History API

Quick Guides for Masterminds

J.D Gauchat
www.jdgauchat.com

Cover Illustration by **Patrice Garden**
www.smartcreativz.com

1st Edition 2018

What is Inside

This guide will teach you how to use the History API to control the navigation history. After reading this guide, you will know how to navigate back and forward in the browser's history, and how to modify and add new URLs from JavaScript.

About this Guide

This guide is a collection of excerpts from the book HTML5 for Masterminds. The information included in this guide will help you understand a particular aspect of web development, but it will not teach you everything you need to know to develop a website or a web application. If you need a complete course on web development, read our book HTML5 for Masterminds. For more information, visit our website at www.formasterminds.com.

What Do You Need

This guide assumes that you have a basic knowledge of HTML, CSS and JavaScript, and you know how to create files and upload them to a server. If you don't know how to program in HTML, CSS or JavaScript, you can download our guides Introduction to HTML, Introduction to CSS, and Introduction to JavaScript. For a complete course on web development, read our book HTML5 for Masterminds.

IMPORTANT: We recommend you to execute the examples in this guide with the latest versions of Google Chrome and Mozilla Firefox (www.google.com/chrome and www.mozilla.com). You can also check the state of the current implementations at www.caniuse.com. To find examples, resources, links and videos, visit our website at www.formasterminds.com.

The Basics: To create the files for the examples of this guide, you can use a text editor (Notepad or Text Edit), or a professional editor like Atom (www.atom.io). If you do not have a server to test the files, you can install a server in your computer with a package like MAMP (www.mamp.info). For more information, read our free guide Web Development.

Table of Contents

History API

History

The browser history is a list of all the web pages (URLs) visited by the user in one single session. It is what makes navigation possible. Using the navigation buttons on the left or right of the navigation bar in every browser, we can move through the list back and forward and load previous documents.

Navigation

With the browser's arrows, we can load a web page we visited before or go back to the last one, but sometimes it may be useful to navigate through the browsing history from inside the document. For this purpose, browsers include the History API. This API provides properties and methods to manipulate the history and manage the list of URLs it contains. The following are the properties and methods available to simulate the navigation buttons from JavaScript code.

length—This property returns the number of entries in the browsing history (the total of URLs on the list).

back()—This method takes the browser one step back in the browsing history (emulating the left arrow).

forward()—This method takes the browser one step forward in the browsing history (emulating the right arrow).

go(steps**)**—This method takes the browser back or forward the specified steps in the browsing history. The attribute may be a negative or positive value according to the direction we choose.

The API is defined by the **History** object, accessible from a property of the **Window** object called **history**. Every time we want to read the API's properties or call its methods, we have to do it from this property, as in **window.history.back()** or **history.back()**.

Listing 1: *Navigating back in the browser history*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>History API</title>
  <script>
```

```
    function goback() {
      window.history.back();
    }
  </script>
</head>
<body>
  <section>
    <button type="button" onclick="goback()">Go to the
previous page</button>
  </section>
</body>
</html>
```

The document of Listing 1 illustrates how simple it is to implement these methods. The document defines a button that calls the **goback()** function when the user clicks on it. In this function, we call the **back()** method of the History API to take the user back to the previous page.

Do It Yourself: Create a new HTML file with the document of Listing 1. Visit a website you know and then load the document in your browser. Click on the button. The browser should load the previous website and show it on the screen.

URLs

It is common practice to program small applications that retrieve information from a server and show it within the current document without refreshing the page or loading a new one. Users interact with websites and applications from the same URL, receiving information, entering data, and getting the results printed on the same page. However, the way browsers keep track of the user's activity is through URLs. URLs are the data inside the browsing history, the addresses that indicate where the user is located. Because new web applications avoid the use of URLs to point to the user's location on the website, important steps are lost in the process. Users can update data on a web page dozens of times but leave no trace in the browser history to indicate the steps followed and help them go back. To solve this issue, the History API includes properties and methods that modify the URL in the location bar as well as the browser history. The following are the most frequently used.

state—This property returns the value of the state of the current entry.

pushState(state, title, url**)**—This method creates a new entry in the browser history. The **state** attribute declares a value for the state of the entry. It is useful to identify the entry later, and it may be specified as a string or as a JSON object. The

title attribute is the title of the entry, and the **url** attribute is the URL of the entry we are generating (this value will replace the current URL in the navigation bar).

replaceState(state, title, url**)**—This method works exactly like **pushState(),** but it does not generate a new entry. Instead, it replaces the information of the current one.

If we want to create a new entry in the browser history and change the URL inside the navigation bar, we have to use the **pushState()** method. The following example shows how it works.

Listing 2: *Creating a document to experiment with the History API*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>History API</title>
  <link rel="stylesheet" href="history.css">
  <script src="history.js"></script>
</head>
<body>
  <section id="maincontent">
    <p>This content is never refreshed</p>
    <p><span id="url">page 2</span></p>
  </section>
  <aside id="databox"></aside>
</body>
</html>
```

