Hands-on with HTML APIs

PETER LEOW

DEDICATION

To my wife and kids for their love and support

CONTENTS

1 INTRODUCTION

The advent of HTML Living Standard, previously known as HTML5, has revolutionized the traditional web scene (and sense). In this book, I will simply call it HTML. In particular, HTML mandatory support for incorporating the ever-increasing number of APIs[1] into browsers has enable web pages to function more and more like desktop applications. This empowerment has brought about new possibilities and opportunities for the next generation of web applications that are more autonomous and can work offline, on multi-platforms, free of third party plug-ins and less reliant on server-side scripting. In the foreseeable future, it is not unimaginable that the web browser will replace our traditional metaphor of desktops on our computers, that of a web-based desktop.

Wow, the future of web landscape looks excitingly promising. However, reaching this stage is not without its challenges. For one thing, the supports of the current browsers must be improved and streamlined. For another, the awareness and education on HTML APIs among the web communities must be stepped up. Some would have argued about "the chicken or the egg" causality dilemma. I would argue that both can proceed in parallel.

Over the years, HTML specification has added a bag full of APIs that cover a wide spectrum of functionality and features that power the future web browsers and mobile devices. In this book, you will dip into the HTML APIs grab bag and draw out several of them for discussion and exploration peppered with plenty of hands-on exercises —

- Geolocation
- Drag and Drop
- Server-Sent Events

- Web Sockets
- Web Workers
- Web Storage

As you indulge in the many exercises on these HTML APIs, you will come to realize that some of them do not work from a mere `file://URI`, instead they must be launched from a web server. Therefore, you should install a web server on your computer so that all the exercises can be hosted and tested realistically.

[1]**API** stands for Application Programming Interface. It is also called library or plug-in. In a nut shell, it is a stand-alone purpose-made software program without a user interface that when referenced to, provides or extends the functionality of a computer application.

.

2 GEOLOCATION

Every one of us occupies a location on Earth. This location is specified by a geographic coordinate system of latitude, longitude, and altitude. With the proliferation of location-aware hardware and software, finding one's location on the Earth has never been easier. There are many techniques available to identify the location of a user. A computer browser generally uses WIFI or IP based positioning techniques whereas a mobile browser may use cell triangulation that based on your relative proximity to the different cellular towers operated by the telcos, GPS, A-GPS, or WIFI. Today, location awareness is an ever-growing trend that finds its way into many applications like:

- Showing one's location on a map especially when you are in an unfamiliar area.
- Providing turn-by-turn navigation while driving on unfamiliar journey.
- Find out the points of interest in one's vicinity.
- Getting the latest weather or news of one's area.
- Tagging the location of picture.
- Location tracking of a fleet of delivery trucks.

Thanks to **HTML Geolocation API**, you can now look up your own location on Earth using a browser, say Firefox. It is as easy as a piece of cake. Doing is believing. Let get your hands dirty.

Setting the Stage

Type the following code in Listing 1 using a text editor, save it as "finding_me.html", and launch it on a browser.

Listing 1 Source Code of finding_me.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Finding Me</title>
<script>
function getLocation()
{
  // Check whether browser supports Geolocation API or not
  if (navigator.geolocation) { // Supported
    navigator.geolocation.getCurrentPosition(getPosition);
  } else { // Not supported
    alert("Oop! This browser does not support HTML
Geolocation.");
  }
}
function getPosition(position)
{
  document.getElementById("location").innerHTML =
    "Latitude: " + position.coords.latitude + "<br>" +
    "Longitude: " + position.coords.longitude;
}
</script>
</head>
<body>
  <h1>Finding Me</h1>
  <button onclick="getLocation()">Where am I?</button>
  <p id="location"></p>
</body>
</html>
```

You should see a web page as shown in Figure 1. Note that this code will work on Firefox and IE, but not on Google Chrome. Chrome does not allow running Geolocation API from a `file://URI`, it will work if you deploy it on a web server like Apache.

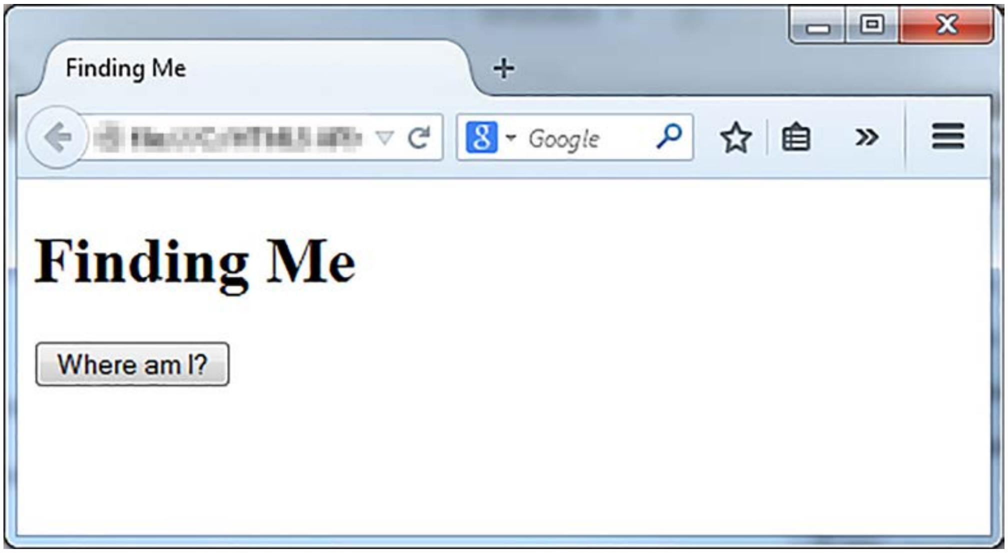


Figure 1 finding_me.html on a Firefox Browser

Click on the "Where am I?" button, what do you see?

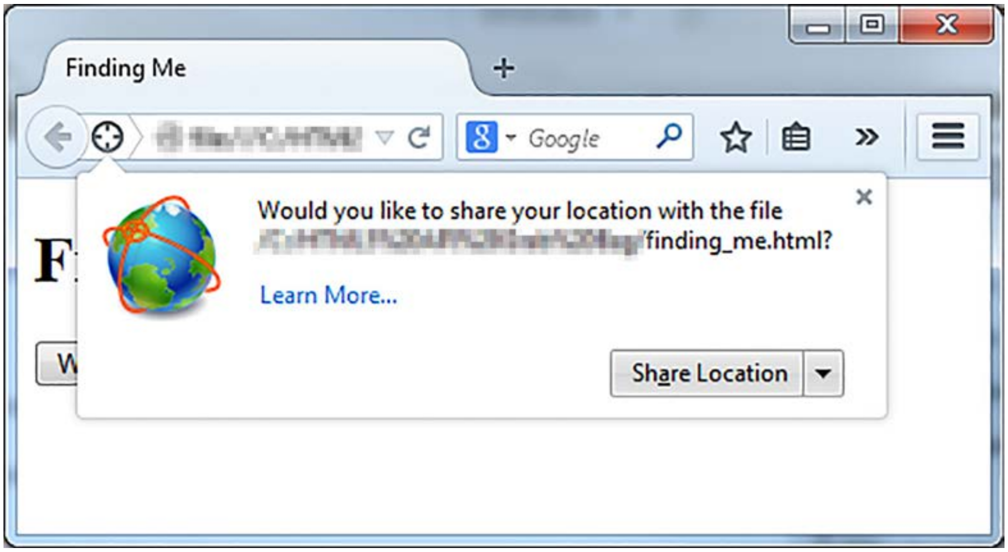
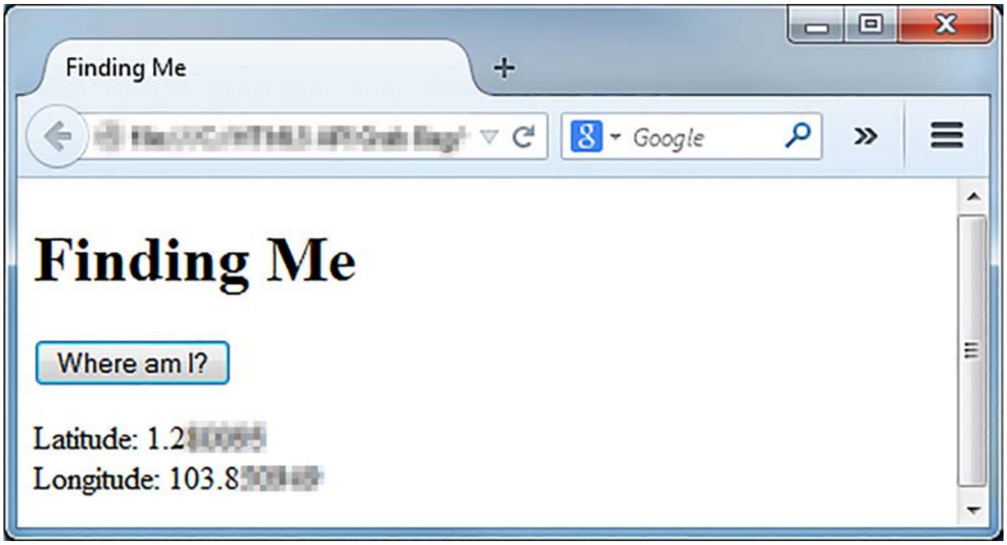


Figure 2 Seeking Permission to Share Location

A dialog box as shown in Figure 2 pops up seeking permission to share location, click on "Share Location" to grant.



Finding Me

Where am I?

Latitude: 1.2

Longitude: 103.8

Figure 3 You are found!

Diving Deeper…

The HTML geolocation API has only one object – the **navigator.geolocation** object. You may liken the navigator.geolocation to a **compass on the browser**. As browser support for this API is still dubious, it is a **de facto practice to check for browser support** before proceeding with any geolocation code, just wrap your code inside this `if-else` template as shown:

```
// Check whether browser supports Geolocation API or not
if (navigator.geolocation) // Supported
{
  // place the geolocation code here
}
else // Not supported
{
  alert("Oop! This browser does not support HTML
Geolocation.");
}
```

The navigator.geolocation object exposes three methods – **getCurrentPosition()**, **watchPosition()**, and **clearWatch()**.

getCurrentPosition()

The `getCurrentPosition()` method is used to obtain the current location of the user. You have already used this method in its simplest form in the code in Listing 1.

```
navigator.geolocation.getCurrentPosition(getPosition);
```

Recall in Figure 3 where a dialog has popped up seeking your permission to share your location with the web page. This is what happens whenever the getCurrentPosition() method is called. Users are given the choice to opt-in in order to allow the method to proceed to retrieve your current position. In other words, your privacy is well-respected and protected.

The `getCurrentPosition()` method takes three parameters:

- The first parameter is a callback function to be called when the call to `getCurrentPosition()` method is successful. The callback function will be called with a **position object** passed from the `getCurrentPosition()` method. This position object consists of 2 properties: **coords** and **timestamp**. In Listing 1, the callback function is `getPosition(position)` which takes a **position object parameter** and outputs the **latitude** and **longitude** through the **coords** property of this parameter. This is illustrated in the following code:

```
function getPosition(position)
{
  document.getElementById("location").innerHTML =
    "Latitude: " + position.coords.latitude + "<br>" +
    "Longitude: " + position.coords.longitude;
}
```

Table 1 shows the properties of the position object.

Table 1: Position Object

Property	**Description**
coords.latitude	The coords.latitude property returns the latitude of the user’s current position in decimal degrees.
coords.longitude	The coords.longitude property returns the longitude of the user’s current position in decimal degrees.
coords.altitude	The coords.altitude property returns the height of the user’s current position in meters above the sea level. It will returns null if this information is not available.
coords.accuracy	The coords.accuracy property returns the accuracy level of the latitude and longitude coordinates in meters.
coords.altitudeAccuracy	The coords.altitudeAccuracy property returns the accuracy level of the altitude in meters.
coords.heading	The coords.heading property returns the direction of travel of the location-aware device in degrees, where 0° starts from the north and counting clockwise. It will returns null if this information is not available.
coords.speed	The coords.speed property returns the speed of the location-aware device in meters per second. It will returns null if this information is not available.
timestamp	The timestamp property returns the time when the position object was acquired.

- The second parameter is an optional error handling callback function to be invoked when the call to `getCurrentPosition()` method encounters any one of the following situations:

    - Request timed out
    - Location information not available
    - User has denied permission to share the location information

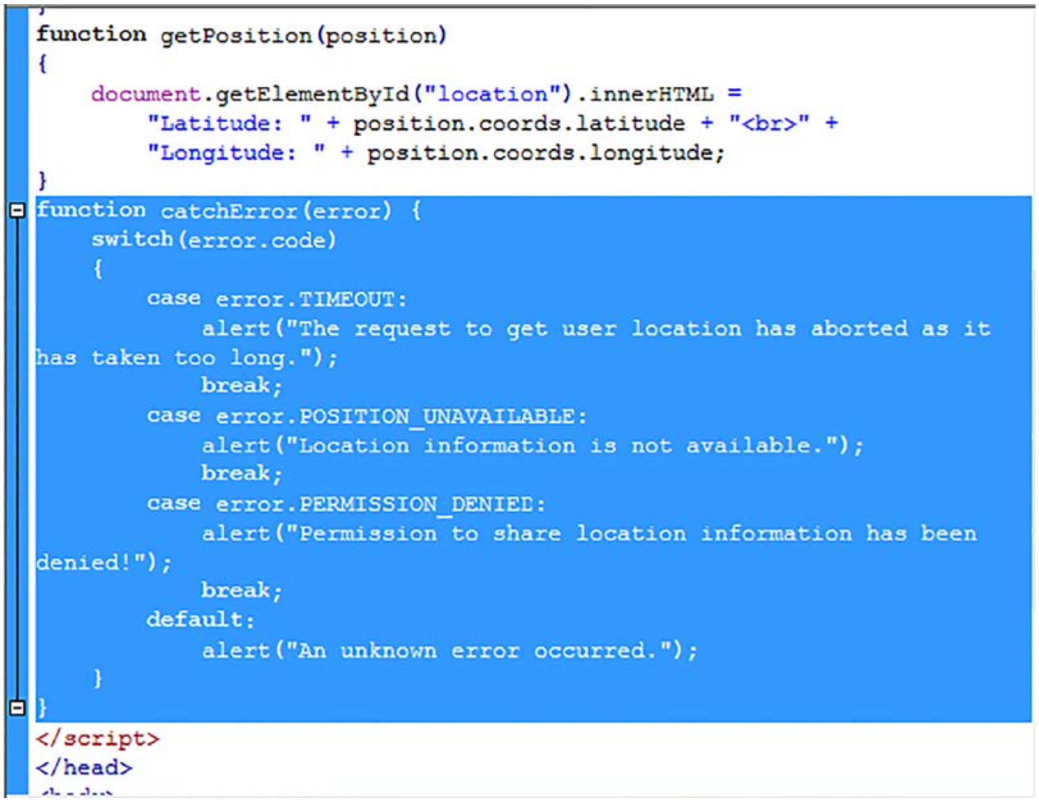
The callback function will be invoked with a **position error object** parameter passed from the `getCurrentPosition()` method. This position error object consists of one property – **code**. This code property takes one of three values corresponding to the error types as shown in Table 2.

Table 2: Location Error Codes

Property	**Description**
TIMEOUT	Request for location information exceeds the timeout property set in the position options object (discussed later).
POSITION_UNAVAILABLE	The position of the location-aware device cannot be determined.
PERMISSION_DENIED	User has denied permission to share the location information.

You will add a second callback function called `catchError(error)` to the `<script>` as shown in the following code and as highlighted in Figure 4:

```
function catchError(error) {
  switch(error.code)
  {
    case error.TIMEOUT:
      alert("The request to get user location has aborted
as it has taken too long.");
      break;
    case error.POSITION_UNAVAILABLE:
      alert("Location information is not available.");
      break;
    case error.PERMISSION_DENIED:
      alert("Permission to share location information has
been denied!");
      break;
    default:
      alert("An unknown error occurred.");
  }
}
```



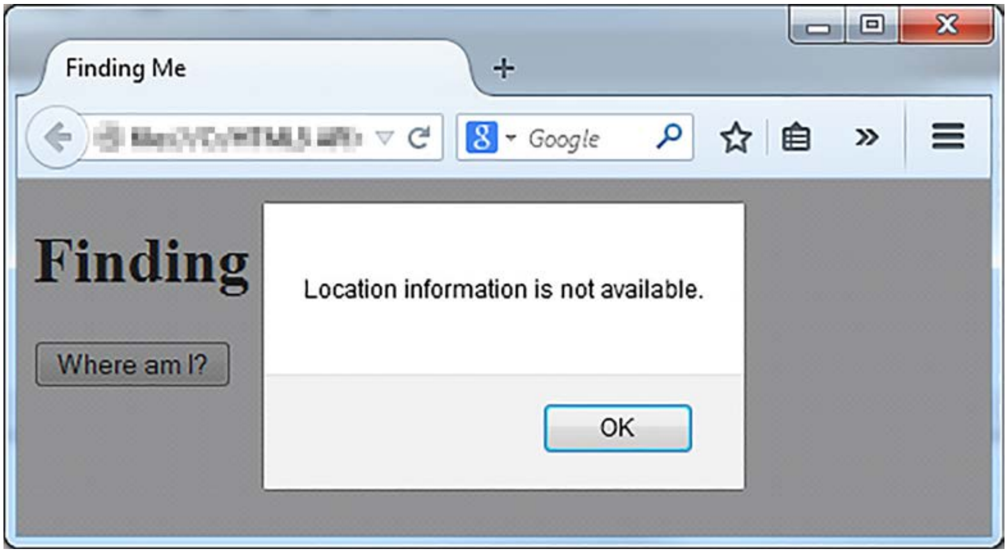
```
function getPosition(position)
{
    document.getElementById("location").innerHTML =
        "Latitude: " + position.coords.latitude + "<br>" +
        "Longitude: " + position.coords.longitude;
}
function catchError(error) {
    switch(error.code)
    {
        case error.TIMEOUT:
            alert("The request to get user location has aborted as it
has taken too long.");
            break;
        case error.POSITION_UNAVAILABLE:
            alert("Location information is not available.");
            break;
        case error.PERMISSION_DENIED:
            alert("Permission to share location information has been
denied!");
            break;
        default:
            alert("An unknown error occurred.");
    }
}
</script>
</head>
```

Figure 4 `catchError()` Function

Next, attach the following function as the second parameter to the `getCurrentPosition()` method as shown:

```
navigator.geolocation.getCurrentPosition(getPosition,
catchError);
```

Save and launch it on a browser, e.g. Firefox, unplug your network cable (or turn off the wireless switch) to simulate the “no network connection” situation, grant the permission for sharing location information as shown in Figure 2, and then click on the “Where am I?” button. What do you see?



Location information is not available.

Figure 5 Simulating an Error Situation

You should see a message box pops up saying "Location information is not available." as shown in Figure 5. If you look back to the code in the `catchError()` function, you would notice that this is the error message specified under the case of `error.POSITION_UNAVAILABLE`. The cause is obviously due to the network cable being unplugged. The `catchError()` function was invoked by the `getCurrentPosition()` method as it is not able to obtain any location information due to the broken network connection. (You may recover the network connection to your computer.) Although this error handling parameter is optional, you should always include it as part of the geolocation code. This is one of the best practices to allow the program to fail gracefully as well as to keep the users informed of any run-time errors.

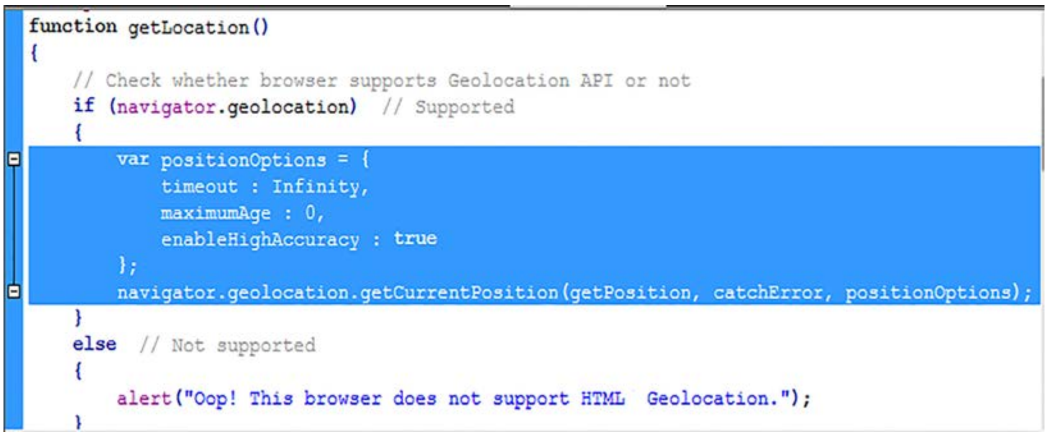
- The third parameter is an optional **position options object** that can be used to fine tune the position object returned by the `getCurrentPosition()` method programmatically. It has three properties as shown in Table 3.

Table 3: Position Options Object Properties

Property	**Description**
timeout	The timeout property denotes the number of milliseconds an application will wait for a position information to become available. The default value is infinity.
maximumAge	The maximumAge property denotes the number of milliseconds an application can keep using the cached (previously obtained) location data before trying to obtain new location data. A zero value indicates that the application must not use the cached location data while infinity value indicates that the cached location data must be used by the application. The default value is zero.
enableHighAccuracy	The enableHighAccuracy property denotes the condition of true or false. If it is true, the application will try to provide the most accurate position. However, this would result in slower response time and higher power consumption. The default value is false.

You will define a position options object called "positionOptions" and add it as the third parameter to the `getCurrentPosition()` method as shown in the following code and as highlighted in Figure 6:

```
var positionOptions = {
  timeout : Infinity,
  maximumAge : 0,
  enableHighAccuracy : true
};
navigator.geolocation.getCurrentPosition(getPosition,
catchError, positionOptions);
```



```
function getLocation()
{
    // Check whether browser supports Geolocation API or not
    if (navigator.geolocation)  // Supported
    {
        var positionOptions = {
            timeout : Infinity,
            maximumAge : 0,
            enableHighAccuracy : true
        };
        navigator.geolocation.getCurrentPosition(getPosition, catchError, positionOptions);
    }
    else  // Not supported
    {
        alert("Oop! This browser does not support HTML  Geolocation.");
    }
```

Figure 6 Position Options Object

In designing your location-aware application, you should choose the degree of accuracy as well as the retention period of old location data that are most appropriate for the purpose of the application by setting the "enableHighAccuracy" and the "maximumAge" properties of the position options object accordingly. For example, if your application is mainly for finding points of interest in your vicinity, you probably do not need high accuracy and the location information do not have to be updated so often. On the other hand, if your application is to provide turn by turn navigation direction for driver, then high accuracy and constant update of location become necessary.

The `getCurrentPosition()` method is most suitable for obtaining a one-off location information. However, for finding location information that is changing continuously, such as the turn by turn navigation that I mentioned above, you will have to call the `getCurrentPosition()` method repeatedly which is obviously very cumbersome and inefficient. Fret not! HTML Geolocation API has provided another method to handle this kind of situation – **watchPosition()**.

watchPosition()

The **watchPosition()** method is almost identical to the `getCurrentPosition()` method. They both return the current location information and have the same method signature – one success callback function, one error callback function, and one position options object. The only difference lies in that the `getCurrentPosition()` method only returns location information **once** upon activation such as a button click, whereas The `watchPosition()` method will **continue** to obtain location information every time the location of the user's device changes after the initial activation.

The `watchPosition()` method returns a **watch ID** that can be used to stop obtaining location information by passing it to the third method of the navigator.geolocation object – `clearWatch()`, such as when a car that uses it for navigation has arrived at the destination.

clearWatch()

The `clearWatch()` method takes the watch ID of a `watchPosition()` method as a parameter and stop the execution of that `watchPosition()` method.

Location Tracking

As the `watchPosition()` method is similar to the `getCurrentPosition()` method, if you are familiar with the later, creating the `watchPosition()` method will be much easier – it is mostly a matter of, believe it or not, "copy and paste"

Open "finding_me.html" in a text editor, save it as "watching_me.html". Then, in "watching_me.html",

- Change "navigator.geolocation.**getCurrentPosition**(getPosition, catchError, positionOptions)" to "navigator.geolocation.**watchPosition**(getPosition, catchError, positionOptions)".

- Store the watch ID returned from the `watchPosition()` method to a variable `watchID`.

