the YYYY-MM-DD format. Then, it sets that date string in the "Enter date" text box and moves the focus to that text box. As a result, when the application loads, it displays the current date in the "Enter date" text box.

After setting up the "Enter date" text box, this code assigns a click() event handler to the View button. Within this event handler, the code begins by getting the date string from the text box. This date string may be today's date or it may be any other date entered by the user. Then, it converts this date string to a Date object.

After creating a Date object from the string, the code checks whether the Date object is invalid. If so, it displays a message that indicates that the date is invalid. Otherwise, it continues by using the getDateString() function to make sure the Date object is in the YYYY-MM-DD format. This is necessary because it's possible for a user to enter a date that's in another format such as MM/DD/YY.

After making sure the date string is in the necessary format, this code builds the URL for the Ajax request. In this figure, the code uses an API key value of DEMO_KEY. This key is appropriate for testing, but a production application should have its own key, which would be a long string of letters and numbers. In addition, this code sets the date parameter of the request to the date string.

After building the URL for the API request, this code uses the methods of the Fetch API to make the Ajax request. If this request is successful, the second then() method displays the picture by passing the JavaScript object that's returned by the json() method to the displayPicture() function defined in part 3. This JavaScript object is created from the JSON stored in the Ajax response. Otherwise, the catch() method displays an error by passing an Error object to the displayError() function defined in part 3.

The last statement of the click() event handler moves the focus to the "Enter date" text box. As a result, after the user clicks the View button, the focus is always moved to this text box, regardless of whether the application displays an error or the picture.

The JavaScript (continued)

```
$(document).ready( () => {

    // on load, get today's date in YYYY-MM-DD format
    const today = new Date();
    let dateStr = getDateString(today);

    // set today's date in textbox
    const dateTextbox = $("#date");
    dateTextbox.val(dateStr);
    dateTextbox.focus();

    $("#view_button").click( () => {

        // get date from textbox
        dateStr = $("#date").val();
        const dateObj = new Date(dateStr);

        if (dateObj == "Invalid Date") {
            const msg = "Please enter valid date in YYYY-MM-DD format."
            $("#display").html(`<span class="error">${msg}</span>`);
        }
        else {
            // make sure date string is in proper format
            dateStr = getDateString(dateObj);

            // build URL for API request
            const domain = `https://api.nasa.gov/planetary/apod`;
            const request = `?api_key=DEMO_KEY&date=${dateStr}`;
            const url = domain + request;

            fetch(url)
                .then( response => response.json() )
                .then( json => displayPicture(json) )
                .catch( e => displayError(e) );
        }
        $("#date").focus();
    });
});
```


Figure 18-6 The Astronomy Picture Of the Day application (part 4)

How to make multiple Ajax requests

Making a single Ajax request as shown so far in this chapter allows you to add all sorts of nifty functionality to your websites. However, if you need to make additional Ajax requests based on the data that's returned by a previous Ajax response, your code becomes more complicated.

That's especially true if you're using the XMLHttpRequest object. In that case, you need to nest those additional requests within the callback function for the initial request. This can lead to repetitive code that includes callback functions nested several layers deep, a situation some developers refer to as "callback hell". Fortunately, the Fetch API provides several ways to solve "callback hell".

The XMLHttpRequest object and "callback hell"

The code example in figure 18-7 illustrates the problem that occurs if you use the XMLHttpRequest object to make nested Ajax requests. Here, the code starts by making an Ajax request for the photo with the id of 1. Then, the callback function for that request makes a second Ajax request. This request uses the photo object that's returned by the first Ajax request to get data for the album the photo is in. To do that, it uses the albumId property of the photo object.

In turn, the callback function for the second Ajax request makes a third Ajax request. This request uses the album object returned by the second request to get data for the user that created the album. To do that, it uses the userId property of the album object. Then, when these objects are ready, the code in the callback function uses them to add HMTL elements to the DOM.

In this figure, the nested callbacks and repetitive code makes the example difficult to read and maintain. Also, each XMLHttpRequest object should have its own callback function for its onerror event, which would make this code even harder to read and maintain.

The jQuery library has an $.ajax() method that can reduce some of the repetitive code shown in this example. However, that method uses callback functions in a similar way, so it doesn't help with "callback hell". Fortunately, the new Fetch API provides a way to make Ajax requests that solves most of these drawbacks.