In this document, we have included permanent content inside a **<section>** element identified as **maincontent**, a text that we will turn into a link to generate the second page of the website, and an element identified as **databox** to show the page's exclusive content. The following are the styles for the document.

Listing 3: *Defining the styles for the boxes and the* <span> *elements*

```
#maincontent {
  float: left;
  padding: 20px;
  border: 1px solid #999999;
}
#databox {
  float: left;
  width: 500px;
  margin-left: 20px;
  padding: 20px;
  border: 1px solid #999999;
}
#maincontent span {
  color: #0000FF;
  cursor: pointer;
}
```

What we are going to do in this example is to add a new entry with the **pushState()** method and update the content without refreshing the page or loading another document.

Listing 4: *Generating a new URL and content*

```
function initiate() {
  databox = document.getElementById("databox");
  url = document.getElementById("url");
  url.addEventListener("click", changepage);
}
function changepage() {
  databox.innerHTML = "The url is page2";
  history.pushState(null, null, "page2.html");
}
window.addEventListener("load", initiate);
```

In the code in Listing 4, we add a listener for the **load** event to the window to execute a function called **initiate()** as soon as the document is loaded. When the event is fired, this function creates a reference to the **databox** element and adds a listener for the **click** event to the **<span>** element. Every time the user clicks on the text inside **<span>,** the **changepage()** function is called. This function performs two tasks: it updates the content of the page with new information and inserts a new URL in the browser history. After the function is executed, the **databox** shows the text "The url is page2", and the URL of the main document is replaced in the location bar by the URL page2.html.

Do It Yourself: Create a new HTML file with the document of Listing 2, a CSS file called history.css with the styles in Listing 3, and a JavaScript file called history.js with the code in Listing 4. Upload all the files to your server or local server and open the document in your browser. Click on the text "page 2" and check how the URL in the location bar changes to the one generated by the code.

IMPORTANT: The URLs generated from these methods are fake URLs in the sense that browsers never check for the validity of these addresses and the existence of the document that they are pointing to. It is your responsibility to make sure that these fake URLs are in fact valid and useful.

The state Property

What we have done so far is to manipulate the browser history. We made the browser believe that the user visited a URL that, at this point, does not exist. After the "page 2" link is clicked, the fake URL page2.html was shown in the navigation bar, and new content was inserted in the **databox**, everything without refreshing or loading another page. But at this point the browser does not consider the new URL a real document. If we go back in our browsing history using the navigation buttons, the URL changes from the new one to the one corresponding to the main document, but the content of the document does not change at all. We need to detect when the fake URLs are revisited and perform the right modifications in the document to show the corresponding state.

We mentioned before the existence of the **state** property. The value of this property may be set during the generation of a new URL, and this is the way we can later identify what is the current URL. The API provides the following event to work with this property.

popstate—This event is fired when a URL is revisited or when the document has been loaded. It provides the **state** property with the value of the state declared when the URL was generated with the **pushState()** or **replaceState()** methods. This value is **null** when the URL is real unless we changed it before through the **replaceState()** method, as we will see next.