```
watchID = navigator.geolocation.watchPosition(getPosition,
catchError, positionOptions);
```

That's all that you need to create a web page that constantly monitors and updates a browser's current location.

However, to stop the monitoring and updating at some point in time, you will add another piece of code and a button to deactivate it. In the `<script>` section of "watching_me.html", create a function called `stopWatch()` that calls the `clearWatch()` method by passing it the `watchID` of the `watchPosition()` method:

```
function stopWatch()
{
  navigator.geolocation.clearWatch(watchID);
}
```

Lastly, replace the HTML code in the `<body>` of “watching_me.html” with:

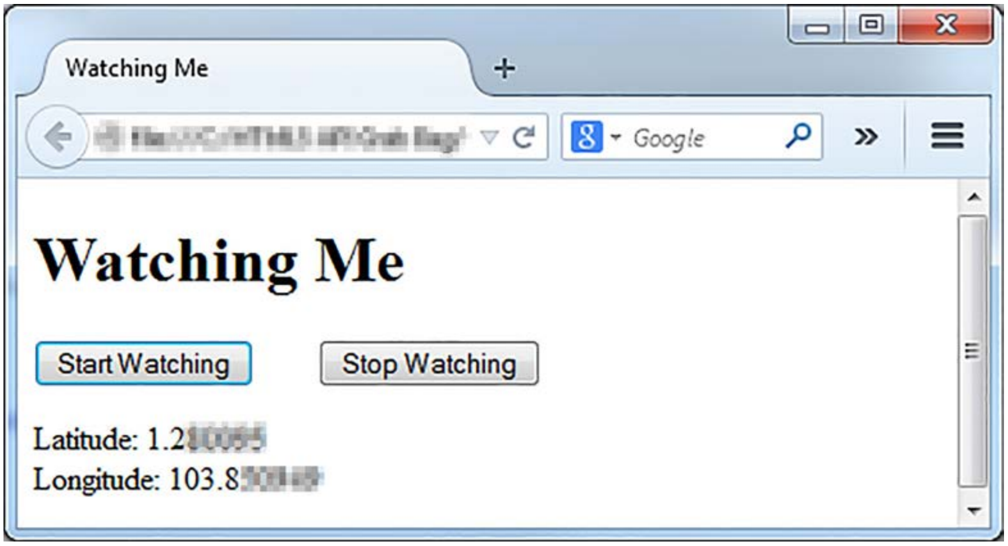
```
<body>
<h1>Watching Me</h1>
<button onclick="getLocation()">Start Watching</button>
&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
<button onclick="stopWatch()">Stop Watching</button>
<p id="location"></p>
</body>
```

The complete source code of “watching_me.html” is given in Listing 2.

Listing 2 Complete Source Code of watching_me.html

```
<!DOCTYPE html>
<html>
<head>
<title>Watching Me</title>
<script>
function getLocation()
{
  // Check whether browser supports Geolocation API or not
  if (navigator.geolocation)  // Supported
  {
   var positionOptions = {
       timeout : Infinity,
       maximumAge : 0,
       enableHighAccuracy : true
   };
   // Set the watchID
   watchID = navigator.geolocation.watchPosition(getPosition,
catchError, positionOptions);
  }
  else  // Not supported
  {
   alert("Oop! This browser does not support HTML 
Geolocation.");
  }
}
function stopWatch()
{
  navigator.geolocation.clearWatch(watchID);
  watchID = null;
}
function getPosition(position)
{
  document.getElementById("location").innerHTML =
     "Latitude: " + position.coords.latitude + "<br>" +
     "Longitude: " + position.coords.longitude;
}
function catchError(error) {
  switch(error.code)
  {
   case error.TIMEOUT:
     alert("The request to get user location has aborted as it 
has taken too long.");
     break;
   case error.POSITION_UNAVAILABLE:
     alert("Location information is not available.");
     break;
   case error.PERMISSION_DENIED:
     alert("Permission to share location information has been 
denied!");
     break;
   default:
     alert("An unknown error occurred.");
  }
}
</script>
</head>
<body>
<h1>Watching Me</h1>
<button onclick="getLocation()">Start Watching</button>
&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
<button onclick="stopWatch()">Stop Watching</button>
<p id="location"></p>
</body>
</html>
```

Save and launch "watching_me.html" on a WIFI connected laptop (Firefox) browser, click the "Start Watching" button, grant the permission to share location, then move around with the laptop. What do you see?



Watching Me

Latitude: 1.2[redacted]
Longitude: 103.8[redacted]

Figure 7 You are being watched!

You should see your position being updated on the browser at a regular interval as shown in an example in Figure 7.

If you copy and paste the latitude and longitude values to the search box of a Google Map on your browser, Google Map will mark your approximate location with a marker. The location is approximate and may differ on different browsers and devices as the accuracy of which is dependent on a number of factors, such as your public IP address, the nearer cellular towers, GPS availability, WIFI access points, and the type of browser that you are using.

Location Tracking on Map

Let's put the location on a map as shown in an example in Figure 8.

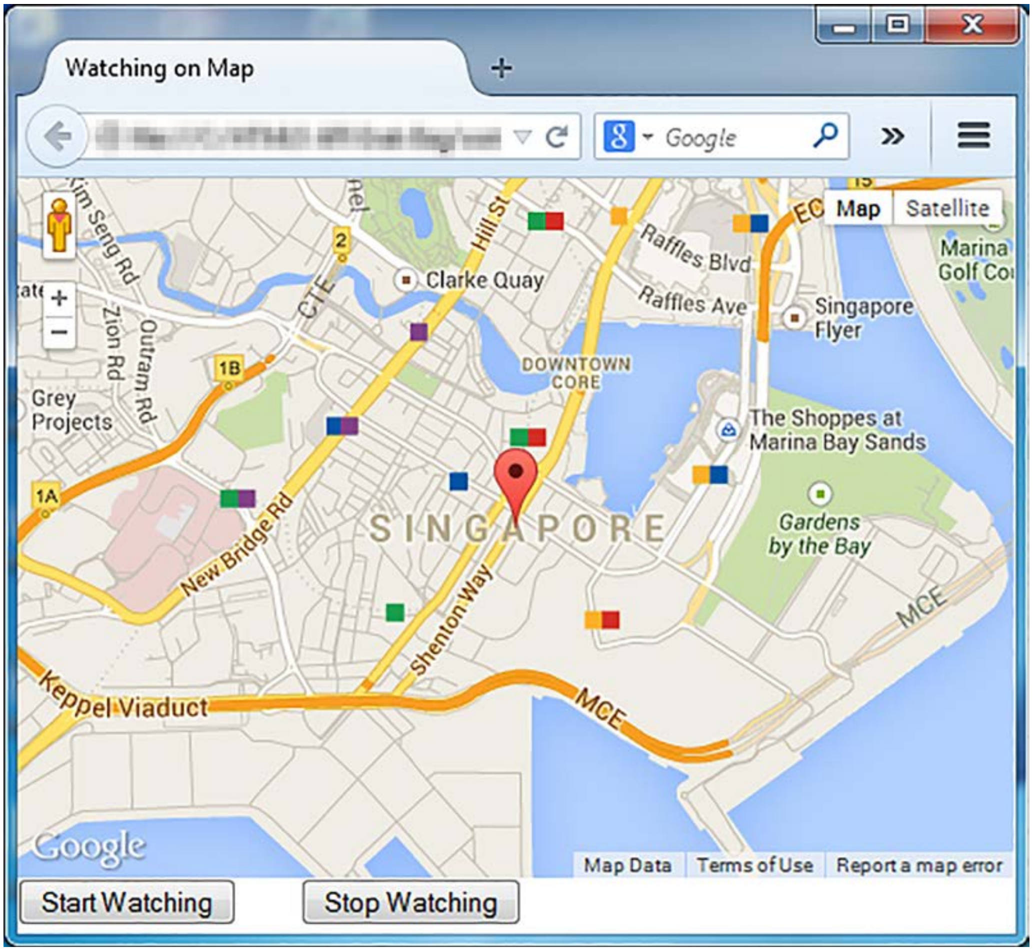


Figure 8 You are on Google map!

The complete code for showing location on map is given in Listing 3.

Listing 3 Source Code for Showing Location on Google Map

```
<!DOCTYPE html>
<html>
<head>
<title>Watching on Map</title>
<style>
html,body {
  height: 100%;
  margin: 0;
  padding: 0;
}
#map-holder {
  height: 350px;
  width: 500px;
}
</style>
<script
src="https://maps.googleapis.com/maps/api/js?v=3.exp&sensor=false
"></script>
<script>
var watchID;
function getLocation()
{
  // Check whether browser supports Geolocation API or not
  if (navigator.geolocation)  // Supported
  {
   var positionOptions = {
     timeout : Infinity,
     maximumAge : 0,
     enableHighAccuracy : true,
   };
   // Obtain the initial location one-off
   navigator.geolocation.getCurrentPosition(getPosition,
catchError, positionOptions);
  }
  else  // Not supported
  {
   alert("Oop! This browser does not support HTML Geolocation.");
  }
}
function watchLocation()
{
  // Check whether browser supports Geolocation API or not
  if (navigator.geolocation)  // Supported
  {
   var positionOptions = {
     timeout : Infinity,
     maximumAge : 0,
     enableHighAccuracy : true,
   };
   // Obtain the location at regularly interval
   watchID = navigator.geolocation.watchPosition(getPosition,
catchError, positionOptions);
  }
  else // Not supported
  {
   alert("Oop! This browser does not support HTML Geolocation.");
  }
}
function stopWatch()
{
  // Discontinue obtaining location
  navigator.geolocation.clearWatch(watchID);
}
function getPosition(position)
{
  var location = new google.maps.LatLng(position.coords.latitude,
position.coords.longitude);
  var mapOptions = {
   zoom : 12,
   center : location,
   mapTypeId : google.maps.MapTypeId.ROADMAP
  };
  var map = new google.maps.Map(document.getElementById('map-
holder'), mapOptions);
  var marker = new google.maps.Marker({
   position: location,
   title: 'Here I am!',
   map: map,
   animation: google.maps.Animation.DROP
  });
  var geocoder = new google.maps.Geocoder();
  geocoder.geocode({
     'latLng' : location
     }, function(results, status) {
      if (status == google.maps.GeocoderStatus.OK) {
        if (results[1]) {
          var options = {
             content : results[1].formatted_address,
             position : location
          };
          var popup = new google.maps.InfoWindow(options);
          google.maps.event.addListener(marker, 'click',
function() {
             popup.open(map);
          });
         }
         else
         {
          alert('No results found.');
         }
       }
       else
       {
         alert('Geocoder failed due to: ' + status);
       }
  });
}
function catchError(error) {
  switch(error.code)
  {
   case error.TIMEOUT:
     alert("The request to get user location has aborted as it
has taken too long.");
     break;
   case error.POSITION_UNAVAILABLE:
     alert("Location information is not available.");
     break;
   case error.PERMISSION_DENIED:
     alert("Permission to share location information has been
denied!");
     break;
   default:
     alert("An unknown error occurred.");
  }
}
</script>
</head>
<body onload="getLocation()">
<div id="map-holder"></div>
<button onclick="watchLocation()">Start Watching</button>
```

```
&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
<button onclick="stopWatch()">Stop Watching</button>
</body>
</html>
```

The logic of the code works as follows:

- When the page is first loaded, the onload event will be triggered and calls the `getLocation()` function which in turn calls the `getCurrentPosition()` method of the navigator.geolocation object to obtain the user's current location.

- If the `getCurrentPosition()` method is successful, it will trigger the callback function `getPosition()` passing it the position object.

- What the `getPosition()` function does is to render a Google Map that is centered on the user's location shown as marker.

- Clicking the "Start Watching" button will call the `watchLocation()` function which in turn calls the `watchPosition()` method of the navigator.geolocation object to obtain the user's current location at a regular interval.

- If the `watchPosition()` method is successful, it will trigger the callback function `getPosition()` passing it the position object.

- What the `getPosition()` function does has been described above.

- Clicking the "Stop Watching" button will stop the watch activity.

If you move around with your WIFI connected laptop while keeping this web page opened, you should see your location being updated on the browser regularly as you move. Have fun!

.

3 DRAG AND DROP

We are all too familiar with the drag and drop of files, folder, and icons on our computer desktop. Drag and drop is a powerful and yet taken-for-granted user interface functionality of any desktop applications. With the help of pointing devices like mouse, drag and drop makes copying, insertion, and deletion of files and objects on your desktop a breeze. On the contrary, to achieve the same drag and drop on browsers, complex client-side scripting like JavaScript is required. This was the main reason that deterred the use of drag and drop in the many traditional web pages that we have seen so far. However, this is expected to change with the introduction of **HTML Drag and Drop API** which brings native support to the browser making it much easier to code. See for yourself how drag and drop in HTML works by getting your hands dirty.

Setting the Stage

Type the following code in Listing 4 using a text editor, save it as “htmldragndrop.html”.

Listing 4 Source Code of htmldragndrop.html

```
<!DOCTYPE HTML>
<html>
<head>
<title>HTML Drag and Drop API</title>
<style type="text/css">
#container {
  width: 100%;
  margin: auto;
  text-align:center;
}
#trash, #dustbin{
  display: inline-block;
  padding:10px;
  margin:10px;
}
#trash {
  background-color: #DFD7D7;
  width:50px;
  height:50px;
}
#dustbin {
  background-color: #A347FF;
  width:150px;
  height:150px;
}
</style>
<script>

</script>
</head>
<body>
<div id="container">
<h2>HTML Drag and Drop</h2>
<h4 id="status">Littering is an offence!</h4>
<div id="dustbin">
  <p>Dustbin</p>
</div>
<div id="trash">
  <p>Trash</p>
</div>
</div>
</body>
</html>
```

This “htmldragndrop.html” will be rendered on a browser as shown in Figure 9.

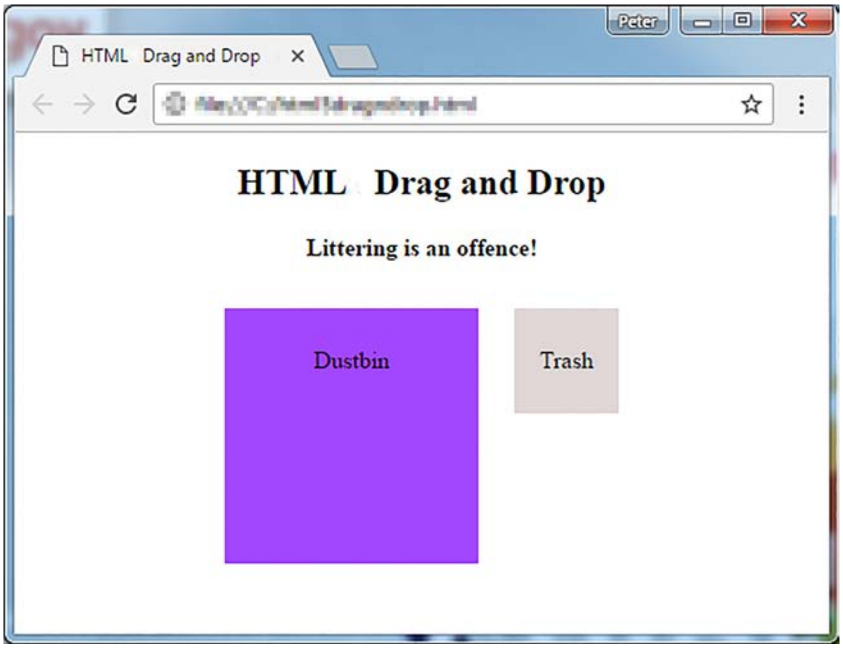


Figure 9 htmldragndrop.html on a Browser

Your mission is to add the code to drag and drop the "Trash" into the "Dustbin" so as not to litter the place. Follow me…

draggable="true"

To make an element draggable, set its **draggable** attribute to **true**. Here, the draggable element is the `<div id="trash">`, you will enable it like this:

```
<div id="trash" draggable="true">
```

ondragstart="dragStart(event)"

Next, you have to specify what should happen when the element is dragged by setting its **ondragstart** attribute to an event handler (function) that handles it. Here, the event handler is called `dragStart(event)`. You may change it to any other name. Add this attribute like this:

```
<div id="trash" draggable="true"
ondragstart="dragStart(event)">
```

dragStart(event)

The `dragStart(event)` event handler is a JavaScript function that is registered with the **ondragstart** attribute of a draggable element. It is called when the user starts dragging the element and is passed a **dragEvent** object (named as **event** in this example) as parameter. Add this function to the `<script>` section as shown:

```
<script>
function dragStart(event)
{
  event.dataTransfer.setData("text/plain", event.target.id);
  document.getElementById("status").innerHTML="Drag Start";
}
</script>
```

The `dragStart(event)` function can set the data for passing to the drop zone through the **dataTransfer.setData(type, data)** method (to be discussed in detail later) of the **dragEvent** object. In this exercise, the **data** parameter is assigned the id of the dragged div which is "trash" via the event.target.id, while the **type** parameter a MIME type called "text/plain" in this exercise. The type parameter can be any value that best describes the type of data and serves as an identifier for retrieving the data subsequently. I have added one line of code to display the status of the drag event, i.e. "Drag Start", on the browser.

Save the code and refresh it on the browser, can you drag the trash now?

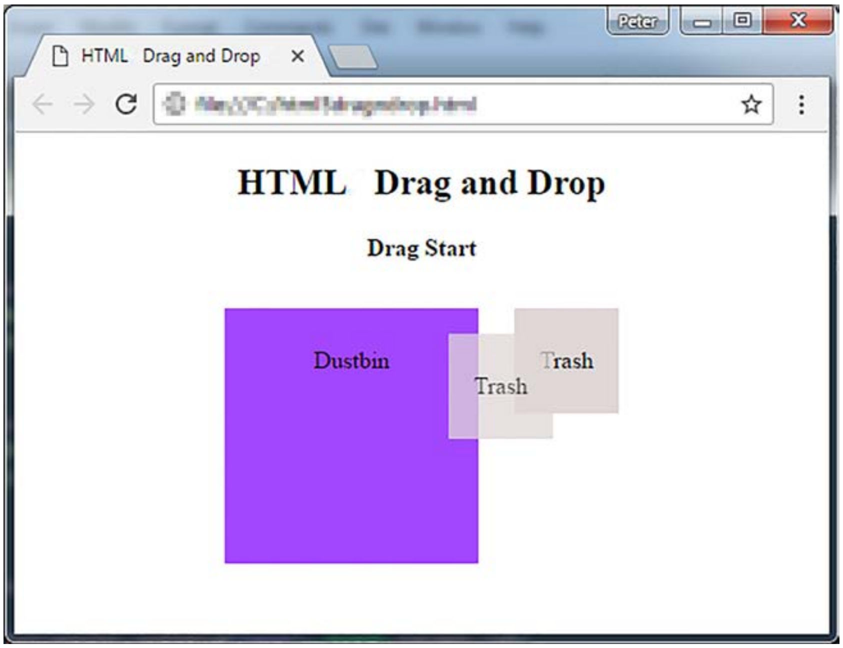


Figure 10 Drag Start

Yes, you can and the status reads "Drag Start" as shown in Figure 10. However, it cannot be dropped anywhere! You will have to add the code for dropping the trash. Move on…

ondragover="dragOver(event)"

You can specify what should happen when a draggable element is being dragged over a drop zone by setting the **ondragover** attribute of this drop zone to an event handler (function) that handles it. In this exercise, the event handler is called **dragOver(event)**. You may change to any other name. Here, the drop zone is the `<div id="dustbin">`, add this code to it:

```
<div id="dustbin" ondragover="dragOver(event)">
```

dragOver(event)

The `dragOver(event)` function is registered with the **ondragover** attribute of a drop zone. It is called when a dragged element is moved over the drop zone and is passed an **Event** object (named as **event** here) as parameter. Note that the browser default handling of a drag over event is to NOT allow a drop. So, to allow a drop, we have to override this default behavior by calling `event.preventDefault()`. Add the `dragOver()` function in the `<script>` section as shown:

```
function dragOver(event)
{
  event.preventDefault();
  document.getElementById("status").innerHTML="Drag Over";
}
```

Save the code and refresh it on the browser, then drag the trash over the dustbin, what do you see?

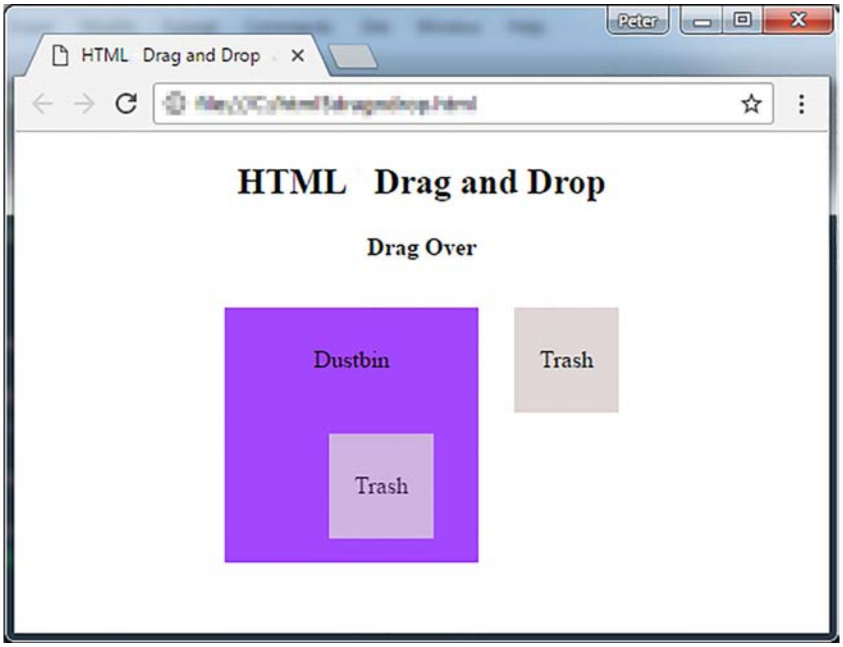


Figure 11 Drag Over

Notice that the status now shows "Drag Over" as shown in Figure 11. However, it still cannot be dropped into the dustbin! Mission not accomplished yet. You have to add the code to instruct the dustbin to accept the trash. Move on…

ondrop=”drop(event)”

In order to accept the dropping of a draggable element onto a drop zone, you have to specify what should happen when that element is dropped onto it by setting the **ondrop** attribute of the drop zone, i.e. `<div id=”dustbin”>` in this exercise, to an event handler that handles it. Here, the event handler is simply called **drop(event)**. You may change to any other name. Add this attribute like this:

```
<div id="dustbin" ondragover="dragOver(event)"
ondrop="drop(event)">
```

drop(event)

The drop function is registered with the “ondrop” attribute of a drop zone. It is called when a draggable element is being released onto the drop zone at the end of a drag and is passed a **dragEvent** object (named as **event** here) as parameter. Add this function in the <script> section as shown:

```
function drop(event)
{
  event.preventDefault();
  var data=event.dataTransfer.getData("text/plain");
  event.target.appendChild(document.getElementById(data));
  document.getElementById(data).innerHTML="Trash dumped!";
  document.getElementById("status").innerHTML="Dropped";
}
```

The drop function can call the **dataTransfer.getData(type)** method of the dragEvent object to retrieve the dragged data that was set by **dataTransfer.setData(type, data)** method of the dragStart(event) function by passing it a type as parameter, i.e. “text/plain” in this exercise. The subsequent code simply drops (append) the dragged data, which contains the id of the dragged div, i.e. <div id=”trash”>, onto the drop zone, i.e. <div id=”dustbin”>. Note that the browser default handling of a drop event is to open a link to some URL. We can override this default behavior by calling **event.preventDefault()**.

Mission Accomplished

You should be able to drag the trash and drop it into the dustbin now as shown in Figure 12. Observe the change of statuses at the different stages of the drag and drop operation.

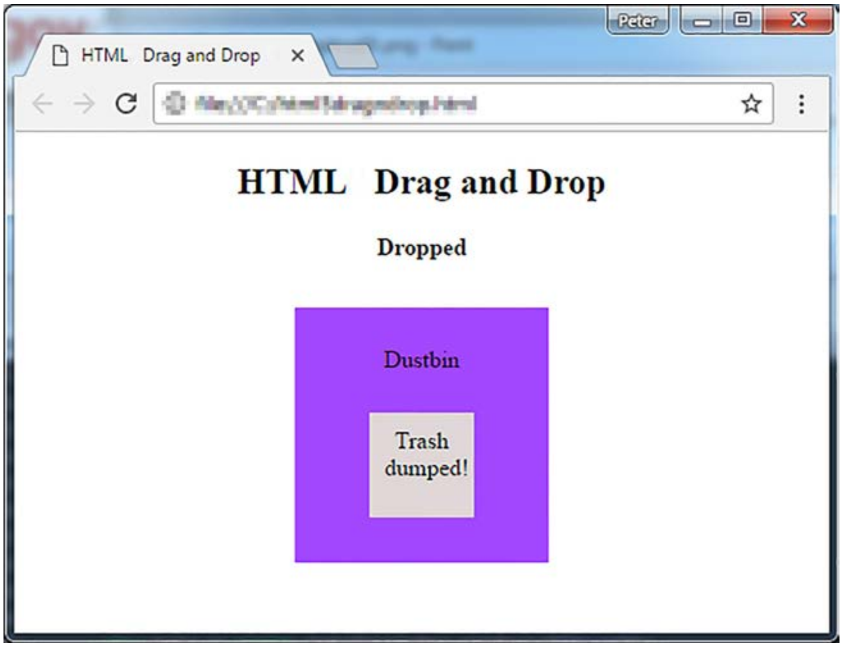


Figure 12 Trash dumped

As a reward, the complete code for "htmldragndrop.html" is given in Listing 5.

Listing 5 Complete Source Code of htmldragndrop.html

```
<!DOCTYPE HTML>
<html>
<head>
<title>HTML Drag and Drop API</title>
<style type="text/css">
#container {
  width: 100%;
  margin: auto;
  text-align:center;
}
#trash, #dustbin{
  display: inline-block;
  padding:10px;
  margin:10px;
}
#trash {
  background-color: #DFD7D7;
  width:50px;
  height:50px;
}
#dustbin {
  background-color: #A347FF;
  width:150px;
  height:150px;
}
</style>
<script>
function dragStart(event)
{
  event.dataTransfer.setData("text/plain", event.target.id);
  document.getElementById("status").innerHTML = "Drag Start";
}
function dragOver(event)
{
  event.preventDefault();
  document.getElementById("status").innerHTML = "Drag Over";
}
function drop(event)
{
  event.preventDefault();
  var data = event.dataTransfer.getData("text/plain");
  event.target.appendChild(document.getElementById(data));
  document.getElementById(data).innerHTML="Trash dumped!";
  document.getElementById("status").innerHTML = "Dropped";
}
</script>
</head>
<body>
<div id="container">
<h2>HTML Drag and Drop</h2>
<h4 id="status">Littering is an offence!</h4>
<div id="dustbin" ondragover="dragOver(event)"
ondrop="drop(event)">
  <p>Dustbin</p>
</div>
<div id="trash" draggable="true"
```

```
ondragstart="dragStart(event)">
  <p>Trash</p>
</div>
</div>
</body>
</html>
```

Diving Deeper…

A total of seven events are being fired at different stages of a drag and drop operation. Individually, they can be monitored and detected by seven event listeners attached to the draggable elements or the drop zone as attributes. You have already met three of the event listeners, i.e. **ondragstart**, **ondragover**, and **ondrop**, in your first drag and drop exercise above.

Each event listener can be assigned an event handler which is a JavaScript function to perform when an event arises, such as the **dragStart(event)** function is being assigned to **ondragstart** event listener in the preceding exercise. You may name the event handlers differently but generally it is a good practice to stay similar to the names of their event listeners minus the prefix "on".

Let's examine the other four event listeners — **ondrag**, **ondragend**, **ondragenter**, and **ondragleave**. You should test out the code snippets provided by adding them to the "htmldragndrop.html" and view their effects on a browser.

ondrag=”drag(event)”

Add an **ondrag** event listener as an attribute to a draggable element. For example:

```
<div id="trash" draggable="true" ondragstart="dragStart(event)"
ondrag="drag(event)">
```

drag(event)

The **ondrag** event listener will detect and fire the event handler assigned to it, i.e. **drag(event)** in this case, whenever the element is being dragged. For example:

```
function drag(event)
{
 document.getElementById("status").innerHTML="<p>Dragging</p>";
}
```

ondragenter=”dragEnter(event)”

Add an **ondragenter** event listener as an attribute to a drop zone (droppable element). For example:

```
<div id="dustbin" ondragover="dragOver(event)"
ondrop="drop(event)" ondragenter="dragEnter(event)">
```

dragEnter(event)

The **ondragenter** event listener will detect and fire the event handler assigned to it, i.e. **dragEnter(event)** in this case, at the moment when the mouse pointer dragging the draggable element enters the drop zone. For example:

```
function dragEnter(event)
{
document.getElementById("dustbin").innerHTML="<p>Dustbin</p><p>Tra
sh Enter</p>";
}
```

ondragleave=”dragLeave(event)”

Add an **ondragleave** event listener as an attribute to a drop zone element. For example:

```
<div id="dustbin" ondragover="dragOver(event)"
ondrop="drop(event)" ondragenter="dragEnter(event)"
ondragleave="dragLeave(event)">
```

dragLeave(event)

The **ondragleave** event listener will detect and fire the event handler assigned to it, i.e. **dragLeave(event)** in this case, at the moment when the mouse pointer dragging the draggable element leaves the drop zone. For example:

```
function dragLeave(event)
{
document.getElementById("dustbin").innerHTML="<p>Dustbin</p><p>Tr
ash Leave</p>";
}
```

ondragend=”dragEnd(event)”

Add an **ondragend** event listener as an attribute to a draggable element. For example:

```
<div id="trash" draggable="true" ondragstart="dragStart(event)"
ondrag="drag(event)" ondragend="dragEnd(event)">
```

dragEnd(event)

The **ondragend** event listener will detect and fire the event handler assigned to it, i.e. **dragEnd(event)** in this case, when the user releases the mouse button while dragging an element. For example:

```
function dragEnd(event)
{
  document.getElementById("status").innerHTML="Drag End";
}
```

DataTransfer Object

The event handlers for all the drag and drop events accept an **Event** parameter that has a **dataTransfer** property which returns a **DataTransfer** object that holds the data that is being dragged during a drag and drop operation. This data can be set and retrieved via the various properties and methods associated with the DataTransfer object.

dataTransfer Properties

There are four dataTransfer properties — **dropEffect**, **effectAllowed**, **types**, and **files**.

dropEffect

The **dropEffect** property specifies one of the four possible values for drop effect of the **dragenter** and **dragover** events and should always be one of the four values set by the effectAllowed property. The four values are:

- **copy**: A copy of the dragged item is made at the new location.
- **move**: The dragged item is moved to a new location.
- **link**: A link is established to the dragged item at the new location.
- **none**: The item cannot be dropped.

For example:

```
function dragEnter(event)
{
  event.dataTransfer.dropEffect='move';
}
```

effectAllowed

The **effectAllowed** property specifies the effects that are allowed for a drag by setting it in the **dragstart** event. The possible values are:

- **copy**: A copy of the dragged item may be made at the new location.
- **move**: The dragged item may be moved to a new location.
- **link**: A link may be established the dragged item at the new location.
- **copyLink**: A copy or link operation is permitted.

- **copyMove**: A copy or move operation is permitted.
- **linkMove**: A link or move operation is permitted.
- **all**: All operations are permitted.
- **none**: The item cannot be dropped.
- **uninitialized**: This is the default value when no effect is set, which is equivalent to all.

For example:

```
function dragStart(event)
{
  event.dataTransfer.effectAllowed='move';
}
```

types

The **types** property returns the list of formats that are set in the **dragstart** event.

files

The **files** property returns the list of files being dragged, if any.

dataTransfer Methods

There are five dataTransfer properties — **clearData()**, **setData()**, **getData()**, **setDragImage()**, and **addElement()**.

clearData()

The `clearData()` method takes a type as parameter and removes the data associated with the type. The type parameter is optional. If the type is omitted, all data is removed. For example:

```
event.dataTransfer.clearData('text/plain');
```

setData()

The `setData()` method sets the data for a given type. The first parameter is the type and the second parameter the data. For example:

```
event.dataTransfer.setData('text/plain', event.target.id);
```

getData()

The `getData()` method takes a type as parameter and retrieves the data for the type, or an empty string if data for that type does not exist or the dataTransfer object contains no data. For example:

```
data = event.dataTransfer.getData('text/plain');
```

setDragImage()

The `setDragImage()` method is used to set a custom image to be used during the dragging process. If this is not set, the default image is created from the source that is being dragged. The setDragImage() method takes three parameters:

- Image source
- x coordinate offset into the image where the mouse cursor should be
- y coordinate offset into the image where the mouse cursor should be

For example, to center the image, set the x and y coordinate offset values to half of the width and height of the image respectively:

```
function dragStart(event)
{
event.dataTransfer.setDragImage(document.getElementById('imageid'
),50,50);
}
```

addElement()

The `addElement()` method is used to set a data source for the drag and drop operation. The default data source is the draggable element itself.

Drag and Drop Animals

Let’s wrap up the HTML Drag and Drop API by creating a simple web application for dragging and dropping animal images into boxes bearing their names as shown in Figure 13. If an animal image goes to a box bearing the wrong name, it will not be allowed to drop there.

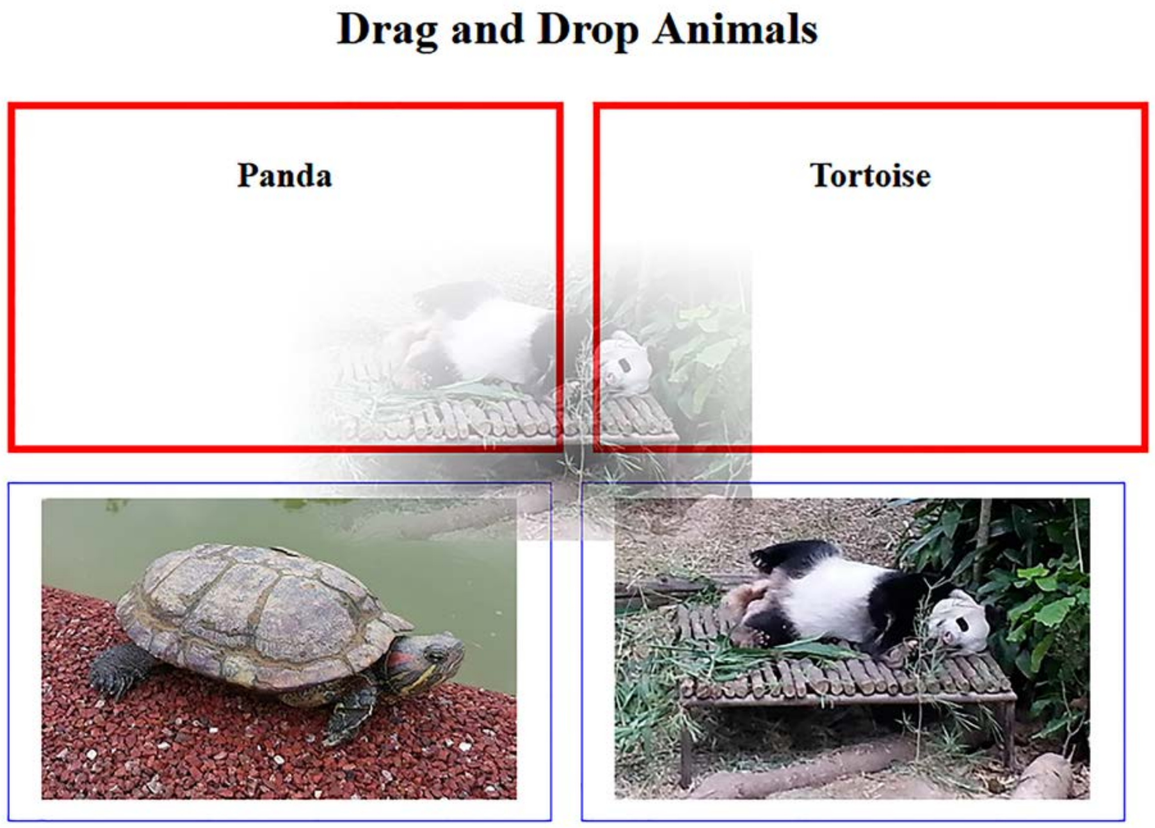


Figure 13 Drag and Drop Animals

This web page is named as “drag_n_drop_animals.html”. The code is given in Listing 6:

Listing 6 Source Code of drag_n_drop_animals.html

```
<!DOCTYPE HTML>
<html>
<head>
<title>Drag and Drop Animals</title>
<style>
body {
  text-align:center;
}
.animal_name, .animal_image {
  width: 800px;
  margin: auto;
}
.animal_name > div, .animal_image > div {
  float: left;
  padding: 10px;
  margin:10px;
  width: 350px;
  height: 210px;
}
.animal_name > div {
  border: 5px solid red
}
.animal_image > div {
  border: 1px solid blue
}
</style>
<script>
function dragOver(event)
{
  event.preventDefault();
}
function dragStart(event)
{
  event.dataTransfer.setData("image_name", event.target.id);
}
function drop(event)
{
  event.preventDefault();

  var data=event.dataTransfer.getData("image_name");
  if (data=="panda_image" && event.target.id=="panda_name")
  {
    event.target.innerHTML="";
    event.target.appendChild(document.getElementById(data));
  }
  else if (data=="tortoise_image" &&
event.target.id=="tortoise_name")
  {
    event.target.innerHTML="";
    event.target.appendChild(document.getElementById(data));
  }
}
</script>
</head>
<body>
<h1>Drag and Drop Animals</h1>
<div class="animal_name">
  <div id="panda_name" ondrop="drop(event)"
ondragover="dragOver(event)"><h2>Panda</h2></div>
  <div id="tortoise_name" ondrop="drop(event)"
ondragover="dragOver(event)"><h2>Tortoise</h2></div>
</div>
<br>
<div class="animal_image">
  <div><img id="tortoise_image" src="tortoise.jpg"
draggable="true" ondragstart="dragStart(event)" width="325"
height="206"></div>
  <div><img id="panda_image" src="panda.jpg" draggable="true"
ondragstart="dragStart(event)" width="325" height="206"></div>
</div>
</body>
</html>
```

The logic for determining whether to accept or reject the dropping of an animal image is elaborated below:

- First, in the `dragStart()` event handler, I assign “image_name” as the type, and the id of the dragged image, i.e. “panda_image” or “tortoise_image”, obtainable from event.target.id as the data to the `event.dataTransfer.setData()` mehod.

```
function dragStart(event)
```

```
{
  event.dataTransfer.setData("image_name",
event.target.id);
}
```

- In the `drop()` event handler, the code will look for the data of the droppable element identified by the type "image_name":

```
var data = event.dataTransfer.getData("image_name")
```

- The data of the draggable image id, i.e. "panda_image" or "tortoise_image", set in the `dragStart()` event handler above, and the id of the drop zone, i.e. "panda_name" or "tortoise_name", obtainable from event.target.id of the `drop()` event handler, will collectively determine if an animal image is allowed to drop into the drop zone.:

```
function drop(event)
{
  event.preventDefault();

  var data=event.dataTransfer.getData("image_name");

  if(data=="panda_image" && event.target.id=="panda_name")
  {
    event.target.innerHTML="";

event.target.appendChild(document.getElementById(data));
  }
  else if(data=="tortoise_image" &&
event.target.id=="tortoise_name")
  {
    event.target.innerHTML="";

event.target.appendChild(document.getElementById(data));
  }
}
```

4 SERVER-SENT EVENTS

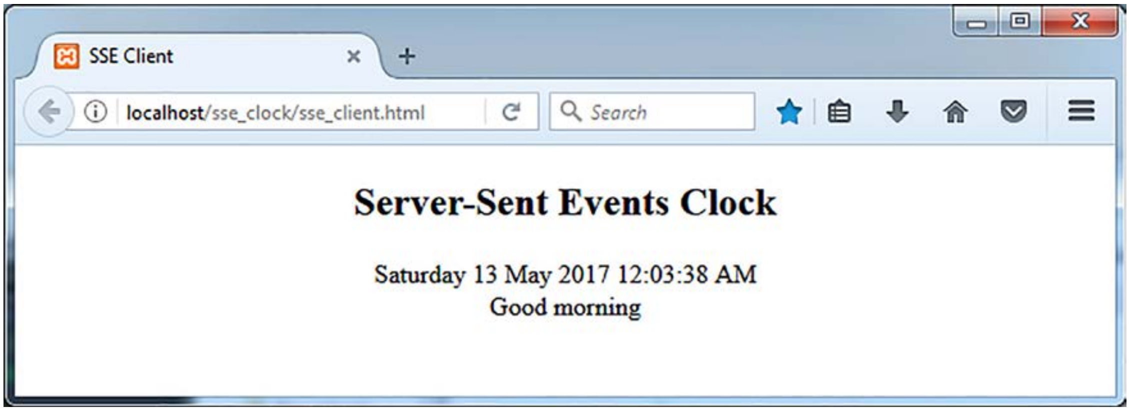
The traditional model of communication between a user and a website is that of a user's initiated request-response type. The user will initiate the request through a browser and await response from the web server. To check for any update on a web page, the user will have to hit the refresh/reload button on the browser to send a new request to the web server. In other words, the client must continuously poll the server for any updates. This is not very useful if the information is changing constantly and unpredictably and being able to receive it in real-time is pivotal for decision making, such as stock quotes, news feeds, weather forecasting, etc.

Here come **HTML Server-Send Events** (**SSE**) to the rescue! HTML SSE enables the server to send (push) information to the client when new information is available, eliminating the need for continuous polling. This has enabled websites to offer push services to their clients without the needs for any third-party plug-ins.

Let’s walk the talk over an exercise that creates a server-push clock.

Server-Push Clock

You will create a simple web application that receives continuous live update of time information from a web server using HTML SSE. Catch a glimpse of the expected outcome in Figure 14.



Server-Sent Events Clock

Saturday 13 May 2017 12:03:38 AM
Good morning

Figure 14 A Glimpse of Server-push Clock via SSE

The web application consists of two parts as follows:

- A client-side HTML page that opens a persistent connection between the browser and a server-side script on a web server, receives and displays the time information provided by the server-side script.
- A server-side script that streams time information to the HTML page on the browser continuously and asynchronously.

I have used PHP for the server-side script for this exercise, you may use any other scripts like PERL or Python. To test the web application, you will have to deploy it on a web server, such as an Apache.

Start by creating a folder called “sse_clock” on the web root of your Apache server Inside this folder, do as follows:

Server-side

Create a "sse_server.php" that contains the following code:

```
<?php
  //Change this to your time zone
  date_default_timezone_set("asia/singapore");

  header("Content-Type: text/event-stream");
  header("Cache-Control: no-store");
  $time=date("l j  F Y h:i:s A");
  if (date("A")=="AM")
    $greeting="Good morning";
  else
    $greeting="Good afternoon";

  // Formulate the data into a JSON string
  $json='{"time":"'.$time.'", "greeting":"'.$greeting.'"}';

  // Send the JSON string via a "data" key
  echo "data:$json";

  // Terminate each block of text with double newlines
  echo "\n\n";
?>
```

In this "sse_server.php", there are only three pieces of SSE related code, these are what they do:

- Set the `Content-Type` response header to `text/event-stream`:

```
header("Content-Type: text/event-stream");
```

- Indicate that the page should not be cached:

```
header("Cache-Control: no-store");
```

Send the data as JSON string via the **data** attribute. Although JSON string is used in this exercise, you can use any string formats, such as XML or comma delimited text. Whichever format you use, the data must be preceded by a `data` attribute and ends with two newlines, i.e. `\n\n` in PHP, as shown:

```
// Formulate the data into a JSON string
$json='{"time":"'.$time.'", "greeting":"'.$greeting.'"}';
```

```
// Send the JSON string via a "data" key
echo "data:$json";

echo "\n\n";
```

This data will be consumed by the **EventSource.onmessage** event listener over at the “sse_client.html” page. You will create this “sse_client.html” page.

Client-side

Create a "sse_client.html" that contains the following code:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>SSE Client</title>
</head>
<body style="text-align:center;">
<h2>Server-Sent Events Clock</h2>
<div id="time"></div>
<div id="greeting"></div>
<script>
if(typeof(EventSource)!==undefined) // Browser supports SSE
{
  var source=new EventSource("sse_server.php");
  source.onmessage=function(event)
  {
   var obj = JSON.parse(event.data);
   document.getElementById("time").innerHTML = obj.time;
   document.getElementById("greeting").innerHTML =
obj.greeting;
  };
}
else // Browser does not support SSE
{
  alert("Oop! This browser does not support HTML Server-Sent
Events.");
}
</script>
```

```
</body>
</html>
```

What the code does are:

- Verify if your browser supports SSE or not:

```
if(typeof(EventSource)!==undefined) //Browser supports SSE
{
  //omitted
}
else //Browser does not support SSE
{
  //omitted
}
```

- If it supports SSE, Create an **EventSource** object that takes the URL of the server-side script as parameter:

```
if(typeof(EventSource)!==undefined) //Browser supports SSE
{
  var source=new EventSource("sse_server.php");
  //omitted
}
```

- Register an event listener **EventSource.onmessage** to the EventSource object and attached it to an event handler (a function):

```
source.onmessage=function(event)
{
  //omitted
};
```

- Whenever a new message type arrives, the onmessage event will fire and its corresponding event handler will act:

```
source.onmessage=function(event)
{
  var obj = JSON.parse(event.data);
  document.getElementById("time").innerHTML = obj.time;
  document.getElementById("greeting").innerHTML =
obj.greeting;
};
```

In this exercise, the event handler will parse the data received via the **event.data** property into a JavaScript object and display its two properties, i.e. "time" and "greeting" on the web page the result of which is shown in Figure 14.

Custom Events

So far, you have used the **onmessage** event listener of the **EventSource** object to handle the default event type which is "**message**". In fact, you can create custom event types and custom event handlers to handle them.

Server-side

Create a server-side script called the "sse_server_custom_event.php" to stream the time information and greeting message via two events, e.g. `timeEvent` and `greetEvent`, respectively as shown:

```
<?php
  // Change this to your time zone
  date_default_timezone_set ("asia/singapore");

  header("Content-Type: text/event-stream");
  header("Cache-Control: no-store");

  $time = date("l j  F Y h:i:s A");
  if (date("A") == "AM")
    $greeting = "Good morning";
  else
    $greeting = "Good afternoon";
  echo "event: timeEvent\n";
  echo "data: $time";
  echo "\n\n";

  echo "event: greetEvent\n";
  echo "data: $greeting";
  echo "\n\n";
?>
```

In this “sse_server_custom_event.php”, an event called **timeEvent** and its associated data, i.e. the value of `$time` variable, are returned via the `event` and `data` attributes respectively. They are separated by a newline, i.e. `\n` in PHP, and terminated with two newlines as shown:

```
echo "event: timeEvent\n";
echo "data: $time";
echo "\n\n";
```

The same goes for the greeting message.

Client-side

Create a "sse_client_custom_event.html" that will receive the streamed data from the “sse_server_custom_event.php”:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>SSE Client</title>
</head>
<body style="text-align:center;">
<h2>Server-Sent Events Clock</h2>
<div id="time"></div>
<div id="greeting"></div>
<script>
if(typeof(EventSource)!==undefined) // Browser supports SSE
{
  var source=new EventSource("sse_server_custom_event.php");
  source.addEventListener('timeEvent', getTime, false);

  source.addEventListener('greetEvent', greet, false);

  function getTime(event){
   document.getElementById("time").innerHTML = event.data;
  };

  function greet(event){
   document.getElementById("greeting").innerHTML = event.data;
  };
}
else // Browser does not support SSE
{
  alert("Oop! This browser does not support HTML Server-Sent
Events.");;
}
</script>
</body>
</html>>
```

In this "sse_client_custom_event.html", a custom event listener is added to the **EventSource** object to listen for an event type called `timeEvent`, and fire an event handler called `getTime()` function whenever such an event occurs as shown below:

```
source.addEventListener('timeEvent', getTime, false);

function getTime(event){
  document.getElementById("time").innerHTML = event.data;
};
```

The same goes for the `greetEvent`.

Other Event Handlers

Besides the **onmessage** event handler, the **EventSource** object comes with two more event listeners — **onopen** and **onerror**.

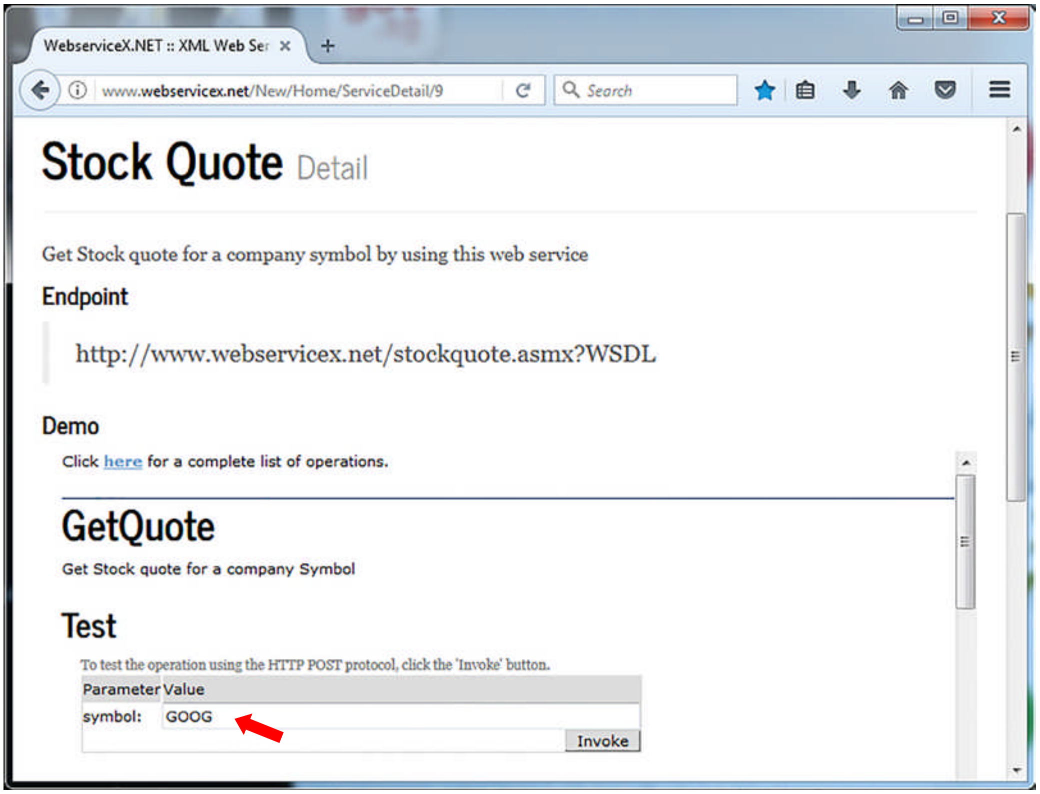
As the names imply, the **onopen** listener fires when a network connection is established while the **onerror** when unexpected errors such as networking errors, script errors, or decoding errors occur.