In the following example, we improve the previous code by implementing the **popstate** event and the **replaceState()** method to detect which URL the user is currently requesting.

***Listing 5:** Keeping track of the user's location in the browser history*
```
function initiate() {
  databox = document.getElementById("databox");
  url = document.getElementById("url");
  url.addEventListener("click", changepage);
  window.addEventListener("popstate", newurl);
  history.replaceState(1, null);
}
function changepage() {
  showpage(2);
  history.pushState(2, null, "page2.html");
}
function newurl(event) {
  showpage(event.state);
}
function showpage(current) {
  databox.innerHTML = "The url is page " + current;
}
window.addEventListener("load", initiate);
```

There are two things we have to do in our application to have full control over the situation. To begin with, we must declare a state value for every URL we are going to use, the fakes and the real ones, and then, we must update the content of the document according to the current URL.

In the **initiate()** function in Listing 5, a listener is added for the **popstate** event. Every time a URL is revisited, the **newurl()** function is executed. This function updates the content of the **databox** according to the current URL. It takes the value of the **state** property and sends it to the **showpage()** function to be shown on the screen.

This works for every fake URL, but as we explained before, the real URLs do not have a state value by default. By using the **replaceState()** method at the end of the **initiate()** function, we change the information of the current entry (the real URL of the main document) and declare the value 1 for its state. Now, every time the user revisits the main document, we are able to detect it by checking this value.

The **changepage()** function is the same except this time it uses the **showpage()** function to update the document's content and declare the value **2** for the state of the fake URL.

The application works as follows: when the user clicks on the "page 2" link, the message "The url is page 2" is shown on the screen, and the URL in the navigation bar is changed to page2.html (including the full path, of course). This is what we had done so far, but here is where things get interesting. If the user clicks on the left arrow in the navigation bar, the URL in this bar changes to the previous in the browsing history (that is, the real URL of our document), and the **popstate** event is fired. This event calls the **newurl()** function that reads the value of the **state** property and sends it to the **showpage()** function. Now the state's value is 1 (the value we declared for this URL using the **replaceState()** method), and the message shown on the screen is "The url is page 1". If the user goes back to the fake URL using the right arrow in the navigation bar, the state's value will be **2**, and the message shown on the screen will again be "The url is page 2".

The value of the **state** property may be any value we want to control which URL is the current one and to adapt the document's content to the situation.

Do It Yourself: Use the files with the codes in Listings 2 and 3 for the HTML document and CSS styles. Copy the code in Listing 5 into the history.js file, and upload the files to your server or local server. Open the document in your browser and click on the text "page 2". The URL and the content of the **databox** should change according to the corresponding URL. Click the back and forward buttons on the navigation bar several times to see the URL changing and the content associated with that URL updated on the screen.

Real-Life Application

The following is a more practical application. We are going to use the History API to load a set of four images from the same document. Each image is associated with a fake URL that can later be used to retrieve it from the server.

The main document is loaded with an image by default. This image will be associated with the first of four links that are part of the permanent content. All these links will point to fake URLs referencing a state, not a real document, including the link of the main document that will be changed to page1.html to be consistent with the rest of the URLs.

Listing 6: *Creating the main document for our application*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>History API</title>
  <link rel="stylesheet" href="history.css">
  <script src="history.js"></script>
</head>
<body>
  <section id="maincontent">
    <p>This content is never refreshed</p>
    <p><span id="url1">image 1</span></p>
    <p><span id="url2">image 2</span></p>
    <p><span id="url3">image 3</span></p>
    <p><span id="url4">image 4</span></p>
  </section>
  <aside id="databox">
    <img id="image" src="monster1.gif">
  </aside>
</body>
</html>
```

The only difference between this new application and the previous one is the number of links and new URLs we are generating. In the code in Listing 5, there were two states: state 1 corresponding to the main document and state 2 for the fake URL (page2.html) generated by the **pushState()** method. In this case, we have to automate the process to generate a total of four fake URLs corresponding to each image available.

Listing 7: *Manipulating history*