```
var source=new EventSource("some_script.php");

source.onopen = function(event) {
  console.log("Connection established.");
};

source.onerror = function(event) {
  console.log("Error occurred.");
};
```

Live Stock Quotes

To really appreciate the power of SSE, you shall create a SSE-incorporated web application to receive live stock quotes from a web service. The web service is provided by http://www.webservicex.net at this end point http://www.webservicex.net/stockquote.asmx?WSDL

How does this web service work? Head to this site http://www.webservicex.net/New/Home/ServiceDetail/9 and try out its demo by entering a company symbol, such as "GOOG" for Alphabet Inc., as shown in Figures 15 and 16.


WebserviceX.NET :: XML Web Ser
www.webservicex.net/New/Home/ServiceDetail/9
Search

Stock Quote Detail

Get Stock quote for a company symbol by using this web service

Endpoint

http://www.webservicex.net/stockquote.asmx?WSDL

Demo

Click here for a complete list of operations.

GetQuote

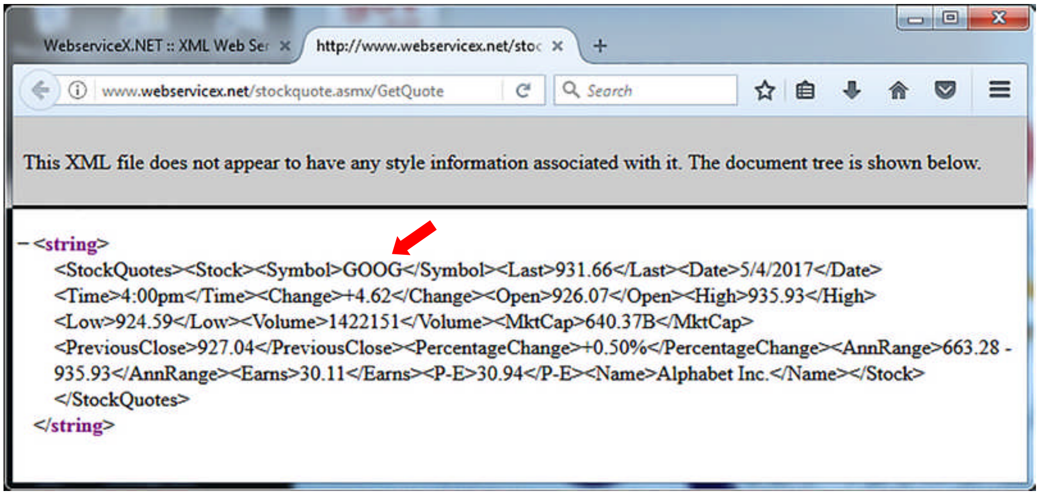
Get Stock quote for a company Symbol

Test

To test the operation using the HTTP POST protocol, click the 'Invoke' button.

Parameter	Value
symbol:	GOOG

Figure 15 Request Stock Quote


WebserviceX.NET :: XML Web Ser
http://www.webservicex.net/stoc
www.webservicex.net/stockquote.asmx/GetQuote
Search

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
- <string>
    <StockQuotes><Stock><Symbol>GOOG</Symbol><Last>931.66</Last><Date>5/4/2017</Date>
    <Time>4:00pm</Time><Change>+4.62</Change><Open>926.07</Open><High>935.93</High>
    <Low>924.59</Low><Volume>1422151</Volume><MktCap>640.37B</MktCap>
    <PreviousClose>927.04</PreviousClose><PercentageChange>+0.50%</PercentageChange><AnnRange>663.28 -
    935.93</AnnRange><Earns>30.11</Earns><P-E>30.94</P-E><Name>Alphabet Inc.</Name></Stock>
    </StockQuotes>
  </string>
```

Figure 16 Received Stock Quote

The site at http://www.webservicex.net/New/Home/ServiceDetail/9 also provides the detail on how to consume this web service from your code. You will find the protocols, including SOAP 1.1, SOAP 1.2, HTTP GET, and HTTP POST, and the parameters involved in consuming the web service. Check out the sample request and response for the respective protocols as follows:

SOAP 1.1

Request

```
POST /stockquote.asmx HTTP/1.1
Host: www.webservicex.net
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://www.webserviceX.NET/GetQuote"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <GetQuote xmlns="http://www.webserviceX.NET/">
      <symbol>string</symbol>
    </GetQuote>
  </soap:Body>
</soap:Envelope>
```

Response

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <GetQuoteResponse xmlns="http://www.webserviceX.NET/">
      <GetQuoteResult>string</GetQuoteResult>
    </GetQuoteResponse>
  </soap:Body>
</soap:Envelope>
```

SOAP 1.2

Request

```
POST /stockquote.asmx HTTP/1.1
Host: www.webservicex.net
Content-Type: application/soap+xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap12:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">
  <soap12:Body>
    <GetQuote xmlns="http://www.webserviceX.NET/">
      <symbol>string</symbol>
    </GetQuote>
  </soap12:Body>
</soap12:Envelope>
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap12:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">
  <soap12:Body>
    <GetQuoteResponse xmlns="http://www.webserviceX.NET/">
      <GetQuoteResult>string</GetQuoteResult>
    </GetQuoteResponse>
  </soap12:Body>
</soap12:Envelope>
```

HTTP GET

Request

```
GET /stockquote.asmx/GetQuote?symbol=string HTTP/1.1
Host: www.webservicex.net
```

Response

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<string xmlns="http://www.webserviceX.NET/">string</string>
```

HTTP POST

Request

```
POST /stockquote.asmx/GetQuote HTTP/1.1
Host: www.webservicex.net
Content-Type: application/x-www-form-urlencoded
Content-Length: length

symbol=string
```

Response

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<string xmlns="http://www.webserviceX.NET/">string</string>
```

The placeholder **string** is to be replaced by the actual value.

In a nutshell, you will request a stock quote from this web service at http://www.webservicex.net/stockquote.asmx?WSDL using one of the four protocols, and invoking the `GetQuote()` method with a "symbol" parameter that carries the stock symbol, e.g. GOOG, of a company whose stock quote you want to query. The stock quote from this requested will be displayed as XML on the browser if you use HTTP GET or HTTP POST, or returned as the value of the `GetQuoteResult` property using SOAP protocols.

To request the stock quote for GOOG via HTTP GET, for example, submit this URL through the browser as shown:

```
http://www.webservicex.net/stockquote.asmx/GetQuote?symbol=GOOG
```

You should get an output displayed on the browser like that shown in Figure 16.

Having learnt the detail of the stock quote web service, it's time to create your SSE-enabled web application to consume this web service.

Server-side

Create a "stockquote_server.php" that contains the following code:

```
<?php
header("Content-Type: text/event-stream");
header("Cache-Control: no-store");

$url = "http://www.webservicex.net/stockquote.asmx?wsdl";
$client = new SoapClient($url);

$quote = $client->GetQuote(array('symbol' => 'GOOG'));

$result =$quote->GetQuoteResult;

$xml = new SimpleXMLElement($result);

$data = '';

foreach ($xml->children()->children() as $child)
{
   $data .= $child->getName() . " - ". $child. "<br>";
}

echo "data: ".$data."\n\n";

?>
```

Apart from the SSE-enabled code that we have discussed in the previous exercise, there is the code that requests and receives the stock quote from the web service as explained below:

- Create a new SoapClient object initialized to the stock quote web service's end point:

```
$url = "http://www.webservicex.net/stockquote.asmx?wsdl";
$client = new SoapClient($url);
```

- Invoke the `GetQuote()` method of the web service and passing the stock symbol of the company as the parameter, e.g. GOOG:

```
$quote = $client->GetQuote(array('symbol' => 'GOOG'));
```

- Retrieve the returned stock quote data via the `GetQuoteResult` property:

```
$result =$quote->GetQuoteResult;
```

- Extract the XML-formatted stock quote data and organize them into key-value pairs:

```
$xml = new SimpleXMLElement($result);

$data = '';

foreach ($xml->children()->children() as $child)
{
   $data .= $child->getName() . " - ". $child. "<br>";
}
```

```
echo "data: ".$data."\n\n";
```

- Attach the key-value pairs to the `data` key and output it to the connected client:

```
echo "data: ".$data."\n\n";
```

This data will be captured by the **EventSource.onmessage** event listener over at the “stockquote_client.html” page.

Client-side

Create a "stockquote_client.html" that contains the following code:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Live Stock Quote</title>
</head>

<body>
<p id="stock_quote"></p>
<script>
if(typeof(EventSource)!==undefined) // Browser supports SSE
{
  var source=new EventSource('stockquote_server.php');
  source.onmessage=function(event)
  {
    var data = event.data.split('\n');

    document.getElementById('stock_quote').innerHTML = data;

  };
}
else // Browser does not support SSE
{
  alert("Oop! This browser does not support HTML Server-Sent
Events");;
}
```

```
</script>
</body>
</html>
```

What the code does is:

- Verify if your browser supports SSE or not:

```
if(typeof(EventSource)!==undefined) //Browser supports SSE
{
  //omitted
}
else //Browser does not support SSE
{
  //omitted
}
```

- If it supports SSE, Create an **EventSource** object that takes the URL of the server-side script as parameter:

```
if(typeof(EventSource)!==undefined) //Browser supports SSE
{
  var source=new EventSource("stockquote_server.php");
  //omitted
}
```

- Register an event listener **EventSource.onmessage** to the EventSource object and attached it to an event handler (a function):

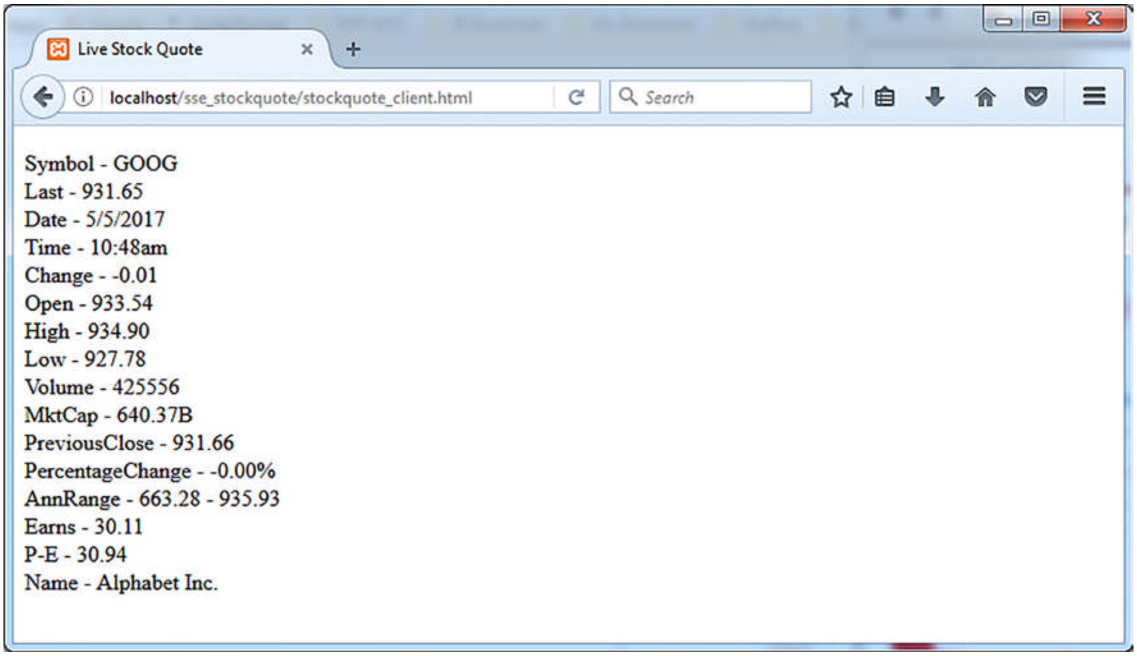
```
source.onmessage=function(event)
{
  //omitted
};
```

- Whenever a new message (stock quote data) arrives, the `onmessage` event will fire and its corresponding event handler will act:

```
source.onmessage=function(event)
{
  var stockQuote = event.data.split('\n');
  document.getElementById('stock_quote').innerHTML =
stockQuote;
};
```

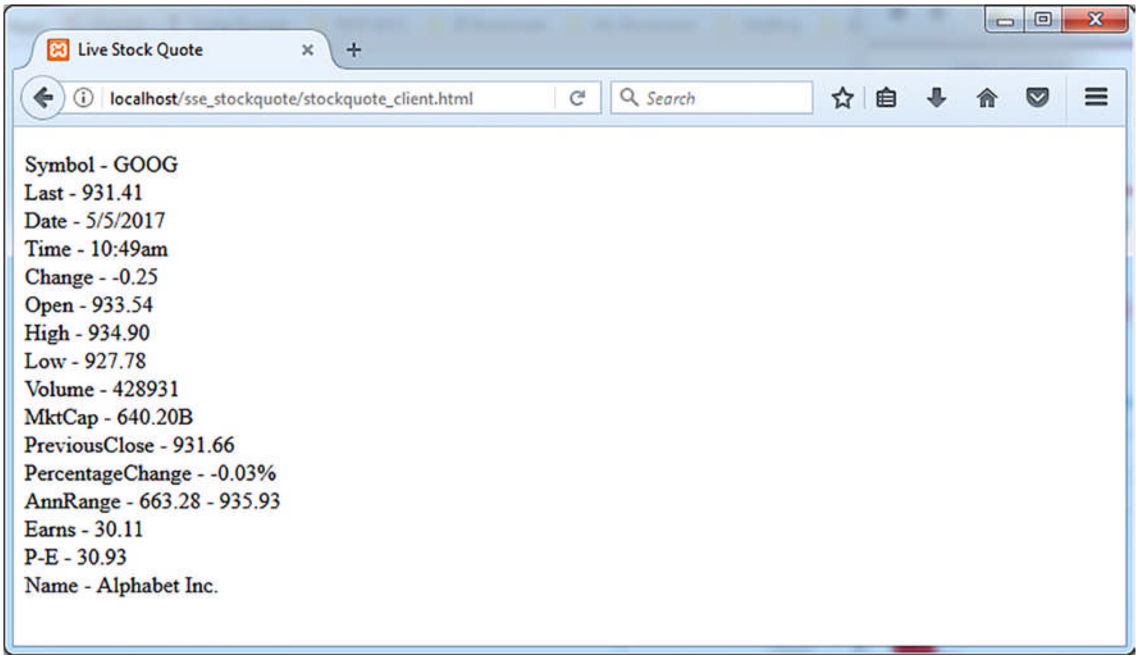
  In this exercise, the event handler simply splits the data received via the `event.data` property into separate key-value pairs and display them on the web page.

Launch this web application on the browser, you should see the constant update of stock quote data on the "stockquote_client.html" page as shown in these successive examples from Figures 17 to 18.



```
Symbol - GOOG
Last - 931.65
Date - 5/5/2017
Time - 10:48am
Change - -0.01
Open - 933.54
High - 934.90
Low - 927.78
Volume - 425556
MktCap - 640.37B
PreviousClose - 931.66
PercentageChange - -0.00%
AnnRange - 663.28 - 935.93
Earns - 30.11
P-E - 30.94
Name - Alphabet Inc.
```

Figure 17 Stock Quote



```
Symbol - GOOG
Last - 931.41
Date - 5/5/2017
Time - 10:49am
Change - -0.25
Open - 933.54
High - 934.90
Low - 927.78
Volume - 428931
MktCap - 640.20B
PreviousClose - 931.66
PercentageChange - -0.03%
AnnRange - 663.28 - 935.93
Earns - 30.11
P-E - 30.93
Name - Alphabet Inc.
```

Figure 18 Stock Quote

The update is being constantly pushed from "stockquote_server.php" which in turn polls the stock quote web service constantly for update.

Congratulation! You have successfully created a live stock quote web application with HTML Server-Side Events. As the requirement of this live stock quote web application only requires one-way read-only data sent from the server, it is well-suited for the role of Server-Sent Events API.

Server-Sent Events, however, cannot provide real-time full-duplex communication required by web applications like real-time chat apps and network games. Fret not! Here comes **HTML Web Sockets** to the rescue…

5 WEB SOCKETS

Since its inception, the web has been using Hypertext Transfer Protocol (HTTP) as the application protocol for data exchange between the client and the server. HTTP uses a request-response paradigm that works like this – a client, typically a web browser, initiates a request message to the server at some designated port, typically port 80. A web server listening at this port, on the other hand, upon receiving the complete request message, performs the necessary processing, and returns the result as a response message to the client. The cycle repeats for each request. While the HTTP model can meet the needs of most web applications, it is greatly handicapped in providing for highly interactive web applications, such as real-time chat and network games, that call for real-time simultaneous bidirectional data exchange or full-duplex communication in short. Workarounds like long-polling where the server holds a request open until the response information is available results in results in poor latency, high cost, and are less efficient. Not forgetting the burden of headers and cookie data that accompanied each HTTP polling request.

The real answer to providing for full-duplex communication over the web finally arrives with the introduction of **Web Socket Protocol** and the **HTML Web Socket API**.

It takes two hands to clap, at least. Although the focus of this chapter is on the HTML Web Socket API, you still need a Web Socket server to complete the discussion and exercises in this chapter. There are many libraries out there that can help you build a Web Socket server. In this chapter, you will make use of the “websocket” library of node.js to create a simple Web Socket server. Let’s go...

Web Socket Server

Follow these steps to set up a Node.js Web Socket Server on your computer.

- If you do not have Node.js installed on your computer, download and install Node.js from the official website at https://nodejs.org.

- Create a folder on your computer, say "C:\websocket", and navigate to it on a command terminal.

- On the command terminal, executing the following command which will install the "websocket" package required for Web Socket server inside a new sub-folder called "node_modules":

```
C:\websocket\npm install websocket
```

- Create a Node.js file called "server.js" with the code as shown in Listing 7. This code will create a Web Socket server.

Listing 7 Source Code server.js

```
var http = require('http');
var fs = require('fs');

var server = require('websocket').server;

var webSocket = new server({
  httpServer: http.createServer(function (req, res) {

    fs.readFile(__dirname + req.url,
      function (err, content) {

        if (err) {
          res.writeHead(500);
          return res.end('Error loading '+req.url);
        }

        res.writeHead(200, {'Content-Type': 'text/HTML'});
        res.write(content);
        res.end();
    });

  }).listen(8080)
});

webSocket.on('request', function(request) {

  var connection = request.accept(null, request.origin);

  connection.sendUTF('Protocol switched successfully!');
});
```

- On the command terminal, executing the following command to start the server that listen on port 8080:

```
C:\websocket\node server.js
```

You have created and launched a Web Socket server. You are ready to create a web client that will interact with the Web Socket server using HTML Web Socket API.

Protocol Switching

The Web Socket protocol was designed to work seamlessly with the existing Web infrastructure. In order to achieve backwards compatibility with pre-Web Socket web, the Web Socket protocol will start with a handshake of protocol switching, then the actual data transfer. In a nutshell, every new Web Socket connection starts with a handshake process whereby an HTTP upgrade request is being sent to the server requesting switching to Web Socket protocol. The server, if it supports the Web Socket protocol, will respond by switching the original HTTP protocol to Web Socket one to complete the handshake process. After the completion of this handshake process, the client and server can start to exchange data in full-duplex way using the Web Socket protocol. Let's create a Web Socket client to demonstrate this protocol switching process. Follow me…

Inside the folder of the "server.js", create a HTML page called "index.html" with the code as shown in Listing 8.

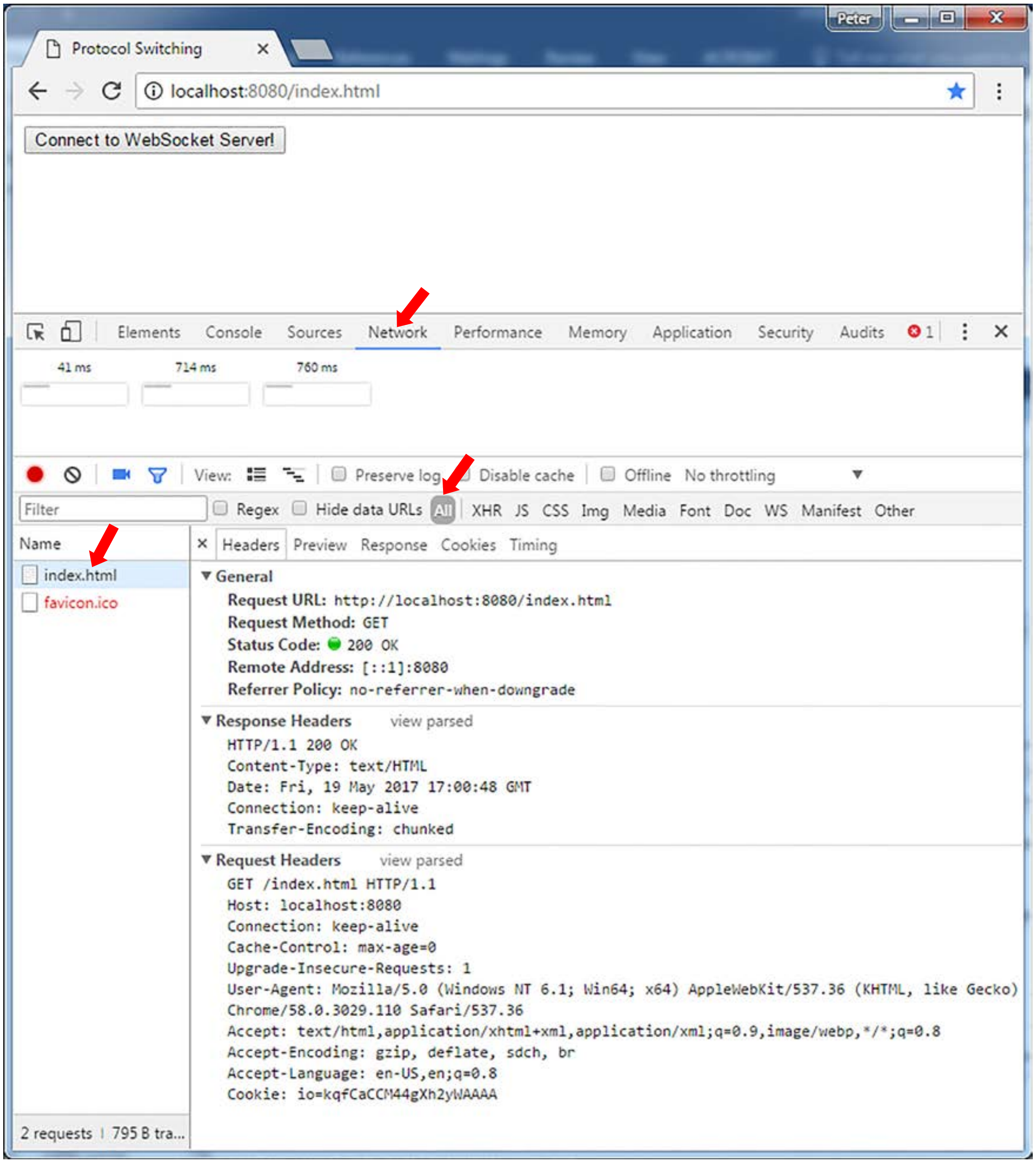
Listing 8 Source Code of index.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Protocol Switching</title>
</head>
<body>
<button type="button" onclick="connect()">Connect to WebSocket
Server!</button>
<script>
if ("WebSocket" in window) // Browser supports HTML WebSocket
{
  function connect()
  {
   // Instantiate a WebSocket object
   webSocket = new WebSocket('ws://localhost:8080/');

   // Event handler to act on receiving message from server
   webSocket.onmessage = function (message)
   {
     alert(message.data);
   };
  };
}
else
{
  alert("Oop! This browser does not support HTML WebSocket.");
}
</script>
</body>
</html>
```

Launch the “index.html” on a browser using this URL:

```
http://localhost:8080/index.html
```

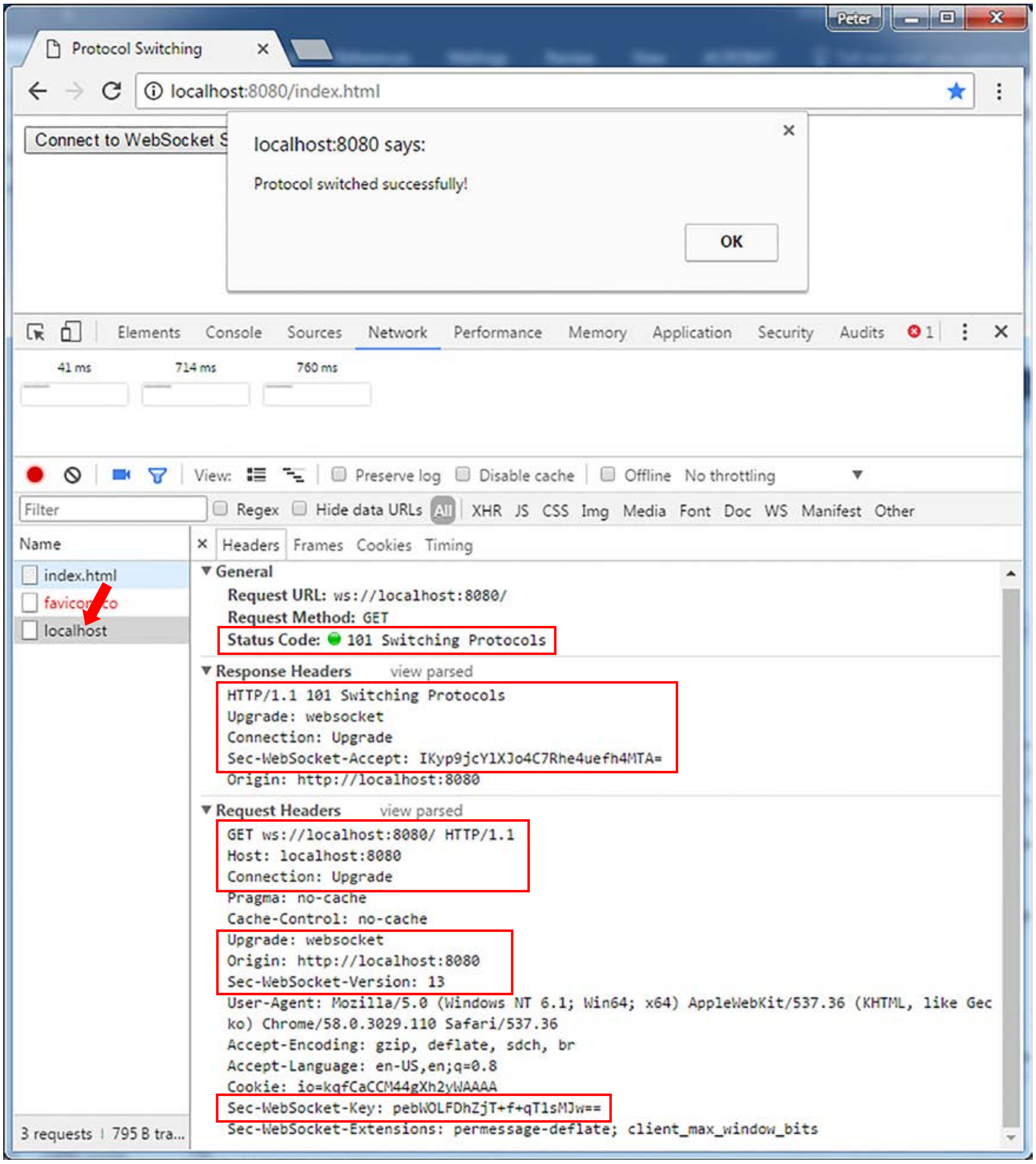
The “index.html” consists of only a button that will request a Web Socket connection upon clicked. Do not click it yet, instead peek at the header information using the browser’s developer tools[2]. It shows the information for request header and response header respectively under the HTTP protocol, an example of which is shown in Figure 19 on Google Chrome.


Protocol Switching
localhost:8080/index.html
Connect to WebSocket Server!
Elements Console Sources Network Performance Memory Application Security Audits
41 ms 714 ms 760 ms
View: Preserve log Disable cache Offline No throttling
Filter Regex Hide data URLs All XHR JS CSS Img Media Font Doc WS Manifest Other
Name
index.html
favicon.ico
Headers Preview Response Cookies Timing
2 requests | 795 B tra...

```
▼ General
   Request URL: http://localhost:8080/index.html
   Request Method: GET
   Status Code: 200 OK
   Remote Address: [::1]:8080
   Referrer Policy: no-referrer-when-downgrade
▼ Response Headers      view parsed
   HTTP/1.1 200 OK
   Content-Type: text/HTML
   Date: Fri, 19 May 2017 17:00:48 GMT
   Connection: keep-alive
   Transfer-Encoding: chunked
▼ Request Headers      view parsed
   GET /index.html HTTP/1.1
   Host: localhost:8080
   Connection: keep-alive
   Cache-Control: max-age=0
   Upgrade-Insecure-Requests: 1
   User-Agent: Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
   Chrome/58.0.3029.110 Safari/537.36
   Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
   Accept-Encoding: gzip, deflate, sdch, br
   Accept-Language: en-US,en;q=0.8
   Cookie: io=kqfCaCCM44gXh2yWAAAA
```

Figure 19 HTTP Connection

Click the button to establish a connection to the Node.js Web Socket server at http://localhost:8080/. View the information pertaining to the protocol switching of the handshaking process using the developer tools, an example of which is shown in Figure 20.


Protocol Switching
localhost:8080/index.html
Connect to WebSocket S
Elements Console Sources Network Performance Memory Application Security Audits
41 ms 714 ms 760 ms
View: Preserve log Disable cache Offline No throttling
Filter Regex Hide data URLs All XHR JS CSS Img Media Font Doc WS Manifest Other
Name
index.html
favicon.ico
localhost
3 requests | 795 B tra...
Headers Frames Cookies Timing

localhost:8080 says:

Protocol switched successfully!

OK

```
General
  Request URL: ws://localhost:8080/
  Request Method: GET
  Status Code: 101 Switching Protocols
Response Headers    view parsed
  HTTP/1.1 101 Switching Protocols
  Upgrade: websocket
  Connection: Upgrade
  Sec-WebSocket-Accept: IKyp9jcY1XJo4C7Rhe4uefh4MTA=
  Origin: http://localhost:8080
Request Headers    view parsed
  GET ws://localhost:8080/ HTTP/1.1
  Host: localhost:8080
  Connection: Upgrade
  Pragma: no-cache
  Cache-Control: no-cache
  Upgrade: websocket
  Origin: http://localhost:8080
  Sec-WebSocket-Version: 13
  User-Agent: Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gec
  ko) Chrome/58.0.3029.110 Safari/537.36
  Accept-Encoding: gzip, deflate, sdch, br
  Accept-Language: en-US,en;q=0.8
  Cookie: io=kqfCaCCM44gXh2yWAAAA
  Sec-WebSocket-Key: pebWOLFDhZjT+f+qT1sMJw==
  Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits
```

Figure 20 Switching from HTTP to Web Socket Protocol

The connection request to the Web Socket server is initiated by instantiating a Web Socket object called “webSocket” which starts the handshake that resulted in the protocol switching:

```
// Instantiate a WebSocket object
webSocket = new WebSocket('ws://localhost:8080/');
```

Once the Web Socket connection is established, the Web Socket server sends a message to the client using the following code:

```
webSocket.on('request', function(request) {

  var connection = request.accept(null, request.origin);

  connection.sendUTF('Protocol switched successfully!');
})
```

The client, “index.html”, on the other hand, detects the incoming message through the **message** event of the Web Socket object called “webSocket”, which then fires the **onmessage** event handler which simply displays the incoming message from the server in a dialog box using the following code:

```
webSocket.onmessage = function (message) {
  alert(message.data);
};
```

[2]To launch the developer tools, press CTRL+SHIFT+I on Google Chrome and Firefox, or F12 on IE. I will proceed to use the developer tools in Google Chrome as shown in Figure 18.

Web Socket API

In the preceding exercise, you have experienced firsthand the ease of establishing a Web Socket connection and the receipt of a message from the Web Socket server using a Web Socket object and its onmessage event handler provided by the HTML Web Socket API. The introduction of HTML Web Socket API has greatly facilitated the development of real-time web applications based on the Web Socket protocol.

Essentially, a Web Socket object of the HTML Web Socket API possesses two methods, two properties, and four events.

Methods

The HTML Web Socket API possesses two methods – **send()** and **close()**. Check out their descriptions in table 4.

Table 4: Web Socket Methods

Method	**Description**
send(message)	The send() method sends messages (text or binary), from the client to the server.
close()	The close() method terminates the connection with the server.

Properties

A Web Socket object possesses two properties – **readyState** and **bufferedAmount**. Check out their descriptions in table 5.

Table 5: Web Socket Properties

Property	Description
readyState	This is a read-only property that represents the state of the Web Socket connection using one of the following four values: • **0** indicates that the connection has not been established yet. • **1** indicates that the connection is established and communication is possible. • **2** indicates that the connection is going through the closing handshake, or the close() method has been invoked. • **3** indicates that the connection has been closed or cannot be opened.
bufferedAmount	This is a read-only property that represents the number of bytes of application data ((UTF-8 text and binary data) that have been queued using the send() method.

Events

The HTML Web Socket API enables real-time full-duplex communication based on an event-driven model that comprises four events – **open**, **message**, **close**, and **error**. Check out their descriptions in table 6.

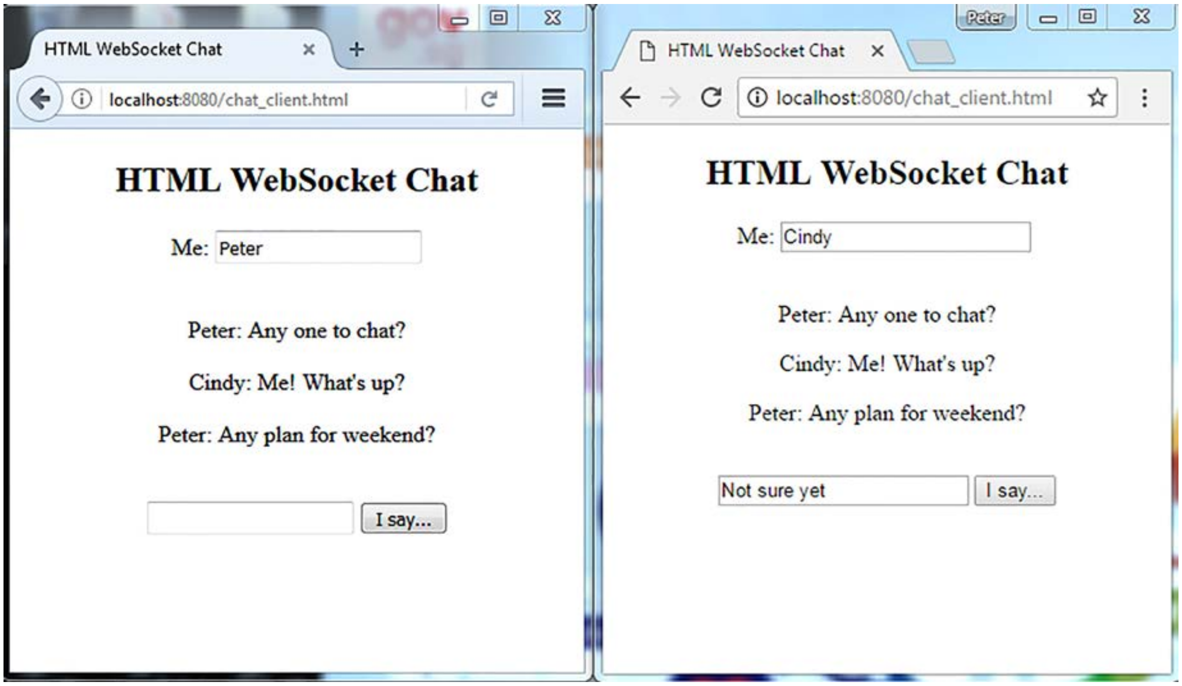
Table 6: Web Socket Events

Event	Event Handler	Description
open	onopen	The open event is triggered on the establishment of a connection as a result of the handshake between the client and the server. Any follow-up action pertaining to this event will be handled by the onopen event handler.
message	onmessage	The message event is triggered on receipt of a message from the client. Any follow-up action pertaining to this event will be handled by the onmessage event handler.

close	onclose	The close event is triggered when a connection is terminated, be it intentionally or error. Any follow-up action pertaining to this event will be handled by the onclose event handler.
error	onerror	The error event is triggered when an error occurs during communication. An error event always results in the termination of connection. Any follow-up action will be handled by the onerror event handler.

Web Chat Application

Having learned the fundamentals of HTML Web Socket API, let's harness its power to create a simple yet truly full-duplex web chat application based on the Web Socket protocol. Catch a glimpse of this web chat application in demonstration between two chat participants as shown in Figure 21.



Peter: Any one to chat?

Cindy: Me! What's up?

Peter: Any plan for weekend?

Figure 21 Web Socket-based Chat Application

The message from a sender will be sent to the Web Socket server which in turn broadcasts it to all connected chat participants (including the sender) in real-time and full-duplex manner.

You will start by building the chat server.

Chat Server

Inside the folder that you have created in the preceding exercise, create a Node.js file called "chat_server.js" with the code as shown in Listing 9. This code will create a Web Socket-based chat server.

Listing 9 Source Code of chat_server.js

```
var http = require('http');
var fs = require('fs');

var server = require('websocket').server;

var webSocket = new server({
  httpServer: http.createServer(function (req, res) {

    fs.readFile(__dirname + req.url,
      function (err, content) {

        if (err) {
          res.writeHead(500);
          return res.end('Error loading '+req.url);
        }

        res.writeHead(200, {'Content-Type': 'text/HTML'});
        res.write(content);
        res.end();
    });

  }).listen(8080)
});

var clients = [];

webSocket.on('request', function(request) {
  var connection = request.accept(null, request.origin);

  clients.push(connection);
  connection.on('message', function(message) {

    // Relay to all chat participants on each new message
received
    for(var i = 0; i < clients.length; i++) {
        clients[i].sendUTF(message.utf8Data);
    }

  });

  connection.on('close', function(connection) {
      console.log('connection closed');
  });
});
```

Apart from creating a Web Socket server which you have learned earlier, the code will:

- Store the connection object of each web client (chat participant) in an array called "clients" as shown:

```
var clients = [];

webSocket.on('request', function(request) {
  var connection = request.accept(null, request.origin);
  clients.push(connection);
//omitted
```

- So that the server can relay any new messages it receives to all participants by looping through the whole "clients" array as shown:

```
connection.on('message', function(message) {

  for(var i = 0; i < clients.length; i++) {
    clients[i].sendUTF(message.utf8Data);
  }
});
```

If the Web Socket server by “server.js” for the previous exercise is still running, stop it by hitting CTRL+C on the command terminal. You can now launch this new chat server by “chat_server.js” by executing the following command on the command terminal:

```
C:\websocket\node chat_server.js
```

You have created and launched a Web Socket-based chat server. You are ready to create a chat client that will interact with this server using HTML Web Socket API.

Chat Client

Inside the folder of the “chat_server.js”, create a HTML page called “chat_client.html” with the code as shown in Listing 10.

Listing 10 Source Code of chat_client.html

```
<!doctype html>
<html>
<head>
<meta charset="utf-8">
<title>HTML WebSocket Chat</title>
</head>
<body style='text-align: center'>
<h2>HTML WebSocket Chat</h2>
<label for='myName'>Me: </label>
<input type='text' id='myName' value='Anonymous'>
<br><br>
<div id='allMessages'></div>
<br>
<input type='text' id='myMessage' value='Silent is gold?'>
<button type='button' onClick='send()'>I say...</button>
<script>
var myName = document.getElementById('myName');
var myMessage = document.getElementById('myMessage');
var allMessage = document.getElementById('allMessages');
if ("WebSocket" in window) // Browser supports HTML WebSocket
{
  var webSocket = new WebSocket('ws://localhost:8080/');

  function send()
  {
    var message = myName.value + ': ' + myMessage.value;

    webSocket.send(message);

    myMessage.value = '';
  }

  webSocket.onopen = function ()
  {
    console.log('Connection established.');
  };

  webSocket.onmessage = function (event)
  {
    allMessage.innerHTML += '<p>'+ event.data +'</p>';
  };

  webSocket.onclose = function ()
  {
    console.log('WebSocket Closed.');
  };

  webSocket.onerror = function (error)
  {
    console.log('WebSocket error: ' + error);
  }
}
else
{
  alert("Oop! This browser does not support HTML WebSocket.");
}
</script>
</body>
</html>
```

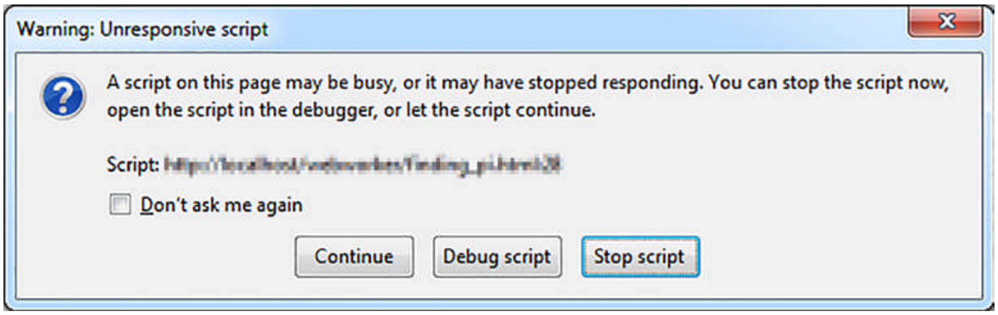
Launch the “chat_client.html” on separate browser tabs or different browsers using this URL:

```
http://localhost:8080/chat_client.html
```

You should see the “chat_client.html” being rendered as shown in Figure 21. Any message sent from a browser page will appear on all connected browser pages including the sender page. If you want to chat from different computers, you will have to replace the "localhost" of the URL to the actual IP address of the computer where the "chat_server.js" is running. Enjoy!

6 WEB WORKERS

JavaScript runs in a single-threaded asynchronous environment. Only a piece of code is executed at any one time while events such as mouse clicks and timers can occur asynchronously, they must queue up for their turn. In other words, events can happen anytime but they must wait for their turn to be handled one at a time. Although single-threaded execution is fine for the mundane computing needs of most web applications, it is not the case for use cases that involve long-running processes, such as machine learning, data analytics, and I/O polling for results. The long-running of one script can potentially block other scripts, freeze up the user interface, and eventually crash the application, resulting in poor user experience. The last thing you want is to be issued with this pop-up warning from your browser as shown in Figure 22.



Warning: Unresponsive script

A script on this page may be busy, or it may have stopped responding. You can stop the script now, open the script in the debugger, or let the script continue.

Script: [illegible]

Figure 22 Unresponsive Script Dialog

Programming languages like Java and C# will have delegated such long-running tasks to background threads which are impossible with JavaScript's single-thread nature. However, this is set to change with the introduction of **Web Workers API** by the HTML Living Standard. With Web Workers API, your web application can now spawn separate threads to run long-running scripts in the background, eliminating the annoyances of blocking other scripts and unresponsive UI.

What better way to experience the power of the Web Worker API than to get your hands dirty in creating one. The challenge is to find a problem that is simple to implement as an exercise yet involves intensive computation that befits the purpose of Web Worker API. One of the good candidates is the computation of Π. The algorithm is simple but the process can go on forever. Let's begin…

Computing Π with Classic JavaScript

Start by creating an HTML page called "pi.html" with the code in Listing 11. This is classic JavaScript that simply computes the value of Π over a number of iterations provided by the user. I have deliberately divided the script into two functions — the `getIteration()` function for UI-related code, and the `computePi()` function for the actual computation. There is a good reason for doing this. You will find out soon.

Listing 11 Source Code of pi.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Compute Pi</title>
</head>
<body>
<button type="button" onclick="getIteration()">Compute
pi</button>
<p id="piValue"></p>
<script>
// UI-related code
function getIteration()
{
  do {
    var iteration = prompt("Enter number of iterations:","100");

    if (iteration === null) return;

    var i = parseInt(iteration);
    var f = parseFloat(iteration);
  } while (isNaN(i) || i != f || i<=0)

  var pi = computePi(iteration);

  document.getElementById("piValue").textContent = pi;
}
// Actual computational code
function computePi(iteration)
{
  var pi=0, n=1;
  for (var c=0; c<=iteration; c++) {
   pi += (4/n)-(4/(n+2));
   n += 4;
  }
  return pi;
}
</script>
</body>
</html>
```

As the number of iterations increases, the responsiveness of the browser slows down gradually as shown in Figure 23. Eventually, it will freeze up and some warning like that dialog in Figure 22 will show up:

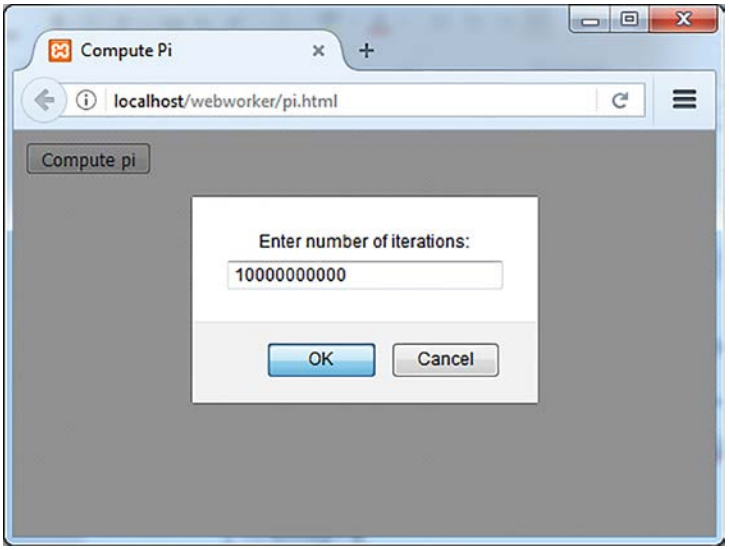


Figure 23 pi.html

It's time to call upon the help of the HTML Web Workers API.

Web Workers API

A **Web Worker** is a JavaScript running in the background without affecting other scripts and the UI thread. The HTML Living Standard provides for two types of web workers — **Dedicated Workers** and **Shared Workers**. Dedicated workers are linked to their creator. Shared workers, on the other hand, can be shared by multiple clients running in different browsing contexts like windows and iframes, if they are in the same domain as the web worker.

One limitation of the Web Workers is that they cannot access the DOM. In other words, a web worker cannot read or modify the HTML document. Instead, the worker will have to rely on their creating HTML pages to read and modify the DOM on their behalf. This is achieved by passing messages between the HTML page and the web worker.

Dedicated Workers

Typically, a web worker, be it dedicated or shared, is created as an external JavaScript file. Apart from those long-running scripts, every dedicated web worker uses an **onmessage** event handler to receive incoming messages, and a **postMessage()** method to send messages. A template of such a dedicated worker is shown below:

```
// Long-running script...

onmessage = function(event){ ... };

postMessage(message);
```

Once the worker file is created, it is ready to be spawned as a background service from a HTML page. Let's call this worker file "dedicated_worker.js". In the HTML page, start by checking if the browser supports the HTML Web Workers API with this code:

```
if (typeof(Worker)!==undefined)
{
    // Go ahead to use web worker
}
else
{
    // Oop! no luck...
}
```

Once the browser's support is confirmed, create a new Dedicated Worker object and launch the "dedicated_worker.js" as a deliciated worker in the background as shown:

```
if (typeof(Worker)!==undefined)
{
  var dedicated_worker = new Worker("dedicated_worker.js");
}
```

The Dedicated Worker object uses an **onmessage** event handler to receive incoming messages stored in **event.data** from the "dedicated_worker.js" as shown:

```
dedicated_worker.onmessage = function(event){
  alert(event.data);
};
```

The HTML page can send messages to the "dedicated_worker.js" using a **postMessage()** method as shown:

```
dedicated_worker.postMessage('some message'); // simple text
dedicated_worker.postMessage({'id': 'some id', 'data_array':
['data1', 'data2', 'data3']}) // structured data
```

A running dedicated worker can be terminated by calling the `terminate()` method of the Dedicated Worker object from the HTML page as shown:

```
dedicated_worker.terminate();
```

Alternatively, the dedicated worker can close itself by calling its own `close()` method as shown:

```
close();
```

Shared Workers

Inside a shared worker, e.g. an external JavaScript file called "shared_worker.js", new client connections are notified via the **onconnect** event handler. Each of the connecting client will be given a port, stored in the **event.source** property, where the worker uses an **onmessage** event handler to receive incoming messages, and a **postMessage()** method to send messages. A template of such a shared worker is shown below:

```
onconnect = function (event)
{
  var port = event.source;
  // set up a listener
  port.onmessage = function(event){ ... };
  // send a message back to the port
  port.postMessage('some message');
};
```

Instead of the onmessage event handler, if the message event is handled using **addEventListener**, the port must be manually started using its `start()` method as shown:

```
port.addEventListener('message', function(event) {
  // do something
});

port.start();
```

On the HTML page, create a new Shared Worker object and launch the "shared_worker.js" as a shared worker in the background as shown:

```
if (typeof(Worker)!==undefined)
{
  var shared_worker = new SharedWorker("shared_worker.js");
}
```

Communication with the shared worker must be carried out via the port property of the Shared Worker object as shown:

```
shared_worker.port.onmessage = function(event){ ... };

shared_worker.port.postMessage('some message'); // simple text

shared_worker.port.postMessage({'id': 'some id', 'data_array':
['data1', 'data2', 'data3']}) // structured data
```

Instead of the onmessage event handler, if the message event is handled using **addEventListener**, the port must be manually started using its `start()` method as shown:

```
shared_worker.port.addEventListener('message', function(event)
{
  // do something
});

shared_worker.port.start();
```

Computing Π with Dedicated Workers

Create a script file called "worker_pi.js" save it in the same folder as the "pi.html". This "worker_pi.js" will be the web worker that contain the scripts that will run in a background thread. Add the code as follows:

- In the "pi.html", what is the candidate code that is potentially long-running and should ideally be executed in a background thread? The answer is none other than the `computePi()` function. So, move the `computePi()` function, from the "pi.html" to the “worker_pi.js”, then replace the “return pi” statement with a `postMessage()` method through which the web worker can post the result to the “pi.html”.

```
function computePi(iteration)
{
  var pi=0, n=1;
  for (var c=0; c<=iteration; c++) {
   pi += (4/n)-(4/(n+2));
   n += 4;
  }
  //return pi;
  postMessage({'piValue': pi});
}
```

- The web worker cannot access the DOM. In our example, “worker_pi.js” will receive the iteration value from the “pi.html” via the onmessage event handler as shown:

```
onmessage = function(event)
{
  computePi(event.data.iteration);
}
```

- Switch to the “pi.html”, comment out or remove `var pi = computePi(iteration);`, then add the code to instantiate a web worker object that will load and execute the “worker_pi.js” as the worker script in the background, and then send the value of iteration as a message to this worker script as shown:

```
//var pi = computePi(iteration);
// Instantiate a web worker
var worker = new Worker('worker_pi.js');
// start and send a message to the worker script
worker.postMessage({'iteration': iteration});
```

- Lastly, comment out or remove `document.getElementById("piValue").textContent = pi;`, then add this code to implement the onmessage event handler of the web worker object to receive the computed value returned from the "worker_pi.js" and display it on the UI as shown:

```
//document.getElementById("piValue").textContent = pi;
// Event handler for incoming message
worker.onmessage = function(event)
{
  document.getElementById("piValue").textContent =
event.data.piValue;
}
```