```
function initiate() {
  for (var f = 1; f < 5; f++) {
    url = document.getElementById("url" + f);
    url.addEventListener("click", function(x){
      return function() {
        changepage(x);
      };
    }(f));
  }
  window.addEventListener("popstate", newurl);
  history.replaceState(1, null, "page1.html");
}
function changepage(page) {
  showpage(page);
  history.pushState(page, null, "page" + page + ".html");
}
function newurl(event) {
  showpage(event.state);
}
function showpage(current) {
  if (current != null) {
    var image = document.getElementById("image");
    image.src = "monster" + current + ".gif";
  }
}
window.addEventListener("load", initiate);
```

In this example, we are using the same functions, but with a few obvious changes. First, the **replaceState()** method in the **initiate()** function has the **url** attribute set to page1.html. We decided to program our application this way, declaring the state of the main document as **1** and the URL as page1.html (independently of the real URL of the document), because this allows us to define the names of all the URLs using the same name and the values of the **state** property. We can see this in practice in the **changepage()** function. Every time the user clicks on one of the links in the document, this function is executed, and the fake URL is built with the value of the **page** variable and added to the history. The value received by the function was previously set in the **for** loop at the beginning of the **initiate()** function. This value is set to **1** for the "image 1" link, **2** for the "image 2" link, and so on.

Every time a URL is visited, the **showpage()** function is executed to update the image according to the current URL. Because the **popstate** event sometimes is fired when the **state** property is **null** (like after the main document is loaded for the first time), we check the value received by the **showpage()** function before doing anything else. If the value is different than **null,** it means that the **state** property is defined for that URL and the image corresponding to that state has to be shown on the screen.

The images used for this example are named monster1.gif, monster2.gif, monster3.gif and monster4.gif, following the same order as the values of the **state** property. Thus, using this value, we can select the image to be shown.

Do It Yourself: To test the last example, use the HTML document of Listing 6 with the CSS code in Listing 3. Copy the code in Listing 7 into the history.js file. Download the files monster1.gif, monster2.gif, monster3.gif and monster4.gif from our website and upload all the files to your server or local server. Open the document in your browser and click on the links. Navigate through the URLs you selected using the navigation buttons. The images on the screen should change according to the URL in the navigation bar.

The Basics: The **for** loop used in the code in Listing 7 to add a listener for the **click** event to every **<span>** element in the document, takes advantage of a common technique in computer programming. To send the current value of the **f** variable to the function, we have to use two anonymous functions. The first function is executed when the **addEventListener()** method is executed. This function receives the current value of the **f** variable (see the parentheses at the end) and stores that value in the **x** variable. Then, the function returns a second anonymous function with the value of the variable **x**. This second function is the one that will be executed when the event is fired.

Quick Reference

History

length—This property returns the number of entries in the browsing history (the total of URLs on the list).

back()—This method takes the browser one step back in the browsing history (emulating the left arrow).

forward()—This method takes the browser one step forward in the browsing history (emulating the right arrow).

go(steps**)**—This method takes the browser back or forward the specified steps in the browsing history. The attribute may be a negative or positive value according to the direction we choose.

State

state—This property returns the value of the state of the current entry.

pushState(state, title, url**)**—This method creates a new entry in the browser history. The **state** attribute declares a value for the state of the entry. It is useful to identify the entry later, and it may be specified as a string or as a JSON object. The **title** attribute is the title of the entry, and the **url** attribute is the URL of the entry we are generating (this value will replace the current URL in the navigation bar).

replaceState(state, title, url**)**—This method works exactly like **pushState(),** but it does not generate a new entry. Instead, it replaces the information of the current one.

popstate—This event is fired when a URL is revisited or when the document has been loaded. It provides the **state** property with the value of the state declared when the URL was generated with the **pushState()** or **replaceState()** methods. This value is **null** when the URL is real unless we changed it before through the **replaceState()** method, as we will see next.