The complete code of "worker_pi.js" and the modified code of "pi.html" are shown in Listings 12 and 13 respectively

Listing 12 Source Code of worker_pi.js

```
function computePi(iteration)
{
  var pi=0, n=1;
  for (var c=0; c<=iteration; c++)
  {
   pi += (4/n)-(4/(n+2));
   n += 4;
  }
  postMessage({'piValue': pi});
}
onmessage = function(event)
{
  computePi(event.data.iteration);
}
```

Listing 13 Source Code of Modified pi.htm

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Compute Pi</title>
</head>
<body>
```

```
<button type="button" onclick="getIteration()">Compute
pi</button>
<p id="piValue"></p>
<script>
function getIteration()
{
  do {
   var iteration = prompt("Enter number of iterations:","100");

   if (iteration === null) return;

   var i = parseInt(iteration);
   var f = parseFloat(iteration);

  } while (isNaN(i) || i != f || i<=0)

  var worker = new Worker('worker_pi.js');

  worker.postMessage({'iteration': iteration});

  worker.onmessage = function(event)
  {
   document.getElementById("piValue").textContent =
event.data.piValue;
  }
}
</script>
</body>
</html>
```

Test it out and see that the browser does not freeze up for very large iteration value any more. However, you still need to wait patiently for the result from a very large iteration value. Note that this code will work on Firefox and IE, but not on Google Chrome. Chrome does not allow running Web Workers API from a `file://URI`, it will work if you deploy it on a web server.

6 WEB STORAGE

The World Wide Web (WWW) is originally stateless. HTTP (HyperText Transfer Protocol) - a set of rules that governs communications on the WWW, provides no means for maintaining states. Using HTTP alone, every request for a web page is a new and isolated one. In other words, you are anonymous or faceless to the website as it does not recognize you or remember your previous visits. This stateless WWW can be liken to your affair with the vending machines you frequent to buy your favorite beverages and snacks. These machines do not recognize you or remember your previous purchases. This type of model is fine if each transaction is independent and can be completed in a single cycle of request (press the product button) and response (out come the product).

The stateless model of WWW cannot meet the sophisticated demands of today's web applications. A typical e-commence transaction session on the web, for example, will consists of a connected series of consecutive and iterative cycle of requests and responses - From browsing and searching items, adding items to cart, displaying cart content, changing cart items (iterative), changing item quantities (iterative), to checking out all within a single session. For this to happen, the e-commerce website must recognize you and remember your preceding activities with the website in order for it to correctly process your subsequent requests such as displaying cart content, changing cart items etc. In order words, today's web applications must employ some mechanism to maintain states in order to function properly.

To mend this shortfall, cookies were invented

Cookies No Enough

Cookies, not your favorite edible snacks, were invented in the early years to meet the demands of maintaining states by websites. A cookie is a little text file of no more than 4KB that stores a user's information pertaining to a particular website. When a user visits a website with a cookie function for the first time, a cookie is created and sent from the web server to the browser and stored with the browser in the user's computer. This cookie will accompany subsequent visits (requests) to the same website using the same browser and from the same computer. In this way, the website will be able to recognize you (or rather the browser and the computer) and any information on your preceding visits that were captured in the cookie.

Cookies have its intrinsic limitations since day one. These have been exacerbated by the ever-increasing sophistication of modern web applications.

- Limited capacity of 4KB is hardly enough and useful.
- Performance adversity as cookies have to accompany each request to be server.
- Security vulnerability such as cross site scripting attack.

Another alternative to maintaining states is through the use of session variables which are created and stored temporarily on the web server. As its name suggests, a session variable only survives for a single session. To keep the value of a session variable beyond a single session, it must be saved to some persistence storage such as a database or a flat file on the server side. This poses resource overhead and increased complexity in implementation.

Are there better alternatives? Yes, the answer is **HTML Web Storage API** provided by the HTML Living Standard.

Web Storage API

HTML Web Storage API was designed with the vision for a better mechanism to store web data on the client-side. It is implemented as a client-side database provided by a web browser that allows web pages to store data in the form of key-value pairs. It has the following characteristics:

- Stores up to 5MB of data per origin (website / domain).

- Has properties and methods that you can access using JavaScript for manipulating data in the web storage.

- Like cookies, you can choose to make the data stays (persists) even after you navigate away from the website, close the browser tab, exit the browser, or shut down your computer.

- Unlike cookies, which are created by server-side scripting, web storage is created by client-side scripting like JavaScript.

- Unlike cookies, the data in the web storage does not automatically accompany every HTTP request to the server.

- Is implemented and supported natively in major web browsers like Chrome, Opera, Firefox, Safari and IE8+. In other words, no 3rd-party plugins are needed.

Web Storage offers two different storage areas — **Session Storage** and **Local Storage** — which differ in scope, lifespan, and are suited for different situations.

Session Storage

Session Storage, as the name suggests, **stores data as strings (texts) pertaining to an origin[3] and lasts only for current session**. The data is deleted when the browser window is close. Session Storage is designed for situations where a user carries out multiple transactions in different browser windows concurrently using the same website. Each transaction in each browser windows will get its own copy of session storage which is different from the session storage used by another transaction in another browser windows. When the user closes a browser window, the session storage data that belongs to that window will cease to exist. In this way, the transaction data would not "leak" from one browser window to the others. Session storage is the answer to the following weakness of cookies as stated in the HTML Web Storage specification:

```
Quote:
if a user buying plane tickets in two different windows, using
the same site. If the site used cookies to keep track of which
ticket the user was buying, then as the user clicked from page
to page in both windows, the ticket currently being purchased
would "leak" from one window to the other, potentially causing
the user to buy two tickets for the same flight without really
noticing.
```

Session storage must be used for web activities that deal with confidential and sensitive information, such as credit card numbers, social security numbers, and login credentials. This information are easy prey for DNS spoofing[4] attacks and should not be stored beyond a single session.

Local Storage

Local Storage, on the other hand, stores data as strings (texts) pertaining to an origin and lasts forever (unless you delete it explicitly). The data will stay (persist) even when the browser window is close and be available on subsequent visits to the same origin using the same browser window. Local Storage is designed for storing data that spans multiple browser windows and lasts beyond the current session. A simple use case would be to store the number of visits a person has made to some web pages.

Unlike their desktop counterparts, web applications have always been lacking in their ability to work offline. Not anymore, the advent of HTML5 Local storage has made its possible now. Imagine while you are filling out a multi-page web form, composing a lengthy CodeProject article when deadline is imminent, suddenly a network outage occurs disrupting all network connections. You would lose all the data that you have painstakingly created. With local storage, you can continue to work offline while the web application saves your work to the local storage intermittently using some client-side scripting like JavaScript.

A website can allow users to customize the theme and layout of the web pages and store these settings in the local storage. In this way, users can get their own personlized web pages on subsequent visit. We will see an example later.

By storing gaming status in the local storage, gamers can continue from where they left off by retrieving their last game status from the local storage.

[3]**Origin** is the combination of protocol, hostname, and port number. For example, "http://www.example.com/dir1/index.html" and "http://www.example.com/dir2/index.html" are of the same origin, while "http://example.com/dir1/index.html", "https://www.example.com/dir1/index.html" and "http://www.example.com/dir2/index.html:8080" are not.

[4]**DNS spoofing** is a computer hacking attack, in which malicious data is injected into a Domain Name System (DNS), a directory for mapping domain names to real IP addresses, fooling the DNS into returning an incorrect IP address, the result of which causes all traffic bound for a domain being diverted to a phoney doman which is often the attacker's computer.

Getting Your Hands Dirty

Sound great! With HTML Web Storage, users can now store and manipulate web data just like any other files in their computer. Naturally, this question arises - how to make it happen? Specifically, how to build one of these web storages? How to save, retrieve, update, and remove data from it? The answer lies with your hands. What better way to learn than to make them happen by getting your hands dirty.

Let's start with the session storage first. You will build a simple article subscription web application that will allow a user to add and modify the number of subscription in a single session using Session Storage.

Building Session Storage

On a web server, create a web folder called “webstorage”, inside which create an HTML page called “sessionstorage.html” with the following code:

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="utf-8">
<title>Session Storage</title>
<script>
// To add code
</script>
</head>
<body onload="init()">
<div style="text-align:center">
<h1>Welcome to Fake Library</h1>
<h2>Subscription Counter: </h2>
<label for="noOfArticleSubscribed">No of Article
Subscribed</label>
<input type="number" id="noOfArticleSubscribed" max="6" min="0"
value="0">
<br>
<button type="button" onclick="save()">Save</button>
<button type="button" onclick="remove()">Clear</button>
<button type="button" onclick="checkout()">Check out</button>
</div>
</body>
</html>
```

Launch this page on a browser as shown in Figure 24.

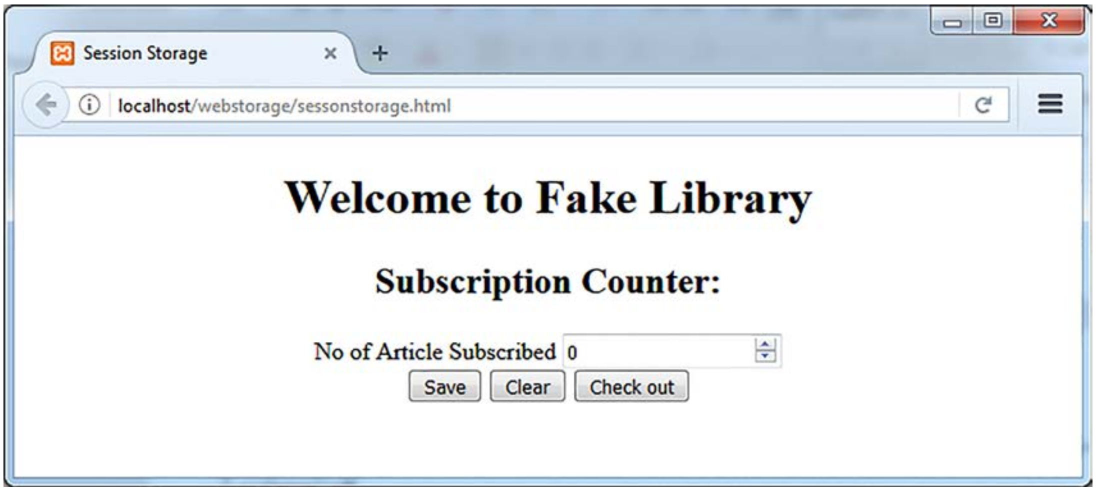
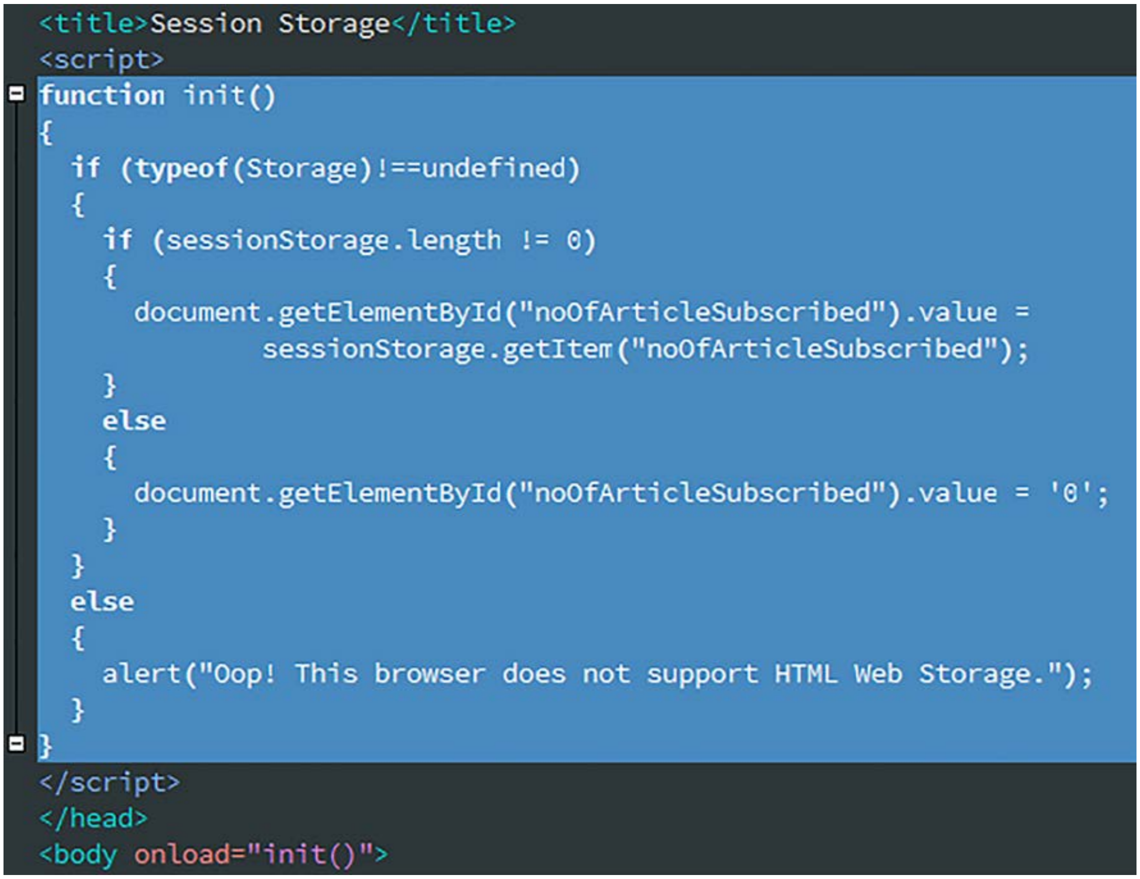


Figure 24 sessionstorage.html

From here on, you will add JavaScript code to implement and manipulate a Session Storage on a browser. The HTML Web Storage API provides a **sessionStorage** object for working with session storage. You will learn to use the various methods and properties of this sessionStorage object to perform **retrieval**, **saving**, **updating**, and **deletion** of data on the Session Storage.

Retrieval

Add an `init()` function to the "sessionstorage.html" as highlighted in Figure 25:



```
<title>Session Storage</title>
<script>
function init()
{
  if (typeof(Storage)!==undefined)
  {
    if (sessionStorage.length != 0)
    {
      document.getElementById("noOfArticleSubscribed").value =
              sessionStorage.getItem("noOfArticleSubscribed");
    }
    else
    {
      document.getElementById("noOfArticleSubscribed").value = '0';
    }
  }
  else
  {
    alert("Oop! This browser does not support HTML Web Storage.");
  }
}
</script>
</head>
<body onload="init()">
```

Figure 25 Adding `init()` Function

This `init()` function will be called when the "sessionstorage.html" page has finished loading through the **onload** event of the `<body>` tag as shown below:

```
<body onload="init()">
```

The `init()` function does the following tasks:.

- Check the existence of session storage for the current session by calling the **length** property of the sessionStorage object to find out the number of key-value pairs exist in the storage.

```
if (sessionStorage.length != 0)
{
  // sessionStorage exists, retrieve data from storage
}
else
{
  // sessionStorage does not exist, set default value
}
```

- If it exists, retrieve one of the previously stored item called "noOfArticleSubscribed" from the Session Storage by calling the `getItem()` method of the sessionStorage object and passing it the key name of "noOfArticleSubscribed" as parameter. The retrieved value is then assigned to the HTML element identified by the id of "noOfArticleSubscribed". Do not confuse the two "noOfArticleSubscribed""s, the former is the key name for a storage item in the session storage, the latter the id of an HTML element. You can give them different values if you wish.

```
document.getElementById("noOfArticleSubscribed").value =
sessionStorage.getItem("noOfArticleSubscribed");
```

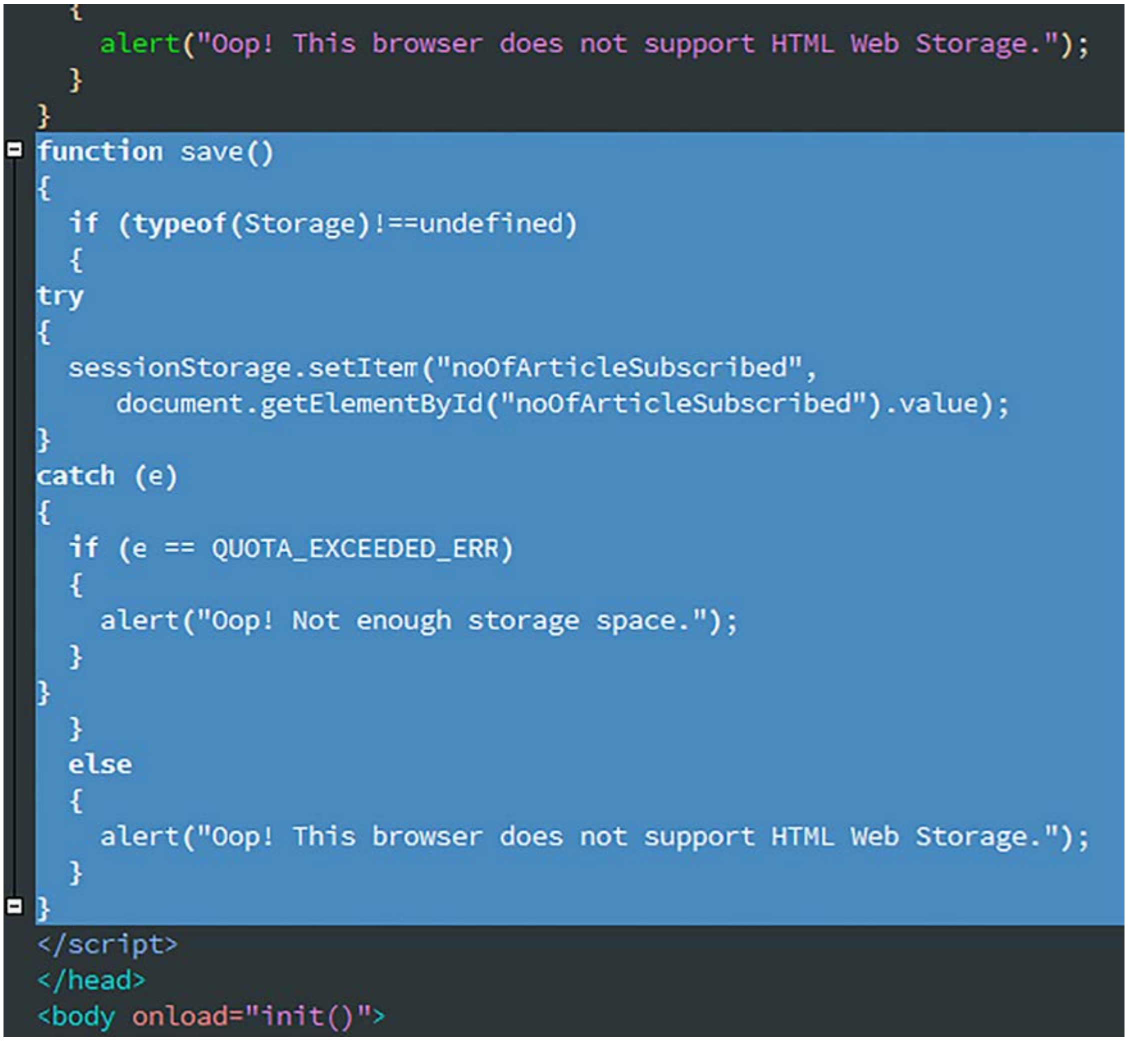
- That is all that you need to retrieve a stored item, in fact, any stored items from the session storage. But wait a second, what is this:

```
if(typeof(Storage)!==undefined)
```

I am glad that you asked. This statement will check whether your browser supports HTML Web Storage or not. It should be used to wrap any code that accesses Web Storage.

Saving / Updating

Continue to add a `save()` function to the “sessionstorage.html” as highlighted in Figure 26:



```
    {
      alert("Oop! This browser does not support HTML Web Storage.");
    }
}
function save()
{
    if (typeof(Storage)!==undefined)
    {
try
{
    sessionStorage.setItem("noOfArticleSubscribed",
       document.getElementById("noOfArticleSubscribed").value);
}
catch (e)
{
    if (e == QUOTA_EXCEEDED_ERR)
    {
      alert("Oop! Not enough storage space.");
    }
}
    }
    else
    {
      alert("Oop! This browser does not support HTML Web Storage.");
    }
}
</script>
</head>
<body onload="init()">
```

Figure 26 Adding `save()` Function

This `save()` function will be called through the **onclick** event of the "Save" button when this button is clicked as shown below:

```
<button type="button" onclick="save()">Save</button>
```

The `save()` function does the following tasks:.

- Save the value of the HTML element whose id is "noOfArticleSubscribed" as a key-value pair in the session storage using the `setItem()` method of the sessionStorage object. If this key does not exist, it will be created else its value will be over-written (updated).

```
sessionStorage.setItem("noOfArticleSubscribed",
document.getElementById("noOfArticleSubscribed").value);
```

- If it exists, retrieve one of the previously stored item called "noOfArticleSubscribed" from the Session Storage by calling the `getItem()` method of the sessionStorage object and passing it the key name of "noOfArticleSubscribed" as parameter. The retrieved value is then assigned to the HTML element identified by the id of "noOfArticleSubscribed". Do not confuse the two "noOfArticleSubscribed""s, the former is the key name for a storage item in the session storage, the latter the id of an HTML element. You can give them different values if you wish.

```
document.getElementById("noOfArticleSubscribed").value =
sessionStorage.getItem("noOfArticleSubscribed");
```

- Notice the try{} and catch{} duo. They are code to catch any run-time errors (exceptions) that may arise during execution of the code inside the try{} block. What could go wrong? You may ask. Remember the 5MB storage quota for Web Storage that I mentioned earlier. So, one possible scenario in this case or rather in any similar cases that involve "save" operation, the intended saved item could have exceed the quota and thus throwing an error. The code that handles any run-time errors is placed inside the catch{} block. We can "catch" the type of error by reading the error message, i.e. "e", and compare it to some pre-determined values to pre-empt any run-time errors dynamically. In this case, we are pre-empting the "QUOTA_EXCEEDED_ERR" error the meaning of which is self-explanatory. In fact, you should always make use of the try{} and catch{} blocks to handle any save or update operations to the Web Storage.

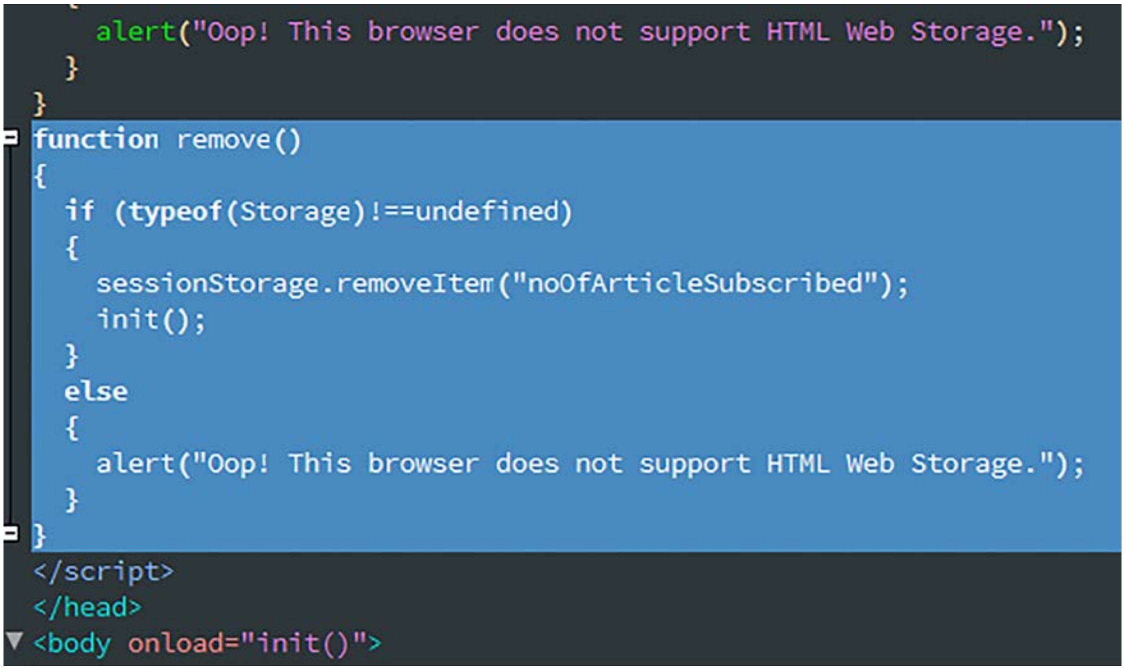
- That is all that you need to save any items to the Session Storage. Lastly, do not forget to wrap your code inside this:

```
if(typeof(Storage)!==undefined){ ... };
```

Make this checking for browser support of HTML Web Storage the de facto practice.

Deletion

You have done the code for retrieving from, saving, and updating to session storage, How do you remove items from the session storage? Continue to add a `remove()` function to the “sessionstorage.html” as highlighted in Figure 27:



```
    alert("Oop! This browser does not support HTML Web Storage.");
  }
}
function remove()
{
  if (typeof(Storage)!==undefined)
  {
    sessionStorage.removeItem("noOfArticleSubscribed");
    init();
  }
  else
  {
    alert("Oop! This browser does not support HTML Web Storage.");
  }
}
</script>
</head>
<body onload="init()">
```

Figure 27 Adding `remove()` Function

This `remove()` function will be called through the **onclick** event of the "Clear" button when this button is clicked as shown below:

```
<button type="button" onclick="remove()">Clear</button>
```

The `remove()` function does the following tasks:.

- Delete the stored item whose key is "noOfArticleSubscribed" from the session storage by calling the removeItem() method of the sessionStorage object and passing it the item key name of "noOfArticleSubscribed" as parameter.

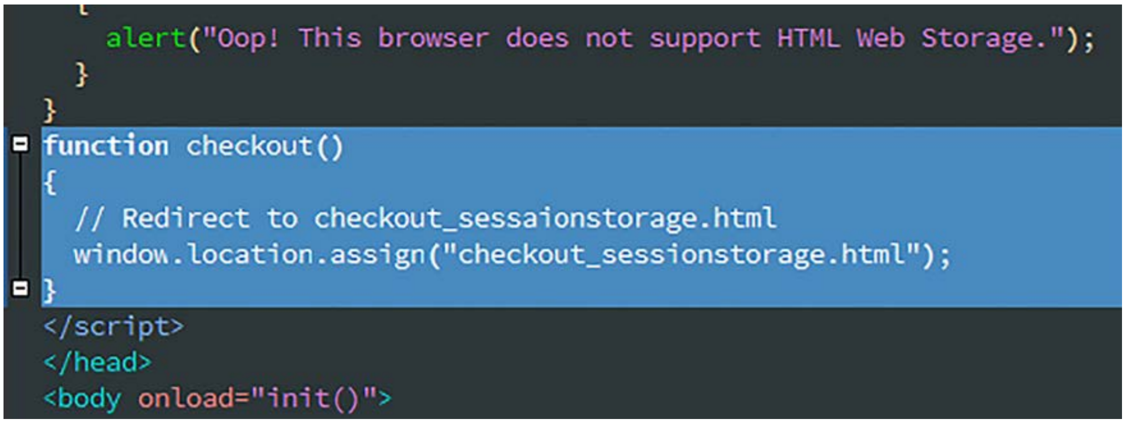
  ```
  sessionStorage.removeItem("noOfArticleSubscribed");
  ```

- After the removal, do not forget to call the `init()` function to reset the value of the HTML element that was previously showing the value of this deleted stored item.

Check Out

To check out, click the "Check out" button of the "sessionstorage.html" and you will be redirected to a check-out page that will simply retrieve the and display the value of the "noOfArticleSubscribed" key created and saved to the session storage at the "sessionstorage.html" page.

Start by adding a `checkout()` function to the "sessionstorage.html" as highlighted in Figure 28:



```
    {
      alert("Oop! This browser does not support HTML Web Storage.");
    }
}
function checkout()
{
  // Redirect to checkout_sessaionstorage.html
  window.location.assign("checkout_sessionstorage.html");
}
</script>
</head>
<body onload="init()">
```

Figure 28 Adding `checkout()` Function

This `checkout()` function will be called through the **onclick** event of the "Check out" button when this button is clicked as shown below:

```
<button type="button" onclick="checkout()">Check out</button>
```

The `checkout()` function will load a check-out page called “checkout_sessionstorage.html".

```
window.location.assign("checkout_sessionstorage.html");
```

Where is this "checkout_sessionstorage.html"? You will create it inside the "webstorage" folder with the code as shown in Listing 14.

Listing 14 Source Code of checkout_sessionstorage.html

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="utf-8">
<title>Session Storage</title>
<script>
function init(){
  if (typeof(Storage)!==undefined) // Browser supports HTML Web
Storage
  {
   if (sessionStorage.length != 0) // If sessionStorage is not
empty
   {
     document.getElementById("subscriptionStatus").innerHTML =
"You have subscribed to " +
sessionStorage.getItem("noOfArticleSubscribed") + " articles.
Thank you.";
   }
   else // If sessionStorage is empty
   {
     document.getElementById("subscriptionStatus").innerHTML =
"You have not subscribed to any article.";
   }
  }
  else
  {
   alert("Oop! This browser does not support HTML Web
Storage.");
  }
}
</script>
</head>
<body onload="init()">
<div style="text-align:center">
<h1>Welcome to Fake Library</h1>
<h2>Check out</h2>
<p id="subscriptionStatus"></p>
</body>
</html>
```

Upon completion of page load, the **onload** event of "checkout_sessionstorage.html" will call its `init()` function to retrieve the value of item key "noOfArticleSubscribed" from the session storage and display it in a message.

Code Completion

You have completed the coding for the exercise on Session Storage. The complete code for the "sessionstorage.html" page is shown in Listing 15.

Listing 15 Complete Source Code of sessionstorage.html

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="utf-8">
<title>Session Storage</title>
<script>
function init()
{
  if (typeof(Storage)!==undefined)
  {
   if (sessionStorage.length != 0)
   {
     document.getElementById("noOfArticleSubscribed").value =
             sessionStorage.getItem("noOfArticleSubscribed");
   }
```

```
    else
    {
      document.getElementById("noOfArticleSubscribed").value =
'0';
    }
  }
  else
  {
    alert("Oop! This browser does not support HTML Web
Storage.");
  }
}
function save()
{
  if (typeof(Storage)!==undefined)
  {
try
{
  sessionStorage.setItem("noOfArticleSubscribed",
     document.getElementById("noOfArticleSubscribed").value);
}
catch (e)
{
  if (e == QUOTA_EXCEEDED_ERR)
  {
    alert("Oop! Not enough storage space.");
  }
}
  }
  else
  {
    alert("Oop! This browser does not support HTML Web
Storage.");
  }
}
function remove()
{
  if (typeof(Storage)!==undefined)
  {
    sessionStorage.removeItem("noOfArticleSubscribed");
    init();
  }
  else
  {
    alert("Oop! This browser does not support HTML Web
Storage.");
  }
}
function checkout()
{
  // Redirect to checkout_sessaionstorage.html
  window.location.assign("checkout_sessionstorage.html");
}
</script>
</head>
<body onload="init()">
<div style="text-align:center">
<h1>Welcome to Fake Library</h1>
<h2>Subscription Counter: </h2>
<label for="noOfArticleSubscribed">No of Article
Subscribed</label>
<input type="number" id="noOfArticleSubscribed" max="6" min="0"
value="0">
<br>
<button type="button" onclick="save()">Save</button>
<button type="button" onclick="remove()">Clear</button>
<button type="button" onclick="checkout()">Check out</button>
</div>
</body>
</html>
```

However, the exercise is not complete without testing it out.

Testing 1,2,3...

You are ready to test the fruits of your labour. I am using Firefox for subsequent illustrations. Launch the “sessionstorage.html” and activate the browser’s developer tools to view the Session Storage node as shown in Figure 29.

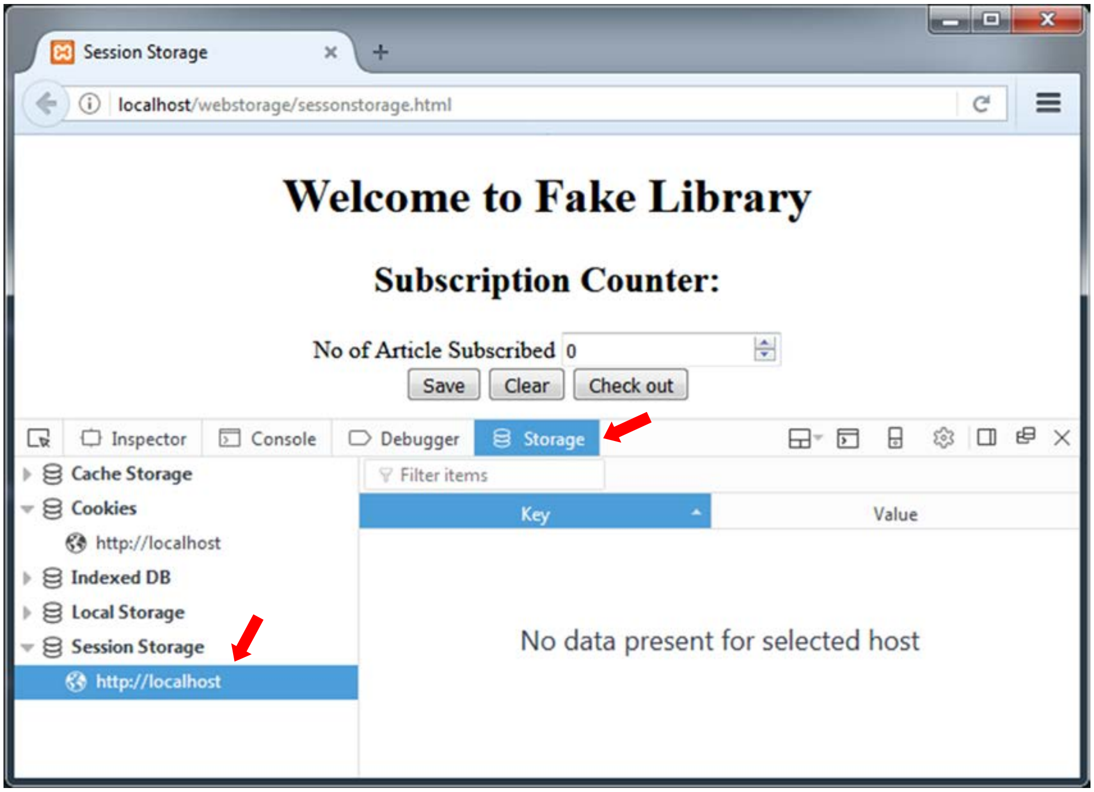
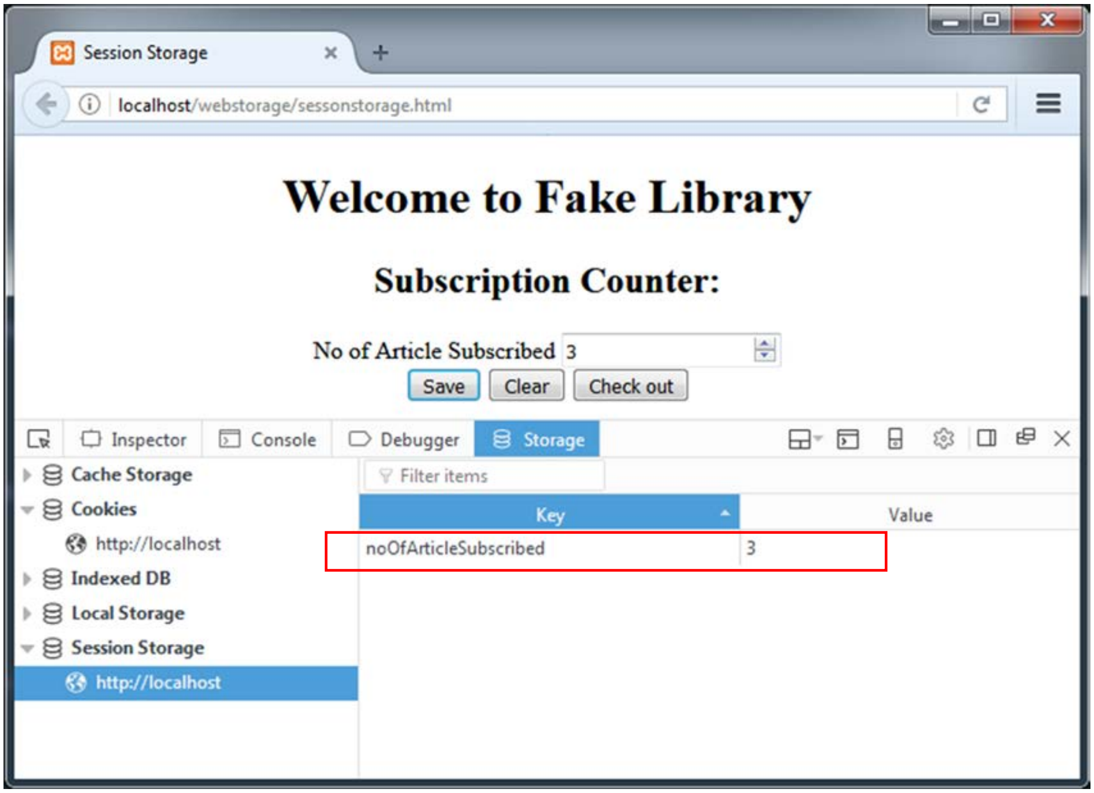


Figure 29 Newly Launched sessionstorage.html

The http://localhost origin under the "Session Storage" node shows that your newly launched "sessionstorage.html" page does not have any session storage data. Try changing the value in the "No of Articles Subscribed" text box to say "3" and click the "Save" button. Do you see what I see in Figure 30?



Welcome to Fake Library

Subscription Counter:

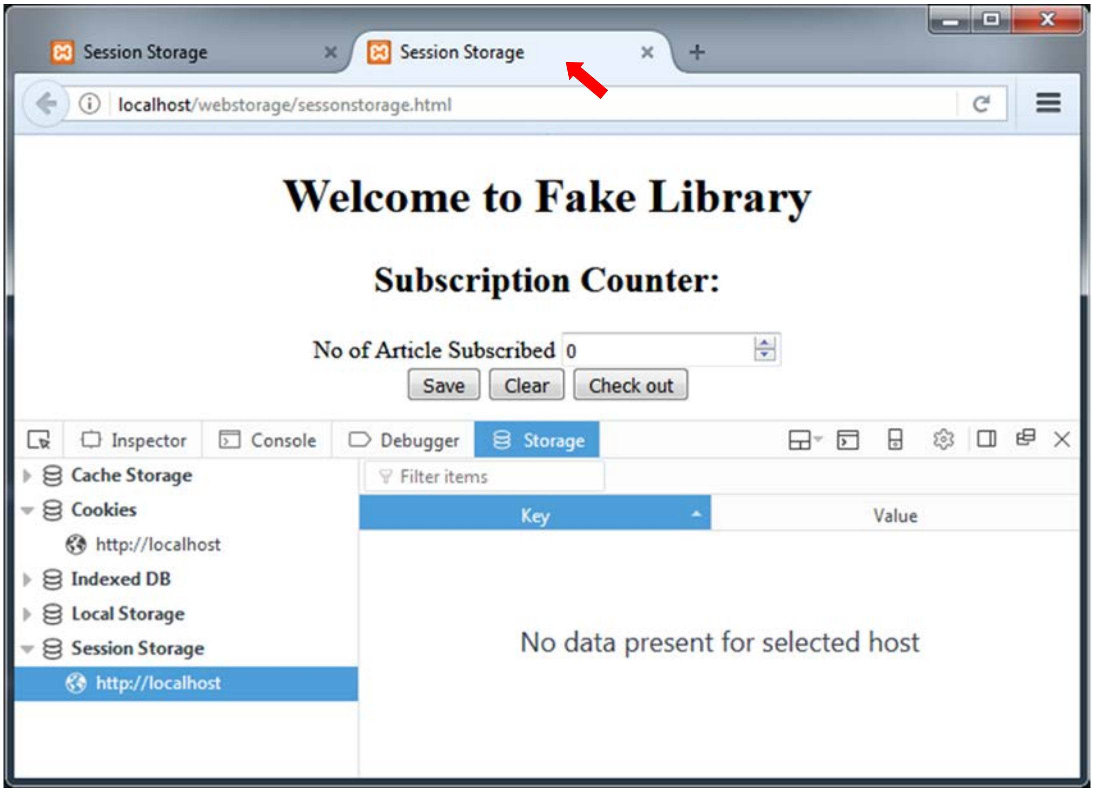
No of Article Subscribed 3

Key	Value
noOfArticleSubscribed	3

Figure 30 Input Value Saved to Session Storage

An item has just been created and stored in the session storage. This item bears the key called "noOfArticleSubscribed" and has a value of "3". That is the meaning of a key-value pair. Items in session storage are identified by their respective keys so that you can add multiple items with different keys. If you add an item whose key already existed, it will be updated with the new value otherwise it will be created.

Launch the “sessionstorage.html” on a new tab as shown in Figure 31:



Welcome to Fake Library

Subscription Counter:

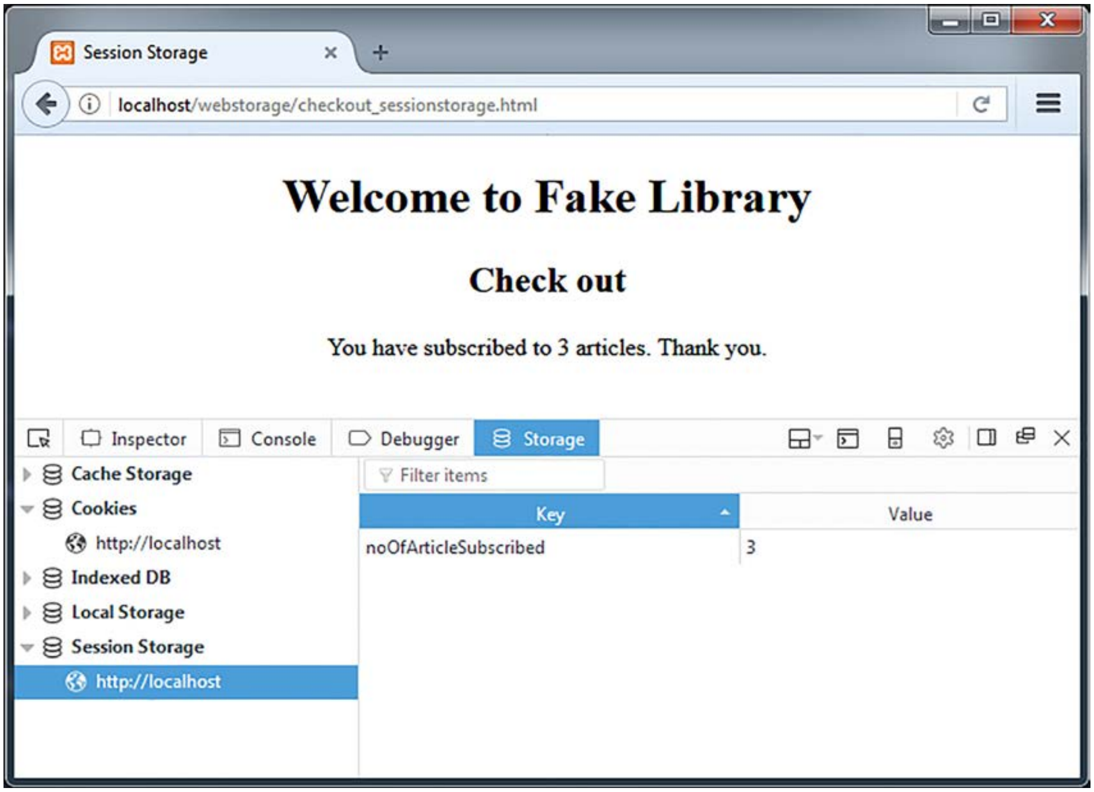
No of Article Subscribed 0

No data present for selected host

Figure 31 sessionstorage.html on New Tab

In Figure 31, you have started a new session with the website by launching the "sessionstorage.html" on a new tab. So, you get the same screen as shown Figure 29 — the Session Storage node is empty and the "No of Article Subscribed" textbox contains the default value of "0". If you switch back to the first tab, you will see that the "noOfArticleSubscribed" key of the Session Storage node remains intact as "3" as shown in Figure 30. Try changing the values in the respective "No of Article Subscribed" text boxes in the two tabs differently and save them, the two session storages will each have their own copies of "noOfArticleSubscribed" with different values. Closing a tab signals the end of a session and all session storage items belonging to this session will also be removed. So, the message is clear: **Session Storage only lives for a single session**. In other words, **Session Storage cannot be shared across different sessions**.

Let's close the second tab and return to the screen in Figure 30, then click the "Check out" button which will redirect you to the "checkout_sessionstorage.html" page as shown in Figure 32:



Welcome to Fake Library

Check out

You have subscribed to 3 articles. Thank you.

Key	Value
noOfArticleSubscribed	3

Figure 32 checkout_sessionstorage.html

You have concluded the exercise for Session Storage with this final message: **Session Storage is shared across different pages of the same session and of the same origin**.

Building Local Storage

Besides the sessionStorage object, HTML Web Storage API also provides a **localStorage** object for working with local storage.

Coding

The good news is: localStorage object shares the same methods and properties with the sessionStorage object such as **getItem()**, **setItem()**, **removeItem()**, and **length**. Therefore, if you are familiar with the session storage, coding for local storage will be much easier - it is mostly a matter of, believe it or not, "copy and paste".

First, open the “sessionstorage.html” file in a text editor, save it as "localstorage.html", then inside the “localstorage.html” file,

- Find and replace all the sessionStorage object with localStorage object. For example,

```
// if (~~sessionStorage~~.length != 0);
if (localStorage.length != 0)
```

- Change the code in the checkout() function to:

```
window.location.assign("checkout/checkout_localstorage.html"
);
```

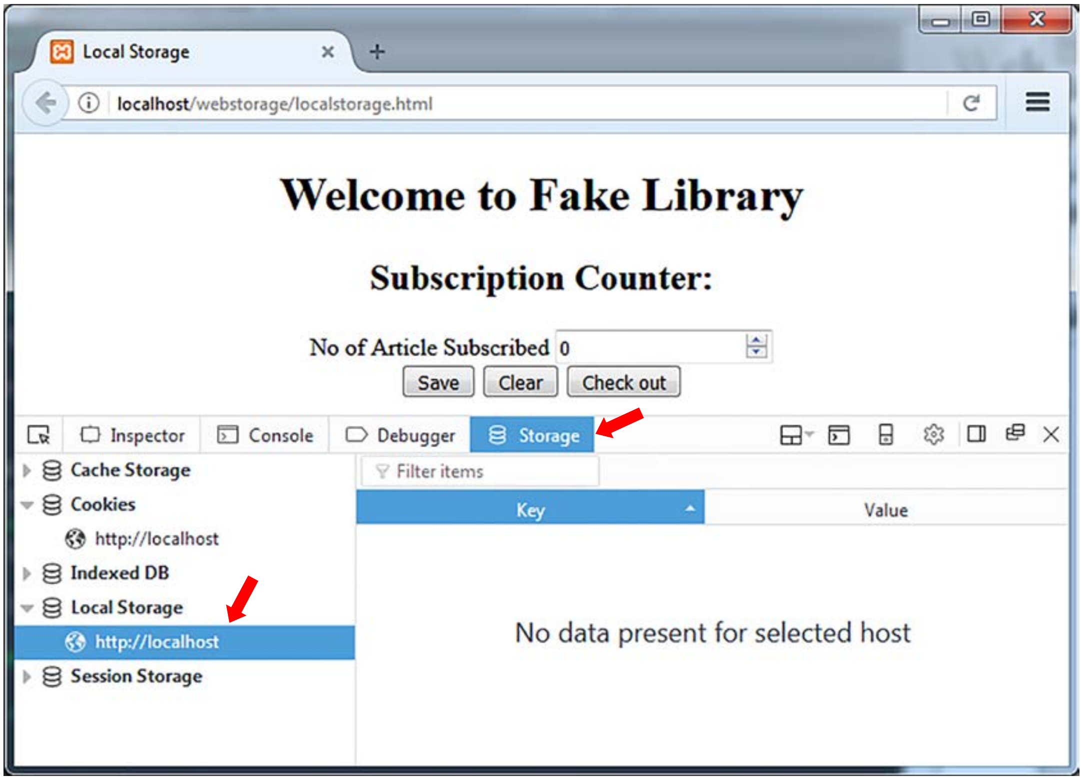
Similarly, open the “checkout_sessionstorage.html” file in a text editor, save it as " checkout_localstorage.html", then inside the “checkout_localstorage.html” file, find and replace all the sessionStorage object with localStorage object.

Lastly, remember to change the titles of both HTML pages to “Local Storage”.

That’s all. You are done with the coding for local storage!

Testing 1,2,3…

You are ready to test the fruits of your labour. I am using Firefox for subsequent illustrations. Launch the "localstorage.html" and activate the browser's developer tools to view the Local Storage node as shown in Figure 33.



Welcome to Fake Library

Subscription Counter:

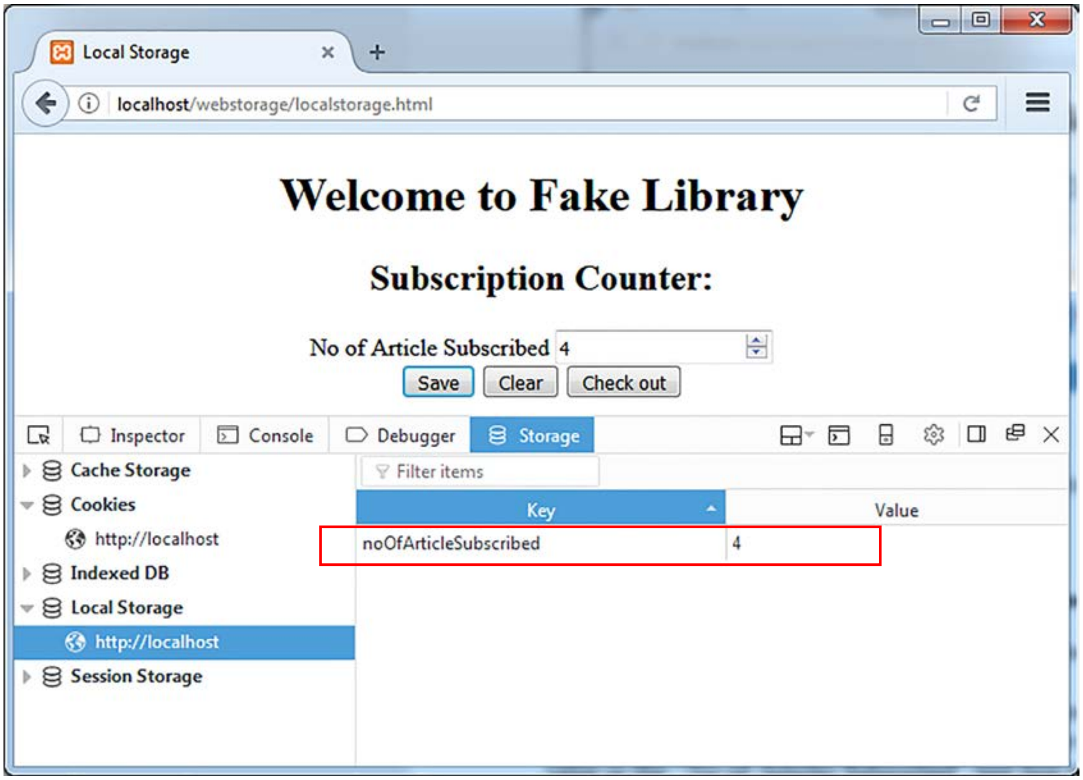
No of Article Subscribed 0

Save Clear Check out

No data present for selected host

Figure 33 Newly Launched localstorage.html

The http://localhost origin under the "Local Storage" node shows that your newly launched "localstorage.html" page does not have any session storage data. Try changing the value in the "No of Articles Subscribed" text box to say "4" and click the "Save" button. Do you see what I see in Figure 34?

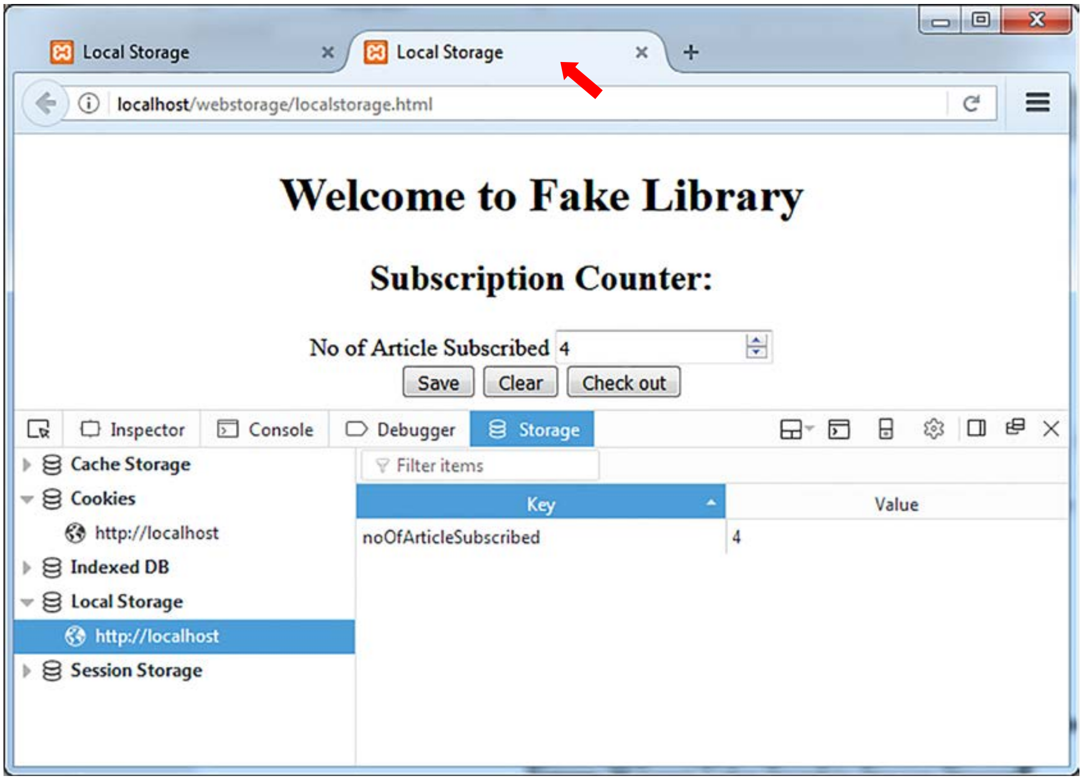


Key	Value
noOfArticleSubscribed	4

Figure 34 Input Value Saved to Local Storage

An item has just been created and stored in the local storage. This item bears the key called "noOfArticleSubscribed" and has a value of "4".

Launch the “localstorage.html” on a new tab as shown in Figure 35:



Welcome to Fake Library

Subscription Counter:

No of Article Subscribed 4

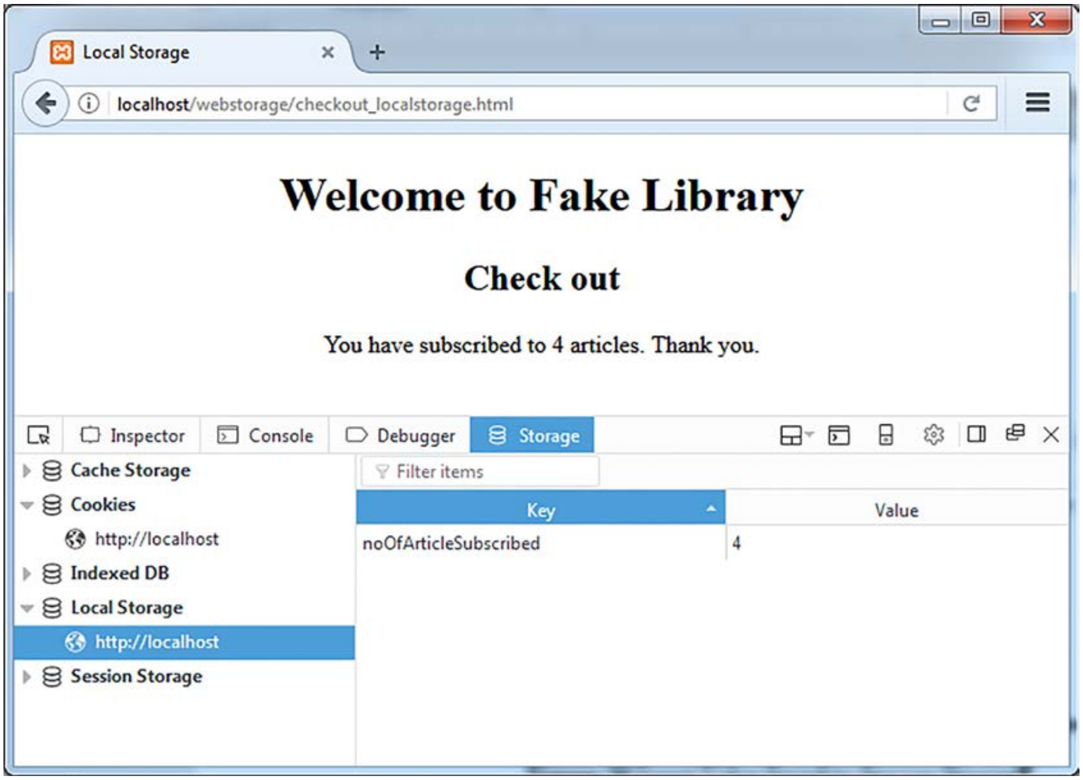
Save Clear Check out

Key	Value
noOfArticleSubscribed	4

Figure 35 localstorage.html on New Tab

In Figure 35, you have started a new session with the website by launching the "localstorage.html" on a new tab. Unlike that screen in Figure 31, this time, the "localstorage.html" page on the new tab can retrieve the display the "noOfArticleSubscribed" value saved on the first tab from the local storage. Closing a tab signals the end of a session but all local storage items created in any session will remain. So, the message is clear: **Local Storage persists and is shared across different sessions of the same origin**.

Let's close the second tab and return to the screen in Figure 34, then click the "Check out" button which will redirect you to the "checkout_localstorage.html" page as shown in Figure 36:


Local Storage

localhost/webstorage/checkout_localstorage.html

Welcome to Fake Library

Check out

You have subscribed to 4 articles. Thank you.

Key	Value
noOfArticleSubscribed	4

Figure 36 checkout_localstorage.html

You have concluded the exercise for Local Storage with this final message: **Local Storage is shared across different pages of the same origin regardless of sessions**.

Web Storage's Property, Methods, and Events

You have experimented with sessionStorage and localStorage objects of the Web Storage API. Both sessionStorage and localStorage objects possess similar property, methods, and events. Let's go through them now using localStorage in the examples that follow. For sessionStorage, simply replace localStorage with sessionStorage in the examples and that's all.

Property

A web storage object possesses one property — **length**.

length

The `length` property returns the number of items (key-value pairs) present in a web storage object. For example:

```
alert(localStorage.length);
```

Methods

A web storage object possesses five methods — **setItem()**, **getItem()**, **removeItem()**, **clear()**, and **key()**.

setItem()

The `setItem()` method attempts to insert a new key-value pair or update the value of an existing key in a storage object.

The `setItem()` method takes 2 parameters: a key and its associated new value. The method will first check if the given key already exists in the storage object. If it does not, then a new key-value pair will be added to the storage object. If the given key already exists in the storage object, and its current value is not equal to the new value, then its value will be updated to the new value. If its current value is equal to the new value, then the method will do nothing.

In the code below, `noOfArticleSubscribed` is the key while its value is `document.getElementById("noOfArticleSubscribed").value`. Always use the `setItem()` method with the `try{}` and `catch{}` duo to detect and handle the `QUOTA_EXCEEDED_ERR` error.

```
try
{
  localStorage.setItem("noOfArticleSubscribed",
document.getElementById("noOfArticleSubscribed").value);
}
```

```
catch (e)
{
  if(e == QUOTA_EXCEEDED_ERR)
  {
    alert("Oop! Not enough storage space.");
  }
}
```

getItem()

The `getItem()` method returns the current value associated with the given key in a storage object. It will return null If the given key does not exist in the storage object.

The `getItem()` method takes one parameter, i.e. a key.

```
document.getElementById("noOfArticleSubscribed").value =
localStorage.getItem("noOfArticleSubscribed");
```

removeItem()

The `removeItem()` method removes an item from a storage object. If the key does not exist, this method will do nothing.

The `removeItem()` method takes one parameter, i.e. a key.

```
localStorage.removeItem("noOfArticleSubscribed");
```

clear()

The `clear()` method removes **all** items from a storage object. This method takes no parameters.

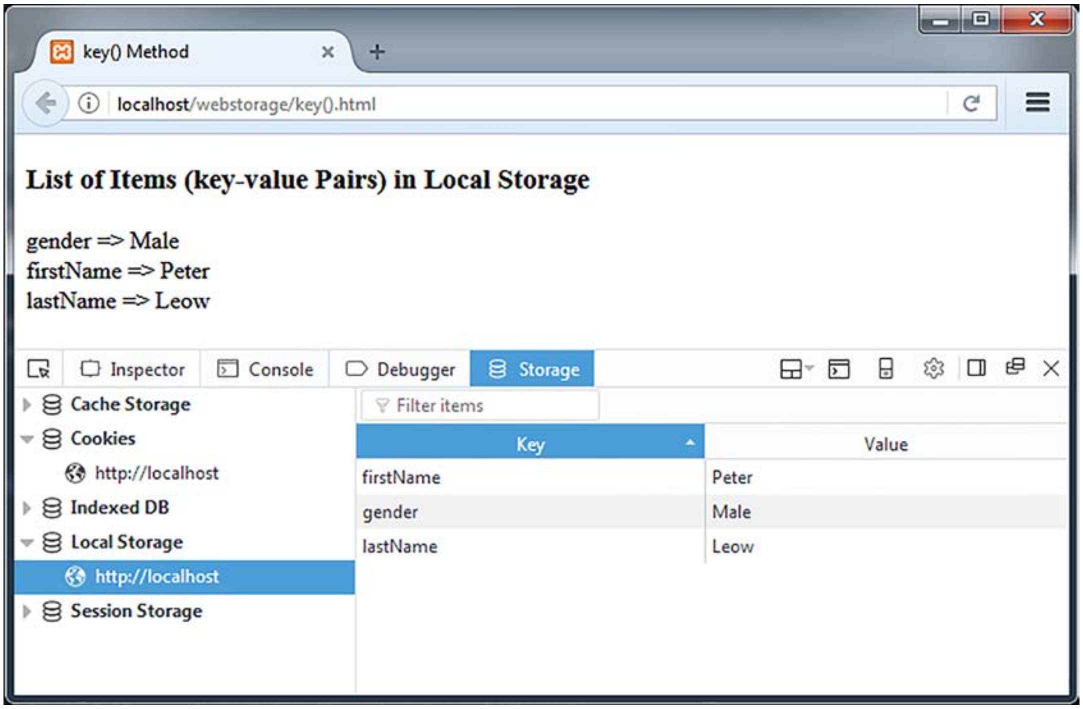
```
localStorage.clear();
```

key()

The `key()` method takes an integer number as parameter and returns the name of the key whose index position in the item (key-value pair) list of the storage is equal to the parameter.

Try out the following code using the `key()` method and see its output on browser as shown in Figure 37.

```
<!DOCTYPE HTML>
<html>
<head>
<title>key() Method</title>
</head>
<body>
<h3>List of Items (key-value Pairs) in Local Storage</h3>
<script>
if (typeof(Storage)!==undefined)
{
  localStorage.setItem("firstName", "Peter");
  localStorage.setItem("lastName", "Leow");
  localStorage.setItem("gender", "Male");
  for (i = 0; i < localStorage.length; i++)
  {
   var key = localStorage.key(i);
   var value = localStorage.getItem(key);
   document.write(key + " => " + value + "<br>");
  }
}
else
{
  alert("Oop! This browser does not support HTML5 Web
Storage.");
}
</script>
</body>
</html>
```



List of Items (key-value Pairs) in Local Storage

gender => Male
firstName => Peter
lastName => Leow

Key	Value
firstName	Peter
gender	Male
lastName	Leow

Figure 37 Demo on `key()` Method

Events

When the `setItem()`, `removeItem()`, or `clear()` method is called and actually changes the state of a storage object, a **storage** event is fired on the window object. Through this storage event, we can track changes in the storage area programmatically. We can add a listener to the event and handle the storage changes in an event handler. For example,

```
function onStorageEvent(storageEvent)
{
  document.getElementById("yourMessage").value =
localStorage.getItem("message");
}
window.addEventListener('storage', onStorageEvent, false);
```

In the example, the storage event passes a **StorageEvent** object called `storageEvent` to the event handler of `onStorageEvent()`. This **StorageEvent** object possesses the following properties, listed in Table 7, that we can use to find out the details of changes in the storage area programmatically:

Table 7: Storage Event Properties

Property	**Description**
key	The key property returns the name of the key that is added, updated, or removed.
oldValue	The oldValue property returns the old value.
newValue	The newValue property returns the newly set value.
url	The url property returns the URL of the page from where the event originated.
storageArea	The storageArea property returns the storage object whose key is changed, i.e. sessionStorage or localStorage.

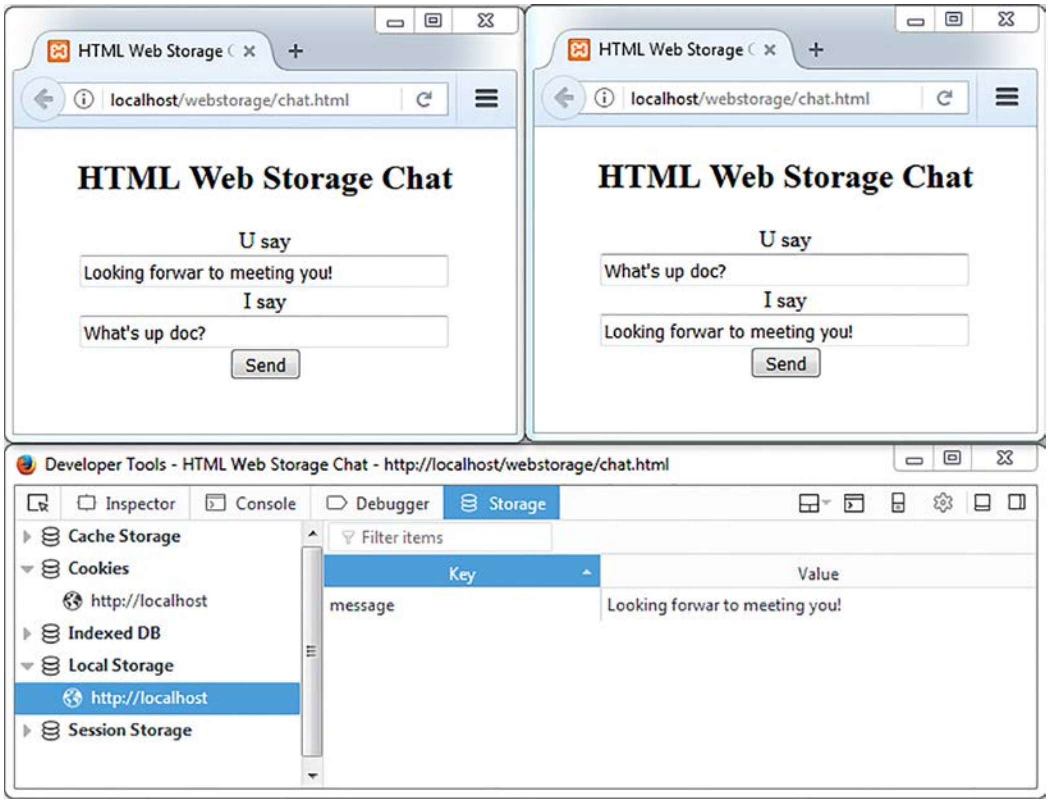
However, **the event gets fired only on other windows and not on the window where the event is triggered**. Let create a simple chat program to illustrate this.

Create an HTML file called “chat.html” with the code in Listing :

Listing 16 Source Code of chat.html

```
<!DOCTYPE HTML>
<html>
<head>
<title>HTML Web Storage Chat</title>
<script>
function get() {
  document.getElementById("yourMessage").value =
localStorage.getItem("message");
}
function send(){
  localStorage.setItem("message",
document.getElementById("myMessage").value);
}
function onStorageEvent(storageEvent){
  document.getElementById("yourMessage").value =
localStorage.getItem("message");
}
window.addEventListener('storage', onStorageEvent, false);
</script>
</head>
<body>
<div style="text-align:center">
<h2>HTML Web Storage Chat</h2>
<label for="yourMessage">U say</label>
<br>
<input type="text" id="yourMessage" readonly size="40">
<br>
<label for="myMessage">I say</label>
<br>
<input type="text" id="myMessage" autofocus size="40">
<br>
<button type="button" onclick="send()">Send</button>
</div>
</body>
</html>
```

Launch the “chat.html” page on two separate windows of the same browser, place them side-by-side, start chatting between the two browser windows, and observe the chat message being saved and updated constantly in the Local Storage node of the developer tools as shown in Figure 38.


HTML Web Storage Chat

U say

Looking forwar to meeting you!

I say

What's up doc?

HTML Web Storage Chat

U say

What's up doc?

I say

Looking forwar to meeting you!

Developer Tools - HTML Web Storage Chat - http://localhost/webstorage/chat.html

Key	Value
message	Looking forwar to meeting you!

Figure 38 chat.html

The logic of the code works as follows:

- When the send button in the first browser window is clicked, the message from the "I say" textbox will be saved to the localStorage object with the key name "message". This causes a change in the state of the localStorage object.

- This change triggers a storage event and is captured by the second browser window which then calls the `onStorageEvent()` function.

- The `onStorageEvent()` function retrieves the value of the key named "message" from the localStorage and assigns it to the "yourMessage" textbox labeled as "U say".

- The process is reversed when the message is sent from the second window.

Have fun!

Best Practices

You have explored the essence of HTML Web Storage API. Let's round up some best practices on how to manage it correctly.

- Always check whether your browser supports HTML Web Storage or not before using it.

```
if(typeof(Storage)!==undefined)
{
  // Storage related code
}
else
{
  alert("Oop! This browser does not support HTML Web
Storage.");
}
```

- Always make use of try() and catch() blocks to catch quota exceed error when adding new or updating existing items using `setItem()` method.

```
try
{
  sessionStorage.setItem("name of key", "some value");
}
catch (e)
{
  if (e == QUOTA_EXCEEDED_ERR)
  {
   alert("Oop! Not enough storage space.");
  }
}
```

- The `clear()` method will wipe up the whole storage object, so use it with extreme care. It is more prudent to use `removeItem()` instead.

- When a storage item is updated or removed, it is a necessarily good practice to update the value or state of those UI elements that reference this item.

- Take note that **data is stored as strings (Texts) in Web Storage**. If you are storing something other than a string, you will have to consciously convert it to their intended data type when you retrieve it. For example, the age of numeric 8 will be automatically stored as letter "8" in the storage:

  ```
  var age = 8;
  localStorage["age"] = age;
  ```

  But when you retrieve this item from the storage, you need to convert it back to an integer, using the `parseInt()` function of JavaScript:

  ```
  var age = parseInt(localStorage["age"]);
  ```

- Last but not least, always use SSL (Secure Sockets Layer) to transmit sensitive information such as credit card numbers, social security numbers, and login credentials across network. Data sent in plain texts between browsers and websites is vulnerable to eavesdropping and DNS spoofing. SSL can be used to prevent this from happening by establishing an encrypted link between a browser and a website.

Tips

Here are some tips on how to code the web storage more efficiently.

- Instead of using `getItem()` method of the storage object to read storage item, you can do the shortcut way like this:

  ```
  document.getElementById("noOfArticleSubscribed").value =
  sessionStorage["noOfArticleSubscribed"];
  ```

- Likewise, there are two shortcuts to save or update storage item instead of using `setItem()` method:

```
sessionStorage["noOfArticleSubscribed"] =
document.getElementById("noOfArticleSubscribed").value;
```

or

```
sessionStorage.noOfArticleSubscribed =
document.getElementById("noOfArticleSubscribed").value;
```

- You have seen an example of iterating through the item lists of a storage object using the `key()` method. A shorter way is like this:

```
for (var key in localStorage) {
  document.write(key+" => "+localStorage[key]+"<br>");
}
```

- Last but not least, why not create a template that is pre-written with the must-have code components mentioned in the **Best Practices** section? Such as this template written for sessionStorage object as shown in Listing 17:

Listing 17 Template for sessionStorage Object

```
<!DOCTYPE HTML>
<html>
<head>
<title>Session Storage Template</title>
<script>
function init(){
  // Check whether browser supports HTML Storage or not
  if (typeof(Storage)!==undefined) {
   if (sessionStorage.length != 0) { // If storage object
is not empty
     document.getElementById("replace with id of an HTML
element").value  = sessionStorage.getItem("replace with a
key name");
   } else { // If storage object is empty, set default
value
```

```
     document.getElementById("replace with id of an HTML
element").value = '1';
   }
  } else {
     alert("Oop! This browser does not support HTML Web
Storage.");
  }
}
function saveSettings() {
  // Check whether browser supports HTML5 Storage or not
  if (typeof(Storage)!==undefined) {
   try {
     sessionStorage.setItem("replace with a key name",
"replace with a value");
   } catch (e) {
     if (e == QUOTA_EXCEEDED_ERR) {
        alert("Oop! Not enough storage space.");
     }
   }
  } else {
   alert("Oop! This browser does not support HTML Web
Storage.");
  }
}
function removeSettings() {
  // Check whether browser supports HTML5 Storage or not
  if (typeof(Storage)!==undefined) {
   // Remove one item from the storage object
   sessionStorage.removeItem("replace with a key name");
   // Call init to reset the HTML elements
   init();
  } else {
   alert("Oop! This browser does not support HTML Web
Storage.");
  }
}
</script>
</head>
<body onload="init()">
</body>
</html>
```

To use the template, simply replace the "replace with id of an HTML element" and "replace with a key name" placeholders with the actual id and key respectively. You can easily create a similar template for localStorage object by replacing the "sessionStorage" in this template with "localStorage".

9 SUMMARY

Glad to see you at the finishing line. You have explored a total of six HTML APIs with plenty of hands-on exercises, including:

- Geolocation for finding and navigating your way on Earth.
- Drag and Drop for dragging and dropping UI elements on HTML page.
- Server-Sent Events for enabling one-way streaming of data from servers.
- Web Sockets for establishing full-duplex bi-directional communication between clients and servers.
- Web Workers for spawning JavaScript on background threads.
- Web Storage for inserting, updating, retrieving, and deletion of data on browsers.

However, they are just the tip of the iceberg. With each passing year, more and more new APIs are expected to be drafted into the HTML API family. These APIs have given unprecedented power and autonomy to our most frequently used piece of software - the web browsers. It is not unimaginable that the web browsers may one day prevails over the traditional desktops and mobile devices. As to how soon this will happen, we will have to wait and see. Till then...

.

ABOUT THE AUTHOR

Peter Leow is a software engineer and system analyst, he has many years of software development and teaching experience in open source as well as proprietary technologies. Besides software development, artificial intelligence is another field that he has a great interest in. An avid pursuer of knowledge and advocate for lifelong learning, he has published many well-received articles at CodeProject.com and contributed solutions to its coding forum and discussion. For his efforts, he has been awarded Code Project MVP (Most Valuable Professional).

Learn more about Peter Leow's reputation at https://www.codeproject.com/Members/peterleow

Check out Peter Leow's other publications at https://www.amazon.com/author/peterleow

Read the many articles by Peter Leow at https://www.peterleowblog.com

Last but not least, follow me on twitter https://twitter.com/peterleowblog